February 2007 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

THE MAGIC NUMBER
Three techniques for testing with databases

COMPLEX CODE?
Get back to basics

The Case of the Missing Fingerprint

Solve the Mystery of Successful End-of-Project Retrospectives

One Step Back…Two Steps Forward

by Michael Bolton

GETTY IMAGES

Regression—the notion that the quality of our product is backsliding in some way—is one important motivation for testing. Regression tests are those that may help us identify failures associated with changes to code that we believed was working before. A regression test happens in response to a change. That change might be a new feature or a bug fix in the software that we're currently developing, but it might also be a change in the platform—a supporting library, component, tool, or anything that is outside the scope of our current project. We can check for regression at any level of the project—from the smallest unit test to an end-to-end system test—but that doesn't mean a regression test must be a repeated or automated test.

When we test for regression, we first try to catch things that have been broken unintentionally by some change. Second (and, in my experience, sometimes forgotten) is that we want to take any opportunity to find new problems. Third (and almost always forgotten) is that we want to identify opportunities to improve our test design.

I'll equate unit tests and developer tests here (even though one might not be the other), and I'll define a unit as "the smallest bit of code that we're interested in at the moment." In a test-driven development (TDD) model, a programmer writes a TDD test, which typically contains a single assertion, before writing the code that allows the test to pass. Then the programmer writes the simplest code that will run successfully with respect to the new test and all previous tests.

Once the code passes that milestone, the development work with respect to that test is considered finished. If the code needs to do more, the programmer repeats the cycle—but in each round of test-then-code, not only should the most recent test pass but prior tests should too. Failure in an old test indicates an unexpected consequence of some change and requires immediate investigation. Until the unit test runs successfully for the first time, it's a milestone to be achieved; thereafter, it's a change detector (as Cem Kaner calls it) or a regression test.

However, there's a cognitive issue. A milestone is usually focused on accomplishment—not on risk. A TDD test is a demonstration that the code *can* work, not a guarantee that it *will* work in a given context. Some change in the context or some unconsidered aspect could cause the program to fail. At a milestone, we're provisionally finished coding; we're not likely to be finished testing. This might be a good time for developers to add extra unit tests that more aggressively challenge the code and our understanding of it, and to collaborate with a contrary, risk-focused tester. Take a critical point of view, focusing not on how the unit should work but on how it could fail, especially at the interfaces between it and the rest of the system. In addition to the extra confidence that we gain by developing and running these new unit tests, we'll garner extra change detectors.

The cost of developing simple, focused unit tests is typically very low compared with the cost of developing higher-level system tests. Passing unit tests provide reassurance; failing unit tests are valuable in that feedback is immediate, locating the problem is easy, and little investigation is required.

When we put two or more units together, we're creating a new system that is capable of performing more elaborate work. As much as we'd like to believe that we can understand a system, increasing complexity may result in surprising behavior. At higher levels, we need tests that are more complex and based on more models than the unit tests are. People have a lot of names for these kinds of tests, such as integration or acceptance tests. I'm going to call them system tests; although they may or may not be testing the whole system, they do test some systems of two or more units.

At the unit level, test coverage tends to be based on structural and functional models. At higher levels of the system, other coverage models—data, platform, operations, and time—come into play.

Moreover, as we build functionality into the system, we increasingly can test for quality criteria that must be satisfied. Unit tests tend to focus on capability; system tests address reliability, usability, security, scalability, performance, installability, and compatibility. Oracles for these tests are sometimes more subjective and more human-oriented than can be modeled easily on a machine. That's OK, because humans matter; programs are generally ways of fulfilling some human task. Programming an example of that task as a sufficiently complex test could be on the same order of difficulty as programming the application under test. As a skilled tester learns more about the program and the task, he will recognize new risks and new test ideas that he can apply; a machine won't.

Can we ever repeat a test? To some degree, no. We're never going to be able to repeat *all* the elements of a test that happened at a given time. But we may be able to reproduce some aspects of the test. When someone says "I want to repeat that test," what he really means is "I want to repeat something I think might be important about that test." The test idea—the motivation to test for a specific risk—might be the most important thing; repeating other aspects of the test—the data, the order of operations, the platform, the scenario—might be less important. Varying those things can expose problems that existing tests might be missing, and that gives us a chance to improve the design of existing tests or our test strategy. Cem Kaner gives a compelling metaphor: Imagine that a border guard is a tester and the person wishing to cross the border is a requirement. Would you really want to limit that guard to asking the same questions every time that person tried to enter the country—especially when something about his appearance had changed?

Consider what is gained and lost in repetition. A suite of repeating automated integration- and system-level tests might have significant value, but what does it cost to develop and maintain them as the program under test changes? Tools can help us see things that we would otherwise miss, and automation can easily extend our ability to vary some data set. But developing automation also incurs opportunity cost, taking time away from new tests that we could run and new discoveries that we could make. At the system level, we don't want merely to check for regression; ideally, we'd like to perform tests that are capable of exposing both regression *and* new problems. When modeling the task as a program or script is both straightforward and worthwhile, when observations and pass/fail rules are simple, and when a data set can be varied and checked easily, then it might be a good idea to automate a test and repeat it. When we lower our investment in developing and running automated system-level tests, we can target specific problems more flexibly and increase our investment in variation.

If we want to ensure that things don't break, we need repetition in some of our test effort. So where do we do it, and how do we keep the value high and the cost low? As it turns out, we may already have a ready source of repeating automated tests. The unit tests, as change detectors, are the first line of defense against backsliding, and they provide simplicity and repeatability. If we have a good set of well-crafted unit tests, we can afford to spend time running more manual and more varied tests at the system level. **{end}**

Michael Bolton lives in Toronto and teaches heuristics and exploratory testing in Canada, the United States, and other countries worldwide as a consultant and co-author 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.*

If you are a strong advocate of high-level test automation, consider some things that automation tests poorly. If you are an automation skeptic, think of things that automation could do well. Any epiphanies?

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

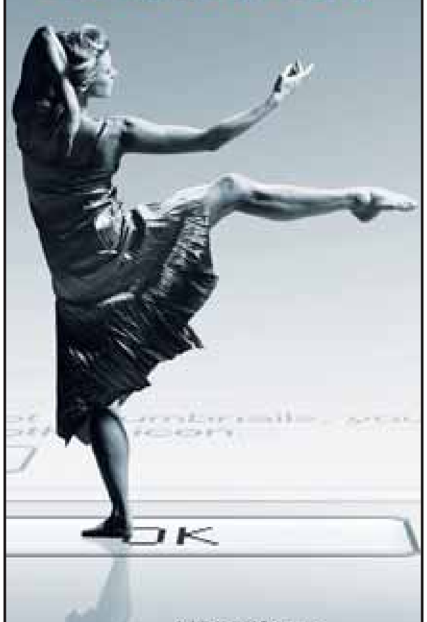
step into a new testing generation

GUIdancer *is the unique new tool from Bredex GmbH for automated software testing, offering the ability to create GUI tests without programming.*

Creating reusable, easily maintainable tests using straightforward high level specifications – welcome to the world of **GUI**dancer.

To find out more – www.guidancer.com

GUIdancer®



Additional features:
- *XML and HTML reports*
- *Intuitive user interface*
- *Runs as standalone application or Eclipse plugin*
- *User-defined hierarchical organization of test elements*
- *Platform independent*
- *Event Handling*
- *Context-sensitive help and sample projects*
- *Comprehensive documentation*
- *Observation Mode*
- *Multi-User Capacity*

Requirements:
- *Java 1.5 or later to run* **GUI**dancer
- *Java 1.3 or later for Applications under Test*
- *Swing corresponding to VM*
- *Microsoft Windows/Linux/Solaris*
- *Oracle 9i or later (optional)*

BREDEX
Software-Entwicklung und Beratung
www.bredexsw.com