Test-Driven Development

What part of the system do you test first? The Proactive Testing Model helps you prioritize, manage a project-level plan and steer clear of risks. Part 4 of 4. BY ROBIN F. GOLDSMITH AND DOROTHY GRAHAM

AS DO DEVELOPERS, MOST TESTERS RECOGnize that models are valuable aids; however, testing doesn't often get the attention cast on other activities in the development lifecycle. The V Model is probably the best-known testing model, but many testers are unfamiliar with it. And the V Model isn't free from criticism: One of the more vocal critics of the V Model is Brian Marick, author of *The Craft of Software Testing* (Prentice Hall, 1995). In part 2 of this series, "V or X, This or That" (Aug. 2002), we described the "X Model," covering points that Marick felt should be present in a suitable testing model. Part 1, "The Forgotten Phase" (July 2002), discussed the V Model itself.

In Part 3, "Proactive Testing" (Sept. 2002), we introduced the Proactive Testing Model, which we believe incorporates the strengths and addresses the weaknesses of both V and X Models. In this final installment, we delineate the benefits of letting testing drive development.

Proactive Test Planning

The diagram "A Sophisticated Standard" (see page 53) depicts the oft-followed test-planning structure suggested by IEEE standard 829-1998. Some critics view the standard as a counterproductive paper-generator. While such practices are common, they need not be. The graphic overview adds value to the text-only standard, making it easier to understand and apply.

The standard suggests conceptualizing test plans at several different levels, either as individual documents or sections within a larger document. The master, overall system test plan is a management document that ultimately becomes part of the project plan.

We use a simple heuristic to drive down to lower and lower levels of test-planning detail: "What must we demonstrate to be confident that it works?"

The master test plan identifies a set of test plans that, when considered together, demonstrate that the system as a whole works properly. Typically, a detailed test plan describes each unit, integration, special (a catch-all term for tests that aren't specifically driven by the application, such as stress, security and usability tests) and system test. In turn, each detailed test plan identifies a set of features and functions that, in concert, demonstrate that the unit, integration, special function or system is working properly. For each feature and function, a test design specification describes how to demonstrate that the feature or function works properly. Each test design specification identifies the set of test cases that together indicate that the feature or function works.

The test-planning structure provides a number of benefits. While it's easy to see why many testers think that the IEEE standard requires voluminous documentation for its own sake, that approach provides no value, and we don't endorse it. However, we don't reject written test plans out of hand, as some X Model advocates seem to do. Instead, we suggest recording important test-planning information as a memory aid and to facilitate sharing, reuse and continual improvement.

For many people, test plans are primarily a collection of test cases—a collection that can grow quite large and unmanageable. One can see from the diagram, though, that the standard's structure provides immediate value by helping to organize and manage the test cases.

We can proactively define reusable test design specifications, identifying how to test common situations. These specifications can prevent enormous amounts of duplicated effort, enabling us to start credible testing with little delay. Similarly, the structure helps us define reusable test cases and selectively allocate resources. Moreover, the structure and test design specifications make it easier to reliably recreate test cases when necessary.

Proactive Prioritization

While risk prioritization should be a part of any test approach, neither the V nor X Models makes it explicit. Moreover, the proactive risk analysis methods we use are far more effective than the traditional ones we've seen applied elsewhere.

***Robin F. Goldsmith** is president and cofounder of Go Pro Management Inc., in Needham, Mass. Reach him at www.gopromanagement.com.*

***Dorothy Graham** is the founder of Grove Consultants in the UK. In 1999, she was awarded the IBM European Excellence Award in Software Testing.*

The proactive method improves testing in two ways. First, traditional approaches tend to assign risk to each test as it's identified. With nothing to compare to, each test tends to be denoted as high risk. The proactive test-planning structure, however, quickly reveals the available options, so that we can prioritize with respect to all our choices. Second, we can eliminate the common reactive technique of rating the risks of the tests that have been defined. Obviously, no risks are assigned to tests that have been overlooked, and overlooked tests are often the biggest risks. In contrast, proactive risk analysis enables us to identify critical risks that the reactive approach overlooks—and *then* to define tests to ensure that these risks don't occur.

While we use proactive test planning at each level, the most important task involves identifying and prioritizing project-level risks in the master test plan. In particular, this proactive technique helps us anticipate many of the traditional showstoppers. Every developer we know can readily name unexpected problems that stopped the project at the worst possible time—usually right before or after implementation.

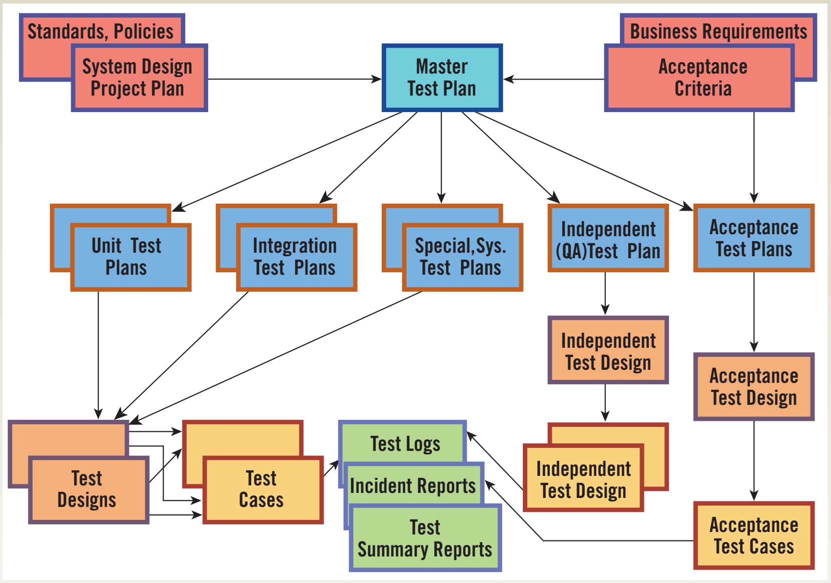
When working with a project team, we invariably find that they're able to use proactive risk analysis to identify a large number of potential showstoppers. The typical group reports that traditional project and test planning would have overlooked about 75 percent of these.

Once we've identified and prioritized risks, we can define the pieces of the system that need to be tested earlier. These often aren't the elements that the organization would ordinarily have scheduled to build early.

Whereas the typical development plan develops entire programs, often in job execution sequence, the Proactive Testing Model defines the system's parts—units or integrations—that must be present to test the high risks. By coding these modules first, *testing drives development*.

Building and testing these high-risk pieces early helps developers to catch problems before they've done additional coding that would have to be redone.

The effect is even more profound with respect to special tests. Rather than being employed within the system test (such as load and security tests), or often overlooked entirely (such as tests of training, documentation and manual procedures), special tests also merit detailed test plans and resulting identity. Therefore, each build may contain components that traditionally would have been addressed by unit, integration and system tests.



A Sophisticated Standard

The test-planning structure suggested by IEEE standard 829-1998 suggests conceptualizing test plans at different levels, either as individual documents or sections within a larger document.

An A-B-C Example

To get a feel for test-driven development, let's look at a simplified example. Assume that we have a system consisting of three programs, A, B and C, that in production would be executed in the A-B-C sequence. It's likely that the programs would also be developed and tested in the same sequence.

However, our risk analysis reveals that the integration of B and C poses the highest risk. Therefore, programs B and C should be built and unit-tested first. Since problems discovered in B or C could also affect A, finding these problems before A is coded can help prevent having to rewrite parts of A.

The next highest risk is program A's message capacity. In a typical development plan, we might build all of program A and then unit test it—but as a whole, A could be large and difficult to test and debug. By breaking A into two units, one dealing with the messaging and the other with the remainder of the program's functions, we can more immediately test the messaging capacity while it's still relatively easy to find and fix errors.

Next, we need an integration test to ensure that the two parts of program A function together correctly. And finally, now that the B–C and A–subparts integrations have been tested, we can address the third-highest risk: testing the integration of all three programs.

Note that the Proactive Testing Model doesn't dictate the specific sequence of tests. Rather, it guides us to plan the sequence of development and testing based on the particulars of each project to quickly and inexpensively arrive at the desired result: higher-quality software.

Developers who've worked with the Proactive Testing Model can immediately identify the nature and magnitude of problems that this approach helps them avoid. They know that these problems are often the cause of delayed, over-budget projects. When managers and developers realize how proactive testing helps them become aware of these risks—and prevent them—it's a WIIFM (What's In It For Me?) they can readily embrace.