SUBJECT7
QUALITY IS A CULTURE

WHY COMPANIES FAIL AT TEST AUTOMATION

By Matt Heusser for Subject7

Testing pioneer James Bach wrote "Test Automation Snake Oil" in 1996. The arguments in that work, with the flaws to how test automation was perceived and sold, continue today. Companies continue to be "trying" test automation, or present successes at conferences, only to abandon the attempt within a year or two. This is particularly true with adding automated test execution to an existing or ongoing effort.

Organizations that start over, with a "green field", can design their process from the ground up, hiring people with the right skills and designing clean systems that have high coverage. Pure online companies that are designed around an application can have great success, and tell the world about that success. The rate of transfer of those ideas is low. In other words, companies that try to replicate that success on top of existing systems end up in the failure hole.

There are many reasons teams end up in the failure hole, generally several of which apply at the same time. In this article we explore those reasons to help you move towards a strategy that, while not full of "best practices," can at least be free from "worst practices."

These problems include "solving" quality problems with tests, unrealistic goals for coverage, and delays between the development work and test automation, which creates a bottleneck at best. Delaying the tooling can create merge issues, and many test tools have problems with version control, especially at scale. Teams can also have "automation blind spots" where tools cannot even see some risks, like skill and process issues. Those process issues commonly include gaps in transition of working style, leading to missed defects. Finally, we look at some infrastructure prerequisites which, if skipped, can limit the success of the project.

By avoiding these traps, companies can set expectations and design a more measured course, understanding the true cost, actual benefit, and have long term sustained success. The report that follows this one will discuss these problems in depth, including alternative ways to look at the problem.

THE GUI TEST AUTOMATION PARADOX

Most teams test a feature before they release it. The reason for GUI test automation is that there is a *regression problem*. A change to one feature might break something somewhere else in the software, somewhere unpredictable. The unpredictability means the teams have to retest everything before a release, which is expensive and time consuming. The hope with GUI test automation may be to find those bugs more quickly, enabling quick release. This is something quality legend W. Edwards Deming called "mass inspection." His goal for the automotive industry in the 1960's and 70's was to change the work-process to eliminate mass inspection. While American companies widely ignored him, Deming's success is part of what enabled the Japanese automotive revolution.

Software teams that pursue automation because of a quality problem are looking to have mass inspection - and lots of it - all the time. This results in an odd process, where the team first injects defects, then uses testing to find and remove them as quickly as possible. More recently than Deming, Elizabeth Hendrickson, recently Vice President of R&D at Pivotal, called the problem "Better Testing - Worse Quality?" Hendrickson pointed out that when testing is a separate activity that is a "safety net," programmers can feel empowered to skip testing. After all, someone else is doing that work. Having a magical, mythical tool to find problems can add to that problem.

Unless the defect injection rate decreases, test tooling *at best* can find the majority of problems earlier. There will be rework and retesting involved. Companies that pursue GUI test automation without taking a hard look at their software engineering practices will find mixed results for their efforts. Combined with the other classic mistakes below, this problem can cripple a test automation project.

MISTAKE : EATING THE WHOLE ELEPHANT

When Deming was talking about mass inspection, he meant every single part on an automobile. With software, there is generally an infinite combination of inputs, states, and timing. Add interaction between components and an incredible expanse of possibilities open up. Dr. Cem Kaner, lead author for *Testing Computer Software* and retired professor of Software Engineering at Florida Tech, calls this the impossibility of complete testing. In his black box software testing course, Dr Kaner suggests one key challenge of testing is selecting the few, most powerful tests to run.

Teams that pursue automation should have a concept of coverage, along with rules to understand just how much of the tests will be automated. Another problem is trying to retrofit all of the tests at the same time, from the beginning, as a project. These projects are not only expensive, but as the software is changing underneath the test-writers, the new tests add inertia that can slow down delivery instead of speeding it up.

When it comes to coverage, three common approaches are to test every feature exhaustively, to create "epics" that are full user walk-throughs, or to create small, easy-to-debug snippets that test core functionality. In our experience, the first two approaches are prone to failure. The first one, testing all the scenarios, will simply be too time and resource intensive. As the user interface changes, the tests will need maintenance, creating extra work. The second approach, to create "epics", will explore the user journey. These are generally full end-to-end scenarios, from logging in to search, add to cart and checkout, all in one test. These tests do multiple things, just like real users, and may do complex things … and they will be brittle. Debugging and tracing the problem will be more difficult since more setup may be required, resulting in the defects needing to be fixed and testing to be re-run. As more time passes, new defects appear, which will require debugging, fixing, re-running tests in a never-ending cycle..

That leads us to an advice matrix like the one below:

Style	**Coverage**		
	Exhaustive by feature	User Journey	Small Snippets
Retrofit tests			
Add scenarios with features			
Smoke proof of concept plus scenarios			

This third strategy, of slicing the tests into elements easy for a computer to set up and evaluate, brings up our next failure point: A straight automation of our existing process.

MISTAKE : AUTOMATE EXISTING PROCESS AS-IS

Our lead contributor, Matt Heusser, is quick to point out that what humans are good at, innovation, adapting, and creativity, are things that computers are not very good at. Computers are good at doing the exact same thing, over and over again, very quickly, without having to adapt. He feels so strongly that he put up some resistance to the term "test automation," as he sees the activities as separate and complimentary.

The thinly sliced scenarios discussed above are one part of the package; they are easy to set up, easy to run, easy to debug, and generally, when they fail, have a single or limited points of failure. It is also easier to run these sorts of scenarios in parallel, which can reduce the time to deliver results.

Most successful groups change the mix to play to their strengths. Some reduced number of small, powerful tests are automated, but humans still do the creation, exploration, and difficult setup work that might only run one time. Candidates for automation include the features that are most commonly used, most critical to customers, most likely to regress (features that are "flaky"), and so on. In the end you may have humans running a large number of manual tests for each feature, creating a smaller number of checks to automate and run every time, and also doing some of the larger, user-journey type of testing.

Conflating the tests that are automated with all of the testing (perhaps using the tragic moniker 100% test automation) may be one of the fastest paths to failure. Lisa Crispin, a co-author of the Agile Testing Book and winner of the Most Influential Agile Test Professional Award, has frequently stated that 100% test automation should be interpreted as 100% regression automation -- that automating any regression test run over and over, every time, might be reasonable. That still leaves room for humans to explore emergent risks when new platforms, operating systems, or use patterns emerge, along with things like canary and user-acceptance oriented testing. With canary testing, a feature is released to small, beta group, for days to weeks, before it is released to the wider audience. Like a canary in a coal mine, beta customers will report problems and be more willing to accept problem behavior in trade for new features.

Use case features, like printing or tab order, that are unlikely to break can be difficult to impossible to automate testing. If the feature is unlikely to break, the impact of the break is small, and the fix is easy, we'd be likely to recommend testing it once manually and letting it go. That still means humans should consider those types of defects and check for them. To allow the defects found to be driven by the test automation tool is something we call "automation blinders." To avoid automation blinders, periodically review the last 100 defects found by the team, especially those that escape to production, and consider what risk-management process should or could have found the defect. If the answer is "our tool could never have predicted that" and the impact is significant, you'd do well to consider ways to find, prevent, and rollback the problem more quickly.

Successful test automation projects change their way of work to include automation, blending the human and the machine.

MISTAKE : “DEFINATION OF DONE” MISSES AUTOMATION

Scrum with two-week sprints has been the de-facto standard way to do software development for the past several years. Larger organizations may add a layer of the Scaled Agile Framework (SAFe) onto that, moving planning to every four to six sprints, and a coordination sprint at the end of every planning increment. Both the sprint and the planning increment (PI) have a "definition of done." Automation can feel like "one more thing" to add to a sprint, and it is common to "catch up" the automation in the beginning of the next sprint.

In the fifty years of combined experience of the authors of this paper, none of us can recall a team that used this approach successfully for a long period of time. In practice, what happens is that the testers automate as "much as they have time for,” - typically the things that are simple and for which it’s easy to create scenarios. Of course, the software that will be the most brittle will likely be the software that is the most complex and hardest to test.

The coverage map for this type of approach is a bit of a patchwork-quilt. Then again, in our experience, teams simply do not have the tools or expertise to build a coverage map anyway. A change breaks the software, the tools do not detect it, the testers get defensive, everyone shrugs, fixes the bug, and the team continues on its merry way. Or, perhaps, senior management chastises the testers, who may do various things to prevent the problem next time, which might or might not happen. This isn't a problem with the people; it is the *system*. As Sir Thomas More wrote in Utopia, first you make thieves, and then punish them.

In our experience, if the definition of done includes automation, the automation will be created and the coverage will at least have a chance to be consistent. Anything else is wishful thinking. The never-ending desire for velocity, sometimes called the feature beast, will limit the potential for automation. Not only will the testers drown in the need to test new features, but they will be expected to do it in their spare time. Of course, the reason for the automation is the tester doesn't have enough time to start with! Under these conditions test automation efforts are unlikely to have any lasting impact.

MISTAKE : SINKING INTO THE “RUNNING -BEHIND” HOLE

The classic reason to adopt automation is because the testers are "running behind" and regression tests take "a long time." So the company purchases a tool with perhaps a day or two of training, and asks the testers to create tests "in their spare time." You can probably guess how well that turns out. Combine that with keeping automation out of the definition of done, and you're done before you've started!

With no clear strategy for adoption, and no spare time (the reason they need the tool is because the testers are behind), the testers are likely to make a token attempt, to create a few examples to get management off their back, and then get back to the business of manual testing. If there is a list of things that "should be automated", this "automation backlog" will grow, creating yet another burden the team is not capable of bearing. Even the goal, perhaps of going faster, will be nonsensical, as until the team has a significant amount of regression test automation, adding the tool will be just adding more work without obvious benefit.

Things do not have to be that way. A more enlightened approach might add the tool like a project, planning time to create a proof of concept and smoke test, and picking scenarios and snippets that are designed to be *powerful* and reduce time. As Enrique Henderson, a co-founder at QA Align once put it "I only introduce automation if It will speed me up inside a sprint. With that as a goal, as a contractor, I became incredibly good at finding leverage points to go faster."

Not every team will be able to accomplish what Enrique can; he has had years to learn his craft. However, with a little planning, our experience is that teams should be seeing results within a few months, not years. During those months, the testers should not have to experience burnout or fear.

MISTAKE: SKILL AND APTITUDE GAPS

Test tools come in all shapes and sizes, from visual record/playback, to writing code, to even machine learning, and predictive analytics. Asking a traditional manual tester to write code in Selenium is essentially asking them to be a Java, C#, or Ruby developer. At best they might write code that is "good enough," but will be hard to maintain and harder to reuse. Without a good understanding of software principles, a single change in the software under test could mean the test code needs to be changed in a dozen places, because the tester copied and pasted the code instead of building the code for re-use. The tester may lack the skill to reuse code or there may simply be no plan or architecture to the code.

That leaves us with one hand, where testers do not have the skills to write code. That leaves them with a preference for visual and record/playback tools. On the other hand, programmers tend to dislike record/playback and visual tools due to the fragility of their automation approach. They will be reluctant to help when the tools break, and may outright refuse to create tests in them. This makes the question "who will create the automation?" critical to picking the right tool. Teams that find a tool "foisted" on them may rebel, ignore it, or simply hope it will go away. The best tool will work with the team's skill and interest, solving problems instead of creating them.

MISTAKE: GLOSSING-OVER THE SETUP GAPS

Before a set of tests can run, the company will need the software undergoing testing loaded, along with databases and web services, with known good information. Tests of search to view a profile will expect to see certain things in that profile. Edit profile may change the profile, so before the test suite runs again, it needs to be reset. The version of the software under test needs to be loaded onto that system. Larger groups will want to test at the same time, which might require multiple test environments, or on-demand, virtual test environments. Doing this setup work automatically is a painful, expensive process for many programmers and teams. As a result, in order for automation to be successful, there is often a need for infrastructure that we find lacking

Companies that start with automation and have no infrastructure can have significant initial success. The proof of concept runs over a lunch hour, at someone's desk … if things were set up correctly in the first place. In addition, the tester is likely to skip any difficult challenge in the proof of concept. Most leaders and executives see the screen fly by, but do not ask for notes on exactly what is being checked and what is covered. The proof of concept is approved, the project moves forward, the difficult work has not been done. In some cases, it is never done.

That setup might take several hours, or require coordination from an operations team, or may tie up the test environment, creating gaps in the setup. If these issues are not addressed, as the test suite grows, the overall time to run testing grows. When an executive asks for an automation run to check the system status, the answer becomes "If we start tomorrow morning we can have it by end of day." The term for this is *automation delay*, and it is certainly not what the executive expected when the effort for automate testing was funded. This problem grows as a project scales in size of people and development effort.

MISTAKE : SCALING PROBLEMS

A single test environment with multiple people can lead to conflicts over who gets to use the test environment, which creates bottlenecks. The versions are likely to conflict with each other. This is an infrastructure problem, which we addressed above, but there is also the potential for a version control problem. This is most prevalent if the test cases and object repository are stored in a binary file format that is saved. Users in the same file can step on each other, leading to either complex, multi-file schemes, or frequent re-recording. Tools that are designed to be for multiple simultaneous users may still have merge problems, but the merge problems may be manageable.

Programmers can also "break" the tests as they make changes. For example, when a field like middle name moves from optional to required, any tests that create a user and leave middle name blank will "fail." At the team level, this is manageable, but as the number of simultaneous testers and programmers increase, unless the tool supports segmenting the work, the build will have more and more of these "false errors."

Over time the test suite will also take longer and longer to run. In our experience, a test suite that takes more than two hours to run will introduce new problems, as the programmers will push code faster than the suite can run. Beyond six hours, the suite can become unmanageable. The common solution to this is to run multiple tests at the same time, perhaps on virtual machines. We call this running tests in parallel. However, if all the tests re-use the same account, and one test runs a search while another creates new web pages, the number and order of the search results may change. Isolating the tests, or designing them to run independently, is another challenge of scale.

Before starting a test automation journey, consider what will happen when the entire engineering group is using the software. Also consider if the group adds 15 minutes of automation per team per sprint to the regression suite -- how long will that be in a year? Two years? A proof of concept written by one person, running on one machine is not a feat of engineering. The entire project might be.

PUTTING IT ALL TOGETHER

This list of problems can be overwhelming and scary, but it doesn't have to be. Instead of being overwhelmed, review the list again and make your own list, the traps *you* might fall into. Then be mindful of that list as you design your test automation project. See the traps. Navigate around them. Or, as the saying goes, if you've found that you're digging yourself a hole, the first thing to do is stop digging.