June 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com

IT'S ELECTRIC
A static analysis tool to energize your code
PAGE 18

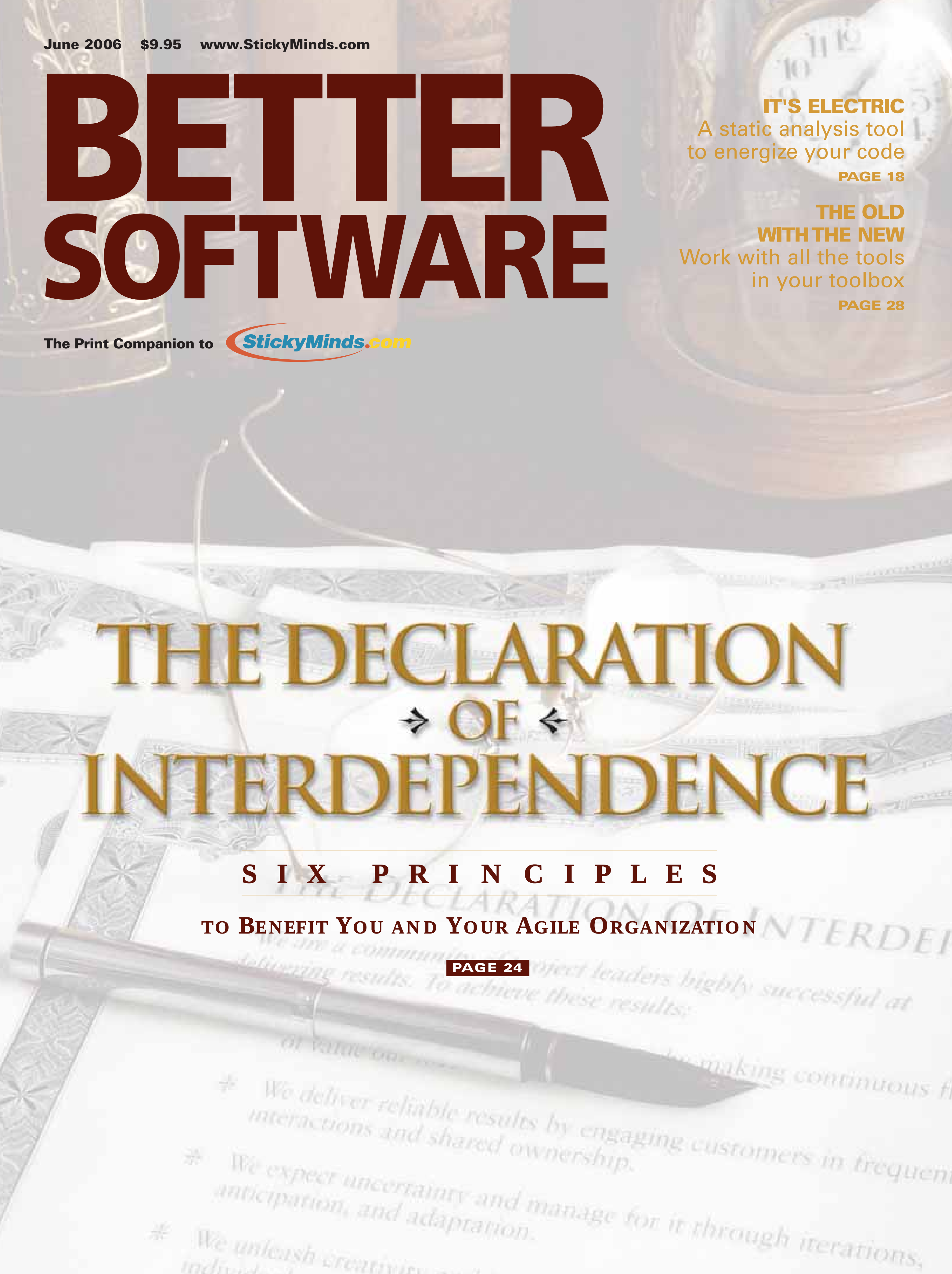
THE OLD WITH THE NEW
Work with all the tools in your toolbox
PAGE 28

THE DECLARATION OF INTERDEPENDENCE

SIX PRINCIPLES

TO BENEFIT YOU AND YOUR AGILE ORGANIZATION

PAGE 24



THE DECLARATION OF INTERDEPE

We are a community of project leaders highly successful at delivering results. To achieve these results:

of value ... making continuous f

✳ We deliver reliable results by engaging customers in frequen interactions and shared ownership.

✳ We expect uncertainty and manage for it through iterations, anticipation, and adaptation.

✳ We unleash creativity

Test Patterns

by Michael Bolton

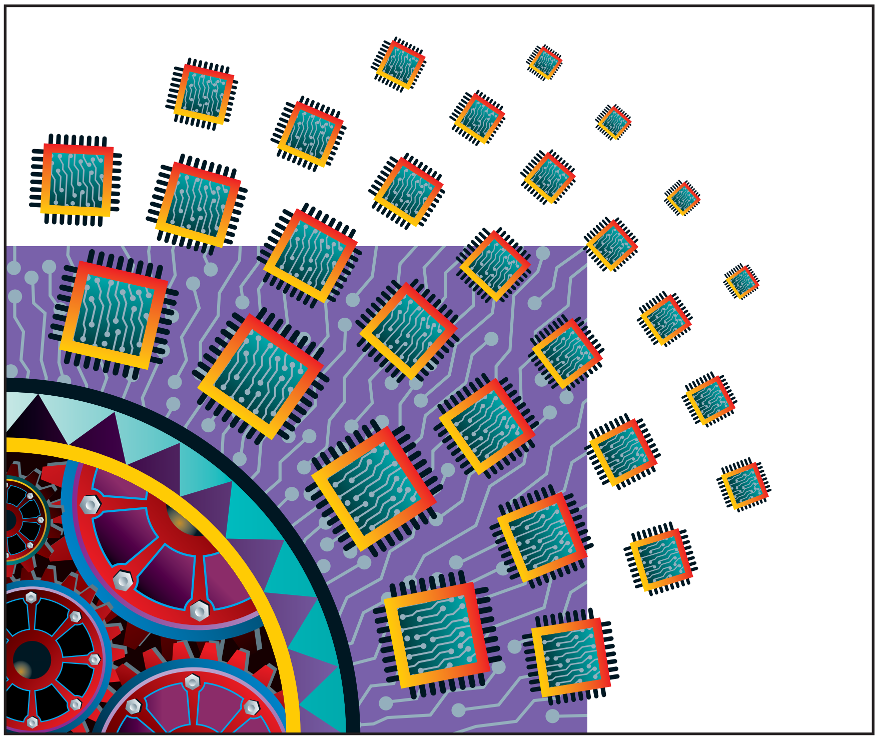
In this series, I've focused on the parts of James Bach's Heuristic Test Strategy Model (HTSM) that help us generate test ideas. For the next several columns, I'll discuss how to exercise these ideas using *test techniques*.

A test technique is a general pattern describing what we do when we test. We start by modeling the test space. We then determine coverage, oracles, and activities. We configure, operate, and observe the system under test, and finally we evaluate the result. Sometimes these tasks happen so quickly and intuitively that we don't even notice we're performing them.

In this column I'll summarize the HTSM's nine categories of techniques for testing systems. Notice the term "systems" rather than "programs." The former, more general term allows flexibility in modeling what we have to test, from a single line of code to a functional module, an application, or a suite of interacting applications and the platforms on which they work. The scope of the effort affects the techniques we choose and the ways in which we use them.

Function testing involves identifying each function and then testing it independently. A function causes the system under test to exercise some behavior, either changing or maintaining the system's state. The object is to ensure that each function does what it should, but also that it doesn't do what it shouldn't. For each function, we need a reliable oracle—a principle or mechanism that will allow us to recognize a problem.

Function testing is good for identifying the capabilities of a system, but the technique has weaknesses, too. First, unless we choose our models carefully, the number of functions in a system could be intractably large. To be effective, we need to make pragmatic choices about the scale of our functional tests. Second, function tests are intended to test each function in isolation, but some of the riskiest areas of any system are in the interactions *between* functions. Third, some defects may depend upon exercising a function more than once.


GETTY IMAGES

Domain testing focuses on clarifying and simplifying the testing effort by classifying things associated with the system. Identifying equivalence classes—groups of things that we deem interchangeable for the purposes of a given test—is the key task in domain testing. Practically anything to do with the system—input and output data, platforms, peripherals, users, functions—can be classified by some criterion. Good domain testing involves selecting items from the identified classes such that we cover the territory (e.g., normal data values vs. exceptional data values, representative platforms vs. unusual platforms, or expert users vs. novice users). If we observe clear divisions between classes in defined ranges, such as elements on a number line, we might test at the boundaries between them, where some theories of error suggest that mistakes are most likely to occur and easiest to detect. Boundary analysis—a subset of domain testing—may reduce the number of tests we believe we need, but some classifications don't have clear linear boundaries. To do domain testing well, we must develop skill at dividing and conquering the data and at identifying risk.

Stress testing is the process of overfeeding or starving the system, or both. This includes overwhelming the system with input, tasks, or users, or removing resources like memory, disk space, network bandwidth, or connectivity. The system under test can be anything from a single input field to the entire business process. We use stress testing to better understand the capacities and limitations of the system, looking for bottlenecks, constraints, and dependencies. The goal is to fix intolerable weaknesses in the program and prepare for and mitigate the tolerable weaknesses.

Flow testing involves running the system without halting or resetting it. When designing a system, we typically simplify by describing single transactions or events in isolation from each other. Flow testing introduces the system to a more realistic, complex world, where transactions happen in sequence, concurrently, or with interruptions. Good flow testing doesn't decompose the system as function testing does, or undermine it as stress testing does. Instead flow tests model realistic system operation, ensuring that things are done correctly and in the right order.

With *scenario testing*, we test to a compelling story about the system and its users. Powerful scenarios motivate interest and empathy. Scenario tests include use cases; user stories; the birth, life, and death of a piece of data within the system; and soap opera tests, which posit highly improbable but possible scenarios. Scenario testing tends to find more business-facing bugs than developer-facing bugs.

In *claims testing*, we seek what anyone (a requirements author or marketer) or anything (a help file or a shrink-wrapped box) says about the system, and we test that claim's validity. Requirements documents and specifications are the most obvious sources, but we can also test claims made in emails and conversations, sales and marketing materials, end-user documentation, and tutorials. Some claims might be inaccurate, but if we identify where the claim and the system are inconsistent, the project team can fix one or the other.

User testing tells us to test with real users or real-user models. A user, in Rapid Testing parlance, is anyone who might use the system or have an interest in the test effort. I regularly run an exercise in which Rapid Testers try to identify all of the project's user roles; we don't stop until we reach thirty. We tend to focus our modeling on end-users (or their managers or customers), but we might also perform tests to serve the interests of the help desk, training staff, or chief financial officer. We'll also perform some security testing because black hat hackers are potential users (or abusers) of the system.

Risk-based testing posits the harm that could come to some person if a threat were to expose some vulnerability in the system; we recognize or imagine a risk and perform tests to reveal it. I've found three useful ways to generate risk-based test ideas:

- Consider something that could go wrong, such as a programming error or an ambiguous requirement.
- Contemplate the consequences of something going wrong, such as a phone call to the help desk or a front-page story in the San Jose Mercury News.
- Determine what the development organization is prepared to risk (e.g., "A bug fix this late might have undesirable side effects; we'll risk leaving it in").

Automatic testing allows us to run a test zillions of times or in a zillion variations. The principal virtues of automatic testing are speed, precision, and repetition. However, test automation is software development, which can be tricky and expensive. We should balance the cost of repetition with the value of the problems it finds. Some tests we run repeatedly; others aren't even worth running twice.

The boundaries between techniques are sometimes blurry, and our definitions are somewhat open. When the name of a given technique suggests multiple interpretations, Rapid Testers try to use as many interpretations as possible to create more diverse tests.

I've noted in previous columns that coverage is the extent to which we've tested the system according to our mental models. Each test technique in the HTSM affords us a different way of modeling the product. Each technique is a heuristic—a fallible method for solving testing problems. No single technique can reveal all of the information that we seek about a system, but a variety of techniques will reveal more bugs—and more varieties of bugs. **{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. 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.*

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.

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 GUIdancer.

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 GUIdancer*
- *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