How Effective is Your Test Automation?

Date: May 6, 2005 By Michael Kelly.

First published on informIT.com

Do you know, for sure, absolutely, that your automated test is testing what you think it's testing? Probably not, now that we've introduced some doubt. Mike Kelly explains how to get from "I think it's working" to "Yes, it's working correctly."

How many times have you been asked, "How do you know that automated test is actually testing what you think it is?" I've been asked that question a lot, and I have even asked it myself on occasion. It's one of the classic issues with test automation: If no human is involved, how do you *really* know that your tests are testing what you told them to test? What if there's a bug in your test code or in the test tool? What if the test wasn't implemented properly? What if a change in the test code in one place causes code to fail in another? Maybe the test code is outdated due to changes in the application and is no longer testing anything.

You get the point.

This article takes two approaches to these questions. First, we'll look at some methods of validating that a test is in fact testing what it's supposed to be testing. Then we'll consider what you can do if you look under the hood and find problems, and how some of these issues might be avoided. We'll focus on traditional scripted regression tests and basic performance tests.

Making Sure That Your Tests Are Actually Testing

I know of a couple of ways to validate automated tests. This is by no means an exhaustive list; these are just methods I've used in the past that have worked (or failed) for me. They may not be practical in your context, or they may not work for your software, but it's at least a place to start.

Manual Reviews

Option 1, the good old-fashioned code review. In a review, an automated test (functional or performance) is examined for accuracy by walking through the code and the log files, and by actually watching the test execute. You might even run the test in debug mode (if that's an option with your tool) and step through the code as it executes. This technique gives you the most visibility into what the script is actually doing, and it also might be easier then trying to read the code without really knowing what it's supposed to be doing.

Manual reviews require a certain level of familiarity with the automation tools, languages, any frameworks developed, and the intention of the test script being reviewed. This may not be an easy set of knowledge to come across. And if you do have staff who can perform the review, most likely they won't have time to do it because they're responsible for implementing *new* tests for the project team. In addition, this is a slow (and sometimes painful) process, which requires a significant investment of person-hours in order to gain any real confidence in the suite of tests.

Introducing Changes to the System Under Test

Another approach you can try is to introduce specific changes into the system under test (sometimes called *error seeding*). For example, if I know I have a series of automated tests that test the search feature on my web site, I might change the order in which results are returned and then run the scripts to ensure that they fail when they test that feature. With this approach, if the scripts pass, they're not testing what they should be testing. For performance tests, I might reconfigure my web server and execute the tests again, looking for change in the performance results.

While giving you better coverage than manual reviews, this method is not as reliable. In the examples I just described, some tests might fail with a different order on search results, but some tests might still pass because they don't check order but rather boundary conditions on the search field, or some other aspect that's not related to results. For the performance-testing example, you may change a server setting that has no noticeable impact to system performance; even though you think it might (I've witnessed this). This means you still need to switch to manual reviews for those tests that passed when you thought they should have failed.

In addition, now you're not taking just one person off normal project tasks for review, but several. Someone needs to add bugs to the code. Someone needs to build and deploy it (let's not talk about configuration-management issues). Someone needs to adjust the web server. Then you still have the tester, who needs to execute and validate the tests. While this is probably faster than manual reviews, it's most likely much more costly and a lot less reliable.

Spinning Off a Short Side Project To Verify Something

In this approach, you simply validate your automated tests by duplicating them using some other tool or implementation model. This is more of a macro approach to the problem, while the other two have been more micro in nature. I've only seen this done with performance tests, but I can imagine a couple of scenarios where it might make sense to try it for functional automation. This method is probably the most costly, as it's a clear duplication of work that has already been done, but sometimes this is the fastest way to gain confidence in something.

In his two-part series on performance tool comparisons (Part 1, Part 2), Suresh Nageswaran shows an example of what this might look like. In one of my current projects, we're trying to do a similar exercise by verifying page-load times using both the Mercury and Rational performance test tools. Often, you only need to duplicate some small portion of the overall set of tests and then work under the assumption that your implementation model is correct. The drawbacks to this approach are the cost of ownership for multiple tools and developing equivalent skills related to the tools and/or implementation models.

Overlapping Your Testing

This approach is probably my favorite, but it's difficult to find a context in which it can be implemented. In this approach, you simply attempt to audit one type of testing (in this case, functional or performance test automation) by using another type of testing. For example, I like to audit my performance tests whenever I run my functional tests. You can read about a specific example of this in my article "Gathering Performance Information While Executing Everyday Automated Tests." Similarly, you can sometimes do the same thing by having your performance tests validate some sort of functional requirement (but this can be a bit more difficult and may be less practical). More traditionally, for projects in which I've planned our testing, I've tried to build in a small overlap on the manual test effort and the automated test effort.

The costs for this approach are typically the lowest, as you're just tacking an extra task or two onto an existing effort. This means that no one stops what they're doing. It's your classic case of scope creep, only now you're attempting to use it to your advantage in testing. The disadvantage to this approach is that you now have all the problems related to scope creep. The original task now takes a little longer to finish, and someone has to use and maintain the overlapped functionality you developed. Don't get too carried away with overlapping, as each time you do it you're introducing inefficiency into the test project.

What To Do If You Find Problems

When you encounter a problem with your automated tests, you need to consider several factors. First, is this just an isolated incident (one faulty test), or is it a larger issue that affects a series of automated tests? If it's isolated, you're probably safe just fixing the test and going on your merry way. However, if it's a bigger issue, you need a bigger solution. The first thing you need to do is ask if you really need these tests any longer. If you do need them, ask yourself whether they should be automated or manual. If you decide that you do need them *and* that they should remain automated, you then need to ask whether this is the best way for these tests to be implemented (going back to the implementation models I mentioned earlier).

Do We Still Need These Tests?

James Bach's minefield analogy illustrates a basic principle of testing: It's probably more meaningful and more valuable to execute a new test rather than an old test. "The minefield heuristic is saying that it's better to try to do something you haven't yet done, than to do something you already have done." When you look at the tests you need to update, ask yourself whether you would still create them today? Are they still interesting, or has the risk been significantly diminished to the point where we just don't need to invest any longer in those tests? Don't work under the assumption that a test is valuable just because someone once found it worth writing.

If you decide that the test is worth repeating, look for overlap between that test and other tests that currently work. If the feature is tested somewhere else (either manual or automated), you may not want to invest the effort to update this instance of the test. Earlier, when I mentioned overlapping your testing, I indicated that this isn't operating at optimal efficiency. Here's a chance to make up for some of that inefficiency. If you have other tests that test this functionality, and you know that those tests work, lose the tests that aren't working and don't look back.

Should These Tests Even *Be* Automated?

Before doing any sort of in-depth analysis, consider whether the test is even appropriate for automation. If it's a usability test, for example, I would seriously question someone's ability to automate it. Similarly, if it's a test involving special hardware (fingerprint readers, for example) or a manual process (such as viewing a printed piece of paper for correctness), you might question the original wisdom in automating the test. Not that it's impossible to automate these things—often it's just not practical.

I don't want to reinvent the wheel here. If you haven't read Brian Marick's "When Should a Test Be Automated?" stop reading and do it now. This is a must-read for anyone doing automated testing. It's probably one of the most complete and well-stated papers on a very difficult topic. In the paper, Marick addresses the costs of automation versus manual tests, the lifespan of any given test, and the ability for any given test to find additional bugs.

He draws two conclusions worth restating here:

- "The cost of automating a test is best measured by the number of manual tests it prevents you from running and the bugs it will therefore cause you to miss."
- "A test is designed for a particular purpose: to see if some aspects of one or more features work. When an automated test that's rerun finds bugs, you should expect it to find ones that seem to have nothing to do with the test's original purpose. Much of the value of an automated test lies in how well it can do that."

In the context of this article, the first point is the more important, as we're concerned with the cost of *fixing* an automated script; thus, your cost is the number of manual tests and bugs missed the first time around plus the number of manual tests and bugs missed the second time around when you fix the test. The second point addresses the ability of the test to find new, unexpected bugs, which it may or may not currently be doing, even though it's not testing what you initially designed it to test or thought it was testing.

Is There a Better Way To Automate These Tests?

Chances are that if you're doing automated testing, you can be doing it better. We all can. As with any testing activity, there's always a way we can do something better. In my experience, automation seems to be a low-hanging fruit with which many organizations can make immediate improvements. If you're going to invest in automated testing, the least you can do is the ensure that you're getting the most bang for your buck.

I'll close this article with my short list of resources for ways to make your automation more effective. Remember, you got to this point because you found an automated test that isn't testing what you thought it should be testing. You then asked whether you still needed the test, and decided that you did. After that, you asked whether the test should remain an automated test, and again you thought it should. So by this point you know that you did something wrong the first time around. These resources will help to ensure that you get it right the second time.

If your organization is un familiar with some of the "classic" frameworks for test automation, take a look at the following resources:

- "Deconstructing GUI Test Automation," by Bret Pettichord
- "Improving the Maintainability of Automated Test Suites," by Cem Kaner
- "Choosing a test automation framework," by Mike Kelly

If you've implemented some of these frameworks, but are interested in even more sophistication and new ways to leverage your existing tests or reduce their maintenance costs even further, the following resources might get you going down a new path:

- "Bypassing the GUI," by Brian Marick
- "Model-Based Testing: Not for Dummies," by Jeff Feldstein (pages 18–23)
- "High Volume Test Automation," by Cem Kaner, Walter P. Bond, and Pat McGee

Finally, perhaps your organization just needs a new way of thinking about automation. If you think that might be the case, these articles might be helpful:

- "Agile Test Automation," by James Bach
- "The Ongoing Revolution in Software Testing," by Cem Kaner
- "Determining the Root Cause of Script Failures," by Scott Barber

Michael Kelly is a senior consultant for Fusion Alliance, with experience in software development and testing. He is currently serving as the Program Director for the Indianapolis Quality Assurance Association and the Membership Chair for the Association for Software Testing. You can reach Mike by email at Mike@MichaelDKelly.com.