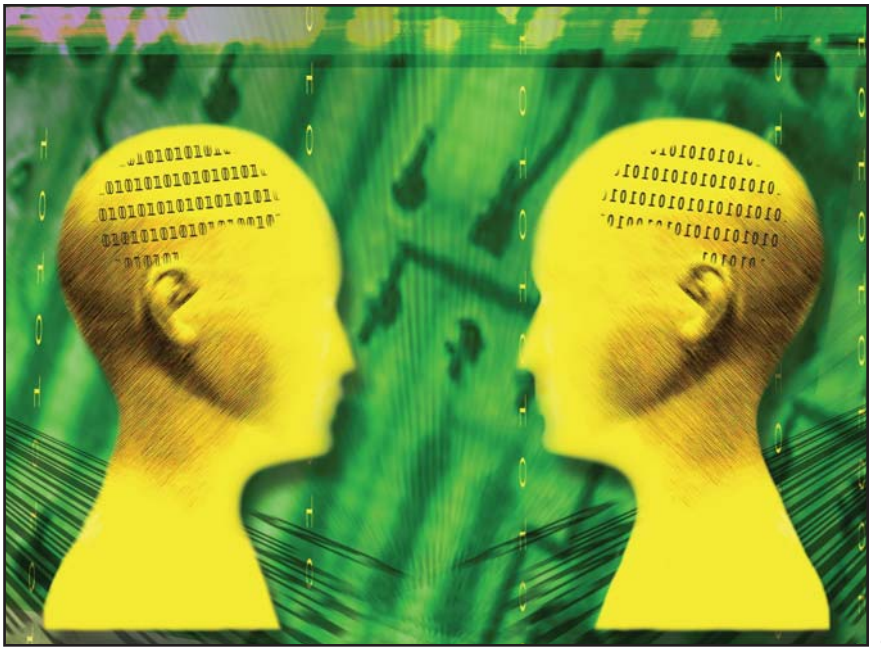
How to Win Friends and Automate Testing

by Michael Kelly

Due to different motivators and differing time pressures, communication between testers and developers can easily break down. It's often easy for a tester to think of a simple test the developer could have run before releasing the code, a simple test which would have found a problem—without appreciating the time pressures imposed on the developer. And it's easy for developers to see an overwhelming number of low-value or non-meaningful defects reported—without appreciating the expectations and metrics management measures against the testers. Use the following techniques to increase communication and to bring closer together the testers and developers on your project. These tips get you working side by side on problems, rather then tossing problems over some invisible wall.



Getty Images

Share Test Scripts with Developers

Problems encountered by test scripts can be lengthy or difficult to reproduce. I've submitted defects that took hours to reproduce manually because of lengthy setup.

If you make all of your test scripts available to the entire team, however, developers can look at the script code and the script logs, rerun the scripts and watch them execute, rerun them in local environments with debug information written to logs, or rerun them in conjunction with other tools.

To make this strategy most effective, reference your scripts in the submitted defect list. The development team can run the scripts without checking with you first, thus removing a manual step.

In addition to sharing the test scripts, provide the development team with a remote automated test-script execution box. Most enterprise tools allow for distributed test execution. If you provide one of your test lab machines to run your scripts, developers can execute the tests they need while simultaneously using their own computers to keep developing. This technique allows developers to work on your problem. Without the testing box, the developers might not be able to run a full test due to time and equipment constraints.

Sharing test scripts with developers also enables everyone on the team to work with the same tools used to develop the scripts. When team members use the same tools, a likely side effect is that developers take the time to offer improvements to the scripts. After running one of my scripts, a developer once told me that he already had a unit test script that did something similar "behind the GUI." Together we reviewed both scripts and ultimately transferred the data from my regression script to work with his unit test script, which executed in a fraction of the time of my regression script and provided easier-to-read results. The more feedback you get from developers on your scripts, the more powerful your scripts will become.

Make Smoke Tests Available to Everyone

Every time a developer, integrator, or build-master creates a build, there's potential for something to go wrong. Something is left out, a file doesn't end up where it is supposed to be, the wrong version is compiled, and so on. When something goes wrong, testers waste their time discovering that the build isn't stable enough to be tested, and programmers get disrupted when they are called back from their next task to fix it. The solution is to create a series of tests that exercises the entire system from end to end and to make it part of the build process. These tests, taken as a whole, are commonly called smoke tests. A smoke test doesn't have to be exhaustive, but it should be capable of exposing major problems.

Make the smoke test available to both testers and developers by using a central interface such as a project website or a test-management tool. If everyone has the ability to execute the smoke test and the results are simple to interpret,

testers won't be pressured to provide this service for everyone else on the team. Getting team members other than testers to run the smoke test may take a little patience and prompting the first couple of times, but after that, most people will prefer not relying on someone else for such a simple task.

Perform Runtime Analysis Together

Runtime analysis provides information on items such as code coverage, memory utilization and leaks, and execution performance. I've never worked with a developer who was actually tasked with performing runtime analysis. The developer was always doing it to solve a problem discovered by someone else. I've found that the most effective way to ensure that runtime analysis gets done is to start performing the analysis myself. As a tester, you don't need to become a runtime analysis expert; all you need to do is learn the basics—learn a little about the technologies you're testing (common problems, bottlenecks, and performance problems) and find some time to actually do some testing.

Of all the techniques described in this article, I've found runtime analysis the most effective at increasing developer/tester communication. Once you find something (or even if you only think you may have found something), you should show a developer what you have. Suddenly, you're no longer a technology-blind tester who doesn't know anything about development. Once a developer knows that a tester has the desire and the aptitude to learn, the developer typically is willing to spend as much time as available helping the tester to 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 on when time pressures are greater. At the very least, the tester will have a basic understanding from which to ask smarter and more meaningful questions.

At the same time the developer is helping the tester, the developer may in turn look to the tester for help in learning the testing tools. This is an opportunity for the tester to share information on the possible risks and long-term effects of the problems found, if they're not fixed immediately. Together, tester and developer uncover and refine performance requirements and simultaneously learn new skills. This technique gets everyone working together by leveraging tools that both teams can share—and some tools specific to each team.

Use Log Files to Isolate Problems

As I was developing new tests for a Java Web application on a past project, I was executing a "view source" in IE. I was surprised when I stumbled across the record of a Java exception in one of the pages. I looked at the browser again, but on the GUI level there was no indication that there was a problem with the software. Because of what I'd seen, I augmented our existing automation framework to look in the page source for exceptions every time a new page was loaded. Once the developers heard I had started parsing the page source, they started to log more information there as a standard practice. Once that was running smoothly, one of the developers started showing me the Web server logs for the application. Together we developed a set of scripts that would parse those logs at the end of any given test run. We then saw the same effect. Once developers knew we were checking these logs at runtime, they again started to log more information for us. The simple technique of leveraging log files as a means of capturing bugs (and assisting in debugging) led us to an increase in the testability of the application under test.

Getting the Most Out of Your New Friendship

My experience has been that as you begin working more closely with members of the development team you will notice that they are just as interested in learning new tools and skills as you are. Use these opportunities to build relationships that will support you as you try something new. For example, if your organization has just purchased a new testing tool that requires knowledge of Java and you only know a scripting language, use your new relationship to get help and guidance as you take on the new challenge. Be sure you are offering similar help in return. I recently worked with a developer while he was doing component testing. As he developed the initial JUnit scripts, I followed along behind him and extended the scripts to test boundary value conditions and equivalence classes. As he reviewed them, we were able to have a discussion about the techniques I was using, and I know that his future unit tests will probably include these types of tests. Developing these relationships will be difficult at first, but the long-term benefits are more than worth it. **{end}**

My experience has been that as you begin working more closely with members of the development team you will notice that they are just as interested in learning new tools and skills as you are.

Michael Kelly is a senior consultant for Fusion Alliance. With experience in software development and testing, Mike writes about and speaks on software testing. Mike is currently serving as the program director for the Indianapolis Quality Assurance Association and membership chair for the Association for Software Testing. You can reach Mike by email at Mike@MichaelDKelly.com.