



Getting Ready for Test Automation Success

By Matt Heusser for Subject7

In our article on [How to fail with test automation](#), we discussed our experiences with how projects go wrong. The next logical question is how to make them go right. "How companies fail ..." is an easy list, as part of the problem is cookie-cutter, one-size-fits all approaches. That makes how to succeed difficult.

If you don't know the starting point, directions are useless. For that matter, unless the goal is defined, the directions could take you to the wrong place.

Any roadmap for test automation will lack context - the things that make your organization different or unique. The roadmaps for test automation are the problem.

Instead of a road to what to do, this article takes a step back and looks at the ingredients most organizations need to be successful, then helps you look internally through conducting a self-assessment. Armed with the self-assessment, you can deploy scarce resources in order to get ready, along with the knowledge of what challenges will come your way. The areas we focus on are *test data, people and skill, setup and deploy, goals and measurement, test design and coverage*.

A thorough analysis will probably take a two-person team which includes a technical person and a portfolio leader to manage the transition.

With the analysis in hand, your team will know what to do to be successful with Test Automation. Combine that with our other research papers on measuring success in order to create a plan to improve speed and quality of feedback through test tooling. While velocity is the speed at which the software is developed, feedback is how quickly problems and opportunities are identified by the team after they are created.

GOALS & HOW TO BUILD YOUR MAP

This article will help your organization implement customer-facing test automation - automation that will drive a user interface, such as web or mobile applications. It will also be relevant for Application Program Interface (API) strategy. The general framework, or way of thinking, about test automation, uses this list:

- Faster feedback enables faster builds
- Regression testing can be automated and integrated with the build pipeline
- The deployment pipeline can be integrated with the builds
- Good engineering practices can reduce defects that escape to production
- Monitoring, canary deploys, and engaged customers reduce time to identify defects
- New styles of delivery plus monitoring tools make fixes a fast, easy, rare priority
- Instead of clear, defined responsibilities by role with a "wall between roles," the organization is pursuing a more collaborative approach, more like DevOps or Agile Development
- Configuration flags, automated deploys and tests reduce mean time to recovery (MTTR)

That is, itself, a sort of cookie cutter approach. It is certainly a style. It might not work for you, and it certainly is not the only way to succeed at test automation. However, our experience does show a correlation between these topics and success. If the approach won't work for your organization, if that is not a fit, that does not mean stop reading, but it does mean carefully consider if the advice is going to be a fit for you.

If those are your goals, the next step will be to build your map - that is, determining where the organization stands now and where it wants or needs to go. If your team has tests in a structure that will be easy to automate, you might focus on other pieces of infrastructure, such as canary deploy or monitoring. Likewise, examine the skills the team has, consider who will do the work, and consider what test approaches they have the interest and aptitude to implement.

Let's talk about the issues to line up when starting a test automation program.

GOALS & MEASUREMENTS

Classic financial measurements for software are either return on investment (ROI) or cost basis. Historically, value is the more challenging measure, so organizations ignore value and go to cost -- which can be disastrous. Unlike programming, where measuring business value is hard to measure, testing appears to have a fixed value, with the greatest opportunity to improve is in cost reduction. Cost, in testing, looks easy to measure -- generally employee salary cost.

Many companies turn to test automation to reduce test cost, yet end up hiring automators or losing velocity as production programmers write automated tests. In the first case, test costs go up; in the second, development speed goes down.

With all those problems, having goals, and the ability to track progress toward those goals, for a test automation project is critical. Yet time and time again we find organizations have ill-defined goals, such as "100% test automation," with no timeline, no roadmap, and no effective measurement in place.

In the Whitepaper, [How to Measure The Success of a Test Automation Project](#), Subject7 walks through the measurements for ROI for a test automation project, that considers actual expense, increase in velocity of software delivered, along with harder to measure variables such as the value of defects fixed more quickly and lower turnover. That document walks the reader through how to create goals, and evaluate if those goals are realistic.

With a strong goal and aligned measurements in place, you can see if the plan is on-track and make fine-tuning adjustments.

PEOPLE & SKILL

This is a matter of who will be doing the testing, if they have the aptitude and knowledge to do testing in that style. This is less about training, although your organization may need to obtain it. Instead, what matters is if the person has the aptitude or interest in doing the work in that style.

For example, several groups we work with attempted a code-driven style of testing, using a tool like Selenium. The existing testers did not know how to program, so they struggled for a year or two. Eventually the group realized the testers simply were not capable of learning to program, at least not without going back to school at night and pursuing something like a college degree. Two companies we worked with hired testers who were programming-testers.

Let's look at these two companies that hired testers who were programming-testers, and a third where the tester learned to code on the job.

Case 1: This company had production programmers that knew Java. The company made the decision to bring in Java testing. However, Java programming and Java testing are different, causing the company to have an entirely new programming language to support. The production programmers had no interest in supporting the Java testing. This company struggles to this day in supporting programming testing, even though they have significantly increased their dollars spent per year on testing.

Case 2: The second company hired testers that used Python. Eventually the testers left and the test suite was thrown away because there was no one who knew the Python test language.

Case 3: In another example, the company had a tester willing to learn to code by going to school at night. Now the tester is a junior programmer. The test-code that was written is no longer in use or supported because the tester has moved on to a new position within the company.

These sorts of issues arose because the company failed to consider temperament, ability, and staffing for the test automation work. If the test-code is written in the same programming language as the production code, it is possible for the programmers to support it, or even write the test-code. The question is whether or not the programmers *are willing* to write test code and then support it? Or, is the company willing to hire and support dedicated personnel working specifically on writing test code and supporting it?

Another option is to use a tool that does not require code skills - or uses a web user interface to abstract out automation coding so that anyone with any level of technical expertise can use.

These tools can have looping, variables, subroutines, and so on, but can be understood by a more traditional tester. In this case, programmers may be reluctant (or refuse) to provide support.

The questions at this point are not "which approach is right", but "which approach could work for the software and the team right now," along with "how large is the gap," which we discuss in the conclusion to this paper.

TEST DATA

Imagine testing the Search function for a popular ecommerce engine. You would need users, products, and existing orders to have predictable search results. The catch-all name for this is "test data." When humans test systems manually, they can create a new account or re-use an account. A human can ignore the order they entered five minutes ago, and know what to expect when running a test to create an order a second time.

Computers are incredibly bad at that.

For an automation script to run well, the "driver" software will need to know how to log in, what to look for, what it should expect to find, what the past orders were, and what it should see when adding a new order. It may further need a fake credit card number that is close enough to a real one to get through the test system. In some cases, you may want to test the number of results to know that the screen will show "Showing orders 1-20 of 207." Run the "create order" automation twice in a row without cleanup, and the software will use 208 orders. Hard-coded dates can be particularly troubling for test data. Humans can deal with a refresh from production data (with key details hidden) or a great deal of old, "dirty" test data, while this can often choke automation.

Put differently, most test automation tools need predictable data every time they run, yet the run is likely to create a paper trail of new data. The two most common approaches to deal with this are to either to 1) isolate the data, creating separate accounts for every run, purge the new data after every run, or 2) have one or more "test databases" that can be added again after every run.

Once a computer is doing the checking, this test data will need to be pristine. If the data storage will need to change, talk to the architects to see how easy those changes will be to make. If the data is stored externally, it may be possible to stand up a test server, sometimes called *service virtualization*, to mimic the test database with good known predictable data. The data will likely need to be stored, so it can be imported and exported on demand.

With that hard work done, the setup of the entire system, including build/deploy, can be automated.

SETUP & DEPLOYS

Let's talk about two ways to develop a test suite. The first, more common way, is to write a large amount of automated checks that run against the user interface. Each of these is a scenario that covers some feature. The tester can install the latest build on the one test server by hand, set up all the test data just right, press the button, run the suite from their laptop, impress the executives, then go to work on building more tests.

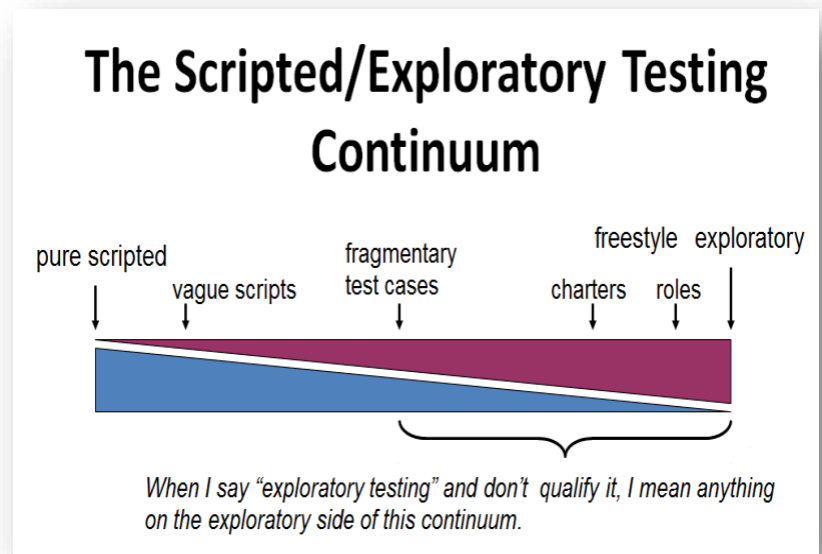
Each time tests are run, installing the build and setting up the server takes several hours, and the "regression pack" that is running takes longer and longer to run as well. Eventually a "test run" is three hours of setup followed by four hours of automation that may be creaky. If something goes wrong, the tester may need to re-run the setup and test again... tomorrow.

The other way to develop tooling is to work on a build pipeline that automatically creates a test server, perhaps in the cloud, and runs the tests somewhere else, not on a laptop. This system will run one test, and run it well. After that, additional scenarios are added carefully, with an eye toward timing and cleanup, so the scenarios cannot "bleed" into each other. The tool will likely need to set up, deploy, and cleanup. Additionally, you may need to provision -- that is, to create test servers in the cloud that did not previously exist.

Organizations that try the first approach, and do not dedicate the resources to create end-to-end automation, will lack either the *will* (no one wants to do it) or the *strength* (the organization will not assign resources to do the work) to make the change. Those terms will appear later as how to address the gap.

TEST DESIGN & STYLE

James Bach, the founder of Satisfice, Inc, created the diagram at right to demonstrate a "tester freedom" scale. Toward the right, the human tester is more in charge of the process; where towards the left, they follow more formalized directions. The purple indicates the amount of tester freedom. Note that even the elements to the far left have some amount of purple. More recently, Satisfice and their partner, Developsense, have changed their language, dropping the use of the term "exploratory," to indicate that all human testing is exploratory to some extent.



Organizations that put the tester in charge may focus on more features being covered and more bugs found, instead of the *how* testing is performed, sometimes called "Test Cases."

Computer-based checking is not exploratory. At Subject7 and Excelon Development, we agree. The part of the testing work done by humans should be unique and interactive, adjusting to changing conditions on the screen - things the computer cannot do. If it can be automated, let the computer do that work. That means the human work will add variety, creating additional coverage over time, as testers try *different* things.

This does create a problem when it comes to what steps should be encoded and run for every build by the computer(*). When companies do document tests, they tend to write up an entire user journey. Our experience is that automation tends to do best by creating test suites. These suite is a large number of small "snippet" tests that can run independently. Each "snippet" represents a scenario, or, better still, a micro-scenario, along with any needed setup or teardown routines.

That approach won't work for every team, and it brings up the question of "how many" automated tests for a given feature are enough.

MEASURING TEST COVERAGE

Perhaps the world's "dirty little secret" in software testing is coverage. That is, how comprehensive the different parts of the application are tested.

Very few companies have any real, meaningful measures of user-facing test coverage. In most cases, they lack any good measure of "how well," which is fine, because they lack any breakdown of the "different parts of the application." One large project gave its dozen teams two weeks to perform testing before deploying a release, but only five of the teams had any meaningful list of test scenarios to run. Those test scenarios were detailed, but they did not correspond to any particular list of features. With the help of a consultant, the group spent two months creating a map of the application, then assigning which teams would test what features. Two of the features were designed by contract-teams that had been disbanded and had no meaningful documentation. At that point, the company simply knew who was responsible for what -- not how well the feature was tested or how to test it. The team spent the next two months creating a regression test plan that at least created charters (a feature and some idea of how/what to explore), along with a short list of what could not be tested due to configurations.

In our experience, where many teams have a "regression pack," they likely have no idea of what scenarios are missing and features are uncovered. Testers tend to "give up" when a feature is too complex ... which is also the feature most likely to have bugs.

Teams that have a good idea of features and coverage, including what flows are chronic and core, will find it easier to get started designing GUI checks that are helpful. Teams that just "run all the tests" or "play with the recent changes" will have a gap to address.

MOVING FORWARD – SIX BOX ANALYSIS

"May God Grant us the wisdom to discover right, the will to choose it, and the strength to make it endure." - Camelot (Movie), 1995

While our work here is very closely tied to software testing, it is loosely inspired by the Six Boxes Model at right, popularized by the [Performance Thinking Network](#). Most tool implementations cover only boxes two and four. Companies buy the tool (box 2) and, perhaps, provide a day or two of training (box 4). When the tool implementation fails, the organization turns to more training, a different tool, or blames the people.



Our suggestion here is a wider approach - look at who will be doing the work (box 5), what the work consists of, and understand the consequences (box 3) that are likely to happen. Then adapt the system forces, such as the software architecture, the expectations (box 1), or the choice of tool, to make the best possible result.

To help our readers navigate the internal reflection that will help set you on a roadmap towards success, we suggest a scorecard to help make informed decisions that are relevant to your business. The approach is pretty simple, list the major issues in the first column, one for each of the headings in this whitepaper. Then use the next columns to score each. We recommend four score columns, the first is the gap, a measure of your readiness in each category, and then three columns to measure organizational will, strength, and incentive to drive toward the value offered by the tool.

Will is the ability and motivation of someone within the group to pick up the reins, be the change, and address the gap, possibly in their free time or as "additional duty." *Strength* is the willingness of the organization to put time and focus on the gap officially. This could be hiring a contractor, new employee, shutting down a different project or moving deadlines to allow for the team to address the gap. *Incentives* represents how people will be rewarded for taking time away from their work to spend on automation. (If you can already hear an executive saying "What? Test automation should immediately speed us up, we don't want to spend more", then strength is low, and incentives probably are too).

This exercise will at least uncover different expectations about how easy it will be to implement the tool, how long it will take, and who is doing what.

Once you have a completed scorecard, do not just sort and consider the smallest score to be the biggest problem. Yes, look at the totals, but also look at the numbers separately. A strength of zero with a large gap could be a serious problem. A few years ago, we worked at one organization that copied tests from production each quarter. They were unable, or unwilling, to create an isolated test database that could be refreshed on demand. This meant tests that ended prematurely would leave "dirty" data, extra transactions that had not been cleaned up, which would be discovered on the next test run. The company was also not able to segregate the data, so the tool could create a new account for each run. As the "can'ts" added up, the value of the tool decreased. Today the company "does" automation, and can release in a week instead of a month, but that is a far shot from the gains they were hoping for when they went down this road. Of course, without [realistic goals or ways to measure them for the effort](#), it is hard to say they failed.

We want our customers to blow the doors off. We hope that describes you.

Please let us know how it goes.

The project you save might just be your own.

END NOTES

(*) - There are methods to introduce randomness and variety into automation, such as [Model Based Testing](#) (MBT), those approaches require advanced programming skill and tend to find edge cases. While it makes sense to use MBT as a second layer of test on the world's largest search engines, the concept has not had broad popularity, despite popular tutorials at the world's largest software conferences. Instead, most user interface test tooling runs the same scenarios, sometimes looping to run on different inputs and expected results.