DevelopSense Newsletter *Volume 2, Number 2*

November 15, 2005

Michael Bolton
mb@developsense.com
http://www.developsense.com

Back issues: http://www.michaelbolton.net/newsletter/index.html
Blog: http://www.developsense.com/blog.html

Welcome!

Or more like, "Welcome Back.". Sorry I've been gone for so long.

Please note: I've sent this newsletter to you either because you asked me for it explicitly, or because I genuinely thought that you would be interested in it. If neither is the case, please accept my apologies and let me know by clicking here, or send a message to remove@developsense.com.

On the other hand, if you like this newsletter, ***please take a moment to forward it to friends or colleagues that you think might be interested.*** If you'd like to get on the list, please click here, or send a message to addme@developsense.com.

Your email address is just between you and me. I won't give your email address to anyone else, nor will I use it for any purpose other than to send you the newsletter and to correspond directly with you.

Your comments and feedback are very important to me, and I'd love to share them with the rest of the recipients of the letter. Please send them on to me at feedback@developsense.com.

What I've Been Up To

It's been maybe the busiest year of my life so far. Since I last sent out a newsletter, I've been teaching Rapid Software Testing in Sunnyvale, California; Mt. Laurel, New Jersey; Chennai, India; Utica, Michigan; and Sophia-Antipolis, France. There were presentations on the subject of Rapid Software Testing and Exploratory Testing in Ottawa, Ontario; Grand Rapids, Michigan; Wellington, New Zealand; Canberra, Australia; Boston, Massachusetts; and right here in Toronto. I attended Joe Rainberger's well-organized and well-attended XP Day Toronto in February, and Fiona Charles and I set up the first Toronto Workshop on Software Testing in June. I've also been writing a column for Better Software Magazine (formerly STQE), and the whole time I've been walking the walk, doing testing for a Canadian financial services organization. There's a seventeen month old daughter and a nine-year-old stepson at home, and there's a new kitchen installed this summer. Newsletter-wise, I've got a lot of catching up to do.

WTST 2005

In February, I participated in the Workshop on Teaching Software Testing, hosted by Cem Kaner and James Bach. It was an honour to be invited and to participate, and a pleasure to meet with some old friends and to make some new ones.

Some of the more interesting topics were in unexpected areas. Some of the highlights included discussions about concept mapping (which some people might have experienced as mind mapping); approaches to education and Bloom's taxonomy; and a vigourous set of discussions on how to incorporate software testing into computer science programs.

For me, the most difficult and painful parts of the session were also the most valuable. Some of the academicians presented several approaches to teaching testing that were simultaneously too much and too little—too much, in the sense that they typically covered a single, graph-based technique in some detail, and too little, in that their curricula only allocated a small percentage of classroom time. Moreover, the approaches they took to teaching testing appeared to miss key elements of testing: risk, coverage, oracles, and test activities. Fortunately, the academics responded to some harsh criticism with good will and determination to learn and teach more about testing in the wild, outside the classroom.

Crossing the Disciplines

In addition to the other stuff that I do, I'm the Program Chair for TASSQ, the Toronto Association of System and Software Quality. In February I was very pleased to introduce to the group Prof. Peter Sawchuk, who is an instructor at the Centre for Industrial Relations, and an Assistant Professor, in Sociology at the Ontario Institute for Studies in Education. He gave a presentation on the enormous changes to welfare reform in Ontario between 1995 and 2003, which was a mind-boggling project that cost the public in excess of CAD$500 million.

One of the goals of the system was to reduce welfare cheating, and yet the designers of the project eliminated human case workers and investigators, replacing them with a telephone call centre. According to Prof. Sawchuk, this was a mistake—case workers, upon visiting a home, could literally smell fraud, but a lot of information gets lost over the phone.

About half of the project's budget was dedicated to building a modern suite of software that could manage welfare delivery. A new government came to power in 2003, and in early 2004 it decided to raise welfare supplements by 3%. The system proved incapable of handling this change until, we, the public, coughed up another $10 million for software development and yet another $10 million for testing. Apparently the capability of processing a rate increase was not included in the requirements; and apparently, despite $260 million worth of software development, either the subject never came up or it came up and was rejected.

One could ask all sorts of questions about this boondoggle. I was curious about a project update for which $10,000,000 could be budgeted. If all that money went to salaries, it would pay 100 testers $100,000 each for a year. Assume ten project managers at $200,000 per year, and you're still left enough money for 80 testers. Yet this was not the bill for building the entire system, just the bill for testing one change. Remarkable.

Prof. Sawchuk certainly had lots of questions, and he answered a few of them too, but other questions still left him baffled after several years of study. He noted that the case workers themselves were completely left out of discussions on what the system—the whole system, not just the computer system—needed. He had few ideas on why intelligent people would make such a fundamental mistake, although he did suggest that the senior politicians and the bureaucrats—were acting out of ideology.

In addition to the software development aspects, he also talked about the social issues associated with big, revolutionary projects. These issues were very interesting to me. I think it's important for us testers to remember one of the principles of the Context Driven School of Software Testing: that the product is a solution, and if the problem isn't solved, the product doesn't work. As testers, do we remember to consider the social implications of the projects that we're testing? We don't make project decisions—those are up to the project's owner and managers—but it *is* our role to shine light on the darker corners of the project. How can we learn to do that with skill and grace, so that we're acting not only as competent and responsible technical workers, but also as good citizens?

Getting Jazzed About Oracles (II)

In traditional testing parlance, an oracle is something that provides a correct answer. W.E. Howden, who believes that he was the person who coined the term "test oracle"[1] provides the definition "any (often automated) means that provides information about the (correct) answer."[2] Some writers in the Mathematical and Factory Schools of software testing would say that an oracle provides a *predicted* outcome of a test. That's because those writers will tend to insist on tests being prepared in advance.

Exploratory testers prepare tests on the fly—exploratory testing is simultaneously planning and designing, executing tests, and learning. Based on the outcome of a test, we might choose immediately to design and execute a new test, but how do we decide correct or incorrect behaviour without a predicted result?

One strategy might be simply to observe a result and then run the oracle program. If that program is of sufficiently high speed, and the result sufficiently deterministic, it might not matter that the "right" answer came before or after the test. But I think there are more profound issues at work.

As you may remember from my last newsletter, I favour James Bach's definition of oracle: "a principle or mechanism by which we may recognize a problem". This expansive, inclusive definition allows us to extend the criteria upon which we can test. It's also much easier to see in James' definition that oracles are heuristic—fallible methods for solving a problem. An oracle in Howden's sense typically answers only one question at a time (although it might provide an answer far more quickly than a human might). Note also that Howden says "*the* correct answer", rather than "*a* correct answer"; the correctness of an answer can easily vary depending on context. Yet at any instant in the running of a program, we humans are capable of making dozens of observations of profoundly different kinds. We can use a computer program to assist us, but we can't make that program think, or question, or evaluate, or judge. We can't very well program computers to make esthetic judgments, nor can we ask the computer to make decisions for which the answer is neither good nor bad, but "good enough".

So last time, I promised that I would suggest some attributes that might strengthen or weaken your oracles in a given context. I emphasize *might* here; oracles are heuristic. As a context-

---

[1] http://www-cse.ucsd.edu/users/howden/

[2] W. E. Howden. "A Functional Approach to Program Testing and Analysis". *IEEE Transactions on Software Engineering*, 12:997-1005. Quoted in Boris Beizer, *Software Testing Techniques, 2nd Edition*, Coriolis Group, Scottsdale AZ, 2003

driven thinking exercise, try to consider a context in which each of the following attributes might be appropriate; then identify a context in which it might be inappropriate or dominated by another oracle.

Oracles, as heuristics, are vulnerable to questioning and contradiction. If your oracles are credible and persuasive to the people that matter in the project community, it will help people to recognize and acknowledge possible threats to the product. If the people that matter *don't* believe that the oracle is trustworthy within some important aspect of the project's context, the oracle will be ineffective at helping people to recognize a bug. Credibility depends on many of the attributes that follow. If your oracle is credible, it will help to make the case that there's a problem. If the oracle doesn't demonstrate the bug in a way that people can understand and relate, people may not acknowledge that there's a bug at all.

If your oracle is appropriately **precise**, it will generally be more credible, but precision can be *inappropriate* in two directions. For example, imagine a daily weather-logging program, designed to record temperatures within plus or minus a tenth of a degree for a certain location. An algorithmic oracle would probably be useful were it to provide precision consistent with the program. An oracle that reports only three gradations of temperature—"icy", "watery", or "steamy"—would probably too imprecise for comparison with the program. Meanwhile, an oracle that reports the temperature to three decimal points of precision might well be overkill[3]; if the oracle is *too* precise, there's likely to be some consequence. Perhaps determining the excess, useless precision takes more time than necessary; perhaps the data needs to be massaged to be compatible with the program under test; perhaps overly precise results could be considered nitpicking. Note, by the way, the difference between **precision** and **accuracy.** My watch says that it's 11:16:22.04; that's precise (down to hundredths of a second), but it's not accurate; the real time is closer to 11:10. Precision is all about the number of decimal places; accuracy is about how close we really are to some target—or oracle.

If your oracle is **consistent**, it is more likely to be respected. The oracle should typically be able to produce the same outputs, given the same inputs and the same preconditions. If the oracle doesn't supply answers that jibe with other predictions, your oracle could be discounted. That goes for strongly algorithmic and more heuristic oracles alike. Inconsistency weakens trust in the oracle; people use words like "flaky" to describe oracles with inconsistent mechanisms. At the same time, people use "flaky" to describe people with inconsistent *principles*, too.

On the other hand, context can change everything: three wristwatches that show the same time might be persuasive, but people will be more inclined to trust the **authoritative** results from one of the National Institute of Standards and Technology's time servers.

If your oracle is **flexible**, it will tend be useful in more circumstances, or over a longer time. Oracles that can't adapt will end up on the scrap heap more quickly. That's not necessarily a bad thing; heuristics are designed to be tossed out as soon as they've outlived their usefulness. But over the long haul, you'll tend to get a lot of use out of oracles that are generalized and adaptable. For example, an oracle for a financial program could provide accurate and useful valuations of net worth based on a specific set of test inputs. However, if the oracle can't deal

[3] In Jerry Weinberg's Quality Software Management Vol. 2, First Order Measurement, he relates someone's story of the Army's approach to precision: measure with a micrometer, mark with a chalk line, cut with an axe.

with multiple currencies, it might not be so useful for testing international versions of the program.

If your oracle is **timely**; it will produce results quickly enough that they're useful to the test effort. A fantastically accurate, precise, consistent, flexible oracle that takes two hours to produce a result won't be helpful when you have ten minutes to test and report.

If your oracle is **explicable**, it will be easier to understand why oracle and the application under test disagree. If no one understands how the oracle works, they won't understand why the test results differ, and that slows down the process of finding and resolving bugs.

Oracles come in three basic flavours, according to James. There's

- Reference – something that some authority has written and to which we could refer. That includes things like requirements documents, specifications, standards. References can contain references to other artifacts.

- Conference – some discussion or conversation, in which someone makes an assertion about a product. Conferences could happen in meetings, around coffee coolers, in a small office. The key to success when you're using an oracle based on conference is that the person making the claim must have the authority to make the claim credibly.

- Inference – some mental construct whereby a tester observes a problem, but about which neither conference nor reference has provided specific information. For example, one doesn't need a specification to see that a program shouldn't crash during operation; shouldn't reset or lose track of data that should be preserved from one moment to the next. Inference suggests that a tester, at least from time to time, can test with her eyes open, rather than buried in some piece of documentation. Good use of inference depends on tester experience, skill, or expertise, but there's nothing intrinsically wrong with that.

Reference, inference, and conference work like a game of Paper, Rock, Scissors. A mistake in a reference can be exposed by a tester's inference, and confirmed by conference with someone who matters; a suggestion from a developer that something should work a certain way might also be discounted by a tester, whose inference might be backed by a reference; and a mistaken inference can be overruled by a conference with an authority who points the tester to a credible reference. By taking all three kinds of sources for information into account, we can ask more questions and get more answers about the product under test, and by understanding which source is credible, we can ourselves be more credible. I'll have more to say about this in future newsletters.

Coming Soon: Rapid Testing in Toronto

In late November, James Bach and I will be teaching two one-day introductions to Rapid Software Testing. The first session is in Kitchener-Waterloo, ON, for the South Western Ontario Software Quality Association, on November 28. That one is sold out already. The second is in Toronto, ON, for the Toronto Association of System and Software Quality, on November 29. As I write, there are still a few seats left for that. This is a great introduction to the principles that we teach through the full, three-day, Rapid Software Testing course.

In addition, James will be giving an address to TASSQ in Toronto on Tuesday evening. The topic is “Skilled Testers and their Enemies”.

You can find details both events; please see the TASSQ Web site at http://www.tassq.org.

That’s it for now—see you in the next issue. I’ll try not to take so long next time.

---Michael B.