March 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

LIFE'S NOT A DRESS REHEARSAL
Plan for emergencies now
PAGE 14

CASTING CALL
Hire a tester with an agile attitude
PAGE 16

Tester PI: Performance Investigator
PAGE 20

Taking Our Act on the Road

by Michael Bolton

GETTY IMAGES

In the Product Elements section of James Bach's Heuristic Test Strategy Model, he extends the traditional definition of "platform" beyond the hardware or operating system, as "any part of the product that is outside of the control of the current project." The platform can include protocols, application frameworks, and runtime libraries—even those written within our own organization, that we cannot change in the scope of the current project. We'll use our extended concept of "platform" to expand the boundaries of what portability means. Portability is about re-engineering the product, including our test strategy for it, for the purpose of adapting the product to some other platform.

Portability is subtly different from compatibility. Compatibility testing tries to answer the questions "Do we play well with others?" and "In our current environment, how might we get into conflict with other things?" Portability asks, "Can we take our act on the road? What might we expect—or not expect—as the result of a deliberate choice to change the product's home base?"

There are two dimensions of change when we port something: We're changing the product's *code* because we're changing its *platform*. One plausible risk is that we'll neglect some aspect of the changing territory. This is a general systems problem. Our concept of the original platform is a model; our concept of the new platform is too. To test effectively, we need to recognize the ways in which any model is an incomplete representation—usefully similar, but necessarily simpler than the thing it represents. We'll likely need to refine our existing models and create new ones. Rapid Testers look for a rich starter set of modeling ideas in the Product Elements section of HTSM—structure, function, data, platform, and operations—and then add other ideas along the way.

The first thing to keep in mind is that concepts like "Windows," "processor," and "Firefox" are themselves incomplete descriptions of some element of the platform. Each element on which the product runs may have a number of versions, each with subtle differences. For example, some browser versions might have bugs in the way that they parse markup languages; others might have bugs in the way they implement some related technology, such as Javascript or CSS. Consequently some of our existing, proven functionality might not work at all with some version of some component. The fixes that we make might break existing functionality in the other versions of that component, and the "final" fix could break our shipping product. Good regression tests should help to defend against that, but every bug is based on something that we've failed to anticipate. If our code doesn't properly address the new platform, our tests might not either.

So What Else Can Testers Do?

One important thing that we can do is to have ongoing conversations with our clients about risk. The client may lose interest in the value of supporting a particular version of some platform component when that value is balanced with the additional costs of testing and support. Also in collaboration with the client, we can choose to sacrifice some coverage depth (for example, testing a small number of browsers exhaustively) and increase the breadth of coverage. To do this, we might use heuristic tools such as test matrices, in which we perform some test ideas on some platforms and other ideas on other platforms, such that we sparsely cover a large table of tests versus platforms (see the StickyNotes for more information). We could try to streamline the test effort by searching the Web for known problems with existing browsers and designing tests that help make sure that our application doesn't trip over those problems.

If we build our product with different compilers or in a different language, our developers might not find library support for some of the functions that we need;

those functions will have to be written for the application. We can apply the heuristic that newly written technology may not be as robust as libraries that come with the operating system or compiler. If those libraries are in wide distribution or have been around for a while, we might choose to focus tests on functionality that our own developers have had to build specifically for the project. A good relationship with our developers can help us identify the areas they believe to be risky. Sketching out any structural diagrams of the application might help us to see the interfaces between changed and unchanged code, where brittleness might expose itself.

There's often a buried presumption that the compiler and operating system will help to manage and make abstract the differences between hardware platforms.

There's often a buried presumption that the compiler and operating system will help to manage and make abstract the differences between hardware platforms. That doesn't mean that the platforms are all the same. As Joel Spolsky says, "All non-trivial abstractions are to some degree leaky" (see the StickyNotes for a link to Joel Spolsky's "The Law of Leaky Abstractions"). For example, if we port an application from a desktop platform to a handheld, we can't reasonably expect access to the same amounts of long-term storage; we'll need to think about setting different constraints for stress tests. We'd like to believe that we will anticipate such problems in our analysis, but will we anticipate all of them? Can we think of tests of different reserved characters, byte ordering, data encoding, threading models, signals, interrupt handling, exception handling, and regular expressions? Note that some operating systems (like Unix) are case-sensitive with respect to file names, while others (like Windows) are not.

Will our product need some resource, such as a font, that might not be available on the new platform? Will it still look good under different display resolutions? If we move some portion of our product to an embedded system, can we even count on having a display and a keyboard, or will we need special equipment to provide input and output for our tests?

We might choose to port a program to a different programming language for the explicit purpose of taking advantage of some of the features of that language, in which case our test oracles might have to change, too. I'm in the process of trying to port PerlClip, a testing tool written by Danny Faught and James Bach. It helps generate test data quickly in the form of text strings, using Perl's string handling syntax, before popping these strings into the Windows Clipboard. When I port that program, I'll have to remember that Perl and Ruby have subtle differences in their syntax. I'll have to figure out whether I want to make the new product compatible with Perl's conventions or Ruby's; my tests will need to reflect that choice. Porting a program usually means porting its test strategy.

Even if we've automated all our tests, our testing tools might suffer from the same kinds of portability problems as the application that we are testing. Will we have to adapt or change our tools to fit the different environment? Will we need new or different equipment for testing? Do we have the training and resources to integrate them into our test lab and to use the new test platforms skillfully?

Another risk is that our tools might be *too* compatible with the new platform. For example, to help make scripts more portable between Windows and Unix, Perl's compiler helpfully and invisibly converts forward slashes to backslashes in filenames. That's valuable most of the time, but what if it helped to obscure an important platform difference—where a slash in the wrong direction would cause a radical and unwelcome change in behavior? In what other ways might our tools inflict help upon us?

General systems thinking tells us that the product is always part of some larger system, and that adapting the product to a new platform is part of maintaining that system. If that's so, then why not just make portability a subset of maintainability? If it helps your thinking not to follow the HTSM strictly, but merely to use it as a point of departure for creating your own test strategy model, then by all means go ahead. Models are useful only to the extent to which they fit the context, and context-driven thinking requires our test strategy to be appropriate to the task. For our purposes, though, we've noticed that our platforms rarely remain static, and thus it helps us to keep portability in mind. We've found that thinking explicitly about portability strengthens our strategies and helps us to anticipate change. **{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 program chair for the Toronto Association of System and Software Quality and is a regular columnist for Better Software *magazine. You can contact Michael at mb@developsense.com.*

STAR EAST SPEAKER

Sticky Notes

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

- Test Matrices
- "The Law of Leaky Abstractions" by Joel Spolsky

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.