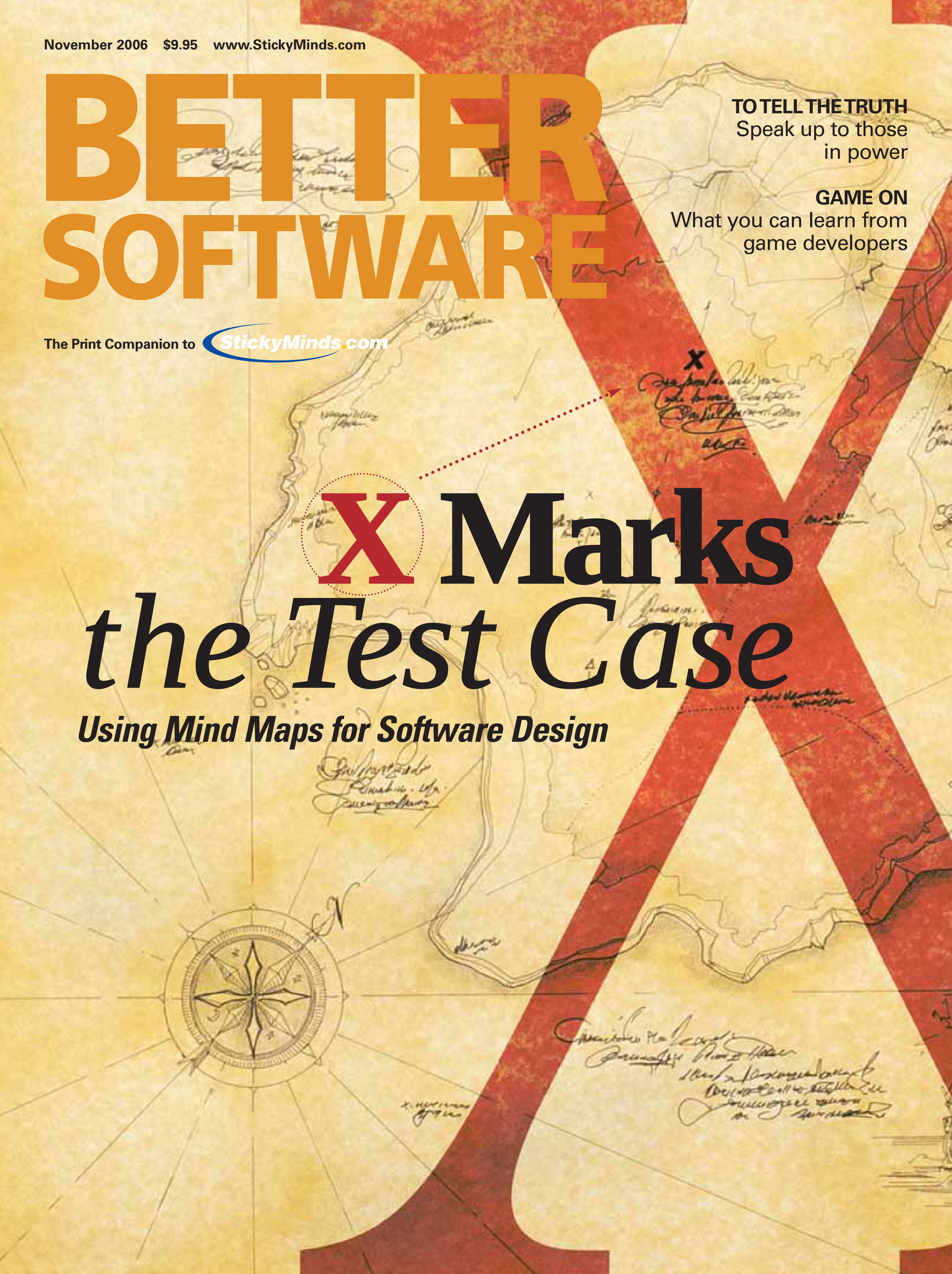
November 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

The Print Companion to StickyMinds.com



TO TELL THE TRUTH
Speak up to those in power

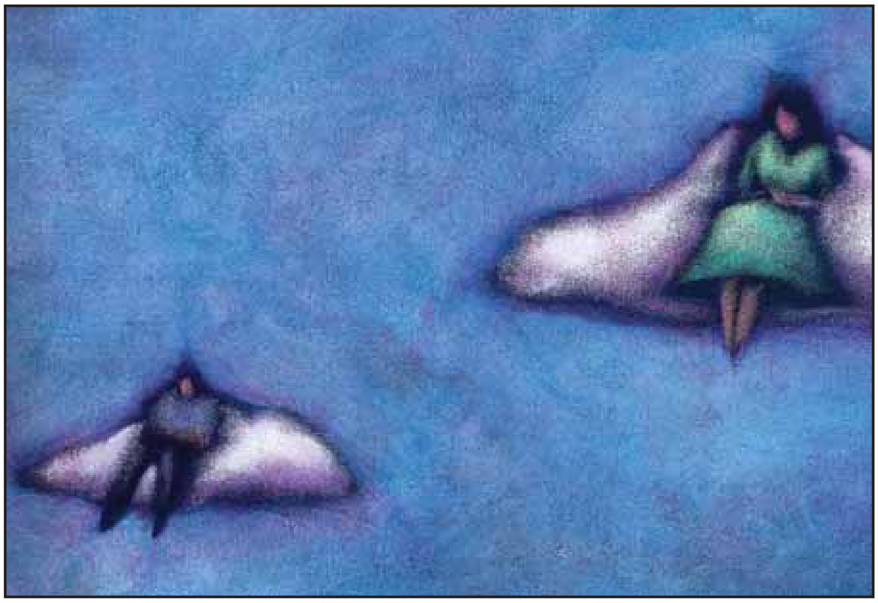
GAME ON
What you can learn from game developers

X Marks the Test Case

Using Mind Maps for Software Design

More Stress, Less Distress

by Michael Bolton


GETTY IMAGES

When Rapid Testers perform stress testing, we start from the notion that all systems have limits, and those limits—typically maximum capacities or minimum requirements—correspond to potential vulnerabilities. Then we find ways to violate those limits to their extremes: overwhelming the application, depriving it of some resource that it needs, or both.

Why test at the extremes? Why not just test at or near the limit? That's a reasonable thing to do if we're testing for specific, known risks, but in the Rapid Testing philosophy, stress testing is a fundamentally exploratory process—we often don't know in advance where or how the system will begin to fall apart. We may not have a specific prediction about the manifestation of the vulnerability, so we simply hypothesize that challenging situations, colossal amounts of data, or corrupt input will cause the system to react badly. In general, an overwhelmingly harsh test will expose vulnerabilities more quickly, with more dramatic symptoms. Still, stress testing requires us to pay special attention to the observation step and to be super-sensitive to the unexpected, using multiple quality criteria and oracles. Exclusively automatic stress tests are very risky, and completely automatic stress-testing oracles tend to spot only the problems that we anticipate. We also need to be prepared to spot unanticipated problems.

Some of our oracles—principles or mechanisms by which we recognize problems—will be driven by inference and our own observation: The application shouldn't crash, slow to a crawl, or write outside its client window, even if the specification is mute on those points. Other oracles will require probing the system with tools that observe otherwise invisible things. If our application writes files to disk, file, folder, or registry, corruption is a potential stress-test outcome. An integrity checker or probe is often easy to write, quick to run, and possibly useful as a support tool. I've seen ideas from testing tools move into the applications under test.

Lightweight tools also can help us flood the application with big data or transactions. The copy/paste and macro features in text editors often can be harnessed to produce a lot of data for quick tests. The Perlclip tool by Danny Faught and James Bach uses a Perl-like syntax to create character strings in patterns of your choosing on the clipboard. These strings can be fed into text fields in browsers or Web applications as quick tests or pasted into other testing tools (be careful that the tools themselves don't collapse under the weight of the data). Some of my most effective stress tests come from writing Perl and Ruby code for targeted tasks and probes that work below the GUI level of the application, and from using programs like JMeter or frameworks like WATIR at higher levels to simulate large user numbers and transaction volumes. None of these tools requires a huge investment in time or learning, and other testers are usually willing to provide support.

We can get some stress ideas from the Product Elements section of the Heuristic Test Strategy Model. We can compromise the program's structure by deleting or renaming a needed file (or setting the file's attributes to read-only), taking away the network or printer, putting the program into an unusual display mode, preventing access to the database, and so forth. We can use automation to start multiple instances of the program and then ask it to perform huge numbers of functions simultaneously, using overwhelming amounts of data. We can also pass the program incomplete, complex, or corrupt data. We can stress the program by running it on a constrained platform, with minimum system requirements in memory, disk space, video support, and so on; we'll test on a system barely equipped to do the job, and then we'll compromise the system further to see how the program reacts. We can simulate complex operations or deprive the program of resources by running many other applications concurrently. We can starve the program by making it wait for us or quickly force-feed it data, stressing the program with respect to time. With each of these ideas, we should see a graceful failure and an informative, helpful report from the program on the nature of the difficulty.

Stress testing is a technique we can use at any time during the development project, as long as someone is willing to hand us something to test. It's ideal

"When a tester pushes a system to extremes and finds a vulnerability, a common reaction is 'That could never happen.'"

when programmers can do some degree of stress testing at the unit level. Programmers might not wish to go to the same extremes as testers, and we might not want to run a full stress-test suite on every build of every component (it could cause the unit tests to become unacceptably long). Still, some stress testing at low levels can help us identify certain vulnerabilities more rapidly and easily.

When a tester pushes a system to extremes and finds a vulnerability, a common reaction is "That could never happen." Alas, the problem *has* happened; you may have to phrase it gently and dispassionately, but it is a fact. Another reaction is "No user would ever do that." There are too many possible users, too many motivations, and too many circumstances to say "never." Users can be malicious, like black hat hackers, identity thieves, or embezzlers. Even benign users might be hurried, clumsy, forgetful, bored, or unexpectedly creative in the use of the program. It would be a rare user who disconnected a network cable in the middle of a transaction—but my wireless modem does something like that on a regular basis.

Aside from being wrong, these reactions are also irrelevant from at least three different, related angles. First, there may be more than one way to trigger this vulnerability. Second, if the problem is not well understood, then its consequences won't be known either. As Richard Feynman said in his appendix to the Rogers Commission Report on the Challenger space shuttle accident, when something is not what the design expected, it's a warning that something is wrong. "The equipment is not operating as expected, and therefore there is a danger that it can operate with even wider deviations in this unexpected and not thoroughly understood way." When a system is in an unpredicted state, it's also in an *unpredictable* state.

Testing may focus on technology and ignore the human component of the system. The 2003 blackout in northeastern North America could have been avoided if the system operators had had sufficient clarity and confidence to intervene appropriately; software bugs and uncertainty undermined them. When there's an abnormality in the system, we probably want to get the user's attention. This means that the application needs to walk the tightrope between giving the user needed information and distracting, annoying, or panicking her. If we model a stressed-out user, we're likely to identify vulnerabilities that she'll trip over in the product.

Have mercy. The numbers and natures of potentially stressful conditions are enormous. Testers should work supportively and sympathetically with developers tasked with shoring up the program's vulnerabilities.

The measure of a system is not how it handles routine transactions, but how it deals with exceptional circumstances. **{end}**

Michael Bolton lives in Toronto and teaches rapid software testing—a methodology and course that he developed in collaboration with James Bach—in Canada, the United States, and other countries. He 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.*

Let's share stories: How do you push the boundaries of a system? What are some of the more creative approaches you've taken to stress testing? What have they revealed?

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

The Answer is Clear.

A) Testing, testing and more testing. With over 25 years of development experience, ACULIS delivers QA / Testing solutions with speed and precision.

B) Headquartered in Utah, ACULIS maintains state-of-the-art testing centers in the US, and off-shore in Mendoza, Argentina.

C) ACULIS provides comprehensive software development, testing, localization, and IT staffing solutions.

D) On-site, off-site, or off-shore, ACULIS global IT services can be maneuvered for each phase of your development project.

E) Contact ACULIS for a free consultation.

RIGHT NOW GET... 10% OFF any ACULIS Software Development Service

Some Restrictions Apply

Contact us for more info.

Software Development Service Solutions

Staff IT
Develop IT
Test IT
Globalize IT

ACULIS

801.377.5360 866.4ACULIS
WWW.ACULIS.COM