SUBJECT7
QUALITY IS A CULTURE

Avoiding False Positives and Negatives in UI Test Automation

By Matthew Heusser for Subject7

It shows up like this.

The programmers make a change. At some point, the tests run, and there are a series of failures. The testers review the failures, and as it turns out, some large number of them were caused by the change. Others are "flakiness," something wrong with the environment, or browser, or the network. The testers then spend time fixing and re-running, fixing and re-running. Hours, or more likely, days later, the testers declare there were no "real" bugs, or at least, no showstoppers.

Over time, the programmers, and probably management as well, stop viewing an automation run as a source of valid feedback. Instead, it is at best something to wait for that will slow the testers down. "We wanted to go faster, but instead we bought another boat anchor," is ironically the ***good*** case.

We have not even considered false positives, where a test should fail but defects slip through.

Automating checks is a skill, and it is one that can be done better or worse. Today we'll discuss a few ways that test tooling fails when it is driving a user interface -- and how to do it better.

A few years ago, my mentor, Danny Faught, pointed out all the energy spent arguing whether a false failure was actually a false positive or a negative. I suspect some readers are already asking about that based on the introduction. From here on out, I'm going to follow his advice and use the term false error and false success, along with a few categories of error. Let's start with false errors created due to changes in the software.

False Error: Maintenance Changes

Computer-Driven test automation is pretty simple. It goes to a particular place in the user interface, takes a series of actions, and expects some result. There are other ways to think about it, of course. Model Driven Testing and what Dr. Kaner calls "High volume test automation" come to mind. Still, perhaps 95% of the test automation I see follows this simple model. Put that in place, reuse the same test data, and you will get the same result over and over again.

Until something changes.

When you think about it, the job of the programmers is to create change. That is when those automated checks really shine. If the expectations were to expect exactly what we saw before -- say a user profile -- and the results now include middle initial, the tests will fail

The most common errors here are probably that an *identifier changed*, some *required information changed*, or the *workflow itself* changed. The *identifiers* are how the tool "hooks," or connects, to a specific UI element. If the submit button is nested inside of three HTML division markers, then inside a pre tag, then the 2nd division marker after that, and the structure of the page changes, then the code will no longer be able to find that element. The fix for this is to try to make sure every button and textbox has a unique identifier, or named element in HTML. Simon Stewart, the lead developer for Webdriver, suggests using accessibility hooks, such as alt tags. Those have the side benefits of either increasing accessibility, which is a feature, or pre-existing, as they may be legally mandated. Required information changed is tricker; it can be that middle name, which is now required. When the automation clicks submit without filling in the new form, the result will be an error message. Workflow is a bigger version of the same problem, where the very button changed, or there is a new page on a wizard.

In my experience, some of these false errors are impossible to prevent. The trick is to catch the error as soon as possible after it is introduced. In some cases, this can be solved by policy. A policy that "a new feature isn't done until the old tests run," will make programmers responsible for "greening" these tests. Otherwise, if they are found by a continuous delivery run, it may be easy to figure out what change created the "error" and easily search for a "fix" for the tests. If the tests run overnight, or worse, once a sprint, then expect the problems I mentioned earlier.

False Errors: "Flaky" tests

Flaky tests are frequently referenced at software conferences as both a source of frustration, and a source of distrust for test results. One common piece of advice is to simply remove them. This creates a coverage problem, as some features are just left to, well, rot. I'm reluctant to call that a "solution," but it might be an option if that code is particularly stable yet the automation is flaky.

Instead of giving up, you could track the failures back to root cause and try to resolve them. These are the failures that you can focus on: :

- Shared data
- Unpredictable locator strategy
- Incorrect wait strategy
- Server/Client Microservices problems

Shared data. Perhaps the organization does a good job and splits the tests into a hundred micro tests, and does an even better job at running two dozen of them at a time. The average test takes two minutes, and suddenly a "full" test run takes under ten minutes. It sounds fantastic -- right up until you realize the tests are using shared accounts and stepping on each other. One test creates test results for another, or deletes things required for another test. These are architecture and infrastructure problems. If you've done a good job separating these, you might not have such problems… maybe not often...until someone forgets, and it turns out that test #21 and test #83 are not compatible, and the problem only occurs when #83 runs first. Another possibility is that a human is running manual tests in that same shared environment, and corrupts the data, causing errors during the automation.

Unpredictable locator strategy. I mentioned earlier that changes to the software code can move the location of an object around. Locators that tie to a specific location on-screen will now find that button, image, or text field are no longer present at that address. In some cases, the results themselves are unpredictable. The new order might be the first row of a table -- until some human clicks the "sort by date" link and the sort order changes. Now the order in row one is going to be the *oldest order*. Consider the test that adds something to the shopping cart, then clicks on the cart and confirms the item is the first result. That will work, unless that user already has an item in the cart. Sometimes the elements are generated at runtime by code, perhaps the date plus the order id. In that case, a test run that spans midnight might suddenly fail, as the software recalculates the object ID and it is incorrect. Tools that grab with a locator that can change will be tools with false errors.

When we were discussing this very topic, Simon Stewart once told me about tests that failed because the HTML ID's were internationalized. The ID's themselves actually changed names in different languages. This had no impact on the user, but caused all the test automation to fail.

Incorrect wait strategy. Some user interfaces have a message they can send when the user interface is drawn, a "page load." Others do not, or the tool cannot reach it. In that case, most people define a timeout. Make the timeout too long, and the tests will be incredibly long. Make it too short, and some of the time, the button you want to click will not be drawn yet.

Server/Client microservices problems. Sometimes what appears on the screen changes over time due to web services messages, or even just straight javascript. That can make waiting for a message that appears - and disappears - an issue. Looking to match an image that changes appearance over time can be even more difficult.

It's easy to list these. Preventing them, at the beginning of a test tool project, is probably your best bet. Discuss how to build the architecture to make the failure impossible, and code to those standards -- or select a tool that can help. If the architecture and code already exist, then fixing them … well that can be a bit of work. In my experience, the problem isn't figuring out how to fix them as much as getting the time to fix them. My advice here is to demonstrate how much pain the problem is causing - the bad information, the delayed test results, the re-work of re-running things and hoping they pass this time. Then, tell management the cost to fix versus the cost to keep things as they are. The next time someone complains about flaky tests, you have the same conversation again, reminding them *this is what the organization chose*.

Sometimes, the third or fourth time around, the organization has the potential to actually learn a lesson and make a change. Another alternative is to just pick off a scenario or two a week and just fix them. Sadly, that won't work for an underpowered test environment, and it may require changes to the production code.

False Pass Rate

False passes can happen for *coverage reasons* or *technical reasons*. In the coverage category, the feature is *not really tested at all*. Perhaps the tester found the scenario just too difficult to set up. Perhaps the tester who usually creates the test took two weeks vacation and no one picked up the slack. It may be that the automation tool came along after the codebase, and tests were created to match changes, not the original functionality. In any event, the results of a passing test run do not quite mean what the team thinks they mean. An audit of features to automation scenarios can find these holes rather easily.

With the technical failure, the test is driven right through the scenario. Something goes wrong, and the *tool doesn't notice*. The most common reason for this is that the test is simply missing assertions. It could be as simple as a demo made to show management the awesome power of automation. Yet the assertions, the things to verify that the features return the right results, were never created.

Getting the assertions right can be tricky. When I was at Socialtext, our checks for user-logged-in were things like "Hello, (firstname) (last name)" appears on the screen, or, when logged off, it does not. It is within the realm of possibility that the hello message is present yet login is still broken in some way. For that matter, hidden deep in expected results of any documented check or exploratory tester's mind is the assertion that "And nothing else strange happened." Computers are remarkably bad at that while humans do it implicitly. The opposite approach is trying to track down everything that could go wrong and have the computer check it on every run. That will be verifying every image (and image location), every bit of text, and style, and text location. Not only will it be impossible, but it will change the work into change detection, creating a maintenance burden.

Artificial Intelligence (AI) and Machine Learning (ML) have potential to analyze screen captures for "differences that matter," but there is no easy general solution available. There are a few niche areas where ML can help, generally with significant human work and training. Visual testing tools are another promising category. The visual tools do comparisons of screen captures and generally have "failures" reviewed by a human.

Sadly neither AI nor Visual Testing offer a complete solution to the false pass rate problem today. Until a general solution appears, testers will need to look for a happy medium between verifying too many things and verifying too few. One approach to this balance is looking for the "powerful few" verification points. These are the things that are unlikely to go wrong and, should they go wrong, indicate major failure.

A Balance - And a Pattern or Two

Finding the right balance of assertions will require a bit of work. Here are two common ways to reduce incorrect test runs.

Build recipes. Break the software down into features, scenarios, and steps, and you'll often find repeated workflows. Knowing two or three variables, such as login info or product information, and you are doing the same thing over and over -- logging in, finding a product and adding it to cart, checking out, and so on. Turn your head and squint, and these start to look a bit like reusable functions. When changes break code, when assertions need to be added or removed, you can at last put them in one place, not thirty, and re-run.

Audit your work. When tests fail when they shouldn't, don't just clean them up. Track the problem down to a root cause and add it to a tally in a spreadsheet. After a month, sort by count and review those root causes. That's doubly true for tests that existed, where the coverage is real, that indicated a pass and should not have. If the problems are bad enough, you may want to review the automation, a test a day for a few months, and invest time cleaning up the assertions. The challenge will be to come up with the assertions that matter for you, in your context.

Re-run if you have to. If you absolutely cannot fix the test environment, if you know the problem is the network sometimes, one final open is the great re-run. That is, run the tests, subtract the ones that pass, and re-run the failures. Then subtract the successes, and run again. Even with flaky tests, if a test fails three times in a row, then you should likely fix it.

Yes, I wrote it down. If you can't do the right thing, you can re-run failures and ignore them.

Only please, no, never do that. If you have to do that, it's time to take another look at your entire approach to testing. Do a serious internal audit about what is wrong, the risks you are assuming, the pain points, and recommend a way to fix it.

A completely ineffective, wasteful, embarrassing test automation program doesn't announce itself as it comes in the door. Instead, you summon it one compromise at a time, just trying to be *practical*, *reasonable*, and *pragmatic*. Banishing the waste, turning the ship around, can be politically painful. *Allowing a program to slowly fade into ineffectiveness could be a lot worse.*