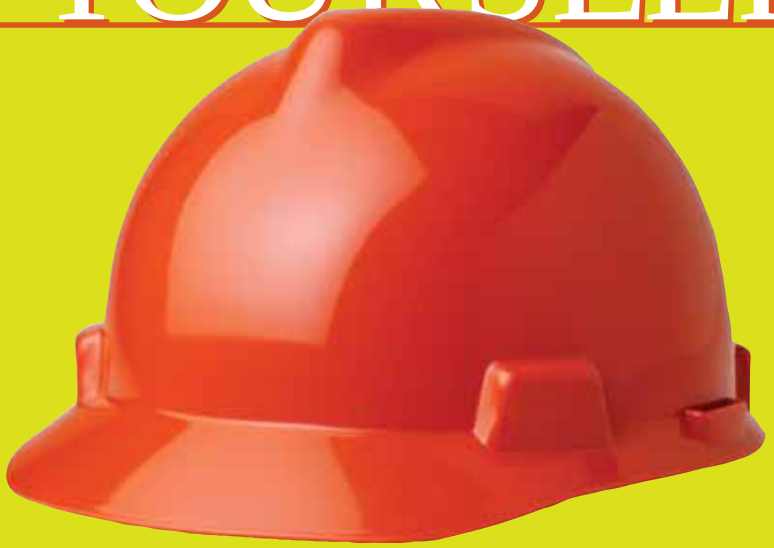
October 2005 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

DO-IT-YOURSELF



A How-to Guide for Fixing a Failing Project

PAGE 18

Elemental Models

by Michael Bolton

In previous columns, I've talked about critical thinking and its importance to Rapid Testing, and I've introduced James Bach's Heuristic Test Strategy Model (HTSM). In the previous issue, I covered its first part, the Project Environment. This time, I'll focus on the Product Elements, which, as a whole, provide one of the model's perspectives on test coverage. To Rapid Testers, coverage is the extent to which we've tested a program according to our mental models.

Why "models" instead of "model"? When we model something, we focus on certain attributes of it while ignoring others. This gives us the opportunity to comprehend some important aspect of the system, but by removing information, we risk being oblivious to other important things. Good mental models are heuristic; they're a set of guidelines to help us solve a problem, but they are both provisional—used for a specific, temporary purpose—and, above all, fallible.

If we are conscious that our models are incomplete, we may be more inclined to think not only about the donut but also about the hole: where the boundaries are between them, whether the hole is an essential part of the donut, and so forth. By using several models, we hope to avoid missing important information when someone produces a donut that has cream or jelly in the middle instead of a hole—or when someone arrives with a cinnamon bun or a bagel. The quality of our coverage depends not only on the quality of our models but also on the extent to which they are diverse.

For example, one way to model a product is to look at its source code as a sequence of lines, each of which could be executed at some point. We could use a tool that shows we've executed every line and dupe ourselves into thinking that we've tested well. But if a feature were *missing*, the world's greatest code coverage tool wouldn't notice. As Rapid Testers we do not restrict ourselves to that single model when we design our tests. Instead, we try

Getty Images

to analyze and decompose the product using several different models of the system. In the Product Elements, the key words are Structure, Function, Data, Platform, and Operations—SFDPO—and to remember it, we think "San Francisco Depot."

Even though we don't usually think of software's physical nature, we can model a product in terms of its *Structure*; a product can be manifested in (or on) concrete, physical parts. Files on a disk might include object code, templates, sample data, configuration files, or Registry settings. Other physical aspects of a product might include cardboard boxes, manuals, pieces of paper, CDs, monitors, workstations, servers, network cables, and so forth. *Does our test strategy incorporate ideas informed by these physical objects?*

When developers are writing or updating a product, changing some aspect of the intended structure can strengthen or weaken the product. What could happen if some component were defective or missing? We can test this easily by renaming a file—rendering it invisible to the application that calls it. James Whittaker, in his wonderful book *How to Break Software*, gives an example of renaming MSRATING.DLL, a library associated with Internet Explorer. The .DLL is supposed to prevent access to restricted sites, but when it's missing, the larger product fails to block access—and the failure happens without an error message. In some conditions, something that is only momentarily absent might as well vanish entirely. *If the network cable is unplugged, does the product topple?*

Function is what the product does. Function is usually the premise of our requirements or stories and the focus of our code, thus apparently straightforward to model. On the other hand, a functional model might gloss over significant details. Try observing a piece of software for a few minutes and consider that every visible or audible change to the system is the result of some function. *Which functions are under the direct control of our product? Which are not? How do they interact? Which functions and interactions might be missing from our tests?*

If we have a function that's intended to delete a customer record from a database, we test to make sure that the record is there before we delete it and that it is gone

afterward. The hole in the donut here is the idea that while a piece of software is intended to perform some function, it should perform no *other* function and no *additional* function. When a user invokes the "delete customer" function, we should be alarmed if the "delete transaction" function were called instead or as well. This is why, when we think of functional tests, Rapid Testers also think quickly: *What functions shouldn't happen? Could we use tools like (on Windows) PC Magazine's INCTRL or the SysInternals Registry Monitor (REGMON) and File Monitor (FILEMON) to watch for the unexpected?*

If we stick strictly to a functional story ("this does this"), we'll miss all kinds of problems that *Data* can trigger ("this does this *with that*"). Material products are not expected to cope gracefully if the input is utterly unreasonable. No one would expect a shopping cart to keep a criminal imprisoned or to survive if someone dropped a steamship on it. But in the world of software, all data arrives in bits, which means that the application itself has to determine reasonableness. Thus hackers regularly—and successfully—attack by sending input that is unexpected or "bad" in context or by overwhelming tiny input fields with colossal amounts of data. Forced-error tests, invalid messages, incompatible file types, or malformed packets can expose dramatic risks to a product. For another hole in the data donut, you might think about how the program behaves when expected data is missing entirely. Think about how our world could be unrecognizable if only one thing were different about it. Now consider: *How would things be different for our product if the expected data were different—even if only by a single byte?*

When we think about the *Platform* on which our software runs, it's easy to fall into the trap of considering only the operating system or computer. In the HTSM, platform represents the stuff that you can't change: all of the aspects of the product—and the system that surrounds it—that are outside the immediate control of your project. Most elements of the platform are developed and built externally. Even components or libraries that are internal to our organization might not be under development or within scope for our current project, in which case we're stuck with them.

We consider platform because no technology is created from scratch. All software is built on some existing technological foundation. If any part of that foundation shifts, the product can fall like a house of cards. If some part of the user's platform is incompatible with our product, the user may not have the option or the inclination to change that part. We also might be blindsided if the user changes the platform and breaks its existing support for our product.

There's one kind of platform that testers sometimes forget, and it's strongly linked to data—previous versions of your product and its data files. We don't have control over old data because it comes from the past. Upgradability and backward compatibility testing focus on problems that arise as a product evolves. *Have we really considered everything upon which our program depends? How would changing a dependency affect the program?*

Other aspects of the model—especially structure and function—describe the product in terms of a kind of Platonic existence, isolated from the rest of the system. *Operations* are patterns of actually using the product; they're where the product meets the people who use it.

Testers often write simple operational tests based on use cases, which can be helpful ways to describe functionality but which typically are instances of single, atomic tasks. Testers also consider extreme operational tests, such as load and stress conditions. Both simple and complex operational models can ignore mundane, but typical, real-world ways of operating the program. People perform a bunch of different actions in a row. They use business products outside of business hours, they use products in different time zones, and they use products in innovative but plausible ways. People also use products in ways we might disfavor. They change their minds, they go back and forth, and they make mistakes. That's why Hans Buwalda's "soap opera" testing (see the February 2004 issue of *Better Software* magazine) is a wonderful (and fun!) way to think about operations. *Have we considered disfavored use? Have we also considered disfavored users—hackers—and disenfranchised users—people with disabilities? Have we considered installation and deployment—especially continued operation during an upgrade cycle—as part of our program's operational life?*

SFDPO is only one set of product models that Rapid Testers find useful for designing tests. As an exercise, you might wish to consider how the Structure, Function, Data, Platform, and Operations models are interdependent—how each is related to and conditioned by the others—and how thinking about the interdependencies could lead to even more test ideas. **{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 a regular columnist for Better Software *magazine and StickyMinds.com. Contact Michael at mb@developsense.com.*