July/August 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

BUILD A BRIDGE
And bring agile to your enterprise
PAGE 26

FOR WHAT IT'S WORTH
The true value of testing
PAGE 32

THE MANY LAYERS OF AJAX

Techniques to Sweeten the User Experience **PAGE 18**

The Factors of Function Testing

by Michael Bolton

In the previous issue, I introduced the nine test techniques that James Bach names in his Heuristic Test Strategy Model (HTSM). In this column, I'll focus on function testing, which is perhaps the most widely applied of those techniques.

In function testing, we test what the program does. The basic idea is to identify each program function and then test that each does what it's supposed to do—and doesn't do anything else. Sounds easy, doesn't it? But there are catches.

One catch is that there may be many ways to invoke a given function. For example, I can make Microsoft Word print a document in the following ways:

- Pressing Ctrl-P
- Choosing File and then Print from Word's main menu
- Right-clicking a Word document in Explorer and choosing Print from the context menu
- Calling Word's Print function from an application that uses Visual Basic for Applications

As an exercise, consider additional ways in which Word's print functions can be triggered; don't stop until you get to at least twelve. Next, consider some function within an application that you might test and how many ways in which that function might be invoked.

Function testing needs oracles—principles or mechanisms by which we recognize a problem. An oracle may be a reference document that defines some expected answer. It may be a reference program that we can run before or after a test to compare with the result of some function. It may be a heuristic principle—when we click a button that says Cancel, we expect the current action to stop or the current dialog to be dismissed. Rapid Testers also use "live oracles," such as interviewing a businessperson who sits with us as we exercise functions in the program. The conversation combined with operation and observation is a very powerful test technique.

GETTY IMAGES

I like to identify a function as "something that happens in response to something else happening." Some functions seem to be triggered by the user. I say "seem to be" because the user performs some action—typically a keystroke or a mouse click—but that action doesn't reach our program directly. A keystroke triggers a chain of events—a message from the keyboard controller, an operating system interrupt, a handler within the operating system (which itself may be interrupted), and a message sent from the operating system to our program. After that message arrives, our program may take many steps itself to process this function. Each of these steps can be considered a function in its own right.

There is a powerful argument in favor of testing functions at the earliest time, at the lowest level, and to the greatest extent that is feasible—what developers call "unit tests." Units, like functions, vary in size and scope. I'll define "unit" here as "the smallest part of the program that we currently care about." Each arbitrarily small sub-function holds the potential for error. Automated unit tests for each function, written as the code itself is written (or in the case of test-driven development, *before* it is written), will provide at least some assurance that each function fulfills some requirement to some degree. At the unit test level, oracles tend to be mechanistic, based on a single principle that the developer had in mind when she wrote the function.

In my experience, developers write the unit tests; testers don't often get involved at this level of testing. That may be OK, as testers and developers have different motivations. The usual goals of developers' function tests are to confirm that the function works and that changes don't break existing functions. Such tests are designed to be quick and frequent, trading thoroughness and variability for simplicity and repeatability. When testers perform functional tests, the task is closer to the user's task; the focus is more investigative, looking for exceptional cases and unanticipated behavior. This requires broader coverage and greater variation than automated unit tests and can involve harsh tests designed to probe for weaknesses in the system by undermining it. Divergent perspectives are valuable; testers and developers can learn a lot from each other. Elisabeth Hendrickson and Jonathan Kohl speak and write compellingly about how to foster collaboration to promote better unit testing (see the StickyNotes for more information).

Functional units form the building blocks of higher-level functions, where the program is exposed to a wider variety of data, platforms, operational profiles, and timing relationships. Rapid Testers use the Product Elements and Quality Criteria sections of the HTSM to inspire test ideas, and we apply them by looking at the program through the lenses of structures and functions. Another way to identify functions at a variety of levels is to use James Whittaker's ideas about the four users of the program—human users, the operating system, the file system, and application program interfaces (see the StickyNotes for more about the four program users).

To build a functional model of the program at the very highest level, try to identify all the things that a user can do. Look at every part of the screen, trigger pop-ups and dialogs, select options within them, choose items from drop-down menus, drag and drop, enter text, perform calculations, and trigger field updates. Check that things can be accomplished by the keyboard, the pointing device, or more indirect means like an API. Choose oracles to determine which input methods should be supported in a given situation. Programs usually run on a variety of platforms (to Rapid Testers, a platform is everything upon which our program depends, but which is outside the scope of our current project). Some functions in our program may be autonomous, others may simply request a service from the platform, and still others may call a set of services while doing some work between each call. Some platforms may interpret requests differently, which requires not only that we test multiple functions but also that we test them on multiple platforms.

For each function, observe not only *output* but also *outcome*—a useful distinction Boris Beizer makes in his book *Black Box Software Testing*. Output is the immediate, apparent result of the function that we're observing (e.g., the answer to our primary question, data that appears in the empty field, the printed document). Outcome is something much more than that; it's the entire state of the system after we've tested the function. Some functions have no discernable output, but certainly have an outcome. Sometimes we want to test to ensure some change happens to the system state, while in other cases we want to make sure that there's been no effective change. Use tools to identify outcomes that might otherwise be invisible. For Windows, I've found the freeware SysInternals tools to be particularly handy for monitoring the file system, the registry, and the process table.

Identifying and testing every function in a program of any significance is effectively impossible because of the number of functions, the variety of data we can use, and so on. So how do we choose which function tests to run? One way is to consider risk. At low levels, function tests might look for coding errors that developers could make. At higher levels, they might look for business risks and problems associated with fulfilling the user's task.

Finally, we don't have to depend upon function tests alone. Other test techniques exercise a program's functions while addressing risks that function tests might miss. Testing is greatly strengthened by a diversified test strategy, so do perform plenty of function testing, but don't leave it at that. **{end}**

Outcome is something much more than that; it's the entire state of the system after we've tested the function. Some functions have no discernable output, but certainly have an outcome.

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. Michael is also program chair for the Toronto Association of System and Software Quality. He is a regular contributor to Better Software *magazine. Contact Michael at mb@developsense.com.*

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware

- More on Elisabeth Hendrickson and Jonathan Kohl
- James Whittaker's four users of a program

Don't Stop Now!

Log on to **StickyMinds.com** and join Michael Bolton and your peers in a conversation about this topic. At the end of the digital column, add your views or just read what others have to say.