CHAPTER

1

Defining What's on Your Plate: The Foundation of a Test Project

Testing requires a tight focus. It's easy to try to do too much. You *could* run an infinite number of tests against any nontrivial piece of software or hardware. Even if you try to focus on what you think might be ''good enough'' quality, you can find that such testing is too expensive or that you have trouble figuring out what ''good enough'' means for your customers and users. Before I start to develop the test system — the testware, the test environment, and the test process — and before I hire the test team, I figure out what I *might* test, then what I *should* test, and finally what I *can* test. Determining the answers to these questions helps me plan and focus my test efforts.

What I might test are all those untested areas that fall within the purview of my test organization. On every project in which I've been involved, some amount of the test effort fell to organizations outside my area of responsibility. Testing an area that another group already covered adds little value, wastes time and money, and can create political problems for you.

What I should test are those untested areas that directly affect the customers' and users' experience of quality. People often use buggy software and computers and remain satisfied nevertheless. Either they never encounter the bugs or the bugs don't significantly hinder their work. Our test efforts should focus on finding the critical defects that will limit people's ability to get work done with our products.

What I can test are those untested, critical areas on which my limited resources are best spent. Can I test everything I should? Not likely, given the schedule and budget I usually have available.[1] On most projects, I must make

[1]You can find the first mention of this difficult test management problem in Glenford Myers's *The Art of Software Testing*.

tough choices, using limited information, on a tight schedule. I also need to sell the test project to my managers to get the resources and the time I need.

What You *Might* Test: The Extended Test Effort

On my favorite software and system projects, testing was pervasive. By this, I mean that a lot of testing went on outside the independent test team. In addition, testing started early. This arrangement not only made sense technically, but also kept my team's workload manageable. This section uses two lenses to examine how groups outside the formal test organization contribute to testing. The first lens is the level of focus — the *granularity* — of a test. The second is the type of testing performed within various test phases. Perhaps other organizations within your company could be (or are) helping you test.

From Microscope to Telescope: Test Granularity

Test *granularity* refers to the fineness or coarseness of a test's focus. A fine-grained test case allows the tester to check low-level details, often internal to the system. A coarse-grained test case provides the tester with information about general system behavior. You can think of test granularity as running along a spectrum ranging from structural (white-box) to behavioral (black-box and live) tests, as shown in Figure 1-1.

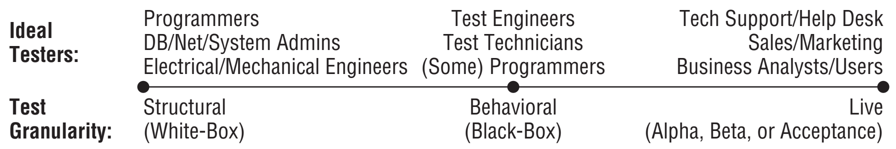


Figure 1-1 The test granularity spectrum and owners

Structural (White-Box) Tests

Structural tests (also known as *white-box tests* and *glass-box tests*) find bugs in low-level structural elements such as lines of code, database schemas, chips, subassemblies, and interfaces. The tester bases structural tests on *how* a system operates. For example, a structural test might reveal that the database that stores user preferences has space to store an 80-character username, but that the field allows the user to enter only 40 characters.

Structural testing involves a detailed technical knowledge of the system. For software, testers create structural tests by looking at the code and the data structures themselves. For hardware, testers create structural tests to compare chip specifications to readings on oscilloscopes or voltage meters. Structural tests thus fit well in the development area. Testers in an independent test team — who often have little exposure to low-level details and might lack programming or engineering skills — find it difficult to perform structural testing.

Structural tests also involve knowledge of structural testing techniques. Not all programmers learn these techniques as part of their initial education and ongoing skills growth. In such cases, having a member of the test team work with the programmers as a subject-matter expert can promote good structural testing. This person can help train the programmers in the techniques needed to find bugs at a structural level.

Behavioral (Black-Box) Tests

Testers use behavioral tests (also known as *black-box* tests) to find bugs in high-level operations, such as major features, operational profiles, and customer scenarios. Testers can create black-box functional tests based on *what* a system should do. For example, if SpeedyWriter should include a feature that saves files in XML format, then you should test whether it does so. Testers can also create black-box non-functional tests based on *how* a system should do what it does. For example, if DataRocket can achieve an effective throughput of only 10 Mbps across two 1-gigabit Ethernet connections acting as a bridge, a black-box network-performance test can find this bug.

Behavioral testing involves a detailed understanding of the application domain, the business problem that the system solves, and the mission the system serves. When testers understand the design of the system, at least at a high level, they can augment their behavioral tests to effectively find bugs common to that type of design. For example, programs implemented in languages like C and C++ can — depending on the programmers' diligence — suffer from serious security bugs related to buffer overflows.

In addition to the application domain and some of the technological issues surrounding the system under test, behavioral testers must understand the special behavioral test techniques that are most effective at finding such bugs. While some behavioral tests look at typical user scenarios, many tests exercise extremes, interfaces, boundaries, and error conditions. Bugs thrive in such boundaries, and behavioral testing involves searching for defects, just as structural testing does. Good behavioral testers use scripts, requirements, documentation, and testing skills to guide them to these bugs. Simply playing around with the system or demonstrating that the system works under average conditions are not effective techniques for behavioral testing, although many test teams make the mistake of adopting these as the sole test techniques. Good behavioral tests, like good structural tests, are structured, methodical, and often repeatable sequences of tester-created conditions that probe suspected system weaknesses and strive to find bugs, but through the external interfaces of the system under test. Most independent test organizations perform primarily behavioral testing.

Live Tests

Live tests involve putting customers, content experts, early adopters, and other end users in front of the system. In some cases, we encourage the testers to try to break the system. Beta testing is a well-known form of bug-driven live testing. For example, if the SpeedyWriter product has certain configuration-specific bugs, live testing might be the best way to catch those bugs specific to unusual or obscure configurations. In other cases, the testers try to demonstrate conformance to requirements, as in acceptance testing, another common form of live testing.

Live tests can follow general scripts or checklists, but live tests are often ad hoc (worst case) or exploratory (best case). They don't focus on system weaknesses except for the ''error guessing'' that comes from experience. Live testing is a perfect fit for technical support, marketing, and sales organizations whose members don't know formal test techniques but do know the application domain and the product intimately. This understanding, along with recollections of the nasty bugs that have bitten them before, allows them to find bugs that developers and testers miss.

The Complementary and Continuous Nature of Test Granularity

The crew of a fishing boat uses a tight-mesh net to catch 18-inch salmon and a loose-mesh net to catch six-foot tuna. They might be able to catch a tuna in a salmon net or vice versa, but it would probably make them less efficient. Likewise, structural, behavioral, and live tests each are most effective at finding certain types of bugs. Many great test efforts include a mix of all three types.

While my test teams focus on behavioral testing typically, I don't feel bound to declare my test group ''the black-box bunch.'' I've frequently used structural test tools and cases effectively as part of my system test efforts. I've also used live production data in system testing. Both required advanced planning, but paid off handsomely in terms of efficiency (saved time and effort) and effectiveness (bugs found that we might have missed). Test granularity is a spectrum, not an either/or categorization. Mixing these elements can be useful in creating test conditions or assessing results. I also mix planned test scenarios with exploratory live testing. I use whatever works.

A Stampede or a March? Test Phases

The period of test execution activity during development or maintenance is sometimes an undifferentiated blob. Testing begins, testers run some (vaguely defined) tests and identify some bugs, and then, at some point, project management declares testing complete. As development and maintenance processes mature, however, companies tend to adopt an approach of partitioning testing into a sequence of phases (sometimes called *levels*). Ownership of those various phases can differ; it's not always the test team. There are various commonly encountered test phases, although these often go by different names.

Unit Testing

Unit testing focuses on an individual piece of code. What constitutes an individual piece of code is somewhat ambiguous in practice. I usually explain to our clients that unit testing should focus on the smallest construct that one could meaningfully test in isolation. With procedural programming languages such as C, unit testing should involve a single function. For object-oriented languages such as Java, unit testing should involve a single class.

Unit testing is not usually a test phase in a project-wide sense of the term, but rather the last step of writing a piece of code. The programmer can use structural and behavioral test design techniques, depending on her preferences and skills, and, possibly, an organizational standard.

Regardless of which test design technique is used, unit tests are white-box in the sense that the programmer knows the internal structure of the unit under test and is concerned with how the testing affects the internal operations. Therefore, programmers usually do the unit testing. Sometimes they test their own code. Sometimes they test other programmers' code, often referred to as *buddy tests* or *code swaps*. Sometimes two programmers collaborate on both the writing and unit testing of code, such as the *pair programming* technique advocated by practitioners of the agile development approach called Extreme Programming.

Component or Subsystem Testing

During the component or subsystem testing, testers focus on the constituent pieces of the system. Component testing applies to some collection of units that provide some defined set of capabilities within the system.

Component test execution usually starts when the first component of the product becomes functional, along with whatever scaffolding, stubs, or drivers[2] are needed to operate this component without the rest of the system. In our SpeedyWriter product, for example, file manipulation is a component. For DataRocket, the component test phase would focus on elements such as the SCSI subsystem: the controller, the hard-disk drives, the CD/DVD drive, and the tape backup unit.

Component testing should use both structural and behavioral techniques. In addition, components often require hand-built, individualized test harnesses. Because of the structural test aspects and the custom harnesses required,

[2]For a discussion on stubs, drivers, and other such frameworks for component testing, you can refer to my book *Pragmatic Software Testing*.

component testing often requires programmers and hardware engineers. However, when components are standalone and have well-defined functionality, behavioral testing conducted by independent test teams can work. For example, I once worked on a Unix operating-system development project in which the test organization used shell scripts to drive each Unix command through its paces using the command-line interface — a typical black-box technique. We later reused these component test scripts in system testing. In this instance, component testing was a better fit for the test organization.

Integration or Product Testing

Integration or product testing focuses on the relationships and interfaces between pairs of components and groups of components in the system under test, often in a staged fashion. Integration testing must happen in coordination with the project-level activity of *integrating* the entire system — putting all the constituent components together, a few components at a time. The staging of integration and integration testing must follow the same plan — sometimes called the *build plan* — so that the right set of components comes together in the right way and at the right time for the earliest possible discovery of the most dangerous integration bugs. For SpeedyWriter, integration testing might start when the developers integrate the file-manipulation component with the graphical user interface (GUI) and continue as developers integrate more components one, two, or three at a time, until the product is feature-complete. For DataRocket, integration testing might begin when the engineers integrate the motherboard with the power supply, continuing until all components are in the case.[3]

Not every project needs a formal integration test phase. If your product is a set of standalone utilities that don't share data or invoke one another, you can probably skip this. However, if the product uses application programming interfaces (APIs) or a hardware bus to coordinate activities, share data, and pass control, you have a tightly integrated set of components that can work fine alone yet fail badly together.

The ownership of integration testing depends on a number of factors. One is skill. Usually, testers will use structural techniques to perform integration testing; some independent test teams do not have sufficient internal system expertise. Another is resources. Project plans sometimes neglect or

[3]Since this is a book on test management, not test design, I don't want to diverge into a long technical discussion of how to do this. Instead, I can recommend three books for your test engineers to read. If they can find a copy, Boris Beizer's *Software System Testing and Quality Assurance* remains one of the best. If they can't find that, my own book *Pragmatic Software Testing* contains a chapter on integration testing that summarizes Beizer's discussion as well as some other useful ideas. Finally, Rick Craig and Stefan Jaskiel's book, *Systematic Software Testing*, does a good job of explaining this topic as well.

undersize this important task, and neither the development manager nor the test manager will have the resources (human or machine) required for integration testing. Finally, unit and component testing tends to happen at the individual-programmer level when owned by the development team — each programmer tests her own component or swaps testing tasks with her programmer peer — but this model won't work for integration testing. In these circumstances, unfortunately, I have seen the development manager assign this critical responsibility to the most junior member of the programming team. In such cases, it would be far better for the test team to add the necessary resources — including appropriately skilled people — to handle the integration testing. When the product I'm testing needs integration testing, I plan to spend some time with my development counterparts working out who should do it.

String Testing

String testing focuses on problems in typical usage scripts and customer operational strings. This phase is a rare bird. I have seen it used only once, when it involved a strictly black-box variation on integration testing. In the case of SpeedyWriter, string testing might involve cases such as encrypting and decrypting a document, or creating, printing, and saving a document.

System Testing

System testing encompasses the entire system, fully integrated. Sometimes, as in installation and usability testing, these tests look at the system from a customer or end-user point of view. Other times, these tests stress particular aspects of the system that users might not notice, but are critical to proper system behavior. For SpeedyWriter, system testing would address such concerns as installation, performance, and printer compatibility. For DataRocket, system testing would cover issues such as performance and network compatibility.

System testing tends to be behavioral. When doing system testing, my test teams apply structural techniques to force certain stressful conditions that they can't create through the user interface — especially load and error conditions — but they usually evaluate the pass/fail criteria at an external interface. Where independent test organizations exist, they often run the system tests.

Acceptance or User-Acceptance Testing

From unit testing through to system testing, finding bugs is a typical test objective. Before you start acceptance testing, though, you generally want to have found all the bugs. The test objective is to demonstrate that the system meets requirements. This phase of testing is common in contractual situations, when successful completion of acceptance tests obligates a buyer to accept a system. For in-house IT development efforts, successful completion of the acceptance tests triggers deployment of the software in a production environment.

In commercial software and hardware development, acceptance tests are sometimes called *alpha tests* (executed by in-house users) and *beta tests* (executed by current and potential customers). Alpha and beta tests, when performed, might be about demonstrating a product's readiness for market, although many organizations also use these tests to find bugs that can't be (or weren't) detected in the system testing process.

Acceptance testing can involve live data, environments, and user scenarios. The focus is usually on typical product-usage scenarios, not extreme conditions. Therefore, marketing, sales, technical support, beta customers, and even company executives are perfect candidates to run acceptance tests. (Two of my clients — one a small software startup and the other a large PC manufacturer — use their CEOs in acceptance testing; the product ships only if the CEO likes it.) Test organizations often support the acceptance testing; provide test tools, suites, and data that they developed during system testing; and, with user ''witnessing,'' sometimes execute the acceptance tests.

Pilot Testing

Hardware development often involves pilot testing, either following or in parallel with acceptance tests. Pilot testing checks the ability of the assembly line to mass-produce the finished system. I have also seen this phase included in in-house and custom software development, where it demonstrates that the system will perform all the necessary operations in a live environment with a limited set of real customers. Unless your test organization is involved in production or operations, you probably won't be responsible for pilot testing.

Why Do I Prefer a Phased Test Approach?

As you've seen, a phased test approach marches methodically across the test focus granularity spectrum, from structural tests to behavioral tests to live tests. Such an approach can provide the following benefits:

- Structural testing can build product stability. Some bugs are simple for developers to fix but difficult for the test organization to live with. You can't do performance testing if SpeedyWriter corrupts the hard disk and crashes the system after 10 minutes of use.

- Structural testing using scaffolding or stubs can start early. For example, you might receive an engineering version of DataRocket that is merely a motherboard, a SCSI subsystem, and a power supply on a foam pad. By plugging in a cheap video card, an old monitor, and a DVD drive, you can start testing basic I/O operations.
- You can detect bugs earlier and more efficiently, as mentioned previously.
- You can precisely and quantitatively manage the bug levels in your system as you move through the project.
- Phases provide real and psychological milestones against which the project team can gauge the quality of the system and thus the project's proximity to completion.

I'll explain the last two benefits in more detail in Chapters 4 and 9.

Test Phase Sequencing

Figure 1-2 shows a common sequence of the execution activities for various test phases. On your projects, the execution activities in these phases might be of different relative lengths. The degree of overlap between execution activities in different phases varies considerably depending on entry and exit criteria for each phase, which I'll discuss in Chapter 2, and on the project life cycle, which I'll discuss in Chapter 12. Quite a few organizations omit the test phases that I've shown with dotted lines in the figure. There's no need to divide your test effort exactly into the six test phases diagrammed in Figure 1-2. Start with the approach that best fits your needs and let your process mature organically.

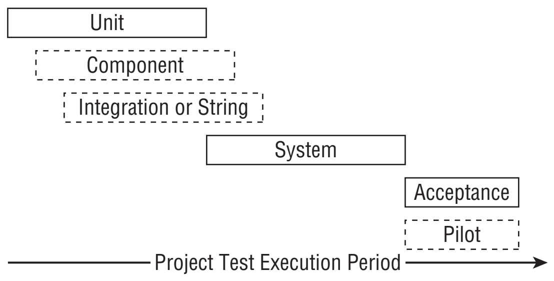


Figure 1-2 The test execution period for various test phases in a development project

When I plan test sequencing, I try to start each test phase as early as possible. Software industry studies have shown that the cost of fixing a bug found just one test phase earlier can be lower by an order of magnitude or more, and my experience leads me to believe that the same argument applies to hardware development.[4] In addition, finding more bugs earlier in testing increases the total number of bugs you'll find. On unique, leading-edge projects, I need to test basic design assumptions. The more realistic I make this testing, the more risk mitigation I achieve.

This rule of starting test phases as early as possible has some caveats. Since the nasty, hard-to-fix problems often first rear their ugly heads in behavioral testing, moving into integration or system testing early can buy the project more time to fix them. However, you need to make sure that the earlier phases of testing found and fixed enough bugs to adequately stabilize the product and make it ready for such testing. Otherwise, you'll enter a later phase of testing before the product is ready, and spend a lot of time working inefficiently, with many blocked tests and hard-to-isolate bugs.

This is complicated by another common project failing. One of the main challenges with unit testing and other phases of testing typically owned by the programmers relates to whether these tests actually get done. Rushed for time, and knowing an independent test team will get the code somewhere down the line, programmers sometimes are tempted to skip these tests. Even if such tests do get done, as I mentioned before, not all programmers know how to do them properly. So, it makes sense to have some amount of engagement between your test team with the development team to help ensure that these tests get done and get done properly.

The First Cut

At this point, you have some ideas about how other organizations attack the division of the test roles. Now you can look at the testing that already goes on in your organization and locate gaps. If you are establishing a new test organization, you might find that folks who tested certain areas on previous projects believe that they needn't continue testing now that *you're* here. (I touch on this topic more in Chapter 9 when I discuss how development groups can become ''addicted'' to the test team.) After identifying past test contributions, I make sure to close the loop and get commitments from individual contributors (and their managers) that they will continue to test in the future.

What You *Should* Test: Considering Quality

Once I've identified the areas of testing that might be appropriate for my test organization, my next step is to figure out what I should test. To do this, I must understand what quality means for the system, and the risks to system quality

[4]For example, see Stephen Kan's *Metrics and Models in Software Quality*, Jack Campanella's *Principles of Quality Costs*, and Capers Jones's *Estimating Software Costs*.

that exist. While quality is sometimes seen as a complex and contentious topic, I have found a pragmatic approach.

Three Blind Men and an Elephant: Can You Define Quality?

There's a management parable about three blind men who came across an elephant. One touched the tail and declared it a snake. Another touched a leg and insisted that it was a tree. The third touched the elephant's side and claimed that it was a wall.

Defining quality can be a similar process. Everyone knows what *they* mean by quality, but disagreements abound. Have you debated with developers over whether a particular test case failure was really a bug? If so, weren't these debates in fact about whether the observed behavior was a quality issue? What, really, is quality? What factors determine its presence or absence? Whose opinions matter most?

J. M. Juran, a respected figure in the field of quality management, defines quality as ''features [that] are decisive as to product performance and as to 'product satisfaction'. . . . The word 'quality' also refers to freedom from deficiencies . . . [that] result in complaints, claims, returns, rework and other damage. Those collectively are forms of 'product *dis*satisfaction.'''[5] Testing should cover the decisive features, those that determine customer and user satisfaction, and try to find as many as possible of the bugs that would result in dissatisfaction.

As the project team develops or maintains a system, the project team is exposed to various risks related to not implementing all of the satisfying features and to implementing some of them improperly. These risks can collectively be called *quality risks*, since these risks relate to the possibility of a negative or undesirable outcome related to the quality of the system. As we execute our tests, we might discover failures related to these risks. At the most general level, the process of testing should allow the test organization to assess the quality risks and to understand the failures that exist in the system under test.

After a system is released, customers or users who encounter bugs might experience dissatisfaction and then make complaints, return merchandise, or call technical support. This makes the users and customers the arbiters of quality. Who are these people, and what do they intend to do with the system? For our purposes, let's assume that customers are people who have paid or will pay money to use your system and that they expect your system to do what a

[5]This is from Juran's book *Juran on Planning for Quality*. In his book *Quality Is Free*, Phillip Crosby argues that quality is conformance to requirements — nothing more and nothing less. But when was the last time you worked on a project with complete, unambiguous requirements?

similar system, in the same class and of the same type, should reasonably do. The users might also be customers, or they might be people who did not pay for the product or its development, but use it or its output to get work done.

Testing looks for situations in which a product fails to meet customers' or users' reasonable expectations in specific areas. For example, IBM evaluates customer satisfaction in terms of capability (functions), usability, performance, reliability, installability, maintainability, documentation/information, service, and overall fitness for use. Hewlett-Packard uses the categories of functionality, usability, reliability, performance, and serviceability.

The Perils of Divergent Experiences of Quality

As people use a product — a car, an espresso machine, a bar of soap — they form opinions about how well that product fulfills their expectations. These impressions, good or bad, become their *experience of quality* for that product. Test teams try to assess quality during test execution. In other words, you and your test team use the test system — the testware, the test environment, and the test process as discussed in Chapter 3 — to gauge, in advance, customers' experiences of quality. I refer to the extent to which the test system allows testers to do this as the *fidelity* of the test system.

Figures 1-3 and 1-4 provide visual representations of two test systems. In both figures, the circles represent the sets of quality risks for a product. In Figure 1-3, test system A allows the tester to cover a majority of the product's quality risks and also to cover those areas that affect user A's experience of quality.

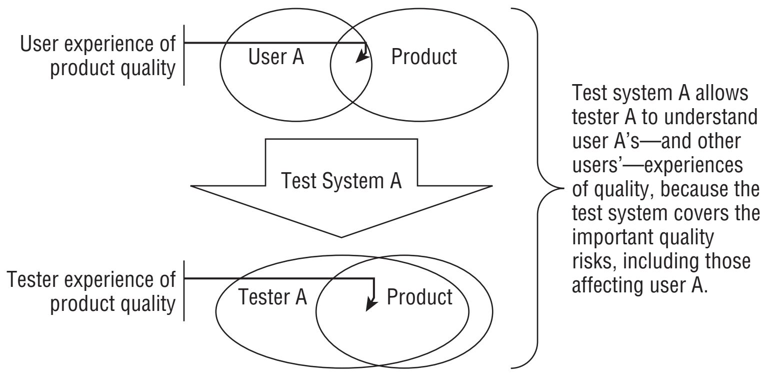

User experience of product quality
User A
Product
Test System A
Tester experience of product quality
Tester A
Product
Test system A allows tester A to understand user A's—and other users'—experiences of quality, because the test system covers the important quality risks, including those affecting user A.

Figure 1-3 A high-fidelity test system

Test system B, shown in Figure 1-4, fails in both respects. It covers a smaller portion of the product's quality risks. Worse yet, the portion tested does not cover user B's experience of quality.

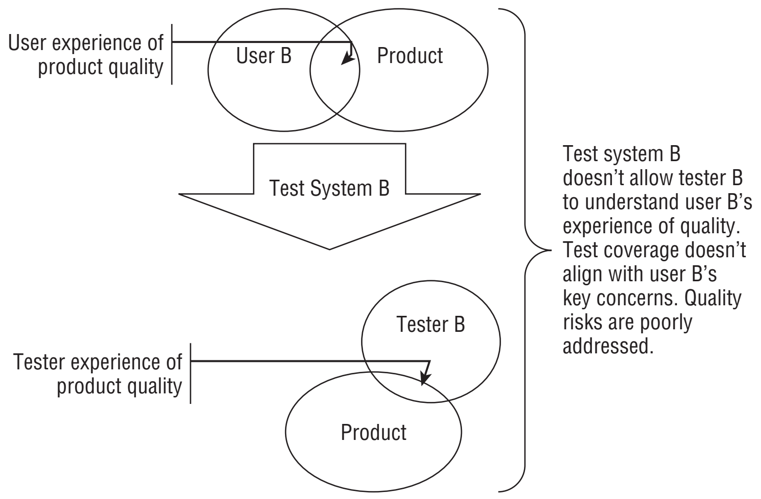

User experience of product quality
User B
Product
Test System B
Test system B doesn't allow tester B to understand user B's experience of quality. Test coverage doesn't align with user B's key concerns. Quality risks are poorly addressed.
Tester B
Tester experience of product quality
Product

Figure 1-4 A low-fidelity test system

Two other scenarios are possible. First, suppose that you have a test system with the same degree of coverage as test system B, but that the coverage area aligns with user B's use of the product. In this case, your test team will do a fine job of catching critical defects — at least from user B's perspective. You'll also be able to explain how those defects will affect the users, which is important in terms of establishing priority. If most users, including your most important ones, use the product the same way user B does, then test system B, coverage limitations notwithstanding, is a good test system.

Second, suppose that you have a test system with the same degree of coverage as test system A, but that the coverage area does not align with user A's usage of the product. In this case, you fail to test the quality risks that user A cares about. In addition, you can't relate the results of the testing you do perform to real-world usage scenarios, which reduces the apparent priority of any defects you find. Since these features will probably reach the field buggy, user A will be dissatisfied. If user A is typical of your customer base — especially your important customers — you have a serious test coverage problem, even though the test system covers most of the product's quality risks.

Figure 1-5 represents these scenarios. Of course, you can't test all of the quality risks and none of the customer uses, or vice versa. In Figure 1-5, these unlikely or unreachable zones are shown in the dotted-line-enclosed areas in the upper left and lower right. The best test systems score on the right-hand side of the chart (covering customer usage), and the farther up the right-hand side you get (the more quality risks you cover), the better your test system. Those quality risks that relate most closely to actual customer usage are the critical quality risks.

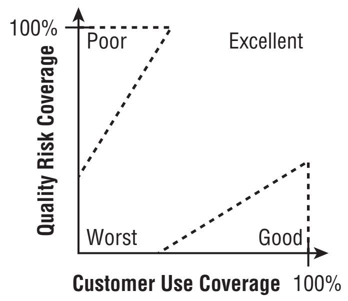


Figure 1-5 Test system coverage and its ability to assess quality

So, how do you determine the customer-critical quality risks? You want to address as many of these quality risks as possible, developing and executing tests consistent with customer priorities in both order and emphasis. Various quality risk analysis techniques will allow you to do so, so let's look at them now.

What to Worry About: How to Analyze Quality Risks

The next few pages discuss risk-based testing. Let me start by clarifying some terms and concepts. First, *risk* generally means the possibility of a negative or undesirable outcome or event. Specific to our area of concern, testing, we can say that a risk is any possible problem that would decrease customer, user, participant, or stakeholder perceptions of product quality or project success.

When the primary effect of a potential problem is to impact project success, we can call it a *project risk*. For example, a possible test environment problem that could delay completion of a project is a project risk. When the primary effect of a potential problem is on the quality of the product itself, we can call it a *quality risk*. For example, a possible performance defect that could cause a system to respond slowly during normal operation is a quality risk. I'll cover how testing should deal with project risks in the next chapter. In this chapter, I focus on quality risks.

Risks differ in terms of importance, which I refer to as the *level of risk*. Later in this section, we'll look at some intricate ways to determine the level of risk, but we'll start with a simple approach that considers two factors:

- The likelihood of the problem occurring
- The impact of the problem should it occur

To determine likelihood, you generally focus on technical considerations, such as the programming language, network throughput, and so forth. To determine impact, you generally focus on business considerations, such as the potential financial impact of a problem, the frequency with which users or customers will encounter a problem, and so forth. While I refer to this as a simple approach to determine the level of risk, it is often sufficient for many organizations.

Analytical risk-based testing strategies start with quality risk analysis to identify risk items and determine the level of risk for each risk item. You then address the quality risks in four ways:

- **Allocation of effort.** During planning, preparation, and execution of testing, you allocate effort for each quality risk item based on the level of risk. Individual testers should match the rigor and extensiveness of the test techniques to the level of risk.
- **Test sequencing.** During planning, preparation, and execution of testing, test managers and testers attack the risks in risk priority order, starting with the most important quality risks first and working their way down to the less important ones.
- **Test triage.** If needed during test execution, should management reduce the time or resources available for testing, you can delete tests from the test plan in reverse-risk priority order, starting with the least important tests.
- **Reporting test results.** Test managers should report test results in terms of risk. You'll look at this more closely in Chapters 4 and 5.

With these preliminaries out of the way, let's get into the details. First I'll review some of the properties and benefits of analytical risk-based testing strategies. Next I'll show you a couple of checklists you can use as frameworks and mental aids for quality risk identification. I'll then discuss processes, techniques, and templates that you can use for quality risk analysis. Finally, I'll cover some tips and challenges associated with quality risk analysis.

Properties and Benefits of Analytical Risk-Based Testing

Analytical risk-based testing has six interesting and useful properties, two of which are fundamental, and four of which are incidental.

First and fundamentally, the testing effort is proportional to the level of risk. The higher the level of risk for any risk item, the more test effort you expend to develop and to execute test cases for that risk item.

Second and also fundamentally, test tasks are sequenced based on risk. The higher the level of risk for any risk item, the earlier you develop the test cases for that risk item. The test case inherits the level of risk belonging to the risk item from which you derive the test case. By using the level risk of risk associated with the test case to sequence the test cases, you can run the test cases in risk order.

The third property, an incidental one, has to do with the way test execution reduces the residual level of risk. This incidental property arises from the way the first two fundamental properties of risk-based testing influence the overall level of risk during the project. Because the test cases relate to risks items, and because you run them in risk order, the overall level of residual quality risk goes down as test execution continues, and the drop in the overall level of residual quality risk is most significant in the first quarter of the test execution period, when you're running the highest-risk test cases.

The fourth property, also incidental, enables risk-based test results reporting. Because the test cases relate to risk items, if you preserve traceability between test cases, bugs found by those test cases, and the risk items from which you derived those test cases, you can do risk-based test results reporting. This means that you report your test results not only in terms of bugs (found and fixed) and test cases (run, passed, and failed), but also in terms of the overall level of residual quality risk and in terms of specific risk items that have known failed test cases or known bugs. (You'll look at traceability in Chapter 3, and results reporting in Chapters 4 and 5.)

The fifth property, incidental as well, allows intelligent test triage. Since each test case inherits the level of quality risk from its parent risk item, if you find yourself forced to reduce the overall test execution effort due to schedule pressure, you can eliminate the cases in reverse risk order. You will run the most important tests (and will run them first) and will drop less important tests (which you would run later in any case) only if you find yourself squeezed at the end.

The sixth property, and the final incidental one, allows for self-correction of errors in the risk analysis. This property relates to a weakness inherent in all analytical test strategies. In any analytical test strategy, you perform an analysis early in a project and use that analysis to determine the test work you will do. However, any early analysis will often be based on incorrect assumptions and information, to some extent, and those invalid concepts become embedded in the testing. By blending reactive test strategies such as bug hunting, software attacks, and exploratory testing (discussed in Chapter 3), you introduce a self-correcting element into the test execution process, since these reactive strategies will tend to identify the holes and mistakes in your test set that arose due to problems with the analysis. In iterative life cycle models (which I'll discuss in Chapter 12), including agile models, you will do the analysis iteratively, which helps to address this problem as well.

Because of some of these properties, analytical risk-based testing strategies provide a number of benefits to you as a test manager and to the project team.

First, due to the ability to allocate effort, prioritize test cases, and, if necessary, to triage test cases, risk-based testing allows you to deal with the common situation of insufficient time, including the situation where you must make intelligent test case deletion decisions when management reduces the test execution period.

Second, due to the same priorities, risk-based testing helps you make smart coverage decisions. Remember that, at the outset of this chapter, I said that an infinite number of tests *could be* run against any system. So, test coverage, measured as a percentage of what *could be* tested, is always 0% because you must select a finite number of actual test cases from this infinite cloud of possible test cases. So, how do you choose a smart subset? Risk-based testing gives you a defensible method for doing this.

A third benefit arises from the process for risk analysis rather than from the properties of risk-based testing. Best practices for quality risk analysis involve a broad cross-section of business and technical stakeholders in the risk-analysis process. Due to this broad stakeholder involvement, even if you receive poor specifications documents, you can fill in the gaps in those documents based on what the stakeholders tell you.

A fourth benefit is one offered primarily to the project team, though you are the bearer of the benefit. Because you can report test results in terms of residual risk, rather than only bug and test counts, this allows you to give the project team a solid understanding of the risks associated with releasing the system at any point in time after test execution begins.

Kick-Starting Quality Risk Analysis with Checklists

In the next section, I describe processes you can use for quality risk analysis. These processes involve starting with a checklist and developing the list of quality risk items using that checklist as a framework. So, in this section, let's look at two checklists you can use.

The Usual Suspects

To develop the list of major quality risk categories, I start by breaking down the test process into the phases of component testing, integration testing, and system testing. Using the guidelines presented earlier in this chapter, you will have already determined which of these test phases you will run and which you can skip because other colleagues are covering them.

During unit and component testing, the following major quality risk categories apply:

- **States.** In some computer systems, especially telephony systems and embedded software of various types, the components or some set of components implement what is called a *state machine*. Incoming events cause a state machine to transition through clearly defined states, while the response (the associated output and the subsequent state) to an event depends on the current state, the event, and any conditions that might exist. State machines present a variety of quality risks related both to the state machine as a whole and to the individual states. Do the transitions from one state to another occur under the proper conditions? Does the system generate the correct outputs? Does the system accept and properly handle both legal and illegal event/condition combinations? Consider an alarm card in the DataRocket server that sends SNMP information over the network if problems arise. This component spends most of its time in a quiescent state, but if it senses that the CPU is overheating, it transitions to a CPU Overtemp Warn state, during which it sends out alerts at regular intervals. If the problem does not clear up, the component transitions to a CPU Overtemp Critical state, at which point it initiates a system shutdown. You will need to verify that the transitions occur at the right points and that the component can't get stuck in a given state. For example, if the CPU returns to a normal temperature but the alarm card remains in a CPU Overtemp Warn state, the alarm card will continue to send (now spurious) alerts over the network and might do something dangerous, such as transitioning incorrectly to the CPU Overtemp Critical state.

- **Transactions.** Components that have transactions with the user or with other components present various risks. For example, creating a new file is a transaction in SpeedyWriter. Can the user select the appropriate file template? How does the product respond to illegal file names?
- **Code coverage.** Untested code in a component presents unknown structural risks. These untested areas often handle unusual or hard-to-create conditions, which make it tempting to skip them. For example, simulating the CPU Overtemp Critical condition described earlier might result in damage to the test configuration. However, how else can you verify that the system shutdown will actually occur? If it is impractical — because of CPU placement, for example — to simulate the overheating using a hair dryer or a soldering iron, you might be forced to sacrifice one CPU to find out.
- **Data-flow coverage.** A data flow is the transfer of information — either through parameters, shared (global) data space, or a stored database — from one component of the system to another. The risks associated with data flows don't receive nearly the attention they deserve Programs allow you to import, export, and link data from other programs, creating complex data flows. Users sometimes report strange and counterintuitive failures while using these features. If SpeedyWriter, for example, includes a component that reads and writes Microsoft Word files, testers must evaluate this feature across multiple Word versions and with files that include more than just text. In the hardware world, signal quality testing is a form of component-level data-flow testing.

- **Functionality.** Each component exists to implement some set of functions, which are internal operations such as calculations and formatting. Functional quality risks are generally of two types: either the function behaves improperly, or the function behaves properly but has undesirable side effects.
- **User interface.** The quality risks in this area are similar to those encountered for functionality, but they also include questions of usability such as understandable prompts and messages, clear control flows, and appropriate color schemes and graphics. User interface testing during component testing often involves prototypes of the interface.[6]
- **Mechanical life.** Any object that can be flexed or moved has a limit to the number of motions it can endure: keys on a keyboard break, hinges fatigue, buttons snap off, latches crack, and contacts fail.
- **Signal quality.** Any circuit that processes data, whether digital or analog, is subject to the constraints imposed by signal quality. Lead times, lag times, rise times, fall times, noise, spikes, transients, and the like can be out of spec, causing a component to fail.

During integration testing, the following major quality risk categories apply:

- **Component or subsystem interfaces.** Every API, every method, every function, every bus, every connector represents an opportunity for misunderstandings between the two (or more) component development engineers. These misunderstandings manifest themselves when two otherwise-correct components fail to work together. Shared data files and especially dynamic data such as configuration files and multi-user databases are interfaces as well. Any place where one component transfers data or control to one or more components, whether immediately or in a delayed fashion, an interface exists that can cause trouble.
- **Functionality.** In integration tests, you again have risks related to the possibility of the wrong action, or the right action with the wrong side effect. Here you focus on functionality that requires the correct operation of two or more components or a flow of data between them.
- **Capacity and volume.** Think of software, a computer, or a network of computers as a system of pipes for bringing in information, operating on it, storing it, and sending it out. The capacities (static) and volumes (dynamic) of these pipes must match the requirements of the application and the expectations of the user. From a structural test perspective, every buffer, queue, storage resource, processor, bus, and I/O channel

[6] As Steve McConnell points out in the *Software Project Survival Guide*, these mock-ups are an excellent opportunity to get real users in front of the interface, and should actually be created during the requirements, design, or detailed design phase.

in the system has a theoretical limit and a (lower) practical limit. For a single-user program on a PC, this might be a simple, well-bounded set of risks. For SpeedyWriter, the effects of network traffic and the speed of the typist might be the only issues. For a network server such as DataRocket, however, a variety of risks can apply. Can the network card handle realistic traffic levels? Can the disk subsystem deal with realistic loads? Is the data-storage capability sufficient? In integration testing, you can begin to evaluate these risks.

- **Error/disaster handling and recovery.** Undesirable events happen. PCs lock up. Servers crash. Networks drop packets. Hard drives die. Building air conditioners and heaters go out. Electrical grids have power surges, brownouts, and failures. It might be depressing, but you should construct a list of such situations and how they can affect your system. Increasingly, people choose to use common PC-based office applications and operating systems in critical infrastructure. This implies a need for true disaster-recovery capability. Mostly, though, there are the mundane mini-catastrophes that will eventually afflict the system. You can start looking at these quality risks early in the integration test phase.
- **Data quality.** If your product stores, retrieves, and shares significant amounts of data — especially data that has delicate links, relationships, and integrity constraints — you should consider testing whether the product can handle that data reliably. For example, I once used an expense-reporting program that had a serious data quality bug in the way it handled the expense-report data file that it managed. Because I needed to analyze data across multiple reports, all reports had to reside in the same file. If the PC operating system crashed while the application had the expense-report file open, the application corrupted the file. The corruption was subtle; I could continue to use the file for quite a while afterward, but in the meantime, the corruption compounded itself. At some point, any attempt to add a new transaction caused the application to crash. The application did not include a file-repair utility. Because data storage and retrieval tend to be clustered in certain components or subsystems, you should start testing these areas as soon as these components are integrated.
- **Performance.** As with capacity and volume, performance concerns apply to most subsystems or components in a product. For real-time and mission-critical applications, performance can be the most important quality risk. Even for systems that are not real-time, important performance issues exist. Most product reviews address performance. Performance is not only ''how many per second,'' but also ''how long.'' Consider the battery life of a laptop. As the system is integrated, you can begin to measure performance.

- **User interface.** As more pieces of real functionality are integrated into the system, you can start to test these through the user interface. (If the user interface is a true throw-away prototype, then you might not have this option.)

During system and acceptance tests, the following major quality risk categories apply:

- **Functionality.** During system testing, you should consider functionality in terms of whole sequences of end-user operations (broad) or an entire area of functionality (deep). For example, with SpeedyWriter you might look at creating, editing, and printing a file, or at all the possible ways of creating a file, all the editing options, and all the printing options.
- **User interface.** If you or the programmers have a chance to work with a prototype in earlier test phases, the remaining usability quality risks during system testing are the irritating behaviors that crop up when everything is connected to the interface. Regrettably, though, the system test phase is often the first point of testing for the complete user interface with all the commands and actions available. (The prototyping I advocated earlier happens on the best projects, but not on all projects.) If you have the opportunity, you must address all usability quality risks at this stage.
- **States.** State machines can exist at the system level as well as at the component level. For example, a voice-mail system is a complex computer-telephony state machine.
- **Transactions.** Transaction handling can also occur at the system level. DataRocket, for example, handles transactions: printing a file, delivering a file (one chunk at a time), and so forth.
- **Data quality.** During the system test phase, I revisit the data quality risks initially covered in integration testing, since the complexity of the data often increases once the entire product is integrated. For example, if SpeedyWriter supports embedded pictures and other nontext objects, this feature might not be dropped in until the end of the integration test. Working with such complex data makes problems more likely.
- **Operations.** Complex systems often require administrators; databases, networks, and servers come to mind. These operators perform essential maintenance tasks that sometimes take the system offline. For DataRocket, consider the following quality risks: Can you back up and restore files? Can you migrate the system from a Windows server to a Linux server? Can you add an external RAID array? Can you add memory? Can you add a second LAN card?

- **Capacity and volume.** During a system test, you have the same kinds of quality risks related to capacity and volume risks covered in integration testing.
- **Reliability, availability, and stability.** Quality risks in this area include unacceptable failure rates (mean time between failures, or MTBF), unacceptable recovery times (mean time to repair, or MTTR), and the inability of the system to function under legitimate conditions without failure.
- **Error/disaster handling and recovery.** As in the case of capacity and volume, I revisit error/disaster handling and recovery from a behavioral perspective. I focus on the external failures.
- **Stress.** This risk category is often an amalgam of capacity, volume, reliability, stability, and error/disaster handling and recovery.
- **Performance.** First broached during integration testing, performance is another risk category that I revisit during the system test phase.
- **Date and time handling.** A decade ago, concerns about the year 2000 raised the level of awareness about these types of quality risks, but many programmers of today were not yet even in college at that point. You might also need to take account of the fact that some countries — for example, Taiwan — base their calendars on events other than the birth of Jesus. Additionally, your product might not work properly in different time zones, or even multiple time zones if it is a distributed system.
- **Localization.** Localization typically refers to problems associated with different languages. Even Romance languages, which use the Latin alphabet, often include special letters, such as the *ñ* in Spanish, that can generate quality risks if your product includes sorting or searching capabilities. Languages such as Chinese, Japanese, Russian, and Greek create bigger difficulties. Besides the software considerations, computers in these environments use different keyboards and different printer drivers. Moreover, language is not the only thing that changes at the border and can affect your system. Can your product handle 220 volts and 110 volts, 50 hertz and 60 hertz? How about the unique dial tones and ring signals found in Europe and Asia? Beyond the technical considerations, there are cultural issues and taboos. What is considered an acceptable way of indicating something in one culture might be a rude or obscene gesture in another.
- **Networked and distributed environments.** If your product works in a networked or distributed environment, you have some special quality risks to consider. For example, what if your system spans time zones? Can the constituent systems talk to each other without getting confused about Central Standard Time and Pacific Standard Time? If your systems must communicate internationally, will the telephone standards affect them?

- **Configuration options and compatibility.** Most PC software these days supports various configuration options. SpeedyWriter, for example, might need to remember a customer's name, address, and company to generate letter outlines. DataRocket might allow various CPU speeds and multiprocessor settings. In addition, many configuration options are dynamic. On-demand loading and unloading of drivers, libraries, and software; cold and hot swapping of devices; and power management can dynamically change the configuration of software and hardware. Moreover, when you look out past the internal variables, the PC world includes a bewildering variety of software, hardware, and network environments that can create problems for your system. Will the system talk to all the printers your customers own? Do network drivers cause your system to fail? Can your software coexist with leading applications?
- **Standards compliance.** In the hardware world, you might need to consider legal and market standards such as UL, FCC, CE, and others that might be required for your target market. In the software and hardware worlds, customers sometimes require compatibility logos such as Microsoft's ''Designed for Windows.'' Innocuous bugs related to standards can have serious repercussions: your company might even find the product legally or effectively barred from the market.
- **Security.** Given your dog's name, your spouse's name, your children's names, and your birthday, I might be able to crack your computer accounts. On a larger scale, if your company has a web site, right now criminals might be trying to break into your network. If security is a feature of or concern for your product, you will need to think about the quality risks that exist.
- **Environment.** Because hardware products must live in the real world, they are subject to environmental risks. How do the shaking and bumping encountered during shipping affect a server? Can power sags and surges cause your system to crash and fail? What about the effects of temperature and humidity?
- **Power input, consumption, and output.** All computers take in electrical current, convert some of it to heat and some to electromagnetic radiation, and send the rest of it to attached devices. Systems with rechargeable batteries, such as laptops, might add some conversion and storage steps to this process, and some systems might use power in unusual modes such as 48 VDC, but ultimately the process is the same. This orchestration of electrical power can fail; insufficient battery life for laptops is a good example.
- **Shock, vibration, and drop.** All computers will at some point be moved. I have never worked with a system that was assembled on the floor

on which it would operate. In the course of this movement, the computer will experience shocks, vibrations, and, occasionally, drops. Some computers are subject to motion while on, others only while packaged. The system test phase is the right time to find out whether the system misbehaves after typical encounters with the laws of Newtonian physics.

- **Installation, cut-over, setup, and initial configuration.** Every instance of a product has an initial use. Does the installation process work? Can you migrate data from an old system? Are there unusual load profiles during the first few weeks of use? These loads can include many user errors as people learn the system. In a multi-user situation, configuration will also include the creation of the initial accounts. Think about the entire process, end to end. Individual actions might work, but the process as a whole could be unworkable. In addition, consider the possibility that someone might want to uninstall the product. Finally, don't forget quality risks related to the licensing and registration processes.
- **Documentation and packaging.** If your product includes documentation, you have risks ranging from the possibly dangerous to the simply embarrassing. Consider instructions in DataRocket's manual, accompanied by an illustration, that led a user to set the input voltage selector for 110 volts in a 220-volt environment. On the less serious side, think of some of the humorous quotations from technical documentation that circulate on the Internet. Do you want your company singled out for such honors? Packaging, likewise, should be appropriately marked.
- **Maintainability.** Even if your system is too simple to require an operator, you might still have maintainability risks. Can you upgrade software to a current version? Can you add memory to your PC? If your software works in a networked environment, does it support remote (possibly automated) software distribution?
- **Alpha, beta, and other live tests.** For general-purpose software and hardware, no amount of artificial testing can cover all the uses and environments to which your customers will subject your product. To address these risks, I like to use a beta or early-release program of some sort.

As long as this list is, it is not complete. I could add other categories of quality risks. However, this list should serve as a good starting point for your quality risk analysis process.

Using the ISO 9126 Standard as a Checklist

If you work in an organization that tends to follow standards, you might want to consider the ISO 9126 standard's quality model as a checklist and framework for your quality risk analysis. The ISO 9126 quality model consists of six quality characteristics for systems, each of which has three or more subcharacteristics:

- **Functionality:** Attributes that bear on the existence of a set of functions and their specified properties. The functions are those that satisfy stated or implied needs. Its subcharacteristics are suitability, accuracy, interoperability, security, and compliance.
- **Reliability:** Attributes that bear on the capability of software to maintain its level of performance under stated conditions for a stated period of time. Its subcharacteristics are maturity, recoverability, fault tolerance, and compliance.
- **Usability:** Attributes that bear on the effort needed for use, and on the individual assessment of such use, by a stated or implied set of users. Its subcharacteristics are learnability, understandability, operability, and compliance.
- **Efficiency:** Attributes that bear on the relationship between the level of performance of the software and the amount of resources used, under stated conditions. Its subcharacteristics are time behavior, resource behavior, and compliance.
- **Maintainability:** Attributes that bear on the effort needed to make specified modifications. Its subcharacteristics are stability, analyzability, changeability, testability, and compliance.
- **Portability:** Attributes that bear on the ability of software to be transferred from one environment to another. Its subcharacteristics are installability, replaceability, adaptability, and compliance.

To use this as a checklist, you will identify one or more quality risks for each quality subcharacteristic. If you cannot identify one or more quality risks for a subcharacteristic, then you can delete it from your framework in the quality risks analysis document or, for auditability, make a notation that you could not identify any quality risks related to this subcharacteristic.

Identify and Assess: Process Options for quality risk Analysis

At a high level, the process for carrying out a quality risk analysis is straightforward:

1. Select a technique for identifying and assessing quality risks and the associated template to capture the information generated.
2. Assemble a cross-functional team to perform the quality risk analysis.
3. Perform the quality risk analysis, documenting the results in the template.

4. Verify the distribution of the risk-level ratings to ensure adequate dispersal, and adjust the ratings if necessary.
5. If specifications documents related to requirements, design, or use cases exist, establish traceability between the elements of these documents and the quality risk items.
6. If a set of test cases already exists, for example, in the case of an existing product for which a new version is being prepared, establish traceability between the existing test cases and the quality risk items.
7. Check the document into whatever repository you use for test system documents.

Let's take a closer look at each of these steps.

1. **Select a technique for identifying and assessing quality risks and the associated template to capture the information generated.** In the next section, I'll cover the various techniques you and your team can use to identify and assess quality risks. They vary in terms of the level of formality involved and in the amount of documentation they tend to produce, so you'll want to be careful here. Typically, I recommend that people start with an informal approach the first time. It's hard enough to institute quality risk analysis in its simplest form without adding additional difficulties related to large, complex documents and highly rigorous and formalized techniques. The selected technique will determine the template you need.
2. **Assemble a cross-functional team to perform the quality risk analysis.** A critical success factor for quality risk analysis is selecting the proper set of participants. You need a cross-functional team that includes technical stakeholders and business stakeholders. Technical stakeholders can include senior development team members, hardware engineers, network and database experts, help-desk or technical-support staff, and senior test staff, among others. Anyone who understands what is likely to go wrong with the system is a good candidate as a technical stakeholder.

   Business stakeholders can include sales people, marketing staff, business analysts, and product managers, among others. Anyone who understands the impact of potential problems in the system is a good candidate as a technical stakeholder. I must stress the importance of the cross-functional team. Having the right mix of participants will minimize the chances that your quality risk analysis will miss important quality risk items or assign improper risk levels to those items. In addition, keep in mind the human aspect of this process. Risk-based testing uses quality risk analysis to decide what to test, in what order, and how much. To put it another way, we are also deciding what not to test, what to test late in the project, and what to test very little if at all. If you successfully use the cross-functional team to build a cross-functional consensus around these decisions, you not only have the most accurate quality risk analysis possible, but also have support across the organization for these decisions.

The team should include the project manager and the development managers. By including them in the process, you'll build a level of comfort between these managers and you about what tests you plan to run. This mutual understanding prevents surprises and confrontations down the road, and assures the managers that you won't pull any ''gotcha'' maneuvers on them by running mysterious tests late in the game.

Finally, if your list of risk items somehow gets contaminated with any gross misunderstandings about what the product should do, the project manager and the development managers can help you clear up the misunderstandings at this early stage. This avoids the lost time and the potential embarrassment of developing new tests for — and even reporting bugs against — broken features that don't exist.

3. **Perform the quality risk analysis, documenting the results in the template.** With the team in place, you can now carry out the quality risk analysis. There are two general approaches to this. One approach is to hold a brainstorming session with the entire team. In this session, you use one or both of the checklists introduced earlier to facilitate the identification of quality risk items. Once the team has identified the quality risk items, you then assess each quality risk item to determine its level of risk, according to the rules of the technique you selected. This approach can work well, but the brainstorming session can consume the better part of day or even more than one day for a large product. You might not want — or be able — to hold such a session.

   The other approach is to hold a sequence of one-on-one or small-group interviews with distinct stakeholder groups. In each interview, you identify quality risk items, again using one or both of the checklists to structure the interview. After the interviews, you organize a smaller focus group of stakeholders to review the list and, during that review, to assess the level of risk for each risk item. This approach can also work well, but you must remember to hold a final review with the whole team to achieve the consensus-building aspect of the process, since not all the participants will have seen the entire list of quality risk items or their assessed risk levels.

   Whichever of these two approaches you use, you'll need to capture the results in the template associated with the selected technique. I usually save this information in a spreadsheet. I call the worksheet with the initial results something like ''Initial Risk Analysis.'' As I update the analysis in the subsequent steps of the process or at later points in the project, I'll use a different worksheet with a different name, preserving the initial results for reference.

4. **Verify the distribution of the risk-level ratings to ensure adequate dispersal, and adjust the ratings if necessary.** One of the challenges of quality risk analysis, which I'll discuss in a subsequent section, is the tendency for the risk-analysis team to turn in initial risk-level assessments that do not do a good job of differentiating the levels of risk associated with risk items. In other words, the risk ratings are clumpy or skewed. Therefore, you should plan to check the distribution of the risk-level ratings. If you use Excel to capture the results, this is easy, because you can create a histogram that shows the distribution. You want to see an approximately normal or bell-curve type of distribution. If not, you should validate the ratings and adjust them.
5. **If specifications documents related to requirements, design, or use cases exist, establish traceability between the elements of these documents and the quality risk items.** At this point, you should evaluate specifications documents, if you have them. These documents could include requirements specifications, design specifications, or use cases. You should establish traceability between the elements of these documents and the quality risk items. This process might result in the discovery of additional risk items. With this traceability, should the specifications change during the project, rather than having to evaluate the effect of the changes on all of the quality risks, you can focus on those risk items related to the changed specification elements.
6. **If a set of test cases already exists, for example, in the case of an existing product for which a new version is being prepared, establish traceability between the existing test cases and the quality risk items.** You (or perhaps senior test engineers on your test team) should also evaluate any existing test cases. For example, with an existing product for which a new version is being created, you are likely to have a significant set of test cases inherited from the previous versions. You should establish traceability between the existing test cases and the quality risk items. This process also might result in the discovery of additional risk items. In addition, this process might result in the discovery of significant areas of under-testing and over-testing, relative to the level of risk associated with various quality risk items. You should plan to address those problems during development of the test system, which I'll discuss in Chapter 3.
7. **Check the document into whatever repository you use for test system documents.** At this point, you should finalize the document for the time

being. You'll need to adjust it later at major project milestones, but for the time being you have a solid foundation for your test estimate, your test plan, and your test system development. You should at least e-mail the final document around to the various project stakeholders, including the participants in the quality risk analysis. You should certainly make sure everyone on the test team has it, and understands how it will affect their work. You should check the document into whatever repository you use for test system documents.

quality risk analysis Techniques and Templates

You might select from any number of techniques for quality risk analysis. I've identified several major types of quality risk analysis techniques, each with significant variations, in use by project teams around the world.[7] These include the following:

- Informal
- ISO 9126
- Cost of exposure
- Hazard analysis
- Failure mode and effect analysis

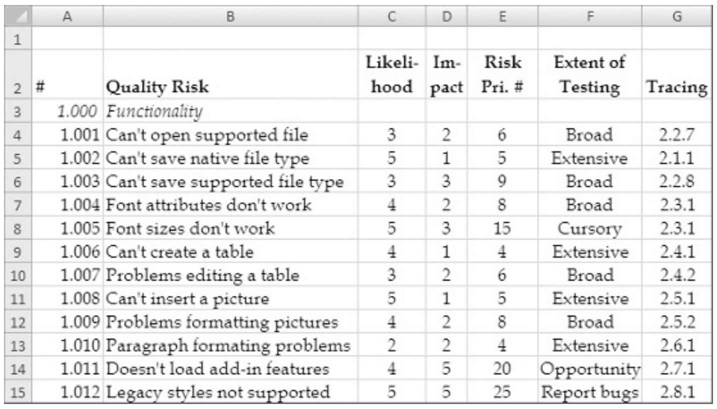
I'm sure there are one or two more out there that I haven't encountered yet.

The two most commonly used techniques, though, are the informal technique and failure mode and effect analysis (FMEA). I have used these and seen them used more often than any other set of techniques. Because they are the most commonly used, I'm going to focus on these techniques in this chapter.

I'll start with the informal technique. I use this one most frequently. If you are new to risk-based testing, you should start with this one. As I mentioned previously, it's hard enough to institute quality risk analysis in its simplest form. If you add all the difficulties related to the large, complex documents and the highly rigorous and formalized technique associated with failure mode and effect analysis, you are making this process much harder. Why do something the hard way, unless you are required to do so by some regulatory agency or company standard?

I'll illustrate this technique and the template by example. Figure 1-6 shows an example of a portion of the quality risk analysis for SpeedyWriter. I'll explain each column.

[7]For a survey of these five techniques, see my article "Quality Risk Analysis," available in the Basic Library at `www.rbcs-us.com`. You can also find this article in *Fundamental Concepts for the Software Quality Engineer, Volume 2*, edited by Taz Daughtry.



	A	B	C	D	E	F	G
1							
2	#	Quality Risk	Likeli-hood	Im-pact	Risk Pri. #	Extent of Testing	Tracing
3	*1.000*	*Functionality*					
4	1.001	Can't open supported file	3	2	6	Broad	2.2.7
5	1.002	Can't save native file type	5	1	5	Extensive	2.1.1
6	1.003	Can't save supported file type	3	3	9	Broad	2.2.8
7	1.004	Font attributes don't work	4	2	8	Broad	2.3.1
8	1.005	Font sizes don't work	5	3	15	Cursory	2.3.1
9	1.006	Can't create a table	4	1	4	Extensive	2.4.1
10	1.007	Problems editing a table	3	2	6	Broad	2.4.2
11	1.008	Can't insert a picture	5	1	5	Extensive	2.5.1
12	1.009	Problems formatting pictures	4	2	8	Broad	2.5.2
13	1.010	Paragraph formating problems	2	2	4	Extensive	2.6.1
14	1.011	Doesn't load add-in features	4	5	20	Opportunity	2.7.1
15	1.012	Legacy styles not supported	5	5	25	Report bugs	2.8.1

Figure 1-6 Example of informal quality risk analysis for SpeedyWriter

The leftmost column is a unique identifier for each row. Each quality risk category appears at the top of a list of one or more quality risk items. Each quality risk category has a sequential number (1, 2, 3, and so forth). Each quality risk item listed for each quality risk category has a sequential number (1.001, 1.002, 1.003, and so forth). This number is useful in capturing traceability from test cases back to the risks.

The Quality Risk column shows the quality risk categories, and, within each category, the specific quality risk items. The quality risk categories thus serve as a framework for the quality risk items associated with them.

The Likelihood column captures the likelihood for each quality risk item. This is the quality risk analysis team's assessment of the likelihood of the risk item becoming an actual problem. In other words, how likely does the team feel it is that one or more bugs will exist for this item? The team should determine likelihood based primarily on technical considerations. In this example, the team rated the likelihood on a 1–5 scale as follows:

- **1: Very high.** Such bugs are almost certain to occur.
- **2: High.** Such bugs are more likely to occur than not to occur.
- **3: Medium.** Such bugs have about an even chance of occurring as not occurring.
- **4: Low.** Such bugs are less likely to occur than not to occur.
- **5: Very low.** Such bugs are almost certain not to occur.

Of course, you can use another scale if you'd like. I have found this five-point scale to work quite well, though.

The Impact column captures the impact for each quality risk item. This is the quality risk analysis team's assessment of the business impact that would result if the system had bugs related to this risk item that escaped. In other words, how bad would it be if we delivered one or more bugs for this item to the customers and users? (Remember, the focus here is on the quality risk analysis team's perception of business impact, not the test team's perception of impact.) The team should determine impact based primarily on business considerations. In this example, the team rated the likelihood on a 1–5 scale as follows:

- **1: Very high.** Such bugs would render the system unsalable.
- **2: High.** Such bugs would seriously impair users and significantly reduce sales.
- **3: Medium.** Such bugs would inconvenience some users and marginally reduce sales.
- **4: Low.** Such bugs would inconvenience a few users and might reduce sales.
- **5: Very low.** Such bugs would only rarely affect users and would not reduce sales.

As before, you can use another scale, but this five-point scale works quite well in practice.

The Risk Priority Number column shows the aggregate level of risk for each risk item. In this example, the spreadsheet calculates the number by multiplying likelihood and impact. You can adopt another formula for calculating the risk priority number if you like, including weighting likelihood or impact to emphasize technical or business considerations, respectively. When the test team develops test cases from the quality risk analysis, each test case inherits the risk priority number from its parent risk item. The test team can then run the test cases in risk-priority order, starting with the tests that have a risk priority number of one. If the testers must delete test cases due to schedule pressure, they can do so in reverse risk-priority-number order.

While the risk priority number allows you to sequence and, if necessary, triage test cases based on risk, you still need some way to determine the amount of effort to expend on testing a particular risk item. In other words, based on the risk priority number, what relative amount of test effort should we put into developing and executing tests against this risk item? The Extent of Testing column shows this determination. In this example, the team has used the following breakdown for the extent of testing, based on the range of the risk priority number:

- **1–5. Extensive.** Run a large number of tests that exercise many combinations and variations of interesting conditions.
- **6–10. Broad.** Run a medium number of tests that exercise many different interesting conditions for the risk item.
- **11–15. Cursory.** Run a small number of tests that sample the most interesting conditions for the risk item.

- **16–20. Opportunity.** Leverage other tests to explore the risk item with minimal effort and only if the opportunity presents itself.
- **21–25. Report bugs.** Only report bugs observed for this risk item if discovered during other tests.

As with the Likelihood and Impact columns, feel free to use some other breakdown if it makes more sense for you. Unlike with the Likelihood and Impact columns, I have quite frequently found it necessary to fine-tune this heuristic for allocation of effort. Of course, if you adjust the risk-priority-number calculation, you'll need to adjust this heuristic, too.

Finally, the Tracing column captures the relationship between the requirements specification elements and the quality risk item. The numbers given in the column in Figure 1-6 correspond to the section numbers in the requirements document. I have not shown tracing for test cases, because I'm assuming this is a new project. Tracing for existing tests, if they exist, would typically involve another column.

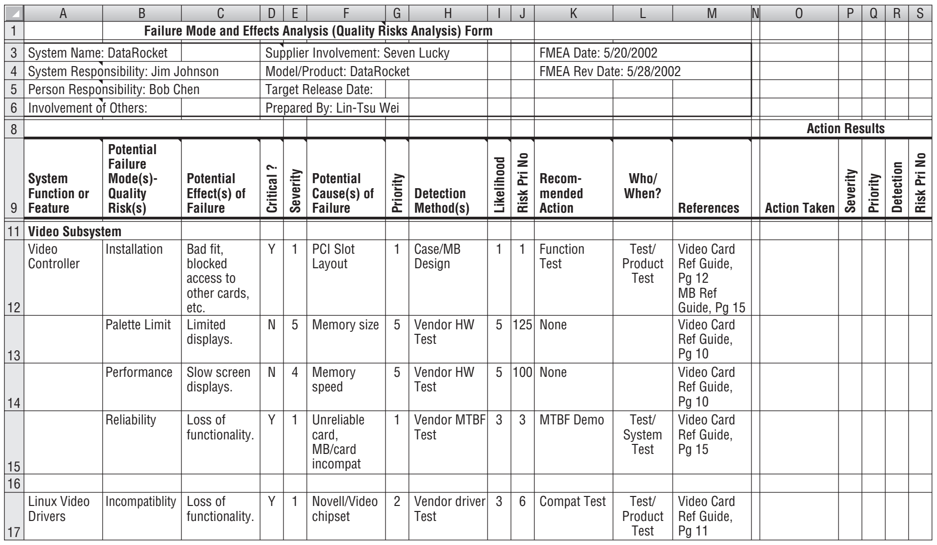
Having discussed the informal technique, let's move on to a formal technique for defining quality risks using an approach called *failure mode and effect analysis* (FMEA).

Fundamentally, FMEA is a technique for understanding and prioritizing possible failure modes (or quality risks) in system functions, features, attributes, behaviors, components, and interfaces. It also provides a means of preventive defect reduction and tracking process improvements. Preventive defect reduction arises because the technique is ideal applied not only to testing decisions, but to product design and implementation decisions. As these design and implementation decisions reduce the level of risk, iterative application of the technique shows the risk reduction, which allows us to re-focus our testing on the remaining higher risks.

Figure 1-7 shows the top page of a sample FMEA chart for DataRocket. Let's go through each of the columns in detail.

The System Function or Feature column is the starting point for the analysis. In most rows, you enter a concise description of a system function. If the entry represents a category, you must break it down into more specific functions or features in subsequent rows. Getting the level of detail right is a bit tricky. With too much detail, you can create an overly long, hard-to-read chart; with too little detail, you will have too many failure modes associated with each function.

In the Potential Failure Mode(s)-Quality Risk(s) column, for each specific function or feature (but not for the category itself), you identify the ways you might encounter a failure. These are quality risks associated with the loss of a specific system function. Each specific function or feature can have multiple failure modes.



Failure Mode and Effects Analysis (Quality Risks Analysis) Form

System Name: DataRocket	Supplier Involvement: Seven Lucky	FMEA Date: 5/20/2002
System Responsibility: Jim Johnson	Model/Product: DataRocket	FMEA Rev Date: 5/28/2002
Person Responsibility: Bob Chen	Target Release Date:	
Involvement of Others:	Prepared By: Lin-Tsu Wei	

													Action Results				
System Function or Feature	**Potential Failure Mode(s)-Quality Risk(s)**	**Potential Effect(s) of Failure**	**Critical ?**	**Severity**	**Potential Cause(s) of Failure**	**Priority**	**Detection Method(s)**	**Likelihood**	**Risk Pri No**	**Recom-mended Action**	**Who/ When?**	**References**	**Action Taken**	**Severity**	**Priority**	**Detection**	**Risk Pri No**
Video Subsystem																	
Video Controller	Installation	Bad fit, blocked access to other cards, etc.	Y	1	PCI Slot Layout	1	Case/MB Design	1	1	Function Test	Test/ Product Test	Video Card Ref Guide, Pg 12 MB Ref Guide, Pg 15					
	Palette Limit	Limited displays.	N	5	Memory size	5	Vendor HW Test	5	125	None		Video Card Ref Guide, Pg 10					
	Performance	Slow screen displays.	N	4	Memory speed	5	Vendor HW Test	5	100	None		Video Card Ref Guide, Pg 10					
	Reliability	Loss of functionality.	Y	1	Unreliable card, MB/card incompat	1	Vendor MTBF Test	3	3	MTBF Demo	Test/ System Test	Video Card Ref Guide, Pg 15					
Linux Video Drivers	Incompatiblity	Loss of functionality.	Y	1	Novell/Video chipset	2	Vendor driver Test	3	6	Compat Test	Test/ Product Test	Video Card Ref Guide, Pg 11					

Figure 1-7 A portion of the FMEA for DataRocket

In the Potential Effect(s) of Failure column, you list how each failure mode can affect the user, in one or more ways. I keep these entries general rather than trying to anticipate every possible unpleasant outcome.

In the Critical? column you indicate whether the potential effect has critical consequences for the user. Is the product feature or function completely unusable if this failure mode occurs?

In the Severity column, you capture the effect of the failure (immediate or delayed) on the system. This example uses a scale from 1 to 5 as follows:

1. Loss of data, hardware damage, or a safety issue
2. Loss of functionality with no workaround
3. Loss of functionality with a workaround
4. Partial loss of functionality
5. Cosmetic or trivial

Some books on FMEA show the use of a reverse scale, in which larger numbers denote greater severity. However, I prefer to use the scale shown here, which is more in line with the typical use of the term *severity* as I've encountered it.

In the Potential Cause(s) of Failure column, you list possible factors that might trigger the failure — for example, operating-system error, user error, or normal use. In my experience, this column is not as important as others when you are using an FMEA strictly as a test design tool.

In the Priority column, you rate the effect of failure on users, customers, or operators. This example uses a scale from 1 to 5, as follows:

1. Complete loss of system value
2. Unacceptable loss of system value
3. Possibly acceptable reduction in system value
4. Acceptable reduction in system value
5. Negligible reduction in system value

Because these are subjective ratings highly dependent on an understanding of the business, you should rely on input from sales, marketing, technical support, and business analysts.

In the Detection Method(s) column, you list a currently existing method or procedure, such as development activities or vendor testing, that can find the problem before it affects users, excluding any future actions (such as creating and executing test suites) you might perform to catch it. (If you do not exclude the tests you might create, the next column will be skewed.)

In the Likelihood column, you have a number that represents the vulnerability of the system, in terms of: a) existence in the product (e.g., based on technical risk factors such as complexity and defect history); b) escape from the current development process; and c) intrusion on user operations. This example uses the following 1-to-5 scale:

1. Certain to affect all users
2. Likely to impact some users
3. Possible impact on some users
4. Limited impact to few users
5. Unimaginable in actual usage

This number requires both technical judgment and an understanding of the user community, which makes participation by programmers and other engineers along with business analysts, technical support, marketing and sales important.

As with the informal technique, the RPN (Risk Priority Number) column tells you how important it is to test this particular failure mode. The risk priority number (RPN) is the product of the severity, the priority, and the likelihood. Because this example used values from 1 to 5 for all three of these parameters, the RPN ranges from 1 to 125.

The Recommended Action column contains one or more simple action items for each potential effect to reduce the related risk (which pushes the risk priority number toward 125). For the test team, most recommended actions involve creating a test case that influences the likelihood rating.

The Who/When? column indicates who is responsible for each recommended action and when they are responsible for it (for example, in which test phase).

The References column provides references for more information about the quality risk. Usually this involves product specifications, a requirements document, and the like.

The Action Results columns allow you to record the influence of the actions taken on the priority, severity, likelihood, and RPN values. You will use these columns after you have implemented your tests, not during the initial FMEA.

As with the informal technique, you can use a cross-functional brainstorming session to populate your FMEA chart. To do so, you gather senior representatives from each team — development, testing, marketing, sales, technical support, business analysts, and so forth — and fill in the chart row by row. This is certainly the best way, but it requires a commitment from each group to send a participant to a meeting that could consume a day or more. If you can't get people to attend a cross-functional brainstorming session like this, you can proceed with the interview approach discussed for the informal technique, too.

I have encountered a few pitfalls in using the FMEA method. In some cases, I have become distracted by quality risks that lie outside the scope of the test project. If I am working on SpeedyWriter, for example, I don't need to worry about operating-system bugs or underlying hardware failures. For DataRocket, I needn't analyze possible low-level failures in drives or chips. If I find a bug related to a given failure mode, will the development team — or some other group — address it? If not, it's out of scope.

The resulting FMEA document will be large. Be sure that you are ready to maintain this document after you create it. Otherwise, you won't be able to use it to focus test development and execution, which defeats the purpose.

Tips and Challenges of quality risk Analysis

I'll finish up this discussion on quality risk analysis by offering a few tips on how to best handle quality risk analysis. I'll also list some challenges you need to take into account.

The first tip relates to keeping the proper degree of detail. How precise should you be with your quality risk items? Sometimes you might feel a risk item covers too much ground. Should you separate it? The rule of thumb is to only separate one risk item into two or more risk items when necessary to distinguish between different levels of risk. Too much detail makes the documents hard to manage. Too little detail makes it impossible to prioritize test cases accurately.

The second tip relates to respecting your quality risk analysis team and their precious time. The quality risk analysis brainstorming sessions can require one or more entire days. Even the interviews can take up to two hours. Make sure people know what they are committing to when you ask them to participate.

My final tip relates to the type of quality risk analysis and thus quality risk management you are doing. It looks quantitative, since you have numbers. However, the numbers are just shorthand for classifications, used to allow you to calculate the risk priority number by using a mathematical equation rather than a table. You must understand that what you are doing is qualitative risk management. Based on the subjective opinions and collective wisdom of the participants, the process assigns relative priorities to risk items. Quantitative risk management is something that insurance companies and banks, armed with statistically valid data, can do. You don't have such data.

Let's look at some challenges. The first is building consensus on risk. Sometimes participants disagree about likelihood or priority ratings. You can try to use some of your political influence to broker a compromise. You can also see if the various participants can educate each other on how they should rate the risks. However, if you can't reach consensus, you need to escalate to some decision-maker who is ultimately responsible for the quality of the delivered system.

Another challenge is to avoid priority inflation. Sure, in the absence of constraints, everyone would want to test everything extensively. That's not connected to reality. If you see that priority inflation is happening, ask people what they would give up for additional test coverage. Is a week or a month of schedule delay acceptable? Is an extra $10,000 or $100,000 in test budget acceptable? Is dropping features that were not extensively tested acceptable? If the answers to these questions are no, no, and absolutely not, then perhaps you can apply a reality check to people's gold-plated test coverage aspiration.

The final challenge is getting the participants to make rational decisions about risk. Consider this example: Which is safer, air travel or driving? Most people know, rationally, that air travel is safer. In fact, on a distance-traveled basis, air travel is about 100 times safer. However, more people have a fear of flying than of driving. This same kind of irrationality can afflict the quality risk analysis process. If you think people are overstating or understating likelihood or impact during the process, you should try to question the reasons for a particular rating to see if you can get people to be more rational in their assessment.

What You *Can* Test: Schedule, Resources, and Budget

Whether you've used an informal approach or the more formal FMEA technique, you now have a prioritized outline of quality risks. This is analogous to the requirements for the overall project; the list of critical quality risks documents the essential requirements for my test effort. Now I need to figure out a test schedule and a budget that will allow me to test the scariest risks.

One of my first managers was fond of this saying: ''Schedule, cost, and quality — pick two.'' This pithy remark means that while for a given feature set you can freely select any two of these variables, doing so determines the third variable. I call this rule, which is illustrated in Figure 1-8, the ''Iron Box and Triangle'' of system development. The clockwise arrow indicates refinement during the planning stage. These refinements balance features, schedule, cost, and quality. Once implementation begins, the feature set becomes more rigid, the schedule more painful to change, and budget increases less likely. Within the fixed box representing the feature set in Figure 1-8, the two lines that show the schedule and the cost determine the third line, quality, that completes the triangle.

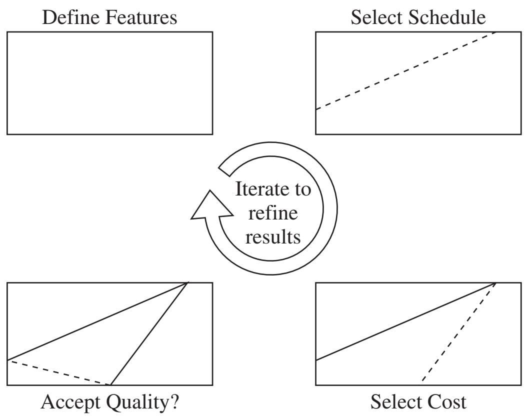


Figure 1-8 The feature, schedule, budget, and quality trade-offs

This creates a planning conundrum in that you have only a rough idea of what your test project is about, but the window of opportunity that might allow a realistic schedule and an adequate budget is closing. And even this scenario assumes that you are on the project team during the planning phase. The situation is worse if you have joined the team later, during implementation, as you might have a fixed budget and schedule. There's no perfect solution, but some project management techniques exist to help.

Shoehorning: Fitting a Test Schedule into the Project

Often, software and hardware project schedules don't evolve according to any textbook approach. You might have to start with a ship date and a list of product features — the negotiability of both varies — and build the schedule from there. How can you construct a workable test schedule within these constraints?

I use a work-breakdown structure, which is a top-down approach.[8] I find it intuitive to start with big categories of work and iteratively decompose them into discrete tasks, especially at the early stages when I don't have a lot of details. I start by breaking the test effort into major phases such as these:

- Planning (the work discussed in this chapter and the next)
- Configuration (getting the necessary hardware and other resources and setting up the test lab)
- Staffing (if applicable)
- Test development (building or deploying the test tools, creating the test suites and the test case library, putting the reporting tools in place, and documenting how the test process is to put these testware items into action)
- Test execution (running the tests, recording test status, and reporting results)

Next, I divide each phase into activities. Within the planning category, for example, I set up activities such as defining quality risks, creating the schedule and the budget, writing test plans, and selecting test tools. Other activities might include getting bids from third parties for their help, or hiring test technicians, test engineers, and system administrators.

After that, I decompose each activity into tasks, and then subtasks if necessary. This decomposition continues until I have constituent tasks that are one or two days long and are the responsibility of one person. (I don't decompose the work-breakdown structure this way for test execution; instead I use the test case estimating and tracking method illustrated in Chapter 5.) These small task definitions allow me to ascertain whether I'm on track during the project. Big tasks can get dangerously out of control, and I won't discover such problems until a long period (a week or more) has slipped past me.

The activities in the configuration phase depend on the test environment I need. Even though I probably don't know all the details at this point, my list of quality risks usually has given me some ideas. Once I think through the quality risks, I have a high-level perspective on the test suites I must create, which gives me a good idea of my test environment needs.

For development, you must deploy your test tools and then develop the test suites themselves. I often list separate major tasks for each test phase and then enter the test suites as individual tasks within each phase. Test suite development should proceed in priority order. Developing test suites is a

[8]If you aren't familiar with work-breakdown structures, I recommend *Effective Project Management* by Robert Wysocki, et al., as a good introduction to the topic.

full-time job, so I'm careful not to set up work on various suites as parallel tasks unless I have multiple test engineers or can give a single engineer twice as long to finish. In addition, I take care to add a task for the test engineers to document how the test system works, both in terms of the design and functionality of the testware and the way the test process uses that testware to find bugs.

For test execution, there are two important questions to answer in coming up with a good estimate. First, how long will it take to run all the tests once (which I often refer to as a single *test pass*)? Second, how many times will I need to run the tests against successive test releases to find all the important bugs and subsequently confirm the fixing of those bugs (which I refer to as the *number of test cycles*)? Suppose I have six person-weeks of testing work defined for the system test phase and three testers allocated to run the tests. Then, each pass takes my test team two weeks. If I have found, on previous projects with the same project team, that we need to run six cycles to find and fix the important bugs, then I have six cycles, say one week each, with three passes in those cycles (see Figure 1-9).

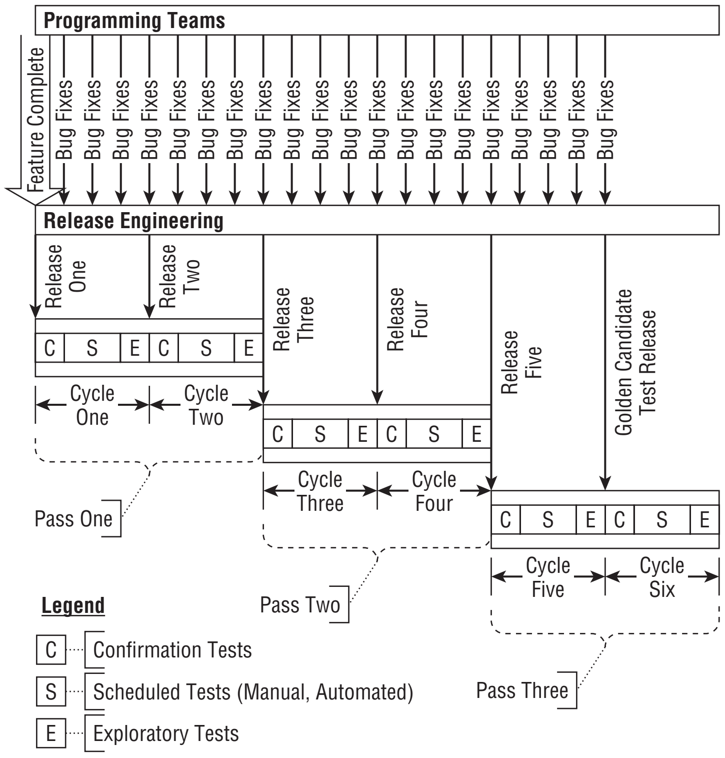


Figure 1-9 System test passes, releases, and cycles

The time required to run the tests is something a test manager can control and measure. As long as I can come up with solid estimates of how long and how much effort each test case requires, I can add up those numbers across the entire test set and use simple math to predict test pass duration. However, the number of cycles is dependent on many factors outside my control as a test manager. If the quality of the software is poor, then more cycles will be required. If the programmers are slow to fix bugs, then more cycles will be required. If bug fixes tend to break other areas of the product (i.e., to introduce regression bugs), then more cycles will be required.

If I have no historical data with the project team, I have to take a guess on the number of cycles. I've successfully used six one-week cycles as a rule of thumb on a number of projects, although you'll need to consider your project's size and complexity carefully before adopting it. If the set of test cases is small enough, you can run a full pass in each cycle, which has benefits in terms of regression risk. (I'll talk more about test sets, suites, cycles, passes, phases, regression risks, and confirmation testing in Chapter 3.) More commonly, I estimate one week per test cycle, two or three cycles per pass, and three passes per phase. One of my clients, though, has a test set that requires about two person-decades of effort. The whole team runs each test case once as part of a year-long test pass.

However the passes and cycles work out, I break the rules a bit on the work-breakdown structure in the area of test execution, and assign the entire team to test tasks at the test cycle level of detail. I've found that trying to assign a single tester to a single test case in a work-breakdown structure doesn't work well. For one thing, I generally don't know all the tests I'm going to run during this early planning stage, so if I account only for the tests I know, I'll underestimate. Moreover, the project management tools I've worked with do a poor job of managing test cases as tasks. My usual rule of thumb is to estimate the test effort, increase it by 50 percent for the tests I don't know about yet, plan on between 20 and 30 hours of testing per tester per week, and just do the math.

In this first cut, I try to capture the basic dependencies. For example, I must develop a test suite before I can run it, and I can't start a test cycle until I receive something to test. Although some dependencies that loom far in the future won't jump out at me, my best effort will probably suffice. I try to build some slack and extra time into my schedule for the inevitable discoveries. Good project management process dictates that I track against and revise this schedule throughout my project, so I can add dependencies as they become apparent, but increases in schedule and budget numbers after the initial plan are often difficult to negotiate.

As I create the tasks, I also assign resources, even if I can't be complete at this point. I don't worry about staples such as desks, workstations, or telephones unless I have genuine concerns about getting an adequate budget for them. I focus on items such as these:

- Expensive resources such as networks, environmental test equipment, and test tools
- Resources that require long lead times, such as lab space that must be rented and set up, or ISDN and T1 lines
- Missing resources such as people I need to hire
- External resources such as third-party labs
- Scarce resources such as my test engineers and technicians

I'm careful not to over-utilize resources that have limited bandwidth or availability. People, for example, can do only one thing at a time, and you can easily overtax shared resources such as servers, printers, networking infrastructure, and the like if you're not careful. Certain types of tests, such as performance and reliability, require a dedicated set of resources for accurate results.

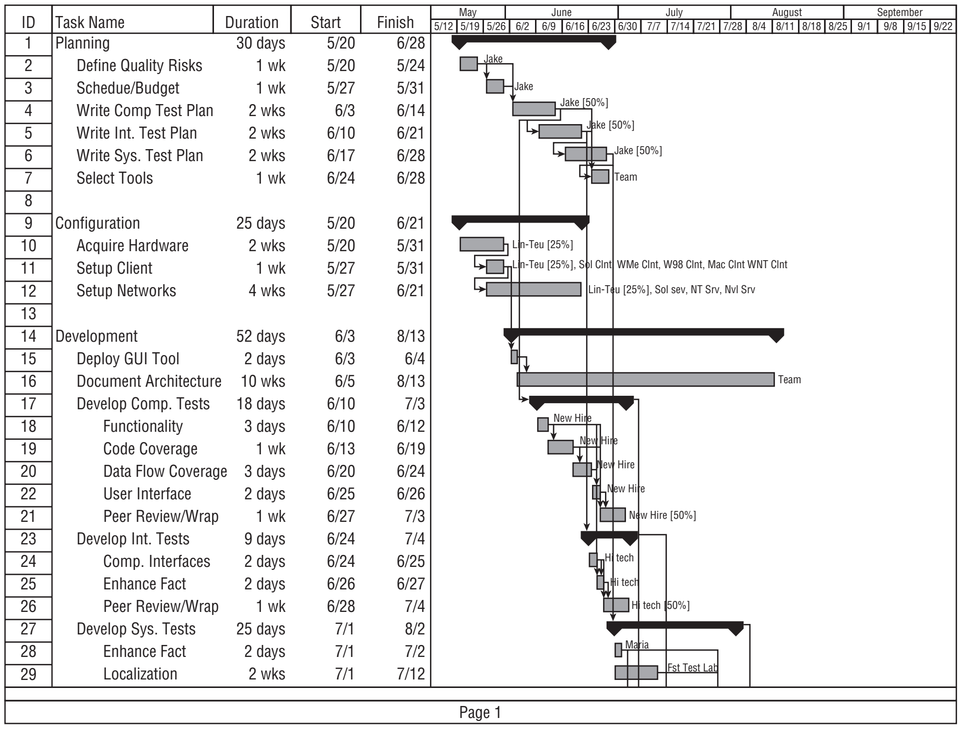
Accurate scheduling requires the participation of the actual contributors wherever possible. For example, it's better for the test engineer who'll design and implement an automated test suite to tell me how long it will take her than for me to guess myself, especially if she has experience doing test automation and I don't! The more experience the contributor has with the task in question and the tool to be used, the more accurate her estimate will be. There are also so-called Delphic oracle approaches where you poll multiple people on the team for best-case, worst-case, and expected-case task durations, then take the averages of those to come up with best-case, worst-case, and expected-case schedules. You might want to apply these types of approaches on long, complex projects where the consequences of error in the schedule are severe. No matter how careful you are, though, some studies have shown that initial estimates are off by 50 to 200 percent.[9]

One thing I do to improve the accuracy of my schedules is to refer to published rules of thumb to sanity-check my estimates.[10] Capers Jones, in *Estimating Software Costs*, includes an entire chapter of such rules. Various presentations and articles on test estimation, including rules of thumb, appear from time to time at conferences and in testing journals. You might want to accumulate a collection of estimation rules that you use to check your work-breakdown structure.

[9]See Rita Hadden's "Credible Estimation for Small Projects," published in *Software Quality Professional*.

[10]For example, see my article, "Software Test Estimation," originally in *Software Testing and Quality Engineering* magazine, now on the Basic Library page of our company web site, `www.rbcs-us.com`.

Test estimation is hard to do perfectly, but not terribly hard to do well. If you follow good project management practices in preparing your work-breakdown structure, don't forget key tasks, estimate conservatively, don't overload people and resources, involve your entire team in estimation, and focus on key dependencies and deliverables, you can construct a draft schedule for your test project that should prove relatively accurate. I also make sure that my milestone dates fit within the constraints of the project. As the project proceeds, I track progress against the schedule, adding details, adjusting durations, resolving resource conflicts, including more dependencies, and so on. Figures 1-10 and 1-11 show an example of this approach, applied to testing for SpeedyWriter.

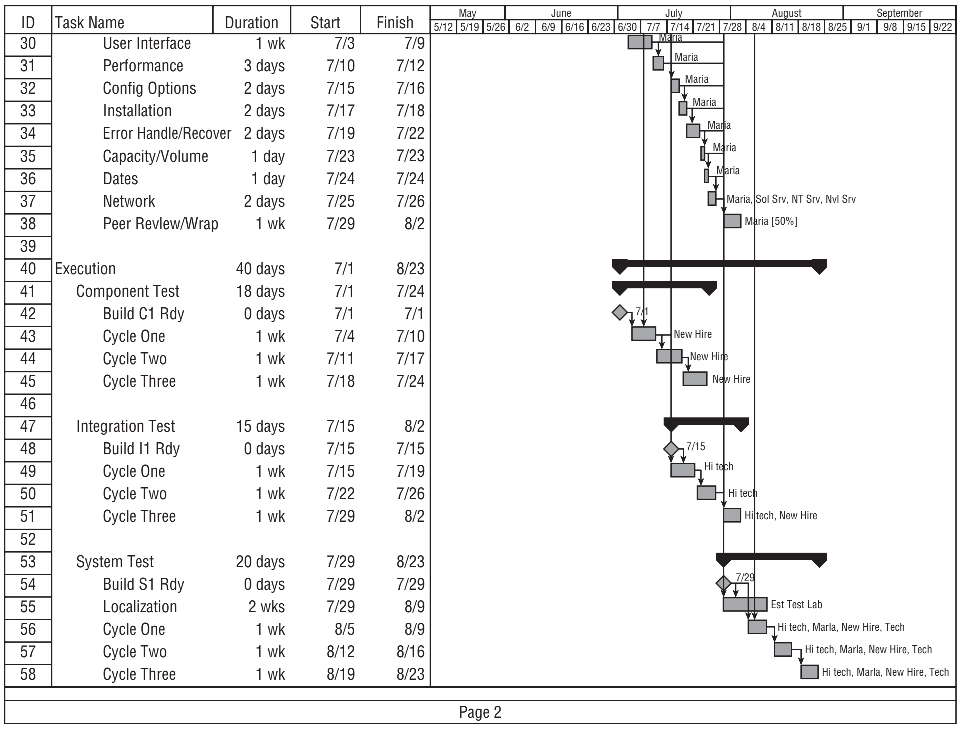


ID	Task Name	Duration	Start	Finish
1	Planning	30 days	5/20	6/28
2	Define Quality Risks	1 wk	5/20	5/24
3	Schedule/Budget	1 wk	5/27	5/31
4	Write Comp Test Plan	2 wks	6/3	6/14
5	Write Int. Test Plan	2 wks	6/10	6/21
6	Write Sys. Test Plan	2 wks	6/17	6/28
7	Select Tools	1 wk	6/24	6/28
8				
9	Configuration	25 days	5/20	6/21
10	Acquire Hardware	2 wks	5/20	5/31
11	Setup Client	1 wk	5/27	5/31
12	Setup Networks	4 wks	5/27	6/21
13				
14	Development	52 days	6/3	8/13
15	Deploy GUI Tool	2 days	6/3	6/4
16	Document Architecture	10 wks	6/5	8/13
17	Develop Comp. Tests	18 days	6/10	7/3
18	Functionality	3 days	6/10	6/12
19	Code Coverage	1 wk	6/13	6/19
20	Data Flow Coverage	3 days	6/20	6/24
22	User Interface	2 days	6/25	6/26
21	Peer Review/Wrap	1 wk	6/27	7/3
23	Develop Int. Tests	9 days	6/24	7/4
24	Comp. Interfaces	2 days	6/24	6/25
25	Enhance Fact	2 days	6/26	6/27
26	Peer Review/Wrap	1 wk	6/28	7/4
27	Develop Sys. Tests	25 days	7/1	8/2
28	Enhance Fact	2 days	7/1	7/2
29	Localization	2 wks	7/1	7/12

Figure 1-10 The first page of a Gantt-chart view of the work-breakdown structure for SpeedyWriter

If you're new to management, you might feel a bit daunted by the prospect of doing a work-breakdown structure. I encourage you to jump in with both feet, picking up a good self-study book first and then cranking out your first test project schedule with one of the project management tools on the market. My ability to schedule projects continues to improve — partly as a result of acquiring skills with the tools, although mostly as a result of experience — but

I started with simple schedules and ran a number of test projects successfully. Scheduling and project management are not trivial skills, so keep it simple to start. Simple schedules are less precise but more accurate. If you try to create complicated 300-task schedules, you can get lost in the minutiae.



ID	Task Name	Duration	Start	Finish
30	User Interface	1 wk	7/3	7/9
31	Performance	3 days	7/10	7/12
32	Config Options	2 days	7/15	7/16
33	Installation	2 days	7/17	7/18
34	Error Handle/Recover	2 days	7/19	7/22
35	Capacity/Volume	1 day	7/23	7/23
36	Dates	1 day	7/24	7/24
37	Network	2 days	7/25	7/26
38	Peer Revlew/Wrap	1 wk	7/29	8/2
39				
40	Execution	40 days	7/1	8/23
41	Component Test	18 days	7/1	7/24
42	Build C1 Rdy	0 days	7/1	7/1
43	Cycle One	1 wk	7/4	7/10
44	Cycle Two	1 wk	7/11	7/17
45	Cycle Three	1 wk	7/18	7/24
46				
47	Integration Test	15 days	7/15	8/2
48	Build I1 Rdy	0 days	7/15	7/15
49	Cycle One	1 wk	7/15	7/19
50	Cycle Two	1 wk	7/22	7/26
51	Cycle Three	1 wk	7/29	8/2
52				
53	System Test	20 days	7/29	8/23
54	Build S1 Rdy	0 days	7/29	7/29
55	Localization	2 wks	7/29	8/9
56	Cycle One	1 wk	8/5	8/9
57	Cycle Two	1 wk	8/12	8/16
58	Cycle Three	1 wk	8/19	8/23

Page 2

Figure 1-11 The second page of a Gantt-chart view of the work-breakdown-structure for SpeedyWriter

Estimating Resources and Creating a Budget

Given a work-breakdown structure with detailed resource allocations, I can hammer out a budget in a couple of hours. Again, I use a top-down approach. I first create a list of resources, starting with general categories such as these:

- **Staff.** This category includes permanent employees, contractors, and consultants.
- **Test tools.** If I'm testing software, I might need code-coverage analyzers, scripting utilities, GUI test automation systems, low-level diagnostic programs, and so forth. Hardware testing can involve oscilloscopes, shock and vibration tables, thermal chambers, and other equipment. Don't forget basic utilities for hardware and software testing.

- **Facilities and overhead.** Items in this category can include travel allowances, lab space, workstations, and infrastructure such as cabling, routers, hubs, bridges, ISDN terminals, and so forth.
- **Test environment.** This category includes the hardware, software, engineering samples, and experimental prototypes.
- **External labs.** I include this category if I intend to use external labs for environmental testing, localization, performance, or other purposes (see Chapter 10).

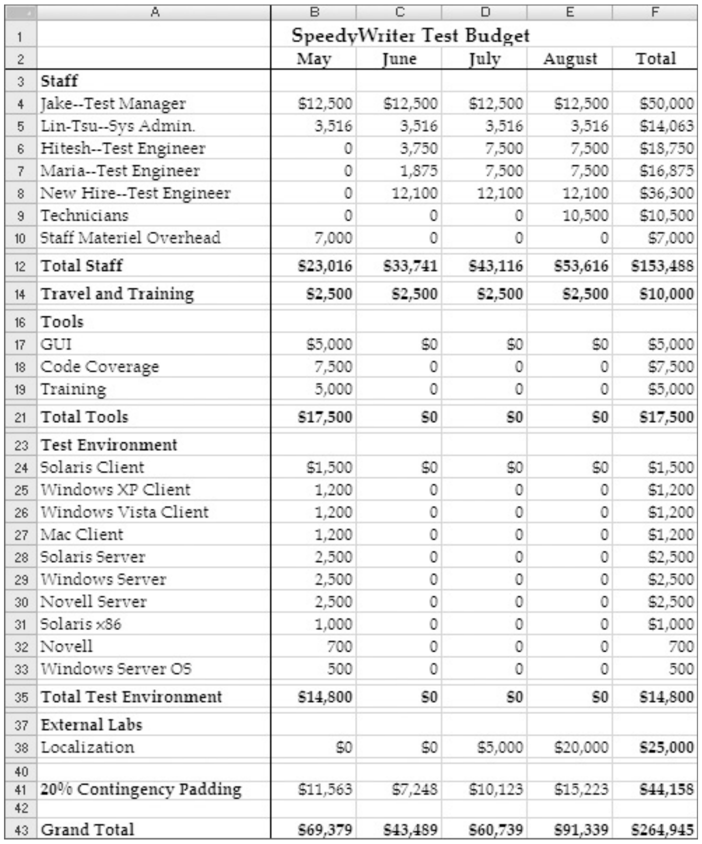
Within each category, I list the individual items I will need. I use placeholders to indicate where I might add items or quantities later.

To transform this resource list into a budget, I load it into a spreadsheet and line up columns to the right for each month of the project. For each item, I enter a cost figure — a monthly figure for variable costs or a one-time figure for fixed costs. Don't forget hidden or invisible costs such as burden rates for staff, agency markups for staff, application software, support contracts, and training.

If you find it difficult to estimate costs for tools, facilities, infrastructure, or the test environment, you can hit the Web or make some telephone calls. Estimating the cost of using an external lab might require an actual bid, although you can probably get a rough estimate by calling the lab. For unknown items — the placeholders on your resource list — you'll simply have to make an educated guess. Pick a comfortable figure with some wiggle room, but don't allow so much wiggle room that you'll be shot down when you approach management.

At this point, you can compare each line item against your schedule. When do you start using the resource? How long do you use it? Are ramp-up and ramp-down times associated with the resource? Answering these questions will tell you which months must absorb charges for each item, and what fraction of the charge applies in beginning and ending months. For fractions, I keep it simple; I find that halves and quarters are usually precise enough.

As I do for my schedules, I run a sanity check to ensure that all the numbers make sense. If allowed, I involve my staff in the process, making sure that they don't see each other's salary information. (Check with your management before circulating any proposed budget among your staff. Some companies don't allow individual contributors to see any budget information.) After coming up with the budget, I usually sleep on it and then review it the next day. I ask myself whether I've forgotten anything. If the budget contains a few gaping holes where I don't have enough information to even hazard a guess, I'll be honest and indicate that. Figure 1-12 provides an example of a budget for SpeedyWriter, assuming the schedule shown in Figures 1-10 and 1-11.



	A	B	C	D	E	F
1		SpeedyWriter Test Budget				
2		May	June	July	August	Total
3	**Staff**					
4	Jake--Test Manager	$12,500	$12,500	$12,500	$12,500	$50,000
5	Lin-Tsu--Sys Admin.	3,516	3,516	3,516	3,516	$14,063
6	Hitesh--Test Engineer	0	3,750	7,500	7,500	$18,750
7	Maria--Test Engineer	0	1,875	7,500	7,500	$16,875
8	New Hire--Test Engineer	0	12,100	12,100	12,100	$36,300
9	Technicians	0	0	0	10,500	$10,500
10	Staff Materiel Overhead	7,000	0	0	0	$7,000
12	**Total Staff**	**$23,016**	**$33,741**	**$43,116**	**$53,616**	**$153,488**
14	**Travel and Training**	**$2,500**	**$2,500**	**$2,500**	**$2,500**	**$10,000**
16	**Tools**					
17	GUI	$5,000	$0	$0	$0	$5,000
18	Code Coverage	7,500	0	0	0	$7,500
19	Training	5,000	0	0	0	$5,000
21	**Total Tools**	**$17,500**	**$0**	**$0**	**$0**	**$17,500**
23	**Test Environment**					
24	Solaris Client	$1,500	$0	$0	$0	$1,500
25	Windows XP Client	1,200	0	0	0	$1,200
26	Windows Vista Client	1,200	0	0	0	$1,200
27	Mac Client	1,200	0	0	0	$1,200
28	Solaris Server	2,500	0	0	0	$2,500
29	Windows Server	2,500	0	0	0	$2,500
30	Novell Server	2,500	0	0	0	$2,500
31	Solaris x86	1,000	0	0	0	$1,000
32	Novell	700	0	0	0	700
33	Windows Server OS	500	0	0	0	500
35	**Total Test Environment**	**$14,800**	**$0**	**$0**	**$0**	**$14,800**
37	**External Labs**					
38	Localization	$0	$0	$5,000	$20,000	**$25,000**
40						
41	**20% Contingency Padding**	$11,563	$7,248	$10,123	$15,223	**$44,158**
42						
43	**Grand Total**	**$69,379**	**$43,489**	**$60,739**	**$91,339**	**$264,945**

Figure 1-12 SpeedyWriter budget

Negotiating a Livable Test Project

With a quality risks list, schedule, and budget, I have a concise package that I can take to management. By speaking management's language, I can address four key questions that will arise:

- What type of risk management are we buying?
- How long will it take?
- What will it cost?
- What's the return on investment (see Chapter 11)?

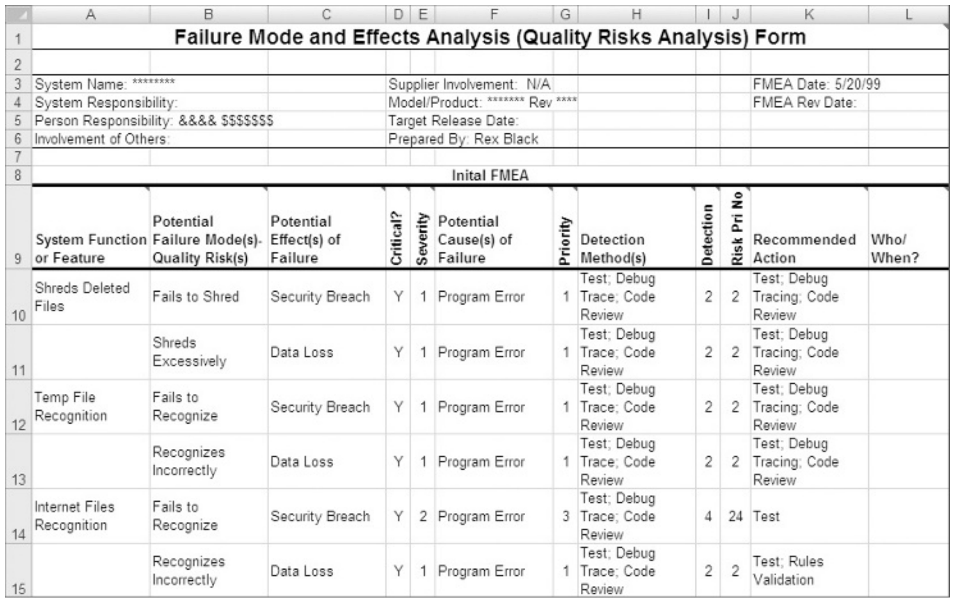
Although each company has a different process for approving a test program, every project I've worked on has required some degree of discussion, explanation, and negotiation. Be flexible. If management insists on reduced costs or a faster schedule (or both), I eliminate tests in reverse priority order. If cost is the major concern but I can add a few weeks to the schedule, perhaps I can get by with one less employee. Outsourcing can also reduce costs when done wisely, as you'll see in Chapter 10. I make the case for what I believe needs to be done, but I'm prepared to do less. The only taboo is agreeing to do everything I initially proposed to do but in less time and/or for less money, unless management wants to cut out the contingency time (schedule slack) and money and run a high risk that later discoveries will break my budget or schedule. If I've created a realistic schedule and budget, then agreeing to some faster schedule and lower budget that fits management desire but not reality is hardly doing anyone any favors, and it certainly won't help my credibility. If handed a non-negotiable dictate — for example, ''You will do this amount of testing in this period of time with this budget, end of discussion'' — then I simply agree to do the best job possible within those parameters, and move on.

At the end of this negotiation, I have an approved budget and schedule and a mutual understanding of the scope and deliverables for my test project. Now it's time to move on to creating a detailed plan, building the testware, and putting the resources in place to carry out the project.

Case Study

On one project, we applied the failure mode and effect analysis (FMEA) technique to analyze possible ways that an application might fail. This application provided secure-file deletion functionality for PC users running some versions of the Windows operating system. We had a six-hour cross-functional meeting with the test team, the programmers, the project manager, a salesperson, and the marketing manager. In this meeting, we discussed ways in which the system might fail and possible approaches to mitigation. These included both testing and various programming-process improvements. Figure 1-13 shows the top portion of this 100-item document. The full document is available at `www.rbcs-us.com`.

Notice that while the DataRocket example shown earlier in the chapter broke down failure modes based on major subsystems of the server, in this case the analysis starts with major functionality and expected behaviors. You can also use categories of quality risk as the starting point.



	A	B	C	D	E	F	G	H	I	J	K	L
1	Failure Mode and Effects Analysis (Quality Risks Analysis) Form											
2												
3	System Name: ********			Supplier Involvement: N/A							FMEA Date: 5/20/99	
4	System Responsibility:			Model/Product: ******* Rev ****							FMEA Rev Date:	
5	Person Responsibility: &&&& $$$$$$$			Target Release Date:								
6	Involvement of Others:			Prepared By: Rex Black								
7												
8	Inital FMEA											
9	System Function or Feature	Potential Failure Mode(s)- Quality Risk(s)	Potential Effect(s) of Failure	Critical?	Severity	Potential Cause(s) of Failure	Priority	Detection Method(s)	Detection	Risk Pri No	Recommended Action	Who/ When?
10	Shreds Deleted Files	Fails to Shred	Security Breach	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	
11		Shreds Excessively	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	
12	Temp File Recognition	Fails to Recognize	Security Breach	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	
13		Recognizes Incorrectly	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Debug Tracing; Code Review	
14	Internet Files Recognition	Fails to Recognize	Security Breach	Y	2	Program Error	3	Test; Debug Trace; Code Review	4	24	Test	
15		Recognizes Incorrectly	Data Loss	Y	1	Program Error	1	Test; Debug Trace; Code Review	2	2	Test; Rules Validation	

Figure 1-13 A case study FMEA

Exercise

1. Based on your reading of the Omninet Marketing Requirements Document, the Omninet System Requirements Document, and your experience with testing and bugs, perform a risk analysis for Omninet.[11]
   a. What will determine how long it takes to finish test execution for the Omninet project? You have to consider the following:
      - The time required to run each test once
      - The time required to find, fix, and confirm the fix for each bug
   b. What other factors should you consider?
   c. What management/stakeholder-expectation issues might also affect the test schedule?

[11] This exercise and solution are adapted from Chapter 7 of my book *Pragmatic Software Testing*.