Part II: Software Requirements Development

Chapter List

Chapter 5: Establishing the Product Vision and Project Scope

Overview

True Stories When my colleague Karen introduced requirements document inspections in her company, she observed that many of the issues the inspectors raised pertained to project scope. The inspectors often held different understandings of the project's intended scope and objectives. Consequently, they had difficulty agreeing on which functional requirements belonged in the SRS.

As we saw in Chapter 1, "The Essential Software Requirement," the business requirements represent the top level of abstraction in the requirements chain: they define the vision and scope for the software system. The user requirements and software functional requirements must align with the context and objectives that the business requirements establish. Requirements that don't help the project achieve its business objectives shouldn't be included.

A project that lacks a clearly defined and well-communicated direction invites disaster. Project participants can unwittingly work at cross-purposes if they have different objectives and priorities. The stakeholders will never agree on the requirements if they lack a common understanding of the business objectives for the product. A clear vision and scope is especially critical for multisite development projects, where geographical separation inhibits the day-to-day interactions that facilitate teamwork.

One sign that the business requirements are insufficiently defined is that certain features are initially included, then deleted, and then added back in later. Vision and scope issues must be resolved before the detailed functional requirements can be fully specified. A statement of the project's scope and limitations helps greatly with discussions of proposed features and target releases. The vision and scope also provide a reference for

making decisions about proposed requirement changes and enhancements. Some companies print the vision and scope highlights on a poster board that's brought to every project meeting so that they can quickly judge whether a proposed change is in or out of scope.

Defining the Vision Through Business Requirements

The *product vision* aligns all stakeholders in a common direction. The vision describes what the product is about and what it eventually could become. The *project scope* identifies what portion of the ultimate long-term product vision the current project will address. The statement of scope draws the boundary between what's in and what's out. That is, the scope also defines the project's limitations. The details of a project's scope are represented by the requirements baseline that the team defines for that project.

The vision applies to the product as a whole. It will change relatively slowly as a product's strategic positioning or an information system's business objectives evolve over time. The scope pertains to a specific project or iteration that will implement the next increment of the product's functionality, as shown in Figure 5-1. Scope is more dynamic than vision because the project manager adjusts the contents of each release within its schedule, budget, resource, and quality constraints. The planner's goal is to manage the scope of a specific development or enhancement project as a defined subset of the grand strategic vision. The scope statement for each project, or for each iteration or enhancement in an evolving product, can appear in that project's SRS, rather than in a separate vision and scope document. Major new projects should have both a complete vision and scope document and an SRS. See Chapter 10, "Documenting the Requirements," for an SRS template.

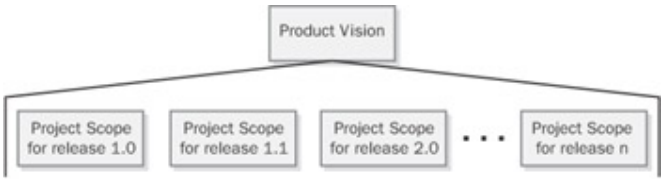


Figure 5-1: The product vision encompasses the scope for each planned release.

True Stories For example, a federal government agency is undertaking a massive five-year information system development effort. The agency defined the business objectives and vision for this system early in the process, and they won't change substantially over the next few years. The agency has planned some 15 individual releases of portions of the ultimate system. Each release is created as a separate project with its own scope description. Each scope description must align with the overall product vision and interlock with the scope statements for the other projects to ensure that nothing is inadvertently omitted.

Blue-Sky Requirements

True Stories A manager at a product development company that suffered near-catastrophic scope creep once told me ruefully, "We blue-skied the requirements too much." She meant that any idea anyone had was included in the requirements. This company had a solid product vision but didn't manage the scope by planning a series of releases and deferring some suggested features to later (perhaps infinitely later) releases. The team finally released an overinflated product after four years of development. Smart scope management and an evolutionary development approach would have let the team ship a useful product much earlier.

Conflicting Business Requirements

Business requirements collected from multiple sources might conflict. Consider a kiosk containing embedded software, which will be used by a retail store's customers. The kiosk developer's business objectives include the following:

- Generating revenue by leasing or selling the kiosk to the retailer
- Selling consumables through the kiosk to the customer
- Attracting customers to the brand
- Making a wide variety of products available

The retailer's business interests could include the following:

- Maximizing revenue from the available floor space
- Attracting more customers to the store
- Increasing sales volume and profit margins if the kiosk replaces manual operations

The developer might want to establish a high-tech and exciting new direction for customers. The retailer wants a simple turnkey system, and the customer favors convenience and features. The tension among these three parties with their different goals, constraints, and cost factors can lead to inconsistent business requirements. The project sponsor must resolve these conflicts before the analyst can detail the kiosk's system and software requirements. The focus should be on the fundamental objectives for the product that will deliver the maximum business value ("increase sales to new and existing customers"). It's easy to be distracted by superficial product characteristics ("innovative user interface that attracts customers") that don't really state the business objective.

It's also up to the project sponsor (or sponsors) to resolve conflicts among various business stakeholders, rather than expecting the software team to somehow figure these out. As more stakeholders are identified and more constituencies with competing interests climb aboard, the risk of scope creep increases. Uncontrolled scope creep in which stakeholders overstuff the new system in an attempt to satisfy every conceivable interest can cause the project to topple under its own weight, never delivering anything of value. Resolving such issues is often a political and power struggle, which lies outside the scope of this book.

Business Requirements and Use Cases

The business requirements determine both the set of business tasks (use cases) that the application enables (the application *breadth*) and the *depth* or level to which each use case is implemented. If the business requirements help you determine that a particular use case is outside the project's scope, you're making a breadth decision. The depth of support can range from a trivial implementation to full automation with many usability aids. The business requirements will imply which use cases demand robust, comprehensive functional implementation and which require merely superficial implementation, at least initially.

The business requirements influence the implementation priorities for use cases and their associated functional requirements. For example, a business objective to generate maximum revenue from a kiosk implies the early implementation of features that sell more products or services to the customer. Exotic, glitzy features that appeal to only a few technology-hungry customers and don't contribute to the primary business objective shouldn't have high priority.

Business requirements also materially influence the way requirements are implemented. For example, one motivation for building the Chemical Tracking System was to purchase fewer new bottles of a chemical by increasing the use of chemicals that are already in the stockroom or in another laboratory. Interviews and

observation should reveal why chemical reuse is not happening now. This information in turn leads to functional requirements and designs that make it easy to track the chemicals in every lab and to help a requester find chemicals in other labs near the requester's location.

Vision and Scope Document

The *vision and scope document* collects the business requirements into a single document that sets the stage for the subsequent development work. Some organizations create a project charter or a business case document that serves a similar purpose. Organizations that build commercial software often create a *market requirements document* (MRD). An MRD might go into more detail than a vision and scope document about the target market segments and the issues that pertain to commercial success.

The owner of the vision and scope document is the project's executive sponsor, funding authority, or someone in a similar role. A requirements analyst can work with the owner to write the vision and scope document. Input on the business requirements should come from individuals who have a clear sense of why they are undertaking the project. These individuals might include the customer or development organization's senior management, a project visionary, a product manager, a subject matter expert, or members of the marketing department.

Figure 5-2 suggests a template for a vision and scope document. Document templates standardize the structure of the documents created by your organization's project teams. As with any template, adapt this one to meet the specific needs of your own projects.

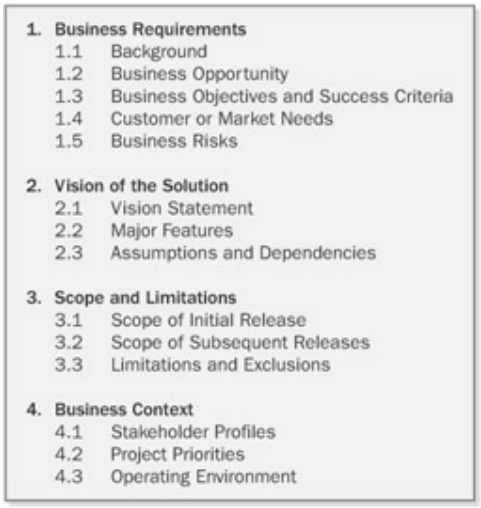


Figure 5-2: Template for vision and scope document.

Parts of the vision and scope document might seem repetitive, but they should interlock in a sensible way. Consider the following example:

Business Opportunity Exploit the poor security record of a competing product.

Business Objective Capture a market share of 80 percent by being recognized as the most secure product in the market through trade journal reviews and consumer surveys.

Customer Need A more secure product.

Feature A new, robust security engine.

1. Business Requirements

Projects are launched in the belief that the new product will make the world a better place in some way for someone. The business requirements describe the primary benefits that the new system will provide to its sponsors, buyers, and users. The emphasis will be different for different kinds of products, such as information systems, commercial software packages, and real-time control systems.

1.1 Background

Summarize the rationale and context for the new product. Provide a general description of the history or situation that led to the decision to build this product.

1.2 Business Opportunity

For a commercial product, describe the market opportunity that exists and the market in which the product will be competing. For a corporate information system, describe the business problem that is being solved or the business process being improved, as well as the environment in which the system will be used. Include a comparative evaluation of existing products and potential solutions, indicating why the proposed product is attractive and the advantages it provides. Describe the problems that cannot currently be solved without the product. Show how it aligns with market trends, technology evolution, or corporate strategic directions. Include a brief description of any other technologies, processes, or resources required to provide a complete customer solution.

1.3 Business Objectives and Success Criteria

Summarize the important business benefits the product will provide in a quantitative and measurable way. Table 5-1 presents some examples of both financial and non-financial business objectives (Wiegers 2002c). If such information appears elsewhere, such as in a business case document, refer to the other document rather than duplicate it here. Determine how the stakeholders will define and measure success on this project (Wiegers 2002c). State the factors that have the greatest impact on achieving that success, including factors both within and outside the organization's control. Specify measurable criteria to assess whether the business objectives have been met.

Table 5-1: Examples of Financial and Nonfinancial Business Objectives

Financial	Nonfinancial
• Capture a market share of X% within Y months. • Increase market share in country X by Y% in Z months. • Reach a sales volume of X units or revenue of $Y within Z months. • Achieve X% profit or return on investment • Achieve positive cash flow on this product within Y months.within Y months. • Save $X per year currently spent on a high-maintenance legacy system. • Reduce support costs by X% within Y months.	• Achieve a customer satisfaction measure of at least X within Y months of release. • Increase transaction-processing productivity by X% and reduce data error rate to no more than Y%. • Achieve a specified time to market that establishes a dominant market presence. • Develop a robust platform for a family of related products. • Develop specific core technology competencies in the organization. • Have X positive product reviews appear in trade journals before a specified date.

• Receive no more than X service calls per unit and Y warranty calls per unit within Z months after shipping. • Increase gross margin on existing business from X% to Y%.	• Be rated as the top product for reliability in published product reviews by a specified date. • Comply with specific federal and state regulations. • Reduce turnaround time to X hours on Y% of customer support calls.

1.4 Customer or Market Needs

Describe the needs of typical customers or of the target market segment, including needs that current products or information systems do not meet. Present the problems that customers currently encounter that the new product will address and provide examples of how customers would use the product. Define at a high level any known critical interface or performance requirements, but do not include design or implementation details.

1.5 Business Risks

Summarize the major business risks associated with developing—or not developing—this product. Risk categories include marketplace competition, timing issues, user acceptance, implementation issues, and possible negative impacts on the business. Estimate the potential loss from each risk, the likelihood of it occurring, and your ability to control it. Identify any potential mitigation actions. If you already prepared this information for a business case analysis or a similar document, refer to that other source rather than duplicating the information here.

2. Vision of the Solution

This section of the document establishes a strategic vision for the system that will achieve the business objectives. This vision provides the context for making decisions throughout the course of the product's life. It should not include detailed functional requirements or project planning information.

2.1 Vision Statement

Write a concise vision statement that summarizes the long-term purpose and intent of the new product. The vision statement should reflect a balanced view that will satisfy the needs of diverse stakeholders. It can be somewhat idealistic but should be grounded in the realities of existing or anticipated markets, enterprise architectures, corporate strategic directions, and resource limitations. The following keyword template works well for a product vision statement (Moore 1991):

- ***For*** [target customer]
- ***Who*** statement of the need or opportunity]
- ***The*** [product name]
- ***Is*** [a product category]
- ***That*** [key benefit, compelling reason to buy or use]
- ***Unlike*** [primary competitive alternative, current system, or current business process],
- ***Our product*** [statement of primary differentiation and advantages of new product].

Here's a sample vision statement for the Chemical Tracking System project at Contoso Pharmaceuticals that was introduced in Chapter 2, with the keywords shown in boldface:

> ***For*** *scientists* ***who*** *need to request containers of chemicals,* ***the*** *Chemical Tracking System* ***is*** *an information system* ***that*** *will provide a single point of access to the chemical stockroom and to vendors. The system will store the location of every chemical container within the company, the quantity of material remaining in it, and the complete history of each container's locations and usage. This system will save the company 25 percent on chemical costs in the first year of use by allowing the company to fully exploit chemicals that are already available within the company, dispose of fewer partially used or expired containers, and use a single standard chemical purchasing process.* ***Unlike*** *the current manual ordering processes,* ***our product*** *will generate all reports required to comply with federal and state government regulations that require the reporting of chemical usage, storage, and disposal.*

2.2 Major Features

Name or number each of the new product's major features or user capabilities in a unique, persistent way, emphasizing those features that distinguish it from previous or competing products. Giving each feature a unique label (as opposed to a bullet) permits tracing it to individual user requirements, functional requirements, and other system elements.

2.3 Assumptions and Dependencies

Record any assumptions that the stakeholders made when conceiving the project and writing this vision and scope document. Often the assumptions that one party holds are not shared by other parties. If you write them down and review them, you can agree on the project's basic underlying assumptions. This avoids possible confusion and aggravation in the future. For instance, the executive sponsor for the Chemical Tracking System assumed that it would replace the existing chemical stockroom inventory system and that it would interface to Contoso's purchasing systems. Also record major dependencies the project has on external factors outside its control. These could include pending industry standards or government regulations, other projects, third-party suppliers, or development partners.

3. Scope and Limitations

When a chemist invents a new reaction that transforms one kind of chemical into another, he writes a paper that includes a "Scope and Limitations" section, which describes what the reaction will and will not do. Similarly, a software project should define its scope and limitations. You need to state both what the system *is* and what it *is not*.

The project scope defines the concept and range of the proposed solution. The limitations itemize certain capabilities that the product will *not* include. The scope and limitations help to establish realistic stakeholder expectations. Sometimes customers request features that are too expensive or that lie outside the intended product scope. Out-of-scope requirements must be rejected unless they are so valuable that the scope should be enlarged to accommodate them, with corresponding changes in budget, schedule, and staff. Keep a record of rejected requirements and why they were rejected because they have a way of reappearing.

More Info Chapter 18, "Requirements Management Principles and Practices," describes how to use a requirements attribute to keep a record of rejected or deferred requirements.

3.1 Scope of Initial Release

Summarize the major features that are planned for inclusion in the initial release of the product. Describe the quality characteristics that will let the product provide the intended benefits to its various user classes. If your goals are to focus the development effort and to maintain a reasonable project schedule, avoid the temptation to

include in release 1.0 every feature that any potential customer might conceivably want someday. Bloatware and slipped schedules are common outcomes of such insidious scope creep. Focus on those features that will provide the most value, at the most acceptable cost, to the broadest community, in the earliest time frame.

True Stories My colleague Scott's last project team decided that users had to be able to run their package delivery business with the first software release. Version 1 didn't have to be fast, pretty, or easy to use, but it had to be reliable; this focus drove everything the team did. The initial release accomplished the basic objectives of the system. Future releases will include additional features, options, and usability aids.

3.2 Scope of Subsequent Releases

If you envision a staged evolution of the product, indicate which features will be deferred and the desired timing of later releases. Subsequent releases let you implement additional use cases and features and enrich the capabilities of the initial use cases and features (Nejmeh and Thomas 2002). You can also improve system performance, reliability, and other quality characteristics as the product matures. The farther out you look, the fuzzier these future scope statements will be. You can expect to shift functionality from one planned release to another and perhaps to add unanticipated capabilities. Short release cycles help by providing frequent opportunities for learning based on customer feedback.

3.3 Limitations and Exclusions

Defining the boundary between what's in and what's out is a way to manage scope creep and customer expectations. List any product capabilities or characteristics that a stakeholder might anticipate but that are not planned for inclusion in the product or in a specific release.

4. Business Context

This section summarizes some of the project's business issues, including profiles of major stakeholder categories and management's priorities for the project.

4.1 Stakeholder Profiles

Stakeholders are the individuals, groups, or organizations who are actively involved in a project, are affected by its outcome, or are able to influence its outcome (Project Management Institute 2000; Smith 2000). The stakeholder profiles describe different categories of customers and other key stakeholders for this project. You needn't describe every stakeholder group, such as legal staff who must check for compliance with pertinent laws. Focus on different types of customers, target market segments, and the different user classes within those segments. Each stakeholder profile should include the following information:

- The major value or benefit that the stakeholder will receive from the product and how the product will generate high customer satisfaction. Stakeholder value might include
    - Improved productivity.
    - Reduced rework.
    - Cost savings.
    - Streamlined business processes.
    - Automation of previously manual tasks.
    - Ability to perform entirely new tasks.

- Compliance with pertinent standards or regulations.
- Improved usability compared to current products.

- Their likely attitudes toward the product.
- Major features and characteristics of interest.
- Any known constraints that must be accommodated.

4.2 Project Priorities

To enable effective decision making, the stakeholders must agree on the project's priorities. One way to approach this is to consider the five dimensions of a software project: features (or scope), quality, schedule, cost, and staff (Wiegers 1996a). Each dimension fits in one of the following three categories on any given project:

A *constraint* A limiting factor within which the project manager must operate

A *driver* A significant success objective with limited flexibility for adjustment

A *degree of freedom* A factor that the project manager has some latitude to adjust and balance against the other dimensions

The project manager's goal is to adjust those factors that are degrees of freedom to achieve the project's success drivers within the limits imposed by the constraints. Not all factors can be drivers, and not all can be constraints. The project manager needs some degrees of freedom to be able to respond appropriately when project requirements or realities change. Suppose marketing suddenly demands that you release the product one month earlier than scheduled. How do you respond? Do you

- Defer certain requirements to a later release?
- Shorten the planned system test cycle?
- Pay your staff overtime or hire contractors to accelerate development?
- Shift resources from other projects to help out?

The project priorities dictate the actions you take when such eventualities arise.

4.3 Operating Environment

Describe the environment in which the system will be used and define the vital availability, reliability, performance, and integrity requirements. This information will significantly influence the definition of the system's architecture, which is the first—and often the most important—design step. A system that must support widely distributed users who require access around the clock needs a significantly different architecture from one that's employed by co-located users only during normal working hours. Nonfunctional requirements such as fault tolerance and the ability to service the system while it is running can consume considerable design and implementation effort. Ask the stakeholders questions such as the following:

- Are the users widely distributed geographically or located close to each other? How many time zones are they in?

- When do the users in various locations need to access the system?
- Where is the data generated and used? How far apart are these locations? Does data from multiple locations need to be combined?
- Are specific maximum response times known for accessing data that might be stored remotely?
- Can the users tolerate service interruptions or is continuous access to the system critical for the operation of their businesses?
- What access security controls and data protection requirements are needed?

The Context Diagram

The scope description establishes the boundary and connections between the system we are developing and everything else in the universe. The *context diagram* graphically illustrates this boundary. It identifies *terminators* outside the system that interface to it in some way, as well as data, control, and material *flows* between the terminators and the system. The context diagram is the top level of abstraction in a data flow diagram developed according to the principles of structured analysis (Robertson and Robertson 1994), but it's a useful model for projects that follow any development methodology. You can include the context diagram in the vision and scope document, in or as an appendix to the SRS, or as part of a data flow model for the system.

Figure 5-3 illustrates a portion of the context diagram for the Chemical Tracking System. The entire system is depicted as a single circle; the context diagram deliberately provides no visibility into the system's internal objects, processes, and data. The "system" inside the circle could encompass any combination of software, hardware, and human components. The terminators in the rectangles can represent user classes ("Chemist" or "Buyer"), organizations ("Health and Safety Department"), other systems ("Training Database"), or hardware devices ("Bar Code Reader"). The arrows on the diagram represent the flow of data ("request for chemical") or physical items ("chemical container") between the system and the terminators.

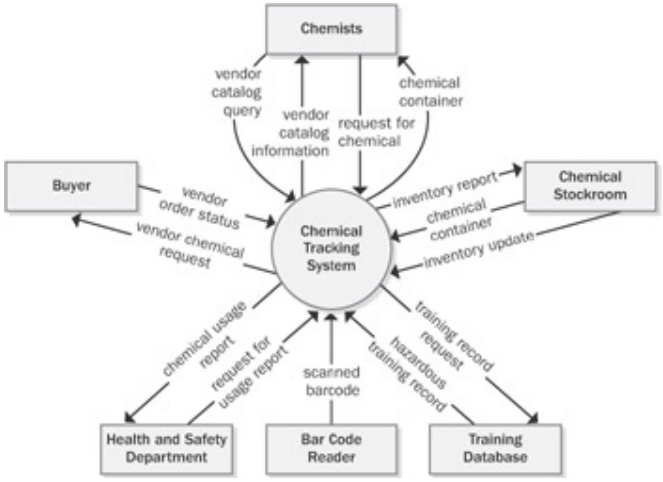


Figure 5-3: Context diagram for the Chemical Tracking System.

You might expect to see chemical vendors shown as a terminator in this diagram. After all, the company will route orders to vendors for fulfillment, the vendors will send chemical containers and invoices to Contoso Pharmaceuticals, and purchasing will send checks to the vendors. However, those processes take place outside

the scope of the Chemical Tracking System, as part of the operations of the purchasing and receiving departments. The context diagram makes it clear that this system is not directly involved in placing orders with the vendors, receiving the products, or paying the bills.

The purpose of tools such as the context diagram is to foster clear and accurate communication among the project stakeholders. That clarity is more important than dogmatically adhering to the rules for a "correct" context diagram. I strongly recommend, though, that you adopt the notation illustrated in Figure 5-3 as a standard for drawing context diagrams. Suppose you were to use a triangle for the system instead of a circle and ovals rather than rectangles for terminators. Your colleagues would have difficulty reading a diagram that follows your personal preferences rather than a team standard.

Keeping the Scope in Focus

The business requirements and an understanding of how customers will use the product provide valuable tools for dealing with scope creep. Scope change isn't a bad thing if it helps you steer the project toward satisfying evolving customer needs. The vision and scope document lets you assess whether proposed features and requirements are appropriate for inclusion in the project. Remember, whenever someone requests a new requirement, the analyst needs to ask, "Is this in scope?"

One response might be that the proposed requirement is clearly out of scope. It might be interesting, but it should be addressed in a future release or by another project. Another possibility is that the request obviously lies within the defined project scope. You can incorporate new in-scope requirements in the project if they are of high priority relative to the other requirements that were already committed. Including new requirements often involves making a decision to defer or cancel other planned requirements.

The third possibility is that the proposed new requirement is out of scope but it's such a good idea that the project scope should be modified to accommodate it. That is, there's a feedback loop between the user requirements and the business requirements. This will require that you update the vision and scope document, which should be placed under change control at the time it is baselined. When the project's scope is increased, you will usually have to renegotiate the planned budget, resources, schedule, and perhaps staff. Ideally, the original schedule and resources will accommodate a certain amount of change because of thoughtfully included contingency buffers (Wiegers 2002d). However, unless you originally budgeted for some requirements growth, you'll need to replan after requirements changes are approved.

Scope Management and Timebox Development

True Stories Enrique, a project manager at Lightspeed Financial Systems, had to deliver an Internet-enabled version of Lightspeed's flagship portfolio management software. It would take years to fully supplant the mature application, but Lightspeed needed an Internet presence right away. Enrique selected a timebox development approach, promising to release a new version every 90 days (McConnell 1996). His marketing team carefully prioritized the product's requirements. The SRS for each quarterly release included a committed set of new and enhanced features, as well as a list of lower-priority "stretch" requirements to be implemented as time permitted. Enrique's team didn't incorporate every stretch requirement into each release, but they did ship a new, stable version every three months through this schedule-driven approach to scope management. Schedule and quality are normally constraints on a timeboxed project and scope is a degree of freedom.

A common consequence of scope creep is that completed activities must be reworked to accommodate the

changes. Quality often suffers if the allocated resources or time are not increased when new functionality is added. Documented business requirements make it easier to manage legitimate scope growth as the marketplace or business needs change. They also help a harried project manager to justify saying "no"—or at least "not yet"—when influential people try to stuff more features into an overly constrained project.

Next Steps

- Ask several stakeholders for your project to write a vision statement using the keyword template described in this chapter. See how similar the visions are. Rectify any disconnects and come up with a unified vision statement that all those stakeholders agree to.
- Whether you're near the launch of a new project or in the midst of construction, write a vision and scope document using the template in Figure 5-2 and have the rest of the team review it. This might reveal that your team doesn't share a common understanding of the product vision or project scope. Correct that problem now, rather than let it slide indefinitely; it will be even more difficult to correct if you wait. This activity will also suggest ways to modify the template to best meet the needs of your organization's projects.

Chapter 6: Finding the Voice of the Customer

Overview

If you share my conviction that customer involvement is a critical factor in delivering excellent software, you'll engage customer representatives from the outset of your project. Success in software requirements, and hence in software development, depends on getting the voice of the customer close to the ear of the developer. To find the voice of the customer, take the following steps:

- Identify the different classes of users for your product.
- Identify sources of user requirements.
- Select and work with individuals who represent each user class and other stakeholder groups.
- Agree on who the requirements decision makers are for your project.

Customer involvement is the only way to avoid an expectation gap, a mismatch between the product that customers expect to receive and the product that developers build. It's not enough simply to ask a few customers what they want and then start coding. If the developers build exactly what customers initially request, they'll probably have to build it again because customers often don't know what they really need.

The features that users present as their "wants" don't necessarily equate to the functionality they need to perform their tasks with the new product. To get a more accurate view of user needs, the requirements analyst must collect user input, analyze and clarify it, and determine just what to build to let users do their jobs. The analyst has the lead responsibility for recording the new system's necessary capabilities and properties and for communicating that information to other stakeholders. This is an iterative process that takes time. If you don't invest the time to achieve this shared understanding—this common vision of the intended product—the certain outcomes are rework, delayed completion, and customer dissatisfaction.

Sources of Requirements

The origins of your software requirements will depend on the nature of your product and your development environment. The need to gather requirements from multiple perspectives and sources exemplifies the communication-intensive nature of requirements engineering. Following are several typical sources of software requirements:

Interviews and discussions with potential users The most obvious way to find out what potential users of a new software product need is to ask them. This chapter discusses how to find suitable user representatives and Chapter 7 describes techniques for eliciting requirements from them.

Documents that describe current or competing products Documents can also describe corporate or industry standards that must be followed or regulations and laws with which the product must comply. Descriptions of both present and future business processes also are helpful. Published comparative reviews might point out shortcomings in other products that you could address to gain a competitive advantage.

System requirements specifications A product that contains both hardware and software has a high-level system requirements specification that describes the overall product. A portion of the system requirements is allocated to each software subsystem (Nelsen 1990). The analyst can derive additional detailed software functional requirements from those allocated system requirements.

Problem reports and enhancement requests for a current system The help desk and field support personnel are valuable sources of requirements. They learn about the problems that users encounter with the current system and hear ideas from users for improving the system in the next release.

Marketing surveys and user questionnaires Surveys can collect a large amount of data from a broad spectrum of potential users. Consult with an expert in survey design and administration to ensure that you ask the right questions of the right people (Fowler 1995). A survey tests your understanding of requirements that you have gathered or think that you know, but it's not a good way to stimulate creative thinking. Always beta test a survey before distributing it. It's frustrating to discover too late that a question was phrased ambiguously or to realize that an important question was omitted.

Observing users at work During a "day in the life" study, an analyst watches users of a current system or potential users of the future system perform their work. Observing a user's workflow in the task environment allows the analyst to validate information collected from previous interviews, to identify new topics for interviews, to see problems with the current system, and to identify ways that the new system can better support the workflow (McGraw and Harbison 1997; Beyer and Holtzblatt 1998). Watching the users at work provides a more accurate and complete understanding than simply asking them to write down the steps they go through. The analyst must abstract and generalize beyond the observed user's activities to ensure that the requirements captured apply to the user class as a whole and not just to that individual. A skillful analyst can also often suggest ideas for improving the user's current business processes.

Scenario analysis of user tasks By identifying tasks that users need to accomplish with the system, the analyst can derive the necessary functional requirements that will let users perform those tasks. This is the essence of the use-case approach described in Chapter 8. Be sure to include the information consumed and generated by a task and the sources of that information.

Events and responses List the external events to which the system must react and the appropriate responses. This works particularly well for real-time systems, which must read and process data streams, error codes,

control signals, and interrupts from external hardware devices.

User Classes

A product's users differ, among other ways, in the following respects:

- The frequency with which they use the product
- Their application domain experience and computer systems expertise
- The features they use
- The tasks they perform in support of their business processes
- Their access privilege or security levels (such as ordinary user, guest user, or administrator)

You can group users into a number of distinct *user classes* based on these differences. An individual can belong to multiple user classes. For example, an application's administrator might also interact with it as an ordinary user at times. The terminators shown outside your system on a context diagram (as discussed in Chapter 5) are candidates for user classes. A user class is a subset of a product's users, which is a subset of a product's customers, which is a subset of its stakeholders, as shown in Figure 6-1.

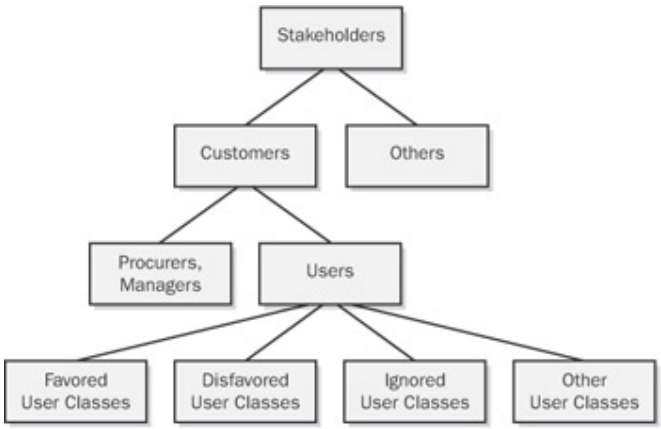


Figure 6-1: A hierarchy of stakeholders, customers, and users.

True Stories It's tempting to separate users into classes based on their geographical location or the kind of company or job that they're in, rather than with regard to how they interact with the system. For example, a company that creates software used in the banking industry initially considered distinguishing users based on whether they worked in a large commercial bank, a small commercial bank, a savings and loan institution, or a credit union. These distinctions really represent potential market segments, not different user classes. The people who accept loan applications at all such financial institutions have similar functional needs for the system, so a more logical user class name might be *application receiver*. This role could be performed by someone with the job title of Loan Officer, Vice-President, Customer Service Agent, or perhaps even Teller, if all those individuals use the system to help someone apply for a loan.

Certain user classes are more important to you than others. Favored user classes receive preferential treatment when you're making priority decisions or resolving conflicts between requirements received from different user

classes. Favored user classes include those groups whose acceptance and use of the system will cause it to meet—or fail to meet—its business objectives. This doesn't mean that the stakeholders who are paying for the system (who might not be users at all) or who have the most political clout should necessarily be favored. Disfavored user classes are those groups who aren't supposed to use the product for legal, security, or safety reasons (Gause and Lawrence 1999). You might elect to ignore still other user classes. They get what they get, but you don't specifically build the product to suit them. The remaining user classes are of roughly equal importance in defining the product's requirements.

Each user class will have its own set of requirements for the tasks that members of the class must perform. They might also have different nonfunctional requirements, such as usability, that will drive user interface design choices. Inexperienced or occasional users are concerned with how easy the system is to learn (or relearn) to use. These users like menus, graphical user interfaces, uncluttered screen displays, verbose prompts, wizards, and consistency with other applications they have used. Once users gain sufficient experience with the product, they become more concerned about ease of use and efficiency. They now value keyboard shortcuts, macros, customization options, toolbars, scripting facilities, and perhaps even a command-line interface instead of a graphical user interface.

Trap Don't overlook indirect or secondary user classes. They might not use your application directly, instead accessing its data or services through other applications or through reports. Your customer once removed is still your customer.

It might sound strange, but user classes need not be human beings. You can consider other applications or hardware components with which your system interacts as additional user classes. A fuel injection system would be a user class for the software embedded in an automobile's engine controller. The fuel injection system can't speak for itself, so the analyst must get the requirements for the fuel-injection control software from the engineer who designed the injection system.

Identify and characterize the different user classes for your product early in the project so that you can elicit requirements from representatives of each important class. A useful technique for this is called "Expand Then Contract" (Gottesdiener 2002). Begin by brainstorming as many user classes as you can think of. Don't be afraid if there are dozens at this stage, because you'll condense and categorize them later. It's important not to overlook a user class because that will come back to bite you later. Next, look for groups with similar needs that you can either combine or treat as a major user class with several subclasses. Try to pare the list down to no more than about 15 distinct user classes.

True Stories One company that developed a specialized product for about 65 corporate customers had regarded each company as a distinct user with unique needs. Grouping their customers into just six user classes greatly simplified their requirements challenges for future releases. Remember, you will have other important project stakeholders who won't actually use the product, so the user classes represent just a subset of the people who need to provide input on the requirements.

Document the user classes and their characteristics, responsibilities, and physical locations in the SRS. The project manager of the Chemical Tracking System discussed in earlier chapters identified the user classes and characteristics shown in Table 6-1. Include all pertinent information you have about each user class, such as its relative or absolute size and which classes are favored. This will help the team prioritize change requests and conduct impact assessments later on. Estimates of the volume and type of system transactions help the testers develop a usage profile for the system so that they can plan their verification activities.

Table 6-1: User Classes for the Chemical Tracking System

Chemists (favored)	Approximately 1000 chemists located in six buildings will use the system to request chemicals from vendors and from the chemical stockroom. Each chemist will use the system several times per day, mainly for requesting chemicals and tracking chemical containers into and out of the

	laboratory. The chemists need to search vendor catalogs for specific chemical structures imported from the tools they currently use for drawing chemical structures.
Buyers	About five buyers in the purchasing department process chemical requests that others submit. They place and track orders with external vendors. They know little about chemistry and need simple query facilities to search vendor catalogs. Buyers will not use the system's container-tracking features. Each buyer will use the system an average of 20 times per day.
Chemical Stockroom Staff	The chemical stockroom staff consists of six technicians and one supervisor who manage an inventory of more than 500,000 chemical containers. They will process requests from chemists to supply containers from three stockrooms, request new chemicals from vendors, and track the movement of all containers into and out of the stockrooms. They are the only users of the inventory-reporting feature. Because of their high transaction volume, the functions that are used only by the chemical stockroom staff must be automated and efficient.
Health and Safety Department Staff (favored)	The Health and Safety Department staff will use the system only to generate predefined quarterly reports that comply with federal and state chemical usage and disposal reporting regulations. The Health and Safety Department manager is likely to request changes in the reports several times per year as government regulations change. These report changes are of the highest priority and implementation will be time critical.

Consider building a catalog of user classes that recur across multiple applications. Defining user classes at the enterprise level lets you reuse those user class descriptions in future projects. The next system you build might serve the needs of some new user classes, but it probably will also be used by user classes from your earlier systems.

To bring the user classes to life, you might create a persona for each one, a description of a representative member of the user class, as follows (Cooper 1999):

> *Fred, 41, has been a chemist at Contoso Pharmaceuticals since he received his Ph.D. 14 years ago. He doesn't have much patience with computers. Fred usually works on two projects at a time in related chemical areas. His lab contains approximately 400 bottles of chemicals and gas cylinders. On an average day, he'll need four new chemicals from the stockroom. Two of these will be commercial chemicals in stock, one will need to be ordered, and one will come from the supply of proprietary Contoso chemical samples. On occasion, Fred will need a hazardous chemical that requires special training for safe handling. When he buys a chemical for the first time, Fred wants the material safety data sheet e-mailed to him automatically. Each year, Fred will generate about 10 new proprietary chemicals to go into the stockroom. Fred wants a report of his chemical usage for the previous month generated automatically and sent to him by e-mail so that he can monitor his chemical exposure.*

As you explore the chemists' requirements, think about Fred as the archetype of this user class and ask, "What would Fred need to do?"

Team LiB PREVIOUS NEXT
Team LiB PREVIOUS NEXT

Finding User Representatives

Every kind of project—including corporate information systems, commercial applications, package solutions, integrated systems, embedded systems, Internet applications, and contracted software—needs suitable user representatives to provide the voice of the customer. The user representatives should be involved throughout the development life cycle, not just in an isolated requirements phase at the beginning. Each user class needs to be represented.

It's easiest to gain access to actual users when you're developing applications for deployment within your own company. If you're developing commercial software, you might engage people from your current beta-testing or early-release sites to provide requirements input much earlier in the development process. (See "External Product Champions" later in this chapter). Consider setting up focus groups of current users of your products or your competitors' products.

True Stories One company asked a focus group to perform certain tasks with various digital cameras and computers. The results indicated that the company's camera software took too long to perform the most common operation because of a design decision to also accommodate less likely scenarios. The company made a change in their next camera's requirements that helped reduce customer complaints about speed.

Be sure that the focus group represents the kinds of users whose needs should drive your product development. Include both expert and less experienced customers. If your focus group represents only early adopters or blue-sky thinkers, you might end up with many sophisticated and technically difficult requirements that many of your target customers don't find interesting.

Figure 6-2 illustrates some typical communication pathways that connect the voice of the user to the ear of the developer. One study indicated that highly successful projects used more kinds of communication links and more direct links between developers and customers than did less successful projects (Keil and Carmel 1995). The most direct communication occurs when developers can talk to appropriate users themselves, which means that the developer is also performing the analyst role. As in the children's game "Telephone," intervening layers between the user and the developer increase the chance of miscommunication. For instance, requirements that come from the manager of the end users are less likely to accurately reflect the real user needs.

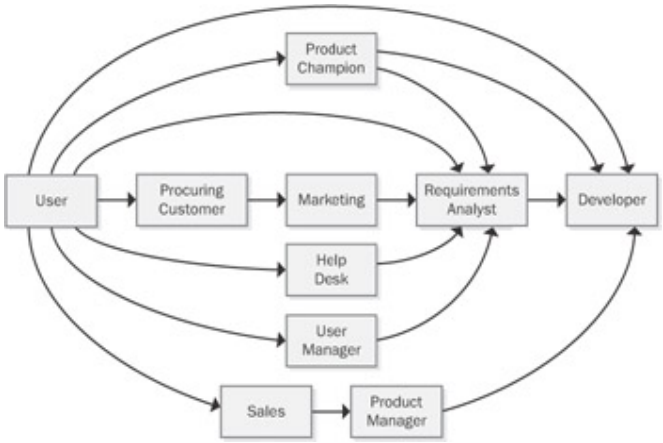


Figure 6-2: Some possible communication pathways between the user and the developer.

Some of these intervening layers add value, though, as when a skilled requirements analyst works with users or other participants to collect, evaluate, refine, and organize their input. Recognize the risks that you assume by using marketing staff, product managers, subject matter experts, or others as surrogates for the actual voice of the customer. Despite the obstacles to, and the cost of, acquiring optimum customer representation, your product and your customers will suffer if you don't talk to the people who can provide the best information.

The Product Champion

True Stories Many years ago I worked in a small software development group that supported the scientific research activities at a major corporation. Each of our projects included a few key members of our user community to provide the requirements. We called these people *product champions* (or *project champions*, although that term more often refers to the management sponsors of a project) (Wiegers 1996a). The product champion approach provides an effective way to structure the customer-development partnership.

Trap Watch out for user managers or for software developers who think they already understand the needs of the real system users without asking.

Each product champion serves as the primary interface between members of a single user class and the project's requirements analyst. Ideally, the champions will be actual users, not surrogates such as funding sponsors, procuring customers, marketing staff, user managers, or software developers pretending to be users. Product champions collect requirements from other members of the user classes they represent and reconcile inconsistencies. Requirements development is thus a shared responsibility of the analysts and selected customers, although the analyst writes the requirements documents.

The best product champions have a clear vision of the new system and are enthusiastic about it because they see how it will benefit them and their peers. The champions should be effective communicators who are respected by their colleagues. They need a thorough understanding of the application domain and the system's operating environment. Great product champions are in demand for other assignments on their principal job, so you'll have to build a persuasive case for why particular individuals are critical to project success. My team and I found that good product champions made a huge difference in our projects, so we happily offered them public reward and recognition for their contributions.

True Stories My software development team enjoyed an additional benefit from the product champion approach. On several projects, we had excellent champions who spoke out on our behalf with their colleagues when the customers wondered why the software wasn't done yet. "Don't worry about it," the champions told their peers and their managers. "I understand and agree with the software group's approach to software engineering. The time we're spending on requirements will help us get the system we really need and will save us time in the long run. Don't worry about it." Such collaboration helps break down the tension that often arises between customers and development teams.

The product champion approach works best if each champion is fully empowered to make binding decisions on behalf of the user class he represents. If a champion's decisions are routinely overruled by managers or by the software group, the champion's time and goodwill are being wasted. However, the champions must remember that they are not the sole customers. Problems arise when the individual filling this critical liaison role doesn't adequately communicate with his peers and presents only his own wishes and ideas.

External Product Champions

When developing commercial software, it can be difficult to find people to serve as product champions from outside your company. If you have a close working relationship with some major corporate customers, they might welcome the opportunity to participate in requirements elicitation. You might give external product champions economic incentives for their participation. Consider offering them discounts on products or paying for the time they spend working with you on requirements. You still face the challenge of how to avoid hearing only the champions' requirements and neglecting the needs of other customers. If you have a diverse customer base, first identify core requirements that are common to all customers. Then define additional requirements that are specific to individual customers, market segments, or user classes.

True Stories Commercial product development companies sometimes rely on internal subject matter experts

or outside consultants to serve as surrogates for actual users, who might be unknown or difficult to engage. Another alternative is to hire a suitable product champion who has the right background. One company that developed a retail point-of-sale and back-office system for a particular industry hired three store managers to serve as full-time product champions. As another example, my longtime family physician, Art, left his medical practice to become the voice-of-the-physician at a medical software company. Art's new employer believed that it was worth the expense to hire a doctor to help the company build software that other doctors would accept. A third company hired several former employees from one of their major customers. These people provided valuable domain expertise as well as insight into the politics of the customer organization.

Any time the product champion is a former or simulated user, watch out for disconnects between the champion's perceptions and the current needs of real users. Some problem domains change rapidly and some are more stable. The medical field is evolving quickly, whereas many corporate business processes persist in similar form for years. The essential question is whether the product champion, no matter what his background or current job, can accurately represent the needs of real users.

Product Champion Expectations

To help the product champion approach succeed, document what you expect your champions to do. These written expectations can help you build a case for specific individuals to fill this critical role. Table 6-2 identifies some activities that product champions might perform. Not every champion will do all of these; use this table as a starting point to negotiate each champion's responsibilities.

Table 6-2: Possible Product Champion Activities

Category	Activities
Planning	• Refine the scope and limitations of the product • Define interfaces to other systems • Evaluate impact of new system on business operations • Define a transition path from current applications
Requirements	• Collect requirements from other users • Develop usage scenarios and use cases • Resolve conflicts between proposed requirements • Define implementation priorities • Specify quality and performance requirements • Evaluate user interface prototypes
Validation and Verification	• Inspect requirements documents • Define user acceptance criteria • Develop test cases from usage scenarios • Provide test data sets • Perform beta testing

User Aids	• Write portions of user manuals and help text • Prepare training materials for tutorials • Present product demonstrations to peers
Change Management	• Evaluate and prioritize defect corrections • Evaluate and prioritize enhancement requests • Evaluate the impact of proposed requirements changes on users and business processes • Participate in making change decisions

Multiple Product Champions

One person can rarely describe the needs for all users of an application. The Chemical Tracking System had four major user classes, so it needed four product champions selected from the internal user community at Contoso Pharmaceuticals. Figure 6-3 illustrates how the project manager set up a team of analysts and product champions to collect the right requirements from the right sources. These champions were not assigned full-time, but each one spent several hours per week working on the project. Three analysts worked with the four product champions to elicit, analyze, and document their requirements. (The same analyst worked with two product champions because the Buyer and the Health and Safety Department user classes were small and had few requirements.) One analyst assembled all the input into a single SRS.

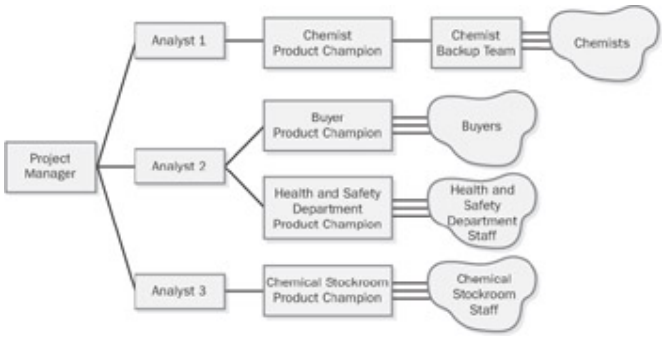


Figure 6-3: Product champion model for the Chemical Tracking System.

One person cannot adequately supply all the diverse requirements for a large user class such as the hundreds of chemists at Contoso. To help him, the product champion for the chemist user class assembled a backup team of five other chemists from other parts of the company. These chemists represented subclasses within the broad "chemist" user class. This hierarchical approach engaged additional users in requirements development while avoiding the expense of massive workshops or dozens of individual interviews. The chemist product champion, Don, always strove for consensus, but he willingly made the necessary decisions when agreement wasn't achieved so that the project could move ahead instead of deadlocking. No backup team was necessary when the user class was small enough or cohesive enough that one individual could adequately represent the group's needs.

The Voiceless User Class

True Stories A requirements analyst at Humongous Insurance was delighted that an influential user, Rebecca, agreed to serve as product champion for the new claim processing system. Rebecca had many ideas about the system features and user interface design. Thrilled to have the guidance of an expert, the

development team happily complied with her requests. After delivery, though, they were shocked to receive many complaints about how hard the system was to use.

Rebecca was a power user. She specified usability requirements that were great for experts, but the 90 percent of users who *weren't* experts found the system unintuitive and difficult to learn. The requirements analyst failed to recognize that the claim processing system had at least two user classes. The large group of non–power users was disenfranchised in the requirements and user interface design processes, and Humongous paid the price in an expensive redesign. The analyst should have engaged a second product champion to represent the large class of nonexpert users.

Selling the Product Champion Idea

Expect to encounter resistance when you propose the idea of having product champions on your projects. "The users are too busy." "Management wants to make the decisions." "They'll slow us down." "We can't afford it." "I don't know what I'm supposed to do as a product champion." Some users won't want to provide input on a system that they fear will make them change the way they work or might threaten their job with elimination. Managers are sometimes reluctant to delegate authority for requirements to ordinary users.

Separating business requirements from user requirements alleviates these discomforts. As an actual user, the product champion makes decisions at the user requirements level within the constraints that the project's business requirements impose. The management sponsor retains the authority to make decisions that affect the product vision, project scope, schedule, and cost. Documenting and negotiating the product champion's role and responsibilities give candidate champions a comfort level about what they're being asked to do.

If you encounter resistance, point out that insufficient user involvement is well established as a leading cause of software project failure (The Standish Group 1995). Remind the protesters of problems they've experienced on previous projects that trace back to inadequate user involvement. Every organization has horror stories of new systems that didn't satisfy user needs or failed to meet unstated usability or performance expectations. You can't afford to rebuild or discard systems that don't measure up because no one understood the requirements. Product champions mitigate this risk.

Product Champion Traps to Avoid

The product champion model has succeeded in many environments. It works only when the product champion understands and signs up for his responsibilities, has the authority to make decisions at the user requirements level, and has the time to do the job. Watch out for the following potential problems:

- Some managers override the decisions that a qualified and duly authorized product champion makes. Perhaps the manager has a wild new idea at the last minute, wants to change the project direction at a whim, or thinks he knows what the users need. This behavior often results in dissatisfied users, late deliveries as the developers chase the latest management notion, and frustrated product champions who feel that management doesn't trust them.

- A product champion who forgets that he is representing other customers and presents only his own requirements won't do a good job. He might be happy with the outcome, but others likely won't be.

- A product champion who lacks a clear mental image of the new system might defer important decisions to the analyst. If any idea that the analyst has is okay with the champion, the champion isn't much help.

- A senior user might nominate a less experienced user as champion because he doesn't have time to do the job himself. This can lead to backseat driving from the senior user who still wishes to strongly influence

the project's direction.

- **True Stories** Beware of users who purport to speak for a user class to which they do not belong. On Contoso's Chemical Tracking System, the product champion for the chemical stockroom staff also insisted on providing what she thought were the needs of the chemist user class. It was difficult to convince her that this wasn't her job, but the analyst didn't let her intimidate him. The project manager lined up a separate product champion for the chemists, and he did a great job of collecting, evaluating, and relaying that community's requirements.

Who Makes the Decisions?

Someone must resolve the conflicting requirements from different user classes, reconcile inconsistencies, and arbitrate the questions of scope that arise. Early in the project, determine who the decision makers will be for requirements issues. If it's not clear who is responsible for making these decisions or if the authorized individuals abdicate their responsibilities, the decisions will fall to the developers by default. This is a poor choice because developers usually don't have the necessary knowledge, experience, and perspective to make the best business decisions.

There's no globally correct answer to the question of who should make decisions about requirements on a software project. Analysts sometimes defer to the loudest voice they hear or to the person highest on the food chain. This is understandable, but it's not the best strategy. Decisions should be made as low in the organization's hierarchy as possible by people who are close to the issues and well informed about them. Every group should select an appropriate *decision rule*—an agreed-upon way to arrive at a decision—prior to encountering their first decision (Gottesdiener 2001). I favor participative or consultative decision making—that is, gathering the ideas and opinions of many stakeholders—over consensus decision making, which is gaining buy-in to the final decision from every affected stakeholder. Reaching a consensus is ideal, but you can't always hold up progress while waiting for every stakeholder to align on every issue.

Following are some requirements conflicts that can arise on projects with suggested ways to handle them. The project leaders need to determine who will decide what to do when such situations arise, who will make the call if agreement is not reached, and to whom significant issues must be escalated for ratifying the decisions made.

- If individual users disagree on requirements, the product champions decide. The essence of the product champion approach is that the champions are empowered and expected to resolve requirements conflicts that arise from those they represent.

- If different user classes or market segments present incompatible needs, go with the needs of the most favored user class or with the market segment that will have the greatest impact on the product's business success.

- Different corporate customers all might demand that the product be designed to satisfy their preferences. Again, use the business objectives for the project to determine which customers most strongly influence the project's success or failure.

- Requirements expressed by user managers sometimes conflict with those from the actual users in their departments. Although the user requirements must align with the business requirements, managers who aren't members of the user class should defer to the product champion who represents their users.

- When the product that the developers think they should build conflicts with what customers say they want, the customer should normally make the decision.

  Trap Don't justify doing whatever any customer demands because "The customer is always right." The customer is *not* always right. The customer always has a point, though, and the software team must understand and respect that point.
- **True Stories** A similar situation arises if marketing or product management presents requirements that conflict with what developers think they should build. As customer surrogates, marketing's input should carry more weight. Nevertheless, I've seen cases where marketing never said no to a customer request, no matter how infeasible or expensive. I've seen other cases in which marketing provided little input and the developers had to define the product and write the requirements themselves.

These negotiations don't always turn out the way the analyst thinks they should. Customers might reject all attempts to consider reasonable alternatives and other points of view. The team needs to decide who will be making decisions on the project's requirements before they confront these types of issues. Otherwise, indecision and the revisiting of previous decisions can cause the project to stall in endless wrangling.

Next Steps

- Relate Figure 6-1 to the way you hear the voice of the customer in your own environment. Do you encounter any problems with your current communication links? Identify the shortest and most appropriate communication paths that you can use to gather user requirements in the future.
- Identify the different user classes for your project. Which ones are favored? Which, if any, are disfavored? Who would make a good product champion for each important user class?
- Use Table 6-2 as a starting point to define the activities you would like your product champions to perform. Negotiate the specific contributions with each candidate product champion and his or her manager.
- Determine who the decision makers are for requirements issues on your project. How well does your current decision-making approach work? Where does it break down? Are the right people making decisions? If not, who should be doing it? Suggest processes that the decision makers should use for reaching agreement on the requirements issues.

PREVIOUS NEXT PREVIOUS NEXT

Chapter 7: Hearing the Voice of the Customer

Overview

"*Good morning, Maria. I'm Phil, the requirements analyst for the new employee information system we're going to build for you. Thanks for agreeing to be the product champion for this project. Your input will really help us a lot. So, can you tell me what you want?"*

"Hmmm, what do I want," mused Maria. "I hardly know where to start. It should be a lot faster than the old system. And you know how the old system crashes if some employee has a really long name and we have to call the help desk and ask them to enter the name? The new system should take long names without crashing. Also, a new law says we can't use Social Security numbers for employee IDs anymore, so we'll have to change all the

employee IDs when the new system goes in. The new IDs are going to be six-digit numbers. Oh, yes, it'd be great if I could get a report of how many hours of training each employee has had so far this year. And I also need to be able to change someone's name even if their marital status hasn't changed."

Phil dutifully wrote down everything Maria said, but his head was starting to spin. He wasn't sure what to do with all these bits of information, and he had no idea what to tell the developers. "Well," he thought, "if that's what Maria says she wants, I guess we'd better do it."

The heart of requirements engineering is *elicitation*, the process of identifying the needs and constraints of the various stakeholders for a software system. Elicitation focuses on discovering the user requirements, the middle level of the software requirements triad. (As described in Chapter 1, business requirements and functional requirements are the other two levels.) User requirements encompass the tasks that users need to accomplish with the system and the users' expectations of performance, usability, and other quality attributes. This chapter addresses the general principles of effective requirements elicitation.

The analyst needs a structure to organize the array of input obtained from requirements elicitation. Simply asking the users, "What do you want?" generates a mass of random information that leaves the analyst floundering. "What do you need to do?" is a much better question. Chapter 8, "Understanding User Requirements," describes how techniques such as use cases and event-response tables provide helpful organizing structures for user requirements.

The product of requirements development is a common understanding among the project stakeholders of the needs that are being addressed. Once the developers understand the needs, they can explore alternative solutions to address those needs. Elicitation participants should resist the temptation to design the system until they understand the problem. Otherwise, they can expect to do considerable design rework as the requirements become better defined. Emphasizing user tasks rather than user interfaces and focusing on root needs more than on expressed desires help keep the team from being sidetracked by prematurely specifying design details.

Begin by planning the project's requirements elicitation activities. Even a simple plan of action increases the chance of success and sets realistic expectations for the stakeholders. Only by gaining explicit commitment on resources, schedule, and deliverables can you avoid having people pulled off elicitation to fix bugs or do other work. Your plan should address the following items:

- Elicitation objectives (such as validating market data, exploring use cases, or developing a detailed set of functional requirements for the system)
- Elicitation strategies and processes (for example, some combination of surveys, workshops, customer visits, individual interviews, and other techniques, possibly using different approaches for different stakeholder groups)
- Products of elicitation efforts (perhaps a list of use cases, a detailed SRS, an analysis of survey results, or performance and quality attribute specifications)
- Schedule and resource estimates (identify both development and customer participants in the various elicitation activities, along with estimates of the effort and calendar time required)
- Elicitation risks (identify factors that could impede your ability to complete the elicitation activities as intended, estimate the severity of each risk, and decide how you can mitigate or control it)

Requirements Elicitation

Requirements elicitation is perhaps the most difficult, most critical, most error-prone, and most communication-intensive aspect of software development. Elicitation can succeed only through a collaborative partnership between customers and the development team, as described in Chapter 2. The analyst must create an environment conducive to a thorough exploration of the product being specified. To facilitate clear communication, use the vocabulary of the application domain instead of forcing customers to understand computer jargon. Capture significant application domain terms in a glossary, rather than assuming that all participants share the same definitions. Customers should understand that a discussion about possible functionality is not a commitment to include it in the product. Brainstorming and imagining the possibilities is a separate matter from analyzing priorities, feasibility, and the constraining realities. The stakeholders must focus and prioritize the blue-sky wish list to avoid defining an enormous project that never delivers anything useful.

Skill in conducting elicitation discussions comes with experience and builds on training in interviewing, group facilitation, conflict resolution, and similar activities. As an analyst, you must probe beneath the surface of the requirements the customers present to understand their true needs. Simply asking "why" several times can move the discussion from a presented solution to a solid understanding of the problem that needs to be solved. Ask open-ended questions to help you understand the users' current business processes and to see how the new system could improve their performance. Inquire about possible variations in the user tasks that the users might encounter and ways that other users might work with the system. Imagine yourself learning the user's job, or actually do the job under the user's direction. What tasks would you need to perform? What questions would you have? Another approach is to play the role of an apprentice learning from the master user. The user you are interviewing then guides the discussion and describes what he or she views as the important topics for discussion.

Probe around the exceptions. What could prevent the user from successfully completing a task? How should the system respond to various error conditions? Ask questions that begin with "What else could…," "What happens when…," "Would you ever need to…," "Where do you get…," "Why do you (or don't you)…," and "Does anyone ever…" Document the source of each requirement so that you can obtain further clarification if needed and trace development activities back to specific customer origins.

When you're working on a replacement project for a legacy system, ask the users, "What three things annoy you the most about the existing system?" This question helps get to the bottom of why a system is being replaced. It also surfaces expectations that the users hold for the follow-on system. As with any improvement activity, dissatisfaction with the current situation provides excellent fodder for the new and improved future state.

Try to bring to light any assumptions the customers might hold and to resolve conflicting assumptions. Read between the lines to identify features or characteristics the customers expect to be included without their having explicitly said so. Gause and Weinberg (1989) suggest using *context-free questions*, high-level and open-ended questions that elicit information about global characteristics of both the business problem and the potential solution. The customer's response to questions such as "What kind of precision is required in the product?" or "Can you help me understand why you don't agree with Miguel's reply?" can lead to insights that questions with standard yes/no or A/B/C answers do not.

Rather than simply transcribing what customers say, a creative analyst suggests ideas and alternatives to users during elicitation. Sometimes users don't realize the capabilities that developers can provide and they get excited when you suggest functionality that will make the system especially useful. When users truly can't express what they need, perhaps you can watch them work and suggest ways to automate portions of the job. Analysts can think outside the box that limits the creativity of people who are too close to the problem domain. Look for opportunities to reuse functionality that's already available in another system.

Interviews with individuals or groups of potential users are a traditional source of requirements input for both commercial products and information systems. (For guidance on how to conduct user interviews, see Beyer and Holtzblatt [1998], Wood and Silver [1995], and McGraw and Harbison [1997].) Engaging users in the elicitation process is a way to gain support and buy-in for the project. Try to understand the thought processes that led the users to present the requirements they state. Walk through the processes that users follow to make decisions about their work and extract the underlying logic. Flowcharts and decision trees are useful ways to depict these logical decision paths. Make sure that everyone understands why the system *must* perform certain functions. Proposed requirements sometimes reflect obsolete or ineffective business processes that should not be incorporated into a new system.

After each interview, document the items that the group discussed and ask the interviewees to review the list and make corrections. Early review is essential to successful requirements development because only those people who supplied the requirements can judge whether they were captured accurately. Use further discussions to resolve any inconsistencies and to fill in any blanks.

Elicitation Workshops

Requirements analysts frequently facilitate requirements elicitation workshops. Facilitated, collaborative group workshops are a highly effective technique for linking users and developers (Keil and Carmel 1995). The facilitator plays a critical role in planning the workshop, selecting participants, and guiding the participants to a successful outcome. When a team is getting started with new approaches to requirements elicitation, have an outside facilitator lead the initial workshops. This way the analyst can devote his full attention to the discussion. A scribe assists the facilitator by capturing the points that come up during the discussion.

According to one authority, "Facilitation is the art of leading people through processes toward agreed-upon objectives in a manner that encourages participation, ownership, and productivity from all involved" (Sibbet 1994). A definitive resource on facilitating requirements elicitation workshops is Ellen Gottesdiener's *Requirements by Collaboration* (2002). Gottesdiener describes a wealth of techniques and tools for workshop facilitation. Following are a few tips for conducting effective elicitation sessions.

Establish ground rules. The participants should agree on some basic operating principles for their workshops (Gottesdiener 2002). Examples include the following:

- Starting and ending meetings on time
- Returning from breaks promptly
- Holding only one conversation at a time
- Expecting everyone to contribute
- Focusing comments and criticisms on issues, not on individuals

Stay in scope. Use the vision and scope document to confirm whether proposed user requirements lie within the current project scope. Keep each workshop focused on the right level of abstraction for that day's objectives. Groups easily dive into distracting detail during requirements discussions. Those discussions consume time that the group should spend initially on developing a higher-level understanding of user requirements; the details will come later. The facilitator will have to reel in the elicitation participants periodically to keep them on topic.

Trap Avoid drilling down into excessive requirements detail prematurely. Recording great detail about what people already understand doesn't reduce the risks due to uncertainty in the requirements.

It's easy for users to begin itemizing the precise layout of items in a report or a dialog box before the team even agrees on the pertinent user task. Recording these details as requirements places unnecessary constraints on the subsequent design process. Detailed user interface design comes later, although preliminary screen sketches can be helpful at any point to illustrate how you might implement the requirements. Early feasibility exploration, which requires some amount of design, is a valuable risk-reduction technique.

Use parking lots to capture items for later consideration. An array of random but important information will surface in an elicitation workshop: quality attributes, business rules, user interface ideas, constraints, and more. Organize this information on flipcharts—parking lots—so that you don't lose it and to demonstrate respect for the participant who brought it up. Don't be distracted into discussing off-track details unless they turn out to be showstopper issues, such as a vital business rule that restricts the way a use case can work.

Timebox discussions. The facilitator might allocate a fixed period of time to each discussion topic, say, 30 minutes per use case during initial use case explorations. The discussion might need to be completed later, but timeboxing helps avoid the trap of spending far more time than intended on the first topic and neglecting the other planned topics entirely.

Keep the team small and include the right participants. Small groups can work much faster than larger teams. Elicitation workshops with more than five or six active participants can become mired in side trips down "rat holes," concurrent conversations, and bickering. Consider running multiple workshops in parallel to explore the requirements of different user classes. Workshop participants should include the product champion and other user representatives, perhaps a subject matter expert, a requirements analyst, and a developer. Knowledge, experience, and authority to make decisions are qualifications for participating in elicitation workshops.

Trap Watch out for off-topic discussions, such as design explorations, during elicitation sessions.

Keep everyone engaged. Sometimes participants will stop contributing to the discussion. These people might be frustrated because they see that the system is an accident waiting to happen. Perhaps their input isn't being taken seriously because other participants don't find their concerns interesting or don't want to disrupt the work that the group has completed so far. Perhaps the stakeholder who has withdrawn has a submissive personality and is deferring to more aggressive participants or a dominating analyst. The facilitator must read the body language, understand why someone has tuned out of the process, and try to bring the person back. That individual might hold an insightful perspective that could make an important contribution.

Too Many Cooks

True Stories Requirements elicitation workshops that involve too many participants can slow to a contentious crawl. My colleague Debbie was frustrated at the sluggish progress of the first use-case workshop she facilitated for a Web development project. The 12 participants held extended discussions of unnecessary details and couldn't agree on how each use case ought to work. The team's progress accelerated nicely when Debbie reduced the number of participants to six who represented the roles of analyst, customer, system architect, developer, and visual designer. The workshop lost some input by using the smaller team but the rate of progress more than compensated for that loss. The workshop participants should exchange information off-line with colleagues who don't attend and then bring the collected input to the workshops.

Classifying Customer Input

Don't expect your customers to present a succinct, complete, and well-organized list of their needs. Analysts must classify the myriad bits of requirements information they hear into various categories so that they can document and use it appropriately. Figure 7-1 illustrates nine such requirement categories.

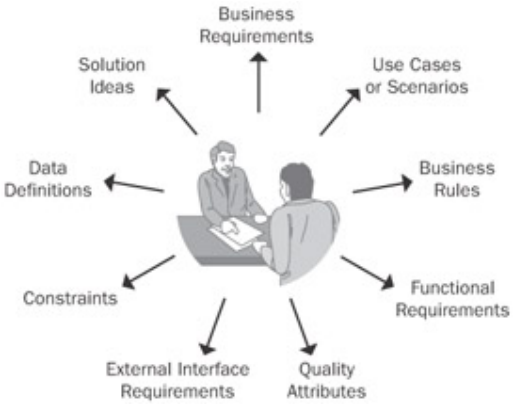


Figure 7-1: Classifying the voice of the customer.

Information that doesn't fit into one of these buckets might be one of the following:

- A requirement not related to the software development, such as the need to train users on the new system
- A project constraint, such as a cost or schedule restriction (as opposed to the design or implementation constraints described in this chapter)
- An assumption
- A data requirement, which can often be associated with some system functionality (you store data in a computer only so that you can get it out again later)
- Additional information of a historical, context-setting, or descriptive nature

The following discussion suggests some phrases to listen for that will help you in this classification process.

Business requirements. Anything that describes the financial, marketplace, or other business benefit that either customers or the developing organization wish to gain from the product is a business requirement. Listen for statements about the value that buyers or users of the software will receive, such as these:

- "Increase market share by X%."
- "Save $Y per year on electricity now wasted by inefficient units."
- "Save $Z per year in maintenance costs that are consumed by legacy system W."

Use cases or scenarios. General statements of user goals or business tasks that users need to perform are use cases; a single specific path through a use case is a usage scenario. Work with the customers to generalize specific scenarios into more abstract use cases. You can often glean use cases by asking users to describe their business workflow. Another way to discover use cases is to ask users to state the goals they have in mind when they sit down to work with the system. A user who says, "I need to <do something>" is probably describing a use case, as in the following examples:

- "I need to print a mailing label for a package."
- "I need to manage a queue of chemical samples waiting to be analyzed."
- "I need to calibrate the pump controller."

Business rules. When a customer says that only certain user classes can perform an activity under specific conditions, he might be describing a business rule. In the case of the Chemical Tracking System, such a business rule might be, "A chemist may order a chemical on the Level 1 hazard list only if his hazardous-chemical training is current." You might derive some software functional requirements to enforce the rules, such as making the training record database accessible to the Chemical Tracking System. As stated, though, business rules are not functional requirements. Following are some other phrases that suggest the user is describing a business rule:

- "Must comply with <some law or corporate policy>"
- "Must conform to <some standard>"
- "If <some condition is true>, then <something happens>"
- "Must be calculated according to <some formula>"

More Info See Chapter 9, "Playing by the Rules," for more examples of business rules.

Functional requirements. Functional requirements describe the observable behaviors the system will exhibit under certain conditions and the actions the system will let users take. Functional requirements derived from system requirements, user requirements, business rules, and other sources make up the bulk of the SRS. Here are some examples of functional requirements as you might hear them from users:

- "If the pressure exceeds 40.0 psi, the high pressure warning light should come on."
- "The user must be able to sort the project list in forward and reverse alphabetical order."
- "The system sends an e-mail to the Idea Coordinator whenever someone submits a new idea."

These statements illustrate how users typically present functional requirements, but they don't represent good ways to write functional requirements in an SRS. In the first case, we would replace *should* with *shall* to make it clear that illuminating the warning light is essential. The second example is a requirement of the user, not of the system. The requirement of the system is to permit the user to do the sorting.

More Info Chapter 10, "Documenting the Requirements," contains more guidance for writing good functional requirements.

Quality attributes. Statements that indicate how well the system performs some behavior or lets the user take some action are quality attributes. Listen for words that describe desirable system characteristics: fast, easy, intuitive, user-friendly, robust, reliable, secure, and efficient. You'll have to work with the users to understand precisely what they mean by these ambiguous and subjective terms and write clear, verifiable quality goals, as described in Chapter 12, "Beyond Functionality: Software Quality Attributes."

External interface requirements. Requirements in this class describe the connections between your system and the rest of the universe. The SRS should include sections for interfaces to users, hardware, and other software systems. Phrases that indicate that the customer is describing an external interface requirement include the following:

- "Must read signals from <some device>"
- "Must send messages to <some other system>"
- "Must be able to read (or write) files in <some format>"
- "Must control <some piece of hardware>"
- "User interface elements must conform to <some UI style standard>"

Constraints. Design and implementation constraints legitimately restrict the options available to the developer. Devices with embedded software often must respect physical constraints such as size, weight, and interface connections. Record the rationale behind each constraint so that all project participants know where it came from and respect its validity. Is it truly a restrictive limitation, as when a device must fit into an existing space? Or is it a desirable goal, such as a portable computer that weighs as little as possible?

Unnecessary constraints inhibit creating the best solution. Constraints also reduce your ability to use commercially available components as part of the solution. A constraint that specifies that a particular technology be used poses the risk of making a requirement obsolete or unattainable because of changes in the available technologies. Certain constraints can help achieve quality attribute goals. An example is to improve portability by using only the standard commands of a programming language, not permitting vendor-specific extensions. The following are examples of constraints that a customer might present:

- "Files submitted electronically may not exceed 10 MB in size."
- "The browser must use 128-bit encryption for all secure transactions."
- "The database must use the Framalam 10.2 run-time engine."

Other phrases that indicate the customer is describing a design or implementation constraint include these:

- "Must be written in <a specific programming language>"
- "Can't require more than <some amount of memory>"
- "Must operate identically to (or be consistent with) <some other system>"
- "Must use <a specific user interface control>"

As with functional requirements, the analyst shouldn't simply transcribe the user's statement of a constraint into the SRS. Weak words such as *identically* and *consistent* need to be clarified and the real constraint stated precisely enough for developers to act on the information. Ask why the constraint exists, verify its validity, and record the rationale for including the constraint as a requirement.

Data definitions. Whenever customers describe the format, data type, allowed values, or default value for a data item or the composition of a complex business data structure, they're presenting a data definition. "The ZIP code consists of five digits, followed by an optional hyphen and an optional four digits that default to 0000" is a data definition. Collect these in a *data dictionary*, a master reference that the team can use throughout the product's development and maintenance.

More Info See Chapter 10 for more information on data dictionaries.

Data definitions sometimes lead to functional requirements that the user community did not request directly.

What happens when a six-digit order number rolls over from 999,999? Developers need to know how the system will handle such data issues. Deferring data-related problems just makes them harder to solve in the future (remember Y2K?).

Solution ideas. Much of what users present as requirements fits in the category of solution ideas. Someone who describes a specific way to interact with the system to perform some action is presenting a suggested solution. The analyst needs to probe below the surface of a solution idea to get to the real requirement. For instance, functional requirements that deal with passwords are just one of several possible solutions for a security requirement.

Suppose a user says, "Then I select the state where I want to send the package from a drop-down list." The phrase *from a drop-down list* indicates that this is a solution idea. The prudent analyst will ask, "Why from a drop-down list?" If the user replies, "That just seemed like a good way to do it," the real requirement is something like, "The system shall permit the user to specify the state where he wants to send the package." However, maybe the user says, "I suggested a drop-down list because we do the same thing in several other places and I want it to be consistent. Also, it prevents the user from entering invalid data, and I thought we might be able to reuse some code." These are fine reasons to specify a specific solution. Recognize, though, that embedding a solution idea in a requirement imposes a design constraint on that requirement. It limits the requirement to being implemented in only one way. This isn't necessarily wrong or bad; just make sure the constraint is there for a good reason.

Some Cautions About Elicitation

Trying to amalgamate requirements input from dozens of users is difficult without using a structured organizing scheme, such as use cases. Collecting input from too few representatives or hearing the voice only of the loudest, most opinionated customer is also a problem. It can lead to overlooking requirements that are important to certain user classes or to including requirements that don't represent the needs of a majority of the users. The best balance involves a few product champions who have authority to speak for their respective user classes, with each champion backed up by several other representatives from that same user class.

During requirements elicitation, you might find that the project scope is improperly defined, being either too large or too small (Christel and Kang 1992). If the scope is too large, you'll collect more requirements than are needed to deliver adequate business and customer value and the elicitation process will drag on. If the project is scoped too small, customers will present needs that are clearly important yet just as clearly lie beyond the limited scope currently established for the project. The present scope could be too small to yield a satisfactory product. Eliciting user requirements therefore can lead to modifying the product vision or the project scope.

It's often stated that requirements are about *what* the system has to do, whereas *how* the solution will be implemented is the realm of design. Although attractively concise, this is an oversimplification. Requirements elicitation should indeed focus on the *what*, but there's a gray area—not a sharp line—between analysis and design. Hypothetical *how*s help to clarify and refine the understanding of what users need. Analysis models, screen sketches, and prototypes help to make the needs expressed during requirements elicitation more tangible and to reveal errors and omissions. Regard the models and screens generated during requirements development as conceptual suggestions to facilitate effective communication, not as constraints on the options available to the designer. Make it clear to users that these screens and prototypes are illustrative only, not necessarily the final design solution.

The need to do exploratory research sometimes throws a monkey wrench into the works. An idea or a suggestion arises, but extensive research is required to assess whether it should even be considered for possible

incorporation into the product. Treat these explorations of feasibility or value as project tasks in their own right, with objectives, goals, and requirements of their own. Prototyping is one way to explore such issues. If your project requires extensive research, use an incremental development approach to explore the requirements in small, low-risk portions.

Finding Missing Requirements

Missing requirements constitute the most common type of requirement defect (Jones 1997). They're hard to spot during reviews because they're invisible! The following techniques will help you detect previously undiscovered requirements.

Trap Watch out for the dreaded *analysis paralysis*, spending too much time on requirements elicitation in an attempt to avoid missing any requirements. You'll never discover them all up front.

- Decompose high-level requirements into enough detail to reveal exactly what is being requested. A vague, high-level requirement that leaves much to the reader's interpretation will lead to a gap between what the requester has in mind and what the developer builds. Imprecise, fuzzy terms to avoid include *support*, *enable*, *permit*, *process*, and *manage*.

- Make sure that all user classes have provided input. Make sure that each use case has at least one identified actor.

- Trace system requirements, use cases, event-response lists, and business rules into their detailed functional requirements to make sure that the analyst derived all the necessary functionality.

- Check boundary values for missing requirements. Suppose that one requirement states, "If the price of the order is less than $100, the shipping charge is $5.95" and another says, "If the price of the order is more than $100, the shipping charge is 5 percent of the total order price." But what's the shipping charge for an order with a price of exactly $100? It's not specified, so a requirement is missing.

- Represent requirements information in multiple ways. It's difficult to read a mass of text and notice that something isn't there. An analysis model visually represents requirements at a high level of abstraction—the forest, not the trees. You might study a model and realize that there should be an arrow from one box to another; that missing arrow represents a missing requirement. This kind of error is much easier to spot in a picture than in a long list of textual requirements that all blur together. Analysis models are described in Chapter 11, "A Picture Is Worth 1024 Words."

- Sets of requirements with complex Boolean logic (ANDs, ORs, and NOTs) often are incomplete. If a combination of logical conditions has no corresponding functional requirement, the developer has to deduce what the system should do or chase down an answer. Represent complex logic using decision tables or decision trees to make sure you've covered all the possible situations, as described in Chapter 11.

A rigorous way to search for missing requirements is to create a CRUD matrix. *CRUD* stands for *Create*, *Read*, *Update*, and *Delete*. A CRUD matrix correlates system actions with data entities (individual data items or aggregates of data items) to make sure that you know where and how each data item is created, read, updated, and deleted. Some people add an *L* to the matrix to indicate that the data item appears as a *List* selection (Ferdinandi 2002). Depending on the requirements analysis approaches you are using, you can examine various types of correlations, including the following:

- Data entities and system events (Robertson and Robertson 1999)
- Data entities and user tasks or use cases (Lauesen 2002)
- Object classes and system events (Ferdinandi 2002)
- Object classes and use cases (Armour and Miller 2001)

Figure 7-2 illustrates an entity/use case CRUDL matrix for a portion of the Chemical Tracking System. Each cell indicates how the use case in the leftmost column uses each data entity shown in the other columns. The use case can **C**reate, **R**ead, **U**pdate, **D**elete, or **L**ist the entity. After creating a CRUDL matrix, see whether any of these five letters do not appear in any of the cells in a column. If a business object is updated but never created, where does it come from? Notice that none of the cells under the column labeled Requester (the person who places an order for a chemical) contains a *D*. That is, none of the use cases in Figure 7-2 can delete a Requester from the list of people who have ordered chemicals. There are three possible interpretations:

Use Case \ Entity	Order	Chemical	Requester	Vendor Catalog
Place Order	C	R	R	R, L
Change Order	U, D		R	R, L
Manage Chemical Inventory		C, U, D		
Report on Orders	R	R, L	R, L	
Edit Requesters			C, U, L	

Figure 7-2: Sample CRUDL matrix for the Chemical Tracking System.

1. Deleting a Requester is not an expected function of the Chemical Tracking System.
2. We are missing a use case that deletes a Requester.
3. The "Edit Requesters" use case is incorrect. It's supposed to permit the user to delete a Requester, but that functionality is missing from the use case at present.

We don't know which interpretation is correct, but the CRUDL analysis is a powerful way to detect missing requirements.

PREVIOUS NEXT
PREVIOUS NEXT

How Do You Know When You're Done?

No simple signal will indicate when you've completed requirements elicitation. As people muse in the shower each morning and talk with their colleagues, they'll generate ideas for additional requirements. You'll never be completely done, but the following cues suggest that you're reaching the point of diminishing returns on requirements elicitation:

- If the users can't think of any more use cases, perhaps you're done. Users tend to identify use cases in sequence of decreasing importance.
- If users propose new use cases but you've already derived the associated functional requirements from other use cases, perhaps you're done. These "new" use cases might really be alternative courses for other use cases that you've already captured.

- If users repeat issues that they already covered in previous discussions, perhaps you're done.
- If suggested new features, user requirements, or functional requirements are all out of scope, perhaps you're done.
- If proposed new requirements are all low priority, perhaps you're done.
- If the users are proposing capabilities that might be included "sometime in the lifetime of the product" rather than "in the specific product we're talking about right now," perhaps you're done, at least with the requirements for the next release.

Another way to determine whether you're done is to create a checklist of common functional areas to consider for your projects. Examples include error logging, backup and restore, access security, reporting, printing, preview capabilities, and configuring user preferences. Periodically compare this list with the functions you have already specified. If you don't find gaps, perhaps you're done.

Despite your best efforts to discover *all* the requirements, you won't, so expect to make changes as construction proceeds. Remember, your goal is to make the requirements good enough to let construction proceed at an acceptable level of risk.

Next Steps

- Think about missing requirements that were discovered late on your last project. Why were they overlooked during elicitation? How could you have discovered each of these requirements earlier?
- Select a portion of any documented voice-of-the-customer input on your project or a section from the SRS. Classify every item in that requirements fragment into the categories shown in Figure 7-1: business requirements, use cases or scenarios, business rules, functional requirements, quality attributes, external interface requirements, constraints, data definitions, and solution ideas. If you discover items that are classified incorrectly, move them to the correct place in the requirements documentation.
- List the requirements elicitation methods used on your current project. Which ones worked well? Why? Which ones did not work so well? Why not? Identify elicitation techniques that you think would work better and decide how you'd apply them next time. Identify any barriers you might encounter to making those techniques work and brainstorm ways to overcome those barriers.

Chapter 8: Understanding User Requirements

Overview

The Chemical Tracking System project was holding its first requirements elicitation workshop to learn what chemists would need to do with the system. The participants included the requirements analyst, Lori, who was also facilitating; the product champion for the chemists, Tim; two other chemist representatives, Sandy and Peter; and the lead developer, Helen. Lori opened the workshop by thanking the participants for attending and then got right down to business.

"Tim, Sandy, and Peter have identified about 15 use cases that chemists would need to perform using the

Chemical Tracking System," she told the group. "In these workshops we'll explore these use cases so that we know what functionality to build into the system. You identified the use case called 'Request a Chemical' as top priority and Tim already wrote a brief description for it, so let's begin there. Tim, how do you imagine requesting a chemical with the system?"

"First," said Tim, "you should know that only people who have been authorized by their lab managers are allowed to request chemicals."

"Okay, that sounds like a business rule," Lori replied. "I'll start a flipchart for business rules because we'll probably find others. It looks like we'll have to verify that the user is on the approved list." She also wrote "user is identified" and "user is authorized to request chemicals" in the "Preconditions" section of the flipchart she had already prepared for the "Request a Chemical" use case. "Are there any other prerequisites before a user can create a request?"

"Before I request a chemical from a vendor, I want to see if it's already available in the stockroom," Sandy said. "The current inventory database should be online when I start to create a request so I don't waste my time."

Lori added that item to the preconditions. Over the next 30 minutes, she guided the group through a discussion of how they envisioned creating a request for a new chemical. She used several flipcharts to collect information about preconditions, postconditions, and the steps in the interaction between the user and the Chemical Tracking System. Lori asked how the use case would be different if the user were requesting a chemical from a vendor rather than from the stockroom. She asked what could go wrong and how the system should handle each error condition. After a half hour, the group had a solid handle on how a user would request a chemical. They moved on to the next use case.

People employ software systems to accomplish useful goals, and the software industry is exhibiting an encouraging trend toward designing software to enhance usability (Constantine and Lockwood 1999; Nielsen 2000). A necessary prerequisite to designing software for use is knowing what the users intend to do with it.

For many years, analysts have employed usage scenarios to elicit user requirements (McGraw and Harbison 1997). A *scenario* is a description of a single instance of usage of the system. Ivar Jacobson and colleagues (1992), Larry Constantine and Lucy Lockwood (1999), Alistair Cockburn (2001), and others have formalized the usage-centered perspective into the use-case approach to requirements elicitation and modeling. Use cases work well for developing the requirements for business applications, Web sites, services that one system provides to another, and systems that let a user control a piece of hardware. Applications such as batch processes, computationally intensive systems, and data warehousing applications might have just a few simple use cases. The complexity of these applications lies in the computations performed or the reports generated, not in the user-system interaction. A requirements technique often used for real-time systems is to list the external events to which the system must react and the corresponding system responses. This chapter describes the application of both use cases and event-response tables to capture user requirements.

Trap Don't try to force every requirement you encounter to fit into a use case. Use cases can reveal most—but probably not all—of the functional requirements.

PREVIOUS NEXT
PREVIOUS NEXT

The Use-Case Approach

A *use case* describes a sequence of interactions between a system and an external actor. An *actor* is a person, another software system, or a hardware device that interacts with the system to achieve a useful goal (Cockburn 2001). Another name for actor is *user role*, because actors are roles that the members of one or

more user classes can perform with respect to the system (Constantine and Lockwood 1999). For example, the Chemical Tracking System's use case called "Request a Chemical" involves an actor named *Requester*. There is no Chemical Tracking System user class named Requester. Both chemists and members of the chemical stockroom staff may request chemicals, so members of either user class may perform the Requester role.

Use cases emerged from the object-oriented development world. However, projects that follow any development approach can use them because the user doesn't care how the software is built. Use cases are at the center of the widely used Unified Software Development Process (Jacobson, Booch, and Rumbaugh 1999).

Use cases shift the perspective of requirements development to discussing what *users* need to accomplish, in contrast to the traditional elicitation approach of asking users what they want the *system* to do. The objective of the use-case approach is to describe all tasks that users will need to perform with the system. The stakeholders ensure that each use case lies within the defined project scope before accepting it into the requirements baseline. In theory, the resulting set of use cases will encompass all the desired system functionality. In practice, you're unlikely to reach complete closure, but use cases will bring you closer than any other elicitation technique I have used.

Use-case diagrams provide a high-level visual representation of the user requirements. Figure 8-1 shows a partial use-case diagram for the Chemical Tracking System, using the UML notation (Booch, Rumbaugh, and Jacobson 1999; Armour and Miller 2001). The box represents the system boundary. Lines from each actor (stick figure) connect to the use cases (ovals) with which the actor interacts. Note the resemblance of this use-case diagram to the context diagram in Figure 5-3. In the use-case diagram, the box separates some top-level internals of the system—the use cases—from the external actors. The context diagram also depicts objects that lie outside the system, but it provides no visibility into the system internals.

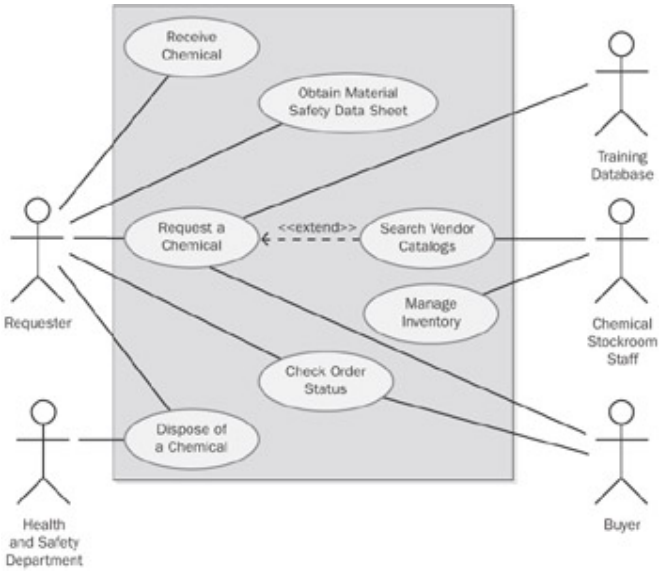


Figure 8-1: Partial use-case diagram for the Chemical Tracking System.

Use Cases and Usage Scenarios

A *use case* is a discrete, stand-alone activity that an actor can perform to achieve some outcome of value. A single use case might encompass a number of similar tasks having a common goal. A use case is therefore a collection of related usage scenarios, and a scenario is a specific instance of a use case. When exploring user requirements, you can start with abstract use cases and develop concrete usage scenarios, or you can generalize from a specific scenario to the broader use case. Later in this chapter we'll see a detailed template for documenting use cases. The essential elements of a use-case description are the following:

- A unique identifier

- A name that succinctly states the user task in the form of "verb + object," such as "Place an Order"
- A short textual description written in natural language
- A list of preconditions that must be satisfied before the use case can begin
- Postconditions that describe the state of the system after the use case is successfully completed
- A numbered list of steps that shows the sequence of dialog steps or interactions between the actor and the system that leads from the preconditions to the postconditions

One scenario is identified as the *normal course* of events for the use case; it is also called the main course, basic course, normal flow, primary scenario, main success scenario, and happy path. The normal course for the "Request a Chemical" use case is to request a chemical that's available in the chemical stockroom.

Other valid scenarios within the use case are described as *alternative courses* or *secondary scenarios* (Schneider and Winters 1998). Alternative courses also result in successful task completion and satisfy the use case's postconditions. However, they represent variations in the specifics of the task or in the dialog sequence used to accomplish the task. The normal course can branch off into an alternative course at some decision point in the dialog sequence and rejoin the normal course later. Although most use cases can be described in simple prose, a flowchart or a UML activity diagram is a useful way to visually represent the logic flow in a complex use case, as shown in Figure 8-2. Flowcharts and activity diagrams show the decision points and conditions that cause a branch from the main course into an alternative course.

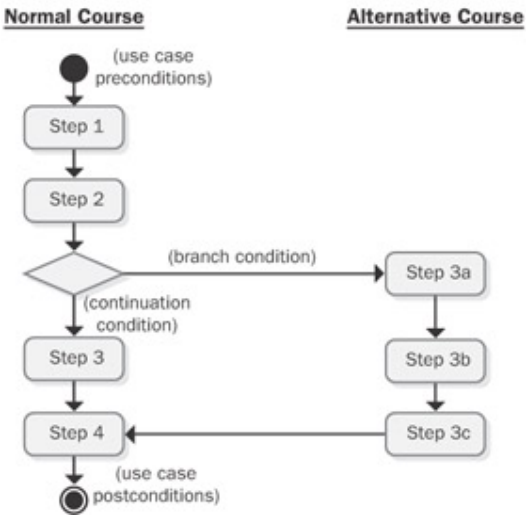


Figure 8-2: UML activity diagram illustrating the dialog flow in the normal and alternative courses of a use case.

An alternative course for the Request a Chemical use case is "Request a Chemical from a Vendor." The actor's ultimate goal—request a chemical—is the same in both situations, so these are two scenarios within the same use case. Some of the steps in an alternative course will be the same as those in the normal course, but certain unique actions are needed to accomplish the alternative path. In this alternative course, the user can search vendor catalogs for a desired chemical. If an alternative course is itself a stand-alone use case, you can *extend* the normal course by inserting that separate use case into the normal flow (Armour and Miller 2001). The use-case diagram in Figure 8-1 illustrates the extends relationship. The "Search Vendor Catalogs" use case extends the "Request a Chemical" use case. In addition, the chemical stockroom staff use the stand-alone "Search Vendor Catalogs" use case by itself.

Sometimes several use cases share a common set of steps. To avoid duplicating these steps in each such use

case, define a separate use case that contains the shared functionality and indicate that the other use cases *include* that sub use case. This is analogous to calling a common subroutine in a computer program.

As an example, consider an accounting software package. Two use cases are "Pay Bill" and "Reconcile Credit Card," both of which involve the user writing a check to make the payment. You can create a separate use case called "Write Check" that contains the common steps involved in writing the check. The two transaction use cases both include the "Write Check" use case, as shown in Figure 8-3.

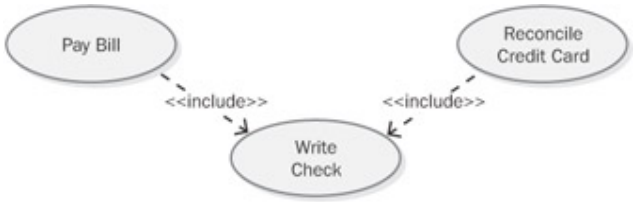


Figure 8-3: An example of the use-case *includes* relationship for an accounting application.

Trap Don't have protracted debates over when, how, and whether to use the *extends* and *includes* relationships. Factoring common steps into an included use case is the most typical usage.

Conditions that prevent a task from succeeding are called *exceptions*. One exception for the "Request a Chemical" use case is "Chemical is not commercially available." If you don't specify exception handling during elicitation, there are two possible outcomes:

1. The developers will make their best guesses at how to deal with the exceptions.

2. The system will fail when a user hits the error condition because no one thought about it.

It's a safe bet that system crashes aren't on the user's list of requirements.

Exceptions are sometimes regarded as a type of alternative course (Cockburn 2001), but it's useful to separate them. You won't necessarily implement every alternative course that you identify for a use case, and you might defer some to later releases. However, you *must* implement the exceptions that can prevent the scenarios that you do implement from succeeding. Anyone who has done computer programming knows that exceptions often generate the bulk of the coding effort. Many of the defects in a finished product reside in exception handlers (or missing exception handlers!). Specifying exception conditions during requirements elicitation is a way to build robust software products.

In many systems, the user can chain together a sequence of use cases into a "macro" use case that describes a larger task. Some use cases for a commercial Web site might be "Search Catalog," "Add Item to Shopping Cart," and "Pay for Items in Shopping Cart." If you can perform each of these activities independently, they are individual use cases. You might also be able to perform all three tasks in a row as a single large use case called "Buy Product," as shown in Figure 8-4. To make this work, each use case must leave the system in a state that enables the user to commence the next use case immediately. That is, the postconditions of one use case must satisfy the preconditions of the next one in the sequence. Similarly, in a transaction-processing application such as an ATM, each use case must leave the system in a state that permits the next transaction to begin. The preconditions and postconditions of each transaction use case must align.

Figure 8-4: Preconditions and postconditions define the boundaries of the individual use cases that can be

chained together to perform a larger task.

Identifying Use Cases

You can identify use cases in several ways (Ham 1998; Larman 1998):

- Identify the actors first, and then identify the business processes in which each participates.
- Identify the external events to which the system must respond, and then relate these events to participating actors and specific use cases.
- Express business processes in terms of specific scenarios, generalize the scenarios into use cases, and identify the actors involved in each use case.
- Derive likely use cases from existing functional requirements. If some requirements don't trace to any use case, consider whether you really need them.

The Chemical Tracking System followed the first approach. The analysts facilitated a series of two- to three-hour use-case elicitation workshops, which were held about twice a week. Members of the various user classes participated in separate, parallel workshops. This worked well because only a few use cases were common to multiple user classes. Each workshop included the user class's product champion, other selected user representatives, and a developer. Participating in the elicitation workshops gives developers early insight into the product they will be expected to build. Developers also serve as the voice of reality when infeasible requirements are suggested.

Prior to beginning the workshops, each analyst asked the users to think of tasks they would need to perform with the new system. Each of these tasks became a candidate use case. A few proposed use cases were judged to be out of scope and weren't pursued. As the group explored the use cases in the workshops, they found that some of them were related scenarios that could be consolidated into a single abstract use case. The group also discovered additional use cases beyond those in the initial set.

Some users proposed use cases that were not phrased as a task, such as "Material Safety Data Sheet." A use case's name should indicate the goal the user wants to accomplish, so you need a verb. Does the user want to view, print, order, e-mail, revise, delete, or create a material safety data sheet? Sometimes a suggested use case was just one step the actor would perform as part of the use case, such as "scan bar code." The analyst needs to learn what objective the user has in mind that involves scanning a bar code. The analyst might ask, "When you scan the bar code on the chemical container, what are you trying to accomplish?" Suppose the reply is, "I need to scan the bar code so that I can log the chemical into my laboratory." The real use case, therefore, is something like "Log Chemical Into Lab." Scanning the bar code label is just one step in the interaction between the actor and the system that logs the chemical into the lab.

Users typically identify their most important use cases first, so the discovery sequence gives some clues as to priority. Another prioritization approach is to write a short description of each candidate use case as it is proposed. Prioritize these candidate use cases and do an initial allocation of use cases to specific product releases. Specify the details for the highest priority use cases first so that the developers can begin implementing them as soon as possible.

Documenting Use Cases

At this stage of the exploration, the participants should be thinking of essential use cases. Constantine and Lockwood (1999) define an *essential use case* as "...a simplified, generalized, abstract, technology-free and implementation-independent description of one task or interaction...that embodies the purpose or intentions underlying the interaction." That is, the focus should be on the goal the user is trying to accomplish and the

system's responsibilities in meeting that goal. Essential use cases are at a higher level of abstraction than *concrete use cases*, which discuss specific actions the user takes to interact with the system. To illustrate the difference, consider the following two ways to describe how a user might initiate a use case to request a chemical:

Concrete Enter the chemical ID number.

Essential Specify the desired chemical.

The phrasing at the essential level allows many ways to accomplish the user's intention of indicating the chemical to be requested: enter a chemical ID number, import a chemical structure from a file, draw the structure on the screen with the mouse, select a chemical from a list, and others. Proceeding too quickly into specific interaction details begins to constrain the thinking of the use-case workshop participants. The independence from implementation also makes essential use cases more reusable than concrete use cases.

The participants in the Chemical Tracking System workshops began each discussion by identifying the actor who would benefit from the use case and writing a short description of the use case. Then they defined the preconditions and postconditions, which are the boundaries of the use case; all use-case steps take place between these boundaries. Estimating the frequency of use provided an early indicator of concurrent usage and capacity requirements. Next, the analyst asked the participants how they envisioned interacting with the system to perform the task. The resulting sequence of actor actions and system responses became the normal course. Numbering the steps made the sequence clear. Although each participant had a different mental image of the future user interface and specific interaction mechanisms, the group was able to reach a common vision of the essential steps in the actor-system dialog.

Staying in Bounds

True Stories While reviewing a use case that contained eight steps, I realized that the postconditions were satisfied after step 5. Steps 6, 7, and 8 therefore were unnecessary, being outside the boundary of the use case. Similarly, a use case's preconditions must be satisfied prior to commencing step 1 of the use case. When you review a use-case description, make sure that its pre- and postconditions properly frame it.

The analyst captured the individual actor actions and system responses on sticky notes, which he placed on a flipchart sheet. Another way to conduct such a workshop is to project a use-case template onto a large screen from a computer and complete the template during the discussion, although this might slow the discussion down.

The elicitation team developed similar dialogs for the alternative courses and exceptions they identified. A user who says, "The default should be…" is describing the normal course of the use case. A phrase such as "The user should also be able to…" suggests an alternative course. Many exception conditions were discovered when the analyst asked questions similar to "What should happen if the database isn't online at that moment?" or "What if the chemical is not commercially available?" The workshop is also a good time to discuss the user's expectations of quality, such as response times, system availability and reliability, user interface design constraints, and security requirements.

After the workshop participants described each use case and no one proposed additional variations, exceptions, or special requirements, they moved on to another use case. They didn't try to cover all the use cases in one marathon workshop or to pin down every detail of every use case they discussed. Instead, they planned to explore the use cases in increments, and then review and refine them into further detail iteratively.

Figure 8-5 shows the sequence of events for developing the use cases. Following the workshop, the analyst wrote a detailed description of each use case like the description in Figure 8-6. There are two ways to represent

the actor-system dialog steps, which is the heart of the use case. Figure 8-6 shows the dialog as a numbered list of steps, indicating which entity (the system or a specific actor) performs each step. The same notation is used to describe alternative courses and exceptions, which also show the step in the normal course at which the alternative branches off or where the exception could take place. Another technique is to present the dialog in a two-column table, as shown in Figure 8-7 (Wirfs-Brock 1993). The actor actions are shown on the left and the system responses on the right. The numbers indicate the sequence of steps in the dialog. This scheme works well when only a single actor is interacting with the system. To improve the readability, you can write each actor action or system response on a separate line so that the alternating sequence is clear, as shown in Figure 8-8.

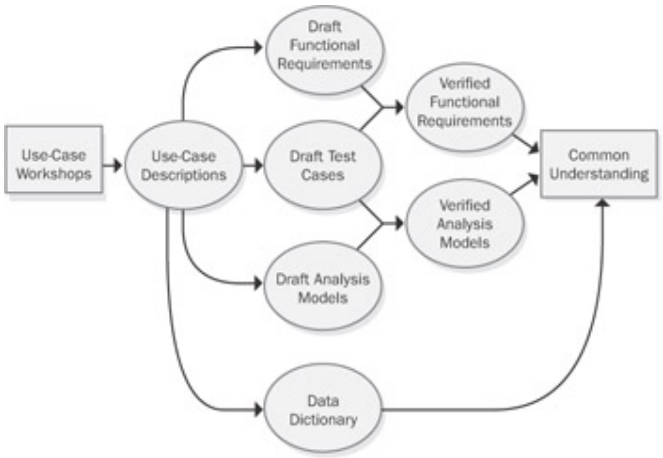


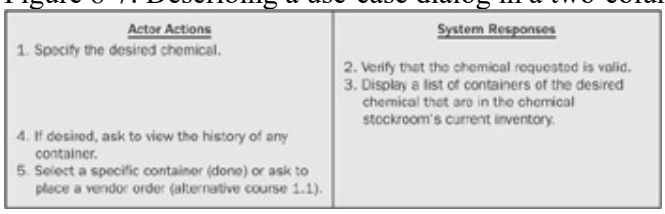
Figure 8-5: Use-case elicitation approach.

Use Case ID	**UC-1**	Use Case Name	**Request a Chemical**
Created By	Tim	Last Updated By	Janice
Date Created	12/4/02	Date Last Updated	12/27/02
Actors	Requester		
Description	The Requester specifies the desired chemical to request by entering its name or chemical ID number or by importing its structure from a chemical drawing tool. The system satisfies the request either by offering the Requester a new or used container of the chemical from the chemical stockroom or by letting the Requester create a request to order from an outside vendor.		
Preconditons	1. User's identity has been authenticated. 2. User is authorized to request chemicals. 3. Chemical inventory database is online.		
Postconditions	1. Request is stored in the Chemical Tracking System. 2. Request was e-mailed to the chemical stockroom or to a Buyer.		
Normal Course	**1.0 Request a Chemical from the Chemical Stockroom** 1. Requester specifies the desired chemical. 2. System verifies that the chemical is valid. 3. System lists containers of the desired chemical that are in the chemical stockroom. 4. Requester has the option to View Container History for any container. 5. Requester selects a specific container or asks to place a vendor order (alternative course 1.1). 6. Requester enters other information to complete the request. 7. System stores request and e-mails it to chemical stockroom.		
Alternative Courses	**1.1 Request a Chemical from a Vendor (branch after step 5)** 1. Requester searches vendor catalogs for a chemical. 2. System displays a list of vendors with available container sizes, grades, and prices. 3. Requester selects a vendor, container size, grade, and number of containers. 4. Requester enters other information to complete the request. 5. System stores request and e-mails it to Buyer.		
Exceptions	**1.0.E.1 Chemical is not valid (at step 2)** 1. System displays message: "That chemical does not exist." 2. System asks Requester whether he wishes to request another chemical or to exit. 3a. Requester asks to request another chemical. 4a. System starts Normal Course over. 3b. Requester asks to exit. 4b. System terminates use case. **1.0.E.2 Chemical is not commercially available (at step 5)** 1. System displays message: "No vendors for that chemical." 2. System asks Requester whether he wishes to request another chemical or to exit. 3a. Requester asks to request another chemical. 4a. System starts Normal Course over. 3b. Requester asks to exit. 4b. System terminates use case.		
Includes	**UC-22** View Container History		
Priority	High		
Frequency of Use	Approximately five times per week by each chemist, 100 times per week by each member of chemical stockroom staff.		
Business Rules	**BR-28** Only staff who are authorized by their laboratory managers may request chemicals.		
Special Requirements	1. The system must be able to import a chemical structure in the standard encoded form from any of the supported chemical drawing packages.		
Assumptions	1. Imported chemical structures are assumed to be valid.		
Notes and Issues	1. Tim will find out whether management approval is needed to request a chemical on the Level 1 hazard list. Due date 1/4/03.		

Figure 8-6: Partial description of the "Request a Chemical" use case.

Actor Actions	System Responses
1. Specify the desired chemical. 4. If desired, ask to view the history of any container. 5. Select a specific container (done) or ask to place a vendor order (alternative course 1.1).	2. Verify that the chemical requested is valid. 3. Display a list of containers of the desired chemical that are in the chemical stockroom's current inventory.

Figure 8-7: Describing a use-case dialog in a two-column format.



Actor Actions	System Responses
1. Specify the desired chemical. 4. If desired, ask to view the history of any container. 5. Select a specific container (done) or ask to place a vendor order (alternative course 1.1).	2. Verify that the chemical requested is valid. 3. Display a list of containers of the desired chemical that are in the chemical stockroom's current inventory.

Figure 8-8: Alternative layout for describing a use-case dialog in a two-column format.

Use cases often involve some additional information or requirements that do not fit within any of the template sections. Use the "Special Requirements" section to record pertinent quality attributes, performance requirements, and similar information. Also note any information that might not be visible to the users, such as the need for one system to communicate behind the scenes with another to complete the use case.

You don't always need a comprehensive use-case description. Alistair Cockburn (2001) describes *casual* and *fully dressed* use-case templates. Figure 8-6 illustrates a fully dressed use case. A casual use case is simply a textual narrative of the user goal and interactions with the system, perhaps just the "Description" section from Figure 8-6. The user stories that serve as requirements in Extreme Programming are essentially casual use cases typically written on index cards (Jeffries, Anderson, and Hendrickson 2001). Fully dressed use-case descriptions are valuable when

- Your user representatives are not closely engaged with the development team throughout the project.
- The application is complex and has a high risk associated with system failures.
- The use-case descriptions are the lowest level of requirements detail that the developers will receive.
- You intend to develop comprehensive test cases based on the user requirements.
- Collaborating remote teams need a detailed, shared group memory.

Trap Instead of being dogmatic about how much detail to include in a use case, remember your goal: to understand the user's objectives with the system well enough to enable developers to proceed at low risk of having to do rework.

The process in Figure 8-5 shows that after each workshop, the analysts on the Chemical Tracking System derived software functional requirements from the use-case descriptions. (For more about this, see the next section in this chapter, "Use Cases and Functional Requirements.") They also drew analysis models for some of the complex use cases, such as a state-transition diagram that showed all possible chemical request statuses and the permitted status changes.

More Info Chapter 11, "A Picture is Worth 1024 Words," illustrates several analysis models for the Chemical Tracking System.

A day or two after each workshop the analyst gave the use-case descriptions and functional requirements to the workshop participants, who reviewed them prior to the next workshop. These informal reviews revealed many errors: previously undiscovered alternative courses, new exceptions, incorrect functional requirements, and missing dialog steps. Leave at least one day between successive workshops. The mental relaxation that comes after a day or two away from an intellectually intensive activity allows people to examine their earlier work

from a fresh perspective. One analyst who held workshops every day learned that the participants had difficulty finding errors in the documents they reviewed because the information was too fresh in their minds. They mentally recited the workshop discussion that had just taken place and didn't see the errors.

Trap Don't wait until requirements elicitation is complete to solicit review feedback from users, developers, and other stakeholders.

Early in the requirements development process, the Chemical Tracking System's test lead created conceptual test cases, independent of implementation specifics, from the use cases (Collard 1999). These test cases helped the team share a clear understanding of how the system should behave in specific usage scenarios. The test cases let the analysts verify whether they had derived all the functional requirements needed to let users perform each use case. During the final elicitation workshop, the participants walked through the test cases together to be sure they agreed on how the use cases should work. As with any quality control activity, they found additional errors in both the requirements and the test cases.

More Info Chapter 15, "Validating the Requirements," discusses generating test cases from requirements in more detail.

Use Cases and Functional Requirements

Software developers don't implement business requirements or use cases. They implement functional requirements, specific bits of system behavior that allow users to execute use cases and achieve their goals. Use cases describe system behavior from an actor's point of view, which omits a lot of details. A developer needs many other views to properly design and implement a system.

Some practitioners regard the use cases as being the functional requirements. However, I have seen organizations get into trouble when they simply handed use-case descriptions to developers for implementation. Use cases describe the user's perspective, looking at the externally visible behavior of the system. They don't contain all the information that a developer needs to write the software. For instance, the user of an ATM doesn't care about any back-end processing the ATM must perform, such as communicating with the bank's computer. This detail is invisible to the user, yet the developer needs to know about it. Of course, the use-case descriptions can include this kind of back-end processing detail, but it typically won't come up in discussions with end users. Developers who are presented with even fully dressed use cases often have many questions. To reduce this uncertainty, I recommend that requirements analysts explicitly specify the detailed functional requirements necessary to implement each use case (Arlow 1998).

Many functional requirements fall right out of the dialog steps between the actor and the system. Some are obvious, such as "The system shall assign a unique sequence number to each request." There is no point in duplicating those details in an SRS if they are perfectly clear from reading the use case. Other functional requirements don't appear in the use-case description. The analyst will derive them from an understanding of the use case and the system's operating environment. This translation from the user's view of the requirements to the developer's view is one of the many ways the requirements analyst adds value to the project.

The Chemical Tracking System employed the use cases primarily as a mechanism to reveal the necessary functional requirements. The analysts wrote only casual descriptions of the less complex use cases. They then derived all the functional requirements that, when implemented, would allow the actor to perform the use case, including alternative courses and exception handlers. The analysts documented these functional requirements in the SRS, which was organized by product feature.

You can document the functional requirements associated with a use case in several ways. The approach you choose depends on whether you expect your team to perform the design, construction, and testing from the use-case documents, from the SRS, or from a combination of both. None of these methods is perfect, so select the approach that best fits with how you want to document and manage your project's software requirements.

Use Cases Only

One possibility is to include the functional requirements right in each use-case description. You'll still need a separate supplementary specification to contain the nonfunctional requirements and any functional requirements that are not associated with specific use cases. Several use cases might need the same functional requirement. If five use cases require that the user's identity be authenticated, you don't want to write five different blocks of code for that purpose. Rather than duplicate them, cross-reference functional requirements that appear in multiple use cases. In some instances, the use-case *includes* relationship discussed earlier in this chapter solves this problem.

Use Cases and SRS

Another option is to write fairly simple use-case descriptions and document the functional requirements derived from each use case in an SRS. In this approach, you'll need to establish traceability between the use cases and their associated functional requirements. The best way to manage the traceability is to store all use cases and functional requirements in a requirements management tool.

More Info See Chapter 21 for more information on requirements management tools.

SRS Only

A third approach is to organize the SRS by use case or by feature and include both the use cases and the functional requirements in the SRS. This is the approach that the Chemical Tracking System team used. This scheme doesn't use separate use-case documents. You'll need to identify duplicated functional requirements or state every functional requirement only once and refer to that initial statement whenever the requirement reappears in another use case.

Benefits of Use Cases

True Stories The power of the use-case approach comes from its task-centric and user-centric perspective. The users will have clearer expectations of what the new system will let them do than if you take a function-centric approach. The customer representatives on several Internet development projects found that use cases clarified their notions of what visitors to their Web sites should be able to do. Use cases help analysts and developers understand both the user's business and the application domain. Carefully thinking through the actor-system dialog sequences can reveal ambiguity and vagueness early in the development process, as does generating test cases from the use cases.

It's frustrating and wasteful for developers to write code that never gets used. Overspecifying the requirements up front and trying to include every conceivable function can lead to implementing unnecessary requirements. The use-case approach leads to functional requirements that will allow the user to perform certain known tasks. This helps prevent "orphan functionality," those functions that seem like a good idea during elicitation but which no one uses because they don't relate directly to user tasks.

The use-case approach helps with requirements prioritization. The highest priority functional requirements are those that originated in the top priority use cases. A use case could have high priority for several reasons:

- It describes one of the core business processes that the system enables.
- Many users will use it frequently.
- A favored user class requested it.
- It provides a capability that's required for regulatory compliance.

- Other system functions depend on its presence.

Trap Don't spend a lot of time detailing use cases that won't be implemented for months or years. They're likely to change before construction begins.

There are technical benefits, too. The use-case perspective reveals some of the important domain objects and their responsibilities to each other. Developers using object-oriented design methods can turn use cases into object models such as class and sequence diagrams. (Remember, though, use cases are by no means restricted to object-oriented development projects.) As the business processes change over time, the tasks that are embodied in specific use cases will change. If you've traced functional requirements, designs, code, and tests back to their parent use cases—the voice of the customer—it will be easier to cascade those business-process changes throughout the entire system.

Use-Case Traps to Avoid

As with any software engineering technique, there are many ways to go astray when applying the use-case approach (Kulak and Guiney 2000; Lilly 2000). Watch out for the following traps:

- **Too many use cases** If you're caught in a use-case explosion, you might not be writing them at the appropriate level of abstraction. Don't create a separate use case for every possible scenario. Instead, include the normal course, alternative courses, and exceptions as scenarios within a single use case. You'll typically have many more use cases than business requirements and features, but many more functional requirements than use cases.

- True Stories **Highly complex use cases** I once reviewed a use case that had four dense pages of dialog steps, with a lot of embedded logic and branching conditions. It was incomprehensible. You don't have control over the complexity of the business tasks, but you can control how you represent them in use cases. Select one success path through the use case, with one combination of true and false values for the various logical decisions, and call that the normal course. Use alternative courses for the other logic branches that lead to success, and use exceptions to handle the branches that lead to failure. You might have many alternative courses, but each one will be short and easy to understand. To keep them simple, write use cases in terms of the essence of the actor and system behaviors instead of specifying every action in great detail.

- **Including user interface design in the use cases** Use cases should focus on what the users need to accomplish with the system's help, not on how the screens will look. Emphasize the conceptual interactions between the actors and the system, and defer user interface specifics to the design stage. For example, say "System presents choices" instead of "System displays drop-down list." Don't let the user interface design drive the requirements exploration. Use screen sketches and dialog maps (user interface architecture diagrams, as described in Chapter 11) to help visualize the actor-system interactions, not as firm design specifications.

- **Including data definitions in the use cases** I've seen use cases that included definitions of the data items and structures that are manipulated in the use case. This approach makes it difficult for project participants to find the definitions they need because it isn't obvious which use case contains each data definition. It can also lead to duplicate definitions, which get out of synch when one instance is changed and others are not. Instead of sprinkling them throughout the use cases, collect data definitions in a project-wide data dictionary, as discussed in Chapter 10, "Documenting the Requirements."

- **Use cases that users don't understand** If users can't relate the use-case description to their business processes or system tasks, there's a problem with the use case. Write use cases from the user's perspective, not the system's point of view, and ask users to review them. Perhaps the complexity of fully dressed use cases is overkill for your situation.

- **New business processes** Users will have difficulty coming up with use cases if software is being built to support a process that doesn't yet exist. Rather than requirements elicitation being a matter of modeling the business process, it might be a matter of inventing one. It's risky to expect a new information system to drive the creation of an effective business process. Perform the business process reengineering before fully specifying the new information system.

- **Excessive use of *includes* and *extends* relationships** Working with use cases for the first time requires a change in the way that analysts, users, and developers think about requirements. The subtleties of the use-case *includes* and *extends* relationships can be confusing. They are best avoided until you gain a comfort level with the use-case approach.

Event-Response Tables

Another way to organize and document user requirements is to identify the external events to which the system must respond. An *event* is some change or activity that takes place in the user's environment that stimulates a response from the software system (McMenamin and Palmer 1984; Wiley 2000). An *event-response table* (also called an *event table* or an *event list*) lists all such events and the behavior the system is expected to exhibit in reaction to each event. There are several types of system events, as shown in Figure 8-9 and described here:

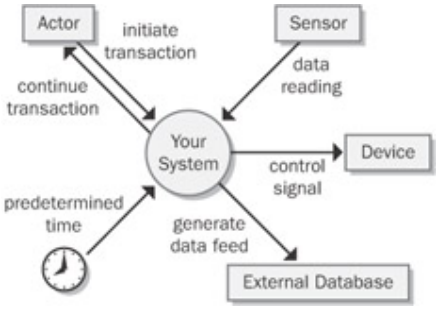


Figure 8-9: Examples of system events and responses.

- An action by a human user that stimulates a dialog with the software, as when the user initiates a use case (sometimes called a *business event*). The event-response sequences correspond to the dialog steps in a use case. Unlike use cases, the event-response table does not describe the user's goal in using the system or state why this event-response sequence provides value to the user.

- A control signal, data reading, or interrupt received from an external hardware device, such as when a switch closes, a voltage changes, or the user moves the mouse.

- A time-triggered event, as when the computer's clock reaches a specified time (say, to launch an automatic data export operation at midnight) or when a preset duration has passed since a previous event (as in a system that logs the temperature read by a sensor every 10 seconds).

Event-response tables are particularly appropriate for real-time control systems. Table 8-1 contains a sample event-response table that partially describes the behavior of an automobile's windshield wipers. Note that the expected response depends not only on the event but also on the state the system is in at the time the event takes place. For instance, events 4 and 5.1 in Table 8-1 result in slightly different behaviors depending on whether the wipers were on at the time the user set the wiper control to the intermittent setting. A response might simply alter some internal system information (events 4 and 7.1 in the table) or it could result in an externally visible result (most other events).

The event-response table records information at the user-requirements level. If the table defines and labels every possible combination of event, state, and response (including exception conditions), the table can also serve as part of the functional requirements for that portion of the system. However, the analyst must supply additional functional and nonfunctional requirements in the SRS. For example, how many cycles per minute does the wiper perform on the slow and fast wipe settings? Do the intermittent wipes operate at the slow or fast speed? Is the intermittent setting continuously variable or does it have discrete settings? What are the minimum and maximum delay times between intermittent wipes? If you stop requirements specification at the user-requirements level, the developer has to track down this sort of information himself. Remember, the goal is to specify the requirements precisely enough that a developer knows what to build.

Notice that the events listed in Table 8-1 are written at the *essential level* (describing the essence of the event), not at the *implementation level* (describing the specifics of the implementation). Table 8-1 shows nothing about what the windshield wiper controls look like or how the user manipulates them. The designer could implement these requirements with anything from the traditional stalk-mounted wiper controls in current automobiles to a system that uses voice recognition of spoken commands: "wipers on," "wipers off," "wipers fast," "wipe once," and so on. Writing user requirements at the essential level avoids imposing unnecessary design constraints. However, record any known design constraints to guide the designer's thinking.

Table 8-1: Event-Response Table for an Automobile Windshield-Wiper System

ID	Event	System State	System Response
1.1	set wiper control to low speed	wiper off	set wiper motor to low speed
1.2	set wiper control to low speed	wiper on high speed	set wiper motor to low speed
1.3	set wiper control to low speed	wiper on intermittent	set wiper motor to low speed
2.1	set wiper control to high speed	wiper off	set wiper motor to high speed
2.2	set wiper control to high speed	wiper on low speed	set wiper motor to high speed
2.3	set wiper control to high speed	wiper on intermittent	set wiper motor to high speed
3.1	set wiper control set to off	wiper on high speed	1. complete current wipe cycle 2. turn wiper motor off
3.2	set wiper control set to off	wiper on low speed	1. complete current wipe cycle 2. turn wiper motor off
3.3	set wiper control set to off	wiper on intermittent	1. complete current wipe cycle 2. turn wiper motor off
4	set wiper control to intermittent	wiper off	1. read wipe time interval setting 2. initialize wipe timer
5.1	set wiper control to intermittent	wiper on high speed	1. read wipe time interval setting 2. complete current wipe cycle 3. initialize wipe timer
5.2	set wiper control to intermittent	wiper on low speed	1. read wipe time interval setting 2. complete current wipe cycle

			3. initialize wipe timer
6	wipe time interval has passed since completing last cycle	wiper on intermittent	perform one low-speed wipe cycle
7.1	change intermittent wiper interval	wiper on intermittent	1. read wipe time interval setting 2. initialize wipe timer
7.2	change intermittent wiper interval	wiper off	no response
7.3	change intermittent wiper interval	wiper on high speed	no response
7.4	change intermittent wiper interval	wiper on low speed	no response
8	immediate wipe signal received	wiper off	perform one low-speed wipe cycle

Next Steps

- Write several use cases for your current project using the template in Figure 8-6. Include any alternative courses and exceptions. Identify the functional requirements that will allow the user to successfully complete each use case. Check whether your current SRS already includes all these functional requirements.
- List the external events that could stimulate your system to behave in specific ways. Create an event-response table that shows the state the system is in when each event is received and how the system is to respond.

Chapter 9: Playing by the Rules

Overview

"*Hi, Tim, this is Jackie. I'm having a problem requesting a chemical with the Chemical Tracking System. My manager suggested that I ask you about it. He said you were the product champion who provided many of the requirements for this system.*"

"Yes, that's correct," Tim replied. "What's the problem?"

"I need to get some more phosgene for those dyes that I make for my research project," said Jackie, "but the Chemical Tracking System won't accept my request. It says I haven't taken a training class in handling hazardous chemicals in more than a year. What's that all about? I've been using phosgene and even nastier chemicals for years with no problem. Why can't I get some more?"

"You're probably aware that Contoso Pharmaceuticals requires an annual refresher class in the safe handling of hazardous chemicals," Tim pointed out. "This is a corporate policy; the Chemical Tracking System just enforces it. I know the stockroom guys used to give you whatever you wanted, but we can't do that anymore for liability insurance reasons. Sorry about the inconvenience, but we have to comply with the rule. You'll have to

take the refresher training before the system will let you request more phosgene."

Every business organization operates according to an extensive set of corporate policies, laws, and industry standards. Industries such as banking, aviation, and medical device manufacture must comply with volumes of government regulations. Such controlling principles are collectively known as *business rules*. Software applications often enforce these business rules. In other cases, business rules aren't enforced in software but are controlled through human enforcement of policies and procedures.

Most business rules originate outside the context of any specific software application. Rules such as the corporate policy requiring annual training in handling hazardous chemicals apply even if all chemical purchasing and dispensing is done manually. Standard accounting practices applied even in the days of green eyeshades and fountain pens. However, business rules are a major source of software functional requirements because they dictate capabilities that the system must possess to conform to the rules. Even high-level business requirements can be driven by business rules, such as the Chemical Tracking System's need to generate reports to comply with federal and state regulations regarding chemical storage and disposal.

Not all companies treat their essential business rules as the valuable asset they are. If this information is not properly documented and managed, it exists only in the heads of individuals. Various individuals can have conflicting understandings of the rules, which can lead to different software applications enforcing a common business rule inconsistently. If you know where and how each application implements its pertinent business rules, it's much easier to change the applications when a rule changes.

Trap Undocumented business rules known only to experts result in a knowledge vacuum when those experts retire or change jobs.

In a large enterprise, only a subset of the collected business rules will pertain to any one software application. Building a collection of rules helps each requirements analyst identify potential sources of requirements that might not otherwise come up in discussions with users. Having a master repository of business rules makes it easier for all applications that are affected by the rules to implement them in a consistent fashion.

PREVIOUS NEXT
PREVIOUS NEXT

The Rules of the Business

According to the Business Rules Group (1993), "A business rule is a statement that defines or constrains some aspect of the business. It is intended to assert business structure or to control or influence the behavior of the business." Whole methodologies have been developed based on the discovery and documentation of business rules and their implementation in automated business rules systems (Ross 1997; von Halle 2002). Unless you're building a system that is heavily business-rule driven, you don't need an elaborate methodology. Simply identify and document the rules that pertain to your system and link them to specific functional requirements.

Many different *taxonomies* (classification schemes) have been proposed for organizing business rules (Ross 2001; Morgan 2002; von Halle 2002). The simple scheme shown in Figure 9-1, with five types of business rules, will work for most situations. A sixth category is *terms*, defined words, phrases, and abbreviations that are important to the business. A glossary is a convenient place to define terms. Recording the business rules in a consistent way so that they add value to your software development efforts is more important than having heated arguments about how to classify each one. Let's look at the different kinds of business rules you might encounter.

Figure 9-1: A simple business rule taxonomy.

Facts

Facts are simply statements that are true about the business. Often facts describe associations or relationships between important business terms. Facts are also called *invariants*—immutable truths about data entities and their attributes. Other business rules might refer to certain facts, but facts themselves don't usually translate directly into software functional requirements. Facts about data entities that are important to the system might appear in data models that the analyst or database designer creates. (See Chapter 11, "A Picture Is Worth 1024 Words," for more information about data modeling.) Examples of facts are:

- Every chemical container has a unique bar code identifier.
- Every order must have a shipping charge.
- Each line item in an order represents a specific combination of chemical, grade, container size, and number of containers.
- Nonrefundable tickets incur a fee when the purchaser changes the itinerary.
- Sales tax is not computed on shipping charges.

Constraints

Constraints restrict the actions that the system or its users may perform. Some words and phrases that suggest someone is describing a constraint business rule are *must*, *must not*, *may not*, and *only*. Examples of constraints are:

- A borrower who is less than 18 years old must have a parent or a legal guardian as cosigner on the loan.
- A library patron may place up to 10 items on hold.
- A user may request a chemical on the Level 1 hazard list only if he has had hazardous-chemical training within the past 12 months.
- All software applications must comply with government regulations for usage by visually impaired persons.
- Correspondence may not display more than four digits of the policyholder's Social Security number.
- Commercial airline flight crews must receive at least eight hours of continuous rest in every 24-hour period.

So Many Constraints

Software projects have many kinds of constraints. Project managers must work within schedule, staff, and budget limitations. These project-level constraints belong in the software project management plan. Product-level design and implementation constraints that restrict the choices available to the developer belong in the

SRS or the design specification. Many business rules impose constraints on the way the business operates. Whenever these constraints are reflected in the software functional requirements, indicate the pertinent rule as the rationale for each such derived requirement.

Your organization likely has security policies that control access to information systems. Such policies typically state that passwords will be used and present rules regarding frequency of required password changes, whether or not previous passwords may be reused, and the like. These are all constraints about application access that could be considered business rules. Tracing each such rule into the specific code that implements it makes it easier to update systems to comply with changes in the rules, such as changing the required frequency of password changes from 90 days to 30 days.

Overruled

True Stories I recently redeemed some of my frequent-flyer miles on Fly-By-Night Airlines to buy a ticket for my wife, Chris. When I attempted to finalize the purchase, FlyByNight.com told me that it had encountered an error and couldn't issue the ticket. It told me to call 800-FLY-NITE immediately. The reservation agent I spoke with told me that the airline couldn't issue a mileage award ticket through the mail or e-mail because Chris and I have different last names. I had to go to the airport ticket counter and show identification to have the ticket issued.

This incident resulted from this constraining business rule: "If the passenger has a different last name from the mileage redeemer, then the redeemer must pick up the ticket in person." The reason for this business rule is probably fraud prevention. The software driving the Fly-By-Night Web site enforces the rule, but in a way that resulted in inconvenience for the customer and in usability shortcomings. Rather than simply telling me about the rule regarding different last names and what I needed to do, the system displayed an alarming error message. It wasted my time and the Fly-By-Night reservation agent's time with an unnecessary phone call. Poorly thought-out business rule implementations can have an adverse effect on your customer and hence on your business.

Action Enablers

A rule that triggers some activity under specific conditions is an *action enabler*. A person could perform the activity in a manual process. Alternatively, the rule might lead to specifying some software functionality that makes an application exhibit the correct behavior when the specified conditions are true. The conditions that lead to the action could be complex combinations of true and false values for multiple individual conditions. A decision table such as that shown in Chapter 11 provides a concise way to document action-enabling business rules that involve extensive logic. A statement in the form "If <some condition is true or some event takes place>, then <something happens>" is a clue that someone is describing an action enabler. Following are some examples of action-enabling business rules:

- If the chemical stockroom has containers of a requested chemical in stock, then offer existing containers to the requester.
- If the expiration date for a chemical container has been reached, then notify the person who currently possesses that container.
- On the last day of a calendar quarter, generate the mandated OSHA and EPA reports on chemical handling and disposal for that quarter.

- If the customer ordered a book by an author who has written multiple books, then offer the author's other books to the customer before accepting the order.

Inferences

Sometimes called *inferred knowledge*, an *inference* is a rule that establishes some new knowledge based on the truth of certain conditions. An inference creates a new fact from other facts or from computations. Inferences are often written in the "if/then" pattern also found in action-enabling business rules, but the "then" clause of an inference implies a fact or a piece of information, not an action to be taken. Some examples of inferences are:

- If a payment is not received within 30 calendar days of the date it is due, then the account is delinquent.
- If the vendor cannot ship an ordered item within five days of receiving the order, then the item is back-ordered.
- A container of a chemical that can form explosive decomposition products is considered expired one year after its manufacture date.
- Chemicals with an LD_{50} toxicity lower than 5 mg/kg in mice are considered hazardous.

Computations

Computers compute, so one class of business rules defines *computations* that are performed using specific mathematical formulas or algorithms. Many computations follow rules that are external to the enterprise, such as income tax withholding formulas. Whereas action-enabling business rules lead to specific software functional requirements to enforce them, computational rules can normally serve as software requirements as they are stated. Following are some examples of computational business rules in text form; alternatively, you could represent these in some symbolic form, such as a mathematical expression. Presenting such rules in the form of a table, as Table 9-1 does, is clearer than writing a long list of complex natural language rule statements.

- The unit price is reduced by 10% for orders of 6 to 10 units, by 20% for orders of 11 to 20 units, and by 35% for orders of more than 20 units.
- The domestic ground shipping charge for an order that weighs more than 2 pounds is $4.75 plus 12 cents per ounce or fraction thereof.
- The commission for securities trades completed on line is $12 per trade of 1 through 5000 shares. The commission for trades conducted through an account officer is $45 per trade of 1 through 5000 shares. Commissions on trades of greater than 5000 shares are one-half of these commissions.
- The total price for an order is computed as the sum of the price of the items ordered, less any volume discounts, plus state and county sales tax for the location to which the order is being shipped, plus the shipping charge, plus an optional insurance charge.

Table 9-1: Using a Table to Represent Computational Business Rules

ID	Number of Units Purchased	Percent Discount
DISC-1	1 through 5	0
DISC-2	6 through 10	10
DISC-3	11 through 20	20
DISC-4	more than 20	35

A single computation could include many elements. The total price computation in the final example in the preceding list includes volume discount, sales tax, shipping charge, and insurance charge computations. That rule is complicated and hard to understand. To correct this shortcoming, write your business rules at the atomic level, rather than combining many details into a single rule. A complex rule can point back to the individual rules upon which it relies. This keeps your rules short and simple. It also facilitates reusing the rules and combining them in various ways. To write inferred knowledge and action-enabling business rules in an atomic way, don't have "or" logic on the left-hand side of an "if/then" construct and avoid "and" logic on the right-hand side (von Halle 2002). Atomic business rules that could affect the total price computation in the last example in the preceding list include the discount rules in Table 9-1 and the following:

- If the delivery address is in a state that has a sales tax, then the state sales tax is computed on the total discounted price of the items ordered.
- If the delivery address is in a county that has a sales tax, then the county sales tax is computed on the total discounted price of the items ordered.
- The insurance charge is 1 percent of the total discounted price of the items ordered.
- Sales tax is not computed on shipping charges.
- Sales tax is not computed on insurance charges.

These business rules are called *atomic* because they can't be decomposed further. You will likely end up with many atomic business rules, and your computations and functional requirements will depend on various groupings of them.

PREVIOUS NEXT
PREVIOUS NEXT

Documenting Business Rules

Because business rules can influence multiple applications, organizations should manage their business rules as enterprise-level—not project-level—assets. A simple business rules catalog will suffice initially. Large organizations or those whose business operations and information systems are heavily business-rule driven should establish a database of business rules. Commercial rule-management tools become valuable if your rules catalog outgrows a word processor or spreadsheet solution or if you want to automate aspects of implementing the rules in your applications. The Business Rules Group maintains a list of business-rule management products at *http://www.businessrulesgroup.org/brglink.htm*. As you identify new rules while working on an application, add them to your catalog rather than embedding them in the documentation for that specific application or—worse—only in its code. Rules related to safety, security, finance, or regulatory compliance pose the highest risk if they are not managed and enforced appropriately.

Trap Don't make your business rules catalog more complex than necessary. Use the simplest form of documenting business rules that ensures that your development teams will use them effectively.

As you gain experience with identifying and documenting business rules, you can apply structured templates for defining rules of different types (Ross 1997; von Halle 2002). These templates describe patterns of keywords and clauses that structure the rules in a consistent fashion. They also facilitate storing the rules in a database, a commercial business-rule management tool, or a rule engine. To begin, though, try the simple format illustrated in Table 9-2 (Kulak and Guiney 2000).

As shown in this table, giving each business rule a unique identifier lets you trace functional requirements back

to a specific rule. The "Type of Rule" column identifies the rule as being a fact, a constraint, an action enabler, an inference, or a computation. The "Static or Dynamic" column indicates how likely the rule is to change over time. Sources of business rules include corporate and management policies, subject matter experts and other individuals, documents such as government regulations, and existing software code or database definitions.

Table 9-2: Sample Business Rules Catalog

ID	Rule Definition	Type of Rule	Static or Dynamic	Source
ORDER-15	A user may request a chemical on the Level 1 hazard list only if he has had hazardous-chemical training within the past 12 months.	Constraint	Static	Corporate policy
DISC-13	An order discount is calculated based on the size of the current order, as defined in Table 9-1.	Computation	Dynamic	Corporate policy

Business Rules and Requirements

Simply asking users "What are your business rules?" doesn't get you very far, just as asking "What do you want?" doesn't help much when eliciting user requirements. Depending on the application, sometimes you invent business rules as you go along and sometimes you discover them during requirements discussions. Often the project stakeholders already know about business rules that will influence the application, and the development team must work within the boundaries that the rules define. Barbara von Halle (2002) describes a comprehensive process for discovering business rules.

During user requirements elicitation workshops, the analyst can ask questions to probe around the rationale for the requirements and constraints that users present. These discussions frequently surface business rules as the underlying rationale. The analyst can harvest business rules during elicitation workshops that also define other requirements artifacts and models (Gottesdiener 2002). Figure 9-2 shows several potential origins of rules (and, in some cases, of use cases and functional requirements). The figure also suggests the types of questions the analyst can ask when workshop participants are discussing various issues and objects. The analyst should document the business rules that come out of these elicitation discussions and ask the right people to confirm their correctness and to supply any missing information.

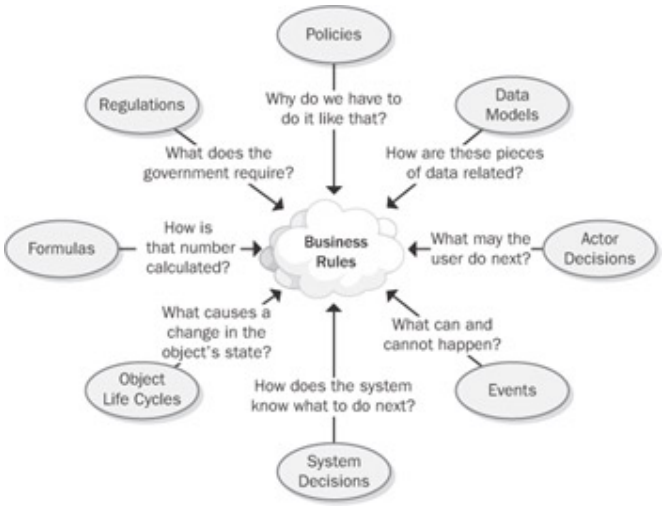


Figure 9-2: Discovering business rules by asking questions from different perspectives.

After identifying and documenting business rules, determine which ones must be implemented in the software. Some rules will lead to use cases and hence to functional requirements that enforce the rule. Consider the following three rules:

- **Rule #1 (action enabler)** "If the expiration date for a chemical container has been reached, then notify the person who currently possesses that container."
- **Rule #2 (inference)** "A container of a chemical that can form explosive decomposition products is considered expired one year after its manufacture date."
- **Rule #3 (fact)** "Ethers can spontaneously form explosive peroxides."

These rules serve as the origin for a use case called "Notify Chemical Owner of Expiration." One functional requirement for that use case is "The system shall e-mail a notification to the current owner of a chemical container on the date the container expires."

You can define the links between a functional requirement and its parent business rules in the following two ways:

- Use a requirement attribute called "Origin" and indicate the rules as the origin of the functional requirement. (See Chapter 18, "Requirements Management Principles and Practices.")
- Define traceability links between a functional requirement and the pertinent business rules in the requirements traceability matrix. (See Chapter 20, "Links in the Requirements Chain.")

Data referential integrity rules frequently are implemented in the form of database triggers or stored procedures. Such rules describe the data updates, insertions, and deletions that the system must perform because of relationships between data entities (von Halle 2002). For example, the system must delete all undelivered line items in an order if the customer cancels the order.

Sometimes business rules and their corresponding functional requirements look much alike. However, the rules are external statements of policy that must be enforced in software, thereby driving system functionality. Every analyst must decide which existing rules pertain to his application, which ones must be enforced in the software, and how to enforce them.

Recall the constraint rule from the Chemical Tracking System requiring that training records be current before a user can request a hazardous chemical. The analyst would derive different functional requirements to comply with this rule, depending on whether the training records database is available on line. If it is, the system can simply look up the user's training record and decide whether to accept or reject the request. If the records aren't available on-line, the system might store the chemical request temporarily and send e-mail to the training coordinator, who in turn could either approve or reject the request. The rule is the same in either situation, but the software functional requirements—the actions to take when the business rule is encountered during execution—vary depending on the system's environment.

To prevent redundancy, don't duplicate rules from your business rules catalog in the SRS. Instead, the SRS should refer back to specific rules as the source of, say, algorithms for income tax withholding. This approach provides several advantages:

- It obviates the need to change both the business rule and the corresponding functional requirements if the rule changes.
- It keeps the SRS current with rule changes because the SRS simply refers to the master copy of the rule.

- It facilitates reusing the same rule in several places in the SRS and across multiple projects without risking an inconsistency, because the rules are not buried in the documentation for any single application.

A developer reading the SRS will need to follow the cross-reference link to access the rule details. This is the trade-off that results when you elect not to duplicate information. There's another risk of keeping the rules separate from the functional requirements: Rules that make sense in isolation might not be so sensible when viewed in an operational context with other requirements. As with so many aspects of requirements engineering, there is no simple, perfect solution to managing business rules that works in all situations.

Organizations that do a masterful job of managing their business rules often don't publicly share their experiences. They view their use of business rules to guide their software portfolio management as providing a competitive advantage over organizations that deal with their rules more casually. Once you begin actively looking for, recording, and applying business rules, the rationale behind your application development choices will become clearer to all stakeholders.

Next Steps

- List all the business rules you can think of that pertain to the project you are currently working on. Begin populating a business rule catalog, classifying the rules according to the scheme shown in Figure 9-1, and noting the origin of each rule.
- Identify the rationale behind each of your functional requirements to discover other business rules.
- Set up a traceability matrix to indicate which functional requirements or database elements implement each of the business rules that you identified.

Chapter 10: Documenting the Requirements

Overview

The result of requirements development is a documented agreement between customers and developers about the product to be built. As we saw in earlier chapters, the vision and scope document contains the business requirements, and the user requirements often are captured in the form of use cases. The product's detailed functional and nonfunctional requirements reside in a software requirements specification (SRS). Unless you write all these requirements in an organized and readable fashion and have key project stakeholders review them, people will not be sure what they're agreeing to.

We can represent software requirements in several ways:

- 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 using mathematically precise formal logic languages

Formal specifications provide the greatest rigor and precision, but few software developers—and fewer

customers—are familiar with them, so they can't review the specifications for correctness. See Davis (1993) for references to further discussions of formal specification methods. Despite its shortcomings, structured natural language, augmented with graphical models, remains the most practical way for most software projects to document their requirements.

Even the finest requirements specification can never replace interpersonal discussions throughout the project. It's impossible to discover every fragment of information that feeds into software development up front. Keep the communication lines open between the development team, customer representatives, testers, and other stakeholders so that they can quickly address the myriad issues that will arise.

This chapter addresses the purpose, structure, and contents of the SRS. It presents guidelines for writing functional requirements, along with examples of flawed requirements and suggestions for improving them. As an alternative to traditional word-processing documents, the requirements can be stored in a database, such as a commercial requirements management tool. These tools make it much easier to manage, use, and communicate the requirements, but they're only as good as the quality of the information they hold. Chapter 21 discusses requirements management tools.g

PREVIOUS NEXT
PREVIOUS NEXT

The Software Requirements Specification

The software requirements specification is sometimes called a *functional specification,* a *product specification*, a *requirements document*, or a *system specification*, although organizations do not use these terms in the same way. 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. It should describe as completely as necessary the system's behaviors under various conditions. It should not contain design, construction, testing, or project management details other than known design and implementation constraints.

Several audiences rely on the SRS:

- Customers, the marketing department, and sales staff need to know what product they can expect to be delivered.
- Project managers base their estimates of schedule, effort, and resources on the product description.
- The SRS tells the software development team what to build.
- The testing group uses the SRS to develop test plans, cases, and procedures.
- The SRS tells maintenance and support staff what each part of the product is supposed to do.
- Documentation writers base user manuals and help screens on the SRS and the user interface design.
- Training personnel use the SRS and user documentation to develop educational materials.
- Legal staff ensure that the requirements comply with applicable laws and regulations.
- Subcontractors base their work on, and can be legally held to, the SRS.

As the ultimate repository for the product requirements, the SRS must be comprehensive. Developers and

customers should make no assumptions. If any desired capability or quality doesn't appear somewhere in the requirements agreement, no one should expect it to appear in the product.

You don't have to write the SRS for the entire product before beginning development, but you do need to capture the requirements for each increment before building that increment. Incremental development is appropriate when the stakeholders cannot identify all the requirements at the outset and if it's imperative to get some functionality into the users' hands quickly. However, every project should baseline an agreement for each set of requirements before the team implements them. *Baselining* is the process of transitioning an SRS under development into one that has been reviewed and approved. Working from the same set of requirements minimizes miscommunication and unnecessary rework.

Organize and write the SRS so that the diverse stakeholders can understand it. Chapter 1, "The Essential Software Requirement," presented several characteristics of high-quality requirements statements and specifications. Keep the following requirements readability suggestions in mind:

- Label sections, subsections, and individual requirements consistently.
- Leave text ragged on the right margin, rather than fully justified.
- Use white space liberally.
- Use visual emphasis (such as bold, underline, italics, and different fonts) consistently and judiciously.
- Create a table of contents and perhaps an index to help readers find the information they need.
- Number all figures and tables, give them captions, 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 locations within a document.
- Use hyperlinks to let the reader jump to related sections in the SRS or in other documents.
- Use an appropriate template to organize all the necessary information.

Labeling Requirements

To satisfy the quality criteria of traceability and modifiability, every functional requirement must be uniquely and persistently identified. This allows you to refer to specific requirements in a change request, modification history, cross-reference, or requirements traceability matrix. It also facilitates reusing the requirements in multiple projects. Bullet lists aren't adequate for these purposes. Let's look at the advantages and shortcomings of several requirements-labeling methods. Select whichever technique makes the most sense for your situation.

Sequence Number

The simplest approach gives every requirement a unique sequence number, such as UR-9 or SRS-43. Commercial requirements management tools assign such an identifier when a user adds a new requirement to the tool's database. (The tools also support hierarchical numbering.) The prefix indicates the requirement type, such as *UR* for *user requirement*. A number is not reused if a requirement is deleted. This simple numbering approach doesn't provide any logical or hierarchical grouping of related requirements, and the labels give no clues as to what each requirement is about.

Hierarchical Numbering

In the most commonly used convention, if the functional requirements appear in section 3.2 of your SRS, they will all have labels that begin with 3.2 (for example, 3.2.4.3). More digits indicate a more detailed, lower-level requirement. This method is simple and compact. Your word processor can assign the numbers automatically. However, the labels can grow to many digits in even a medium-sized SRS. In addition, numeric labels tell you nothing about the purpose of each requirement. More seriously, this scheme does not generate persistent labels. If you insert a new requirement, the numbers of all following requirements in that section will be incremented. Delete or move a requirement, and the numbers following it in that section will be decremented. These changes disrupt any references to those requirements elsewhere in the system.

Trap An analyst once told me, "We don't let people insert requirements—it messes up the numbering." Don't let ineffective practices hamper your ability to work effectively and sensibly.

An improvement over hierarchical numbering 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 3.5—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 less positionally dependent.

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 to print more than 10 copies." This requirement might be tagged Print.ConfirmCopies. This indicates that it is part of the print function and relates to the number of copies to print. Hierarchical textual tags are structured, meaningful, and unaffected by adding, deleting, or moving other requirements. Their drawback is that they are bulkier than the numeric labels, but that's a small price to pay for stable 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 external interface description, or build a prototype before you can resolve this uncertainty. Use the notation *TBD* (to be determined) to flag these knowledge gaps.

Resolve all TBDs before implementing a set of requirements. Any uncertainties that remain increase the risk of a developer or a tester making errors and having to perform rework. When the developer encounters a TBD, he might not go back to the requirement's originator to resolve it. The developer might make his best guess, which won't always be correct. If you must proceed with construction while TBDs remain, either defer implementing the unresolved requirements or design those portions of the product to be easily modifiable when the open issues are resolved.

Trap TBDs won't resolve themselves. Record who is responsible for resolving each issue, how, and by when. Number the TBDs to help track them to closure.

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 describe solutions (designs), not requirements. Delaying baselining of the SRS until the user interface design is complete can try the patience of people who are already concerned about spending too much time on requirements. Including UI design in the requirements can result in the visual design driving the requirements, which leads to functional gaps.

True Stories Screen layouts don't replace user and functional requirements. Don't expect developers to deduce the underlying functionality and data relationships from screen shots. One Internet development

company got in trouble repeatedly because the team went directly from signing a contract with a customer into an eight-hour visual design workshop. They never sufficiently understood what the user would be able to do at the Web site, so they spent a lot of time fixing it after delivery.

On the plus side, exploring possible user interfaces (such as with a working prototype) makes the requirements tangible to both users and developers. User interface displays can assist with project planning and estimation. Counting graphical user interface (GUI) elements or the number of function points[1] associated with each screen yields a size estimate, which leads to an estimate of implementation effort.

A sensible balance is to include conceptual images—sketches—of selected displays in the SRS without demanding that the implementation precisely follow those models. This enhances communication without handicapping the developers with unnecessary constraints. For example, a preliminary sketch of a complex dialog box will illustrate the intent behind a portion of the requirements, but a skilled visual designer might turn it into a tabbed dialog box to improve usability.

[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 2002).

PREVIOUS NEXT
PREVIOUS NEXT

A Software Requirements Specification Template

Every software development organization should adopt one or more standard SRS templates for its projects. Various SRS templates are available (Davis 1993; Robertson and Robertson 1999; Leffingwell and Widrig 2000). Many people use templates derived from the one described in IEEE Standard 830-1998, "IEEE Recommended Practice for Software Requirements Specifications" (IEEE 1998b). This template is suitable for many kinds of projects, but it has some limitations and confusing elements. If your organization tackles various kinds or sizes of projects, such as new, large system development as well as minor enhancements to legacy systems, adopt an SRS template for each major project class.

Figure 10-1 illustrates an SRS template that was adapted from the IEEE 830 standard; the standard gives many examples of additional specific product requirements to include. Appendix D of this book contains a sample SRS that generally follows this template. Modify the template to fit the needs and nature of your projects. If a section of your template doesn't apply to a particular project, leave the section heading in place but indicate that it isn't applicable. This will keep the reader from wondering whether something important was omitted inadvertently. If your projects consistently omit the same sections, it's time to tune up the template. 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. Sometimes a bit of information could logically be recorded in several template sections. It's more important to capture information thoroughly and consistently than to have religious arguments over where each item belongs.

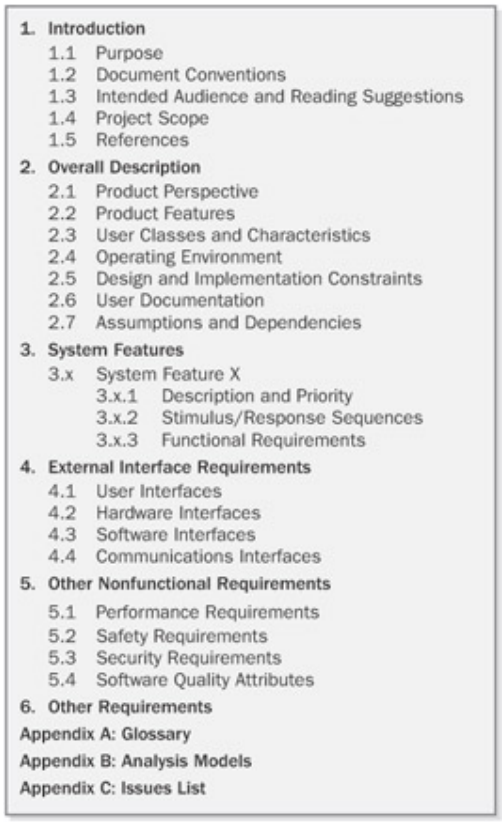


Figure 10-1: Template for software requirements specification.

There is a bit of overlap between this SRS template and the vision and scope document template in Figure 5-2 (for example, the project scope, product features, and operating environment sections). This is because for some projects you will choose to create only a single requirements document. If you do use both templates, adjust them to eliminate redundancy between the two, combining sections as appropriate. It might be appropriate to use the corresponding section in the SRS to detail some preliminary or high-level information that appeared in the vision and scope document. If you find yourself using cut-and-paste from one project document to another, the documents contain unnecessary duplication.

The rest of this section describes the information to include in each section of the SRS. You can incorporate material by reference to other existing project documents (such as a vision and scope document or an interface specification) instead of duplicating information in the SRS.

1. Introduction

The introduction presents an overview to help the reader understand how the SRS is organized and how to use it.

1.1 Purpose

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

1.2 Document Conventions

Describe any standards or typographical conventions, including text styles, highlighting, or significant notations. For instance, state whether the priority shown for a high-level requirement is inherited by all its detailed requirements or whether every functional requirement statement is to have its own priority rating.

1.3 Intended Audience and Reading Suggestions

List the different readers to whom the SRS is directed. 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 Project Scope

Provide a short description of the software being specified and its purpose. Relate the software to user or corporate goals and to business objectives and strategies. If a separate vision and scope document is available, refer to it rather than duplicating its contents here. An SRS that specifies an incremental release of an evolving product should contain its own scope statement as a subset of the long-term strategic product vision.

1.5 References

List any documents or other resources to which this SRS refers, including hyperlinks to them if possible. These might include user interface style guides, contracts, standards, system requirements specifications, use-case documents, interface specifications, concept-of-operations 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 (such as network folder or URL).

2. Overall Description

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

2.1 Product Perspective

Describe the product's context and origin. Is it the next member of a growing product family, the next version of a mature system, a replacement for an existing application, or an entirely new product? If this SRS defines a component of a larger system, state how this software relates to the overall system and identify major interfaces between the two.

2.2 Product Features

List the major features the product contains or the significant functions that it performs. Details will be provided in Section 3 of the SRS, so you need only a high-level summary here. A picture of the major groups of requirements and how they are related, such as a top-level data flow diagram, a use-case diagram, or a class diagram, might 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 6, "Finding the Voice of the Customer.") Some requirements might pertain only to certain user classes. Identify the favored user classes. User classes represent a subset of the stakeholders described in the vision and scope document.

2.4 Operating Environment

Describe the environment in which the software will operate, including the hardware platform, the operating systems and versions, and the geographical locations of users, servers, and databases. List any other software components or applications with which the system must peacefully coexist. The vision and scope document might contain some of this information at a high level.

2.5 Design and Implementation Constraints

Describe any factors that will restrict the options available to the developers and the rationale for each constraint. Constraints might include the following:

- Specific technologies, tools, programming languages, and databases that must be used or avoided.
- Restrictions because of the product's operating environment, such as the types and versions of Web browsers that will be used.
- Required development conventions or standards. (For instance, if the customer's organization will be maintaining the software, the organization might specify design notations and coding standards that a subcontractor must follow.)
- Backward compatibility with earlier products.
- Limitations imposed by business rules (which are documented elsewhere, as discussed in Chapter 9).
- Hardware limitations such as timing requirements, memory or processor restrictions, size, weight, materials, or cost.
- Existing user interface conventions to be followed when enhancing an existing product.
- Standard data interchange formats such as XML.

2.6 User Documentation

List the user documentation components that will be delivered along with the executable software. These could include user manuals, online help, and tutorials. Identify any required documentation delivery formats, standards, or tools.

2.7 Assumptions and Dependencies

An *assumption* is a statement that is believed to be true in the absence of proof or definitive knowledge. Problems can arise if assumptions are incorrect, are not shared, or change, so certain assumptions will translate into project risks. One SRS reader might assume that the product will conform to a particular user interface convention, whereas another assumes something different. A developer might assume that a certain set of functions will be custom-written for this application, but the analyst assumes that they will be reused from a previous project, and the project manager expects to procure a commercial function library.

Identify any *dependencies* the project has on external factors outside its control, such as the release date of the next version of an operating system or the issuing of an industry standard. If you expect to integrate into the system some components that another project is developing, you depend 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. System Features

The template in Figure 10-1 is organized by system feature, which is just one possible way to arrange the functional requirements. Other organizational options include by use case, mode of operation, user class, stimulus, response, object class, or functional hierarchy (IEEE 1998b). Combinations of these elements are also possible, such as use cases within user classes. There is no single right choice; you should select an organizational approach that makes it easy for readers to understand the product's intended capabilities. I'll

describe the feature scheme as an example.

3.x System Feature X

State the name of the feature in just a few words, such as "3.1 Spell Check." Repeat subsections 3.x.1 through 3.x.3 for each system feature.

3.x.1 Description and Priority

Provide a short description of the feature and indicate whether it is of high, medium, or low priority. (See Chapter 14, "Setting Requirement Priorities.") Priorities are dynamic, changing over the course of the project. If you're using a requirements management tool, define a requirement attribute for priority. Requirements attributes are discussed in Chapter 18 and requirements management tools in Chapter 21.

3.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 stimuli correspond to the initial dialog steps of use cases or to external system events.

3.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 feature's services or to perform a use case. Describe how the product should respond to anticipated error conditions and to invalid inputs and actions. Uniquely label each functional requirement.

4. External Interface Requirements

According to Richard Thayer (2002), "External interface requirements specify hardware, software, or database elements with which a system or component must interface...." This section provides information to ensure that the system will communicate properly with external components. If different portions of the product have different external interfaces, incorporate an instance of this section within the detailed requirements for each such portion.

Reaching agreement on external and internal system interfaces has been identified as a software industry best practice (Brown 1996). Place detailed descriptions of the data and control components of the interfaces in the data dictionary. A complex system with multiple subcomponents should use a separate interface specification or system architecture specification (Hooks and Farry 2001). The interface documentation could incorporate material from other documents by reference. For instance, it could point to a separate application programming interface (API) specification or to a hardware device manual that lists the error codes that the device could send to the software.

Interface Wars

True Stories Two software teams collaborated to build the A. Datum Corporation's flagship product. The knowledge base team built a complex inference engine in C++ and the applications team implemented the user interface in Microsoft Visual Basic. The two subsystems communicated through an API. Unfortunately, the knowledge base team periodically modified the API, with the consequence that the system would not build and execute correctly. The applications team needed several hours to diagnose each problem they discovered and determine the root cause as being an API change. These changes were not agreed upon, were not communicated to all stakeholders, and were not coordinated with corresponding modifications in the Visual Basic code. The interfaces glue your system components—including the users—together, so document the

interface details and synchronize necessary modifications through your project's change-control process.

4.1 User Interfaces

Describe the logical characteristics of each user interface that the system needs. Some possible items to include are

- References to GUI standards or product family style guides that are to be followed.
- Standards for fonts, icons, button labels, images, color schemes, field tabbing sequences, commonly used controls, and the like.
- Screen layout or resolution constraints.
- Standard buttons, functions, or navigation links that will appear on every screen, such as a help button.
- Shortcut keys.
- Message display conventions.
- Layout standards to facilitate software localization.
- Accommodations for visually impaired users.

Document the user interface design details, such as specific dialog box layouts, in a separate user interface specification, not in the SRS. Including screen mock-ups in the SRS to communicate another view of the requirements is helpful, but make it clear that the mock-ups are not the committed screen designs. If the SRS is specifying an enhancement to an existing system, it sometimes makes sense to include screen displays exactly as they are to be implemented. The developers are already constrained by the current reality of the existing system, so it's possible to know up front just what the modified, and perhaps the new, screens should look like.

4.2 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 data and control interactions between the software and the hardware, and the communication protocols to be used.

4.3 Software Interfaces

Describe the connections between this product and other software components (identified by name and version), including databases, operating systems, tools, libraries, and integrated commercial components. State the purpose of the messages, data, and control items exchanged between the software components. Describe the services needed by external software components 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, specify this as a constraint.

4.4 Communications Interfaces

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

5. Other Nonfunctional Requirements

This section specifies nonfunctional requirements other than external interface requirements, which appear in section 4, and constraints, recorded in section 2.5.

5.1 Performance Requirements

State specific performance requirements for various system operations. Explain their rationale to guide the developers in making appropriate design choices. For instance, stringent database response time demands might lead the designers to mirror the database in multiple geographical locations or to denormalize relational database tables for faster query responses. Specify the number of transactions per second to be supported, response times, computational accuracy, and timing relationships for real-time systems. You could also specify memory and disk space requirements, concurrent user loads, or the maximum number of rows stored in database tables. If different functional requirements or features have different performance requirements, it's appropriate to specify those performance goals right with the corresponding functional requirements, rather than collecting them all in this one section.

Quantify the performance requirements as specifically as possible—for example, "95 percent of catalog database queries shall be completed within 3 seconds on a single-user 1.1-GHz Intel Pentium 4 PC running Microsoft Windows XP with at least 60 percent of the system resources free." An excellent method for precisely specifying performance requirements is Tom Gilb's Planguage, described in Chapter 12, "Beyond Functionality: Software Quality Attributes."

5.2 Safety Requirements

Safety and security are examples of quality attributes, which are more fully addressed in section 5.4. I've called these two attributes out in separate sections of the SRS template because if they are important at all, they are usually critical. In this section, specify those requirements that are concerned with possible loss, damage, or harm that could result from the use of the product (Leveson 1995). 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. Examples of safety requirements are

- **SA-1** The system shall terminate any operation within 1 second if the measured tank pressure exceeds 95 percent of the specified maximum pressure.
- **SA-2** The radiation beam shield shall remain open only through continuous computer control. The shield shall automatically fall into place if computer control is lost for any reason.

5.3 Security Requirements

Specify any requirements regarding security, integrity, or privacy issues that affect access to the product, use of the product, and protection of data that the product uses or creates. Security requirements normally originate in business rules, so identify any security or privacy policies or regulations to which the product must conform. Alternatively, you could address these requirements through the quality attribute called *integrity*. Following are sample security requirements:

- **SE-1** Every user must change his initially assigned login password immediately after his first successful login. The initial password may never be reused.
- **SE-2** A door unlock that results from a successful security badge read shall keep the door unlocked for 8.0 seconds.

5.4 Software Quality Attributes

State any additional product quality characteristics that will be important to either customers or developers. (See Chapter 12.) These characteristics should be specific, quantitative, and verifiable. Indicate the relative priorities of various attributes, such as ease of use over ease of learning, or portability over efficiency. A rich specification notation such as Planguage clarifies the needed levels of each quality much better than can simple descriptive statements.

6. Other Requirements

Define any other requirements that are not covered elsewhere in the SRS. Examples include internationalization requirements (currency, date formatting, language, international regulations, and cultural and political issues) and 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. Omit this section if all your requirements are accommodated in other sections.

Appendix A: Glossary

Define any specialized terms that a reader needs to know to properly interpret the SRS, including acronyms and abbreviations. Spell out each acronym and provide its definition. Consider building an enterprise-level glossary that spans multiple projects. Each SRS would then define only those terms that are specific to an individual project.

Appendix B: Analysis Models

This optional section includes or points to pertinent analysis models such as data flow diagrams, class diagrams, state-transition diagrams, or entity-relationship diagrams. (See Chapter 11, "A Picture Is Worth 1024 Words.")

Appendix C: Issues List

This is a dynamic list of the open requirements issues that remain to be resolved. Issues could include items flagged as TBD, pending decisions, information that is needed, conflicts awaiting resolution, and the like. This doesn't necessarily have to be part of the SRS, but some organizations always attach a TBD list to the SRS. Actively manage these issues to closure so that they don't impede the timely baselining of a high-quality SRS.

PREVIOUS NEXT
PREVIOUS NEXT

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 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 sys provide input to this system, must other software be insta before your software can run, or does your system rely o 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 people to use.
fast, rapid	Specify the minimum acceptable speed at which the syst 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	The list of items should include all possibilities. Otherwi

on, etc., such as	can't be used for design or testing.
maximize, minimize, optimize	State the maximum and minimum acceptable values of s parameter.
normally, ideally	Also describe the system's behavior under abnormal or n ideal conditions.
optionally	Clarify whether this means a system choice, a user choic a developer choice.
reasonable, when necessary, where appropriate	Explain how to make this judgment.
robust	Define how the system is to handle exceptions and respo 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 th 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 th 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.

True Stories 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

PREVIOUS NEXT
PREVIOUS NEXT

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.

PREVIOUS NEXT
PREVIOUS NEXT

The Data Dictionary

True Stories Long ago I led a project on which the three programmers sometimes inadvertently used different variable names, lengths, and validation criteria for the same data item. This caused confusion about the real definition, corrupted data, and maintenance headaches. We suffered from the lack of a *data dictionary*—a shared repository that defines the meaning, data type, length, format, necessary precision, and allowed range or list of values for all data elements or attributes used in the application.

The information in the data dictionary binds together the various requirements representations. Integration problems are reduced if all developers respect the data dictionary definitions. Begin collecting data definitions as you encounter them during requirements elicitation. The data dictionary complements the project glossary, which defines individual terms. You might want to merge the glossary entries into the data dictionary, although I prefer to keep them separate. The data dictionary can be managed as an appendix to the SRS or as a separate document or file.

Compared with sprinkling data definitions throughout the functional requirements, a separate data dictionary makes it easy to find the information you need, and it avoids redundancy and inconsistencies. Some commercial analysis and design tools include a data dictionary component. If you set up the data dictionary manually, consider using a hypertext approach. Clicking on a data item that is part of a data structure definition would take you to the definition of that individual data item, thereby making it easy to traverse the hierarchical tree of definitions.

Items in the data dictionary are represented using a simple notation (DeMarco 1979; Robertson and Robertson 1994). The item being defined 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, optional data items, iteration (repeats) of a data item, and a list of values for a data item. The following examples are from the Chemical Tracking System (of course!).

Primitive data elements A primitive data element is one for which no further decomposition is possible or necessary. The definition identifies the primitive's data type, size, range of allowed values, and other pertinent attributes. Primitives typically are defined with a comment, which is any text delimited by asterisks:

```
Request ID = * a 6-digit 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
                   + Number of Containers
                   + Grade
                   + Amount
                   + Amount Units
                   + (Vendor)
```

This structure identifies all the information associated with a request for a specific chemical. *Vendor* is optional because the person placing the request might not care which vendor supplies the chemical. Each data item that appears in a structure must itself be defined 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. Show the allowed number of possible repeats in the form *minimum:maximum* in front of the opening brace:

```
Request = Request ID
        + Request Date
        + Charge Number
        + 1:10{Requested Chemical}
```

This example shows that a chemical request must contain at least one chemical but may not contain more than 10 chemicals. Each request has the additional attributes of an identifier, the date the request was created, and a charge number to be used for payment.

Selection A data element that can take on a limited number of discrete values is called an *enumerated primitive*. Show the possible values in a list contained within square brackets, with vertical bars separating the list items:

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

This entry shows that there are just four allowed values for the text string *Quantity Units*. The comment inside the asterisks provides the textual definition of the data element.

The time you invest in creating a data dictionary will be more than repaid by the time you save avoiding mistakes that result when project participants have different understandings of key data. If you keep the data dictionary current, it will remain a valuable tool throughout the system's maintenance life and when developing related systems.

Next Steps

- Examine a page of functional requirements from your project's SRS to see whether each statement exhibits the characteristics of excellent requirements. Rewrite any requirements that don't measure up.
- If your organization doesn't already have a standard SRS template, convene a small working group to adopt one. Begin with the template in Figure 10-1 and adapt it to best meet the needs of your organization's projects and products. Agree on a convention for labeling individual requirements.

- Convene three to six project stakeholders to inspect the SRS for your project (Wiegers 2002a). Make sure each requirement demonstrates the desirable characteristics discussed in Chapter 1. Look for conflicts between different requirements in the specification, for missing requirements, and for missing sections of the SRS. Ensure that the defects you find are corrected in the SRS and in any downstream work products based on those requirements.

- Take a complex data object from one of your applications and define it using the data dictionary notation presented in this chapter.

Chapter 11: A Picture Is Worth 1024 Words

Overview

The Chemical Tracking System project team was holding its first SRS review. The participants were Dave (project manager), Lori (requirements analyst), Helen (lead developer), Ramesh (test lead), Tim (produ to the same system response. If the user isn't ct champion for the chemists), and Roxanne (product champion for the chemical stockroom staff). Tim began by saying, "I read the whole SRS. Most of the requirements seemed okay to me, but I had a hard time following parts of it. I'm not sure whether we identified all the steps in the chemical request process."

"It was hard for me to think of all the test cases I'll need to cover the status changes for a request," Ramesh added. "I found a bunch of requirements sprinkled throughout the SRS about the status changes, but I couldn't tell whether any were missing. A couple of requirements seemed to conflict."

Roxanne had a similar problem. "I got confused when I read about the way I would actually request a chemical," she said. "The individual functional requirements made sense, but I had trouble visualizing the sequence of steps I would go through."

After the reviewers raised several other concerns, Lori concluded, "It looks like this SRS doesn't tell us everything we need to know about the system. I'll draw some pictures to help us visualize the requirements and see whether that clarifies these problem areas. Thanks for the feedback."

According to requirements authority Alan Davis, no single view of the requirements provides a complete understanding (Davis 1995). You need a combination of textual and visual requirements representations to paint a full picture of the intended system. Requirements views include functional requirements lists and tables, graphical analysis models, user interface prototypes, test cases, decision trees, and decision tables. Ideally, different people will create various requirements representations. The analyst might write the functional requirements and draw some models, whereas the user interface designer builds a prototype and the test lead writes test cases. Comparing the representations created by different people through diverse thought processes reveals inconsistencies, ambiguities, assumptions, and omissions that are difficult to spot from any single view.

Diagrams communicate certain types of information more efficiently than text can. Pictures help bridge language and vocabulary barriers between different team members. The analyst will need to explain to other stakeholders the purpose of the models and the notations used. This chapter introduces several requirements modeling techniques, with illustrations and pointers to other sources for further details.

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).

True Stories 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."

PREVIOUS NEXT
PREVIOUS NEXT

Data Flow Diagram

The *data flow diagram* is the basic tool of structured analysis (DeMarco 1979; Robertson and Robertson 1994). A DFD identifies the transformational processes of a system, the collections (stores) of data or material that the

system manipulates, and the flows of data or material between processes, stores, and the outside world. Data flow modeling takes a functional decomposition approach to systems analysis, breaking complex problems into progressive levels of detail. This works well for transaction-processing systems and other function-intensive applications. Through the addition of control flow elements, the DFD technique has been extended to permit modeling of real-time systems (Hatley, Hruschka, and Pirbhai 2000).

The DFD provides a way to represent the steps involved in a business process or the operations of a proposed software system. I've often used this tool when interviewing customers, scribbling a DFD on a whiteboard while we discuss how the user's business operates. Data flow diagrams can represent systems over a wide range of abstraction. High-level DFDs provide a holistic, bird's-eye view of the data and processing components in a multistep activity, which complements the precise, detailed view embodied in the SRS. A DFD illustrates how the functional requirements in the SRS combine to let the user perform specific tasks, such as requesting a chemical.

Trap Don't assume that customers already know how to read analysis models, but don't conclude that they're unable to understand them, either. Explain the purpose and notations of each model to your product champions and ask them to review the diagrams.

The context diagram in Figure 5-3 is the highest level of abstraction of the DFD. The context diagram represents the entire system as a single black-box process, depicted as a circle (a *bubble*). It also shows the *terminators,* or external entities, that connect to the system and the data or material flows between the system and the terminators. Flows on the context diagram often represent complex data structures, which are defined in the data dictionary.

You can elaborate the context diagram into the level 0 DFD, which partitions the system into its major processes. Figure 11-1 shows the level 0 DFD for the Chemical Tracking System (somewhat simplified). The single process bubble that represented the entire Chemical Tracking System on the context diagram has been subdivided into seven major processes (the bubbles). As with the context diagram, the terminators are shown in rectangles. All data flows (arrows) from the context diagram also appear on the level 0 DFD. In addition, the level 0 diagram contains several *data stores*, depicted as a pair of parallel horizontal lines, which are internal to the system and therefore do not appear on the context diagram. A flow from a bubble to a store indicates that data is being placed into the store, a flow out of the store shows a read operation, and a bidirectional arrow between a store and a bubble indicates an update operation.

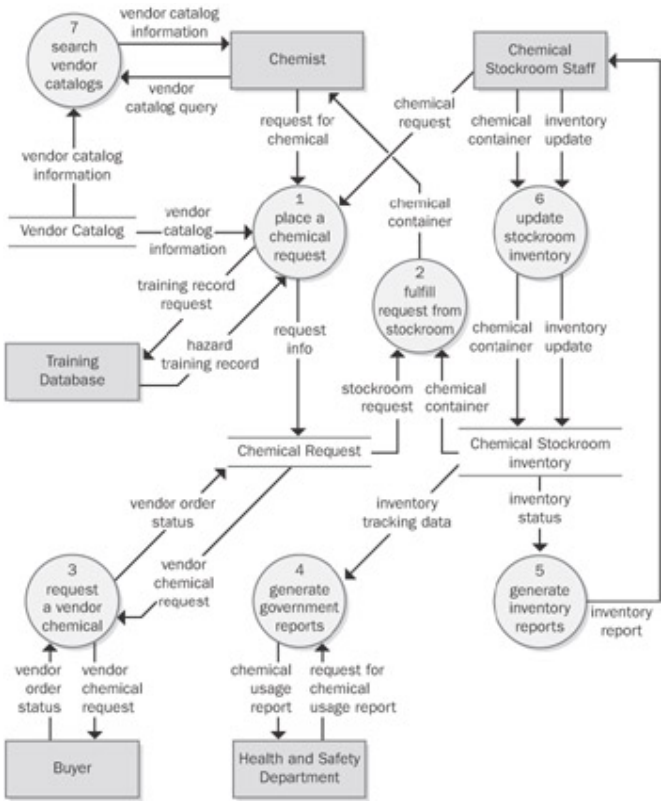


Figure 11-1: Level 0 data flow diagram for the Chemical Tracking System.

Each process that appears as a separate bubble on the level 0 diagram can be further expanded into a separate DFD to reveal more detail about its functioning. The analyst continues this progressive refinement until the lowest-level diagrams contain only primitive process operations that can be clearly represented in narrative text, pseudocode, a flowchart, or an activity diagram. The functional requirements in the SRS will define precisely what happens within each primitive process. Every level of the DFD must be balanced and consistent with the level above it so that all the input and output flows on the child diagram match up with flows on its parent. Complex data flows on the high-level diagrams can be split into their constituent elements, as defined in the data dictionary, on the lower-level DFDs.

Figure 11-1 looks complex at first glance. However, if you examine the immediate environment of any one process, you will see the data items that it consumes and produces and their sources and destinations. To see exactly how a process uses the data items, you'll need to either draw a more detailed child DFD or refer to the functional requirements for that part of the system.

Following are several conventions for drawing data flow diagrams. Not everyone adheres to the same conventions (for example, some analysts show terminators only on the context diagram), but I find them helpful. Using the models to enhance communication among the project participants is more important than dogmatic conformance to these principles.

- Place data stores only on the level 0 DFD and lower levels, not on the context diagram.
- Processes communicate through data stores, not by direct flows from one process to another. Similarly, data cannot flow directly from one store to another; it must pass through a process bubble.
- Don't attempt to imply the processing sequence using the DFD.
- Name each process as a concise action: verb plus object (such as Generate Inventory Reports). Use names that are meaningful to the customers and pertinent to the business or problem domain.

- Number the processes uniquely and hierarchically. On the level 0 diagram, number each process with an integer. If you create a child DFD for process 3, number the processes in that child diagram 3.1, 3.2, and so on.
- Don't show more than eight to ten processes on a single diagram or it becomes difficult to draw, change, and understand. If you have more processes, introduce another layer of abstraction by grouping related processes into a higher-level process.
- Bubbles with flows that are only coming in or only going out are suspect. The processing that a DFD bubble represents normally requires both input and output flows.

When customer representatives review a DFD, they should make sure that all the known processes are represented and that processes have no missing or unnecessary inputs or outputs. DFD reviews often reveal previously unrecognized user classes, business processes, and connections to other systems.

Entity-Relationship Diagram

Just as a data flow diagram illustrates the processes that take place in a system, a data model depicts the system's data relationships. A commonly used data model is the *entity-relationship diagram*, or ERD (Wieringa 1996). If your ERD represents logical groups of information from the problem domain and their interconnections, you're using the ERD as a requirements analysis tool. An analysis ERD helps you understand and communicate the data components of the business or system, without implying that the product will necessarily include a database. In contrast, when you create an ERD during system design, you're defining the logical or physical structure of the system's database.

Entities are physical items (including people) or aggregations of data items that are important to the business you're analyzing or to the system you intend to build (Robertson and Robertson 1994). Entities are named as singular nouns and are shown in rectangles in an ERD. Figure 11-2 illustrates a portion of the entity-relationship diagram for the Chemical Tracking System, using one of several common ERD modeling notations. Notice that the entities named Chemical Request, Vendor Catalog, and Chemical Stockroom Inventory appeared as data stores in the data flow diagram in Figure 11-1. Other entities represent actors who interact with the system (Requester), physical items that are part of the business operations (Chemical Container), and blocks of data that weren't shown in the level 0 DFD but would appear on a lower-level DFD (Container History, Chemical).

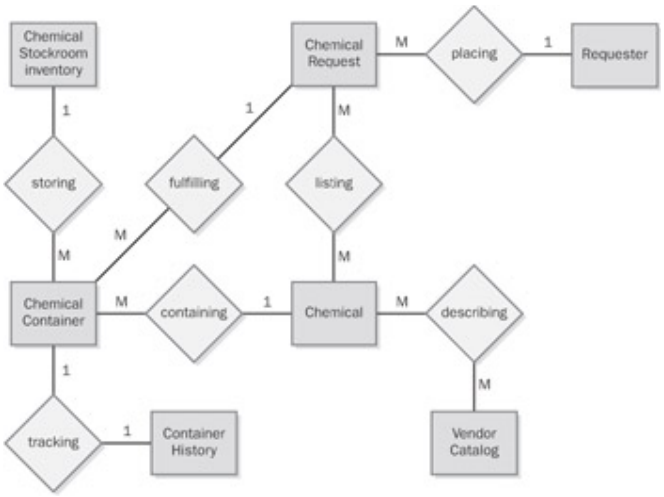


Figure 11-2: Partial entity-relationship diagram for the Chemical Tracking System.

Each entity is described by several attributes; individual instances of an entity will have different attribute values. For example, the attributes for each chemical include a unique chemical identifier, its formal chemical name, and a graphical representation of its chemical structure. The data dictionary contains the detailed definitions of those attributes, which guarantees that entities in the ERD and their corresponding data stores from the DFD are defined identically.

The diamonds in the ERD represent *relationships*, which identify the logical and numeric linkages between pairs of entities. Relationships are named in a way that describes the nature of the connections. For example, the relationship between the Requester and the Chemical Request is a *placing* relationship. You can read the relationship as either "a Requester places a Chemical Request" or as "a Chemical Request is placed by a Requester." Some conventions would have you label the relationship diamond "is placed by," which makes sense only if you read the diagram from left to right. When you ask customers to review an ERD, ask them to check whether the relationships shown are all correct and appropriate. Also ask them to identify any possible relationships with entities that the model doesn't show.

The *cardinality*, or multiplicity, of each relationship is shown with a number or letter on the lines that connect entities and relationships. Different ERD notations use different conventions to represent cardinality; the example in Figure 11-2 illustrates one common approach. Because each Requester can place multiple requests, there's a one-to-many relationship between Requester and Chemical Request. This cardinality is shown with a *1* on the line connecting Requester and the *placing* relationship and an *M* (for many) on the line connecting Chemical Request and the *placing* relationship. Other possible cardinalities are as follows:

- One-to-one (every Chemical Container is tracked by a single Container History)
- Many-to-many (every Vendor Catalog contains many Chemicals, and some Chemicals appear in multiple Vendor Catalogs)

If you know that a more precise cardinality exists than simply *many*, you can show the specific number or range of numbers instead of the generic *M*.

Modeling Problems, Not Software

True Stories I once served as the IT representative on a team that was doing some business process reengineering. Our goal was to reduce by a factor of 10 the time that it took to make a new chemical available for use in a product. The reengineering team included the following representatives of the various functions involved in chemical commercialization:

- The synthetic chemist who first makes the new chemical
- The scale-up chemist who develops a process for making large batches of the chemical
- The analytical chemist who devises techniques for analyzing the chemical's purity
- The patent attorney who applies for patent protection
- The health and safety representative who obtains government approval to use the chemical in consumer products

After we invented a new process that we believed would greatly accelerate the chemical commercialization activity, I interviewed the person on the reengineering team who was responsible for each process step. I asked each owner two questions: "What information do you need to perform this step?" and "What information does this step produce that we should store?" By correlating the answers for all process steps, I found steps that needed data that no one had available. Other steps produced data that no one needed. We fixed all those problems.

Next, I drew a data flow diagram to illustrate the new chemical commercialization process and an entity-relationship diagram to model the data relationships. A data dictionary defined all our data items. These analysis models served as useful communication tools to help the team members arrive at a common understanding of the new process. The models would also be a valuable starting point to scope and specify the requirements for software applications that supported portions of the process.

State-Transition Diagram

All software systems involve a combination of functional behavior, data manipulation, and state changes. Real-time systems and process control applications can exist in one of a limited number of states at any given time. A state change can take place only when well-defined criteria are satisfied, such as receiving a specific input stimulus under certain conditions. An example is a highway intersection that incorporates vehicle sensors, protected turn lanes, and pedestrian crosswalk buttons and signals. Many information systems deal with business objects—sales orders, invoices, inventory items, and the like—with life cycles that involve a series of possible statuses. System elements that involve a set of states and changes between them can be regarded as *finite-state machines* (Booch, Rumbaugh, Jacobson 1999).

Describing a complex finite-state machine in natural language creates a high probability of overlooking a permitted state change or including a disallowed change. Depending on how the SRS is organized, requirements that pertain to the state machine's behavior might be sprinkled throughout it. This makes it difficult to reach an overall understanding of the system's behavior.

The *state-transition diagram* provides a concise, complete, and unambiguous representation of a finite-state machine. A related technique is the *statechart diagram* included in the Unified Modeling Language (UML), which has a somewhat richer set of notations and which models the states an object goes through during its lifetime (Booch, Rumbaugh, Jacobson 1999). The STD contains three types of elements:

- Possible system states, shown as rectangles.
- Allowed state changes or *transitions*, shown as arrows connecting pairs of rectangles.

- Events or conditions that cause each transition to take place, shown as text labels on each transition arrow. The label might identify both the event and the corresponding system response.

Figure 11-3 illustrates a portion of an STD for a home security system. The STD for a real-time system includes a special state usually called *Idle* (equivalent to *Disarmed* in the figure), to which the system returns whenever it isn't doing other processing. In contrast, the STD for an object that passes through a defined life cycle, such as a chemical request, will have one or more termination states, which represent the final status values that an object can have.

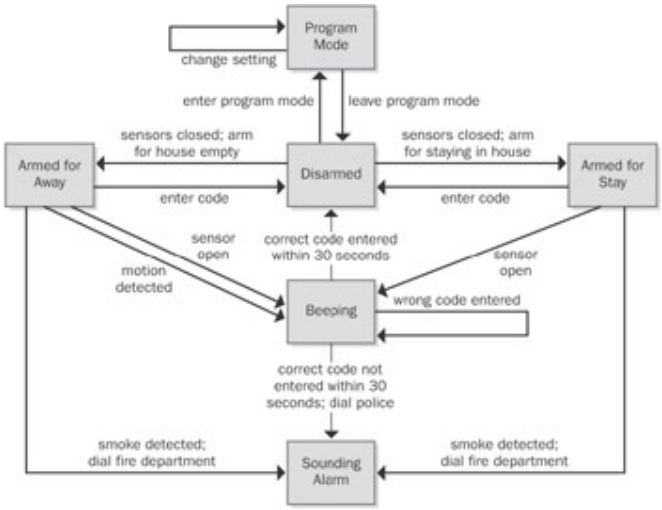


Figure 11-3: Partial state-transition diagram for a home security system.

The STD doesn't show the details of the processing that the system performs; it shows only the possible state changes that result from that processing. An STD helps developers understand the intended behavior of the system. It's a good way to check whether all the required states and transitions have been correctly and completely described in the functional requirements. Testers can derive test cases from the STD that cover all allowed transition paths. Customers can read an STD with just a little coaching about the notation—it's just boxes and arrows.

Recall from Chapter 8 that a primary function of the Chemical Tracking System is to permit actors called Requesters to place requests for chemicals, which can be fulfilled either from the chemical stockroom's inventory or by placing orders to outside vendors. Each request will pass through a series of states between the time it's created and the time it's either fulfilled or canceled (the two termination states). Thus, we can treat the life cycle of a chemical request as a finite-state machine and model it as shown in Figure 11-4.

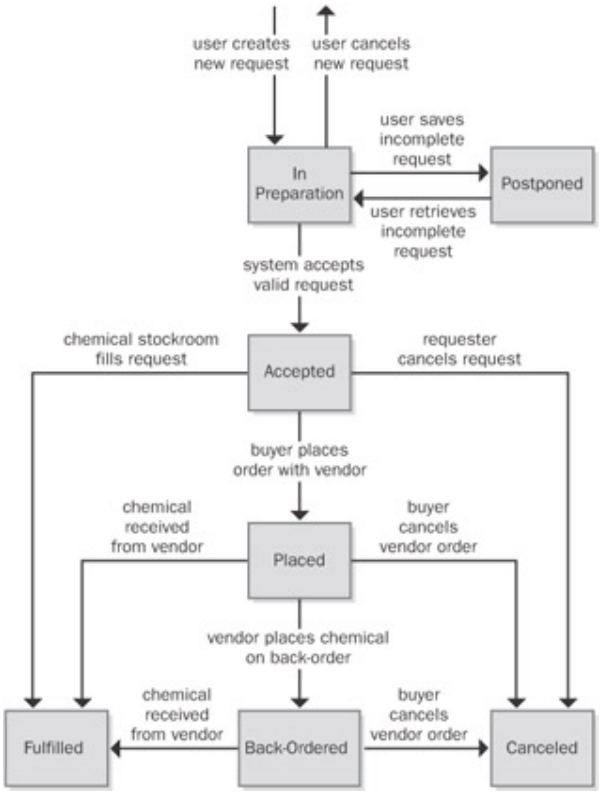


Figure 11-4: State-transition diagram for a chemical request in the Chemical Tracking System.

This STD shows that an individual request can take on one of the following seven possible states:

- **In Preparation** The Requester is creating a new request, having initiated that function from some other part of the system.
- **Postponed** The Requester saved a partial request for future completion without either submitting the request to the system or canceling the request operation.
- **Accepted** The user submitted a completed chemical request and the system accepted it for processing.
- **Placed** The request must be satisfied by an outside vendor and a buyer has placed an order with the vendor.
- **Fulfilled** The request has been satisfied, either by delivering a chemical container from the chemical stockroom to the Requester or by receiving a chemical from a vendor.
- **Back-Ordered** The vendor didn't have the chemical available and notified the buyer that it was back-ordered for future delivery.
- **Canceled** The Requester canceled an accepted request before it was fulfilled or the buyer canceled a vendor order before it was fulfilled or while it was back-ordered.

True Stories When the Chemical Tracking System user representatives reviewed the initial chemical request STD, they identified one state that wasn't needed, saw that another essential state was missing, and pointed out two incorrect transitions. No one had seen those errors when they reviewed the corresponding functional requirements. This underscores the value of representing requirements information at more than one level of abstraction. It's often easier to spot a problem when you step back from the detailed level and see the big picture that an analysis model provides. However, the STD doesn't provide enough detail for a developer to

know what software to build. Therefore, the SRS for the Chemical Tracking System included the functional requirements associated with processing a chemical request and its possible state changes.

Dialog Map

A user interface also can be regarded as a finite-state machine. Only one dialog element (such as a menu, workspace, dialog box, line prompt, or touch screen display) is available at any given time for user input. The user can navigate to certain other dialog elements based on the action he takes at the active input location. The number of possible navigation pathways can be large in a complex graphical user interface, but the number is finite and the options usually are known. Therefore, many user interfaces can be modeled with a form of state-transition diagram called a *dialog map* (Wasserman 1985; Wiegers 1996a). Constantine and Lockwood (1999) describe a similar technique called a *navigation map*, which includes a richer set of notations for representing different types of interaction elements and context transitions.

The dialog map represents a user interface design at a high level of abstraction. It shows the dialog elements in the system and the navigation links among them, but it doesn't show the detailed screen designs. A dialog map allows you to explore hypothetical user interface concepts based on your understanding of the requirements. Users and developers can study a dialog map to reach a common vision of how the user might interact with the system to perform a task. Dialog maps are also useful for modeling the visual architecture of a Web site. Navigation links that you build into the Web site appear as transitions on the dialog map. Dialog maps are related to system storyboards, which also include a short description of each screen's purpose (Leffingwell and Widrig 2000).

Dialog maps capture the essence of the user–system interactions and task flow without bogging the team down in detailed screen layouts. Users can trace through a dialog map to find missing, incorrect, or unnecessary transitions, and hence missing, incorrect, or unnecessary requirements. The abstract, conceptual dialog map formulated during requirements analysis serves as a guide during detailed user interface design.

Just as in ordinary state-transition diagrams, the dialog map shows each dialog element as a state (rectangle) and each allowed navigation option as a transition (arrow). The condition that triggers a user interface navigation is shown as a text label on the transition arrow. There are several types of trigger conditions:

- A user action, such as pressing a function key or clicking a hyperlink or a dialog box button
- A data value, such as an invalid user input that triggers an error message display
- A system condition, such as detecting that a printer is out of paper
- Some combination of these, such as typing a menu option number and pressing the Enter key

Dialog maps look a bit like flowcharts but they serve a different purpose. A flowchart explicitly shows the processing steps and decision points, but not the user interface displays. In contrast, the dialog map does *not* show the processing that takes place along the transition lines that connect one dialog element to another. The branching decisions (usually user choices) are hidden behind the display screens shown as rectangles on the dialog map, and the conditions that lead to displaying one screen or another appear in the labels on the transitions. You can think of the dialog map as a sort of negative image of—or a complement to—a flowchart.

To simplify the dialog map, omit global functions such as pressing the F1 key to bring up a help display from every dialog element. The SRS section on user interfaces should specify that this functionality will be

available, but showing lots of help-screen boxes on the dialog map clutters the model while adding little value. Similarly, when modeling a Web site, you needn't include standard navigation links that will appear on every page in the site. You can also omit the transitions that reverse the flow of a Web page navigation sequence because the Web browser's Back button handles that navigation.

True Stories A dialog map is an excellent way to represent the interactions between an actor and the system that a use case describes. The dialog map can depict alternative courses as branches off the normal course flow. I found that sketching dialog map fragments on a whiteboard was helpful during use-case elicitation workshops in which a team explored the sequence of actor actions and system responses that would lead to task completion.

Chapter 8 presented a use case for the Chemical Tracking System called "Request a Chemical." The normal course for this use case involved requesting a chemical container from the chemical stockroom's inventory. An alternative course was to request the chemical from a vendor. The user placing the request wanted the option to view the history of the available stockroom containers of that chemical before selecting one. Figure 11-5 shows a dialog map for this fairly complex use case.

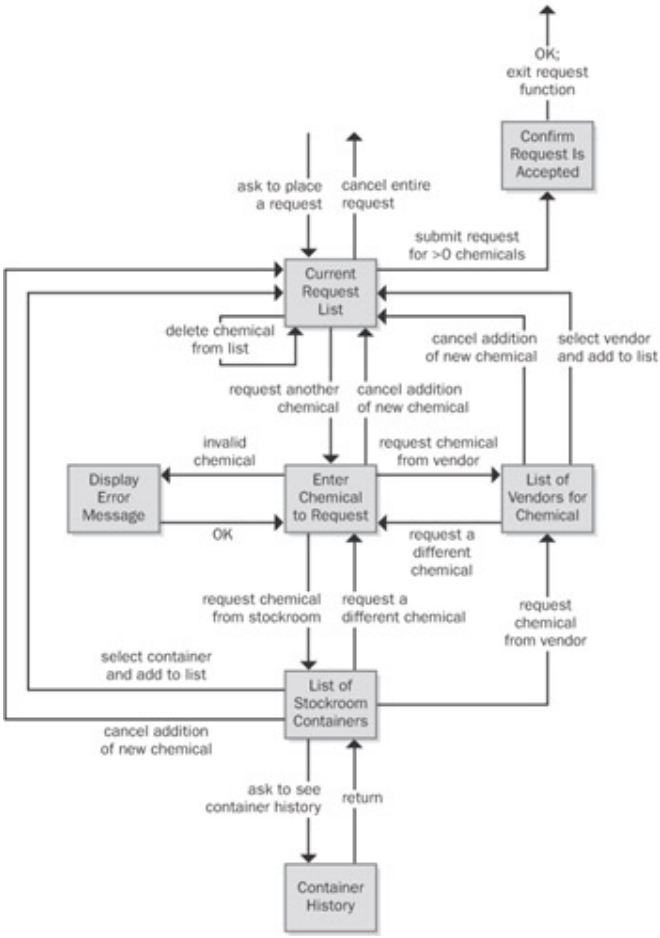


Figure 11-5: Dialog map for the "Request a Chemical" use case from the Chemical Tracking System.

This diagram might look complicated at first, but if you trace through it one line and one box at a time, it's not difficult to understand. The user initiates this use case by asking to place a request for a chemical from some menu in the Chemical Tracking System. In the dialog map, this action brings the user to the box called Current Request List, along the arrow in the upper-left part of the dialog map. That box represents the main workspace for this use case, a list of the chemicals in the user's current request. The arrows leaving that box on the dialog map show all the navigation options—and hence functionality—available to the user in that context:

- Cancel the entire request.
- Submit the request if it contains at least one chemical.
- Add a new chemical to the request list.
- Delete a chemical from the list.

The last operation, deleting a chemical, doesn't involve another dialog element; it simply refreshes the current request list display after the user makes the change.

As you trace through this dialog map, you'll see elements that reflect the rest of the "Request a Chemical" use case:

- One flow path for requesting a chemical from a vendor
- Another path for fulfillment from the chemical stockroom
- An optional path to view the history of a container in the chemical stockroom
- An error message display to handle entry of an invalid chemical identifier or other error conditions that could arise

Some of the transitions on the dialog map allow the user to back out of operations. Users get annoyed if they are forced to complete a task even though they change their minds partway through it. The dialog map lets you enhance usability by designing in those back-out and cancel options at strategic points.

A user who reviews this dialog map might spot a missing requirement. For example, a cautious user might want to confirm the operation that leads to canceling an entire request to avoid inadvertently losing data. It costs less to add this new function at the analysis stage than to build it into a completed product. Because the dialog map represents just the conceptual view of the possible elements involved in the interaction between the user and the system, don't try to pin down all the user interface design details at the requirements stage. Instead, use these models to help the project stakeholders reach a common understanding of the system's intended functionality.

PREVIOUS NEXT
PREVIOUS NEXT

Class Diagrams

Object-oriented software development has superseded structured analysis and design on many projects, spawning object-oriented analysis and design. *Objects* typically correspond to real-world items in the business or problem domain. They represent individual instances derived from a generic template called a *class*. Class descriptions encompass both attributes (data) and the operations that can be performed on the attributes. A *class diagram* is a graphical way to depict the classes identified during object-oriented analysis and the relationships among them.

Products developed using object-oriented methods don't demand unique requirements development approaches. This is because requirements development focuses on what the users need to do with the system and the functionality it must contain, not with how it will be constructed. Users don't care about objects or classes. However, if you know that you're going to build the system using object-oriented techniques, it can be helpful to begin identifying domain classes and their attributes and behaviors during requirements analysis.

This facilitates the transition from analysis to design, as the designer maps the problem-domain objects into the system's objects and further details each class's attributes and operations.

The standard object-oriented modeling language is the Unified Modeling Language (Booch, Rumbaugh, and Jacobson 1999). At the level of abstraction that's appropriate for requirements analysis, you can use the UML notation to draw class diagrams, as illustrated in Figure 11-6 for a portion of—you guessed it—the Chemical Tracking System. A designer can elaborate these conceptual class diagrams, which are free from implementation specifics, into more detailed class diagrams for object-oriented design and implementation. Interactions among the classes and the messages they exchange can be shown using sequence diagrams and collaboration diagrams, which are not addressed further in this book; see Booch, Rumbaugh, and Jacobson (1999) and Lauesen (2002).

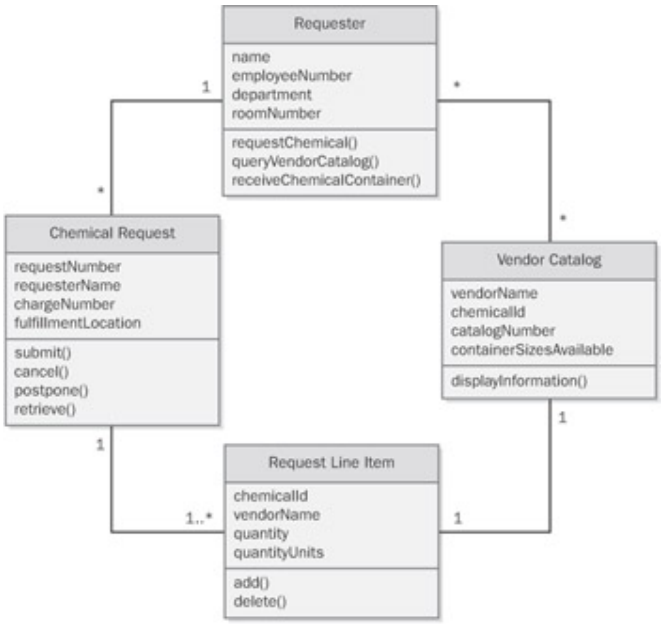


Figure 11-6: Class diagram for part of the Chemical Tracking System.

Figure 11-6 contains four classes, each shown in a large box: Requester, Vendor Catalog, Chemical Request, and Request Line Item. There are some similarities between the information in this class diagram and that shown in the other analysis models presented in this chapter (no surprise there—all the diagrams are representing the same problem). The Requester appeared in the entity-relationship diagram in Figure 11-2, where it represented an actor role that could be played by a member of either the Chemist or Chemical Stockroom Staff user class. The data flow diagram in Figure 11-1 also showed that both of these user classes could place requests for chemicals. Don't confuse a *user class* with an *object class;* despite the similarity in names, there's no intentional connection.

The *attributes* associated with the Requester class are shown in the middle portion of its class box: name, employeeNumber, department, and roomNumber (this capitalization convention is commonly used with UML diagrams). Those are the properties or data items associated with each object that is a member of the Requester class. Similar attributes will appear in the definitions of the stores on a data flow diagram and in the data dictionary.

The *operations* are services that an object in the Requester class can perform. Operations are shown in the bottom portion of the class box and typically are followed by empty parentheses. In a class diagram that represents a design, these operations will correspond to the class's functions or methods, and the function arguments often appear in the parentheses. The analysis class model simply has to show that a Requester can request chemicals, query vendor catalogs, and receive chemical containers. The operations shown in the class diagram correspond roughly to the processes that appear in bubbles on low-level data flow diagrams.

The lines that connect the class boxes in Figure 11-6 represent associations between the classes. The numbers on the lines show the multiplicity of the association, just as the numbers on lines in the entity-relationship diagram show the multiplicity relationships between entities. In Figure 11-6, an asterisk indicates a one-to-many association between a Requester and a Chemical Request: one requester can place many requests, but each request belongs to just one requester.

Decision Tables and Decision Trees

A software system is often governed by complex logic, with various combinations of conditions leading to different system behaviors. For example, if the driver presses the accelerate button on a car's cruise control system and the car is currently cruising, the system increases the car's speed, but if the car isn't cruising, the input is ignored. The SRS needs functional requirements that describe what the system should do under all possible combinations of conditions. However, it's easy to overlook a condition, which results in a missing requirement. These gaps are hard to spot by manually reviewing a textual specification.

Decision tables and decision trees are two alternative techniques for representing what the system should do when complex logic and decisions come into play (Davis 1993). The *decision table* lists the various values for all the factors that influence the behavior and indicates the expected system action in response to each combination of factors. The factors can be shown either as statements with possible conditions of true and false or as questions with possible answers of yes and no. Of course, you can also use decision tables with factors that can have more than two possible values.

Trap Don't create both a decision table and a decision tree to show the same information; either one will suffice.

Table 11-2 shows a decision table with the logic that governs whether the Chemical Tracking System should accept or reject a request for a new chemical. Four factors influence this decision:

- Whether the user who is creating the request is authorized to do so
- Whether the chemical is available either in the chemical stockroom or from a vendor
- Whether the chemical is on the list of hazardous chemicals that require special training in safe handling
- Whether the user who is creating the request has been trained in handling this type of hazardous chemical

Each of these four factors has two possible conditions, true or false. In principle, this gives rise to 2^4, or 16, possible true/false combinations, for a potential of 16 distinct functional requirements. In practice, though, many of the combinations lead to the same system response. If the user isn't authorized to request chemicals, then the system won't accept the request, so the other conditions are irrelevant (shown as a dash in a cell in the decision table). The table shows that only five distinct functional requirements arise from the various logic combinations.

Table 11-2: Sample Decision Table for the Chemical Tracking System

	Requirement Number				
Condition	**1**	**2**	**3**	**4**	**5**

User is authorized	F	T	T	T	T
Chemical is available	—	F	T	T	T
Chemical is hazardous	—	—	F	T	T
Requester is trained	—	—	—	F	T
Action					
Accept request			X		X
Reject request	X	X		X	

Figure 11-7 shows a decision tree that represents this same logic. The five boxes indicate the five possible outcomes of either accepting or rejecting the chemical request. Both decision tables and decision trees are useful ways to document requirements (or business rules) to avoid overlooking any combinations of conditions. Even a complex decision table or tree is easier to read than a mass of repetitious textual requirements.

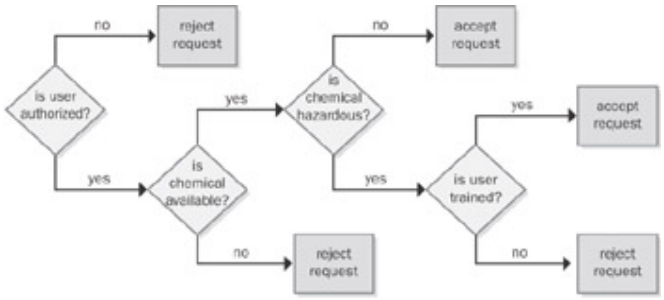


Figure 11-7: Sample decision tree for the Chemical Tracking System.

A Final Reminder

Each of the modeling techniques described in this chapter has its advantages and its limitations. They overlap in the views they provide, so you won't need to create every kind of diagram for your project. For instance, if you create an ERD and a data dictionary, you might not want to create a class diagram (or vice versa). Keep in mind that you draw analysis models to provide a level of understanding and communication that goes beyond what a textual SRS or any other single view of the requirements can provide. Avoid getting caught up in the dogmatic mindsets and religious wars that sometimes take place in the world of software development methods and models. Instead, use what you need to best explain your system's requirements.

Next Steps

- Practice using the modeling techniques described in this chapter by documenting the design of an existing system. For example, draw a dialog map for an automated teller machine or for a Web site that you use.
- Identify a portion of your SRS that has proven difficult for readers to understand or in which defects have been found. Choose an analysis model described in this chapter that's appropriate for representing that portion of the requirements. Draw the model and assess whether it would have been helpful had you created it earlier.
- The next time you need to document some requirements, select a modeling technique that complements the textual description. Sketch the model on paper or a whiteboard once or twice to make sure you're on the right track, and then use a commercial CASE tool that supports the model notation you're using.

Chapter 12: Beyond Functionality – Software Quality Attributes

Overview

"*Hi, Phil, this is Maria again. I have a question about the new employee system you programmed. As you know, this system runs on our mainframe, and every department has to pay for its disk storage and CPU processing charges every month. It looks like the new system's files are using about twice as much disk space as the old system did. Even worse, the CPU charges are nearly three times what they used to be for a session. Can you tell me what's going on, please?"*

"Sure, Maria," said Phil. "Remember that you wanted this system to store much more data about each employee than the old system did, so naturally the database is much larger. Therefore, you're paying more for disk space usage each month. Also, you and the other product champions requested that the new system be much easier to use than the old one, so we designed that nice graphical user interface. However, the GUI consumes a lot more computer horsepower than the old system's simple character-mode display. So that's why your per-session processing charges are so much higher. The new system's a lot easier to use, isn't it?"

"Well, yes, it is," Maria replied, "but I didn't realize it would be so expensive to run. I could get in trouble for this. My manager's getting nervous. At this rate, he'll burn through his whole year's computing budget by April. Can you fix the system so that it costs less to run?"

Phil was frustrated. "There's really nothing to fix. The new employee system is just what you said you needed. I assumed you'd realize that if you store more data or do more work with the computer, it costs more. Maybe we should have talked about this earlier because we can't do much about it now. Sorry."

Users naturally focus on specifying their functional, or behavioral, requirements—the things the software will let them do—but there's more to software success than just delivering the right functionality. Users also have expectations about *how well* the product will work. Characteristics that fall into this category include how easy it is to use, how quickly it runs, how often it fails, and how it handles unexpected conditions. Such characteristics, collectively known as *software quality attributes* or *quality factors*, are part of the system's nonfunctional (also called nonbehavioral) requirements.

Quality attributes are difficult to define, yet often they distinguish a product that merely does what it's supposed to from one that delights its customers. As Robert Charette (1990) pointed out, "In real systems, meeting the nonfunctional requirements often is more important than meeting the functional requirements in the determination of a system's perceived success or failure." Excellent software products reflect an optimum balance of competing quality characteristics. If you don't explore the customers' quality expectations during requirements elicitation, you're just lucky if the product satisfies them. Disappointed users and frustrated developers are the more typical outcome.

From a technical perspective, quality attributes drive significant architectural and design decisions, such as partitioning system functions onto various computers to achieve performance or integrity objectives. It's far more difficult and costly to rearchitect a completed system to achieve essential quality goals than to design for them at the outset.

Customers generally don't present their quality expectations explicitly, although the information they provide during elicitation supplies some clues about what they have in mind. The trick is to pin down just what the users are thinking when they say the software must be user-friendly, fast, reliable, or robust. Quality, in its many dimensions, must be defined both by the customers and by those who will build, test, and maintain the software. Questions that explore the customers' implicit expectations can lead to quality goal statements and design criteria that help the developers create a fully satisfactory product.

Quality Attributes

Several dozen product characteristics can be called quality attributes (Charette 1990), although most projects need to carefully consider only a handful of them. If developers know which of these characteristics are most crucial to project success, they can select architecture, design, and programming approaches to achieve the specified quality goals (Glass 1992; DeGrace and Stahl 1993). Quality attributes have been classified according to various schemes (Boehm, Brown, and Lipow 1976; Cavano and McCall 1978; IEEE 1992; DeGrace and Stahl 1993). One way to classify attributes distinguishes those characteristics that are discernible at run time from those that aren't (Bass, Clements, and Kazman 1998). Another approach is to separate the visible characteristics that are primarily important to the users from under-the-hood qualities that are primarily significant to technical staff. The latter indirectly contribute to customer satisfaction by making the product easier to change, correct, verify, and migrate to new platforms.

Table 12-1 lists several quality attributes in both categories that every project should consider. Some attributes are critical to embedded systems (efficiency and reliability), whereas others might be especially pertinent to Internet and mainframe applications (availability, integrity, and maintainability) or to desktop systems (interoperability and usability). Embedded systems often have additional significant quality attributes, including safety (which was discussed in Chapter 10), installability, and serviceability. Scalability is another attribute that's important to Internet applications.

Table 12-1: Software Quality Attributes

Important Primarily to Users	Important Primarily to Developers
Availability	Maintainability
Efficiency	Portability
Flexibility	Reusability
Integrity	Testability
Interoperability	
Reliability	
Robustness	
Usability	

In an ideal universe, every system would exhibit the maximum possible value for all its attributes. The system would be available at all times, would never fail, would supply instantaneous results that are always correct, and would be intuitively obvious to use. Because nirvana is unattainable, you have to learn which attributes from Table 12-1 are most important to your project's success. Then define the user and developer goals in terms of these essential attributes so that designers can make appropriate choices.

Different parts of the product need different combinations of quality attributes. Efficiency might be critical for certain components, while usability is paramount for others. Differentiate quality characteristics that apply to

the entire product from those that are specific to certain components, certain user classes, or particular usage situations. Document any global quality goals in section 5.4 of the SRS template presented in Chapter 10, and associate specific goals with individual features, use cases, or functional requirements.

Defining Quality Attributes

Most users won't know how to answer questions such as "What are your interoperability requirements?" or "How reliable does the software have to be?" On the Chemical Tracking System, the analysts developed several prompting questions based on each attribute that they thought might be significant. For example, to explore integrity they asked, "How important is it to prevent users from viewing orders they didn't place?" or "Should everyone be able to search the stockroom inventory?" They asked the user representatives to rank each attribute on a scale of 1 (don't give it another thought) to 5 (critically important). The responses helped the analysts to determine which attributes were most important. Different user classes sometimes had different quality preferences, so the favored user classes got the nod whenever conflicts arose.

The analysts then worked with users to craft specific, measurable, and verifiable requirements for each attribute (Robertson and Robertson 1997). If the quality goals are not verifiable, you can't tell whether you've achieved them. Where appropriate, indicate the scale or units of measure for each attribute and the target, minimum, and maximum values. The notation called Planguage, which is described later in this chapter, helps with this specification. If you can't quantify all the important quality attributes, at least define their priorities and customer preferences. *The IEEE Standard for a Software Quality Metrics Methodology* presents an approach for defining software quality requirements in the context of an overall quality metrics framework (IEEE 1992).

Trap Don't neglect stakeholders such as maintenance programmers when exploring quality attributes.

Consider asking users what would constitute *unacceptable* performance, usability, integrity, or reliability. That is, specify system properties that would violate the user's quality expectations, such as allowing an unauthorized user to delete files (Voas 1999). By defining unacceptable characteristics—a kind of inverse requirement—you can devise tests that attempt to force the system to demonstrate those characteristics. If you can't force them, you've probably achieved your attribute goals. This approach is particularly valuable for safety-critical applications, in which a system that violates reliability or performance tolerances poses a risk to life or limb.

The remainder of this section briefly describes each of the quality attributes in Table 12-1 and presents some sample quality attributes (a bit simplified) from various projects. Soren Lauesen (2002) provides many excellent examples of quality requirements.

Attributes Important to Users

Users rightfully consider the quality attributes described in the following sections important.

Availability Availability is a measure of the planned *up time* during which the system is actually available for use and fully operational. More formally, availability equals the mean time to failure (MTTF) for the system divided by the sum of the MTTF and the mean time to repair the system after a failure is encountered. Scheduled maintenance periods also affect availability. Some authors view availability as encompassing reliability, maintainability, and integrity (Gilb 1988).

Certain tasks are more time-critical than others are, and users become frustrated—even irate—when they need to get essential work done and the system isn't available. Ask users what percentage of up time is really needed

and whether there are any times for which availability is imperative to meet business or safety objectives. Availability requirements become more complex and more important for Web sites or global applications with worldwide users. An availability requirement might read like this:

> *AV-1. The system shall be at least 99.5 percent available on weekdays between 6:00 a.m. and midnight local time, and at least 99.95 percent available on weekdays between 4:00 p.m. and 6:00 p.m. local time.*

As with many of the examples presented here, this requirement is somewhat simplified. It doesn't define the level of performance that constitutes being *available*. Is the system considered available if only one person can use it on the network in a degraded mode? Write quality requirements to make them measurable and to establish a precise agreement on expectations between the requirements analyst, the development team, and the customers.

The Cost of Quality

True Stories Beware of specifying 100 percent of a quality attribute such as reliability or availability because it will be impossible to achieve and expensive to strive for. One company had an availability requirement for its shop floor manufacturing systems of 100 percent, 24 hours a day, 365 days a year. To try to achieve such stringent availability, the company installed two independent computer systems so that software upgrades could be installed on the machine that wasn't running at the moment. This was an expensive solution to the availability requirement, but it was cheaper than *not* manufacturing their highly profitable product.

Efficiency Efficiency is a measure of how well the system utilizes processor capacity, disk space, memory, or communication bandwidth (Davis 1993). Efficiency is related to performance, another class of nonfunctional requirement, which is discussed later in this chapter. If a system consumes too much of the available resources, users will encounter degraded performance, a visible indication of inefficiency. Poor performance is an irritant to the user who is waiting for a database query to display results. But performance problems can also represent serious risks to safety, such as when a real-time process control system is overloaded. Consider minimum hardware configurations when defining efficiency, capacity, and performance goals. To allow engineering margins for unanticipated conditions and future growth, you might specify something like the following:

> *EF-1. At least 25 percent of the processor capacity and RAM available to the application shall be unused at the planned peak load conditions.*

Typical users won't state efficiency requirements in such technical terms. They'll think primarily in terms of response times or disk space consumption. The analyst must ask the questions that will surface user expectations regarding issues such as acceptable performance degradation, demand spikes, and anticipated growth.

I Haven't Got All Day

True Stories A major corporation once designed an elaborate graphical store metaphor for its e-business component. The customer could enter the Web site's store, browse various services, and buy a variety of products. The graphics were stunning, the metaphor was sound, and the performance was appalling. The user interface worked fine for the developers working over a high-speed Internet connection to local servers. Unfortunately, the standard 14.4 or 28.8 KBps modem connection that customers used back then resulted in a painfully slow download of the huge image files. Beta testers invariably lost interest before the main page was fully displayed. In their enthusiasm about the graphical metaphor, the development team hadn't considered operating environment constraints, efficiency, or performance requirements. The whole approach was abandoned after it was completed, an expensive lesson in the importance of discussing software quality

attributes early in the project.

Trus Stories **Flexibility** Also known as *extensibility, augmentability, extendability*, and *expandability*, flexibility measures how easy it is to add new capabilities to the product. If developers anticipate making many enhancements, they can choose design approaches that maximize the software's flexibility. This attribute is essential for products that are developed in an incremental or iterative fashion through a series of successive releases or by evolutionary prototyping. A project I worked on set the following flexibility goal:

> *FL-1. A maintenance programmer who has at least six months of experience supporting this product shall be able to make a new hardcopy output device available to the product, including code modifications and testing, with no more than one hour of labor.*

The project isn't a failure if it takes a programmer 75 minutes to install a new printer, so the requirement has some latitude. If we hadn't specified this requirement, though, the developers might have made design choices that made installing a new device to the system very time-consuming. Write quality requirements in a way that makes them measurable.

Integrity Integrity—which encompasses security, as discussed in Chapter 10—deals with blocking unauthorized access to system functions, preventing information loss, ensuring that the software is protected from virus infection, and protecting the privacy and safety of data entered into the system. Integrity is a major issue with Internet software. Users of e-commerce systems want their credit card information to be secure. Web surfers don't want personal information or a record of the sites they visit to be used inappropriately, and providers want to protect against denial-of-service or hacking attacks. Integrity requirements have no tolerance for error. State integrity requirements in unambiguous terms: user identity verification, user privilege levels, access restrictions, or the precise data that must be protected. A sample integrity requirement is the following:

> *IN-1. Only users who have Auditor access privileges shall be able to view customer transaction histories.*

As with many integrity requirements, this one is also a constraining business rule. It's a good idea to know the rationale for your quality attribute requirements and to trace them back to specific origins, such as a management policy. Avoid stating integrity requirements in the form of design constraints. Password requirements for access control are a good example. The real requirement is to restrict access to the system to authorized users; passwords are merely one way (albeit the most commonly used way) to accomplish that objective. Depending on which user authentication technique is chosen, this underlying integrity requirement will lead to specific functional requirements that implement the system's authentication functions.

Interoperability Interoperability indicates how easily the system can exchange data or services with other systems. To assess interoperability, you need to know which other applications the users will employ in conjunction with your product and what data they expect to exchange. Users of the Chemical Tracking System were accustomed to drawing chemical structures with several commercial tools, so they presented the following interoperability requirement:

> *IO-1. The Chemical Tracking System shall be able to import any valid chemical structure from the ChemiDraw (version 2.3 or earlier) and Chem-Struct (version 5 or earlier) tools.*

You could also state this requirement as an external interface requirement and define the standard file formats that the Chemical Tracking System can import. Alternatively, you could define several functional requirements dealing with the import operation. Sometimes, though, thinking about the system from the perspective of quality attributes reveals previously unstated, implicit requirements. The customers hadn't stated this need when discussing external interfaces or system functionality. As soon as the analyst asked about other systems to which the Chemical Tracking System had to connect, though, the product champion immediately mentioned

the two chemical structure drawing packages.

Reliability The probability of the software executing without failure for a specific period of time is known as reliability (Musa, Iannino, and Okumoto 1987). Robustness is sometimes considered an aspect of reliability. Ways to measure software reliability include the percentage of operations that are completed correctly and the average length of time the system runs before failing. Establish quantitative reliability requirements based on how severe the impact would be if a failure occurred and whether the cost of maximizing reliability is justifiable. Systems that require high reliability should also be designed for high testability to make it easier to find defects that could compromise reliability.

True Stories My team once wrote some software to control laboratory equipment that performed daylong experiments using scarce, expensive chemicals. The users required the software component that actually ran the experiments to be highly reliable. Other system functions, such as logging temperature data periodically, were less critical. A reliability requirement for this system was

> *RE-1. No more than five experimental runs out of 1000 can be lost because of software failures.*

True Stories **Robustness** A customer once told a company that builds measurement devices that its next product should be "built like a tank." The developing company therefore adopted, slightly tongue-in-cheek, the new quality attribute of "tankness." Tankness is a colloquial way of saying *robustness*, which is sometimes called *fault tolerance*. Robustness is the degree to which a system continues to function properly when confronted with invalid inputs, defects in connected software or hardware components, or unexpected operating conditions. Robust software recovers gracefully from problem situations and is forgiving of user mistakes. When eliciting robustness requirements, ask users about error conditions the system might encounter and how the system should react. One example of a robustness requirement is

> *RO-1. If the editor fails before the user saves the file, the editor shall be able to recover all changes made in the file being edited up to one minute prior to the failure the next time the same user starts the program.*

True Stories Several years ago I led a project to develop a reusable software component called the Graphics Engine, which interpreted data files that defined graphical plots and rendered the plots on a designated output device (Wiegers 1996b). Several applications that needed to generate plots invoked the Graphics Engine. Because the developers had no control over the data that these applications fed into the Graphics Engine, robustness was an essential quality. One of our robustness requirements was

> *RO-2. All plot description parameters shall have default values specified, which the Graphics Engine shall use if a parameter's input data is missing or invalid.*

With this requirement, the program wouldn't crash if, for example, an application requested a color that the plotter being used couldn't generate. The Graphics Engine would use the default color of black and continue executing. This would still constitute a product failure because the end user didn't get the desired color. But designing for robustness reduced the severity of the failure from a program crash to generating an incorrect color, an example of fault tolerance.

Usability Also referred to as *ease of use* and *human engineering*, usability addresses the myriad factors that constitute what users often describe as *user-friendliness*. Analysts and developers shouldn't talk about friendly software but about software that's designed for effective and unobtrusive usage. Several books have been published recently on designing usable software systems; for examples see Constantine and Lockwood (1999) and Nielsen (2000). Usability measures the effort required to prepare input for, operate, and interpret the output of the product.

The Chemical Tracking System requirements analysts asked their user representatives questions such as "How important is it that you be able to request chemicals quickly and simply?" and "How long should it take you to

complete a chemical request ?" These are simple starting points toward defining the many characteristics that will make the software easy to use. Discussions about usability can lead to measurable goals (Lauesen 2002) such as this one:

> *US-1. A trained user shall be able to submit a complete request for a chemical selected from a vendor catalog in an average of four and a maximum of six minutes.*

Inquire whether the new system must conform to any user interface standards or conventions, or whether its user interface needs to be consistent with those of other frequently used systems. You might state such a usability requirement in the following way:

> *US-2. All functions on the File menu shall have shortcut keys defined that use the Control key pressed simultaneously with one other key. Menu commands that also appear on the Microsoft Word XP File menu shall use the same shortcut keys that Word uses.*

Usability also encompasses how easy it is for new or infrequent users to learn to use the product. Ease-of-learning goals can be quantified and measured:

> *US-3. A chemist who has never used the Chemical Tracking System before shall be able to place a request for a chemical correctly with no more than 30 minutes of orientation.*

Attributes Important to Developers

The following sections describe quality attributes that are particularly important to software developers and maintainers.

Maintainability Maintainability indicates how easy it is to correct a defect or modify the software. Maintainability depends on how easily the software can be understood, changed, and tested. It is closely related to flexibility and testability. High maintainability is critical for products that will undergo frequent revision and for products that are being built quickly (and are perhaps cutting corners on quality). You can measure maintainability in terms of the average time required to fix a problem and the percentage of fixes that are made correctly. The Chemical Tracking System included the following maintainability requirement:

> *MA-1. A maintenance programmer shall be able to modify existing reports to conform to revised chemical-reporting regulations from the federal government with 20 labor hours or less of development effort.*

On the Graphics Engine project, we knew we would be doing frequent software surgery to satisfy evolving user needs. We specified design criteria such as the following to guide developers in writing the code to enhance the program's maintainability:

> *MA-2. Function calls shall not be nested more than two levels deep.*

> *MA-3. Each software module shall have a ratio of nonblank comments to source code statements of at least 0.5.*

State such design goals carefully to discourage developers from taking silly actions that conform to the letter, but not the intent, of the goal. Work with maintenance programmers to understand what properties of the code would make it easy for them to modify it or correct defects.

Hardware devices with embedded software often have maintainability requirements. Some of these lead to software design choices, whereas others influence the hardware design. An example of the latter is the following requirement:

MA-4. The printer design shall permit a certified repair technician to replace the printhead cable in no more than 10 minutes, the ribbon sensor in no more than 5 minutes, and the ribbon motor in no more than 5 minutes.

Portability The effort required to migrate a piece of software from one operating environment to another is a measure of portability. Some practitioners include the ability to internationalize and localize a product under the heading of portability. The design approaches that make software portable are similar to those that make it reusable (Glass 1992). Portability is typically either immaterial or critical to project success. Portability goals should identify those portions of the product that must be movable to other environments and describe those target environments. Developers can then select design and coding approaches that will enhance the product's portability appropriately.

For example, some compilers define an *integer* as being 16 bits long and others define it as 32 bits. To satisfy a portability requirement, a programmer might symbolically define a data type called *WORD* as a 16-bit unsigned integer and use the WORD data type instead of the compiler's default integer data type. This ensures that all compilers will treat data items of type WORD in the same way, which helps to make the system work predictably in different operating environments.

Reusability A long-sought goal of software development, reusability indicates the relative effort involved to convert a software component for use in other applications. Developing reusable software costs considerably more than creating a component that you intend to use in just one application. Reusable software must be modular, well documented, independent of a specific application and operating environment, and somewhat generic in capability. Reusability goals are difficult to quantify. Specify which elements of the new system need to be constructed in a manner that facilitates their reuse, or stipulate the libraries of reusable components that should be created as a spin-off from the project, such as:

RU-1. The chemical structure input functions shall be designed to be reusable at the object code level in other applications that use the international standard chemical structure representations.

Testability Also known as *verifiability*, testability refers to the ease with which software components or the integrated product can be tested to look for defects. Designing for testability is critical if the product has complex algorithms and logic, or if it contains subtle functionality interrelationships. Testability is also important if the product will be modified often because it will undergo frequent regression testing to determine whether the changes damaged any existing functionality.

Because my team and I knew that we'd have to test the Graphics Engine many times while it was repeatedly enhanced, we included the following design guideline in the SRS to enhance testability:

TE-1. The maximum cyclomatic complexity of a module shall not exceed 20.

Cyclomatic complexity is a measure of the number of logic branches in a source code module (McCabe 1982). Adding more branches and loops to a module makes it harder to test, to understand, and to maintain. The project wasn't going to be a failure if some module had a cyclomatic complexity of 24, but documenting such design guidelines helped the developers achieve a desired quality objective. Had we not stated that design guideline (presented here in the form of a quality requirement), the developers might not have considered cyclomatic complexity when writing their programs. This could have resulted in a program with complex code that was nearly impossible to test thoroughly, difficult to extend, and a nightmare to debug.

Performance Requirements

Performance requirements define how well or how rapidly the system must perform specific functions. Performance requirements encompass speed (database response times, for instance), throughput (transactions per second), capacity (concurrent usage loads), and timing (hard real-time demands). Stringent performance requirements seriously affect software design strategies and hardware choices, so define performance goals that

are appropriate for the operating environment. All users want their applications to run instantly, but the real performance requirements will be different for the spell-check feature of a word processing program and a radar guidance system for a missile. Performance requirements should also address how the system's performance will degrade in an overloaded situation, such as when a 911 emergency telephone system is flooded with calls. Following are some simple performance requirement examples:

PE-1. The temperature control cycle must execute completely in 80 milliseconds.

PE-2. The interpreter shall parse at least 5000 error-free statements per minute.

PE-3. Every Web page shall download in 15 seconds or less over a 50 KBps modem connection.

PE-4. Authorization of an ATM withdrawal request shall not take more than 10 seconds.

Trap Don't forget to consider how you'll evaluate a product to see whether it satisfies its quality attributes. Unverifiable quality requirements are no better than unverifiable functional requirements.

Defining Nonfunctional Requirements By Using Planguage

Some of the quality attribute examples presented in this chapter are incomplete or nonspecific, a limitation of the attempt to state them in a concise sentence or two. You can't evaluate a product to judge whether it satisfies imprecisely worded quality requirements. In addition, simplistic quality and performance goals can be unrealistic. Specifying a maximum response time of two seconds for a database query might be fine for a simple lookup in a local database but impossible for a six-way join of relational tables residing on geographically separated servers.

To address the problem of ambiguous and incomplete nonfunctional requirements, consultant Tom Gilb (1988; 1997) has developed *Planguage*, a planning language with a rich set of keywords that permits precise statements of quality attributes and other project goals (Simmons 2001). Following is an example of how to express a performance requirement using just a few of the many Planguage keywords. This example is a Planguage version of a performance requirement from Chapter 10: "95 percent of catalog database queries shall be completed within 3 seconds on a single-user 1.1-GHz Intel Pentium 4 PC running Microsoft Windows XP with at least 60 percent of the system resources free."

TAG Performance.QueryResponseTime

AMBITION Fast response time to database queries on the base user platform.

SCALE Seconds of elapsed time between pressing the Enter key or clicking OK to submit a query and the beginning of the display of query results.

METER Stopwatch testing performed on 250 test queries that represent a defined usage operational profile.

MUST No more than 10 seconds for 98 percent of queries. <-- Field Support Manager

PLAN No more than three seconds for category one queries, eight seconds for all queries.

WISH No more than two seconds for all queries.

base user platform DEFINED 1.1-GHz Intel Pentium 4 processor, 128 MB RAM, Microsoft Windows XP,

QueryGen 3.3 running, single user, at least 60 percent of system resources free, no other applications running.

Each requirement receives a unique *tag,* or label. The *ambition* states the purpose or objective of the system that leads to this requirement. *Scale* defines the units of measurement and *meter* describes precisely how to make the measurements. All stakeholders need to have the same understanding of how to measure the performance. Suppose that a user interprets the measurement to be from the time that she presses the Enter key until the complete set of query results appears, rather than until the beginning of the result display, as stated in the example. The developer might claim that the requirement is satisfied while this user insists that it is not. Unambiguous quality requirements and measurements prevent these sorts of arguments.

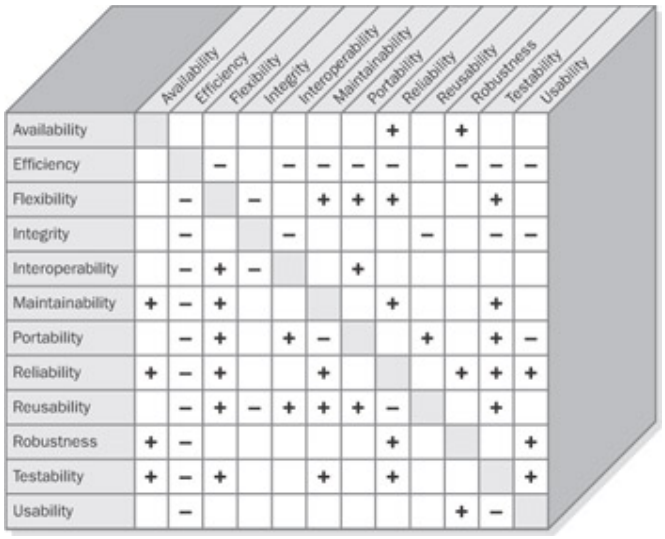
You can specify several target values for the quantity being measured. The *must* criterion is the minimum acceptable achievement level. The requirement isn't satisfied unless every *must* condition is completely satisfied, so the *must* condition should be justifiable in business terms. An alternative way to state the *must* requirement is to define the *fail* (another Planguage keyword) condition: "More than 10 seconds on more than 2 percent of all queries." The *plan* value is the nominal target, and the *wish* value represents the ideal outcome. Also, show the origin of performance goals; for instance, the preceding *must* criterion shows that it came from the Field Support Manager. Any specialized terms in the Planguage statement are *defined* to make them perfectly clear to the reader.

Planguage includes many additional keywords to permit great flexibility and precision in requirements specification. See *http://www.gilb.com* for the most recent information on the still-evolving Planguage vocabulary and syntax. Planguage provides a powerful ability to specify unambiguous quality attribute and performance requirements. Specifying multiple levels of achievement yields a far richer statement of a quality requirement than a simple black-and-white, yes-or-no construct can.

PREVIOUS NEXT
PREVIOUS NEXT

Attribute Trade-Offs

Certain attribute combinations have inescapable trade-offs. Users and developers must decide which attributes are more important than others, and they must respect those priorities consistently when they make decisions. Figure 12-1 illustrates some typical interrelationships among the quality attributes from Table 12-1, although you might encounter exceptions to these (Charette 1990; IEEE 1992; Glass 1992). A plus sign in a cell indicates that increasing the attribute in the corresponding row has a positive effect on the attribute in the column. For example, design approaches that increase a software component's portability also make the software more flexible, easier to connect to other software components, easier to reuse, and easier to test.



	Availability	Efficiency	Flexibility	Integrity	Interoperability	Maintainability	Portability	Reliability	Reusability	Robustness	Testability	Usability
Availability							+		+			
Efficiency			−		−	−	−	−		−	−	−
Flexibility		−		−		+	+	+			+	
Integrity		−			−			−			−	−
Interoperability		−	+	−			+					
Maintainability	+	−	+					+			+	
Portability		−	+		+	−			+		+	−
Reliability	+	−	+			+				+	+	+
Reusability		−	+	−	+	+	+	−			+	
Robustness	+	−						+				+
Testability	+	−	+			+		+				+
Usability		−							+	−		

Figure 12-1: Positive and negative relationships among selected quality attributes.

True Stories A minus sign in a cell means that increasing the attribute in that row adversely affects the attribute in the column. A blank cell indicates that the attribute in the row has little impact on the attribute in the column. Efficiency has a negative impact on most other attributes. If you write the tightest, fastest code you can, using coding tricks and relying on execution side effects, it's likely to be hard to maintain and enhance; in addition, it won't be easy to port to other platforms. Similarly, systems that optimize ease of use or that are designed to be flexible, reusable, and interoperable with other software or hardware components often incur a performance penalty. Using the general-purpose Graphics Engine component described earlier in the chapter to generate plots resulted in degraded performance compared with the old applications that incorporated custom graphics code. You have to balance the possible performance reductions against the anticipated benefits of your proposed solution to ensure that you're making sensible trade-offs.

The matrix in Figure 12-1 isn't symmetrical because the effect that increasing attribute A has on attribute B isn't necessarily the same as the effect that increasing B will have on A. For example, Figure 12-1 shows that designing the system to increase efficiency doesn't necessarily have any effect on integrity. However, increasing integrity likely will hurt efficiency because the system must go through more layers of user authentications, encryption, virus scanning, and data checkpointing.

To reach the optimum balance of product characteristics, you must identify, specify, and prioritize the pertinent quality attributes during requirements elicitation. As you define the important quality attributes for your project, use Figure 12-1 to avoid making commitments to conflicting goals. Following are some examples:

- Don't expect to maximize usability if the software must run on multiple platforms (portability).
- It's hard to completely test the integrity requirements of highly secure systems. Reused generic components or interconnections with other applications could compromise security mechanisms.
- Highly robust code will be less efficient because of the data validations and error checking that it performs.

As usual, overconstraining system expectations or defining conflicting requirements makes it impossible for the developers to fully satisfy the requirements.

PREVIOUS NEXT
PREVIOUS NEXT

Implementing Nonfunctional Requirements

The designers and programmers will have to determine the best way to satisfy each quality attribute and performance requirement. Although quality attributes are nonfunctional requirements, they can lead to derived functional requirements, design guidelines, or other types of technical information that will produce the desired quality characteristics. Table 12-2 indicates the likely categories of technical information that different types of quality attributes will generate. For example, a medical device with stringent availability requirements might include a backup battery power supply (architecture) and a functional requirement to visibly or audibly indicate that the product is operating on battery power. This translation from user-centric or developer-centric quality requirements into corresponding technical information is part of the requirements and high-level design process.

Table 12-2: Translating Quality Attributes into Technical Specifications

Quality Attribute Types	Likely Technical Information Category
Integrity, interoperability, robustness, usability, safety	Functional requirement
Availability, efficiency, flexibility, performance, reliability	System architecture
Interoperability, usability	Design constraint
Flexibility, maintainability, portability, reliability, reusability, testability, usability	Design guideline
Portability	Implementation constraint

Next Steps

- Identify several quality attributes from Table 12-1 that might be important to users on your current project. Formulate a few questions about each attribute that will help your users articulate their expectations. Based on the user responses, write one or two specific goals for each important attribute.
- Rewrite several of the quality attribute examples in this chapter by using Planguage, making assumptions when necessary for the sake of illustration. Can you state those quality requirements with more precision and less ambiguity using Planguage?
- Write one of your own quality attribute requirements using Planguage. Ask customer, development, and testing representatives to judge whether the Planguage version is more informative than the original version.
- Examine your users' quality expectations for the system for possible conflicts and resolve them. The favored user classes should have the most influence on making the necessary trade-off choices.
- Trace your quality attribute requirements into the functional requirements, design and implementation constraints, or architectural and design choices that implement them.

Chapter 13: Risk Reduction Through Prototyping

Overview

"*Sharon, today I'd like to talk with you about the requirements that the buyers in the Purchasing Department have for the new Chemical Tracking System," began Lori, the requirements analyst. "Can you tell me what you want the system to do?"*

"Wow, I'm not sure how to do this," replied Sharon with a puzzled expression. "I don't know how to describe what I need, but I'll know it when I see it."

IKIWISI—"I'll know it when I see it"—is a phrase that chills the blood of requirements analysts. It conjures an image of the development team having to make their best guess at the right software to build, only to have users tell them, "Nope, that's not right; try again." To be sure, envisioning a future software system and articulating its requirements is hard to do. Many people have difficulty describing their needs without having something tangible in front of them to contemplate, and critiquing is much easier than creating.

Software prototyping makes the requirements more real, brings use cases to life, and closes gaps in your understanding of the requirements. Prototyping puts a mock-up or an initial slice of a new system in front of users to stimulate their thinking and catalyze the requirements dialog. Early feedback on prototypes helps the stakeholders arrive at a shared understanding of the system's requirements, which reduces the risk of customer dissatisfaction.

Even if you apply the requirements development practices described in earlier chapters, portions of your requirements might still be uncertain or unclear to customers, developers, or both. If you don't correct these problems, an expectation gap between the user's vision of the product and the developer's understanding of what to build is guaranteed. It's hard to visualize exactly how software will behave by reading textual requirements or studying analysis models. Users are more willing to try out a prototype (which is fun) than to read an SRS (which is tedious). When you hear *IKIWISI* from your users, think about what you can provide that would help them articulate their needs (Boehm 2000). However, if no stakeholder really has an idea of what the developers should build, your project is doomed.

Prototype has multiple meanings, and participants in a prototyping activity can have significantly different expectations. A prototype airplane actually flies—it's the first instance of the real airplane. In contrast, a software prototype is only a portion or a model of a real system—it might not do anything useful at all. Software prototypes can be working models or static designs; highly detailed screens or quick sketches; visual displays or slices of actual functionality; simulations; or emulations (Constantine and Lockwood 1999; Stevens et al. 1998). This chapter describes different kinds of prototypes, how to use them during requirements development, and ways to make prototyping an effective part of your software engineering process (Wood and Kang 1992).

PREVIOUS NEXT
PREVIOUS NEXT

Prototyping: What and Why

A software prototype is a partial or possible implementation of a proposed new product. Prototypes serve three major purposes:

- **Clarify and complete the requirements.** Used as a requirements tool, the prototype is a preliminary implementation of a part of the system that's not well understood. User evaluation of the prototype points out problems with the requirements, which you can correct at low cost before you construct the actual product.

- **Explore design alternatives.** Used as a design tool, a prototype lets stakeholders explore different user interaction techniques, optimize system usability, and evaluate potential technical approaches. Prototypes

can demonstrate requirements feasibility through working designs.

- **Grow into the ultimate product.** Used as a construction tool, a prototype is a functional implementation of an initial subset of the product, which can be elaborated into the complete product through a sequence of small-scale development cycles.

The primary reason for creating a prototype is to resolve uncertainties early in the development process. Use these uncertainties to decide which parts of the system to prototype and what you hope to learn from the prototype evaluations. A prototype is useful for revealing and resolving ambiguity and incompleteness in the requirements. Users, managers, and other nontechnical stakeholders find that prototypes give them something concrete to contemplate while the product is being specified and designed. Prototypes, especially visual ones, are easier to understand than the technical jargon that developers sometimes use.

PREVIOUS NEXT
Team LiB
PREVIOUS NEXT

Horizontal Prototypes

When people say "software prototype," they are usually thinking about a *horizontal prototype* of a possible user interface. A horizontal prototype is also called a *behavioral prototype* or a *mock-up*. It is called *horizontal* because it doesn't dive into all the layers of an architecture or into system details but rather primarily depicts a portion of the user interface. This type of prototype lets you explore some specific behaviors of the intended system, with the goal of refining the requirements. The prototype helps users judge whether a system based on the prototype will let them get the job done.

Like a movie set, the horizontal prototype implies functionality without actually implementing it. It displays the facades of user interface screens and permits some navigation between them, but it contains little or no real functionality. Think of the typical Western movie: the cowboy walks into the saloon and then walks out of the livery stable, yet he doesn't have a drink and he doesn't see a horse because there's nothing behind the false fronts of the buildings.

Horizontal prototypes can demonstrate the functional options the user will have available, the look and feel of the user interface (colors, layout, graphics, controls), and the information architecture (navigation structure). The navigations might work, but at some points the user might see only a message that describes what would really be displayed. The information that appears in response to a database query could be faked or constant, and report contents are hardcoded. Try to use actual data in sample reports, charts, and tables to enhance the validity of the prototype as a model of the real system.

The horizontal prototype doesn't perform any useful work, although it looks as if it should. The simulation is often good enough to let the users judge whether any functionality is missing, wrong, or unnecessary. Some prototypes represent the developer's concept of how a specific use case might be implemented. The user's prototype evaluation can point out alternative courses for the use case, missing interaction steps, or additional exception conditions.

When working with a horizontal prototype, the user should focus on broad requirements and workflow issues without becoming distracted by the precise appearance of screen elements (Constantine 1998). Don't worry at this stage about exactly where the screen elements will be positioned, fonts, colors, graphics, or controls. The time to explore the specifics of user interface design is after you've clarified the requirements and determined the general structure of the interface.

PREVIOUS NEXT

Vertical Prototypes

A *vertical prototype*, also known as a *structural prototype* or *proof of concept*, implements a slice of application functionality from the user interface through the technical services layers. A vertical prototype works like the real system is supposed to work because it touches on all levels of the system implementation. Develop a vertical prototype when you're uncertain whether a proposed architectural approach is feasible and sound, or when you want to optimize algorithms, evaluate a proposed database schema, or test critical timing requirements. To make the results meaningful, vertical prototypes are constructed using production tools in a production-like operating environment. Vertical prototypes are used to explore critical interface and timing requirements and to reduce risk during design.

True Stories I once worked with a team that wanted to implement an unusual client/server architecture as part of a transitional strategy from a mainframe-centric world to an application environment based on networked UNIX servers and workstations (Thompson and Wiegers 1995). A vertical prototype that implemented just a bit of the user interface client (on a mainframe) and the corresponding server functionality (on a UNIX workstation) allowed us to evaluate the communication components, performance, and reliability of our proposed architecture. The experiment was a success, as was the implementation based on that architecture.

Throwaway Prototypes

Before constructing a prototype, make an explicit and well-communicated decision as to whether the prototype will be discarded after evaluation or become part of the delivered product. Build a *throwaway prototype* (or *exploratory prototype*) to answer questions, resolve uncertainties, and improve requirements quality (Davis 1993). Because you'll discard the prototype after it has served its purpose,[1] build it as quickly and cheaply as you can. The more effort you invest in the prototype, the more reluctant the project participants are to discard it.

When developers build a throwaway prototype, they ignore much of what they know about solid software construction techniques. A throwaway prototype emphasizes quick implementation and modification over robustness, reliability, performance, and long-term maintainability. For this reason, don't allow low-quality code from a throwaway prototype to migrate into a production system. Otherwise, the users and the maintainers will suffer the consequences for the life of the product.

The throwaway prototype is most appropriate when the team faces uncertainty, ambiguity, incompleteness, or vagueness in the requirements. Resolving these issues reduces the risk of proceeding with construction. A prototype that helps users and developers visualize how the requirements might be implemented can reveal gaps in the requirements. It also lets users judge whether the requirements will enable the necessary business processes.

Trap Don't make a throwaway prototype more elaborate than is necessary to meet the prototyping objectives. Resist the temptation—or the pressure from users—to keep adding more capabilities to the prototype.

Figure 13-1 shows a sequence of development activities that move from use cases to detailed user interface design with the help of a throwaway prototype. Each use-case description includes a sequence of actor actions and system responses, which you can model using a dialog map to depict a possible user interface architecture. A throwaway prototype elaborates the dialog elements into specific screens, menus, and dialog boxes. When

users evaluate the prototype, their feedback might lead to changes in the use-case descriptions (if, say, an alternative course is discovered) or to changes in the dialog map. Once the requirements are refined and the screens sketched out, each user interface element can be optimized for usability. This progressive refinement approach is cheaper than leaping directly from use-case descriptions to a complete user interface implementation and then discovering major problems with the requirements that necessitate extensive rework.

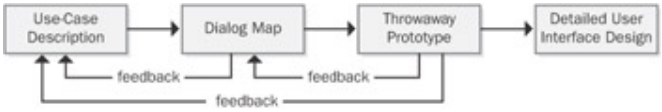


Figure 13-1: Activity sequence from use cases to user interface design using a throwaway prototype.

[1]You don't actually have to throw the prototype away if you see merit in keeping it for possible future reuse. However, it won't be incorporated into the deliverable product. For this reason, you might prefer to call it a *nonreleasable prototype*.

Evolutionary Prototypes

In contrast to a throwaway prototype, an *evolutionary prototype* provides a solid architectural foundation for building the product incrementally as the requirements become clear over time. Evolutionary prototyping is a component of the spiral software development life cycle model (Boehm 1988) and of some object-oriented software development processes (Kruchten 1996). In contrast to the quick-and-dirty nature of throwaway prototyping, an evolutionary prototype must be built with robust, production-quality code from the outset. Therefore, an evolutionary prototype takes longer to create than a throwaway prototype that simulates the same system capabilities. An evolutionary prototype must be designed for easy growth and frequent enhancement, so developers must emphasize software architecture and solid design principles. There's no room for shortcuts in the quality of an evolutionary prototype.

Think of the first increment of an evolutionary prototype as a pilot release that implements a well-understood and stable portion of the requirements. Lessons learned from user acceptance testing and initial usage lead to modifications in the next iteration. The full product is the culmination of a series of evolutionary prototypes. Such prototypes quickly get useful functionality into the hands of the users. Evolutionary prototypes work well for applications that you know will grow over time, such as projects that gradually integrate various information systems.

True Stories Evolutionary prototyping is well suited for Internet development projects. On one such project that I led, my team created a series of four prototypes, based on requirements that we developed from a use-case analysis. Several users evaluated each prototype, and we revised each one based on their responses to questions we posed. The revisions following the fourth prototype evaluation resulted in our production Web site.

Figure 13-2 illustrates several ways to combine the various types of prototyping. For example, you can use the knowledge gained from a series of throwaway prototypes to refine the requirements, which you might then implement incrementally through an evolutionary prototyping sequence. An alternative path through Figure 13-2 uses a throwaway horizontal prototype to clarify the requirements prior to finalizing the user interface design, while a concurrent vertical prototyping effort validates the architecture, application components, and core algorithms. What you *cannot* do successfully is turn the intentional low quality of a throwaway prototype into the maintainable robustness that a production system demands. In addition, working prototypes that appear to get the job done for a handful of concurrent users likely won't scale up to handle thousands of users without

major architectural changes. Table 13-1 summarizes some typical applications of throwaway, evolutionary, horizontal, and vertical prototypes.

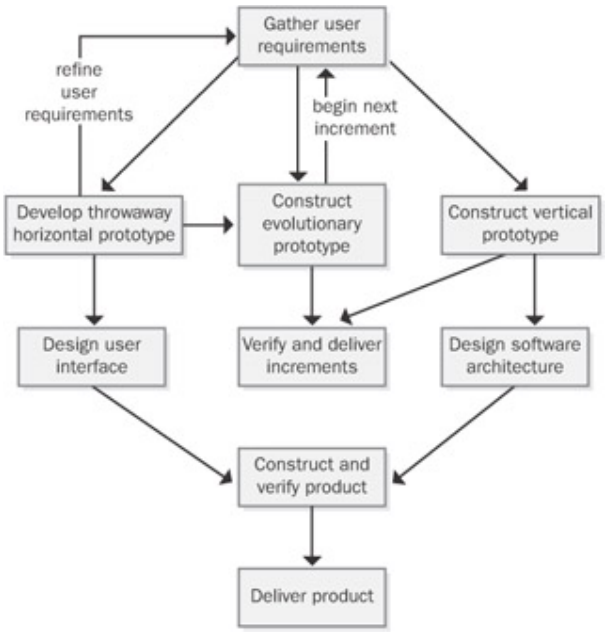


Figure 13-2: Several possible ways to incorporate prototyping into the software development process.

Table 13-1: Typical Applications of Software Prototypes

	Throwaway	**Evolutionary**
Horizontal	• Clarify and refine use cases and functional requirements. • Identify missing functionality. • Explore user interface approaches.	• Implement core use cases. • Implement additional use cases based on priority. • Implement and refine Web sites. • Adapt system to rapidly changing business needs.
Vertical	• Demonstrate technical feasibility.	• Implement and grow core client/server functionality and communication layers. • Implement and optimize core algorithms. • Test and tune performance.

Paper and Electronic Prototypes

You don't always need an executable prototype to resolve requirements uncertainties. A *paper prototype* (sometimes called a *lo-fi prototype*) is a cheap, fast, and low-tech way to explore what a portion of an implemented system might look like (Rettig 1994; Hohmann 1997). Paper prototypes help you test whether users and developers hold a shared understanding of the requirements. They let you take a tentative and low-risk step into a possible solution space prior to developing production code. A similar technique is called a *storyboard* (Leffingwell and Widrig 2000). Storyboards often illustrate the proposed user interface without

engaging the user in interacting with it.

Paper prototypes involve tools no more sophisticated than paper, index cards, sticky notes, and clear plastic sheets. The designer sketches ideas of what the screens might look like without worrying about exactly where the controls appear and what they look like. Users willingly provide feedback that can lead to profound changes on a piece of paper. Sometimes, though, they're less eager to critique a lovely computer-based prototype in which it appears the developer has invested a lot of work. Developers, too, might resist making substantial changes in a carefully crafted electronic prototype.

With lo-fi prototyping, a person plays the role of the computer while a user walks through an evaluation scenario. The user initiates actions by saying aloud what she would like to do at a specific screen: "I'm going to select Print Preview from the File menu." The person simulating the computer then displays the page or index card that represents the display that would appear when the user takes that action. The user can judge whether that is indeed the expected response and whether the item displayed contains the correct elements. If it's wrong, you simply take a fresh piece of paper or a blank index card and draw it again.

Off to See the Wizard

True Stories A development team that designed photocopiers once lamented to me that their last copier had a usability problem. A common copying activity required five discrete steps, which the users found clumsy. "I wish we'd prototyped that activity before we designed the copier," one developer said wistfully.

How do you prototype a product as complex as a photocopier? First, buy a big-screen television. Write *COPIER* on the side of the box that it came in. Have someone sit inside the box, and ask a user to stand outside the box and simulate doing copier activities. The person inside the box responds in the way he expects the copier to respond, and the user representative observes whether that response is what he has in mind. A simple, fun prototype like this—sometimes called a "Wizard of Oz prototype"—often gets the early user feedback that effectively guides the development team's design decisions. Plus you get to keep the big-screen television.

No matter how efficient your prototyping tools are, sketching displays on paper is faster. Paper prototyping facilitates rapid iteration, and iteration is a key success factor in requirements development. Paper prototyping is an excellent technique for refining the requirements prior to designing detailed user interfaces, constructing an evolutionary prototype, or undertaking traditional design and construction activities. It also helps the development team manage customer expectations.

If you decide to build an electronic throwaway prototype, several appropriate tools are available (Andriole 1996). These include the following:

- Programming languages such as Microsoft Visual Basic, IBM VisualAge Smalltalk, and Inprise Delphi
- Scripting languages such as Perl, Python, and Rexx
- Commercial prototyping tool kits, screen painters, and graphical user interface builders
- Drawing tools such as Microsoft Visio and Microsoft PowerPoint

Web-based approaches using HTML (Hypertext Markup Language) pages, which can be modified quickly, are useful for creating prototypes that are intended to clarify requirements. They aren't as useful for exploring detailed interface designs quickly. Suitable tools will let you easily implement and modify user interface components, regardless of how inefficient the code behind the interface is. Of course, if you're building an evolutionary prototype, you must use the production development tools from the outset.

Prototype Evaluation

To improve the evaluation of horizontal prototypes, create scripts that guide the users through a series of steps and ask specific questions to elicit the information you need. This supplements a general invitation to "tell me what you think of this prototype." Derive the evaluation scripts from the use cases or functions that the prototype addresses. The script asks evaluators to perform specific tasks and steers them through the parts of the prototype that have the most uncertainty. At the end of each task, and possibly at intermediate points, the script presents specific task-related questions. In addition, you might ask the following general questions:

- Does the prototype implement the functionality in the way you expected?
- Is any functionality missing?
- Can you think of any error conditions that the prototype doesn't address?
- Are any unnecessary functions present?
- How logical and complete does the navigation seem to you?
- Were any tasks overly complex?

Make sure the right people evaluate the prototype from the appropriate perspectives. Include both experienced and inexperienced user class representatives. When you present the prototype to the evaluators, stress that it addresses only a portion of the functionality; the rest will be implemented when the actual system is developed.

Trap Watch out for users who put production data into a prototype they're evaluating because the prototype seems to really work. They'll be unhappy if their data disappears when the prototype goes away.

You'll learn more by watching users work with the prototype than by just asking them to tell you what they think of it. Formal usability testing is powerful, but simple observation also is illuminating. Watch where the user's fingers try to go instinctively. Spot places where the prototype conflicts with the behavior of other applications that the evaluators use or where there are no clues concerning the user's next action. Look for the furrowed brow that indicates a puzzled user who can't tell what to do next, how to navigate to a desired destination, or how to take a side trip to another part of the application to look something up.

Ask evaluators to share their thoughts aloud as they work with the prototype so that you understand what they're thinking and can detect any requirements that the prototype handles poorly. Create a nonjudgmental environment in which the evaluators feel free to express their thoughts, ideas, and concerns. Avoid coaching users on the "right" way to perform some function with the prototype.

Document what you learn from the prototype evaluation. For a horizontal prototype, use the information to refine the requirements in the SRS. If the prototype evaluation led to some user interface design decisions or to the selection of specific interaction techniques, record those conclusions and how you arrived at them. Decisions that are missing the attendant thought processes tend to be revisited over and over, a waste of time. For a vertical prototype, document the evaluations you performed and their results, culminating in the decisions you made about the technical approaches explored. Look for any conflicts between the SRS and the prototype.

The Risks of Prototyping

Although prototyping reduces the risk of software project failure, it introduces its own risks. The biggest risk is that a stakeholder will see a running prototype and conclude that the product is nearly completed. "Wow, it looks like you're almost done!" says the enthusiastic prototype evaluator. "It looks great. Can you just finish this up and give it to me?"

In a word: NO! A throwaway prototype is never intended for production use, no matter how much it looks like the real thing. It is merely a model, a simulation, or an experiment. Unless there's a compelling business motivation to achieve a marketplace presence immediately (and management accepts the resulting high maintenance burden), resist the pressure to deliver a throwaway prototype. Delivering this prototype will actually delay the project's completion because the design and code intentionally were created without regard to quality or durability.

Trap Watch out for stakeholders who think that a prototype is just an early version of the production software. Expectation management is a key to successful prototyping. Everyone who sees the prototype must understand its purpose and its limitations.

Don't let the fear of premature delivery pressure dissuade you from creating prototypes. Make it clear to those who see the prototype that you will not release it as production software. One way to control this risk is to use paper, rather than electronic, prototypes. No one who evaluates a paper prototype will think the product is nearly done. Another option is to use prototyping tools that are different from those used for actual development. This will help you resist pressure to "just finish up" the prototype and ship it. Leaving the prototype looking a bit rough and unpolished also mitigates this risk.

Another risk of prototyping is that users become fixated on the *how* aspects of the system, focusing on how the user interface will look and how it will operate. When working with real-looking prototypes, it's easy for users to forget that they should be primarily concerned with the *what* issues at the requirements stage. Limit the prototype to the displays, functions, and navigation options that will let you clear up uncertainty in the requirements.

A third risk is that users will infer the expected performance of the final product from the prototype's performance. You won't be evaluating a horizontal prototype in the intended production environment. You might have built it using tools that differ in efficiency from the production development tools, such as interpreted scripts versus compiled code. A vertical prototype might not use tuned algorithms, or it might lack security layers that will compromise the ultimate performance. If evaluators see the prototype respond instantaneously to a simulated database query using hard-coded query results, they might expect the same fabulous performance in the production software with an enormous distributed database. Consider building in time delays to more realistically simulate the expected behavior of the final product (and perhaps to make the prototype look even less ready for immediate delivery).

Finally, beware of prototyping activities that consume so much effort that the development team runs out of time and is forced to deliver the prototype as the product or to rush through a haphazard product implementation. Treat a prototype as an experiment. You're testing the hypothesis that the requirements are sufficiently defined and the key human-computer interface and architectural issues are resolved so that design and construction can proceed. Do just enough prototyping to test the hypothesis, answer the questions, and refine your understanding of the requirements.

Prototyping Success Factors

Software prototyping provides a powerful set of techniques that can shorten development schedules, increase customer satisfaction, and produce higher-quality products. To make prototyping an effective part of your requirements process, follow these guidelines:

- Include prototyping tasks in your project plan. Schedule time and resources to develop, evaluate, and modify the prototypes.
- State the purpose of each prototype before you build it.
- Plan to develop multiple prototypes. You'll rarely get them right on the first try (which is the whole point of prototyping!).
- Create throwaway prototypes as quickly and cheaply as possible. Invest the minimum effort in developing prototypes that will answer questions or resolve requirements uncertainties. Don't try to perfect a throwaway prototype.
- Don't include extensive input data validations, defensive coding techniques, error-handling code, or code documentation in a throwaway prototype.
- Don't prototype requirements that you already understand, except to explore design alternatives.
- Use plausible data in prototype screen displays and reports. Evaluators can be distracted by unrealistic data and fail to focus on the prototype as a model of how the real system might look and behave.
- Don't expect a prototype to fully replace an SRS. A lot of behind-the-scenes functionality is only implied by the prototype and should be documented in an SRS to make it complete, specific, and traceable. The visible part of an application is the proverbial tip of the iceberg. Screen images don't give the details of data field definitions and validation criteria, relationships between fields (such as UI controls that appear only if the user makes certain selections in other controls), exception handling, business rules, and other essential bits of information.

Next Steps

- Identify a portion of your project, such as a use case, that reflects confusion about requirements. Sketch out a portion of a possible user interface that represents your understanding of the requirements and how they might be implemented—a paper prototype. Have some users walk through your prototype to simulate performing a usage scenario. Identify places where the initial requirements were incomplete or incorrect. Modify the prototype accordingly and walk through it again to confirm that the shortcomings are corrected.
- Summarize this chapter for your prototype evaluators to help them understand the rationale behind the prototyping activities and to help them have realistic expectations for the outcome.

Team LiB
Team LiB
PREVIOUS NEXT
PREVIOUS NEXT

Chapter 14: Setting Requirement Priorities

Overview

After most of the user requirements for the Chemical Tracking System were documented, the project manager, Dave, and the requirements analyst, Lori, met with two of the product champions. Tim represented the chemist community and Roxanne spoke for the chemical stockroom staff.

"As you know," Dave began, "the product champions provided many requirements for the Chemical Tracking System. Unfortunately, we can't include all the functionality you requested in the first release. Since most of the requirements came from the chemists and the chemical stockroom, I'd like to talk with you about prioritizing your requirements."

Tim was puzzled. "Why do you need the requirements prioritized? They're all important, or we wouldn't have given them to you."

Dave explained, "I know they're all important, but we can't fit everything in and still deliver a high-quality product on schedule. We want to make sure we address the most urgent requirements in the first release, which is due at the end of next quarter. We're asking you to help us distinguish the requirements that must be included initially from those that can wait."

"I know the reports that the Health and Safety Department needs to generate for the government have to be available by the end of the quarter or the company will get in trouble," Roxanne pointed out. "We can use our current inventory system for a few more months if we have to. But the barcode labeling and scanning features are essential, more important than searchable vendor catalogs for the chemists."

Tim protested. "I promised the catalog search function to the chemists as a way for this system to save them time. The catalog search has to be included from the beginning," he insisted.

Lori, the analyst, said, "While I was discussing use cases with the chemists, it seemed that some would be performed very often and others just occasionally or only by a few people. Could we look at your complete set of use cases and figure out which ones you don't need immediately? I'd also like to defer some of the bells and whistles from the top-priority use cases if we can."

Tim and Roxanne weren't thrilled that they'd have to wait for some of the system's functionality to be delivered later. However, they realized that it was unreasonable to expect miracles. If the product couldn't contain every requirement in release 1.0, it would be better if everyone could agree on the set to implement first.

Every software project with resource limitations needs to define the relative priorities of the requested product capabilities. Prioritization helps the project manager resolve conflicts, plan for staged deliveries, and make the necessary trade-offs. This chapter discusses the importance of prioritizing requirements, suggests a simple priority classification scheme, and describes a more rigorous prioritization analysis based on value, cost, and risk.

PREVIOUS NEXT
PREVIOUS NEXT

Why Prioritize Requirements?

When customer expectations are high and timelines are short, you need to make sure the product delivers the most valuable functionality as early as possible. Prioritization is a way to deal with competing demands for limited resources. Establishing the relative priority of each capability lets you plan construction to provide the highest value at the lowest cost. Prioritization is critical for timeboxed or incremental development with tight, immovable release schedules. In the Extreme Programming methodology, customers select which *user stories*

they'd like implemented in each two- to three-week incremental release, and the developers estimate how many of these stories they can fit into each increment.

A project manager must balance the desired project scope against the constraints of schedule, budget, staff, and quality goals. One way to accomplish this is to drop—or to defer to a later release—low-priority requirements when new, more vital requirements are accepted or when other project conditions change. If the customers don't distinguish their requirements by importance and urgency, project managers must make these decisions on their own. Not surprisingly, customers might not agree with a project manager's priorities; therefore, customers must indicate which requirements are needed initially and which can wait. Establish priorities early in the project, when you have more options available for achieving a successful project outcome, and revisit them periodically.

It's difficult enough to get any one customer to decide which of his requirements are top priority. Achieving consensus among multiple customers with diverse expectations is even harder. People naturally have their own interests at heart and aren't eager to compromise their needs for someone else's benefit. However, contributing to requirements prioritization is one of the customer's responsibilities in the customer-development partnership, as was discussed in Chapter 2, "Requirements from the Customer's Perspective." More than simply defining the sequence of requirements implementation, discussing priorities helps to clarify the customers' expectations.

Customers and developers both should provide input to requirements prioritization. Customers place a high priority on those functions that provide the greatest business or usability benefit. However, once a developer points out the cost, difficulty, technical risk, or trade-offs associated with a specific requirement, the customers might conclude that it isn't as essential as they initially believed. The developer might also decide to implement certain lower priority functions early on because of their impact on the system's architecture.

Games People Play with Priorities

True Stories The knee-jerk response to a request for customers to set priorities is, "I need all these features. Just make it happen somehow." It can be difficult to persuade customers to discuss priorities if they know that low-priority requirements might never be implemented. One developer told me that it wasn't politically acceptable in his company to say that a requirement had low priority. The priority categories they adopted were high, super-high, and incredibly high. Another developer claimed that priorities weren't necessary: if he wrote something in the SRS, he intended to build it. That doesn't address the issue of *when* each piece of functionality gets built, though. Some developers shun prioritization because it conflicts with the "we can do it all" attitude they want to convey.

Trap Avoid "decibel prioritization," in which the loudest voice heard gets top priority, and "threat prioritization," in which stakeholders holding the most political power always get what they demand.

In reality, some system capabilities are more essential than others. This becomes apparent during the all-too-common "rapid descoping phase" late in the project, when nonessential features are jettisoned to ensure that the critical capabilities ship on schedule. Setting priorities early in the project and reassessing them in response to changing customer preferences, market conditions, and business events let the team spend time wisely on high-value activities. Implementing most of a feature before you conclude that it isn't necessary is wasteful and frustrating.

If left to their own devices, customers will establish perhaps 85 percent of the requirements as high priority, 10 percent as medium, and 5 percent as low. This doesn't give the project manager much flexibility. If nearly all requirements truly are of top priority, your project has a high risk of not being fully successful and you need to

plan accordingly. Scrub the requirements to eliminate any that aren't essential and to simplify those that are unnecessarily complicated (McConnell 1996). To encourage customer representatives to identify lower-priority requirements, the analyst can ask questions such as the following:

- Is there some other way to satisfy the need that this requirement addresses?
- What would the consequence be of omitting or deferring this requirement?
- What would the impact be on the project's business objectives if this requirement weren't implemented immediately?
- Why would a user be unhappy if this requirement were deferred to a later release?

True Stories The management steering team on one large commercial project displayed impatience over the analyst's insistence on prioritizing the requirements. The managers pointed out that often they can do without a particular feature but that another feature might need to be beefed up to compensate. If they deferred too many requirements, the resulting product wouldn't achieve the projected revenue. When you evaluate priorities, look at the connections and interrelationships among different requirements and their alignment with the project's business objectives.

A Prioritization Scale

A common prioritization approach groups requirements into three categories. No matter how you label them, if you're using three categories they boil down to high, medium, and low priority. Such prioritization scales are subjective and imprecise. The stakeholders must agree on what each level means in the scale they use.

One way to assess priority is to consider the two dimensions of *importance* and *urgency* (Covey 1989). Every requirement can be considered as being either important or not important and as being either urgent or not urgent. As Table 14-1 shows, these alternatives yield four possible combinations, which we can use to define a priority scale:

- *High priority* requirements are both important (the user needs the capability) and urgent (the user needs it in the next release). Contractual or legal obligations might dictate that the requirement must be included, or there might be compelling business reasons to implement it promptly.
- *Medium priority* requirements are important (the user needs the capability) but not urgent (they can wait for a later release).
- *Low priority* requirements are not important (the user can live without the capability if necessary) and not urgent (the user can wait, perhaps forever).
- Requirements in the fourth quadrant appear to be urgent but they really aren't important. Don't waste your time working on these. They don't add sufficient value to the product.

Table 14-1: Requirements Prioritization Based on Importance and Urgency

	Important	**Not Important**
Urgent	High Priority	Don't do these!
Not Urgent	Medium Priority	Low Priority

Include the priority of each requirement in the use-case descriptions, the SRS, or the requirements database. Establish a convention so that the reader knows whether the priority assigned to a high-level requirement is inherited by all its subordinate requirements or whether every individual functional requirement is to have its own priority attribute.

Even a medium-sized project can have hundreds of functional requirements, too many to classify analytically and consistently. To keep it manageable, choose an appropriate level of abstraction for the prioritization—features, use cases, or functional requirements. Within a use case, some alternative courses could have a higher priority than others. You might decide to do an initial prioritization at the feature level and then to prioritize the functional requirements within certain features separately. This will help you to distinguish the core functionality from refinements to schedule for later releases. Document even the low-priority requirements. Their priority might change later, and knowing about them now will help the developers to plan for future enhancements.

Prioritization Based on Value, Cost, and Risk

On a small project the stakeholders can agree on requirement priorities informally. Large or contentious projects demand a more structured approach, which removes some of the emotion, politics, and guesswork from the process. Several analytical and mathematical techniques have been proposed to assist with requirements prioritization. These methods involve estimating the relative value and relative cost of each requirement. The highest priority requirements are those that provide the largest fraction of the total product value at the smallest fraction of the total cost (Karlsson and Ryan 1997; Jung 1998). Subjectively estimating the cost and value by pairwise comparisons of all the requirements is impractical for more than a couple dozen requirements.

Another alternative is Quality Function Deployment (QFD), a comprehensive method for relating customer value to proposed product features (Zultner 1993; Cohen 1995). A third approach, borrowed from Total Quality Management (TQM), rates each requirement against several weighted project success criteria and computes a score to rank the priority of the requirements. However, few software organizations seem to be willing to undertake the rigor of QFD or TQM.

Table 14-2 illustrates a spreadsheet model to help you to estimate the relative priorities for a set of use cases, features, or functional requirements. This Microsoft Excel spreadsheet is available for downloading at *http://www.processimpact.com/goodies.shtml*. The example in Table 14-2 lists several features from (what else?) the Chemical Tracking System. This scheme borrows from the QFD concept of basing customer value on both the benefit provided to the customer if a specific product feature is present and the penalty paid if that feature is absent (Pardee 1996). A feature's attractiveness is directly proportional to the value it provides and inversely proportional to its cost and the technical risk associated with implementing it. All other things being equal, those features with the highest risk-adjusted value/cost ratio should have the highest priority. This approach distributes a set of estimated priorities across a continuum, rather than grouping them into just a few discrete levels.

Apply this prioritization scheme to discretionary requirements, those that aren't top priority. You wouldn't include in this analysis items that implement the product's core business functions, items that you view as key product differentiators, or items that are required for compliance with government regulations. Unless there's a chance that those capabilities could shift to lower priority if conditions change, you're going to incorporate them into the product promptly. Once you've identified those features that absolutely must be included for the product to be releasable, use the model in Table 14-2 to scale the relative priorities of the remaining capabilities. The typical participants in the prioritization process include:

- The project manager, who leads the process, arbitrates conflicts, and adjusts input from the other participants if necessary
- Customer representatives, such as product champions or marketing staff, who supply the benefit and penalty ratings
- Development representatives, such as team technical leads, who provide the cost and risk ratings

Table 14-2: Sample Prioritization Matrix for the Chemical Tracking System

Relative Weights	**2**	**1**			**1**		**0.5**		
Feature	**Relative Benefit**	**Relative Penalty**	**Total Value**	**Value %**	**Relative Cost**	**Cost %**	**Relative Risk**	**Risk %**	**Priority**
1. Print a material safety data sheet.	2	4	8	5.2	1	2.7	1	3.0	1.22
2. Query status of a vendor order.	5	3	13	8.4	2	5.4	1	3.0	1.21
3. Generate a Chemical Stockroom inventory report.	9	7	25	16.1	5	13.5	3	9.1	0.89
4. See history of a specific chemical container.	5	5	15	9.7	3	8.1	2	6.1	0.87
5. Search vendor catalogs for a specific chemical.	9	8	26	16.8	3	8.1	8	24.2	0.83
6. Maintain a list of hazardous chemicals.	3	9	15	9.7	3	8.1	4	12.1	0.68
7. Modify a pending chemical request.	4	3	11	7.1	3	8.1	2	6.1	0.64
8. Generate an individual laboratory inventory report.	6	2	14	9.0	4	10.8	3	9.1	0.59
9. Check training database for hazardous	3	4	10	6.5	4	10.8	2	6.1	0.47

chemical training record.									
10. Import chemical structures from structure drawing tools.	7	4	18	11.6	9	24.3	7	21.2	0.33
Totals	**53**	**49**	**155**	**100.0**	**37**	**100.0**	**33**	**100.0**	

Follow these steps to use this prioritization model:

1. List in the spreadsheet all the features, use cases, or requirements that you want to prioritize; in the example, I've used features. All the items must be at the same level of abstraction—don't mix functional requirements with product features. If certain features are logically linked (for example, you'd implement feature B only if feature A were included), list only the driving feature in the analysis. This model will work with up to several dozen features before it becomes unwieldy. If you have more items than that, group related features together to create a manageable initial list. You can do a second round of analysis at a finer level of detail if necessary.

2. Have your customer representatives estimate the relative benefit that each feature would provide to the customer or to the business on a scale of 1 to 9. A rating of 1 indicates that no one would find it useful and 9 means it would be extremely valuable. These benefit ratings indicate alignment of the features with the product's business requirements.

3. Estimate the relative penalty that the customer or business would suffer if the feature were not included. Again, use a scale of 1 to 9. A rating of 1 means that no one will be upset if it's excluded; 9 indicates a serious downside. Requirements that have both a low benefit and a low penalty add cost but little value; they might be instances of gold plating, functions that look appealing but aren't worth the investment. When assigning penalty ratings, consider how unhappy the customers will be if a specific capability isn't included. Ask yourselves questions such as the following:

    - Would your product suffer in comparison with other products that do contain that capability?
    - Would there be any legal or contractual consequences?
    - Would you be violating some government or industry standard?
    - Would users be unable to perform some necessary or expected functions?
    - Would it be a lot harder to add that capability later as an enhancement?
    - Would problems arise because marketing has promised a feature to satisfy some potential customers but the team decided to omit it?

4. The spreadsheet calculates the total value for each feature as the sum of its benefit and penalty scores. By default, benefit and penalty are weighted equally. As a refinement, you can change the relative weights for these two factors in the top row of the spreadsheet. In the example in Table 14-2, all benefit ratings are weighted twice as heavily as the corresponding penalty ratings. The spreadsheet sums the feature values and calculates the percentage of the total value contributed by this set of features that comes from

each of the features (the Value % column).

5. Have developers estimate the relative cost of implementing each feature, again on a scale of 1 (quick and easy) to 9 (time consuming and expensive). The spreadsheet will calculate the percentage of the total cost that each proposed feature contributes. Developers estimate the cost ratings based on the feature's complexity, the extent of user interface work required, the potential ability to reuse existing code, the amount of testing and documentation that will be needed, and so forth.

6. Similarly, have developers rate the relative degree of technical or other risks associated with each feature on a scale of 1 to 9. Technical risk is the probability of *not* getting the feature right on the first try. A rating of 1 means that you can program it in your sleep. A 9 indicates serious concerns about feasibility, the lack of staff with the necessary expertise, or the use of unproven or unfamiliar tools and technologies. Ill-defined requirements that might need rework will incur a high risk rating. The spreadsheet will calculate the percentage of the total risk that comes from each feature.

   In the standard model, the benefit, penalty, cost, and risk terms are weighted equally, but you may adjust all four weightings. In the spreadsheet in Table 14-2, risk has half the weight of the cost factor, which has the same weight as the penalty term. If you don't want to consider risk at all in the model, set the risk weighting value to zero.

7. Once you've entered all the estimates into the spreadsheet, it will calculate a priority value for each feature using the following formula:

   $$\text{priority} = \frac{\text{value \%}}{(\text{cost \%} * \text{cost weight}) + (\text{risk \%} * \text{risk weight})}$$

8. Sort the list of features in descending order by calculated priority. The features at the top of the list have the most favorable balance of value, cost, and risk and thus—all other factors being equal—should have highest priority.

Or, We Could Arm Wrestle

True Stories One company that introduced a requirements prioritization procedure based on the spreadsheet described in this chapter found that it helped a project team to break through an impasse. Several stakeholders had different opinions about which features were most important on a large project, and the team was deadlocked. The spreadsheet analysis made the priority assessment more objective and less emotionally charged, enabling the team to reach some appropriate conclusions and move ahead.

Consultant Johanna Rothman (2000) reported that, "I have suggested this spreadsheet to my clients as a tool for decision-making. Although the ones who tried it have never completely filled out the spreadsheet, they found the discussion it stimulated extremely helpful in deciding the relative priorities of the different requirements." That is, use the framework of benefit, penalty, cost, and risk to guide the discussions about priorities. This is more valuable than the formalism of completely working through the spreadsheet analysis and relying exclusively on the calculated priority sequence.

This technique's accuracy is limited by the team's ability to estimate the benefit, penalty, cost, and risk for each item. Therefore, use the calculated priority sequence only as a guideline. Customer and development representatives should review the completed spreadsheet to agree on the ratings and the resulting sorted priority sequence. Calibrate this model for your own use with a set of completed requirements from a previous project. Adjust the weighting factors until the calculated priority sequence correlates well with your after-the-fact evaluation of how important the requirements in your calibration set really were. This will give you some confidence in using the tool as a model of how you make priority decisions on your projects.

Trap Don't overinterpret small differences in calculated priority numbers. This semiquantitative method is not mathematically rigorous. Look for groups of requirements that have similar priority numbers.

Different stakeholders often have conflicting ideas about the relative importance of a specific requirement or the penalty of omitting it. The prioritization spreadsheet has a variant that accommodates input from several user classes or other stakeholders. In the Multiple Stakeholders worksheet tab in the downloadable spreadsheet, duplicate the Relative Benefit and Relative Penalty columns so that you have a set for each stakeholder who's contributing to the analysis. Then assign a weighting factor to each stakeholder, giving higher weights to favored user classes than to groups who have less influence on the project's decisions. Have each stakeholder representative provide his own benefit and penalty ratings for each feature. The spreadsheet will incorporate the stakeholder weights when it calculates the total value scores.

This model can also help you to make trade-off decisions when you're evaluating proposed requirements changes. See how their priorities align with those of the existing requirements baseline so that you can choose an appropriate implementation sequence.

Keep your prioritization process as simple as possible, but no simpler. Strive to move requirements away from the political arena into a forum in which stakeholders can make honest assessments. This will give you a better chance of building products that deliver the maximum business value.

Next Steps

- Apply the prioritization model described in this chapter to 10 or 15 features or use cases from a recent project. How well do the calculated priorities compare with the priorities you had determined by some different method? How well do they compare with your subjective sense of the proper priorities?
- If there's a disconnect between what the model predicts for priorities and what you think is right, analyze which part of the model isn't giving sensible results. Try using different weighting factors for benefit, penalty, cost, and risk. Adjust the model until it provides results consistent with what you expect.
- Once you've calibrated and adjusted the prioritization model, apply it to a new project. Incorporate the calculated priorities into the decision-making process. See whether this yields results that the stakeholders find more satisfying than those from their previous prioritization approach.

Team LiB Team LiB PREVIOUS NEXT PREVIOUS NEXT

Chapter 15: Validating the Requirements

Overview

Barry, a test lead, was the moderator for an inspection meeting whose participants were carefully examining a software requirements specification for problems. The meeting included representatives from the two major user classes, a developer named Jeremy, and Trish, the analyst who wrote the SRS. Requirement 83 stated, "The system shall provide unattended terminal timeout security of workstations accessing DMV." Jeremy presented his interpretation of this requirement to the rest of the group. "Requirement 83 says that the system will automatically log off the current user of any workstation logged into the DMV system if there hasn't been any activity within a certain period of time."

"Does anyone have any issues with requirement 83?" asked Barry.

Hui-Lee, one of the product champions, spoke first. "How does the system determine that the terminal is unattended? Is it like a screen saver, so if there isn't any mouse or keyboard activity for, say, five minutes, it logs the user off? That could be annoying if the user were just talking to someone for a few minutes."

Trish added, "Actually, the requirement doesn't say anything about logging off the user. I guess I'm not exactly sure what timeout security means. I assumed it would be a logoff, but maybe the user just has to type his password again."

Jeremy was confused also. "Does this mean any workstation that can connect to the DMV system, or just workstations that are actively logged into the DMV system at the moment? How long a timeout period are we talking about? Maybe there's a security guideline for this kind of thing."

Barry made sure that the inspection recorder had captured all these concerns accurately. "Okay, it looks like requirement 83 contains several ambiguities and some missing information, such as the timeout period. Trish, can you please check with the department security coordinator to clear this up?"

Most software developers have experienced the frustration of being asked to implement requirements that were ambiguous or incomplete. If they can't get the information they need, the developers have to make their own interpretations, which aren't always correct. Substantial effort is needed to fix requirement errors after work based on those requirements has already been completed. Studies have shown that it can cost approximately 100 times more to correct a customer-reported requirement defect than to correct an error found during requirements development (Boehm 1981; Grady 1999). Another study found that it took an average of 30 minutes to fix an error discovered during the requirements phase. In contrast, 5 to 17 hours were needed to correct a defect identified during system testing (Kelly, Sherif, and Hops 1992). Clearly, any measures you can take to detect errors in the requirements specifications will save you substantial time and money.

On many projects, testing is a late-stage activity. Requirements-related problems linger in the product until they're finally revealed through time-consuming system testing or by the customer. If you start your test planning and test-case development early, you'll detect many errors shortly after they're introduced. This prevents them from doing further damage and reduces your testing and maintenance costs.

Figure 15-1 illustrates the V model of software development, which shows test activities beginning in parallel with the corresponding development activities (Davis 1993). This model indicates that acceptance testing is based on the user requirements, system testing is based on the functional requirements, and integration testing is based on the system's architecture.

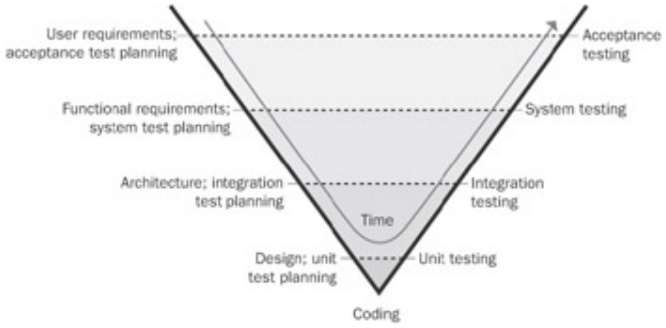


Figure 15-1: The V model of software development incorporates early test planning and test design.

Plan your testing activities and begin developing preliminary test cases during the corresponding development phase. You can't actually run any tests during requirements development because you don't have any software to execute yet. However, conceptual (that is, implementation-independent) test cases based on the requirements will reveal errors, ambiguities, and omissions in your SRS and analysis models long before the team writes any code.

Requirements validation is the fourth component—with elicitation, analysis, and specification—of requirements development (Abran and Moore 2001).[1] Requirements validation activities attempt to ensure that

- The SRS correctly describes the intended system capabilities and characteristics that will satisfy the various stakeholders' needs.
- The software requirements were correctly derived from the system requirements, business rules, or other sources.
- The requirements are complete and of high quality.
- All requirements representations are consistent with each other.
- The requirements provide an adequate basis to proceed with design and construction.

Validation ensures that the requirements exhibit the desirable characteristics of excellent requirement statements (complete, correct, feasible, necessary, prioritized, unambiguous, and verifiable) and of excellent requirements specifications (complete, consistent, modifiable, and traceable). Of course, you can validate only requirements that have been documented, not implicit requirements that exist only in someone's mind.

Validation isn't a single discrete phase that you perform after gathering and documenting all the requirements. Some validation activities, such as incremental reviews of the growing SRS, are threaded throughout the iterative elicitation, analysis, and specification processes. Other activities, such as formal SRS inspection, provide a final quality gate prior to baselining the SRS. Include requirements validation activities as discrete tasks in your project plan.

Project participants sometimes are reluctant to invest time in reviewing and testing an SRS. Intuitively, it seems that inserting time into the schedule to improve requirements quality would delay the planned ship date by that same duration. However, this expectation assumes a zero return on your investment in requirements validation. In reality, that investment can actually *shorten* the delivery schedule by reducing the rework required and by accelerating system integration and testing (Blackburn, Scudder, and Van Wassenhove 1996). Capers Jones (1994) reports that every dollar spent to prevent defects will reduce your defect repair costs by 3 to 10 dollars. Better requirements lead to higher product quality and customer satisfaction, which reduce the product's lifetime costs for maintenance, enhancement, and customer support. Investing in requirements quality always saves you more money than you spend.

Various techniques can help you to evaluate the correctness and quality of your requirements (Wallace and Ippolito 1997). One approach is to quantify each requirement so that you can think of a way to measure how well a proposed solution satisfies it. Suzanne and James Robertson use the term *fit criteria* to describe such quantifications (Robertson and Robertson 1999). This chapter addresses the validation techniques of formal and informal requirements reviews, developing test cases from requirements, and having customers define their acceptance criteria for the product.

[1] Some authors use the term *verification* for this step (Thayer and Dorfman 1997). Verification determines whether the product of a development activity meets the requirements established for it (doing the thing right). *Validation* assesses whether a product actually satisfies the customer needs (doing the right thing). In this book, I've adopted the terminology of the *Software Engineering Body of Knowledge* (Abran and Moore 2001) and refer to the fourth subdomain of requirements development as *validation*.

PREVIOUS NEXT
PREVIOUS NEXT

Reviewing Requirements

Any time someone other than the author of a software work product examines the product for problems, a *peer review* is taking place. Reviewing requirements documents is a powerful technique for identifying ambiguous or unverifiable requirements, requirements that aren't defined clearly enough for design to begin, and other problems.

Different kinds of peer reviews go by a variety of names (Wiegers 2002a). Informal reviews are useful for educating other people about the product and collecting unstructured feedback. However, they are not systematic, thorough, or performed in a consistent way. Informal review approaches include:

- A *peer deskcheck*, in which you ask one colleague to look over your work product
- A *passaround*, in which you invite several colleagues to examine a deliverable concurrently
- A *walkthrough*, during which the author describes a deliverable and solicits comments on it

In contrast to the ad hoc nature of informal reviews, formal peer reviews follow a well-defined process. A formal review produces a report that identifies the material, the reviewers, and the review team's judgment as to whether the product is acceptable. The principal deliverable is a summary of the defects found and the issues raised. The members of a formal review team share responsibility for the quality of the review, although authors are ultimately responsible for the quality of the products they create (Freedman and Weinberg 1990).

The best-established type of formal peer review is called an *inspection* (Gilb and Graham 1993; Radice 2002). Inspection of requirements documents is arguably the highest-leverage software quality technique available. Several companies have avoided as much as 10 hours of labor for every hour they invested in inspecting requirements documents and other software deliverables (Grady and Van Slack 1994). A 1000 percent return on investment is not to be sneezed at.

If you're serious about maximizing the quality of your software, your team will inspect every requirements document it creates. Detailed inspection of large requirements documents is tedious and time consuming. Nonetheless, the people I know who have adopted requirements inspections agree that every minute they spent was worthwhile. If you don't have time to inspect everything, use risk analysis to differentiate those requirements that demand inspection from less critical material for which an informal review will suffice. Begin holding SRS inspections when the requirements are perhaps only 10 percent complete. Detecting major defects early and spotting systemic problems in the way the requirements are being written is a powerful way to prevent—not just find—defects.

The Closer You Look, the More You See

True Stories On the Chemical Tracking System, the user representatives informally reviewed their latest contribution to the growing SRS after each elicitation workshop. These quick reviews uncovered many errors. After elicitation was complete, one analyst combined the input from all user classes into a single SRS of about 50 pages plus several appendices. Two analysts, one developer, three product champions, the project manager, and one tester then inspected this full SRS in three two-hour inspection meetings held over the course of a week. The inspectors found 223 additional errors, including dozens of major defects. All the inspectors agreed that the time they spent grinding through the SRS, one requirement at a time, saved the project team countless more hours in the long run.

The Inspection Process

Michael Fagan developed the inspection process at IBM in the mid-1970s (Fagan 1976), and others have extended or modified his methods (Gilb and Graham 1993). Inspection has been recognized as a software industry best practice (Brown 1996). Any software work product can be inspected, including requirements and design documents, source code, test documentation, and project plans.

Inspection is a well-defined multistage process. It involves a small team of trained participants who carefully examine a work product for defects and improvement opportunities. Inspections provide a quality gate through which documents must pass before they are baselined. There's some debate whether the Fagan method is the best form of inspection (Glass 1999), but there's no question that inspections are a powerful quality technique. Many helpful work aids for software peer reviews and inspections are available at *http://www.processimpact.com/goodies.shtml*, including a sample peer review process description, defect checklists, and inspection forms.

Participants

The participants in an inspection should represent four perspectives (Wiegers 2002a):

- **The author of the work product and perhaps peers of the author** The analyst who wrote the requirements document provides this perspective.
- **The author of any predecessor work product or specification for the item being inspected** This might be a system engineer or an architect who can ensure that the SRS properly details the system specification. In the absence of a higher-level specification, the inspection must include customer representatives to ensure that the SRS describes their requirements correctly and completely.
- **People who will do work based on the item being inspected** For an SRS, you might include a developer, a tester, a project manager, and a user documentation writer. These inspectors will detect different kinds of problems. A tester is most likely to catch an unverifiable requirement; a developer can spot requirements that are technically infeasible.
- **People who are responsible for work products that interface with the item being inspected** These inspectors will look for problems with the external interface requirements. They can also spot ripple effects, in which changing a requirement in the SRS being inspected affects other systems.

Try to limit the team to six or fewer inspectors. This means that some perspectives won't be represented in every inspection. Large teams easily get bogged down in side discussions, problem solving, and debates over whether something is really an error. This reduces the rate at which they cover the material during the inspection and increases the cost of finding each defect.

Inspection Roles

All participants in an inspection, including the author, look for defects and improvement opportunities. Some of the inspection team members perform the following specific roles during the inspection.

Author The author created or maintains the work product being inspected. The author of an SRS is usually the analyst who gathered customer requirements and wrote the specification. During informal reviews such as walkthroughs, the author often leads the discussion. However, the author takes a more passive role during an inspection. The author may not assume any of the other assigned roles—moderator, reader, or recorder. By not having an active role and by parking his ego at the door, the author can listen to the comments from other inspectors, respond to—but not debate—their questions, and think. The author will often spot errors that other inspectors don't see.

Moderator The moderator, or *inspection leader*, plans the inspection with the author, coordinates the activities, and facilitates the inspection meeting. The moderator distributes the materials to be inspected to the

participants several days before the inspection meeting. Moderator responsibilities include starting the meeting on time, encouraging contributions from all participants, and keeping the meeting focused on finding defects rather than resolving problems. Reporting the inspection results to management or to someone who collects data from multiple inspections is another moderator activity. The moderator follows up on proposed changes with the author to ensure that the defects and issues that came out of the inspection were addressed properly.

Reader One inspector is assigned the role of reader. During the inspection meeting, the reader paraphrases the SRS one requirement at a time. The other participants then point out potential defects and issues. By stating a requirement in her own words, the reader provides an interpretation that might differ from that held by other inspectors. This is a good way to reveal an ambiguity, a possible defect, or an assumption. It also underscores the value of having someone other than the author serve as the reader.

Recorder The recorder, or *scribe*, uses standard forms to document the issues raised and the defects found during the inspection meeting. The recorder should review aloud what he wrote to confirm the record's accuracy. The other inspectors should help the recorder capture the essence of each issue in a way that clearly communicates to the author the location and nature of the issue.

Entry Criteria

You're ready to inspect a requirements document when it satisfies specific prerequisites. These *entry criteria* set some clear expectations for authors to follow while preparing for an inspection. They also keep the inspection team from spending time on issues that should be resolved prior to the inspection. The moderator uses the entry criteria as a checklist before deciding to proceed with the inspection. Following are some suggested inspection entry criteria for requirements documents:

- The document conforms to the standard template.
- The document has been spell-checked.
- The author has visually examined the document for any layout errors.
- Any predecessor or reference documents that the inspectors will require to examine the document are available.
- Line numbers are printed on the document to facilitate referring to specific locations during the inspection meeting.
- All open issues are marked as TBD (to be determined).
- The moderator didn't find more than three major defects in a ten-minute examination of a representative sample of the document.

Inspection Stages

An inspection is a multistep process, as illustrated in Figure 15-2. The purpose of each inspection process stage is summarized briefly in the rest of this section.

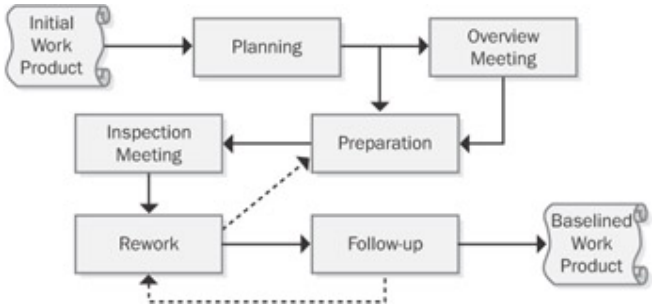


Figure 15-2: Inspection is a multistep process. The dotted lines indicate that portions of the inspection process might be repeated.

Planning The author and moderator plan the inspection together. They determine who should participate, what materials the inspectors should receive prior to the inspection meeting, and how many inspection meetings will be needed to cover the material. The inspection rate has a large impact on how many defects are found (Gilb and Graham 1993). As Figure 15-3 shows, proceeding through the SRS slowly reveals the most defects. (An alternative interpretation of this frequently reported relationship is that the inspection slows down if you encounter a lot of defects.) Because no team has infinite time available for requirements inspections, select an appropriate rate based on the risk of overlooking major defects. Two to four pages per hour is a practical guideline, although the optimum inspection rate for maximum defect-detection effectiveness is about half that rate (Gilb and Graham 1993). Adjust this rate based on the following factors:

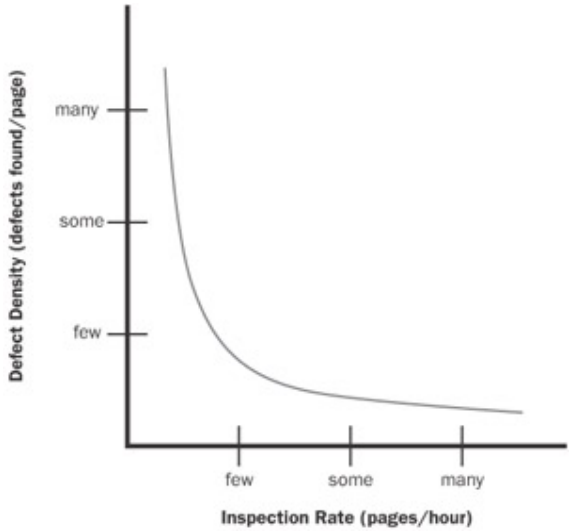


Figure 15-3: The number of defects found depends on the inspection rate.

- The team's previous inspection data
- The amount of text on each page
- The specification's complexity
- The risk of having errors remain undetected
- How critical the material being inspected is to project success
- The experience level of the person who wrote the SRS

Overview meeting During the overview meeting, the author describes the background of the material the team will be inspecting, any assumptions he made, and his specific inspection objectives. You can omit this stage if

all inspectors are already familiar with the items being inspected.

Preparation Prior to the inspection meeting, each inspector examines the product to identify possible defects and issues to be raised, using the checklists of typical defects described later in this chapter or other analysis techniques (Wiegers 2002a). Up to 75 percent of the defects found by an inspection are discovered during preparation (Humphrey 1989), so don't omit this step.

More Info The techniques for finding missing requirements described in Chapter 7, "Hearing the Voice of the Customer," can be helpful during inspection preparation.

Trap Don't proceed with an inspection meeting if the participants haven't already examined the work product on their own. Ineffective meetings can lead to the erroneous conclusion that inspections are a waste of time.

Written requirements documents can never replace clarifying discussions between project participants, but serious shortcomings in the specification need to be corrected. Consider asking developers who review the SRS to rate each requirement as to how well they understand it on a scale of 1 to 5 (Pfleeger 2001). A rating of 1 means that the developer doesn't have a clue what the requirement is about. A rating of 5 says that it's perfectly clear, complete, and unambiguous.

Inspection meeting During the inspection meeting, the reader leads the other inspectors through the SRS, describing one requirement at a time in his or her own words. As inspectors bring up possible defects and other issues, the recorder captures them on a form that becomes the action item list for the requirements author. The purpose of the meeting is to identify as many major defects in the requirements document as possible. The inspection meeting shouldn't last more than about two hours because tired people aren't effective inspectors. If you need more time to cover all the material, schedule another meeting.

At the meeting's conclusion, the team decides whether to accept the requirements document as is, accept it with minor revisions, or indicate that major revision is needed. An outcome of *major revision needed* indicates problems with the requirements development process. Consider holding a retrospective to explore how the process can be improved prior to the next specification activity (Kerth 2001).

Sometimes inspectors report only superficial and cosmetic issues. In addition, inspectors are easily sidetracked into discussing whether an issue really is a defect, debating questions of project scope, and brainstorming solutions to problems. These activities are useful, but they distract attention from the core objective of finding significant defects and improvement opportunities.

Some researchers suggest that inspection meetings are too labor intensive to justify holding them (Votta 1993). I've found that the meetings can reveal additional defects that no inspectors found during their individual preparation. As with all quality activities, make a risk-based decision as to how much energy to devote to improving requirements quality before you proceed with design and construction.

Rework Nearly every quality control activity reveals some defects. The author should plan to spend some time reworking the requirements following the inspection meeting. Uncorrected requirement defects will be expensive to fix down the road, so this is the time to resolve the ambiguities, eliminate the fuzziness, and lay the foundation for a successful development project. There's little point in holding an inspection if you don't intend to correct the defects it reveals.

Follow-up In this final inspection step, the moderator or a designated individual works with the author to ensure that all open issues were resolved and that errors were corrected properly. Follow-up brings closure to the inspection process and enables the moderator to determine whether the inspection's exit criteria have been satisfied.

Exit Criteria

Your inspection process should define the exit criteria that must be satisfied before the moderator declares the inspection complete. Here are some possible exit criteria for requirements document inspections:

- All issues raised during the inspection have been addressed.
- Any changes made in the document and related work products were made correctly.
- The revised document has been spell-checked.
- All TBDs have been resolved, or each TBD's resolution process, target date, and owner has been documented.
- The document has been checked into the project's configuration management system.

Defect Checklists

To help inspectors look for typical kinds of errors in the products they inspect, develop a defect checklist for each type of requirements document your organization creates. Such checklists call the inspectors' attention to historically frequent requirement problems. Figure 15-4 illustrates a use-case inspection checklist, and Figure 15-5 presents a checklist for inspecting an SRS.

- Is the use case a stand-alone, discrete task?
- Is the goal, or measurable value, of the use case clear?
- Is it clear which actor or actors benefit from the use case?
- Is the use case written at the essential level, free of unnecessary design and implementation details?
- Are all anticipated alternative courses documented?
- Are all know exception conditions documented?
- Are there any common action sequences that could be split into separate use cases?
- Are the steps for each course clearly written, unambiguous, and complete?
- Is each actor and step in the use case pertinent to performing that task?
- Is each altemative course defined in the use case feasible and verifiable?
- Do the preconditions and postconditions properly frame the use case?

Figure 15-4: Defect checklist for use-case documents.

No one can remember all the items on a long checklist. Pare the lists to meet your organization's needs, and modify them to reflect the problems that people encounter most often in your own requirements. You might ask certain inspectors to use different subsets of the overall checklist during preparation. One person could check that all internal document cross-references are correct, while another determines whether the requirements can serve as the basis for design and a third evaluates verifiability. Some studies have shown that giving inspectors specific defect-detection responsibilities—providing structured thought processes or scenarios to help them hunt for particular kinds of errors—is more effective than simply handing all inspectors the same checklist (Porter, Votta, and Basili 1995).

Organization and Completeness

- Are all internal cross-references to other requirements correct?
- Are all requirements written at a consistent and appropriate level of detail?
- Do the requirements provide an adequate basis for design?
- Is the implementation priority of each requirement included?
- Are all external hardware, software, and communication interfaces defined?
- Are algorithms intrinsic to the functional requirements defined?
- Does the SRS include all the known customer or system needs?
- Is any necessary information missing from a requirement? If so, is it identified as a TBD?
- Is the expected behavior documented for all anticipated error conditions?

Correctness

- Do any requirements conflict with or duplicate other requirements?
- Is each requirement written in clear, concise, and unambiguous language?
- Is each requirement verifiable by testing, demonstration, review, or analysis?
- Is each requirement in scope for the project?
- Is each requirement free from content and grammatical errors?
- Can all the requirements be implemented within known constraints?
- Are all specified error messages unique and meaningful?

Quality Attributes

- Are all performance objectives properly specified?
- Are all security and fafety considerations properly specified?
- Are other pertinent quality attribute goals explicitly documented and quantified, with the acceptable trade-offs specified?

Traceability

- Is each requirement uniquely and correctly identified?
- Is each software functional requirement traced to a higher-level requirement (for example, system requirement or use case)?

Special Issues

- Are all requirements actually requirements, not design or implementation solutions?
- Are the time-critical functions identified and their timing criteria specified?
- Have internationalization issues been adequately addressed?

Figure 15-5: Defect checklist for software requirements specifications.

Requirements Review Challenges

Peer reviews are both a technical practice and a social activity. Asking some colleagues to tell you what's wrong with your work is a learned—not instinctive—behavior. It takes time for a software organization to instill peer reviews into its culture. Following are some common challenges that an organization must face when it reviews its requirements documents, with suggestions for how to address each one (Wiegers 1998a; Wiegers 2002a).

Large requirements documents The prospect of thoroughly inspecting a several-hundred-page SRS is daunting. You might be tempted to skip the inspection entirely and just proceed with construction—not a wise choice. Even given an SRS of moderate size, all inspectors might carefully examine the first part and a few stalwarts will study the middle, but it's unlikely that anyone will look at the last part.

To avoid overwhelming the inspection team, perform incremental reviews throughout requirements development, prior to inspecting the completed document. Identify the high-risk areas that need a careful look, and use informal reviews for less risky material. Ask specific inspectors to start at different locations in the document to make certain that fresh eyes have looked at every page. Consider using several small teams to inspect different portions of the material, although this increases the chance of overlooking inconsistencies. To judge whether you really need to inspect the entire specification, examine a representative sample. The number and types of errors found will help you determine whether investing in a complete inspection is likely to pay off (Gilb and Graham 1993).

Large inspection teams Many project participants and customers hold a stake in the requirements, so you might have a long list of potential participants for requirements inspections. However, large inspection teams make it hard to schedule meetings, tend to hold side conversations during inspection meetings, and have difficulty reaching agreement on issues.

True Stories I once participated in a meeting with 13 other inspectors, which had been set up by someone who didn't understand the importance of keeping the team small. Fourteen people can't agree to leave a burning room, let alone agree on whether a particular requirement is correct. Try the following approaches to deal with a potentially large inspection team:

- Make sure every participant is there to find defects, not to be educated or to protect a political position.
- Understand which perspective (such as customer, developer, or tester) each inspector represents. Politely decline the participation of people who duplicate a perspective that's already covered. Several people who represent the same community can pool their input and send just one representative to the inspection meeting.
- Establish several small teams to inspect the SRS in parallel and combine their defect lists, removing any duplicates. Research has shown that multiple inspection teams find more defects in a requirements document than does a single large group (Martin and Tsai 1990; Schneider, Martin, and Tsai 1992; Kosman 1997). The results of parallel inspections are primarily additive rather than redundant.

Geographical separation of inspectors More and more development organizations are building products through the collaboration of geographically dispersed teams. This separation makes reviews more challenging. Videoconferencing can be an effective solution, but teleconferencing doesn't let you read the body language and expressions of other reviewers like you can in a face-to-face meeting.

Document reviews of an electronic file placed in a shared network folder provide an alternative to a traditional inspection meeting (Wiegers 2002a). In this approach, reviewers use word processor features to insert their comments into the text under review. Each comment is labeled with the reviewer's initials, and each reviewer can see what previous reviewers had to say. Web-based collaboration software embedded in tools such as ReviewPro from Software Development Technologies (*http://www.sdtcorp.com*) also can help. If you choose not to hold an inspection meeting, recognize that this can reduce the inspection's effectiveness by perhaps 25 percent (Humphrey 1989), but it also reduces the cost of the inspection.

Testing the Requirements

It's hard to visualize how a system will function under specific circumstances just by reading the SRS. Test cases that are based on the functional requirements or derived from user requirements help make the expected system behaviors tangible to the project participants. The simple act of designing test cases will reveal many problems with the requirements even if you don't execute the tests on an operational system (Beizer 1990). If you begin to develop test cases as soon as portions of the requirements stabilize, you can find problems while it's still possible to correct them inexpensively.

Trap Watch out for testers who claim that they can't begin their work until the requirements are done and for testers who claim that they don't need requirements to test the software. Testing and requirements have a synergistic relationship because they represent complementary views of the system.

Writing black box (functional) test cases crystallizes your vision of how the system should behave under certain conditions. Vague and ambiguous requirements will jump out at you because you won't be able to describe the expected system response. When analysts, developers, and customers walk through the test cases together, they'll achieve a shared vision of how the product will work.

Making Charlie Happy

True Stories I once asked my group's UNIX scripting guru, Charlie, to build a simple e-mail interface extension for a commercial defect-tracking system we were using. I wrote a dozen functional requirements that described how the e-mail interface should work. Charlie was thrilled. He'd written many scripts for people, but he'd never seen written requirements before.

Unfortunately, I waited a couple of weeks before I wrote the test cases for this e-mail function. Sure enough, I had made an error in one of the requirements. I found the mistake because my mental image of how I expected the function to work, represented in about 20 test cases, was inconsistent with one requirement. Chagrined, I corrected the defective requirement before Charlie had completed his implementation, and when he delivered the script, it was defect free. It was a small victory, but small victories add up.

You can begin deriving conceptual test cases from use cases or other user requirements representations very early in the development process (Ambler 1995; Collard 1999; Armour and Miller 2001). You can then use the test cases to evaluate textual requirements, analysis models, and prototypes. The test cases should cover the

normal course of the use case, alternative courses, and the exceptions you identified during elicitation and analysis. These conceptual (or abstract) test cases are independent of implementation. For example, consider a use case called "View Order" for the Chemical Tracking System. Some conceptual test cases would be the following:

- User enters order number to view, order exists, user placed the order. Expected result: show order details.
- User enters order number to view, order doesn't exist. Expected result: Display message "Sorry, I can't find that order."
- User enters order number to view, order exists, user didn't place the order. Expected result: Display message "Sorry, that's not your order."

Ideally, an analyst will write the functional requirements and a tester will write the test cases from a common starting point, the user requirements, as shown in Figure 15-6. Ambiguities in the user requirements and differences of interpretation will lead to inconsistencies between the views represented by the functional requirements, models, and test cases. As developers translate the requirements into user interface and technical designs, testers can elaborate the conceptual tests into detailed test procedures for formal system testing (Hsia, Kung, and Sell 1997).

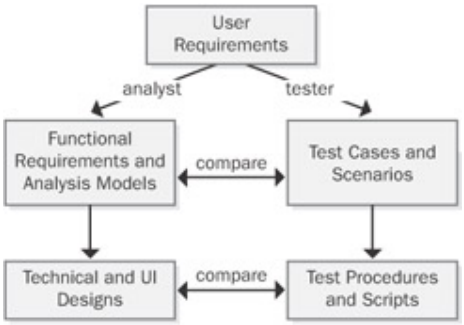


Figure 15-6: Development and testing work products are derived from a common source.

The notion of testing requirements might seem abstract to you at first. Let's see how the Chemical Tracking System team tied together requirements specification, analysis modeling, and early test-case generation. Following are a business requirement, a use case, some functional requirements, part of a dialog map, and a test case, all of which relate to the task of requesting a chemical.

Business requirement One of the primary business objectives for the Chemical Tracking System was the following:

> *This system will save the company 25 percent on chemical costs in the first year of use by allowing the company to fully exploit chemicals that are already available within the company….*

More Info This business requirement is from the vision and scope document described in Chapter 5, "Establishing the Product Vision and Project Scope."

Use case A use case that supports this business requirement is "Request a Chemical." This use case includes a path that permits the user to request a chemical container that's already available in the chemical stockroom. Here's the use-case description:

> *The Requester specifies the desired chemical to request by entering its name or chemical ID number or by importing its structure from a chemical drawing tool. The system satisfies the request either by offering the Requester a new or used container of the chemical from the chemical*

stockroom or by letting the Requester create a request to order from an outside vendor.

More Info This use case appeared in Figure 8-6.

Functional requirement Here's a bit of functionality associated with this use case:

1. If the stockroom has containers of the chemical being requested, the system shall display a list of the available containers.

2. The user shall either select one of the displayed containers or ask to place an order for a new container from a vendor.

Dialog map Figure 15-7 illustrates a portion of the dialog map for the "Request a Chemical" use case that pertains to this function. The boxes in this dialog map represent user interface displays, and the arrows represent possible navigation paths from one display to another.

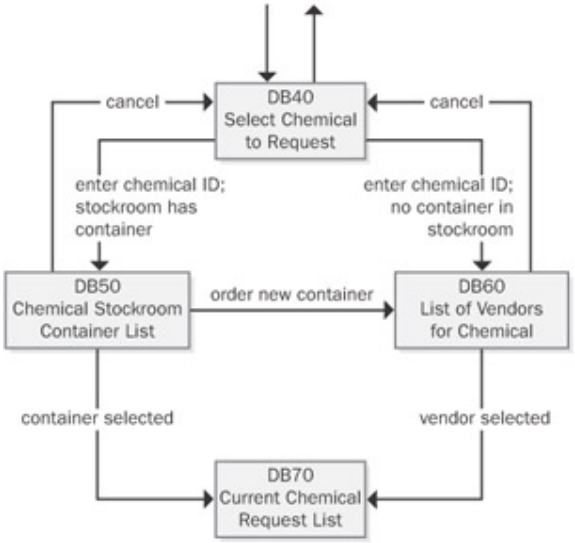


Figure 15-7: Portion of the dialog map for the "Request a Chemical" use case.
More Info See Chapter 11, "A Picture Is Worth 1024 Words," for more information on dialog maps.

Test case Because this use case has several possible execution paths, you can envision several test cases to address the normal course, alternative courses, and exceptions. The following is just one test case, based on the path that shows the user the available containers in the chemical stockroom. This test case was derived from both the use-case description of this user task and the dialog map in Figure 15-7.

At dialog box DB40, enter a valid chemical ID; the chemical stockroom has two containers of this chemical. Dialog box DB50 appears, showing the two containers. Select the second container. DB50 closes and container 2 is added to the bottom of the Current Chemical Request List in dialog box DB70.

Ramesh, the test lead for the Chemical Tracking System, wrote several test cases like this one, based on his understanding of how the user might interact with the system to request a chemical. Such abstract test cases are independent of implementation details. They don't discuss entering data into fields, clicking buttons, or other specific interaction techniques. As the user interface design activities progressed, Ramesh refined these abstract test cases into specific test procedures.

Now comes the fun part—testing requirements with the test cases. Ramesh first mapped each test case to the functional requirements. He checked to make certain that every test case could be executed by the existing set

of requirements. He also made sure that at least one test case covered every functional requirement. Next, Ramesh traced the execution path for every test case on the dialog map with a highlighter pen. The shaded line in Figure 15-8 shows how the preceding sample test case traces onto the dialog map.

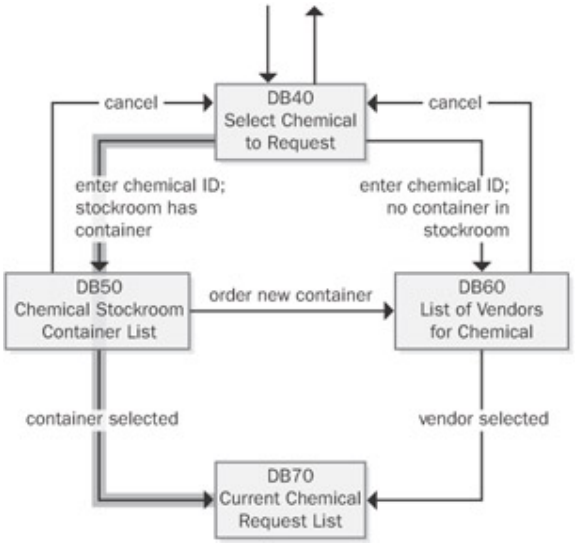


Figure 15-8: Tracing a test case onto the dialog map for the "Request a Chemical" use case.

By tracing the execution path for each test case, you can find incorrect or missing requirements, correct errors in the dialog map, and refine the test cases. Suppose that after "executing" all the test cases in this fashion, the dialog map navigation line labeled "order new container" that goes from DB50 to DB60 in Figure 15-7 hasn't been highlighted. There are two possible interpretations, as follows:

- The navigation from DB50 to DB60 is not a permitted system behavior. The analyst needs to remove that line from the dialog map. If the SRS contains a requirement that specifies the transition, that requirement must also be removed.
- The navigation is a legitimate system behavior, but the test case that demonstrates the behavior is missing.

Similarly, suppose that a test case states that the user can take some action to move directly from dialog box DB40 to DB70. However, the dialog map in Figure 15-7 doesn't contain such a navigation line, so the test case can't be executed with the existing requirements. Again, there are two possible interpretations, and you'll need to determine which of the following is correct:

- The navigation from DB40 to DB70 is not a permitted system behavior, so the test case is wrong.
- The navigation from DB40 to DB70 is a legitimate function, but the dialog map and perhaps the SRS are missing the requirement that allows you to execute the test case.

In these examples, the analyst and the tester combined requirements, analysis models, and test cases to detect missing, erroneous, or unnecessary requirements long before any code was written. Conceptual testing of software requirements is a powerful technique for controlling a project's cost and schedule by finding requirement ambiguities and errors early in the game. Every time I use this technique, I find errors in all the items I'm comparing to each other. As Ross Collard (1999) pointed out,

> *Use cases and test cases work well together in two ways: If the use cases for a system are complete, accurate, and clear, the process of deriving the test cases is straightforward. And if the use cases are not in good shape, the attempt to derive test cases will help to debug the use cases.*

Defining Acceptance Criteria

Software developers might believe that they've built the perfect product, but the customer is the final arbiter. Customers perform acceptance testing to determine whether a system satisfies its *acceptance criteria* (IEEE 1990). If it does, the customer can pay for a product developed under contract or the user can cut over to a new corporate information system. Acceptance criteria—and hence acceptance testing—should evaluate whether the product satisfies its documented requirements and whether it is fit for use in the intended operating environment (Hsia, Kung, and Sell 1997). Having users devise acceptance tests is an effective requirements development strategy. The earlier in the development process that users write acceptance tests, the sooner they can begin to filter out defects in the requirements and in the implemented software.

Acceptance testing should focus on anticipated usage scenarios (Steven 2002; Jeffries, Anderson, and Hendrickson 2001). Key users should consider the most commonly used and most important use cases when deciding how to evaluate the software's acceptability. Acceptance tests focus on the normal courses of the use cases, not on the less common alternative courses or whether the system handles every exception condition properly. Automate acceptance tests whenever possible. This makes it easier to repeat the tests when changes are made and additional functionality is added in future releases. Acceptance tests also ought to address nonfunctional requirements. They should ensure that performance goals are achieved on all platforms, that the system complies with usability standards, and that all committed user requirements are implemented.

Trap User acceptance testing does not replace comprehensive requirements-based system testing, which covers both normal and exception paths and a wide variety of data combinations.

Having customers develop acceptance criteria thus provides another opportunity to validate the most important requirements. It's a shift in perspective from the requirements-elicitation question of "What do you need to do with the system?" to "How would you judge whether the system satisfies your needs?" If the customer can't express how she would evaluate the system's satisfaction of a particular requirement, that requirement is not stated sufficiently clearly.

Writing requirements isn't enough. You need to make sure that they're the right requirements and that they're good enough to serve as a foundation for design, construction, testing, and project management. Acceptance test planning, informal reviews, SRS inspections, and requirements testing techniques will help you to build higher-quality systems faster and more inexpensively than you ever have before.

Next Steps

- Choose a page of functional requirements at random from your project's SRS. Ask a group of people who represent different stakeholder perspectives to carefully examine that page of requirements for problems, using the defect checklist in Figure 15-5.
- If you found enough errors during the random sample review to make the reviewers nervous about the overall quality of the requirements, persuade the user and development representatives to inspect the entire SRS. For maximum effectiveness, train the team in the inspection process.
- Define conceptual test cases for a use case or for a portion of the SRS that hasn't yet been coded. See whether the user representatives agree that the test cases reflect the intended system behavior. Make sure you've defined all the functional requirements that will permit the test cases to be executed and that there are no superfluous requirements.

- Work with your product champions to define the criteria that they and their colleagues will use to assess whether the system is acceptable to them.

Chapter 16: Special Requirements Development Challenges

Overview

This book has described requirements development as though you were beginning a new software or system development project, sometimes called a *green-field project*. However, many organizations devote most of their effort to maintaining existing legacy systems or building the next release of an established commercial product. Other organizations build few new systems from scratch, instead integrating, customizing, and extending commercial off-the-shelf (COTS) products to meet their needs. Still others outsource their development efforts to contract development companies. This chapter describes appropriate requirements practices for these types of projects, as well as for emergent projects with volatile and uncertain business needs.

Requirements for Maintenance Projects

Maintenance refers to modifications made to software that is currently in operation. Sometimes called *continuing engineering* or *ongoing development*, maintenance often consumes the majority of a software organization's resources. Maintenance programmers typically correct defects, add new features or reports to existing systems, and modify functionality to comply with revised business rules. Few legacy systems have adequate documentation. The original developers who held all the critical information in their heads are often long gone. The approaches described in this chapter can help you to deal with requirements for ongoing maintenance and support projects to improve both product quality and your ability to perform future maintenance (Wiegers 2001).

The Case of the Missing Spec

True Stories The requirements specification for the next release of a mature system often says essentially, "The new system should do everything the old system does, except add these new functions and fix those bugs." An analyst once received just such a specification for version 5 of a major product. To find out exactly what the current release did, she looked at the SRS for version 4. Unfortunately, it also said in essence, "Version 4 should do everything that version 3 does, except add these new functions and fix those bugs." She followed the trail back but never found a real requirements specification. Every SRS described the differences that the new version should exhibit compared to the previous version, but nowhere was there a description of the original system. Consequently, everyone had a different understanding of the current system's capabilities. If you're in this situation, document the requirements for the next release more thoroughly so that all the stakeholders understand what the system does.

Begin Capturing Information

In the absence of accurate requirements documentation, maintainers must reverse-engineer an understanding of what the system does from the code. I think of this as "software archaeology." To maximize the benefit from reverse engineering, the archaeology expedition should record what it learns in the form of requirements or design descriptions. Accumulating accurate information about certain portions of the current system positions the team to perform future enhancements more efficiently.

Perhaps your current system is a shapeless mass of history and mystery like the one in Figure 16-1. Imagine that you've been asked to implement some new functionality in region *A* in this figure. Begin by recording the new requirements in a structured, if incomplete, SRS or in a requirements management tool. When you add the new functionality, you'll have to figure out how new screens and features will interface to the existing system. The bridges in Figure 16-1 between region *A* and your current system represent these interfaces. This analysis provides insight into the white portion of the current system, region *B*. That insight is the new knowledge you need to capture.

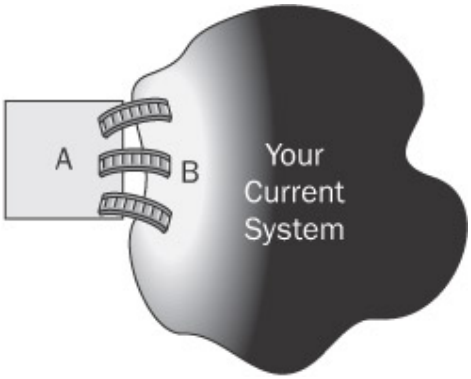


Figure 16-1: Adding enhancement *A* to an ill-documented legacy system provides some visibility into the *B* area.

One useful technique is to draw a dialog map for the new screens you have to add, showing the navigation connections to and from existing display elements. Other modeling techniques such as class and interaction diagrams, data flow diagrams, and entity-relationship diagrams are also useful. A context diagram or use-case diagram depicts the external entities or actors that interact with the system. Another way to begin filling the information void is to create data dictionary entries when you add new data elements to the system and modify existing definitions.

Building a requirements representation that includes portions of the current system achieves three useful goals:

- It halts the knowledge drain, so future maintainers better understand the changes that were just made.
- It collects some information about the current system that previously was undocumented. When the team performs additional maintenance over time, it can extend these fractional knowledge representations, thereby steadily improving the system documentation. The incremental cost of recording this newly found knowledge is small compared with the costs of someone having to rediscover it later on.
- It provides an indication of functional coverage by the current set of system tests. (Conversely, the test suite will be useful as an initial source of information to recover the software requirements.)

If you have to perform software surgery because of a new or revised government regulation or company policy, begin listing the business rules that affect the system. If you don't do this, vital information will remain distributed among many team members' brains. Information fragmentation contributes to wasted effort as people work from different interpretations of the important business influences on your project.

True Stories Many problems show up at interfaces, including software-to-software, software-to-hardware, and software-to-people junctures. When you implement a change, take the time to document what you discovered about the current system's external user, software, hardware, and communication interfaces. Architectural information about how the system's components fit together can help you to add future extensions safely and efficiently. A contractor once had to add a major enhancement to a robot-control system that my team had built. Our existing design models got him up to speed quickly, so he could efficiently mesh the new capabilities with the existing system architecture. Without this information, the contractor would have had to reverse-engineer the code to understand the architecture and design.

It's not always worth taking the time to generate a complete SRS for an entire production system. Many options lie between the two extremes of continuing forever with no requirements documentation and reconstructing a perfect SRS. Knowing why you'd like to have written requirements available lets you judge whether the cost of rebuilding all or part of the specification is a sound investment. The earlier you are in the product life cycle, the more worthwhile it will be to tune up the requirements.

True Stories I worked with one team that was just beginning to develop the requirements for version 2 of a major product with embedded software. They hadn't done a good job on the requirements for version 1, which was currently being implemented. The lead requirements analyst wondered, "Is it worth going back to improve the SRS for version 1?" The company anticipated that this product line would be a major revenue generator for at least 10 years. They also planned to reuse some of the core requirements in several spin-off products. In this case, it made sense to improve the requirements documentation for version 1 because it was the foundation for all subsequent development work in this product family. Had they been working on version 5.3 of a legacy system that they expected to retire within a year, reconstructing an accurate SRS wouldn't be a wise investment.

From Code to Requirements to Tests

True Stories A. Datum Corporation needed a comprehensive set of test cases for their flagship product, a complex and mission-critical financial application that was several years old. They decided to reverse-engineer use cases from the existing code and then derive the test cases from those use cases.

First, an analyst reverse-engineered class diagrams from the production software by using a tool that could generate models from source code. Next, the analyst wrote use-case descriptions for the most common user tasks, some of which were highly complex. These use cases addressed all the scenario variations that users performed, as well as many exception conditions. The analysts drew UML sequence diagrams to link the use cases to the system classes (Armour and Miller 2001). The final step was to manually generate a rich set of test cases to cover all the normal and exceptional courses of the use cases. When you create requirements and test artifacts by reverse engineering, you can be confident that they reflect the system as actually built and its known usage modes.

Practice New Requirements Techniques

All software development is based on requirements, perhaps just in the form of a simple change request. Maintenance projects provide an opportunity to try new requirements engineering methods in a small-scale and low-risk way. It's tempting to claim that a small enhancement doesn't warrant writing any requirements. The pressure to get the next release out might make you think that you don't have time for requirements. But enhancement projects let you tackle the learning curve in bite-sized chunks. When the next big project comes along, you'll have some experience and confidence in better requirements practices.

Suppose that marketing or a customer requests that a new feature be added to a mature product. Explore the new feature from the use-case perspective, discussing with the requester the tasks that users will perform with that feature. If you haven't worked with use cases before, try documenting the use case according to a standard

template, such as that illustrated in Figure 8-6. Practicing reduces the risk of applying use cases for the first time on a green-field project, when your skill might mean the difference between success and high-profile failure. Some other techniques that you can try on a small scale during maintenance include:

- Creating a data dictionary (Chapter 10)
- Drawing analysis models (Chapter 11)
- Specifying quality attributes and performance goals (Chapter 12)
- Building user interface and technical prototypes (Chapter 13)
- Inspecting requirements specifications (Chapter 15)
- Writing test cases from requirements (Chapter 15)
- Defining customer acceptance criteria (Chapter 15)

Follow the Traceability Chain

Requirements traceability data will help the maintenance programmer determine which components he might have to modify because of a change in a specific requirement. However, a poorly documented legacy system won't have traceability information available. When someone on your team has to modify the existing system, he should create a partial requirements traceability matrix to link the new or changed requirements to the corresponding design elements, code, and test cases. Accumulating traceability links as you perform the development work takes little effort, whereas it's nearly impossible to regenerate the links from a completed system.

More Info Chapter 20, "Links in the Requirements Chain," describes requirements traceability.

Because of the large ripple effect of many software modifications, you need to make sure that each requirement change is correctly propagated into the downstream work products. Inspections are a good way to check for this consistency. Chapter 15 described the following four perspectives that the inspection participants should represent; these also apply to maintenance work.

- The author of the requirements for the modification or enhancement
- The requesting customer or a marketing representative who can ensure that the new requirements accurately describe the needed change
- Developers, testers, or others who will have to do work based on the new requirements
- People whose work products might be affected by the change

Update the Documentation

Any existing requirements representations must be kept current during maintenance to reflect the capabilities of the modified system. If the updating is burdensome, it will quickly fall by the wayside as busy people rush on to the next change request. Obsolete requirements and design documents aren't helpful for future maintenance. There's a widespread fear in the software industry that writing documentation will consume too much time; the knee-jerk reaction is to neglect all updating of requirements documentation. But what's the cost if you *don't* update the requirements and a future maintainer (perhaps you!) has to regenerate that information? The answer to this question will let you make a thoughtful business decision concerning whether to revise the

requirements documentation when you change the software.

Requirements for Package Solutions

Even if you're purchasing a commercial off-the-shelf package as part or all of the solution for a new project, you still need requirements. COTS products typically need to be configured, customized, integrated, and extended to work in the target environment. These activities demand requirements. Requirements also let you evaluate the candidates to identify the most appropriate package for your needs. One evaluation approach includes the following activities (Lawlis et al. 2001):

- Weight your requirements on a scale of 0 to 10 to distinguish their importance.
- Rate each candidate package as to how well it satisfies each requirement. Use a rating of 1 for full satisfaction, 0.5 for partial satisfaction, and 0 for no coverage. You can find the information to make this assessment from product literature, a vendor's response to a request for proposal (an invitation to bid on a project), or direct examination of the product.
- Evaluate each package for the nonfunctional requirements that are important to your users, such as performance and usability, as well as the other quality attributes listed in Chapter 12, "Beyond Functionality: Software Quality Attributes."
- Evaluate product cost, vendor viability, vendor support for the product, external interfaces that will enable extension and integration, and compliance with any technology requirements or constraints for your environment.

This section describes several ways to approach requirements definition when you plan to acquire a commercial package to meet your needs, as shown in Figure 16-2.

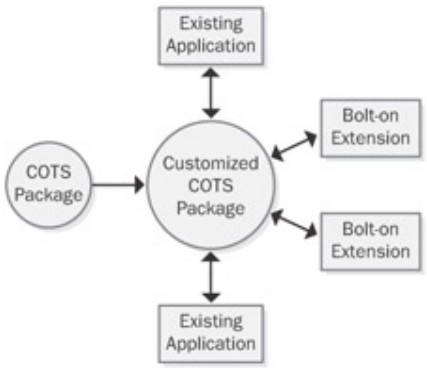


Figure 16-2: COTS packages are customized, integrated into the existing application environment, and extended with bolt-ons.

Develop Use Cases

There's little point in specifying detailed functional requirements or designing a user interface if you're going to buy an existing product. Focus the requirements for COTS acquisition at the user requirements level. Use cases work well for this purpose. Any package you select must let your users accomplish their task objectives, although each package will do so in a different way. The use cases permit a gap analysis, so you can see the places where customization or extension will be needed to meet your needs. Many packages purport to provide

canned solutions for some portion of your enterprise information-processing needs. Therefore, specify the reports that the COTS product must generate and determine to what extent the product will let you customize its standard reports. No package solution is likely to accommodate every use case you identify, so prioritize the use cases and reports. Distinguish capabilities that must be available on day one from those that can wait for future extensions and those that your users can live without.

To determine whether—and how well—the package will let the users perform their tasks, derive test cases from the high-priority use cases. Include test cases that explore how the system handles significant exception conditions that might arise. A similar approach is to run the COTS product through a suite of scenarios that represent the expected usage patterns, called an *operational profile* (Musa 1996).

Consider Business Rules

Your requirements exploration should identify pertinent business rules to which the COTS product must conform. Can you configure the package to comply with your corporate policies, industry standards, or government regulations? How easily can you modify the package when these rules change? Does it generate mandated reports in the correct formats? You might also define the data structures required to satisfy your use cases and business rules. Look for major disconnects between your data structures and the package vendor's data model.

Some packages incorporate global business rules, such as income tax withholding computations or printed tax forms. Do you trust that these are implemented correctly? Will the package vendor provide you with new versions when those rules and computations change? If so, how quickly? Will the vendor supply a list of the business rules the package implements? If the product implements any intrinsic business rules that don't apply to you, can you disable or modify them?

Define Quality Requirements

The quality attribute and performance goals discussed in Chapter 12 are another aspect of user requirements that feeds into package solution selection. Explore at least the following attributes:

- **Performance** What maximum response times are acceptable for specific operations? Can the package handle the anticipated load of concurrent users and transaction throughput?
- **Usability** Does the package conform to any established user interface conventions? Is the interface similar to that used in other applications with which the users are familiar? How easily can your customers learn to use the package?
- **Flexibility** How hard will it be for your developers to modify or extend the package to meet your specific needs? Does the package provide appropriate "hooks" (connection and extension points) and application programming interfaces for adding extensions?
- **Interoperability** How easily can you integrate the package with your other enterprise applications? Does it use standard data interchange formats?
- **Integrity** Does the package permit control over which users are allowed to access the system or use specific functions? Does it safeguard data from loss, corruption, or unauthorized access? Can you define various user privilege levels?

COTS packages give the acquiring organization less flexibility over requirements than does custom development. You need to know which requested capabilities aren't negotiable and which you can adjust to fit within the package's constraints. The only way to choose the right package solution is to understand your user and the business activities the package must let them perform.

Requirements for Outsourced Projects

Contracting product development to a separate company demands high-quality written requirements because your direct interactions with the engineering team are likely to be minimal. You'll be sending the supplier a request for proposal, a requirements specification, and some product acceptance criteria, and they'll return the finished product and supporting documentation, as shown in Figure 16-3.

Figure 16-3: Requirements are the cornerstone of an outsourced project.

You won't have the opportunity for the day-to-day clarifications, decision making, and changes that you experience when developers and customers work in close proximity. Poorly defined and managed requirements are a common cause of outsourced project failure (Wiegers 2003). Keep in mind the suggestions in the following sections when you prepare requirements for outsourcing.

Provide the details. If you distribute a request for proposal, suppliers need to know exactly what you're requesting before they can produce realistic responses and estimates (Porter-Roth 2002).

True Stories **Avoid ambiguity.** Watch out for the ambiguous terms from Table 10-1 that cause so much confusion. I once read an SRS intended for outsourcing that contained the word *support* in many places. However, a contractor who was going to implement the software wouldn't know what *support* meant in each case. It's the requirements author's responsibility to express the acquirer's intentions clearly.

Arrange touch points with the supplier. In the absence of real-time, face-to-face communication, you need other mechanisms to stay on top of what the supplier is doing. Peer reviews and prototypes provide insight into how the supplier is interpreting the requirements. Incremental development, in which the supplier delivers small portions of the system periodically, is a risk-management technique. This permits quick course corrections when a misunderstanding sends the supplier's developers in the wrong direction.

Trap Don't assume that suppliers will interpret ambiguous and incomplete requirements the same way that you do. Some suppliers will interpret the requirements literally and will build precisely what the acquirer specifies. The burden is on the acquirer to communicate all necessary information to the supplier, using frequent conversations to resolve requirements questions.

Define a mutually acceptable change-control process. Some 45 percent of outsourced projects are at risk from creeping user requirements (Jones 1994). Change always has a price, so using change management to

control scope creep is vital in a contract development situation. The contract should specify who will pay for various kinds of changes, such as newly requested functionality or corrections made in the original requirements.

True Stories **Plan time for multiple cycles and reviews of requirements.** The project schedule for one failed outsourced project included a one-week task named "Hold requirements workshops," followed by tasks to implement several subsystems. The supplier forgot to include vital intermediate tasks to document, review, and revise the requirements specifications. The iterative and communication-intensive nature of requirements development dictates that you must allow time for these review cycles. The acquirer and the supplier on this project were a continent apart. They experienced slow turnaround on the myriad questions that arose as the SRS cycled back and forth. Failure to resolve requirements issues in a timely way derailed the schedule and eventually helped to send the two parties into litigation.

Define acceptance criteria. In keeping with Stephen Covey's recommendation to "begin with the end in mind" (Covey 1989), define how you'll assess whether the contracted product is acceptable to you and your customers. How will you judge whether to make the final payment to the supplier?

More Info Chapter 15 suggested some approaches to defining acceptance criteria.

Requirements for Emergent Projects

When I hear the phrase "gathering requirements," I get a mental image of looking for requirements lying around the office, like flowers that we can pick and place in a basket. If only it were that simple! On some projects, typically those in well-established problem domains, it is indeed possible to specify much of the intended functionality early on. It makes sense to have a concerted requirements development process up front for such projects.

True Stories On more exploratory or volatile projects, however, the intended system capabilities become known only over time. Such *emergent* projects are characterized by uncertain requirements and frequent changes. They demand iterative, incremental, and adaptive approaches to both requirements development and software development (Highsmith 2000). Rapidly moving projects with evolving market needs often shy away from established requirements engineering approaches. Instead, they might base development on vague, high-level statements of business objectives. I once spoke to the president of an Internet development company that had encountered many requirements-related problems. One reason was that the development team went directly from signing a contract with a client into a full-day visual design workshop. They didn't take the time to understand how users would be able to obtain value from the Web site, which resulted in considerable rework.

The prevalence of emergent and fast-moving projects has led to the creation of various *agile development* methodologies. These methodologies emphasize rapidly putting useful functionality into the hands of users (Gilb 1988; Beck 2000; Cockburn 2002). The agile methodologies take a lean approach to requirements development and an informal approach to requirements management. Requirements are described in terms of product features or in the form of *user stories*, which are similar to the casual use cases described in Chapter 8, "Understanding User Requirements." Detailed requirements documentation is rejected in favor of continuous interaction with an on-site customer representative, who supplies the requirement details and answers questions.

The agile development philosophy views software change as both inevitable and desirable. The system evolves in response to customer feedback and changing business needs. To cope with change, systems are built in small increments with the expectation that subsequent increments will modify what already exists, enrich the initial

features, and add new ones. This approach works well for dealing with a high degree of requirements uncertainty in information systems or Internet projects. It's less well suited to applications whose requirements are well understood and to embedded systems development.

Clearly, development work must be based on some understanding of what users want to accomplish with the software. The requirements for rapidly changing projects are too unstable to justify investing a lot of requirements development effort up front. As Jeff Gainer (1999) points out, though, "The difficulty for the software developers, however, is that iterative development cycles actually encourage requirements change and scope creep." Properly managed, these cycles help the software to meet current business needs, albeit at the expense of reworking previously completed designs, code, and tests.

Stakeholders sometimes claim that the requirements are unknown and unknowable in their desire to dive right into coding, disregarding the potential need for extensive recoding. Resist the temptation to use "Internet time" as an excuse to skimp on sound requirements development. If you really are working on an emergent project, consider using the following approaches to requirements.

Casual User Requirements Specification

Informally documented requirements are appropriate for evolving systems being built by small, co-located teams. The agile methodology called Extreme Programming advocates recording requirements in the form of simple user stories written on index cards (Beck 2000; Jeffries, Anderson, and Hendrickson 2001). Note that even this approach demands that the customers understand their requirements well enough to describe the system behavior in the form of stories.

Trap Don't expect unwritten requirements communicated telepathically to suffice for project success. Every project should represent its requirements in forms that can be shared among the stakeholders, be updated, and be managed throughout the project. Someone needs to be responsible for this documenting and updating.

On-Site Customer

Frequent conversations between project team members and appropriate customers are the most effective way to resolve many requirements issues. Written documentation, however detailed, is an incomplete substitute for these ongoing communications. A fundamental tenet of Extreme Programming is the presence of a full-time, on-site customer for these discussions. As we saw in Chapter 6, "Finding the Voice of the Customer," though, most projects have multiple user classes, as well as additional stakeholders.

On-Sight Customer

True Stories I once wrote programs for a research scientist who sat about 10 feet from my desk. John could provide instantaneous answers to my questions, give feedback on user interface displays, and clarify our informally written requirements. One day John moved to a new office on the same floor of the same building. I perceived an immediate drop in my programming productivity because of the cycle time delay in getting John's input. I spent more time correcting problems because sometimes I proceeded down the wrong path before I could get a course correction. There's no substitute for having the right customers continuously available to the developers both on-site and on-sight. Beware, though, of too-frequent interruptions that make it hard for people to refocus their attention on their work. It can take 15 minutes to reimmerse yourself into the highly productive, focused state of mind called *flow* (DeMarco and Lister 1999).

True Stories An on-site customer doesn't guarantee the desired outcome. My colleague Chris helped to establish a development team environment with minimal physical barriers and engaged two product

champions. Chris offered this report: "While the close proximity seems to work for the development team, the results with product champions have been mixed. One sat in our midst and still managed to avoid us all. The current champion does a fine job of interacting with the developers and has truly enabled the rapid development of software."

Trap Don't expect a single individual to resolve all the requirements issues that arise. The product champion approach, with a small number of user representatives, is a more effective solution.

Early and Frequent Prioritization

Incremental development succeeds when customers and developers collaborate on choosing the sequence of implementing features. The development team's goal is to get useful functionality and quality enhancements into the users' hands on a regular basis, so they need to know what capabilities are planned for each increment.

Simple Change Management

Your software development processes should be as simple as they can be to get the job done right, but no simpler. A sluggish change-control process won't work on emergent projects that demand frequent modifications. Streamline your change process so that the minimum number of people can make decisions about requested changes as quickly as possible. This doesn't mean every developer should simply make any changes he likes or that customers request, though; that's a path to chaos, not rapid delivery.

Next Steps

- If you're working on a maintenance, package-solution, outsourced, or emergent project, study the requirements engineering good practices from Chapter 3, "Good Practices for Requirements Engineering," to see which ones would add value to your project and increase its chance of success.
- If you had problems related to requirements on previous projects of the types described in this chapter, perform a project retrospective to diagnose the problems and identify root causes. Would any of the good practices from Chapter 3 or the approaches described in this chapter prevent those same problems from hampering your next project of that type?

Chapter 17: Beyond Requirements Development

Overview

The Chemical Tracking System was coming along nicely. The project sponsor, Gerhard, and the chemical stockroom product champion, Roxanne, had been skeptical of the need to spend much time defining requirements. However, they joined the development team and other product champions at a one-day training class on software requirements. This class stressed the importance of having all project stakeholders reach a shared understanding of their business objectives and the users' needs. The class exposed all the team members to the requirements terminology, concepts, and practices they'd be using. It also motivated them to put some improved requirements techniques into action.

As the project progressed, Gerhard received excellent feedback from the user representatives about how well requirements development had gone. He even sponsored a luncheon for the analysts and product champions to

celebrate reaching the significant milestone of baselined requirements for the first system release. At the luncheon, Gerhard thanked the requirements elicitation participants for their contributions and teamwork. Then he said, "Now that the requirements are in good order, I look forward to seeing some code coming out of the development group very soon."

"We aren't quite ready to start writing the production code yet," the project manager explained. "We plan to release the system in stages, so we need to design the system to accommodate the future additions. Our prototypes provided good ideas about technical approaches and helped us understand the interface characteristics the users prefer. If we spend some time now on design, we can avoid problems as we add functionality to the system over the next several months."

Gerhard was frustrated. It looked like the development team was stalling rather than getting down to the real work of programming. But was he jumping the gun?

Experienced project managers and developers understand the value of translating software requirements into robust designs and rational project plans. These steps are necessary whether the next release represents 1 percent or 100 percent of the final product. This chapter explores some approaches for bridging the gap between requirements development and a successful product release. We'll look at several ways in which requirements influence project plans, designs, code, and tests, as shown in Figure 17-1.

Figure 17-1: Requirements drive project planning, design, coding, and testing activities.

From Requirements to Project Plans

Because the requirements define the project's intended outcomes, you should base project plans, estimates, and schedules on the requirements. Remember, though, that the most important project outcome is a system that meets its business objectives, not necessarily one that implements all the initial requirements according to the original project plan. The requirements and plans represent the team's initial assessment of what it will take to achieve that outcome. But the project's scope might have been off target, or the initial plans might not have been realistic. Also, business needs, business rules, and project constraints all can change. The project's business success will be problematic if you don't update your plans to align with evolving objectives and realities.

Project managers often wonder how much of their schedule and effort they should devote to requirements engineering. Small projects that my teams worked on typically spent 12 to 15 percent of their total effort on requirements engineering (Wiegers 1996a), but the appropriate percentage depends on the size and complexity of the project. Despite the fear that exploring requirements will slow down a project, considerable evidence shows that taking time to understand the requirements actually accelerates development, as the following examples illustrate:

- A study of 15 projects in the telecommunications and banking industries revealed that the most

successful projects spent 28 percent of their resources on requirements engineering (Hofmann and Lehner 2001). Eleven percent of this work went into requirements elicitation, 10 percent into modeling, and 7 percent into validation and verification. The average project devoted 15.7 percent of its effort and 38.6 percent of its schedule to requirements engineering.

- NASA projects that invested more than 10 percent of their total resources on requirements development had substantially lower cost and schedule overruns than projects that devoted less effort to requirements (Hooks and Farry 2001).

- In a European study, teams that developed products more quickly devoted more of their schedule and effort to requirements than did slower teams, as shown in Table 17-1 (Blackburn, Scudder, and Van Wassenhove 1996).

Table 17-1: Investing in Requirements Accelerates Development

	Effort Devoted to Requirements	**Schedule Devoted to Requirements**
Faster Projects	14%	17%
Slower Projects	7%	9%

Not all of the requirements development effort should be allocated to the beginning of the project, as is done in the waterfall, or sequential, life cycle model. Projects that follow an iterative life cycle model will spend time on requirements during every iteration through the development process. Incremental development projects aim to release functionality every few weeks, so they will have frequent but small requirements development efforts. Figure 17-2 illustrates how different life cycle models allocate requirements effort across the product development period.

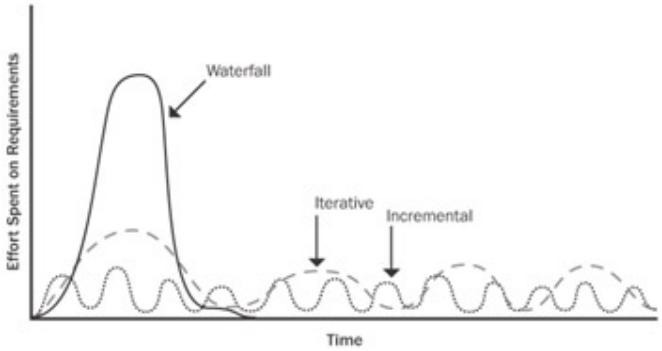


Figure 17-2: The distribution of requirements effort over time varies for projects that follow different development life cycle models.

Trap Watch out for analysis paralysis. A project with massive up-front effort aimed at perfecting the requirements "once and for all" often delivers little useful functionality in an appropriate time frame. On the other hand, don't avoid requirements development because of the specter of analysis paralysis.

Requirements and Estimation

The first step in project estimating is to assess the size of the software product. You can base size estimates on textual requirements, analysis models, prototypes, or user interface designs. Although there's no perfect measure of software size, the following are some frequently used metrics:

- The number of individually testable requirements (Wilson 1995)

- Function points and feature points (Jones 1996b) or 3-D function points that incorporate data, function, and control (Whitmire 1995)

- The number, type, and complexity of graphical user interface (GUI) elements
- Estimated lines of code needed to implement specific requirements
- A count of object classes or other object-oriented metrics (Whitmire 1997)

All these methods can work for estimating size. Base whatever approach you choose on your experience and on the nature of the software you're developing. Understanding what the development team has successfully achieved on similar projects using similar technologies lets you gauge team productivity. Once you have a size estimate, you can use a commercial estimating tool that suggests feasible combinations of development effort and schedule. These tools let you adjust estimates based on factors such as the skill of the developers, project complexity, and the team's experience in the application domain. Information about some of the available software estimation tools is available at *http://www.laatuk.com/tools/effort_estimation_tools.html*.

If you don't compare your estimates to the actual project results and improve your estimating ability, your estimates will forever remain guesses. It takes time to accumulate enough data to correlate some measure of software size with development effort. Your objective is to develop equations so that you can reliably judge the size of the completed software from its requirements.

Even the best estimating process will break down if your project must cope with requirements that customers, managers, or lawmakers frequently change. If the changes are so great that the analyst and the development team can't keep up, the project team can become paralyzed, unable to make meaningful progress. In that case, perhaps the project should be deferred until the customer needs are clearer.

Imposed delivery deadlines are a common frustration for project managers. Any time an imposed deadline and a carefully estimated schedule don't agree, negotiation is in order. A project manager who can justify an estimate based on a well-thought-out process and historical data is in a better bargaining position than someone who simply makes his best guess. If the stakeholders can't resolve the schedule conflict, the project's business objectives should dictate whether to reduce scope, add resources, or compromise on quality. These decisions aren't easy, but making them is the only way to maximize the delivered product value.

Got an Hour?

True Stories A customer once asked our development team to port a tiny program that he had written for his personal use to our shared computer network so that his colleagues could also use it. "Got an hour?" my manager asked me, giving his top-of-the-head assessment of the project's size. When I spoke with the customer and his colleagues to understand what they needed, the problem turned out to be a bit larger. I spent 100 hours writing the program they were looking for, without any gold plating. A 100-fold expansion factor suggested that my manager's initial estimate of one hour was a trifle hasty and was based on incomplete information. The analyst should explore the requirements, evaluate scope, and judge size *before* anyone makes estimates or commitments.

Uncertain requirements inevitably lead to uncertain estimates of size, effort, and schedule. Because requirements uncertainty is unavoidable early in the project, include contingency buffers in your schedule and budget to accommodate some requirements growth (Wiegers 2002d). Scope growth takes place because business situations change, users and markets shift, and stakeholders reach a better understanding of what the software can or should do. Extensive requirements growth, however, usually indicates that many requirements were missed during elicitation.

Spend some time selecting appropriate size metrics for the kinds of projects your teams work on. Complex, nonlinear relationships exist between product size, effort, development time, productivity, and staff buildup time (Putnam and Myers 1997). Understanding these relationships can keep you from being trapped in the

"impossible region," combinations of product size, schedule, and staff where no similar project has ever been completed successfully.

Trap Don't let your estimates be swayed by what you think someone else wants to hear. Your prediction of the future doesn't change just because someone doesn't like it. Too large a mismatch in predictions indicates the need for negotiation, though.

Requirements and Scheduling

Many projects practice "right-to-left scheduling": a delivery date is cast in concrete and then the product's requirements are defined. In such cases, it often proves impossible for the developers to meet the specified ship date while including all the demanded functionality at the expected quality level. It's more realistic to define the software requirements *before* making detailed plans and commitments. A design-to-schedule strategy can work, however, if the project manager can negotiate what portion of the desired functionality can fit within the schedule constraints. As always, requirements prioritization is a key project success factor.

For complex systems in which software is only a part of the final product, high-level schedules are generally established after developing the product-level (system) requirements and a preliminary architecture. At this point, the key delivery dates can be established and agreed on, based on input from sources including marketing, sales, customer service, and development.

Consider planning and funding the project in stages. An initial requirements exploration stage will provide enough information to let you make realistic plans and estimates for one or more construction stages. Projects that have uncertain requirements benefit from an incremental development life cycle. Incremental development lets the team begin delivering useful software long before the requirements become fully clear. Your prioritization of requirements dictates the functionality to include in each increment.

Software projects frequently fail to meet their goals because the developers and other project participants are poor planners, not because they're poor software engineers. Major planning mistakes include overlooking common tasks, underestimating effort or time, failing to account for project risks, not anticipating rework, and deceiving yourself with unfounded optimism. Effective project planning requires the following elements:

- Estimated product size
- Known productivity of the development team, based on historical performance
- A list of the tasks needed to completely implement and verify a feature or use case
- Reasonably stable requirements
- Experience, which helps the project manager adjust for intangible factors and the unique aspects of each project

  Trap Don't succumb to pressure to make commitments that you know are unachievable. This is a recipe for a lose-lose outcome.

PREVIOUS NEXT
PREVIOUS NEXT

From Requirements to Designs and Code

The boundary between requirements and design is not a sharp line, but keep your specifications free from

implementation bias except when you have a compelling reason to intentionally constrain the design. Ideally, the descriptions of what the system is intended to do should not be slanted by design considerations (Jackson 1995). Practically speaking, many projects contain design constraints from prior products, and backward compatibility is a common requirement. Because of this, a requirements specification almost always contains some design information. However, the specification should not contain *inadvertent* design—that is, needless or unintended restrictions on the final design. Include designers or developers in requirements inspections to make sure that the requirements can serve as a solid foundation for design.

A product's requirements, quality attributes, and user characteristics drive its architecture (Bass, Clements, and Kazman 1998). Studying a proposed architecture provides a different perspective that helps to verify the requirements and tune their precision, as does prototyping. Both methods use the following thought process: "If I understand the requirements correctly, this approach I'm reviewing is a good way to satisfy them. And now that I have a preliminary architecture (or a prototype) in hand, does it help me better understand the requirements?"

Architecture is especially critical for systems that include both software and hardware components and for complex software-only systems. An essential requirements analysis step is to allocate the high-level system requirements to the various subsystems and components. A system engineer, an analyst, or an architect decomposes the system requirements into functional requirements for both the software and the hardware subsystems (Leffingwell and Widrig 2000). Traceability information lets the development team track where each requirement is addressed in the design.

True Stories Poor allocation decisions can result in the software being expected to perform functions that should have been assigned to hardware components (or the converse), in poor performance, or in the inability to replace one component easily with an improved version. On one project, the hardware engineer blatantly told my group that he expected our software to overcome all limitations of his hardware design! Although software is more malleable than hardware, engineers shouldn't use software's flexibility as a reason to skimp on hardware design. Take a system engineering approach to make optimal decisions about which capabilities each system component will satisfy within its imposed constraints.

Allocation of system capabilities to subsystems and components must be done from the top down (Hooks and Farry 2001). Consider a DVD player, which includes motors to open and close the disk tray and to spin the disk, an optical subsystem to read the data on the disk, image-rendering software, a multifunction remote control, and more, as shown in Figure 17-3. The subsystems interact to control the behaviors that result when, say, the user presses a button on the remote control to open the disk tray while the disk is playing. The system requirements drive the architecture design for such complex products, and the architecture influences the requirements allocation.

Figure 17-3: Complex products such as DVD players contain multiple software and hardware subsystems.

Software design receives short shrift on some projects, yet the time spent on design is an excellent investment. A variety of software designs will satisfy most products' requirements. These designs will vary in their performance, efficiency, robustness, and the technical methods employed. If you leap directly from requirements into code, you're essentially designing the software on the fly. You come up with *a* design but not necessarily with *an excellent* design, and poorly structured software is the likely result. Refactoring the code can improve the design (Fowler 1999), but an ounce of up-front design is worth a pound of post-release refactoring. Thinking about design alternatives will also help to ensure that developers respect any stated design constraints.

The Incredible Shrinking Design

True Stories I once worked on a project that simulated the behavior of a photographic system with eight computational processes. After working hard on requirements analysis, the team was eager to start coding. Instead, we took the time to create a design model—now we were thinking about how we'd build a solution rather than trying to understand the problem. We quickly realized that three of the simulation's steps used identical computational algorithms, three more used another set, and the remaining two shared a third set. The design perspective simplified the problem from eight complex calculations to just three. Had we begun coding immediately after our requirements analysis, we doubtless would have noticed the code repetition at some point, but we saved a lot of time by detecting these simplifications early on. It's more efficient to revise design models than to rewrite code!

As with requirements, excellent designs are the result of iteration. Make multiple passes through the design to refine your initial concepts as you gain information and generate additional ideas. Shortcomings in design lead to products that are difficult to maintain and extend and that don't satisfy the customer's performance, usability, and reliability objectives. The time you spend translating requirements into designs is an excellent investment in building high-quality, robust products.

You needn't develop a complete, detailed design for the entire product before you begin implementation, but you should design each component before you code it. Design planning is of most benefit to particularly difficult projects, projects involving systems with many internal component interfaces and interactions, and projects staffed with inexperienced developers (McConnell 1998). All kinds of projects, however, will benefit from the following actions:

- Develop a solid architecture of subsystems and components that will hold up during enhancement.
- Identify the key object classes or functional modules you need to build, defining their interfaces, responsibilities, and collaborations with other units.
- For parallel-processing systems, understand the planned execution threads or allocations of functionality to concurrent processes.
- Define each code unit's intended functionality, following the sound design principles of strong cohesion, loose coupling, and information hiding (McConnell 1993).
- Make sure that your design accommodates all the functional requirements and doesn't contain unnecessary functionality.
- Ensure that the design accommodates exceptional conditions that can arise.
- Ensure that the design will achieve performance, robustness, reliability, and other stated quality goals.

As developers translate requirements into designs and code, they'll encounter points of ambiguity and confusion. Ideally, developers can route these issues back to customers or analysts for resolution. If an issue can't be resolved immediately, any assumptions, guesses, or interpretations that a developer makes should be documented and reviewed with customer representatives. If you encounter many such problems, the requirements weren't sufficiently clear or adequately detailed before the analyst passed them to the developers. Review the remaining requirements with a developer or two and tune them up before continuing with construction. Also, revisit your requirements validation processes to see how the low-quality requirements made it into the developers' hands.

From Requirements to Tests

Testing and requirements engineering have a synergistic relationship. As consultant Dorothy Graham points out, "Good requirements engineering produces better tests; good test analysis produces better requirements" (Graham 2002). The requirements provide the foundation for system testing. The product should be tested against what it was intended to do as recorded in the requirements documentation, not against its design or code. System testing that's based on the code can become a self-fulfilling prophecy. The product might correctly exhibit all the behaviors described in test cases based on the code, but that doesn't mean that it correctly implements the user or functional requirements. Include testers in requirements inspections to make sure the requirements are verifiable and can serve as the basis for system testing.

What to Test?

True Stories A seminar attendee once said, "I'm in our testing group. We don't have written requirements, so we have to test what we think the software is supposed to do. Sometimes we're wrong, so we have to ask the developers what the software does and test it again." Testing what the developers built isn't the same as testing what they were *supposed* to build. Requirements are the ultimate reference for system and user acceptance testing. If the system has poorly specified requirements, the testers will discover many implied requirements that developers implemented. Some of these implied requirements will reflect appropriate developer decisions, but others might constitute gold plating or misunderstandings. The analyst should incorporate the legitimate implied requirements and their origins into the SRS to make future regression testing more effective.

The project's testers should determine how they'd verify each requirement. Possible methods include

- Testing (executing the software to look for defects)
- Inspection (examining the code to ensure that it satisfies the requirements)
- Demonstration (showing that the product works as expected)
- Analysis (reasoning through how the system should work under certain circumstances)

The simple act of thinking through how you'll verify each requirement is itself a useful quality practice. Use analytical techniques such as cause-and-effect graphs to derive test cases based on the logic described in a requirement (Myers 1979). This will reveal ambiguities, missing or implied *else* conditions, and other problems. Every functional requirement should trace to at least one test case in your system test suite so that no expected system behavior goes unverified. You can measure testing progress in part by tracking the percentage of the requirements that have passed their tests. Skillful testers will augment requirements-based testing with additional testing based on the product's history, intended usage scenarios, overall quality characteristics, and quirks.

Specification-based testing applies several test design strategies: action-driven, data-driven (including boundary value analysis and equivalence class partitioning), logic-driven, event-driven, and state-driven (Poston 1996). It's possible to generate test cases automatically from formal specifications, but you'll have to develop test cases manually from the more common natural-language requirements specifications.

As development progresses, the team will elaborate the requirements from the high level of abstraction found in use cases, through the detailed software functional requirements, and ultimately down to specifications for

individual code modules. Testing authority Boris Beizer (1999) points out that testing against requirements must be performed at every level of software construction, not just the end-user level. Much of the code in an application isn't directly accessed by users but is needed for infrastructure operations. Each module must satisfy its own specification, even if that module's function is invisible to the user. Consequently, testing the system against user requirements is a necessary—but not sufficient—strategy for system testing.

PREVIOUS NEXT
PREVIOUS NEXT

From Requirements to Success

True Stories I recently encountered a project in which a contract development team came on board to implement an application for which an earlier team had developed the requirements. The new team took one look at the dozen three-inch binders of requirements, shuddered in horror, and began coding. They didn't refer to the SRS during construction. Instead, they built what they thought they were supposed to, based on an incomplete and inaccurate understanding of the intended system. Not surprisingly, this project encountered a lot of problems. The prospect of trying to understand a huge volume of even excellent requirements is certainly daunting, but ignoring them is a decisive step toward project failure. It's faster to read the requirements, however extensive, before implementation than it is to build the wrong system and then build it again. It's even faster to engage the development team early in the project so that they can participate in the requirements work and perform early prototyping.

True Stories A more successful project had a practice of listing all the requirements that were incorporated in a specific release. The project's quality assurance (QA) group evaluated each release by executing the tests for those requirements. A requirement that didn't satisfy its test criteria was counted as an error. The QA group rejected the release if more than a predetermined number of requirements weren't met or if specific high-impact requirements weren't satisfied. This project was successful largely because it used its documented requirements to decide when a release was ready to ship.

The ultimate deliverable from a software development project is a software system that meets the customers' needs and expectations. The requirements are an essential step on the path from business need to satisfied customers. If you don't base your project plans, software designs, and system tests on a foundation of high-quality requirements, you're likely to waste a great deal of effort trying to deliver a solid product. Don't become a slave to your requirements processes, though. There's no point in spending so much time generating unnecessary documents and holding ritualized meetings that no software gets written and the project is canceled. Strive for a sensible balance between rigorous specification and off-the-top-of-the-head coding that will reduce to an acceptable level the risk of building the wrong product.

Next Steps

- Try to trace all the requirements in an implemented portion of your SRS to individual design elements. These might be processes in data flow models, tables in an entity-relationship diagram, object classes or methods, or other design elements. Missing design elements might indicate that developers did their design work mentally on the short path from requirements to code. If your developers don't routinely create designs before cutting code, perhaps they need some training in software design.

- Record the number of lines of code, function points, object classes, or GUI elements that are needed to implement each product feature or use case. Also record your estimates for the effort needed to fully implement and verify each feature or use case, as well as the actual effort required. Look for correlations between size and effort that will help you make more accurate estimates from future requirements specifications.

- Estimate the average percentage of unplanned requirements growth on your last several projects. Can you build contingency buffers into your project schedules to accommodate a similar scope increase on future projects? Use the growth data from previous projects to justify the schedule contingency so that it doesn't look like arbitrary padding to managers, customers, and other stakeholders.

PREVIOUS NEXT