CHAPTER 2

Plotting and Presenting Your Course: The Test Plan

This chapter offers a practical approach to writing one or more test plans for your project. I'll walk through a sample template that I use to develop a solid test plan—and I'll look at the issue of getting the plan approved once I've drawn it up.

Why I Write Test Plans

In Chapter 1, "Defining What's on Your Plate: The Foundation of a Test Project," I discussed how I set the scope, schedule, and budget parameters for my test projects. Given a budget, resource commitments, and a schedule, can you claim that you have a test plan? Some people do. In my experience, however, you need more detail to successfully manage a test project. Below the objectives and estimates featured in Chapter 1 lurks another layer of complexity, right above the specific details of test suites—and it pays to consider this complexity in advance.

Writing a test plan gives you a chance to collect your thoughts, your ideas, and your memories. Undoubtedly you've learned a great deal throughout the course of your career. Writing a thorough test plan gives you a chance to crystallize that knowledge into a concrete way of tackling the tasks ahead.

I see the test plan also as an opportunity to communicate with my test team, my development colleagues, and my managers. The most intense discussion of what testing is all about often occurs when I hold a test plan review. I appreciate the chance to have a forum focused solely on testing before a project enters the often-chaotic test execution periods, in which everyone can become so oriented toward minutiae that they lose sight of the big picture.

In some organizations, the test plan encompasses the entirety of the test effort, all the way down to defining all the individual test cases—often called the *test set* or the *test suites*—that the team will run. However, I recommend creating two distinct types of documents: first a test plan, and then an accompanying document detailing the test cases. The difference between a test plan and a test suite is a matter of strategy versus tactics: strategy consists of your overall plan for hunting down and identifying as many bugs as possible; tactics are the specific steps you will take to do this. This chapter focuses on the test plan itself; Chapter 3, "Test System Architecture, Cases, and Coverage," discusses the process of creating test suites, test cases, and other testware.

How Many Test Plans?

Suppose that you are working on SpeedyWriter. Further suppose that, as the test manager, you have responsibility for the component, integration, and system test phases, with an aggressive beta testing program during the system test phase. You thus have three distinct test subprojects to plan and manage. Do you write one plan or three? I favor using separate plans for test subprojects that are distinct in one or more of the following ways:

Different time periods. If the test planning, test development, test environment configuration, and test execution tasks for the subprojects start and end on different dates, I find that I have the information needed to plan some subprojects well before I get the necessary information for the other subprojects. (See Chapter 11, "Testing in Context: Economics, Lifecycles, and Process Maturity," for more information on system lifecycle models and test phase timing.) If I try to write only one test plan, I find myself forced to leave large sections of it marked "TBD" ("to be determined"), which can make the overall plan hard for people to understand and approve.

Different methodologies. Detailed discussions of code coverage instrumentation and platform-independent automated GUI test tools don't really go together. Likewise, in the case of hardware, discussions of thermal chambers, accelerometers, and business application compatibility testing can create a rather eclectic mix of topics in a plan.

Different objectives. If I'm trying to accomplish three different goals—in the current example, finding bugs in components, finding bugs in the relationships and interfaces between incrementally integrated components, and finding bugs in a fully integrated system—writing one plan each for component test, integration test, and system test allows me to focus my thinking on each goal in turn.

Different audiences. My test plan is not only my chance to inform my colleagues, my testers, and my managers of my vision, it is also a chance to discover their perspectives. This input is especially valuable to the extent that it gives me a better idea of how to focus my test efforts. If I write one lengthy test plan that addresses every issue, I might have trouble getting people to read it, not to mention getting them to participate in a three- or four-hour review. However, if I segment my planning, I'm better able to create documents that speak to the specific concerns of the individuals involved.

Multiple test plans can lead to overlapping content, though, which some people deal with by cutting and pasting shared sections such as test tracking, bug reporting and management, and revision control. Having the same information spread across multiple documents leaves you open to the possibility of inadvertent discrepancies or contradictory statements. When I have multiple test plans, I'll write a single master test plan that addresses these common topics and include references to them in the detailed test plans.

Using Drafts to Stimulate Discussion

I expect to release several versions of any test plan I write. Far from finding this frustrating, this sequence of drafts is a dialog that provides me an opportunity to pose questions to the readers. I use brackets in my plans (as opposed to colored fonts, which don't show up as well in hard copy) to indicate questions and open issues. My first drafts are always full of bracketed questions and statements such as these:

[TBD: Need to figure out what the hardware allocation plan is.]

[TBD: Need the Configuration Management team to define the revision numbering schema and the packaging.]

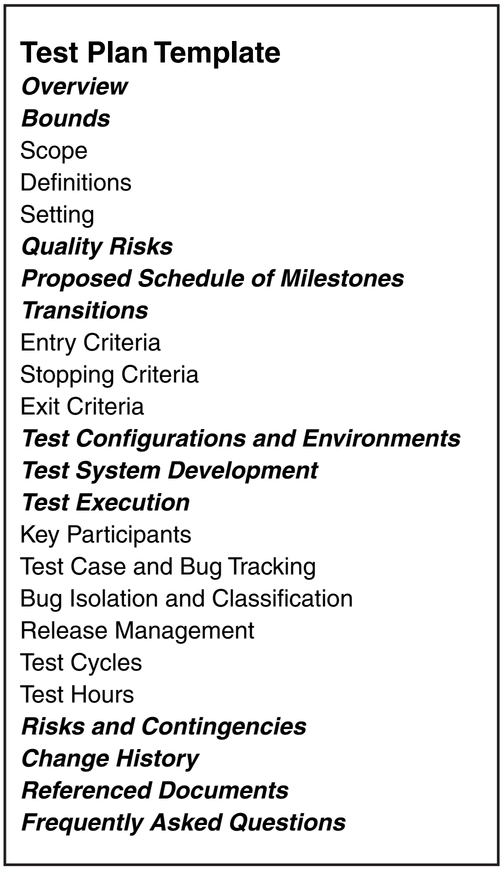
[TBD: Mary, please tell me how this should work?]

Although this might seem like "copping out," identifying and documenting open issues are among the most useful aspects of the planning exercise. Writing the plan forces me to think through the entire test effort—tools, processes, people, and technology—and to confront issues that I might otherwise miss. I then use the first few drafts of the plan as a method of bringing these issues to the attention of my peers and my managers.

That said, I do spend some time thinking about the questions, concerns, and issues I raise. If possible, rather than simply asking a question, I also include a suggested answer or a set of possible answers. A test plan that consists largely of notations about matters that are "to be determined" or issues that await resolution by someone else doesn't add a lot of value.

A Test Plan Template

The template presented in Figure 2.1 is one that I often use for developing test plans. (I have also used the IEEE 829 test plan template, which I'll discuss a little later in this chapter.) It isn't a tool for cutting and pasting a test plan in little time with little thought; rather, it is a logical set of topics that I've found I need to consider carefully for my test efforts. Feel free to add or delete topics as your needs dictate. The following sections examine the parts of the test plan template one by one.



Test Plan Template

Overview

Bounds

Scope

Definitions

Setting

Quality Risks

Proposed Schedule of Milestones

Transitions

Entry Criteria

Stopping Criteria

Exit Criteria

Test Configurations and Environments

Test System Development

Test Execution

Key Participants

Test Case and Bug Tracking

Bug Isolation and Classification

Release Management

Test Cycles

Test Hours

Risks and Contingencies

Change History

Referenced Documents

Frequently Asked Questions

Figure 2.1 A test plan template.

Overview

The overview section of a test plan allows me to introduce readers of the plan to my test project, including what's to be tested and the general test approach. What I've found is that oftentimes managers one or two levels above me don't really have a good idea of what testing buys them. In the overview, I present a concise explanation of my goals, methodologies, and objectives. Although this section should be fairly brief, it's often useful to include simple pictures or charts. You might want to illustrate concepts such as the architecture of the system under test, the decomposition or segmentation of the system for component or integration testing, or how this test effort fits into other test efforts that might precede, run concurrently, or follow.

Bounds

In this section, I set boundaries for the test plan by discussing what I will and will not test, by defining important terms and acronyms related to the testing I plan to perform, and by determining where and in what context the test efforts associated with this test subproject will take place.

Scope

Webster's Dictionary defines *scope*, in the context of a project or an operation, as the "extent of treatment, activity, or influence; [the] range of operation." When I describe the scope of my project, I am essentially demarcating what I will and will not pay attention to during the course of the project. I often use an "Is/Is Not" table to define the scope of testing, with the Is column listing the elements that are included within the scope of a particular test phase, and the Is Not column specifying elements that are not covered by this test effort. Table 2.1 shows an example of such a table, used to describe the scope of the system testing for SpeedyWriter based on the risk analysis shown in Chapter 1.

Table 2.1 Is/Is Not Format for Presenting Test Project Scope

IS	IS NOT
Functionality	File conversion
Localization (Spanish, French, and German only)	Localization (other than Spanish, French, and German)
Capacity and volume	Network compatibility
Basic file sharing	Network file sharing options
Configuration options	Security or privacy
Install, setup, initial configuration, update, and uninstall	Usability including any time and motion studies
Performance	Date handling
Windows, Unix (Solaris, Linux), and Mac compatibility and functionality	OS/2, Unix (beyond Solaris and Linux), or other platforms
Standards compliance	Structural testing
Error handling and recovery	
User interface (functional)	
Web browser compatibility	
Behavioral testing	
Beta testing by customers	

This compact form allows me to present a precise statement of scope. It's usually unnecessary to define each item at this point; the details about each aspect of testing belong in the test cases themselves.

Definitions

Testing, like other disciplines in the computer world, has its own terms and phrases. Therefore, I include a table of definitions in my test plans. Such a table can help to clarify terminology for those who are not experienced in the field of testing, and can also help to ensure that everyone on the test team is operating from the same set of definitions.

Feel free to use the glossary contained in this book as a starting point for compiling your own list. You should edit the definitions as necessary, deleting phrases that do not apply to your project. Putting a lot of extraneous verbiage in a test plan can lead to a severe case of MEGO ("my eyes glaze over") for readers.

Setting

This section of the test plan describes where I intend to perform the testing and the way those organizations doing the testing relate to the rest of the organization. The description might be as simple as "our test lab." In some cases, though, you might have testing spread hither and yon. I once managed a test project in which work took place in Taipei and Lin Kuo, Taiwan; in Salt Lake City, Utah; and in San Jose and Los Angeles, California. In cases such as this, I would present a table or diagram (such as the one shown in Figure 2.2) that shows how work will be allocated among the various participants. (For more information on managing distributed test efforts, see Chapter 10, "Involving Other Players: Distributing a Test Project.")

Quality Risks

If you followed the process discussed in Chapter 1, you already have the material you need for this section. Either you can summarize the quality risk documents you've prepared, or simply reference them in the test plan. If you suspect that many of your readers won't look at the referenced documents, it makes sense to summarize the quality risks here, given that your purpose is to communicate as well as to plan. However, if you know that people support your test planning process and will take the time to read your outline of quality risks or your *failure mode and effect analysis* (FMEA) chart, you can save yourself some work by referencing them.

I also like to cross-reference the test strategy and the test environments against the various risk categories. For example, if I know I'm going to use a particular configuration of a server and run primarily behavioral, manual tests to find potential functionality bugs (those mitigating the risks to quality in terms of functionality), then one row in my quality risks table might look as shown in Table 2.2.

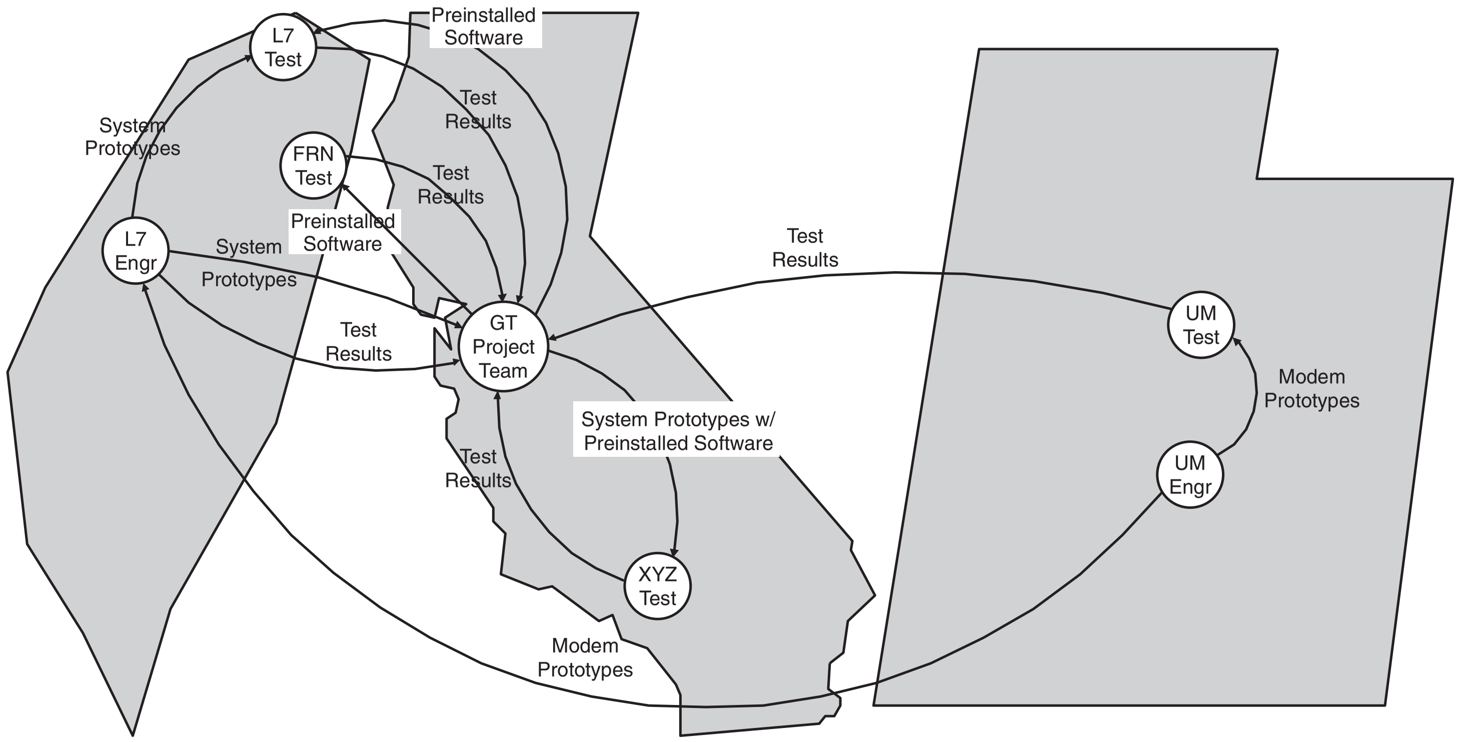


Figure 2.2 A context diagram for a distributed test effort.

Table 2.2 An Extract of a Quality Risks Table

QUALITY RISK CATEGORY	SERVER CONFIGURATION	TEST STRATEGY
Functionality	Solaris/Oracle/Apache	Manual Behavioral Primarily scripted Some exploratory

Proposed Schedule of Milestones

Most of my test plans contain a schedule for the test project's major milestones. You can extract these from the work-breakdown-structure, which I discussed in Chapter 1. I focus on the high-level milestones and deliverables that are visible to management. Table 2.3 provides an example of such a schedule.

Table 2.3 An Example Schedule from a Test Plan

MILESTONE	DATE
Test Development and Configuration	
Test plan complete	8/9/2004
Test lab defined	8/12/2004
FMEA complete	8/16/2004
Test lab configured	8/26/2004
Test suite complete	9/5/2004
Test Execution	
System test entry	9/2/2004
Cycle 1 Complete	9/16/2004
Cycle 2 Complete	10/3/2004
Cycle 3 Complete	10/13/2004
System test exit	10/13/2004

Transitions

For each test phase, the system under test must satisfy a minimal set of qualifications before the test organization can effectively and efficiently run tests. For example, it makes little sense to start extensive user-scenario testing of SpeedyWriter if the application cannot open or save a file or display text on the screen. Likewise, the DataRocket server can't undergo environmental testing—especially thermal testing—if you don't have even a prototype case. This section of the test plan should specify the criteria essential for beginning and completing various test phases (and for continuing an effective and efficient test process). I usually refer to these as *entry*, *exit*, and *continuation* criteria, respectively, but some test professionals use the terms *entry*, *stopping*, and *suspension/resumption* criteria.

As you write criteria for test phases and transitions, be aware of what you're actually saying: "If someone outside the test group fails to comply with these rules, I'm going to object to starting this phase of testing, ask to stop this phase of testing, or suggest that we not move this project forward." While these are technical criteria, invoking them can create a political firestorm. I only include criteria that I seriously believe will affect the test team's ability to provide useful services to the project. While test team efficiency is important to the project, making life convenient for the test team is not, so I'm careful to avoid criteria that could be construed as trying to shuffle make-work on to other departments. Finally, when test phase milestones—especially phase entry and exit meetings—occur, I explicitly measure the project against the criteria and report in those terms. I typically rate each criteria as "green" (totally satisfied), "yellow" (not entirely satisfied, but perhaps not a problem), or "red" (unsatisfied and creating a major problem), bringing the data I need to back up the criteria identified as "yellow" or "red."

The data to substantiate each violated criterion's status is critical, and should connect to an important business reality. What I've seen with entry, continuation, and exit criteria is that any reasonable-sounding criterion will usually pass muster in the test plan review, but, if that criterion isn't anchored in a solid business case for project delay, it will create all sorts of controversy when invoked.

Some people use hard-and-fast, bug-related exit criteria such as "zero open severity 1 bugs," "10 or fewer open severity 2 bugs," and so forth. I have found that these approaches can lead to counterproductive arguments about whether a bug is correctly classified in terms of severity, which ignores the bigger picture of quality and how quality fits into the other business issues such as schedule, budget, and features. However, in some contexts—for example, safety-critical systems, defense systems, outsource development, and so forth—using quantifiable exit criteria might be necessary.

Entry Criteria

Entry criteria spell out what must happen to allow a system to move into a particular test phase. These criteria should address questions such as:

- Are the necessary documentation, design, and requirements information available that will allow testers to operate the system and judge correct behavior?

- Is the system ready for delivery, in whatever form is appropriate for the test phase in question?[1]
- Are the supporting utilities, accessories, and prerequisites available in forms that testers can use?
- Is the system at the appropriate level of quality? Such a question usually implies that some or all of a previous test phase has been successfully completed, although it could refer to the extent to which code review issues have been handled. Passing a smoke test is another frequent measure of sufficient quality to enter a test phase.
- Is the test environment—lab, hardware, software, and system administration support—ready?

Figure 2.3 shows an example of entry criteria that might apply for SpeedyWriter.

Continuation Criteria

Continuation criteria define those conditions and situations that must prevail in the testing process to allow testing to continue effectively and efficiently. Typically, I find that the test environment must remain stable, the bug backlog manageable, the tests for the most part unblocked (e.g., by large bugs), installable and stable test releases must be delivered regularly and properly, and the change to the system under test must be known and controlled. Figure 2.4 shows an example of continuation criteria that might apply to SpeedyWriter.

Exit Criteria

Exit criteria address the issue of how to determine when testing has been completed. For example, one exit criterion might be that all the planned test cases and the regression tests have been run. Another might be that project management deems your results "OK," by whatever definition they use to decide such questions. (I'll look at some metrics that can shed light on product quality in Chapter 4, "An Exciting Career in Entomology Awaits You: A Bug Tracking Database," and Chapter 5, "Managing Test Cases: The Test Tracking Spreadsheet," along with examining some political issues associated with results reporting in Chapter 9, "The Triumph of Politics: Organizational Challenges for Test Managers.") In the case of System Test exit criteria—provided System Test is the last test phase on your project—these exit criteria often become the criteria by which the customer-ship or deployment decision is made. An example of a business-driven set of System Test exit criteria for SpeedyWriter is shown in Figure 2.5.

[1]In the component test phase (assuming that the test organization is involved at that point), I usually accept whatever development is ready to provide as long as it includes sufficient scaffolding, or harnesses to run my tests. Once I reach the system test phase, however, I ask for customer packaging, especially in the case of software, whose installation process has a significant impact on whether the system works at all.

System Test can begin when:

1. Bug tracking and test tracking systems are in place.
2. All components are under formal, automated configuration and release management control.
3. The Operations team has configured the System Test server environment, including all target hardware components and subsystems. The Test Team has been provided with appropriate access to these systems.
4. The Development Teams have completed all features and bug fixes scheduled for release.
5. The Development Teams have unit-tested all features and bug fixes scheduled for release.
6. Less than 50 must-fix bugs (per Sales, Marketing, and Customer Service) are open against the first test release slated. (50 being the number of bug reports that can be effectively reviewed in a one-hour bug triage meeting.)
7. The Development Teams provide software to the Test Team three business days prior to starting System Test.
8. The Test Team completes a three-day “smoke test” and reports on the results to the System Test Phase Entry meeting.
9. The Project Management Team agrees in a System Test Phase Entry Meeting to proceed. The following topics will be resolved in the meeting:
   - Whether code is complete.
   - Whether unit-testing is complete.
   - Assign a target fix date for any known “must-fix” bugs (no later than one week after System Test Phase Entry).

Figure 2.3 Entry criteria for SpeedyWriter.

System Test can continue if:

1. All software released to the Test Team is accompanied by Release Notes.
2. No change is made to the system, whether in source code, configuration files, or other setup instructions or processes, without an accompanying bug report. Should a change be made without a bug report, the Test Manager will open an urgent bug report requesting information and escalate to his manager.
3. The open bug backlog (“quality gap”) remains less than 50. The average time to close a bug remains less than 14 days.
4. Twice-weekly bug review meetings (under the Change Control Board) occur until System Test Phase Exit to manage the open bug backlog and bug closure times.

Figure 2.4 Continuation criteria for SpeedyWriter.

System Test will end when:

1. No changes (design/code/features), except to address System Test defects, occurred in the prior three weeks.
2. No panic, crash, halt, wedge, unexpected process termination, or other stoppage of processing has occurred on any server software or hardware for the previous three weeks.
3. No client systems have become inoperable due to a failed update during System Test.
4. The Test Team has executed all the planned tests against the GA-candidate software.
5. The Development Teams have resolved all "must-fix" bugs per Sales, Marketing, and Customer Service.
6. The Test Team has checked that all issues in the bug tracking system are either closed or deferred, and, where appropriate, verified by regression and confirmation testing.
7. The test metrics indicate that we have achieved product stability and reliability, that we have completed all planned tests, and the planned tests adequately cover the critical quality risks.
8. The Project Management Team agrees that the product, as defined during the final cycle of System Test, will satisfy the customer's reasonable expectations of quality.
9. The Project Management Team holds a System Test Phase Exit Meeting and agrees that we have completed System Test.

Figure 2.5 SpeedyWriter System Test exit criteria.

Test Configurations and Environments

This section of the test plan is where I document which hardware, software, networks, and lab space I will use to perform the testing. For these various test systems, I'll describe whatever important configuration details bear mentioning as well. For a PC application or utility, this task can be as simple as listing the half-dozen or so test PCs, the two or three test networks (assuming that networking is even an issue), and the printers, modems, terminal adapters, and other accessories you might require from time to time. For commercial software or hardware, I find it useful to have competitors' systems available in my test lab as reference platforms. For example, when testing an Internet appliance (which provided Web and email features only), we had PCs configured with the various browsers and email clients so we could answer the question, "What happens when we run the same test using X (browser or email client)?" when confronted by potentially buggy behavior on the appliance.

Suppose, however, that you are testing a system with significant custom hardware elements (such as a new laptop or a server), one with many hardware elements (such as a network operating system or a network application), or one with expensive hardware elements (such as a mainframe, a high-availability server, or a server cluster). In these complex cases, using a simple table or a spreadsheet might not be sufficient. In Chapter 6, "Tips and Tools for Crunch Time: Managing the Dynamic," I'll introduce a database that can help you stay on top of complicated situations such as these. In Chapter 7, "Stocking and Managing a Test Lab," I'll show how to extend this database to include managing lab space and equipment. This database also models human resources and network needs. You can include the reports produced by this database in this section of the test plan.

When custom hardware is involved, you can present a scheme for hardware allocation in this portion of the test plan or in a separate document. Whatever the location, I've found it extremely important to prepare this allocation plan. Failing to establish a detailed plan for allocating hardware is tantamount to assuming that the hardware I need will magically make itself available, properly configured and ready for testing, at the very moment I need it. If you lead a charmed life, such things probably happen to you all the time, but they never happen to me. I always worry about hardware allocation, and work to have a hardware allocation plan in place around the time I'm finishing the test plan.

What goes into a test hardware allocation plan? I usually list the test purpose or use, the systems needed (including the quantities and revision levels), the infrastructure, the time period, the location, and any other hardware necessary for a particular test. Table 2.4 shows a prototype allocation plan for DataRocket's integration test and system test phases.

Test Development

In some cases, I find my test teams rerunning tests that were created in previous test efforts. In other cases, I've used a purely exploratory approach where I created the test data during testing and followed my muse in terms of procedures and specific test steps. Typically, though, my test projects include some amount of work to design and develop various test objects such as test cases, test tools, test procedures, test suites, automated test scripts, and so forth. Collectively, I refer to these objects as *test systems*.

In such cases, in this section, I'll describe how my test team will create each of those objects. (I'll look at some particulars of testware development in Chapter 3.) If we're going to use manual testing, then I'll let the readers know if we intend to write detailed test cases or use test charters.[2] If I need test data, then I'll explain how we're getting that data and why we picked those approaches. If we're doing test automation using existing (commercial or freeware) test tools, then I'll describe why we're using the particular tools we've chosen and how we intend to develop test scripts. If we're creating custom test tools or utilities, then I'll describe what those utilities are and how we intend to use them.

At some point, test system or testware development can become a software development project in its own right. I've worked on custom tool projects for clients where we created completely free-standing test automation and management tools. Some of my test projects have included test development efforts that involved person-decades of work. In such cases, my preference would be to have a separate plan that describes that development effort. Various good templates are available for software development plans.[3]

[2]Test charters are something I learned from James Bach in the context of his approach for exploratory testing. See his material on exploratory testing at www.satisfice.com.
[3]For example, see www.construx.com for templates drawn on Steve McConnell's *Software Project Survival Guide*.

Table 2.4 A DataRocket Hardware Allocation Plan

TEST USAGE	SYSTEM [QTY]	NETWORK	WHEN	WHERE	OTHER[QTY]
Integration Test Phase					
Component interfaces/ signal quality	Engineering prototype [2]	Novell NetWare, Network File System, Microsoft Windows NT	9/15–10/15	Engr lab	MS mouse, MS kbd, VGA mon, USB mouse, USB mon, USB kbd, 3COM LAN, USR mdm, Epson prn, Quantum HD, oscilloscope
Mechanical life	Engineering prototype [2]	None	8/1–10/1	Engr lab	None
Stress, capacity, volume	Engineering prototype [1]	Novell, NFS, NT	9/15–10/15	Test lab	MS mouse, VGA mon
Performance	Engineering prototype [1]	Novell, NFS, NT	9/15–10/15	Test lab	MS mouse, MS kbd, VGA mon, Quantum HD, IBM HD
System Test Phase					
MTBF demonstration	Validation prototype [4]	Novell	10/17–1/17	Engr lab	MS mouse, MS kbd, VGA mon, MUX
Functionality	Validation prototype [2]	Novell, NFS, NT	10/17–12/1	Test lab	MS mouse, MS kbd, VGA mon, USB mouse, USB mon, USB kbd, USR mdm, Epson prn, ISDN T. adptr
Stress, capacity, volume	Validation prototype [1]	Novell, NFS, NT	10/17–12/1	Test lab	MS mouse, VGA mon
Performance	Validation prototype [1]	Novell, NFS, NT	10/17–12/1	Test lab	MS mouse, MS kbd, VGA mon, Quantum HD, IBM HD
Compatibility	Validation prototype [3]	N/A	10/24–12/1	System Cookers, Inc.	MS mouse [3], MS kbd [3], VGA mon [3]
Environmental	Validation prototype [2]	N/A	10/24–12/1	System Cookers, Inc.	MS mouse [2], MS kbd [2], VGA mon [2]

Test Execution

This portion of the test plan addresses important factors affecting test execution. For example, in order to run tests, you often need to receive items from the outside world, primarily resources (or funding for those resources) and systems to test. In the course of running tests, you will gather data that you must track in a way presentable to your team, your peers, and your managers. In addition, you will run through distinct test cycles in each test phase. I find that the level of detail required here varies from team to team and project to project. With my more senior test teams on well-run projects, I can leave much of this section to the discretion of my crack team of testers. With junior testers, especially on chaotic projects, the more I can nail down the test execution realities during the planning phase, the more confusion I can anticipate and resolve ahead of time.

Key Participants

In this section, I identify the key participants in the test effort and the role they'll play in testing. I find it especially important to identify the external participants, the handoff points, and each participant's contact information. Another useful portion of this subsection can be the escalation process; in other words, if some key participants do not or cannot fulfill their agreed-upon role, then what happens next? In the case of external participants, I work out the roles and hand-off points with the appropriate peer-level managers first, before putting that information in the test plan. Surprising fellow development-team managers with new and urgent assignments for their staff is unlikely to win any popularity points for the test manager who tries it.

Test Case and Bug Tracking

This section deals with the systems used to manage and track test cases and bugs. Test case tracking refers to the spreadsheet or database I use to manage all the test cases in the test suites and how I track progress through that listing. (If I don't track the tests I plan to run, how can I gauge my test coverage later on?) Bug tracking has to do with the process my team uses to manage the bugs we find in the course of testing. Since these systems form your principal communication channels inward to your own team, outward to other teams such as development, and upward to your management, you should define them well here. Chapters 4 and 5 deal with these topics in more detail. Even if you choose not to use the approaches described there, you might find some ideas in those chapters that will help you complete this section of the plan.

Bug Isolation and Classification

This section of the test plan is where I explain the degree to which I intend to isolate bugs and to classify bug reports. Isolating a bug means to experiment with the system under test in an effort to find connected variables, causal or otherwise. I find it's important to be explicit about bug isolation; otherwise, the test organization can end up involved in debugging, a developer task that can consume lots of my testers' time with very little to show for it in terms of test coverage (see Chapter 4.)

Classifying a bug report assigns the underlying bug to a particular category that indicates how the bug should be communicated and handled. For example, I've used classifications such as the following:

Requirements failure. The bug report concerns a failure of the system to meet its requirements. The appropriate party will resolve the problem.

Nonrequirements failure. The bug reported is not covered by the system requirements, but it significantly affects the quality of the system in unacceptable ways. The appropriate party will resolve the problem.

Waiver requested. The bug report does indeed describe a failure, but the developers request a waiver because they believe that it will not significantly affect the customers' and users' experiences of quality.

External failure. The bug report addresses a failure that arises from a factor or factors external to or beyond the control of the system under test.

Test failure. The developers believe that the test has returned a spurious or invalid error.

Rather than classifying bugs, some project teams use a single bug management process. Many successful development projects I've worked on used a bug triage process (sometimes as part of the change control board process) to assign bug priority and determine which bugs must be fixed prior to release. In that case, you might want to describe that process here.

Test Release Management

One of the major interfaces between the overall project and testing occurs when new revisions, builds, and components are submitted to the test team for testing. In the absence of a predefined plan for this, I have seen this essential hand-off point degrade into absolute chaos. On one project, a software development manager was emailing a new version of the software out to the entire project team, including the test team, every time any member of his team of programmers fixed a bug. We received a dozen test releases a day! I find that kind of anarchy unacceptable on my test projects, so in this section of the test plan I define the key elements of the test release process that can affect my effort. To revisit a figure from the previous chapter, in the ideal case, I try to create a test release process that looks as shown in Figure 2.6.

One element of that process is regular, predictable timing. How often will you accept a new test release into the testing process? My preference is usually once a week. On the one hand, once-a-week test releases give me ample time to complete plenty of scheduled testing and even do some exploratory testing as part of each test cycle, rather than just doing confirmation testing of the bug fixes in that test release. Test releases that show up once a day—or more frequently—leave my test teams little time for anything but confirmation testing the fixes. On the other hand, test releases that come every other week or even less frequently can introduce major regressions and other changes in behavior that I'd like to know about sooner. Therefore, the weekly test release strikes me as a reasonable balance in most contexts I've worked in, although your situation may differ.

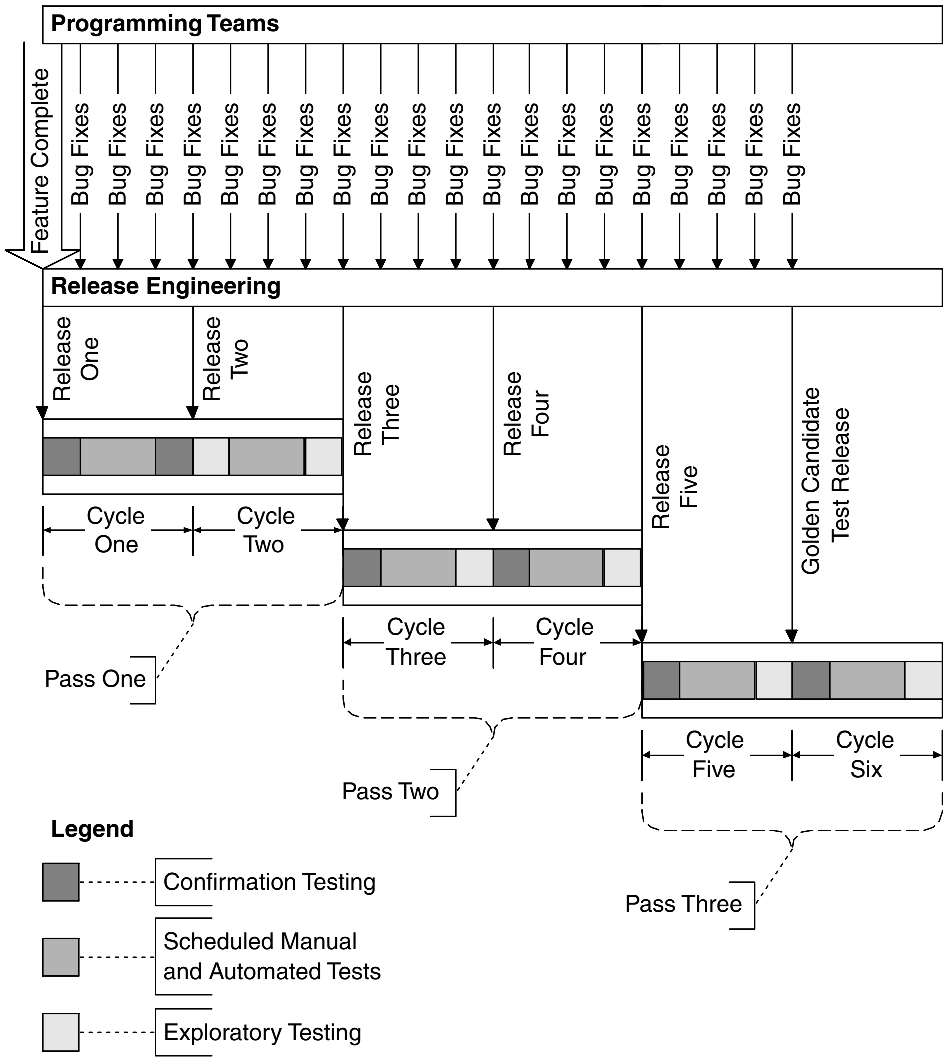


Figure 2.6 Test passes, test releases, and test cycles.

Every new release of a software or hardware component into the test lab should have a release (revision) number or identifier attached. This identifier is essential for determining which version of the system contains a bug, which version fixes that bug, which pieces are compatible with other pieces, and which versions you have tested.

You should also get release notes with the release. These release notes should specify which bugs were fixed, what changes were made, how the changes will affect system behavior, and so forth. This is especially true for complex systems that have to interface with other systems. For example, if an application accesses a database, you might need to change the schema of the database to accommodate changes in the program. This can be a real challenge if this database is shared across multiple systems, because plans must be in place to handle the ripple effects.

In addition, I need to be able to count on receiving new releases in a certain format. I specify for each test phase—and therefore in each test plan—a specific process and format for delivering new releases. For example, for software delivered during component and integration test phases, this format might be as simple as a tar or zip archive sent via email or posted to the network. Once I enter system testing, however, I usually want the software releases to arrive in the same format and with the same installation process as the initial customer release. I consider testing the installation process a key part of most System Test efforts, so I cover that here.

I also find I need a defined uninstall process for many projects. The reason is twofold. First, if an uninstall process exists and is accessible to users and/or customers, then I again need to test this capability. Second, if a test release comes along that, after installation, proves so unstable or nonfunctional that further testing is pointless, then reverting to a "known testable" release is the only way to return to making some type of forward progress. In the case of a simple application, an uninstall utility might be included, but complex systems—especially those that have shared databases—can have very complex installation and uninstallation processes.

In some cases, the proposed approach for accelerating a test process blocked or impeded by a bad test release is to install a new release. Therefore, you'll need to consider (for both software and hardware) whether you will accept new revisions in the middle of a test cycle. The key issues here are regression testing and the implications of the revision for the validity of your previous test cycle results. Will you have to do a complete reset and start the test cycle over if a driver, a configuration file, or an application build shows up midway? These unexpected arrivals can cause real problems, especially toward the end of the system test phase, if you are receiving new releases every few days or even every few hours. Without a completely automated regression test system that can repeat every test flawlessly in a day or so, you will never be able to eliminate the possibility that the new release has introduced a major new bug. (For a discussion of coverage problems and regression testing, see Chapter 3.)

This problem occurs even in hardware development. It is true that motherboards, cases, and so forth have long lead times and that engineering prototypes cost a lot to produce. Nevertheless, you might still receive the BIOS du jour or the daily "gold master" hard drive. My worst experiences with what I call "churn and burn" have occurred on laptop development projects. Because of the manual nature of PC testing, you will probably have time for little more than confirmation testing on new releases if they come daily, leading to dismal test escapes. (I'll talk more about test escapes in Chapter 3.)

These chaotic releases have affected the logistics and operation of a number of my test projects. Some amount of effort is required for flashing a BIOS, installing applications, or replacing a motherboard, especially when my lab contains half a dozen test platforms. If the process requires specialized skills, such as system administration, network administration, or database administration, I need to have the right person available to handle the job. Moreover, the system will be unavailable for test execution throughout the upgrade. Such abrupt, unexpected changes can also impose communication overhead and confuse testers. I have to inform everyone that a new release has dropped in, circulate the release notes (assuming that they exist), give my staff time to study them, and hope that everyone can keep straight just what the latest chunk of software is supposed to do.

As you can tell, I don't like midstream releases. However, it's also true that instituting significant, project-wide process changes from within the test team through the mechanism of the test plan can be difficult.[4] If your company's development "process" includes a system test phase comprised of 18-hour days, with a spinning disk containing the allegedly final test release landing on your desk hourly, you can't transform that by fiat in the test plan. However, you can—and I certainly would—attempt to convince people there is a better way, and, if I succeeded, I'd capture that in this section of the test plan.

Test Cycles

In Chapter 1, I used the phrase "test cycle" rather cavalierly, but perhaps a more formal definition is in order. By a test cycle, I mean running one, some, or all of the test suites planned for a given test phase as part of that phase. Test cycles are usually associated with a single test release of the system under test, such as a build of software or a motherboard. Generally, new test releases occur during a test phase, triggering another test cycle. For example, if test suites 3.1 through 3.5 are planned for a three-cycle system test phase, the first cycle could entail executing 3.1 and 3.2; the second cycle, 3.3, 3.4, and 3.5; and the third cycle, 3.1, 3.2, 3.3, 3.4, and 3.5.

Any given test phase involves at least one cycle through the test suites planned for that phase. (As I mentioned in Chapter 1, it's a challenge to estimate how many cycles will be needed in any given phase.) Each subsequent cycle generally involves a new release of one or more components in the system. This section of the test plan should spell out your specific assumptions and estimates about the number, timing, and arrangement of test cycles. A picture like Figure 2.6 can help.

Test Hours

In some cases, I find I need to define the specific hours of testing. In addition, on some projects I use multiple shifts to keep scarce resources humming 16 or 24 hours a day to accelerate the testing. In such cases, I'll define in this section the specific hours and shifts that will be used.

[4]Aesop's fables include the apt story about belling the cat. The mice are tired of scurrying about in fear of the cat, and they convene a council to decide how to deal with the problem. One suggests that they put a bell on the cat's collar so that they can hear him coming. All agree this is a capital idea, until one young whippersnapper asks, "But who will bell the cat?" When I'm tempted to push for a change in "the way things are done around here," I recall this tale and often think better of the effort.

Risks and Contingencies

I sometimes use the title "Open Issues" for this section. In it, I address potential or likely events that could make the test plan difficult or impossible to carry out. Topics might include training needs, the availability of additional development support for debugging if an exceptional number of bugs are found, and so forth. Alternatively, you can include this type of information when you discuss continuation criteria.

Strictly speaking, most advocates of good development processes encourage a global approach to risk management.[5] If you work on a project in which the entire team has a single risk-management plan, you might be able to omit this section by including these concerns in that plan.

Change History

This part of the document records the changes and revisions that have been made to the test plan itself to this point. Specifically, you can assign a revision number and record who made the changes, what those changes were, and when the revision was released.

Referenced Documents

As a rule, a test plan refers to other documents such as design specifications, requirements, the test suites, any quality-risk analysis documents, and other pertinent information. Listing these documents in this section lets me avoid extensive repetition of their contents (which can create complications when these documents change).

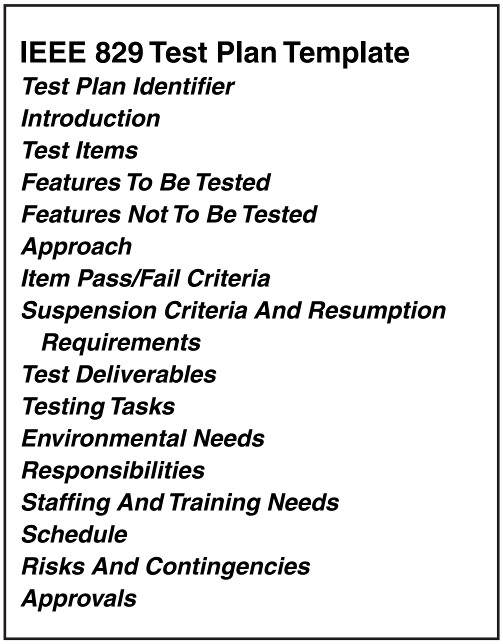
Frequently Asked Questions

On projects where I use neophyte test engineers and test technicians, I find that a frequently asked questions section is useful. Many of these questions entail describing the importance of the escalation process.

The IEEE 829 Template: Compare and Contrast

The test plan template presented previously complies with the IEEE 829 Standard in that it contains all the same information—at least as I use the template—but presents it in a way that I consider to be inherently lighter-weight; in other words, smaller. Let me describe each field in the template briefly and how the information in my template maps to the IEEE 829 template, shown in Figure 2.7.

[5]See, for example, McConnell's *Software Project Survival Guide*.



IEEE 829 Test Plan Template

Test Plan Identifier
Introduction
Test Items
Features To Be Tested
Features Not To Be Tested
Approach
Item Pass/Fail Criteria
Suspension Criteria And Resumption Requirements
Test Deliverables
Testing Tasks
Environmental Needs
Responsibilities
Staffing And Training Needs
Schedule
Risks And Contingencies
Approvals

Figure 2.7 IEEE 829 Standard test plan template.

The Test Plan Identifier is a just a unique name or document ID. I don't explicitly include this in my template, but rather follow whatever document naming and storage conventions are used by the project.

The Introduction is used in IEEE 829 test plans the same way I use the Overview section in my template.

The Test Items, Features to Be Tested, and Features Not to Be Tested cover the issue of scope. In most of my test plans, the implicit scope is the entire system, and testing will focus on the high-risk areas of that system, which are detailed in the Scope and Quality Risks sections. As discussed previously and as I'll show in the case study at the end of this chapter, my approach is to use a bullet item and tabular style of listing this information. The IEEE format is more narrative and more explicit.

The Approach section of the IEEE plan is where the tester presents the test strategies she'll use for specific tests. In my template, I present this information in the Quality Risks section.

The Item Pass/Fail Criteria and the Suspension Criteria and Resumption Requirements sections map back into the Entry Criteria and Exit Criteria, and Continuation Criteria sections, respectively. For lower-level testing, like component testing, the idea of defining discrete pass/fail criteria for test items in the test plan makes some sense, but I've found that all I can present is aggregate information during system testing.

To me, the Test Deliverables are bug reports and test case findings, along with aggregate-level metrics. (I'll examine bug and test tracking and metrics in Chapters 4 and 5.) In some cases, other deliverables might exist. The IEEE plan makes this discussion explicit, where I discuss deliverables in the various subsections of the Test Execution section. If deliverables other than bug reports, test results, and related metrics were required, I would define them in additional subsections of that section.

The Test Tasks section presents some of the same kind of information I present in Test Cycles; in other words, the procedures and activities that testers will need to undertake to make test execution happen. However, the IEEE plan can also deal with test development and environment configuration issues that I would discuss in the Test Development section.

The Environmental Needs section maps directly into the Test Configurations and Environments section in my template.

The Responsibilities and Staffing and Training Needs section correspond to my Key Participants subsection. I don't tend to discuss training within the context of my test plans, since I prefer to deal with the issue of training as a team-wide effort driven more by the long-term needs of the test organization rather than a reactive, project driven measure, but your approach might differ.

Both the Schedule and the Risks and Contingencies sections map directly into the corresponding sections in my template.

Finally, the Approval section in the IEEE template is where the key stakeholders in the test effort—and presumably the project as a whole—sign to indicate approval of the test plan. I'll talk more about my approach to getting approval in the next few pages, but suffice it to say that formal sign-offs is an approach I use infrequently.

Selling the Plan

After finishing a test plan, I want to obtain approval from project management and development peers. I have used the approach of attaching a formal sign-off sheet to the plan, a la the IEEE 829 template. The approach I usually prefer, though, is that of holding a review meeting with all the involved managers.

If your company tends to become mired in politics, a sign-off might be necessary to protect yourself. If such formalism is rare in your company, there are other ways to document approval, such as a circulation list that you keep. In any case, I recommend a tactful approach if you pursue a formal sign-off. Goose-stepping around the office with a document, requiring management signatures before you proceed with the test project, is not likely to endear you to your colleagues.[6]

I am often tempted to send the test plan via email, demanding that the recipients "speak now or forever hold your peace." After failing to receive any criticisms or concerns, you might assume that everyone has read your plan and agrees with it. This assumption is almost universally false. Lack of response usually means that no one has read it.

[6]You might want to consult the discussion on test plan approval in *Testing Computer Software*, by Cem Kaner, Jack Falk, and Hung Quoc Nguyen. These authors seem to have had positive experiences using a sign-off approach.

I find that review meetings are the best way to get people to read plans. One benefit of writing and circulating the plan is to provide a forum for discussing the test part of the project. What better way to achieve that than assembling the appropriate parties to hash out the plan?

Before I hold a review, I email the test plan with a note mentioning that we'll hold a meeting to review it in the next week. (Since email attachments are notoriously flaky, I offer to print a hard copy for anyone who can't open the file.) I invite every manager who is directly affected by the plan—usually the development manager, the project manager, the build/release manager, the technical support manager, the sales and marketing managers, the business analyst or other in-house domain experts, and my own manager—as well as the lead test engineer on the project. I send courtesy invitations to others who might have an interest, but I try to limit the total list to 10 or fewer. I then schedule the review.

At the meeting, I provide extra hard copies of the plan for forgetful participants. As the test manager, I lead the group through the plan section by section or page by page. If anyone has concerns, we discuss—and try to resolve—them on the spot. My goal is to leave the meeting with a marked-up copy of the test plan that will enable me to produce a "final" version. Shortly after the review, I make the requested changes to the document and recirculate it, this time marked as "Released for Execution" or "Release 1.0," to flag the fact that we intend to proceed according to this plan.

Clarity, Pertinence, and Action

One of my clients refers to long test documents as "shelfware." He has seen too many test managers and senior test engineers spend too much of their time filling enormous binders that then sit on shelves, untouched during test execution. My client's comments are a cautionary tale: veer off into a morass of documentation for its own sake—just to follow a standard or to "fill in the blanks"—and you can lose focus, relevance, and credibility.

The style of writing matters, too. Avoid passive verbs. Don't say that something "is to be done"; instead, say exactly who is to do what and when. Defining roles and responsibilities keeps you out of the trap I saw one client's test manager fall into when she wrote a test plan that was just a collection of statements of principle and pieties. ("Testing is good," "Bugs are bad," etc.) Keep jargon, buzzwords, and so forth limited to those appropriate for the audience, and use a Definitions section to promote further clarity.

I keep my test documents practical, focused, and short, and they work well for me. This approach to planning, customized to fit your particular needs, will help you write effective test plans that you can really use.

Bonus Test Planning Templates

A colleague of mine, Johanna Rothman, president of Rothman Consulting Group, has shared two of her own testing templates with me. They are included here by permission. Rothman tells me that she has her clients work through the strategy first (see Figure 2.8),

System Test Strategy

1. Product, Revision and Overview
   Describe the product and revision designator.
   Describe briefly how the product works. Reference other material as appropriate.
2. Product History
   Include a short history of previous revisions of this product. (3-4 sentences).
   Include defect history.
3. Features to be tested
   List all features to be tested. Organize the list in the way that makes most sense—user features, or by level:
   Application
   Demo software
   Client substrate
   Server
   Network (this may be more layers)
4. Features not to be tested
   Describe any features not to be tested.
5. Configurations to be tested and excluded
   I recommend a table showing which hardware configurations will be tested with which software.
6. Environmental requirements
   Enumerate hardware, firmware, software, networks, etc. required to carry out the testing.
7. System test methodology
   Brief description of work items to be performed from the beginning to the end of the product development.
8. Initial Test requirements
   Test strategy (this document), written by test personnel, reviewed by product team, agreed to by project manager.
9. System test entry and exit criteria

9.1 Entry Criteria
   The software must meet the criteria below before the product can start system test. Specifically enumerate any project-specific departures from the Generic Criteria. This list must be negotiated with and agreed upon by the project leader.

Figure 2.8 Rothman's test strategy template.

and then proceed to planning (see Figure 2.9). In the case where there is more than one test subproject—for example, hardware and software subsystems that require testing before overall system testing—then there will be a single strategy document that drives the general testing strategy, with a test plan for each test subproject.

Which to pick, Rothman's or mine? That's a contextual question. Based on format, I think Rothman's templates are more IEEE-like. Perhaps you don't like the IEEE template but want to use another template that is clearly compliant. You can map between the sections of Rothman's template and the IEEE template more directly than with mine. In addition, her process of having a single strategy document and a separate test plan for each test subproject is IEEE-compliant. (The two sets of documentation are called a

Generic criteria:

1. All basic functionality must work.
2. All unit tests run without error.
3. The code is frozen and contains complete functionality.
4. The source code is checked into the CMS.
5. All code compiles and builds on the appropriate platforms.
6. All known problems posted to the bug-tracking system.

9.1 Exit Criteria
The software must meet the criteria below before the product can exit from system test. Specifically enumerate any project-specific departures from the Generic Criteria. This list must be negotiated with and agreed upon by the project leader.

Generic criteria:

1. All system tests executed (not passed, just executed).
2. Successful execution of any "Getting Started" sequence.
3. Results of executed tests must be discussed with product management team.
4. Successful generation of executable images for all appropriate platforms.
5. Code is completely frozen.
6. Documentation review is complete.
7. There are 0 showstopper bugs.
8. There are fewer than <x> major bugs, and <y> minor bugs.
9. Test Deliverables
   - *Automated tests in <framework>*
   - *Test Strategy and SQA project plan*
   - *Test procedure*
   - *Test logs*
   - *Bug-tracking system report of all issues raised during SQA process*
   - *Test Coverage measurement*
10. References
    Other documents referring the project or testing.

Figure 2.8 *(Continued)*

"Master Test Plan" and a "Level Test Plan" in Software Quality Engineering's IEEE 829-based testing course, "Systematic Software Testing.") Rothman's templates call for explicit content in terms of test deliverables, which might be helpful, especially if it's unclear how information and results will be transmitted to the rest of the project team.

My template, however, includes a section for quality risks and how those tie back to test strategies. In other words, if you are going to use the quality risk management concepts I discussed in Chapter 1, then my template specifically accommodates that. I also have explicit sections for details about the test execution processes and the organizational context of the test team.

Of course, there is no reason why you can't blend Rothman's template with mine and with the IEEE's. Just make sure that you resolve any redundancy issues; in other words, if you include sections that might possibly overlap, either merge them or change the headings and content of the sections to be clearly distinct. You can build

System Test Plan

1. Product purpose
   Briefly describe the product, why you want to produce it, what benefits accrue to the company, etc. (3-4 sentences)
2. History
   Include a short history of previous revisions of this product. (3-4 sentences)
3. Technical Requirements
   If you have a requirements document, reference it. Otherwise, include the bulleted list of functionality from the project plan. Be as specific as possible (this should be more in depth than the project plan.), Include features, performance, installation requirements.
4. System test approach
   Describe how much manual and automated testing you want to accomplish, and how you expect to use personnel (permanent, temporary, sqa, devo, etc.) Highlight issues from below that you think are problems!
5. Entry and Exit Criteria
   (If you have a small project, you can reference the entry and exit criteria from the system test strategy. If you have a large project of subprojects, create entry and exit criteria for each subproject.)
   Determine the objective criteria by which the software is known to be ready for entry into system test and exit from system test.
   For example, based on recent conversations, you already have entry criteria of:

1. Basic functionality must work.
2. All unit tests run without error.
3. The code is frozen and contains complete functionality.

Possible exit criteria:

1. All system tests execute (not pass, just execute).
2. Zero open priority 1 bugs.
3. Fewer than 20 priority 2 bugs.
4. Usability testing complete.

Include configuration issues for these next 6 items.

6. Features to be tested
7. Features not to be tested
8. Performance to be tested
9. Performance not to be tested
10. Installation to be tested
11. Installation not to be tested
12. Overall system test schedule

Milestone	*Expected date*
design test procedures for <feature a>	
review test procedures for <feature a>	
verify tests for <feature a> work	
install <feature a> tests in automated framework	

Figure 2.9 Rothman's system test plan template.

your own template out of all three, provided you take care to avoid having your template turn into a hodgepodge of ill-fitting sections.

Case Study

On one project, my test team and I tested a browser-based program that provided home equity loan processing capabilities to call center banking agents. The project was somewhat quirky, since we executed tests developed by another team and didn't have a chance to do a formal risk analysis. However, this plan gives an example of how I applied the template shown earlier to a large in-house IT project. The document, "Case Study Loan Processing Test Plan.doc," is available at www.rexblackconsulting.com.

I thank my client on this project, a major mid-western financial institution, and the executive in charge of development for allowing me to share this example with readers of this book. I have purged all identifying and otherwise private information from this document. My client contact has asked that the organization remain anonymous.

Exercises

1. You are responsible for the System Test phase for a calculator program, as discussed in Chapter 1.
   - Assume that you are working in the same location (e.g., same cubicle or adjacent tables) as the sole programmer working on the calculator. How would you document the entry criteria, bug reporting, and test release process?
   - Assume that the development team is in another city. What are the entry criteria, bug reporting, and test release implications?
2. Based on the risk analysis, schedule, and budget you prepared in exercises 3 and 4 in Chapter 1, write a test plan following my template.
3. Translate the test plan you wrote in exercise 2 into the IEEE 829 format. Was there any particular information in one that was not accounted for in another? Which one do you prefer? Justify your answers.