Bonding Over Bugs: How to use defects to bring developers and testers closer together.

Advice by Michael Kelly

First published: MARCH 07, 2005 (COMPUTERWORLD)

It's safe to say that developers and testers are guided by different motives, pressures and perspectives. Developers are motivated to complete code quickly and accurately and move on to the next problem. Testers are required to find and report problems within the system quickly. Someone should be urging developers to improve the quality of the code they create or modify, but that often takes a back seat to "more code, faster." And someone should be inspiring testers to find more-meaningful bugs, but I rarely see that happening.

In the course of a project, the more defects testers find and submit to developers, the less time developers have to work on them, and communication may break down. Members of the project team commonly start to rely on entering short descriptions and comments into a defect-tracking system as a means of communication. This often leads to misunderstandings and unnecessary tension between the two groups. I have found some techniques that can improve communication between testers and developers and help them fix problems faster.

Share Automated Tests Between Teams

Test teams often develop system-level test scripts using commercial tools, and developers frequently design unit-level test scripts using open-source frameworks. If your project team isn't doing either, ask them why. There are plenty of good reasons not to use automation, but make sure the teams have at least considered it. If either team *has* automated tests, get them to start sharing.

Sharing test scripts gets everyone on the teams using the same tools and languages. That makes developers more likely to offer improvements to the scripts and testers more inclined to offer advice on data selection and common test patterns. Sharing test scripts also minimizes redundancy and typically leads to increased test coverage. The more testers and developers collaborate, the more powerful both their scripts become. Microsoft Corp., IBM and others have excellent integrated development environments built for this type of collaboration.

Distribute the Ability to Execute Smoke Tests

Every time developers compile code (often referred to as a "build"), there's the potential for something to go wrong. To detect a bad build early, it's helpful to create a series of preliminary tests -- commonly called smoke tests -- that exercise the system from end to end. A smoke test doesn't have to be exhaustive, but it should be capable of exposing major problems. If the build fails the smoke test, the developers will probably need to go back to the code to debug and find the problem.

If you don't have a smoke test, create one. If it's not automated, automate it. Automated smoke tests are particularly powerful for the following reasons:

• They're used often, possibly many times a day.

• They provide meaningful information, such as whether the system is at an acceptable state for testing and all services are up -- or not.

• They provide feedback quickly, typically in minutes.

• They're easy to execute and distribute.

The easiest way to ensure that a smoke test is executed is to include it in the build process. Make the smoke test available to both testers and developers through a central interface such as a project Web site or a test management tool. Not only does this get everyone using the same tools, but it can also get developers and testers collaborating on script development and maintenance.

Perform Runtime Analysis Together

I've found that sharing runtime analysis is one of the most effective ways to increase developer/tester communication.

Runtime analysis is just what it sounds like: an analysis of the code as it's executing. It can provide information on things like execution paths, code coverage, memory errors and leaks, performance and bottlenecks, and threading problems.

For example, on one project, we had a problem with pages taking more than 60 seconds to load. We ran numerous performance tests and couldn't isolate the problem. Then, using a runtime analysis technique, the testing team found that a call was being executed 4 million times when a page loaded. Armed with that information, an architect fixed the problem the next day.

There are many very good open-source and commercial runtime analysis tools, but it's still a difficult and thankless job. I've found that the most effective way to make sure that runtime analysis gets done is to have testers do it. Your testers don't need to become runtime analysis experts. They just need to learn the basics about some tools, learn a little about the problems common to the technologies they're testing and find some time to actually do the testing. As the testers begin to discover problems, developers will begin to try to prevent those problems -- and that will require them to use the tools themselves.

As a tester, I like to show a developer what I've found with my limited runtime analysis. When I do so, the developer no longer sees me as a technology-blind tester who doesn't know anything about development, and he'll likely be interested in helping me understand what I'm seeing. Once a developer knows that a tester has the desire and the aptitude to learn, he typically is willing to spend time helping the tester understand the applicable

technologies. From the developer's point of view, explaining the technologies once early in the project saves him from having to answer many small questions later. At the very least, the tester gains a basic understanding from which to ask smarter and more meaningful questions.

When developers and testers work together with a runtime analysis tool, testers can share information on the risks and long-term effects of not fixing problems. Developers can educate testers on the technical aspects of the application technology and project environment. Together, they can uncover and refine performance requirements while learning new skills.

Use Log Files to Isolate Problems

A simple technique for capturing bugs and debugging is to leverage log files. These are the files developers create at runtime that contain information about things like server and software activity and performance, as well as any problems that may be occurring. Often, when a problem happens behind the scenes of an application, it won't manifest on the user interface. For example, most Java exceptions don't appear on-screen. But if developers give testers access to the execution log files for the application, the testers can use scripts to parse through the log files, looking for abnormalities and exceptions. Once developers know what the testers are looking for, they may be more willing to take the time to write this information to the log files in a common format for testers to parse.

Use Defect-Tracking Systems Effectively

Your project team probably uses some form of automated defect-tracking system. Developers should tell testers what specific information in a defect report or "ticket" is most helpful. This enables testers to provide the right type and amount of information, such as screenshots, source code, steps taken or a script test case that can reproduce the bug, as well as any relevant log files.

Testers and developers should also work out a defect-prioritization scheme. Without one, developers may miss serious problems while sorting through lots of reports of little bugs that the customer is unlikely to encounter. By prioritizing defects, you ensure that critical bugs get fixed immediately and small problems get attention when time is available.

Creating good software requires a partnership between testers and developers. Most developers and testers want to help in any way they can, assuming that they're given time to do so. It's up to each group to let the others know what they need, and it's up to you, as manager, to make sure they have that time.

Sidebar: Face to Face

Bug-tracking tools are great for organizing software development issues and managing what goes into each release, but by removing the need for face-to-face contact, they can become a barrier to communication between developers and testers. However necessary

these tools may be, it's important to understand that they can't replace more-effective means of communication.

There's no substitute for talking face to face. Asking questions, reading body language and building rapport all help to create open and effective communication. When I encounter a possible defect, I often show the problem to a developer before I write the defect ticket. That strategy helps in the following ways:

• I may find that the developer is already aware of the problem.

• The developer can confirm that the problem is not caused by some setting or incorrect configuration; it's actually a bug.

• The developer sees the bug, so it's compelling.

• The developer can reproduce the problem more easily.

• The developer can request inclusion of additional information on the ticket to help jog his memory.

• The developer may be able to specify the person best suited to work on this problem.

• By watching the developer interact with the bug, I may discover other areas I need to test that might not have occurred to me otherwise.

Make sure your project environment is organized so that this type of interaction can take place. Think about where the testers sit in relation to the developers. Are they within walking distance? They should be.

Michael Kelly is a testing consultant at Fusion Alliance in Indianapolis. Contact him at Mike@MichaelDKelly.com.