From:

Software Requirements, Second Edition
by Karl E. Wiegers
Microsoft Press © 2003 (516 pages) Citation
ISBN:9780735618794

This resource shows you how to define and get more out of software requirements with dozens of "best practices" and tips on getting better design input from your customers and then using these requirements to generate a variety of design documents.

Designing Requirements

Why Write Requirements?

Taking Your Requirements Pulse

For a quick check of the current requirements engineering practices in your organization, ask yourself how many of the following conditions apply to your most recent project. If you check more than three or four boxes, this book is for you.

- The project's vision and scope are never clearly defined.
- Customers are too busy to spend time working with analysts or developers on the requirements.
- User surrogates, such as product managers, development managers, user managers, or marketers, claim to speak for the users, but they don't accurately represent user needs.
- Requirements exist in the heads of "the experts" in your organization and are never written down.
- Customers claim that all requirements are critical, so they don't prioritize them.
- Developers encounter ambiguities and missing information when coding, so they have to guess.
- Communications between developers and customers focus on user interface displays and not on what the users need to do with the software.
- Your customers sign off on the requirements and then change them continuously.

- The project scope increases when you accept requirements changes, but the schedule slips because no additional resources are provided and no functionality is removed.
- Requested requirements changes get lost, and you and your customers don't know the status of all change requests.
- Customers request certain functionality and developers build it, but no one ever uses it.
- The specification is satisfied, but the customer is not.

Clearly, there's no universal definition of what a requirement is. To facilitate communication, we need to agree on a consistent set of adjectives to modify the overloaded term *requirement*, and we need to appreciate the value of recording these requirements in a shareable form.

Trap

Don't assume that all your project stakeholders share a common notion of what requirements are. Establish definitions up front so that you're all talking about the same things.

Levels of Requirements

This section presents definitions that I will use for some terms commonly encountered in the requirements engineering domain. Software requirements include three distinct levels—business requirements, user requirements, and functional requirements. In addition, every system has an assortment of nonfunctional requirements. The model in Figure 1-1 illustrates a way to think about these diverse types of requirements. As with all models, it is not all-inclusive, but it provides a helpful organizing scheme. The ovals represent types of requirements information and the rectangles indicate containers (documents, diagrams, or databases) in which to store that information.

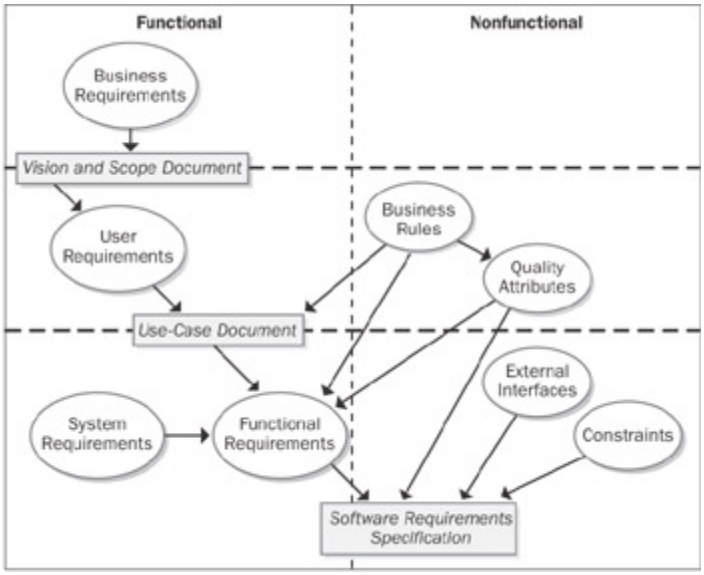

Figure 1-1: Relationship of several types of requirements information.

More Info Chapter 7, "Hearing the Voice of the Customer," contains many examples of these different types of requirements.

Business requirements represent high-level objectives of the organization or customer who requests the system. Business requirements typically come from the funding sponsor for a project, the acquiring customer, the manager of the actual users, the marketing department, or a product visionary. Business requirements describe why the organization is implementing the system—the objectives the organization hopes to achieve. I like to record the business requirements in a *vision and scope document*, sometimes called a *project charter* or a *market requirements document*. Creating such a document is the subject of Chapter 5, "Establishing the Product Vision and Project Scope." Defining the project scope is the first step in controlling the common problem of scope creep.

User requirements describe user goals or tasks that the users must be able to perform with the product. Valuable ways to represent user requirements include use cases, scenario descriptions, and event-response tables. User requirements therefore describe what the user will be able to do with the system. An example of a use case is "Make a Reservation" using an airline, a rental car, or a hotel Web site.

Functional requirements specify the software functionality that the developers must build into the product to enable users to accomplish their tasks, thereby satisfying the business requirements. Sometimes called *behavioral requirements*, these are the traditional "shall" statements: "The system shall e-mail a reservation confirmation to the user." As Chapter 10 ("Documenting the Requirements") illustrates, functional requirements describe what the developer needs to implement.

The term *system requirements* describes the top-level requirements for a product that contains multiple subsystems—that is, a *system* (IEEE 1998c). A system can be all software or it can include both software and hardware subsystems. People are a part of a system, too, so certain system functions might be allocated to human beings.

Business rules include corporate policies, government regulations, industry standards, accounting practices, and computational algorithms. As you'll see in Chapter 9, "Playing By the Rules," business rules are not themselves software requirements because they exist outside the boundaries of any specific software system. However, they often restrict who can perform certain use cases or they dictate that the system must contain functionality to comply with the pertinent rules. Sometimes business rules are the origin of specific quality attributes that are implemented in functionality. Therefore, you can trace the genesis of certain functional requirements back to a particular business rule.

Functional requirements are documented in a *software requirements specification* (SRS), which describes as fully as necessary the expected behavior of the software system. I'll refer to the SRS as a document, although it can be a database or spreadsheet that contains the requirements, information stored in a commercial requirements management tool—see Chapter 21, "Tools for Requirements Management"—or perhaps even a stack of index cards for a small project. The SRS is used in development, testing, quality assurance, project management, and related project functions.

In addition to the functional requirements, the SRS contains nonfunctional requirements. These include performance goals and descriptions of quality attributes. *Quality attributes* augment the description of the product's functionality by describing the product's characteristics in various dimensions that are important either to users or to developers. These characteristics include usability, portability, integrity, efficiency, and robustness. Other nonfunctional requirements describe external interfaces between the system and the outside world, and design and implementation constraints. *Constraints* impose restrictions on the choices available to the developer for design and construction of the product.

People often talk about product features. A *feature* is a set of logically related functional requirements that provides a capability to the user and enables the satisfaction of a business objective. In the commercial software arena, a feature is a group of requirements recognizable to a stakeholder that aids in making a purchase decision—a bullet item in the product description. A customer's list of desired product features is not equivalent to a description of the user's task-related needs. Web browser favorites or bookmarks, spell check, macro recording, automobile power windows, online update of tax code changes, telephone speed-dialing, and automatic virus signature updating are examples of product features. A feature can encompass multiple use cases, and each use case requires that multiple functional requirements be implemented to allow the user to perform the task.

To get a better grasp on some of the different kinds of requirements I've been discussing, consider a word processing program. A business requirement might read, "The product will allow users to correct spelling errors in a document efficiently." The product's box cover announces that a spell checker is included as a feature that satisfies this business requirement. Corresponding user requirements might include tasks—use cases—such as "Find spelling errors" and "Add word to global dictionary." The spell checker has many individual functional requirements, which deal with operations such as finding and highlighting a misspelled word, displaying a dialog box with suggested replacements, and globally replacing misspelled words with corrected words. The quality attribute called *usability* would specify just what is meant by the word "efficiently" in the business requirement.

Managers or marketing define the business requirements for software that will help their company operate more efficiently (for information systems) or compete successfully in the marketplace (for commercial products). All user requirements must align with the business requirements. The user requirements permit the analyst to derive the bits of functionality that will let the users perform their tasks with the product. Developers use the functional and nonfunctional requirements to design solutions that implement the necessary functionality and achieve the specified quality and performance objectives, within the limits that the constraints impose.

Although the model in Figure 1-1 shows a top-down flow of requirements, you should expect cycles and iteration between the business, user, and functional requirements. Whenever someone proposes a new feature, use case, or functional requirement, the analyst must ask, "Is this in scope?" If the answer is "yes," the requirement belongs in the specification. If the answer is "no," it does not. If the answer is "no, but it ought to be," the business requirements owner or the funding sponsor must decide whether to increase the project scope to accommodate the new requirement. This is a business decision that has implications for the project's schedule and budget.

What Requirements Are Not

Requirements specifications do not include design or implementation details (other than known constraints), project planning information, or testing information (Leffingwell and Widrig 2000). Separate such items from the requirements so that the requirements activities can focus on understanding what the team intends to build. Projects typically have other kinds of requirements, including development environment requirements, schedule or budget limitations, the need for a tutorial to help new users get up to speed, or requirements for releasing a product and moving it into the support environment. These are *project* requirements but not *product* requirements; they don't fall within the scope of this book.

Guidelines for Writing Requirements

There is no formulaic way to write excellent requirements; the best teacher is experience. The problems you have encountered in the past will teach you much. Excellent requirements documents follow effective technical-writing style guidelines and employ user terminology rather than computer jargon. Kovitz (1999) presents many recommendations and examples for writing good requirements. Keep the following suggestions in mind:

- Write complete sentences that have proper grammar, spelling, and punctuation. Keep sentences and paragraphs short and direct.
- Use the active voice. (For example, "The system shall do something," not "Something shall happen.")
- Use terms consistently and as defined in the glossary. Watch out for synonyms and near-synonyms. The SRS is not a place to creatively vary your language in an attempt to keep the reader's interest.
- Decompose a vague top-level requirement into sufficient detail to clarify it and remove ambiguity.

- 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. Specify the trigger condition or action that causes the system to perform the specified behavior. For example, "If the requested chemical is found in the chemical stockroom, the system shall display a list of all containers of the chemical that are currently in the stockroom." You may use "must" as a synonym for "shall," but avoid "should," "may," "might," and similar words that don't clarify whether the function is required.
- When stating a requirement in the form "The user shall...," identify the specific actor whenever possible (for example, "The Buyer shall...").
- Use lists, figures, graphs, and tables to present information visually. Readers glaze over when confronting a dense mass of turgid text.
- Emphasize the most important bits of information. Techniques for emphasis include graphics, sequence (the first item is emphasized), repetition, use of white space, and use of visual contrast such as shading (Kovitz 1999).
- Ambiguous language leads to unverifiable requirements, so avoid using vague and subjective terms. Table 10-1 lists several such terms, along with some suggestions for how to remove the ambiguity.

Trap

Requirements quality is in the eye of the beholder. The analyst might believe that a requirement he has written is crystal clear, free from ambiguities and other problems. However, if a reader has questions about it, the requirement needs additional work.

Table 10-1: Ambiguous Terms to Avoid in Requirements Specifications

Ambiguous Terms	**Ways to Improve Them**
acceptable, adequate	Define what constitutes acceptability and how the system can judge this.
as much as practicable	Don't leave it up to the developers to determine what's practicable. Make it a TBD and set a date to find out.
at least, at a minimum, not more than, not to exceed	Specify the minimum and maximum acceptable values.
between	Define whether the end points are included in the range.
depends on	Describe the nature of the dependency. Does another system provide input to this system, must other software be installed before your software can run, or does your system rely on another one to perform some calculations or services?

efficient

Define how efficiently the system uses resources, how quickly it performs specific operations, or how easy it is for people to use.

fast, rapid

Specify the minimum acceptable speed at which the system performs some action.

flexible

Describe the ways in which the system must change in response to changing conditions or business needs.

improved, better, faster, superior

Quantify how much better or faster constitutes adequate improvement in a specific functional area.

including, including but not limited to, and so on, etc., such as

The list of items should include all possibilities. Otherwise, it can't be used for design or testing.

maximize, minimize, optimize

State the maximum and minimum acceptable values of some parameter.

normally, ideally

Also describe the system's behavior under abnormal or non-ideal conditions.

optionally

Clarify whether this means a system choice, a user choice, or a developer choice.

reasonable, when necessary, where appropriate

Explain how to make this judgment.

robust

Define how the system is to handle exceptions and respond to unexpected operating conditions.

seamless, transparent, graceful

Translate the user's expectations into specific observable product characteristics.

several

State how many, or provide the minimum and maximum bounds of a range.

shouldn't

Try to state requirements as positives, describing what the system will do.

state-of-the-art

Define what this means.

sufficient

Specify how much of something constitutes sufficiency.

support, enable

Define exactly what functions the system will perform that constitute supporting some capability.

user-friendly, simple, easy

Describe system characteristics that will achieve the customer's usage needs and usability expectations.

Write requirements specifically enough so that if the requirement is satisfied, the customer's need will be met, but avoid unnecessarily constraining the design. Provide enough detail to reduce the risk of misunderstanding to an acceptable level, based on the development team's knowledge and experience. If a developer can think of several ways to satisfy a requirement and all are acceptable, the specificity and detail are about right. Precisely stated requirements increase the chance of people receiving what they expect; less specific requirements give the developer more latitude for interpretation. If a developer who reviews the SRS isn't clear on the customer's intent, include additional information to reduce the risk of having to fix the product later.

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's probably at an appropriate level of detail. If the tests you envision are numerous and diverse, perhaps several requirements are lumped together that ought to be separated. Testable requirements have been proposed as a metric for software product size (Wilson 1995).

Write requirements at a consistent level of detail. I've seen requirement statements in the same SRS that varied widely in their scope. For instance, "The keystroke combination Ctrl+S shall be interpreted as File Save" and "The keystroke combination Ctrl+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 (or a product!), not a single functional requirement.

Avoid long narrative paragraphs that contain multiple requirements. Words such as "and," "or," and "also" in a requirement suggest that several requirements might have been combined. This doesn't mean you can't use "and" in a requirement; just check to see whether the conjunction is joining two parts of a single requirement or two separate requirements. Never use "and/or" in a requirement; it leaves the interpretation up to the reader. Words such as "unless" and "except" also indicate multiple requirements: "The buyer's credit card on file shall be charged for payment, unless the credit card has expired." Split this into two requirements for the two conditions of the credit card: expired and nonexpired.

Avoid stating requirements redundantly. Writing the same requirement in multiple places makes the document easier to read but harder to maintain. The multiple instances of the requirement all have to be modified 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 a database solves the redundancy problem and facilitates reuse of common requirements across multiple projects.

Think about the most effective way to represent each requirement. I once reviewed an SRS that contained a set of requirements that fit the following pattern: "The Text Editor shall be able to parse <format> documents that define <jurisdiction> laws." There were three possible values for <format> and four possible values for <jurisdiction>, for a total of 12 similar requirements. The SRS had 12 requirements all right, but one was missing and another was duplicated. The only way to find that error was to build a table of all the possible combinations and look for them. This error is prevented if the SRS represents requirements that follow such a pattern in a table, as illustrated in Table 10-2. 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 the jurisdictions shown in Table 10-2." If any of the combinations don't have a corresponding functional requirement, put *N/A* (not applicable) in that table cell.

Table 10-2: 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, wasted effort, and rework down the road, strive to correct any problems early. Following are several functional requirements, adapted from real projects, that are less than ideal. Examine each statement for the preceding quality characteristics to see whether you can spot the problems. Verifiability is a good starting point. If you can't devise tests to tell whether the requirement was correctly implemented, it's probably ambiguous or lacks necessary information.

I've presented some observations about what's wrong with these requirements and offered a suggested improvement to each one. Additional review passes would improve them further, but at some point you need to write software. More examples of rewriting poor requirements are available from Hooks and Farry (2001), Florence (2002), and Alexander and Stevens (2002).

Trap

Watch out for analysis paralysis. You can't spend forever trying to perfect the requirements. Your goal 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 Background Task Manager shall provide status messages at regular intervals not less than every 60 seconds."*

What are the status messages? Under what conditions and in what fashion are they provided to the user? If displayed, how long do they remain visible? The timing interval is not clear, and the word "every" just muddles the issue. One way to evaluate a requirement is to see whether a ludicrous but legitimate interpretation is all right with the user. If not, the requirement needs more work. In this example, is the interval between status messages supposed to be at least 60 seconds, so providing a new message once per year is okay? Alternatively, if the intent is to have no more than 60 seconds elapse between messages, would one millisecond be too short an interval? These extreme interpretations are consistent with the original requirement, but they certainly aren't what the user had in mind. Because of these problems, this requirement is not verifiable.

Here's one way to rewrite the preceding requirement to address those shortcomings, after we get some more information from the customer:

1. The Background Task Manager (BTM) shall display status messages in a designated area of the user interface.

1.1

The messages shall be updated every 60 plus or minus 10 seconds after background task processing begins.

1.2

The messages shall remain visible continuously.

1.3

Whenever communication with the background task process is possible, the BTM shall display the percent completed of the background task.

1.4

The BTM shall display a "Done" message when the background task is completed.

1.5

The BTM shall display a message if the background task has stalled.

I split this into multiple child requirements because each will demand separate test cases and to make each one individually traceable. If several requirements are grouped together in a paragraph, it's easy to overlook one during construction or testing. The revised requirements don't specify how the status messages will be displayed. That's a design issue; 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 XML Editor shall switch between displaying and hiding nonprinting characters instantaneously."*

Computers can't do anything instantaneously, so this requirement isn't feasible. In addition, it's incomplete because it doesn't state the cause of the state switch. Is the software making the change on its own under certain conditions, or does the user initiate the change? What is the scope of the display change within the document: selected text, the entire document, the current page, or something else? What are nonprinting characters: hidden text, control characters, markup tags, or something else? This requirement cannot be verified until these questions are answered. The following might be a better way to write it:

The user shall be able to toggle between displaying and hiding all XML tags in the document being edited with the activation of a specific triggering mechanism. The display shall change in 0.1 second or less.

Now it's clear that the nonprinting characters are XML markup tags. We know that the user triggers the display change, but the requirement doesn't constrain the design by defining the precise mechanism. We've also added a performance requirement that defines how rapidly the display must change. "Instantaneously" really meant "instantaneously to the human eye," which is achievable with a fast enough computer.

Example 3 *"The XML parser shall produce a markup error report that allows quick resolution of errors when used by XML novices."*

The ambiguous word "quick" refers to an activity that's performed by a person, not by the parser. The lack of definition of what goes into the error report indicates incompleteness, nor do we know when the report is generated. How would you verify this requirement? Find someone who considers herself an XML novice and see whether she can resolve errors quickly enough using the report?

This requirement incorporates the important notion of a specific user class—in this case, the XML novice who needs help from the software to find XML syntax errors. The analyst should find a suitable representative of that user class to identify the information that the parser's markup error report should contain. Let's try this instead:

1. After the XML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any XML errors found in the parsed file and a description of each error found.

2. If no parsing errors are found, the parser shall not produce an error report.

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 that the original requirement didn't address: if there aren't any errors, don't generate a report.

Example 4 *"Charge numbers should be validated on line against the master corporate charge number list, if possible."*

What does "if possible" mean? If it's technically feasible? If the master charge number list can be accessed at run time? If you aren't sure whether a requested capability can be delivered, use TBD to indicate that the issue is unresolved. After investigation, either the TBD goes away or the requirement goes away. This requirement doesn't specify what happens when the validation passes or fails. Avoid imprecise words such as "should." Some requirements authors attempt to convey subtle distinctions by using words such as "shall," "should," and "may" to indicate importance. I prefer to stick with "shall" or "must" as a clear statement of the requirement's intent and to specify the priorities explicitly. Here's a revised version of this requirement:

At the time the requester enters a charge number, the system shall validate the charge number against the master corporate charge number list. If the charge number is not found on the list, the system shall display an error message and shall not accept the order.

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

Example 5 *"The editor shall not offer search and replace options that could have disastrous results."*

The notion of "disastrous results" is open to interpretation. An unintended global change could be disastrous if the user doesn't detect the error or has no way to correct it. Be judicious in the use of inverse requirements, which describe things that the system will *not* do. The underlying concern in this example seems to pertain to protecting the file contents from inadvertent damage or loss. Perhaps the real requirements are

1. The editor shall require the user to confirm global text changes, deletions, and insertions that could result in data loss.

2. The application shall provide a multilevel undo capability limited only by the system resources available to the application.

Example 6 *"The device tester shall allow the user to easily connect additional components, including a pulse generator, a voltmeter, a capacitance meter, and custom probe cards."*

This requirement is for a product containing embedded software that's used to test several kinds of measurement devices. The word *easily* implies a usability requirement, but it is neither measurable nor verifiable. "Including" doesn't make it clear whether this is the complete list of external devices that must be connected to the tester or whether there are many others that we don't know about. Consider the following alternative requirements, which contain some deliberate design constraints:

1. The tester shall incorporate a USB port to allow the user to connect any measurement device that has a USB connection.

2. The USB port shall be installed on the front panel to permit a trained operator to connect a measurement device in 15 seconds or less.

Modeling the Requirements

When I began drawing analysis models many years ago, I hoped to find one technique that could pull everything together into a holistic depiction of a system's requirements. Eventually I concluded that there is no such all-encompassing diagram. An early goal of structured systems analysis was to replace entirely the classical functional specification with graphical diagrams and notations more formal than narrative text (DeMarco 1979). However, experience has shown that analysis models should augment—rather than replace—a natural language requirements specification (Davis 1995).

Visual requirements models include data flow diagrams (DFD), entity-relationship diagrams (ERD), state-transition diagrams (STD) or statecharts, dialog maps, use-case diagrams (discussed in Chapter 8), class diagrams, and activity diagrams (also in Chapter 8). The notations presented here provide a common, industry-standard language for project participants to use. Of course, you may also use ad hoc diagrams to augment your verbal and written project communications, but readers might not interpret them the same way. Unconventional modeling approaches sometimes are valuable, however. One project team used a project-scheduling tool to model the timing requirements for an embedded software product, working at the millisecond time scale rather than in days and weeks.

These models are useful for elaborating and exploring the requirements, as well as for designing software solutions. Whether you are using them for analysis or for design depends on the timing and the intent of the modeling. Used for requirements analysis, these diagrams let you model the problem domain or create conceptual representations of the new system. They depict the logical aspects of the problem domain's data components, transactions and transformations, real-world objects, and changes in system

state. You can base the models on the textual requirements to represent them from different perspectives, or you can derive the detailed functional requirements from high-level models that are based on user input. During design, models represent specifically how you intend to implement the system: the actual database you plan to create, the object classes you'll instantiate, and the code modules you'll develop.

Trap

Don't assume that developers can simply translate analysis models into code without going through a design process. Because both types of diagrams use the same notations, clearly label each one as an analysis model (the concepts) or a design model (what you intend to build).

The analysis modeling techniques described in this chapter are supported by a variety of commercial computer-aided software engineering, or CASE, tools. CASE tools provide several benefits over ordinary drawing tools. First, they make it easy to improve the diagrams through iteration. You'll never get a model right the first time you draw it, so iteration is a key to success in system modeling (Wiegers 1996a). CASE tools also know the rules for each modeling method they support. They can identify syntax errors and inconsistencies that people who review the diagrams might not see. Many tools link multiple diagrams together and to their shared data definitions in a data dictionary. CASE tools can help you keep the models consistent with each other and with the functional requirements in the SRS.

Rarely does a team need to create a complete set of analysis models for an entire system. Focus your modeling on the most complex and riskiest portions of the system and on those portions most subject to ambiguity or uncertainty. Safety-, security-, and mission-critical system elements are good candidates for modeling because the impact of defects in those areas is so severe.

From Voice of the Customer to Analysis Models

By listening carefully to how customers present their requirements, the analyst can pick out keywords that translate into specific model elements. Table 11-1 suggests possible mappings of significant nouns and verbs from the customer's input into model components, which are described later in this chapter. As you craft customer input into written requirements and models, you should be able to link every model component to a specific user requirement.

Table 11-1: Relating the Customer's Voice to Analysis Model Components

Type of Word

Examples

Analysis Model Components

Noun

- People, organizations, software systems, data items, or objects that exist

- Terminators or data stores (DFD)
- Actors (use-case diagram)
- Entities or their attributes (ERD)
- Classes or their attributes (class diagram)

Verb

- Actions, things a user can do, or events that can take place
- Processes (DFD)
- Use cases (use-case diagram)
- Relationships (ERD)
- Transitions (STD)
- Activities (activity diagram)

Throughout this book, I've used the Chemical Tracking System as a case study. Building on this example, consider the following paragraph of user needs supplied by the product champion who represented the Chemist user class. Significant unique nouns are highlighted in **bold** and verbs are in *italics;* look for these keywords in the analysis models shown later in this chapter. For the sake of illustration, some of the models show information that goes beyond that contained in the following paragraph, whereas other models depict just part of the information presented here:

"A **chemist** or a member of the **chemical stockroom staff** can *place* a **request** for one or more **chemicals**. The request can be *fulfilled* either by *delivering* a **container** of the chemical that is already in the **chemical stockroom's inventory** or by *placing* an **order** for a new container of the chemical with an outside **vendor**. The **person** placing the request must be able to *search* **vendor catalogs** on line for specific chemicals while *preparing* his or her request. The system needs to *track* the **status** of every chemical request from the time it is prepared until the request is either fulfilled or *canceled*. It also needs to track the **history** of every chemical container from the time it is *received* at the **company** until it is fully *consumed* or *disposed* of."

Benefits from a High-Quality Requirements Process

Organizations that implement effective requirements engineering processes can reap multiple benefits. A great reward comes from reducing unnecessary rework during the late development stages and throughout the lengthy maintenance period. The high leveraging effect of quality requirements isn't obvious, and many people mistakenly believe that time spent discussing requirements simply delays delivery by the same duration. A holistic cost-of-quality perspective reveals the value of emphasizing early-stage quality practices (Wiegers 1996a).

Sound requirements processes emphasize a collaborative approach to product development that involves multiple stakeholders in a partnership throughout the project. Collecting requirements enables the development team to better understand its user community or market, a critical success factor for any project. It's far cheaper to reach this understanding before you build the product than after your customers receive it.

Engaging users in the requirements-gathering process generates enthusiasm for the product and builds customer loyalty. By emphasizing user tasks instead of superficially attractive features, the team can avoid writing code that won't ever be used. Customer involvement reduces the expectation gap between what the user needs and what the developer delivers. You're going to get the customer feedback eventually. Try to get it early rather than late, perhaps with the help of prototypes that stimulate user feedback. Requirements development takes time, but it takes less time than correcting a lot of problems in beta testing or after release.

There are additional benefits, too. Explicitly allocating selected system requirements to various software, hardware, and human subsystems emphasizes a systems approach to product engineering. An effective change-control process will minimize the adverse impact of requirements changes. Documented, unambiguous requirements greatly facilitate system testing, which in turn increases your chances of delivering high-quality products that satisfy all stakeholders.

No one can promise a specific return on investment from an improved requirements process. You can go through an analytical thought process to imagine how better requirements could help you, though. First, consider the cost of investing in a better process. This includes the cost of assessing your current practices, developing new procedures and document templates, training the team, buying books and tools, and perhaps using outside consultants. Your greatest investment is the time your teams spend gathering, documenting, reviewing, and managing their requirements. Next, think about the possible benefits you might enjoy and how much time or money they could save you. It's impossible to quantify all the following benefits, but they are real:

- Fewer requirements defects
- Reduced development rework
- Fewer unnecessary features
- Lower enhancement costs
- Faster development
- Fewer miscommunications
- Reduced scope creep

- Reduced project chaos
- More accurate system-testing estimates
- Higher customer and team member satisfaction

Characteristics of Excellent Requirements

How can you distinguish good requirements specifications from those with problems? Several characteristics that individual requirement statements should exhibit are discussed in this section, followed by desirable characteristics of the SRS as a whole (Davis 1993; IEEE 1998b). The best way to tell whether your requirements have these desired attributes is to have several project stakeholders carefully review the SRS. Different stakeholders will spot different kinds of problems. For example, analysts and developers can't accurately judge completeness or correctness, whereas users can't assess technical feasibility.

Requirement Statement Characteristics

In an ideal world, every individual user, business, and functional requirement would exhibit the qualities described in the following sections.

Complete

Each requirement must fully describe the functionality to be delivered. It must contain all the information necessary for the developer to design and implement that bit of functionality. If you know you're lacking certain information, use *TBD* (to be determined) as a standard flag to highlight these gaps. Resolve all TBDs in each portion of the requirements before you proceed with construction of that portion.

Correct

Each requirement must accurately describe the functionality to be built. The reference for correctness is the source of the requirement, such as an actual user or a high-level system requirement. A software requirement that conflicts with its parent system requirement is not correct. Only user representatives can determine the correctness of user requirements, which is why users or their close surrogates must review the requirements.

Feasible

It must be possible to implement each requirement within the known capabilities and limitations of the system and its operating environment. To avoid specifying unattainable requirements, have a developer work with marketing or the requirements analyst throughout the elicitation process. The developer can provide a reality check on what can and cannot be done technically and what can be done only at excessive cost. Incremental development approaches and proof-of-concept prototypes are ways to evaluate requirement feasibility.

Necessary

Each requirement should document a capability that the customers really need or one that's required for conformance to an external system requirement or a standard. Every requirement should originate from a source that has the authority to specify requirements. Trace each requirement back to specific voice-of-the-customer input, such as a use case, a business rule, or some other origin.

Prioritized

Assign an implementation priority to each functional requirement, feature, or use case to indicate how essential it is to a particular product release. If all the requirements are

considered equally important, it's hard for the project manager to respond to budget cuts, schedule overruns, personnel losses, or new requirements added during development.

More Info Chapter 14, "Setting Requirement Priorities," discusses prioritization in further detail.

Unambiguous

All readers of a requirement statement should arrive at a single, consistent interpretation of it, but natural language is highly prone to ambiguity. Write requirements in simple, concise, straightforward language appropriate to the user domain. "Comprehensible" is a requirement quality goal related to "unambiguous": readers must be able to understand what each requirement is saying. Define all specialized terms and terms that might confuse readers in a glossary.

Verifiable

See whether you can devise a few tests or use other verification approaches, such as inspection or demonstration, to determine whether the product properly implements each requirement. If a requirement isn't verifiable, determining whether it was correctly implemented becomes a matter of opinion, not objective analysis. Requirements that are incomplete, inconsistent, infeasible, or ambiguous are also unverifiable (Drabick 1999).

Requirements Specification Characteristics

It's not enough to have excellent individual requirement statements. Sets of requirements that are collected into a specification ought to exhibit the characteristics described in the following sections.

Complete

No requirements or necessary information should be absent. Missing requirements are hard to spot because they aren't there! Chapter 7 suggests some ways to find missing requirements. Focusing on user tasks, rather than on system functions, can help you to prevent incompleteness.

Consistent

Consistent requirements don't conflict with other requirements of the same type or with higher-level business, system, or user requirements. Disagreements between requirements must be resolved before development can proceed. You might not know which single requirement (if any) is correct until you do some research. Recording the originator of each requirement lets you know who to talk to if you discover conflicts.

Modifiable

You must be able to revise the SRS when necessary and to maintain a history of changes made to each requirement. This dictates that each requirement be uniquely labeled and expressed separately from other requirements so that you can refer to it unambiguously. Each requirement should appear only once in the SRS. It's easy to generate inconsistencies by changing only one instance of a duplicated requirement. Consider cross-referencing subsequent instances back to the original statement instead of duplicating the requirement. A table of contents and an index will make the SRS easier to modify. Storing requirements in a database or a commercial requirements management tool makes them into reusable objects.

Traceable

A traceable requirement can be linked backward to its origin and forward to the design elements and source code that implement it and to the test cases that verify the implementation as correct. Traceable requirements are uniquely labeled with persistent identifiers. They are written in a structured, fine-grained way as opposed to crafting long narrative paragraphs. Avoid lumping multiple requirements together into a single statement; the different requirements might trace to different design and code elements.

More Info Chapter 10 discusses writing high-quality functional requirements, and Chapter 20, "Links in the Requirements Chain," addresses requirements tracing.

You'll never create an SRS in which *all* requirements demonstrate *all* these ideal attributes. However, if you keep these characteristics in mind while you write and review the requirements, you will produce better requirements documents and you will build better products.

Next Steps

- Write down requirements-related problems that you have encountered on your current or previous project. Identify each as a requirements development or requirements management problem. Identify the impact that each problem had on the project and its root causes.
- Facilitate a discussion with your team members and other stakeholders regarding requirements-related problems from your current or previous projects, the problems' impacts, and their root causes. Explain that the participants have to confront these difficult issues if they ever hope to master them. Are they ready to try?
- Arrange a one-day training class on software requirements for your entire project team. Include key customers, marketing staff, and managers, using whatever it takes to get them into the room. Training is an effective team-building activity. It gives project participants a common vocabulary and a shared understanding of effective techniques and behaviors so that they all can begin addressing their mutual challenges.

Chapter 3: Good Practices for Requirements Engineering

Overview

Ten or fifteen years ago, I was a fan of software development methodologies—packaged sets of models and techniques that purport to provide holistic solutions to our project challenges. Today, though, I prefer to identify and apply industry best practices. Rather than devising or purchasing a whole-cloth solution, the best-practice approach stocks your software tool kit with a variety of techniques you can apply to diverse problems.

Even if you do adopt a commercial methodology, adapt it to best suit your needs and augment its components with other effective practices from your tool kit.

The notion of best practices is debatable: who decides what is "best" and on what basis? One approach is to convene a body of industry experts or researchers to analyze projects from many different organizations (Brown 1996; Brown 1999; Dutta, Lee, and Van Wassenhove 1999). These experts look for practices whose effective performance is associated with successful projects and which are performed poorly or not at all on failed projects. Through these means, the experts reach consensus on the activities that consistently yield superior results. Such activities are dubbed *best practices*. This implies that they represent highly effective ways for software professionals to increase the chance of success on certain kinds of projects and in certain situations.

Table 3-1 lists nearly 50 practices, grouped into seven categories, that can help most development teams do a better job on their requirements activities. Several of the practices contribute to more than one category, but each practice appears only once in the table. These practices aren't suitable for every situation, so use good judgment, common sense, and experience instead of ritualistically following a script. Note that not all of these items have been endorsed as industry best practices, which is why I've titled this chapter "Good Practices for Requirements Engineering," not "Best Practices." I doubt whether all of these practices will ever be systematically evaluated for this purpose. Nonetheless, many other practitioners and I have found these techniques to be effective (Sommerville and Sawyer 1997; Hofmann and Lehner 2001). Each practice is described briefly in this chapter, and references are provided to other chapters in this book or to other sources where you can learn more about the technique. The last section of this chapter suggests a requirements development process—a sequence of activities—that is suitable for most software projects.

Table 3-1: Requirements Engineering Good Practices

Knowledge

Requirements Management

Project Management

- Train requirements analysts
- Educate user reps and managers about requirements
- Train developers in application domain
- Create a glossary
- Define change-control process
- Establish change control board
- Perform change impact analysis
- Baseline and control versions of requirements
- Maintain change history

- Track requirements status
- Measure requirements volatility
- Use a requirements management tool
- Create requirements traceability matrix
- Select appropriate life cycle
- Base plans on requirements
- Renegotiate commitments
- Manage requirements risks
- Track requirements effort
- Review past lessons learned

Requirements Development

Elicitation

Analysis

Specification

Validation

- Define requirements development process
- Define vision and scope
- Identify user classes
- Select product champions
- Establish focus groups
- Identify use cases
- Identify system events and responses
- Hold facilitated elicitation workshops
- Observe users performing their jobs
- Examine problem reports
- Reuse requirements
- Draw context diagram

- Create prototypes
- Analyze feasibility
- Prioritize requirements
- Model the requirements
- Create a data dictionary
- Allocate requirements to subsystems
- Apply Quality Function Deployment
- Adopt SRS template
- Identify sources of requirements
- Uniquely label each requirement
- Record business rules
- Specify quality attributes
- Inspect requirements documents
- Test the requirements
- Define acceptance criteria

Knowledge

Few software developers receive formal training in requirements engineering. However, many developers perform the role of requirements analyst at some point in their careers, working with customers to gather, analyze, and document requirements. It isn't reasonable to expect all developers to be instinctively competent at the communication-intensive tasks of requirements engineering. Training can increase the proficiency and comfort level of those who serve as analysts, but it can't compensate for missing interpersonal skills or a lack of interest.

Because the requirements process is essential, all project stakeholders should understand the concepts and practices of requirements engineering. Bringing together the various stakeholders for a one-day overview on software requirements can be an effective team-building activity. All parties will better appreciate the challenges their counterparts face and what the participants require from each other for the whole team to succeed. Similarly, developers should receive grounding in the concepts and terminology of the application domain. You can find further details on these topics in the following chapters:

- Chapter 4—Train requirements analysts.
- Chapter 10—Create a project glossary.

Train requirements analysts. All team members who will function as analysts should receive basic training in requirements engineering. Requirements analyst specialists need several days of training in these activities. The skilled requirements analyst is patient and well organized, has effective interpersonal and communication skills, understands the application domain, and has an extensive tool kit of requirements-engineering techniques.

Educate user representatives and managers about software requirements. Users who will participate in software development should receive one or two days of education about requirements engineering. Development managers and customer managers will also find this information useful. The training will help them understand the value of emphasizing requirements, the activities and deliverables involved, and the risks of neglecting requirements processes. Some users who have attended my requirements seminars have said that they came away with more sympathy for the software developers.

Train developers in application domain concepts. To help developers achieve a basic understanding of the application domain, arrange a seminar on the customer's business activities, terminology, and objectives for the product being created. This can reduce confusion, miscommunication, and rework down the road. You might also match each developer with a "user buddy" for the life of the project to translate jargon and explain business concepts. The product champion could play this role.

Create a project glossary. A glossary that defines specialized terms from the application domain will reduce misunderstandings. Include synonyms, terms that can have multiple meanings, and terms that have both domain-specific and everyday meanings. Words that can be both nouns and verbs—such as "process" and "order"—can be particularly confusing.

Requirements Elicitation

Chapter 1 discussed the three levels of requirements: business, user, and functional. These come from different sources at different times during the project, have different audiences and purposes, and need to be documented in different ways. The business requirements expressed in the project scope must not exclude any essential user requirements, and you should be able to trace all functional requirements back to specific user requirements. You also need to elicit nonfunctional requirements, such as quality and performance expectations, from appropriate sources. You can find additional information about these topics in the following chapters:

- Chapter 3—Define a requirements development process.
- Chapter 5—Write a vision and scope document.
- Chapter 6—Identify user classes and their characteristics; select a product champion for each user class; observe users performing their jobs.
- Chapter 7—Hold facilitated elicitation workshops.
- Chapter 8—Work with user representatives to identify use cases; identify system events and responses.
- Chapter 22—Define a requirements development process.

Define a requirements development process. Document the steps your organization follows to elicit, analyze, specify, and validate requirements. Providing guidance on how to perform the key steps will help analysts do a consistently good job. It will also make it easier to plan each project's requirements development tasks, schedule, and required resources.

Write a vision and scope document. The vision and scope document contains the product's business requirements. The vision statement gives all stakeholders a common understanding of the product's objectives. The scope defines the boundary between what's in and what's out for a specific release. Together, the vision and scope provide a

reference against which to evaluate proposed requirements. The product vision should remain relatively stable from release to release, but each release needs its own project scope statement.

Identify user classes and their characteristics. To avoid overlooking the needs of any user community, identify the various groups of users for your product. They might differ in frequency of use, features used, privilege levels, or skill levels. Describe aspects of their job tasks, attitudes, location, or personal characteristics that might influence product design.

Select a product champion for each user class. Identify at least one person who can accurately serve as the voice of the customer for each user class. The product champion presents the needs of the user class and makes decisions on its behalf. This is easiest for internal information systems development, where your users are fellow employees. For commercial development, build on your current relationships with major customers or beta test sites to locate appropriate product champions. Product champions must have ongoing participation in the project and the authority to make decisions at the user-requirements level.

Establish focus groups of typical users. Convene groups of representative users of your previous product releases or of similar products. Collect their input on both functionality and quality characteristics for the product under development. Focus groups are particularly valuable for commercial product development, for which you might have a large and diverse customer base. Unlike product champions, focus groups generally do not have decision-making authority.

Work with user representatives to identify use cases. Explore with your user representatives the tasks they need to accomplish with the software—their use cases. Discuss the interactions between the users and the system that will allow them to complete each such task. Adopt a standard template for documenting use cases and derive functional requirements from those use cases. A related practice that is often used on government projects is to define a concept of operations (ConOps) document, which describes the new system's characteristics from the user's point of view (IEEE 1998a).

Identify system events and responses. List the external events that the system can experience and its expected response to each event. Events include signals or data received from external hardware devices and temporal events that trigger a response, such as an external data feed that your system generates at the same time every night. Business events trigger use cases in business applications.

Hold facilitated elicitation workshops. Facilitated requirements-elicitation workshops that permit collaboration between analysts and customers are a powerful way to explore user needs and to draft requirements documents (Gottesdiener 2002). Specific examples of such workshops include Joint Requirements Planning (JRP) sessions (Martin 1991) and Joint Application Development (JAD) sessions (Wood and Silver 1995).

Observe users performing their jobs. Watching users perform their business tasks establishes a context for their potential use of a new application (Beyer and Holtzblatt 1998). Simple workflow diagrams—data flow diagrams work well—can depict when the user has what data and how that data is used. Documenting the business process flow will help you identify requirements for a system that's intended to support that process.

You might even determine that the customers don't really need a new software application to meet their business objectives (McGraw and Harbison 1997).

Examine problem reports of current systems for requirement ideas. Problem reports and enhancement requests from customers provide a rich source of ideas for capabilities to include in a later release or in a new product. Help desk and support staff can provide valuable input to the requirements for future development work.

Reuse requirements across projects. If customers request functionality similar to that already present in an existing product, see whether the requirements (and the customers!) are flexible enough to permit reusing or adapting the existing components. Multiple projects will reuse those requirements that comply with an organization's business rules. These include security requirements that control access to the applications and requirements that conform to government regulations, such as the Americans with Disabilities Act.