Where do you Start Building your Testing Program?

By Matthew Heuser for Subject7

Exactly what to do in your test program is context-dependent. Don't take this as ***the*** way to create a test program, but instead as ***one way***, a way I have seen work at several companies, paying less for testing and getting better results than their peers. Some of the readers here won't really have a test “program,” not really anyway, but there are inevitably folks doing various "things” that will help find bugs before releasing new code. These things may vary per team, per project, per person, or even per day.

In software engineering terms, I am going to lay out a ***reference implementation***. We'll start without a test layout, talk a little bit of theory, then pretty quickly move into a real practical design for a middle-small software system. That is, somewhere between three and nine software teams create this -- but I have seen the approach used by as many as fourteen teams on three continents. Beyond that, I find we can break the software into subsystems as big as each of those.

Getting Started

Last time, I mentioned Karen Johnson's RCRCRC heuristic -- that you can find test ideas by looking at elements of the system that are Recent, Core, Risky, Configuration Sensitive, recently Repaired, and Chronically defective. It is a good source of insights, but not quite a system. Over the years I've drawn on version control tools to find Recent changes, logs to find the Core, bug trackers to find the Chronic, Risky and Repaired code. To get started though, I act like an emperor in a science fiction novel: focus on the core. If the core is lost, nothing else matters.

Within the core, there is the uber-core, the things one simply ***has*** to do it in order to use the system. In eCommerce, that is the homepage, search, product display, add to cart, and check out. You might add fulfillment (the warehouse gets the order) and making sure the credit card gets charged to that list. If you have logs, you can search through them to see how often operations are performed. We had these logs at Socialtext and they were helpful; there were features we had no idea were so important to the customer that we uncovered that way. It turned out that among our entire user population, “download an attached file” was more common than “tag a page,” or even use “tag to search,” for example.

But let's say you don't have those logs. You don't have any formal documentation on features. Your product managers say they are too busy, no one reads documents, the software changes too often, and they don't have time for that anyway. Sound familiar?

Here’s something you can do that will inevitably help. Make a list of all core features, along with their critical flows. This is all just one spreadsheet. Tab A contains the core features. Tab B has the flows, and in a second column, lists all the features that flow covers. If the flows are disconnected and independent, you might not have tab B at all, instead, just use a spreadsheet of micro-features that are different enough to have problems that most users will hit 90% of the time. Once the spreadsheet exists, share it with the team and get them to add the micro-features that are missing.

If you want to get really fancy, and anticipating a post-Covid era, print them on note cards and put them in the biggest conference room you have. Allow the team to walk through and sort the feature in terms of use, from most-used to least. After people have walked around, they walk around again, making corrections and debating. By three or four rounds, people will probably agree on popularity. Then hold up the top three or four note cards; these are your starting point. We can call them the 90+%, basically the most critical features and capabilities in your app. The next group of three or four are your 80-89%, and so on and so forth.

When you've completed that, you have a completed priority list for test automation. Preparing the spreadsheet might take an hour; the meeting might take an hour. It's super fast, relatively accurate, and any team can do this.

Working on the list

Now just order and group them in your spreadsheet, so you can pull the test ideas as stories and create them in a tool, one at a time. Assuming the complexity of the stories is roughly the same, you can count the number of stories that are done in two weeks and predict when you get to the 90% cutline, and then the 80%, and so on. If that isn't fast enough, management can add time, people, or lower their expectations.

This approach provides a constantly-increasing amount of coverage, starting with the highest "bang for the buck." As core functionality tends to be relatively stable, the tests you create are less likely to be brittle or "flaky." Doing the work to create the plan and running it for a week or two also gives management a budget, in terms of time and money, that is backed by real data, instead of wishful thinking. The plan is so inexpensive that it is almost free. I have also seen teams take those lists of features and flows and create main-maps, wikis, and other forms of documentation. As it turns out, the test effort can drive a documentation effort as a side benefit.

The list itself will change over time. Creating the tests is very different than "imagineering" on a spreadsheet, so test ideas will occur to the testers as they do the work. Expect some small amount of "test case bloat" and maintenance cost. What this system does provide is that it minimizes this effort and makes those costs explicit.

As I said before, this isn't the only way to do it. Next time I'll provide a different approach based on recent changes to the codebase. It's a little tricky, and harder to get right – but for today, I focused on the Core. Try it, and keep us posted on what you find….

Ready to learn more about the Subject7 test automation platform? Contact us today to request a free demo.