November/December 2005 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

PULL SOME STRINGS
Metrics that inspire change
PAGE 34

ONE FIX AT A TIME
Put an end to context switching
PAGE 50

Brushing Up On Functional Test Effectiveness

PAGE 26

More Than One Answer; More Than One Question

by Michael Bolton

Getty Images

Imagine that you're testing part of an application. It's a simple piece of business, one that adds two single-digit numbers together. Sound implausible? It would have sounded that way to me, too, until children arrived in my life, and with them the prospect of using educational software.

Somewhere, someone is testing a Web-based application for kids in which the player's character hands a number to a bunny and hands another number to a beaver, and then the bunny and the beaver carry their numbers to a wise old owl, who tells them the sum and displays it on his blackboard. The developers have created unit tests that show that the adding function adds single-digit numbers and calculates the result beautifully. They tried one plus one and got two; two plus two and got four. Nine plus nine returned eighteen. Even zero plus zero returned zero. There's not much work for a tester to do here—or is there? When the question is "What is two plus two?", how could "four" be an unsatisfactory answer?

As I've mentioned before, one of the biggest risks in testing is to assess a product from an insufficient number of perspectives. James Bach's Heuristic Test Strategy Model (HTSM), which I've introduced in previous columns, reminds us of different ways of looking at the product. Last time, we modeled the product by its *elements*: Structure, Function, Data, Platform, and Operations—SFDPO, or "San Francisco Depot." This time, we'll look at *quality criteria*, particularly the ones that are likely to be important to the customer. (There are criteria that are important to the organization producing the software, too; we'll deal with those next time.) To remember the customer-facing quality criteria, we think of CRUSPIC—Capability, Reliability, Usability, Scalability, Performance, Installability, and Compatibility.

Some testing authorities claim that all of the requirements for a program should be specified in advance, but Rapid Testers believe that it would be impossibly prescient to capture on paper everything that could constitute or compromise the quality dimensions for a program. So as we operate the product and think about the domain in which it will work, Rapid Testers continually cycle through these attributes, looking for weaknesses, making observations, and asking questions.

Capability has traditionally been the principal focus of testing. If the product can't serve its intended purpose—if it can't add two plus two at all—it's clearly a dud. The primary questions around capability are: What is the job that the program is expected to do? Can it do that job? Capability is something for which our oracles—principles or mechanisms by which we recognize a problem—are typically specific, well understood, or deterministic.

Still, there are traps in capability testing. One trap is that we might be inclined to check that the program does what it's supposed to do, while failing to check that the program *doesn't* do what it's *not* supposed to do. Another trap is that oracles are heuristic—fallible, and subject to being overruled by more powerful heuristics in a given context. Yet another trap is to overemphasize capability tests, which are mostly functional in nature. For a product to be successful, it must be successful in aspects other than the functional ones. These are sometimes called "non-functional" criteria, which is about as confusing a term as I can imagine—"Yes, the non-functional stuff works fine!"—so I like to take Cem Kaner's lead and call these attributes "parafunctional." Capability is primarily a functional criterion; the other quality criteria under the CRUSPIC acronym are mostly parafunctional.

Reliability is all about our ability to trust the program to perform the things of which it is capable. A piece of educational software is less than perfectly helpful if it produces an incorrect result or a crash, even if it does so only occasionally. Reliability questions include: Does the program perform consistently? Are its results accurate with respect to some oracle? Can we depend on the program to perform its intended function without introducing side effects? Does it cause data corruption, intermittent interruptions,

or crashes? The HTSM also puts security under the reliability banner, so we would ask authorization questions: Does the program permit access to its capabilities to the people who are authorized to use it? Does it forbid access to those who are not authorized? We would also ask authentication questions: Does the program get confused over who's who?

Even when there's nothing functionally wrong with the program, it can suffer from **Usability** problems. Our program might add two and two like a champ, but suppose that, due to a design misunderstanding, the owl writes the numbers in chalk on a whiteboard? Suppose that it's unclear that the player's character needs to hand the numbers to the bunny and the beaver? Suppose that the drop targets for the numbers are so small that a child has a hard time manipulating the mouse? In general: What might annoy, frustrate, or stymie any person, thing, or program that might try to use our product?

Usability has at least two interpretations: One is ease of learning, and the other is ease of use. Sometimes one of these attributes is compromised in the service of the other. An application that I once worked with was an order-entry system designed for waiters in a restaurant. New users of that application often complained that it was hard to learn, but after a few weeks of practice, they swore they would never happily use another system. We've heard the same kind of debate from command line users and GUI aficionados; sadly, the discussion is usually oblivious to the needs of a specific user performing a specific task.

Usability is often controversial because it's seen to be subjective and not quantifiable. Rapid Testers don't see anything wrong with subjectivity. The key to reporting subjective problems is the ability to articulate them clearly and to acknowledge the ways in which they might pose obstacles for some users, while supporting others. Context matters. In the case of an educational program, we're bound to ask if the program is suitable for the age group to which the product will be marketed. As testers, we don't make the final decisions; our job is to provide information to management and to try to be as thorough and as inclusive as possible.

Scalability is related to how a program behaves as its sandbox gets bigger. Our program might add two and two perfectly in the lab with only one person testing it. But will the application continue to function when hundreds or thousands of kids are logged in? When we add application servers?

One of the symptoms of a scalability problem is a decline in **Performance**. Questions about performance can easily demonstrate a weakness in capability testing. The answer to What is two plus two? might be entirely correct, but if the answer takes three minutes to arrive, we might suspect something abnormal. On the other hand, we might also suspect a problem if performance were *too quick*. If the calculation were requested from Earth, to be performed on a Mars lander eight light-minutes away, we would not expect an answer to return for at least sixteen minutes.

A piece of software does no good if it won't work on a user's machine, so we ask questions about **Installability**. This covers not only the installation of the program, but the new user's first encounter with it. As the commercial says, "You never get a second chance to make a first impression." Were all of the components installed properly onto the user's machine? With the right defaults? In the right language? Was anything else installed inadvertently? How does the program cope with older versions of itself? If the user decides to uninstall, can the product be removed cleanly? Is the user's data safe from an overly eager uninstallation routine?

Even if we're not dealing with kids' software, "plays well with others" is usually an important quality dimension. **Compatibility** requires us to consider how the program interacts with other programs and with its platform, which we defined expansively in the last column as "all of the aspects of the product—and the system that surrounds it—that are not under our control." (That's another inclusive definition; by defining things this way, we welcome expansions and extensions to traditional notions.) So, does the product introduce instability on the system? Is it vulnerable to side effects from other products?

A lot of the precepts of Rapid Software Testing aren't terribly new. Instead, they're important ideas, wrapped in mnemonics and governed by heuristics that expert testers have been using forever. We've been reminded for years to test for the "-ilities." Rapid Testing allows us to put a handle on the things that we've done and thought about all along. **{end}**

Michael Bolton lives in Toronto and teaches heuristics and exploratory testing in Canada, the United States, and other countries as part of James Bach's Rapid Software Testing course. He is the program chair for the Toronto Association of System and Software Quality, and a regular columnist for Better Software *magazine and StickyMinds.com. Contact Michael at mb@developsense.com.*