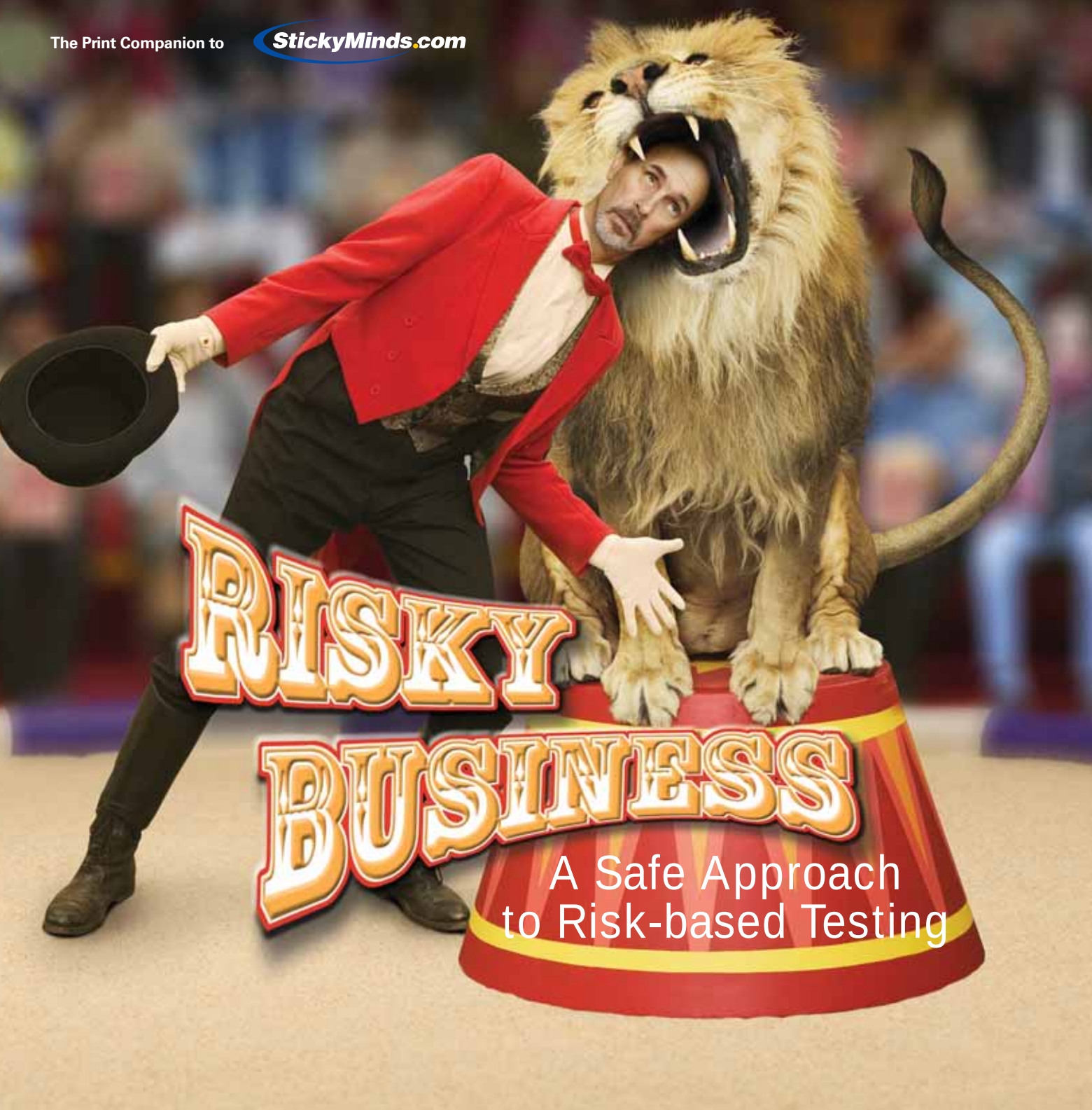
October 2006 $9.95 www.StickyMinds.com

BETTER SOFTWARE

OFFICE SPACE AS ART
Improve your team's surroundings and its output

GET CULTURED
Fourteen principles to better your products

The Print Companion to StickyMinds.com



RISKY BUSINESS

A Safe Approach to Risk-based Testing

Master of Your Domain

by Michael Bolton

GETTY IMAGES

Function testing is easy: Identify all functions in the program (and the parts of the platform upon which the program depends), and test each one. Domain testing is only a little harder: Identify anything in or around the program that can vary, determine sets of all valid and invalid instances of each value, and use at least one representative of each set in a test. But there are catches.

Like functions, the number of things that can vary about a program is large. The set of valid values for each variable is even larger, and the set of invalid values for any variable is infinite. Good domain testing also involves variables in combination with one another. Once you've tested every function—a huge number for all but the most trivial programs—dealing with a large number of infinities (and combinations of them) should be a minor step.

One risk in domain testing is that we're not talking about the same thing. Some use "domain" to describe the business and product's development context, as in Eric Evans' book *Domain Driven Design*. Traditional testing sources refer to the program domain as the data that the program accepts and processes. In mathematics, the domain is the set of all input values for a function.

"Domain testing" could mean any of these things. Identify and discuss differing interpretations to highlight potential ambiguities and to welcome and generate test ideas based on as many meanings as possible. To some degree, Rapid Testers embrace ambiguity: We can identify it and thereby can mitigate it and reduce risk, and it can help us think more expansively about test ideas. Consider a variety of possibilities; risk is buried beneath unconsidered possibilities.

In traditional parlance, domain testing involves identifying equivalence classes—sets of things that we expect the program to (mis)treat in the same way—making partitions between these classes, and identifying interesting elements within them. Rapid Testers call this "dividing and conquering the data"—not only data processed by the program but also data about operating platforms, peripheral devices, user profiles, and everything else surrounding the program. Domain analysis can make this huge task easier.

When modeling the product, we save time by using equivalence classes as shorthand. Saying "Internet Explorer versions 5.0 and above and compatible browsers" saves us from listing every supported browser—but be sure to ask, "Compatibility or equivalence with respect to *what*?" Two "compatible" browsers might have equivalent JavaScript interpreters but not equivalent handling of Cascading Style Sheets, different display resolutions, or malformed HTML. We save time by identifying classes that are equivalent with respect to some risk, oracle, or theory of error. So how do we choose what to test?

Within each equivalence class, there are subclasses of elements:

- *Boundary values* represent a limit or transition point between one equivalence class and another.
- *Best representatives* are most likely to reveal important bugs.
- *Typical values* are equivalent values that have a high probability of being used.
- *Convenient values* are easier to use or to test with.

Choosing an element with several of these attributes may allow us to perform fewer tests. If one element doesn't fulfill all of these attributes, then another might be better, or multiple tests might be necessary to address multiple theories of error. Don't try to get to them all—you can't. Domain testing is heuristic; it helps us to solve a problem, but it's fallible. Focus on risk, and use a diverse set of ideas to choose values.

Maximum and minimum acceptable values for input fields (and one more than the maximum and one less than the minimum) are worth testing because of risks like the developer's mistaking greater than (>) for greater than or equal to (>=), or equal to (==) for the assignment operator (=). Most tests of this kind focus on business-oriented boundaries, such as the maximum dollar amount for a given field, which indeed might be important.

However, there might also be invisible, internal boundaries, such as the limits of a (signed or unsigned) data type, over which the program treats the data differently. Good unit testing helps with that, but you may not be in a context in

which unit tests are trustworthy; try a few quick tests of values on either side of important powers-of-two boundaries: 256 (8 bits) or 32,767 (a signed 16-bit value). On my Windows XP system, Perl's Time::Local date-formatting module seems to display a valid date when fed 2,147,465,643, but 2,147,465,644 gives no output. Feeding the function 2,147,483,648 (the largest positive signed 32-bit value) helped me to find that unexpected boundary. (See the StickyNotes for more on data storage and display.) Rapid Testers usually start big; if the program doesn't constrain input properly, an outrageous value is most likely to expose the problem.

Good equivalence partitioning isn't just a simple division between ranges of valid and invalid *input* data. Two valid 16-bit integers can be added or multiplied to produce a result larger than a 16-bit integer. If the developer hasn't allocated enough space for the result, bizarre effects can follow; so consider using combinations of values, and force out-of-range output values, too.

Some compilers and languages handle data typing and overflow issues invisibly; others don't, so knowing the programming environment for the product can influence our view of the risks. On the other hand, we may know nothing at all about the development tools used to build parts of the platform. Platform testing—testing the things upon which our program depends but that aren't part of our current development project—can be partitioned for equivalence as well. Cover the territory better by running one set of tests on one platform and another set on another platform; then vary the platforms in later test cycles.

Since domain testing is about sets, read about elementary set theory. The empty set is a subset of any set, so entering no value at all or clearing out a provided default value often can reveal bugs.

Another subset of any set is the set itself. In a program I tested recently, merchants who accepted credit cards were classified by a four-digit category code. I had a reference table that identified the valid and invalid values, but the values weren't sequential and there was no pattern to the data that could be expressed as a set of boundaries. Rejecting a valid value, or accepting an invalid one, would lead to bad data in the database.

I did a couple of quick manual tests—typed a few alpha keys and held down the shift key while running my finger across the number keys. The program only beeped, so I inferred that only digits would be accepted. There were 3,000 valid values and 7,000 invalid ones but no evident boundaries. The most rapid approach in this case was to try all values using automation. I wrote a quick Perl script, did some other tests while the script ran in the background, and found exactly one bug in the table: The most recent addition was being rejected. "New ones vs. old ones" has since become an equivalence partitioning heuristic for me, and testing an entire set—when feasible—might be the most appropriate domain-testing technique. **{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.*

Sticky Notes

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

- Data storage and display
- More on Domain Testing

I love hearing stories about hidden or unexpected boundaries—where something challenges our preconceptions of what is a boundary and what isn't. Got any stories to share?

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