CHAPTER 17

Validating the requirements

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 two user classes, a developer named Jeremy, and Trish, the business analyst who wrote the SRS. One requirement stated, "The system shall provide unattended terminal timeout security of workstations accessing the training system." Jeremy presented his interpretation of this requirement to the rest of the group. "This requirement says the system will automatically log off the current user of any workstation logged into the training system if there hasn't been any activity within a certain period of time."

Hui-Lee, one of the product champions, chimed in. "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 several minutes, it logs the user off? That could be annoying if the user was just talking to someone briefly."

Trish added, "The requirement doesn't say anything about logging off the user. I assumed that timeout security meant a logoff, but maybe the user just has to retype her password to keep going."

Jeremy was confused also. "Does this mean any *workstation that* can *connect to the training system, or just workstations that are actively logged into the system at the moment? How long of 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. He followed up with Trish after the meeting to ensure that she understood all of the issues so she could resolve them.

Most software developers have experienced the frustration of being presented with 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. As you saw in Chapter 1, "The essential software requirement," it costs far more to correct a requirement error after implementation than to correct one found during requirements development. One study found that it took an average of 30 minutes to fix a defect 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 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—worse—by the end user. If you start your test planning and test-case development in parallel with requirements development, you'll detect many errors shortly after they're introduced. This prevents them from doing further damage and minimizes your development and maintenance costs.

Figure 17-1 illustrates the V model of software development. It shows test activities beginning in parallel with the corresponding development activities. This model indicates that acceptance tests are derived from the user requirements, system tests are based on the functional requirements, and integration tests are based on the system's architecture. This model is applicable whether the software development activities being tested are for the product as a whole, a particular release, or a single development increment.

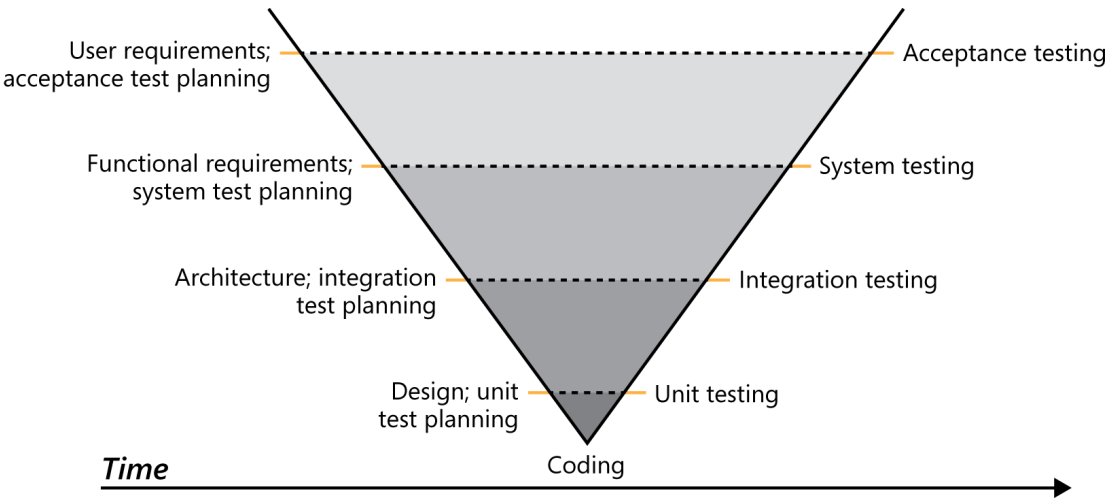

FIGURE 17-1 The V model of software development incorporates early test planning and test design.

As we will discuss later in the chapter, you can use the tests to validate each of these requirement types during requirements development. You can't actually execute any tests during requirements development because you don't have any running software yet. However, conceptual (that is, implementation-independent) tests based on the requirements will reveal errors, ambiguities, and omissions in your requirements and models before the team writes any code.

Project participants sometimes are reluctant to spend time reviewing and testing requirements. Their intuition tells them 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). 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 usually saves you much more 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 (2013) use the term *fit criteria* to describe such quantifications. This chapter addresses the validation techniques of formal and informal requirements reviews, developing tests from requirements, and having customers define their acceptance criteria for the product.

Validation and verification

Requirements validation is the fourth component of requirements development, along with elicitation, analysis, and specification. Some authors use the term "verification" for this step. In this book, we've adopted the terminology of the *Software Engineering Body of Knowledge* (Abran et al. 2004) and refer to this aspect of requirements development as "validation." Verifying requirements to ensure that they have all the desired properties of high-quality requirements is also an essential activity. Precisely speaking, validation and verification are two different activities in software development. *Verification* determines whether the product of some development activity meets its requirements (doing the thing right). *Validation* assesses whether a product satisfies customer needs (doing the right thing).

Extending these definitions to requirements, verification determines whether you have written the requirements right: your requirements have the desirable properties described in Chapter 11, "Writing excellent requirements." Validation of requirements assesses whether you have written the right requirements: they trace back to business objectives. These two concepts are closely intertwined. For simplicity in this chapter, we talk about validating the requirements, but the techniques we describe contribute both to having the correct requirements and to having high-quality requirements.

Validating requirements allows teams to build a correct solution that meets the stated business objectives. Requirements validation activities attempt to ensure that:

- The software requirements accurately describe the intended system capabilities and properties that will satisfy the various stakeholders' needs.
- The software requirements are correctly derived from the business requirements, system requirements, business rules, and other sources.
- The requirements are complete, feasible, and verifiable.
- All requirements are necessary, and the entire set is sufficient to meet the business objectives.
- All requirements representations are consistent with each other.
- The requirements provide an adequate basis to proceed with design and construction.

Validation isn't a single discrete phase that you perform after eliciting and documenting all the requirements. Some validation activities, such as incremental reviews of the growing requirements set, are threaded throughout the iterative elicitation, analysis, and specification processes. Other activities, such as formal inspections, provide a final quality gate prior to baselining a set of requirements. Include requirements validation activities as tasks in your project plan. Of course, you can validate only requirements that have been documented, not implicit requirements that exist only in someone's mind.

Reviewing requirements

Anytime someone other than the author of a work product examines the product for problems, a *peer review* is taking place. Reviewing requirements 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 2002). 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.

Informal reviews are good for catching glaring errors, inconsistencies, and gaps. They can help you spot statements that don't meet the characteristics of high-quality requirements. But it's hard for a reviewer to catch all of the ambiguous requirements on his own. He might read a requirement and think he understands it, moving on to the next without a second thought. Another reviewer might read the same requirement, arrive at a different interpretation, and also not think there is an issue. If these two reviewers never discuss the requirement, the ambiguity will go unnoticed until later in the project.

Formal peer reviews follow a well-defined process. A formal requirements review produces a report that identifies the material examined, the reviewers, and the review team's judgment as to whether the requirements are acceptable. The principal deliverable is a summary of the defects found and the issues raised during the review. The members of a formal review team share responsibility for the quality of the review, although authors ultimately are responsible for the quality of the deliverables they create.

The best-established type of formal peer review is called an *inspection*. Inspection of requirements documents is one of the highest-leverage software quality techniques available. Several companies have avoided as many as 10 hours of labor for every hour they invested in inspecting requirements documents and other software deliverables (Grady and Van Slack 1994). A 1,000 percent return on investment is not to be sneezed at.

If you're serious about maximizing the quality of your software, your teams will inspect most of their requirements. Detailed inspection of large requirements sets is tedious and time consuming. Nonetheless, the teams 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, less complex, or less novel material for which an informal review will suffice. Inspections are not cheap. They're not even that much fun. But they are cheaper—and more fun—than the alternative of expending lots of effort and customer goodwill fixing problems found much later on.

The closer you look, the more you see

On the Chemical Tracking System project, 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 of the BAs combined the input from all user classes into a single SRS of about 50 pages plus several appendices. Two BAs, one developer, three product champions, 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 (Fagan 1976; Radice 2002), and others have extended or modified his method (Gilb and Graham 1993; Wiegers 2002). Inspection has been recognized as a software industry best practice (Brown 1996). Any software work product can be inspected, including requirements, design documents, source code, test documentation, and project plans.

Inspection is a well-defined multistage process. It involves a small team of participants who carefully examine a work product for defects and improvement opportunities. Inspections serve as a quality gate through which project deliverables must pass before they are baselined. There are several forms of inspection, but any one of them is a powerful quality technique. The following description is based on the Fagan inspection technique.

Participants

Ensure that you have all of the necessary people in an inspection meeting before proceeding. Otherwise you might correct issues only to find out later that someone important disagrees with the change. The participants in an inspection should represent four perspectives (Wiegers 2002):

- **The author of the work product and perhaps peers of the author** The business analyst who wrote the requirements document provides this perspective. Include another experienced BA if you can, because he'll know what sorts of requirements-writing errors to look for.
- **People who are the sources of information that fed into the item being inspected** These participants could be actual user representatives or the author of a predecessor specification. In the absence of a higher-level specification, the inspection must include customer representatives, such as product champions, to ensure that the requirements describe their needs 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 because they 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 interfacing systems that will be affected by 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 seven or fewer inspectors. This might mean 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.

The author's manager normally should not attend an inspection meeting, unless the manager is actively contributing to the project and his presence is acceptable to the author. An effective inspection that reveals many defects might create a bad impression of the author to a hypercritical manager. Also, the manager's presence might stifle discussion from other participants.

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 (Wiegers 2002).

Author The author created or maintains the work product being inspected. The author of a requirements document is usually the business analyst who elicited customer needs and wrote the requirements. 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 should not assume any of the other assigned roles—moderator, reader, or recorder. By not having an active role, the author can listen to the comments from other inspectors, respond to—but not debate—their questions, and think. This way the author can often spot errors that other inspectors don't see.

Moderator The moderator plans the inspection with the author, coordinates the activities, and facilitates the inspection meeting. The moderator distributes the materials to be inspected, along with any relevant predecessor documents, to the participants a few 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 major defects rather than resolving problems or being distracted by minor stylistic issues and typos. The moderator follows up on proposed changes with the author to ensure that the 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 requirements and model elements being examined one at a time. The other participants then point out potential defects and issues that they see. 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. In less formal types of peer reviews, the reader role is omitted, with the moderator walking the team through the work product and soliciting comments on one section at a time.

Recorder The recorder uses standard forms to document the issues raised and the defects found during the meeting. The recorder should review aloud or visually share (by projecting or sharing in a web conference) what he wrote to confirm its 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 so he can address it efficiently and correctly.

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 and doesn't have obvious spelling, grammatical, or formatting issues.
- ❑ Line numbers or other unique identifiers are printed on the document to facilitate referring to specific locations.
- ❑ All open issues are marked as TBD (to be determined) or accessible in an issue-tracking tool.
- ❑ 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 17-2. You can inspect small sets of requirements at a time—perhaps those allocated to a specific development iteration—thereby eventually covering the full requirements collection. The purpose of each inspection process stage is summarized briefly in this section.

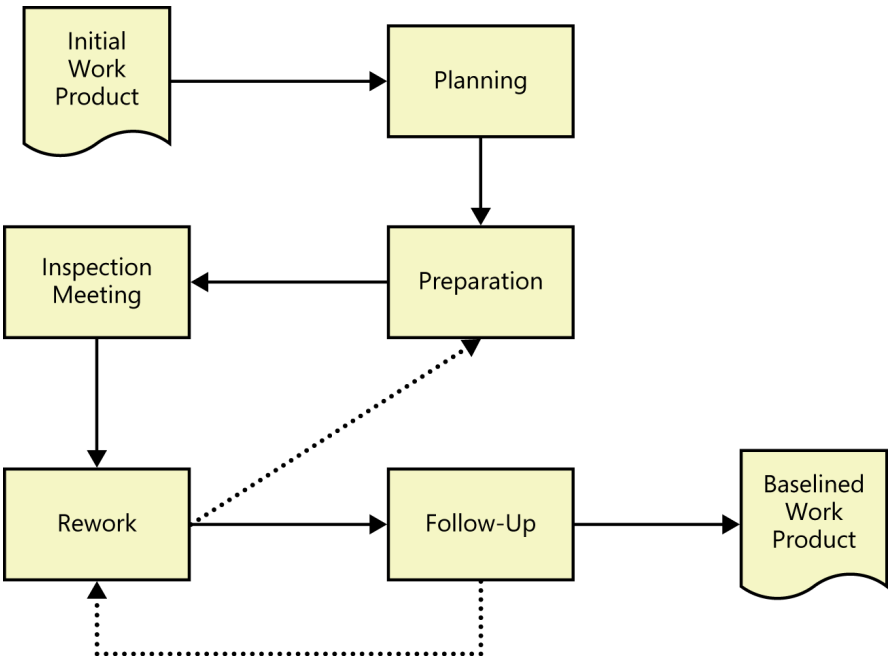

FIGURE 17-2 Inspection is a multistep process. The dotted lines indicate that portions of the inspection process might be repeated if reinspection is necessary because of extensive rework.

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, the total meeting time needed to cover the material, and when the inspection should be scheduled. The number of pages reviewed per hour has a large impact on how many defects are found (Gilb and Graham 1993). As Figure 17-3 shows, proceeding through a requirements document 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. It's not totally clear which is cause and which is effect.) Because no team has infinite time available for requirements inspections, select an appropriate inspection rate based on the risk of overlooking major defects. Two to four pages per hour is a practical guideline, although the optimum rate for maximum defect-detection effectiveness is about half that rate (Gilb and Graham 1993). Adjust this rate based on the following factors:

- The team's previous inspection data, showing inspection effectiveness as a function of rate
- The amount of text on each page
- The complexity of the requirements
- The likelihood and impact of having errors remain undetected
- How critical the material being inspected is to project success
- The experience level of the person who wrote the requirements

Preparation Prior to the inspection meeting, the author should share background information with inspectors so they understand the context of the items being inspected and know the author's objectives for the inspection. Each inspector then examines the product to identify possible defects and issues, using the checklist of typical requirements defects described later in this chapter or other

analysis techniques (Wiegers 2002). Up to 75 percent of the defects found by an inspection are discovered during preparation, so don't omit this step (Humphrey 1989). The techniques described in the "Finding missing requirements" section in Chapter 7, "Requirements elicitation," can be helpful during preparation. Plan on spending at least half as much time on individual preparation as is scheduled for the team inspection meetings.

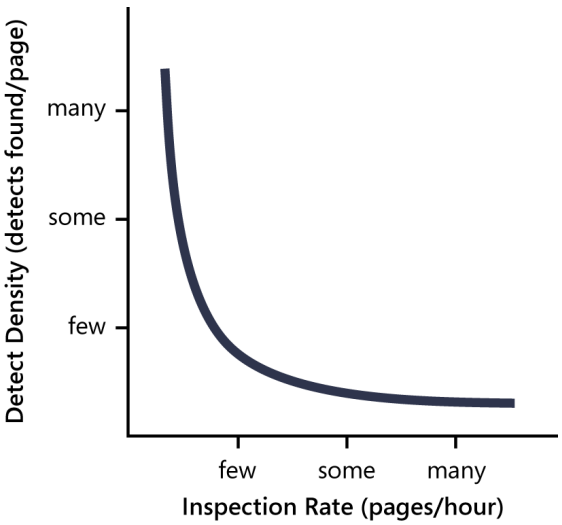

FIGURE 17-3 The number of defects found depends on the inspection rate.

> **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.

Inspection meeting During an inspection meeting, the reader leads the other inspectors through the document, describing one requirement at a time in his own words. As inspectors bring up possible defects and other issues, the recorder captures them in the action item list for the requirements author. The purpose of the meeting is to identify as many major defects as possible. The inspection meeting shouldn't last more than about two hours; tired people aren't effective inspectors. If you need more time to cover all the material, schedule additional meetings.

After examining all the material, 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" could suggest that the requirements development process has some shortcomings or that the BA who wrote the requirements needs additional training. Consider holding a retrospective to explore how the process can be improved prior to the next specification activity (Kerth 2001). If major revisions are necessary, the team might elect to re-examine portions of the product that require extensive rework, as shown by the dotted line between Rework and Preparation in Figure 17-2.

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 can be useful, but they distract attention from the core objective of finding significant defects and improvement opportunities.

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.

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. The follow-up step might reveal that some of the modifications made were incomplete or not performed correctly, leading to additional rework, as shown by the dotted line between Follow-up and Rework in Figure 17-2.

Exit criteria

Your inspection process should define the exit criteria that must be satisfied before the moderator declares the full inspection process (not just the meeting) complete. Here are some possible exit criteria for requirements inspections:

- ❑ All issues raised during the inspection have been addressed.
- ❑ Any changes made in the requirements and related work products were made correctly.
- ❑ All open issues have been resolved, or each open issue's resolution process, target date, and owner have been documented.

Defect checklist

To help reviewers look for typical kinds of errors in the products they review, develop a defect checklist for each type of requirements document your projects create. Such checklists call the reviewers' attention to historically frequent requirement problems. Checklists serve as reminders. Over time, people will internalize the items and look for the right issues in each review out of habit. Figure 17-4 illustrates a requirements review checklist, which is included with the companion content for this book. If you create particular requirements representations or models, you might expand the items in the checklist to be more thorough for those. Business requirements, such as a vision and scope document, might warrant their own checklist. Cecilie Hoffman and Rebecca Burgess (2009) provide several detailed review checklists, including one to validate software requirements against business requirements.

No one can remember all the items on a long checklist. If there are more than six or eight items on the list, a reviewer will likely have to do multiple passes through the material to look for everything on the list; most reviewers won't bother. Pare the lists to meet your organization's needs, and modify the items to reflect the problems that people encounter most often with your own requirements. Some studies have shown that giving reviewers 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 reviewers the same checklist and hoping for the best (Porter, Votta, and Basili 1995).

Completeness

- ❑ Do the requirements address all known customer or system needs?
- ❑ Is any needed information missing? If so, is it identified as TBD?
- ❑ Have algorithms intrinsic to the functional requirements been defined?
- ❑ Are all external hardware, software, and communication interfaces defined?
- ❑ Is the expected behavior documented for all anticipated error conditions?
- ❑ Do the requirements provide an adequate basis for design and test?
- ❑ Is the implementation priority of each requirement included?
- ❑ Is each requirement in scope for the project, release, or iteration?

Correctness

- ❑ Do any requirements conflict with or duplicate other requirements?
- ❑ Is each requirement written in clear, concise, unambiguous, grammatically correct language?
- ❑ Is each requirement verifiable by testing, demonstration, review, or analysis?
- ❑ Are any specified error messages clear and meaningful?
- ❑ Are all requirements actually requirements, not solutions or constraints?
- ❑ Are the requirements technically feasible and implementable within known constraints?

Quality Attributes

- ❑ Are all usability, performance, security, and safety objectives properly specified?
- ❑ Are other quality attributes documented and quantified, with the acceptable trade-offs specified?
- ❑ Are the time-critical functions identified and timing criteria specified for them?
- ❑ Have internationalization and localization issues been adequately addressed?
- ❑ Are all of the quality requirements measurable?

Organization and Traceability

- ❑ Are the requirements organized in a logical and accessible way?
- ❑ Are all cross-references to other requirements and documents correct?
- ❑ Are all requirements written at a consistent and appropriate level of detail?
- ❑ Is each requirement uniquely and correctly labeled?
- ❑ Is each functional requirement traced back to its origin (e.g., system requirement, business rule)?

Other Issues

- ❑ Are any use cases or process flows missing?
- ❑ Are any alternative flows, exceptions, or other information missing from use cases?
- ❑ Are all of the business rules identified?
- ❑ Are there any missing visual models that would provide clarity or completeness?
- ❑ Are all necessary report specifications present and complete?

FIGURE 17-4 A defect checklist for reviewing requirements documents.

Requirements review tips

Chapter 8 of Karl Wiegers' *More About Software Requirements: Thorny Issues and Practical Advice* (Microsoft Press, 2006) offers suggestions to improve your requirements reviews. The following tips apply whether you are performing informal or formal reviews on your projects, and whether you're storing your requirements in traditional documents, in a requirements management tool, or in any other tangible form.

Plan the examination When someone asks you to review a document, the temptation is to begin at the top of page one and read it straight through. But you don't need to do that. The consumers of the requirements specification won't be reading it front-to-back like a book; reviewers don't have to, either. Invite certain reviewers to focus on specific sections of documents.

Start early Begin reviewing sets of requirements when they are perhaps only 10 percent complete, not when you think they're "done." 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.

Allocate sufficient time Give reviewers sufficient time to perform the reviews, both in terms of actual hours to review (effort) and calendar time. They have other important tasks that the review has to fit around.

Provide context Give reviewers context for the document and perhaps for the project if they are not all working on the same project. Seek out reviewers who can provide a useful perspective based on their knowledge. For example, you might know a co-worker from another project who has a good eye for finding major requirement gaps even without being intimately familiar with the project.

Set review scope Tell reviewers what material to examine, where to focus their attention, and what issues to look for. Suggest that they use a defect checklist like the one described in the preceding section. You might want to maximize availability and skills by asking different reviewers to review different sections or to use different parts of the checklists.

Limit re-reviews Don't ask anyone to review the same material more than three times. He will be tired of looking at it and won't spot major issues after a third cycle because of "reviewer fatigue." If you do need someone to review it multiple times, highlight the changes so he can focus on those.

Prioritize review areas Prioritize for review those portions of the requirements that are of high risk or have functionality that will be used frequently. Also, look for areas of the requirements that have few issues logged already. It might be the case that those sections have not yet been reviewed, not that they are problem-free.

Requirements review challenges

A peer review is both a technical activity 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 organizations face regarding requirements reviews, some of which apply specifically to formal inspections, with suggestions for how to address each one (Wiegers 1998a; Wiegers 2002).

Large requirements documents The prospect of thoroughly examining a several-hundred-page requirements document is daunting. You might be tempted to skip the review entirely and just proceed with construction—not a wise choice. Even given a document of moderate size, all reviewers 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 review team, perform incremental reviews throughout requirements development. Identify high-risk areas that need a careful look through inspection, and use informal reviews for less risky material. Ask particular reviewers to start at different locations in the document to make certain that fresh eyes have looked at every page. To judge whether you really need to inspect the entire specification, examine a representative sample (Gilb and Graham 1993). The number and types of errors you find will help you determine whether investing in a full inspection is likely to pay off.

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 review teams increase the cost of the review, make it hard to schedule meetings, and have difficulty reaching agreement on issues. I once participated in a meeting with 13 other inspectors. Fourteen people cannot 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 position.
- Understand which perspective (such as customer, developer, or tester) each inspector represents. 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 requirements in parallel and combine their defect lists, removing any duplicates. Research has shown that multiple inspection teams find more requirements defects 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.

Geographically separated reviewers Organizations often build products through the collaboration of geographically dispersed teams. This makes reviews more challenging. Teleconferencing doesn't reveal the body language and expressions of other reviewers like a face-to-face meeting does, but videoconferencing can be an effective solution. Web conferencing tools allow reviewers to ensure that they are all looking at the same material during the discussion.

Reviews of an electronic document placed in a shared network repository provide an alternative to a traditional review meeting. In this approach, reviewers use word-processor features to insert their comments into the text. (This is how Karl and Joy reviewed each other's work as we were writing this book.) Each comment is labeled with the reviewer's initials, and each reviewer can see what previous reviewers had to say. Web-based collaboration tools also can help. Some requirements management tools include components to facilitate distributed asynchronous reviews that do not involve live meetings. If you choose not to hold a meeting, recognize that this can reduce a review's effectiveness, but it's certainly better than not performing the review at all.

Unprepared reviewers One of the prerequisites to a formal review meeting is that the participants have examined the material being reviewed ahead of time, individually identifying their initial sets of issues. Without this preparation, you risk people spending the meeting time doing all of their thinking on the spot and likely missing many important issues.

One project had a 50-page SRS to be reviewed by 15 people, far too many to be effective and efficient. Everyone had one week to review the document on their own and send issues back to the author. Not surprisingly, most people didn't look at it at all. So the lead BA scheduled a mandatory meeting for the reviewers to review the document together. He projected the SRS on the screen, dimmed the lights, and began reading through the requirements one by one. (The room had one very bright light shining down in the middle, directly on the lead BA—talk about being in the

spotlight!) A couple of hours into the review meeting, the participants were yawning, their attention fading. Not surprisingly, the rate of issue detection decreased. Everyone longed for the meeting to end. This BA let the participants leave, suggesting that they review the document on their own time to speed up the next review meeting. Sure enough, being bored during the meeting triggered them to do their prep work. See the "Workshops" section in Chapter 7 for suggestions about how to keep participants engaged during review meetings.

Prototyping requirements

It's hard to visualize how a system will function under specific circumstances just by reading the requirements. Prototypes are validation tools that make the requirements real. They allow the user to experience some aspects of what a system based on the requirements would be like. Chapter 15, "Risk reduction through prototyping," has more information on different types of prototypes and how they improve requirements. Here we describe how prototypes can help stakeholders judge whether a product built according to the requirements would meet their needs, and whether the requirements are complete, feasible, and clearly communicated.

All kinds of prototypes allow you to find missing requirements before more expensive activities like development and testing take place. Something as simple as a paper mock-up can be used to walk through use cases, processes, or functions to detect any omitted or erroneous requirements. Prototypes also help confirm that stakeholders have a shared understanding of the requirements. Someone might implement a prototype based on his understanding of the requirements, only to learn that a requirement wasn't clear when prototype evaluators don't agree with his interpretation.

Proof-of-concept prototypes can demonstrate that the requirements are feasible. Evolutionary prototypes allow the users to see how the requirements would work when they are implemented, to validate that the result is what they expect. Additional levels of sophistication in prototypes, such as simulations, allow more precise validation of the requirements; however, building more sophisticated prototypes will also take more time.

Testing the requirements

Tests 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 tests will reveal many problems with the requirements long before you can execute those tests on running software. Writing functional tests 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 BAs, developers, and customers walk through tests together, they'll achieve a shared vision of how the product will work and increase their confidence that the requirements are correct. Testing is a powerful tool for both validating and verifying requirements.

Trap Watch out for testers who claim they can't begin their work until the requirements are done, as well as for testers who claim they don't need requirements to test the software. Testing and requirements have a synergistic relationship; they represent complementary views of the system.

Making Charlie happy

I once asked my group's UNIX scripting guru, Charlie, to build a simple email interface extension for a commercial defect-tracking system we had adopted. I wrote a dozen functional requirements that described how the email interface should work. Charlie was thrilled. He'd written many scripts for people, but he'd never seen written requirements before.

Unfortunately, I waited two weeks before writing the tests for this email function. Sure enough, one of the requirements had an error. I found the mistake because my mental image of how I expected the function to work, represented in about 20 tests, conflicted with one requirement. Chagrined, I corrected the bad requirement before Charlie had completed his implementation, and when he delivered the script, it was defect-free. Finding the error before implementation was a small victory, but small victories add up.

You can begin deriving conceptual tests from user requirements early in the development process (Collard 1999; Armour and Miller 2001). Use the tests to evaluate functional requirements, analysis models, and prototypes. The tests should cover the normal flow of each use case, alternative flows, and the exceptions you identified during elicitation and analysis. Similarly, if you identified business process flows, the tests should cover the business process steps and all possible decision paths.

These conceptual tests are independent of implementation. For example, consider a use case called "View a Stored Order" for the Chemical Tracking System. Some conceptual tests are:

- User enters order number to view, order exists, user had 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 hadn't placed the order. Expected result: Display message "Sorry, that's not your order."

Ideally, a BA will write the functional requirements and a tester will write the tests from a common starting point: the user requirements, as shown in Figure 17-5. Ambiguities in the user requirements and differences of interpretation will lead to inconsistencies between the views represented by the functional requirements, models, and tests. As developers translate requirements into user interface and technical designs, testers can elaborate the conceptual tests into detailed test procedures (Hsia, Kung, and Sell 1997).

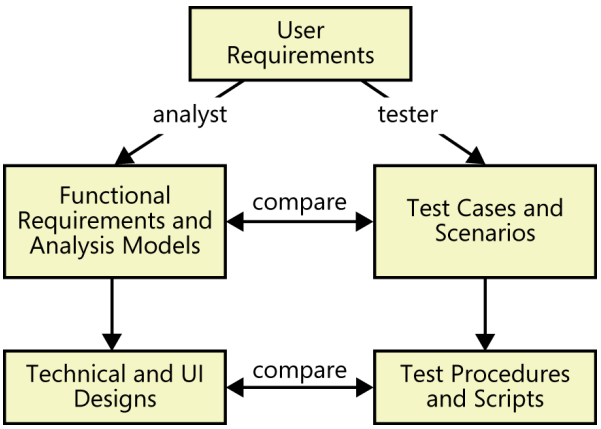

FIGURE 17-5 Development and testing work products are derived from a common source.

Let's see how the Chemical Tracking System team tied together requirements and visual models with early test thinking. Following are several pieces of requirements-related information, all of which pertain to the task of requesting a chemical.

Business requirement As described in Chapter 5, "Establishing the business requirements," one of the primary business objectives for the Chemical Tracking System was to:

> *Reduce chemical purchasing expenses by 25% in the first year.*

Use case A use case that aligns with 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 from Figure 8-3 in Chapter 8, "Understanding user requirements":

> *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 either offers the Requester a container of the chemical from the chemical stockroom or lets the Requester order one from a vendor.*

Functional requirement Here's a bit of functionality derived from 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 17-6 illustrates a portion of the dialog map for the "Request a Chemical" use case that pertains to this function. As was described in Chapter 12, "A picture is worth 1024 words," the boxes in this dialog map represent user interface displays, and the arrows represent possible navigation paths from one display to another. This dialog map was created far enough along in requirements development that the project participants were beginning to identify specific screens, menus, dialog boxes, and other dialog elements so they could give them names and contemplate a possible user interface architecture.

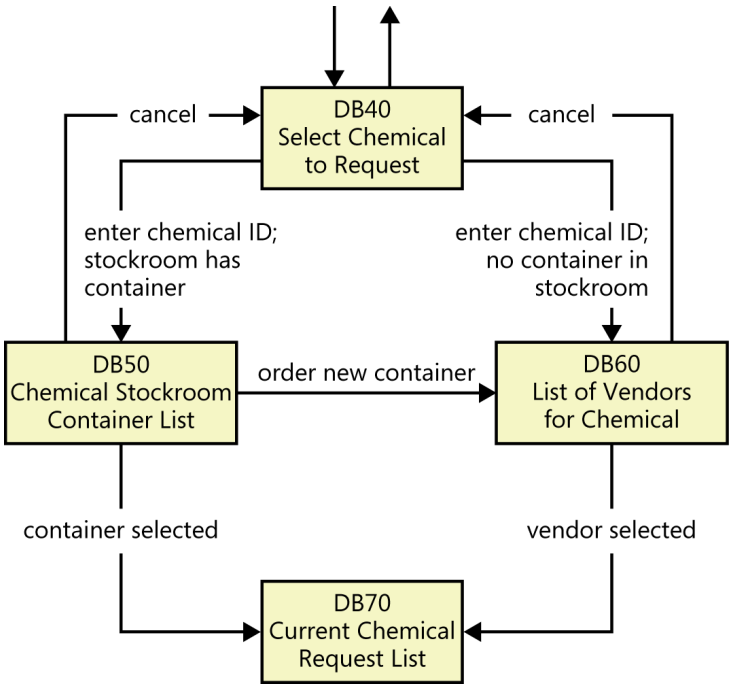

FIGURE 17-6 Portion of the dialog map for the "Request a Chemical" use case.

Test Because this use case has several possible execution paths, you can envision multiple tests to address the normal flow, alternative flows, and exceptions. The following is just one test, based on the flow that shows the user the available containers in the chemical stockroom.

> *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 tests like this one based on his understanding of the use case. Such abstract tests are independent of implementation details. They don't discuss entering data into specific fields, clicking buttons, or other specific interaction techniques. As development progresses, the tester can refine such conceptual tests into specific test procedures.

Now comes the fun part—testing the requirements. Ramesh first mapped each test to the functional requirements. He checked to make certain that every test could be "executed" by going through a set of existing requirements. He also made sure that at least one test covered each functional requirement. Next, Ramesh traced the execution path for every test on the dialog map with a highlighter pen. The shaded line in Figure 17-7 shows how the preceding test traces onto the dialog map.

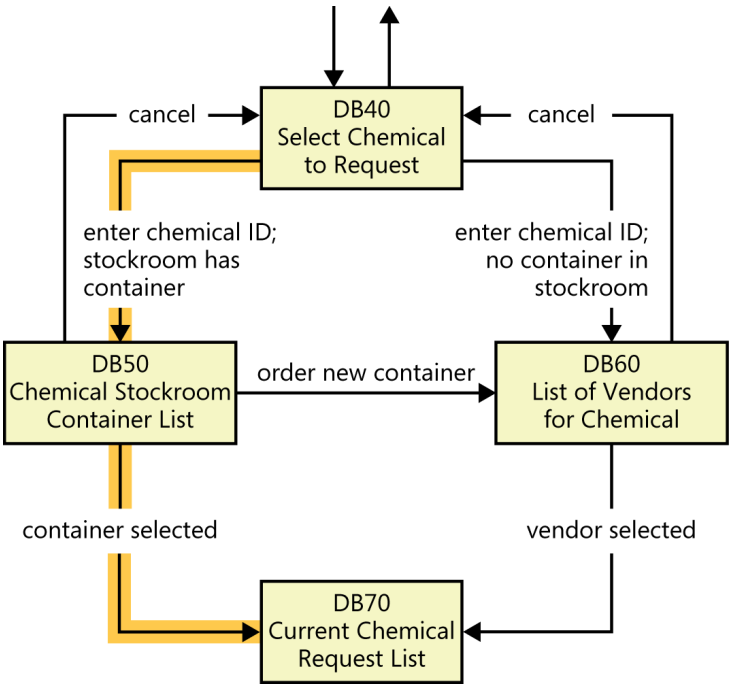

FIGURE 17-7 Tracing a test onto the dialog map for the "Request a Chemical" use case.

By tracing the execution path for each test, you can find incorrect or missing requirements, improve the user's navigation options, and refine the tests. Suppose that after "executing" all the tests in this fashion, the dialog map navigation line labeled "order new container" that goes from DB50 to DB60 in Figure 17-6 hasn't been highlighted. There are two possible interpretations:

- That navigation is not a permitted system behavior. The BA 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 legitimate, but the test that demonstrates the behavior is missing.

In another scenario, suppose a tester wrote a test based on his interpretation of the use case that says the user can take some action to move directly from dialog box DB40 to DB70. However, the dialog map in Figure 17-6 doesn't contain such a navigation line, so that test can't be "executed" with the existing requirements set. Again, there are two possible interpretations. 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 is wrong.
- The navigation from DB40 to DB70 is legitimate, but the dialog map and perhaps the SRS are missing the requirement that is exercised by the test.

In these examples, the BA and the tester combined requirements, analysis models, and tests 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. As Ross Collard (1999) pointed out,

Use cases and tests work well together in two ways: If the use cases for a system are complete, accurate, and clear, the process of deriving the tests is straightforward. And if the use cases are not in good shape, the attempt to derive tests will help to debug the use cases.

Validating requirements with acceptance criteria

Software developers might believe that they've built the perfect product, but the customer is the final arbiter. Customers need to assess whether a system satisfies its predefined *acceptance criteria*. 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; Leffingwell 2011; Pugh 2011). Having users devise acceptance tests is a valuable contributor to effective requirements development. The earlier that acceptance tests are written, the sooner they can help the team filter out defects in the requirements and, ultimately, in the implemented software.

Acceptance criteria

Working with customers to develop acceptance criteria provides a way to validate both the requirements and the solution itself. If a customer can't express how she would evaluate the system's satisfaction of a particular requirement, that requirement is not clear enough. Acceptance criteria define the minimum conditions for an application to be considered business-ready.

Thinking about acceptance criteria offers a shift in perspective from the elicitation question of "What do you need to do with the system?" to "How would you judge whether the solution meets your needs?" Encourage users to use the SMART mnemonic—*Specific*, *Measurable*, *Attainable*, *Relevant*, and *Time-sensitive*—when defining acceptance criteria. The criteria should be specified such that multiple objective observers would reach the same conclusion about whether they were satisfied. Acceptance criteria keep the focus on stakeholders' business objectives and the conditions that would allow the project sponsor to declare victory. This is more important than simply delivering on a requirements specification that might not really solve the stakeholders' business problems.

Defining acceptance criteria is more than just saying that all the requirements are implemented or all the tests passed. Acceptance tests constitute just a subset of acceptance criteria. Acceptance criteria could also encompass dimensions such as the following:

- Specific high-priority functionality that must be present and operating properly before the product could be accepted and used. (Other planned functionality could perhaps be delivered later, or capabilities that aren't working quite right could be fixed without delaying an initial release.)
- Essential nonfunctional criteria or quality metrics that must be satisfied. (Certain quality attributes must be at least minimally satisfied, although usability improvements, cosmetics, and performance tuning could be deferred. The product might have to meet quality metrics such as a certain minimum duration of operational usage without experiencing a failure.)

- Remaining open issues and defects. (You might stipulate that no defects exceeding a particular severity level remain open against high-priority requirements, although minor bugs could still be present.)
- Specific legal, regulatory, or contractual conditions. (These must be fully satisfied before the product is considered acceptable.)
- Supporting transition, infrastructure , or other project (not product) requirements. (Perhaps training materials must be available and data conversions completed before the solution can be released.)

It can also be valuable to think of "rejection criteria," conditions or assessment outcomes that would lead a stakeholder to deem the system not yet ready for delivery. Watch out for conflicting acceptance criteria, such that meeting one could block the satisfaction of another. In fact, looking for conflicting acceptance criteria early on is a way to discover conflicting requirements.

Agile projects create acceptance criteria based on user stories. As Dean Leffingwell (2011) put it,

> *Acceptance criteria are not functional or unit tests; rather, they are the conditions of satisfaction being placed on the system. Functional and unit tests go much deeper in testing all functional flows, exception flows, boundary conditions, and related functionality associated with the story.*

In principle, if all of the acceptance criteria for a user story are met, the product owner will accept the user story as being completed. Therefore, customers should be very specific in writing acceptance criteria that are important to them.

Acceptance tests

Acceptance tests constitute the largest portion of the acceptance criteria. Creators of acceptance tests should consider the most commonly performed and most important usage scenarios when deciding how to evaluate the software's acceptability. Focus on testing the normal flows of the use cases and their corresponding exceptions, devoting less attention to the less frequently used alternative flows. Ken Pugh (2011) offers a wealth of guidance for writing requirements-based acceptance tests.

Agile development approaches often create acceptance tests in lieu of writing precise functional requirements. Each test describes how a user story should function in the executable software. Because they are largely replacing detailed requirements, the acceptance tests on an agile project should cover all success and failure scenarios (Leffingwell 2011). The value in writing acceptance tests is that it guides users to think about how the system will behave following implementation. The problem with writing *only* acceptance tests is that the requirements exist only in people's minds. By not documenting and comparing alternate views of requirements—user requirements, functional requirements, analysis models, and tests—you can miss an opportunity to identify errors, inconsistencies, and gaps.

Automate the execution of acceptance tests whenever possible. This makes it easier to repeat the tests when changes are made and functionality is added in future iterations or releases. Acceptance

tests must also address nonfunctional requirements. They should ensure that performance goals are achieved, that the system complies with usability standards, and that security expectations are fulfilled.

Some acceptance testing might be performed manually by users. The tests used in user acceptance testing (UAT) should be executed after a set of functionality is believed to be release-ready. This allows users to get their hands on the working software before it is officially delivered and permits users to familiarize themselves with the new software. The customer or product champion should select tests for UAT that represent the highest risk areas of the system. The acceptance tests will validate that the solution does what it is supposed to. Be sure to set up these tests using plausible test data. Suppose the test data used to generate a sales report isn't realistic for the application. A user who is performing UAT might incorrectly report a defect just because the report doesn't look right to him, or he might miss an erroneous calculation because the data is implausible.

Trap Don't expect user acceptance testing to replace comprehensive requirements-based system testing, which covers all the normal and exception paths and a wide variety of data combinations, boundary values, and other places where defects might lurk.

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 peer reviews, 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 17-4.
- If you found enough errors during the random sample review to make the team nervous about the overall quality of the requirements, persuade the user and development representatives to inspect the entire SRS. Train the team in the inspection process.
- Define conceptual tests for a use case or for a portion of the functionality that hasn't yet been coded. See whether the user representatives agree that the tests reflect the intended system behavior. Make sure you've defined all the functionality that will permit the tests to be passed and that there are no superfluous requirements.
- Work with your product champions to define the acceptance criteria that they and their colleagues will use to assess whether the system is acceptable to them. Have them define acceptance tests that could be used to judge completeness.