Traipsing through the QA Tools Desert

Terry Coatta, Silicon Chalk

The Jeremiahs of the software world are out there lamenting, “Software is buggy and insecure!” Like the biblical prophet who bemoaned the wickedness of his people, these malcontents tell us we must repent and change our ways. But as someone involved in building commercial software, I’m thinking to myself, “I don’t need to repent. I do care about software quality.” Even so, I know that I have transgressed. I have shipped software that has bugs in it. Why did I do it? Why can’t I ship perfect software all the time?

Like anything in life, the reasons are complex, but a big factor is just how hard it is to do QA (quality assurance). You can spend days or even weeks looking for a single bug, and eventually you get to the point where it doesn’t make sense to hold up shipping the product looking for a bug that it seems you may never find.

At moments like this, when I seem to be losing the battle against the bugs, I get really annoyed at the time we must spend building tools that should have been part of the basic development infrastructure. Some folks might accuse me of being a whiner. “Look, you’ve got excellent tools such as Purify and BoundsChecker. Stop your sniveling and start producing better software,” they say. I love programs like that, don’t get me wrong. But these aren’t QA tools to me. If your organization is like mine, you’ll find the developers using these tools, not the QA folks.

Let’s step back a moment and ask what happens in QA, because that’s really the only way that we can understand what tools are needed. QA lives at the interface between requirements and the reality of what the software does. That means we care about transforming requirements into test cases, running the software to execute those tests, tracking the results, and communicating the results back to the developers.

Every one of these is a place where software can help to automate processes or improve efficiency. It’s not as though it’s a complete desert. There are lots of good defect-tracking systems, both open source and commercial. Tools to help represent and execute tests are starting to appear (<http://www.eclipse.org/test-and-performance/faq.html> and <http://www.junit.org/index.htm>). There are even some standards that might help create a more com-

Who’s really

TO BLAME FOR
BUGGY CODE?

ponent-like marketplace for testing tools (<http://www.omg.org/cgi-bin/doc?ptc/2004-04-02>).

But the purveyors of development tools and operating systems could easily add some pretty obvious functionality to help the situation. If you’re doing QA for a living, you probably run into it every day, and, if you’re like me, you find yourself muttering dark and disturbing oaths because obvious, easy-to-implement, and incredibly useful bits of functionality are just not there. And there’s no sign that they’re going to be arriving anytime soon.

How much time do you spend trying to deterministically reproduce problems so that you can avoid looking lame as you’re explaining to a developer: “I swear the bug was there yesterday, but I just can’t seem to get it to happen today.” Consider the number of bugs that have shipped because they could not be reproduced reliably enough for development to track down and fix them.

The only way of solving this problem is through programmatic driving of the user interface (UI). Where I work, we build for Microsoft operating systems. You might expect that the development environment would include a standard test harness that would allow you to easily build a test script, execute it, and verify results. After all, the operating system clearly knows about the UI layout, knows how to drive the UI, and can observe the results. Sadly, there is no such thing. At first, we just tried to do more testing. We made sure that all of our test machines had development environments, so that if a bug occurred, we could grab a developer and get to work on it right away. Even so, it became pretty clear we were losing the war. We looked at commercial software that allowed you to record mouse movements and operations, but the resulting tests were too fragile. If screen layouts changed, the corresponding tests were invalidated.

Even though it seemed as if we were investing more effort in testing tools than in the actual software we produced, we decided to build a decent test harness. We added code to the product so that it would register all UI components with a central registry. The registry helps

Continued on page 63

Continued from page 64

preserve an abstract notion of the product's user interface that is somewhat independent of the actual layout of the UI components on the screen. We designed and built a compiler for a UI scripting language that can drive the application externally. We added a generic interception mechanism so that we could trace application activity in response to UI stimuli. This system has helped deterministically reproduce errors that had previously eluded us.

We really can produce better software now. But every piece of this system is something that should have been a basic part of the development environment. The operating system already has a registry of every UI component, so why did we have to build another one? The operating system already has mechanisms to drive the UI, so why did we have to build more? The operating system knows when components call one another, so why did we need to build an interception mechanism? Sadly, the answer appears to be that testing simply was not on the minds of the people designing and building these systems.

It's not as if this is just a small omission in an otherwise robust set of tools supplied by the vendors. Rather, it seems like the reverse. Consider the task of supplying meaningful data to the developers about the state of the application when a bug is found: state traces that follow

logical threads of control, dependency analysis for deadlocks, and integrated flow of control analysis linking different underlying styles of interaction (such as Windows messages and function calls)—to say nothing of QA that involves testing an application that is distributed over a number of machines. Most of these would not be that challenging for the vendors to provide.

So what's stopping them? I think the basic problem is that QA is not as sexy as coding (to the extent that anything within the field of software development can be considered sexy). Developers tend to develop for developers. So here's a call to action: QA folks of the world unite! Let's demand the tools that are our due. Q

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

TERRY COATTA is a member of the *ACM Queue* editorial advisory board. He is the vice president of development at Silicon Chalk, which is creating realtime collaborative software for use in higher education. Prior to that he was the director of development for distributed systems at Open Text Corporation. He has a Ph.D. in computer science from the University of British Columbia.

© 2005 ACM 1542-7730/05/0200