October 2008 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

CODERS GONE WILD
Three common archetypes and how to handle them

GOOGLE WEB TOOLKIT
Your key to simplifying Ajax app builds

Got You Covered

by Michael Bolton

ISTOCKPHOTO

How much of the product have you tested? How completely have you tested it?

These are innocent little questions—the kinds of questions that a client might ask about *test coverage*. Their simplicity hides an enormous amount of complexity that clients and many testers might not understand. Answering these questions can be surprisingly difficult and can lead us and our clients into several traps.

Suppose that we were to define *complete testing* as "all possible tests having been perfectly performed." Most people would quickly recognize that this goal is impossible to achieve in anything less than an infinite amount of time, so when reasonable people ask, "Have you tested this product completely?" or "How much of it have you tested?" they *must* mean something else. They might mean "Have you identified all of the important risks that we've anticipated?" But they might also mean "Have you discovered important problems that we *didn't* anticipate?" They might mean "Have you tested on a wide variety of platforms?" or "Have you tested each element of the user interface?" Perhaps they simply mean to ask questions about the quantity or quality of our testing: "What has—and hasn't—been tested?"

In his book *Software Testing Techniques*, Boris Beizer defines coverage as "any metric of test completeness with respect to a test selection criterion." This makes some kind of sense when our test selection criterion is quantifiable: If there are 100,000 lines of code and we've only tested 80,000 of them, it follows that there is code that we haven't covered. In order to detect uncovered code, we could enlist the aid of code coverage tools. Such tools tell us about the statements or branches that have been touched (or not touched) during testing. For the code that we haven't touched, the tool tells us that we haven't touched it, and that might be important. Some people might believe that if we touch a line of code once, with a particular data value, it's been "covered," and in one sense it has. But coverage tools don't (and can't) tell us what we've looked for, how carefully we've looked, what we've observed, and what we've missed. We could perform tests that verify that a given function was performed, that a particular path was executed, that certain data values were entered, and that an expected result was returned, but while doing that, we could easily miss problems related to other data values, usability, robustness, security, performance, platform variations, timing, and so forth.

Yet there are other problems here, too. Code coverage tools can't read our minds and can't comprehend the intention of the product. Tools can't decide whether the code we're testing performs a function that we value, and tools can't recognize that a potentially valuable function is missing. Moreover, our source code doesn't give us the whole picture, because the program that we're producing is rarely part of a simple system. Our software tends to interact with other software and hardware in systems that are increasingly beyond our capacity to comprehend. In a presentation at STAR*WEST* 2007, Lee Copeland raised the problem that we have no clue of how to define coverage that includes behavior that emerges from complex systems given our limited understanding of them. This problem inevitably presents itself when our product interacts with the rest of a system—an operating system, third-party libraries, interconnected machines on a network. For such systems, we can't know about the completeness of the coverage we're getting because we can't comprehend what 100 percent is. We could do that for a closed system, but when software interacts with anything outside of itself, the system isn't closed any more.

There is a way out of the trap, though: We can model. A model is some idea, activity, or object that represents (literally, "re-presents") something—another idea, activity, or object—such that understanding something about the model may help us understand or manipulate the thing that it represents. Models hide or ignore certain kinds of information in order to help us focus on information that we care about. In his lengthy and vigorous (and quite amusing) attack on using lines of code as a metric, Beizer points out many valuable ways to model the code and its structure. For example, we could graph the control-flow, state-related, or logical structures in the program and then execute the program to cover the graphs—a much more robust metric of test completeness in terms of code coverage.

Yet the source code itself is a model of a computer program, because a computer program isn't just code. As Cem Kaner suggests, a program is not merely "a set of instructions for a computer." Instead, he says, a computer program is "a communication among several humans and computers who are separated over space and time that contains instructions that can be run by a com-

puter" and adds that "the point of the program is to provide value to stakeholders." That's why *code* coverage isn't the same thing as *test* coverage. Test execution involves configuring, operating, observing, and evaluating the product in some way, and operating the product always produces some code coverage. But our clients don't value the code as such; they value the things that the code does for them. That's what they're really asking when they ask about "how much of the product we've tested." The value of our work comes from describing what we've observed and evaluated, so if we want to understand, explain, discuss, or improve our coverage, we need a rich set of models.

The first step when we're seeking to evaluate or enhance the quality of our test coverage is to ask who is interested and what they're interested in—that is, who and what we're determining coverage *for*. Excellent testing starts by first questioning the mission. The higher-level we go at first, the more context we have for identifying things that might be valuable to observe. So think "higher," or "hire"—*what have we been hired for?* Who is our client? Who are the people who will use the product, directly or indirectly? Who are the other stakeholders? What is our mission for this particular cycle of testing? What is the overall mission for the project and the product? What do people say about it? What is its history? Are there comparable products available—competitive products, or past versions of this product? These questions can help us set context and focus our attention on things that are known to matter, while also helping us recognize things that haven't yet been anticipated as being important. As Donald Rumsfeld said, "There are known unknowns, and there are unknown unknowns." One key objective of testing is to move things that we don't know in the direction of things that we do know. Known unknowns hide risk. But the biggest risks may be the ones that we haven't thought of—the unknown unknowns.

One of the most effective and efficient ways that I know to start addressing the unknowns is to learn about the knowns. Get close to your clients and start asking questions. Begin with, "May I ask questions?" If the answer is yes, then ask away. If the answer is no, then it might be useful and important to outline the things you don't know about and the risks associated with proceeding with insufficient information. Jot down some notes about that, and don't forget to include a date and time.

If the client is willing to answer questions but isn't clear on what to cover, then you can provide some suggestions as to what you might look for, and ask if they'd be relevant. If you're inclined to start by thinking about verifying functional correctness, that's OK, but you will soon want to broaden your models to think about other kinds of problems and risks, too—and how your tests will cover them. I'll talk about how to do that in the next column. {end}

How do you start thinking about coverage? Where does that take you?

Follow the link on the StickyMinds.com homepage to join the conversation.