SUBJECT7
QUALITY IS A CULTURE

Starting Points for Test-Automation

By Matthew Heuser for Subject7

Retrofitting a test automation tool on top of an existing application is no joke. As Fred Brooks puts it in the software engineering classic The Mythical Man Month, many a dinosaur has died in those tar pits. Unlike the dinosaurs of old, we keep re-creating the problem with automation projects. Even if we start the development effort from the very beginning intending to write automation to drive the user interface, there is still the question of what scenarios to automate *first* - where to *start*. Choose wisely, because where you start will turn into a strategy (such as "one feature a time" or "model the user journey"). Today we'll cover a few approaches I have seen over the years and provide you guidance to figure out what works for you.

Let's start with the various approaches to get started, then offer commentary on what their strengths and weaknesses are.

A fistfull of automation strategies

The easy stuff. Sadly, perhaps the most common approach is to do a proof of concept (POC) that defers any challenging work. These sorts of POCs often look impressive, but simply fail to invoke any part of the user interface that is a problem. Calculations, like order ID, or freshness of groceries, or simulating a new order entered elsewhere, or having an external event injected to see if it appears on the screen - just don't get done under the banner of expediency. Hopefully someone collects these on a wiki somewhere, to be done later. The most common reason to do the easy stuff is to hit a deadline to "just show something" to an executive group. The executives see the screen flying by and are impressed, unlikely to dig deep enough to understand what the automation *is not doing*. In the worst cases, it might not even check to make sure the result is correct. This approach can make something that looks like progress happen fast -- but it doesn't actually determine whether the tool is fit for use. The trap of doing what is easy is easy to fall into, and hard to get out of.

The smoke test. Smoke tests derive their names from hardware assembly in the 1960's and 1970's, when people were making consoles and arcade games. When the operator finished putting the machine together, they would plug it in and turn on the screen. If the arcade screen came on and the machine did not smoke, it "passed." Many teams have a quick, one-hour or so "rundown" of new builds. This is usually done by humans, time-intensive, and provides information at the level of "login is broken," "search does not return results," "add to cart fails," or "unable to check out."

One feature, end-to-end. This approach has the benefit of matching development flow. Done well, testers can get into the flow of development and automate right behind the programmers. One could even argue that a completed feature test provides a working demonstration that the feature is complete -- the tests are a sort of documentation of what the feature should do. Sadly, adding up all the features is unlikely to be the same as having software that is fit for use for customers.

One user journey. In eCommerce there is a path-to-purchase; in marketing, there is the customer sales funnel. Social Media has the create account/read/share cycle. All of these have a standard user journey. The journey has bunny trails; there are hundreds of ways to configure checkout at Amazon. Yet if you can find the one most popular, you can model the test after that, to show what the customer will do.

The riskiest components. If you actually have data on what features break the most often, you can model your tests on finding problems early. Put the tests in CI/CD, and the developer that introduced the breaking change can find out about it minutes after introducing the change, when the change is fresh in their mind, without wasting any human's time. In my experience, these tend to be complex scenarios that are hard to set up. The corresponding challenge is that people often just don't do them.

The hard-to-set-up-manually scenarios. "If it hurts, do more of it" isn't just a gym slogan to build muscles -- it can build all kinds of team skills and ability. This strategy starts with the things that are hardest to set up and evaluate. Testers who start here are likely to be testing complex, buggy things, or testing flows that are too hard to set up to test well manually. They will create all kinds of interesting infrastructure that can be reused later for setup and automation. If the tool is just not up to the job (perhaps it cannot "find" the complex user interface elements on the screen), then it is best to find this out as early as possible. These are all strong reasons to start with the hardest pieces first. On the other hand, impatient executives may find this approach feels slow. It is also possible the team is learning about the tool while using it and may want to throw away their first tests that were determined to have been poorly structured. In that case, the team may want to learn the tool a bit before starting ambitious or challenging scenarios.

Where to get started

Let's assume that you are evaluating a tool, perhaps building a proof of concept, and want to automate two to three scenarios that are good representations of the complexity of the test you want to create in the long run.

Starting with smoke tests, is probably your best bet. If "bad builds" that compile but don't run are a frequent problem, then smoke tests might add a lot of value fast. A good Agile Coach would say that recurring build issues are a solvable problem -- gather the data, conduct a retrospective, and provide whatever training or incentives you need to prevent those bad builds. Perhaps they are both right.

What almost never works, in my experience, is the easy path. Often testers do this to obtain budget, or to prove something to a stakeholder or sponsor. "We'll figure that out later," goes the thinking. "For now, let's make user interface elements fly across the screen." I've never taken it down that far, but I have heard of teams that continued this into development, eventually creating large test suites that needed a great deal of maintenance (in one case, an entire team) yet never seemed to find any important bugs. The case I am thinking of, near Lansing, Michigan, the team eventually folded....

Don't let that be you.

If no one is breathing down your neck for results, you might start with the hardest tests. Figure out if the tool can really take on those complicated scenarios. It may take two weeks to find out a tool won't work for you, but I would submit it is better to find that out now rather than later, when the tool has been purchased and you're stuck with it.

Most of the teams I work with end up with feature tests combined with a few high-level explorations of the user journey. Those are the type of tests they make once the tool is integrated into the team; it keeps the work integrated with ongoing development. Sometimes those high-level scenarios run in production, sometimes they fold into the smoke suite. The important thing is to realize when the strategy needs to change and to change with the times.

The advantage of the smoke test is that it is reasonably cheap, it covers the most important parts of the application, and you can put it to good use quickly. Moreover, it gives you a good sense of whether the tool will work out, and you can keep doing it. Eventually, additional "smoke tests" turn into feature tests or user journeys, as there is no more smoke testing to do.

In most cases, smoke tests are the place to start, as they are light, cheap, almost immediately valuable, and lead to better things.

Just don't stop there.