A Test Scaling Story

By Matthew Heusser for Subject7

Previously I explored what the word scaling means and in particular what it might mean for you and your organization. My short explanation is to allow testing to continue to "work," without introducing delays, excessive effort, or problems, as something gets bigger. What that thing is depends on your group. It could be more features, more teams, deeper testing, more frequent releases … something.

Today I'll tell a story of how one organization could scale testing. Due to non-disclosure agreements, I've decided to fictionalize a real experience. That is to say, I have actually done, or worked with teams that did the things I am writing about. The stories are primarily one company but I've sprinkled in enough other experiences that things will be unclear -- this is not some sort of cleverly invented story where everything is wrapped up neatly at the end. Testing is hard, keeping testing going is an ongoing journey. It does not, however, need to be a death march.

Our story so far

You are a consultant brought into a company that makes a complex integrated application. There is an internet of things component that deals with the real world, a web application that also runs on a mobile phone, a lot of APIs, and a fair bit of physical control issues. Let's call it a video camera and door lock, with a battery. There are also a fair number of security measures to make sure you really do have rights to unlock the door and a keypad with a code.

There are several front-end teams, an integration team, a team that works with the security company, and one that handles the online finances. That is, once you buy the device (we'll call it a "thing" as in the internet of things), the company wants to cooperate with a security company, so you pay a monthly service for monitoring and notification. Plus there are security tokens you can put on the windows to sense other kinds of intrusions. They all work together.

It's also your typical big company. The thing company was once small, then chose to sell out instead of going public. That meant a payday for the founders and investors, but it also meant the software had to work with the other family of security products the big company offers. To make a request to open a door, the phone has to make a back-end security API call. If that passes, the software calls the old APIs that track door-locked-ness, which call the home wifi device, which unlocks the door. Unlock/Lock can take sixty seconds to work, sometimes you'll get an error message, then the door will unlock. Sometimes you get success, and the door does not unlock. Most of the time, if someone unlocks the door, you'll get a notice on your phone the door was unlocked. As if I have to say it, testing takes too long.

Here's how things work now: Every twelve weeks there is a testing sprint. This takes two weeks during which there is very little forward progress. At the end of that two weeks, there are bugs, then perhaps two weeks of fixing, then two more weeks of testing. At the end of that third "*hardening*" sprint, management makes a ship decision, which might include two more weeks of fixing and retesting. This entire process requires coordination between a dozen teams in six locations on three continents, though six teams on the front-end are mostly in one office.

For those that are counting, that is twelve weeks of development and six to eight weeks of hardening. That means *33% to 40% of the time is spent hardening*. Management calls you, a consultant, to "fix it." This is the exact definition of a scaling problem as we defined it last time - too much of something is causing the project to break down. Too much software, too many teams, too many components, too much of everything. The goal of "fixing" it is to get more effective testing in less time.

Scaling Testing

Being a good consultant, your first question is probably the classic "what is the goal?" If the company can just decide one thing, you can give it to them. Do they want better coverage, do they want testing to get done more quickly, or to enable continuous delivery? Do they want feedback earlier? What do they want?

And the answer, of course, is "Yes." If management just wanted one of those things, they could find an employee to get it for them. Instead, the division president says "We want to have our cake and eat it too. Fix it."

Welcome to the day in the life of a consultant. What are you going to do now? Go ahead, take a minute. Take a walk. Get back to work. Think about what you would recommend as a consultant.

Now I will tell you what our team came up with, what the customer implemented, and how it turned out.

Organizing the test effort

The first thing we decided to do was figure out what the software actually did, so we created a feature map. That is, a list of all the features, and which team was responsible for testing it during the test-phase. Call it SAFe if you want, call them "product increments." Just know that in reality it was a test/fix phase. And it was long.

During the feature-list process, we found features that no one owned. The team had been re-organized out of existence or had complete team member turnover, so no one remembers what the feature did. In one case, a major feature was created by a team of contractors funded for a year that "went away." My role was in getting the user interface element removed so we did not even pretend to offer the feature, as it did not really work.

Once we finished, executive management had to step in and delegate teams to plug the holes. That left the problem of knowing what to test, how to test it, and coordinating the test effort. As for what to test, we had the teams learn about the products and write down what they would do to test, while we reviewed their work. We created a visualization of the application that allowed people to drill into the features to see how they were tested -- or to make a copy for any given release. That made independent testing of just one feature possible. Some features, like an API, could be deployed independently, and flipped-back if they were broken. Once the contract of how the API worked was codified, we could test just that API and release it, instead of waiting eighteen to twenty weeks for a new release.

Meanwhile, we developed a multi-user online spreadsheet to describe the status of a test run, and compressed the test effort to one day. This asked the question "what is the most powerful, valuable testing you can do with this group of people in a day?" The test effort we came up with combined GUI testing with the physical thing inside a simulated house, along with other sensors. So yes, we did a fair bit of breaking into our own building -- and had to simulate the call to the security service so they would not get a call "for real." The whole time I was there, we only accidentally called security twice, and they knew we tested out of that building, so there was no tense gun-draw, don't worry. Then we had the automation to deal with.

Scaling Test Automation

It's probably no surprise that the automation was flaky. It was home-grown; the install process was a list of components that would take a new tester perhaps a week to learn and install. The software mostly consisted of various Open Source components that had to be installed in just the right version or else they failed to work together. As a consultant, I fought my way through the install process but documented it as I went, down to URLs to copy and paste. Eventually, we turned it into a bash script that could run from the command line that we put into version control. That would work until Apple released a new version of macOS, or something else, and the tool would need to be tweaked again.

Then there's that flakiness problem.

To fix that, we made a list of the flaky tests, what they covered, and how important each one was to the test effort. We went through and either fixed the root cause, cut out that part of the test, or cut out the test entirely. We had a visualization of feature coverage and "just knew" that feature coverage would be less. I did an actual deep-dive RCRCRC analysis using logs, GitHub churn data, and defect data, to figure out what areas of the code needed heavier (or lighter) testing.

Once the tests actually ran consistently, we bought an integration machine and started to run them *all the time*. The integration machine would check out the code, then spin up four (later eight) browsers at the same time to run the automated GUI tests. Our consulting team helped start a similar effort for the API tests, though that required a fair bit more infrastructure. On the GUI code, when a change was merged into master, the tool would pick it up on the next run. With eight browsers at the same time running headless on Firefox, we could get an entire test run in under two hours. Granted, the time from the developer commit to merged to the master branch master could be two days due to code review. Still, those worst-case two days compared to two weeks when the system ran before on one person's desk, for a business day. Those prior runs would also take a full business day … unless the run broke somehow in the middle. Those test results could end up coming in two or three days after the test run started, as the tester would need to analyze, re-run, reproduce manually, document, argue about if it was a bug, and so on.

By the end of the process, developers got test results in-sprint that could generally be traced back to a single change. Regression testing would take a day and could be scheduled at any time; releases no longer needed to happen every twenty-weeks. That meant the amount of uncertainty, the number of bugs, never had to get large. The team spent less time documenting, testing and re-testing, and arguing about release requirements. Instead, they got to spend that time improving and supporting the software.

The British people have a saying for when an elegant solution sort of makes the problem disappear "... and Bob's your uncle." I wouldn't claim that here; the actual process was a lot of slogging, with fits and starts. However, at the end of it, the results were better, faster, and cheaper. On my last day, ironically, my sponsor bought two cakes for the large testing group and our farewell. I suggested he literally eat one and keep the other.

How is your test improvement effort going? Are you scaling, or are little chips falling off as the scope grows over time?

Ready to learn more about Subject7?
Contact us today to request a free demo.