SUBJECT7
QUALITY IS A CULTURE

When Should You Rewrite or Retire a Test?

By Matthew Heusser and Subject7

In the preface to his book Extreme Programming Explained, Kent Beck said that he likes teams to run fast. Design documents, technical specifications and anything that needs to be changed as the software changes create *cruft*. Cruft will either become outdated (and incorrect) over time, or else need to be maintained, and maintaining cruft slows the team down.

Beck suggests teams pack light to resolve this, carrying only two things: code and tests. Yet even tests can become cruft. This came up during our Test Automation Pyramid webinar in a question asked by Julian:

What are your thoughts on removing/cleaning tests that are old or no longer add much value?

It's a reasonable question -- say, for example, an automated test never finds a bug, perhaps ever. What is changing is the *workflow*. The airline company is constantly making offers appear for upgrades to seating, or Wi-Fi, or early onboarding. This makes the tests report a failure when the code is actually working properly. That creates extra work for someone to debug and "fix" the tests.

Another way tests can become cruft is if they slow you down. When a continuous integration system starts to add dozens or hundreds of tests, it can start to slow down feedback. The difference between feedback in minutes and feedback in hours can be significant. With a multi-hour feedback loop, programmers have to jump out of whatever they were doing into the failing test. One way to resolve this is by removing the tests that are considered cruft[1].

But how can you tell which tests are still useful and which have become crufty?

Let’s discuss a way to analyze tests to decide if they are cruft or not, and then what to do about the ones that are cruft — rewrite, retire, run rarely, or something else. Finally, I'll provide a simple quick win that resolves a common anti-pattern. Let's get started.

Making the List of Cruft

One company I worked with brought me in to be a test coach. The most visible problems were with their front-end team. The division, part of a Global 50 company, had objective problems with their quality, simply based on the app store ratings compared to competitors. So they did the right thing: They funded a quality initiative that included both better testing (check before it goes out) and a lot of bug fixing and features for usability.

The team had a mission, but no data. So they ran off doing … things … designed to improve the application experience. There was a lot of adding instrumentation or "redoing the sign-on experience" and so on, but none of it was based on data.

As you might guess, the team did not have much interest in working with me. They were too busy — you know, working on the quality stuff.

You might not have quite that level of challenge, but I suspect the problem is at least familiar. The first step is to get data. When it comes to tests, there are several useful kinds of data:

- How long the test takes to set up
- How long the test takes to run
- Recent bugs the test has found
- The human effort the test saves
- Features the test exercises
- The relative priority of those features
- The problems the test introduces: delay, support, complexity, and maintenance
- Tags for the test (more on this later)

It's tempting to run off and make a spreadsheet that lists these numbers, come up with a formula, and calculate the "goodness" of the tests, cutting the tests from the bottom of the list. You could do this if you like.

However, when I do this kind of work, I find that getting all the information above is just not possible. Instead, I'll create a spreadsheet with as much information as I can get, starting with the test name. The point here is to *dig into* the end-to-end tests. On a larger installation, the scope of this might not be possible, but hopefully, one leader per team can do this kind of analysis.

The analysis should be more personal than pure data, though. Sort by the various columns and find the things that seem to show up as consistently the worst. I'm talking about the tests that take a long time to run, haven't found bugs lately, are covered by other tests that run faster, cover low-priority features, or introduce a maintenance burden. If the maintenance burden exceeds the value, or the speed causes the build-test server to go too slowly, those are your cruft.

A Solution to the Speed Problem

When I was at Socialtext we started to add tags to tests, such as "Happy path," "Search," "Slow," "Fast," "Create a page," "Profile," and so on. If a programmer was working on the Profile feature, they could run *just the profile tests* in their development environment to get fast, scoped feedback.

Likewise, we could run just the happy path tests as a forever commit as part of continuous integration. We watched them run the happy path tests to make sure the entire build stayed under about 20 minutes. With a build-test loop that fast, programmers getting alerts

remembered exactly what the change was and could debug it quickly. This gave programmers the potential to fix the tests or the code before it was ever given over to the tester. The difference between this and running (and fixing) tests at the end of the sprint is night and day.

Speaking of night and day, the complete test run, even "headless" without a user interface, took between two and four hours, or more on multiple browsers. Doing this kind of testing on every commit would take a great deal of processing power and slow down feedback. However, there was no reason not to do one big run overnight.

That provided three levels of testing: tag by feature and filter for work within a feature, filter by "happy path" for work outside of a feature, and tag "current" tests for part of an overnight run. Tagging also makes it possible to list tests as "old," which is the first step toward retirement, or as "investigate," or a signal to see if the test can be redeemed through rewriting.

The strategy above could take a fair bit of time to run, but by starting with a list of the worst offenders, a first pass might only take a day or two. However, there is something else you can try that takes less time.

An API Quick Win

One common point of contention is redundant tests — that is, tests where the unit, integration, user-interface-only (perhaps mocking the back end) and end-to-end tests are all essentially doing the same thing. For example, if an API looks up abbreviations and returns with what they stand for, the end-to-end tests might look, more or less, like a copy of the API tests.

In traditional software engineering, we allowed these redundant tests. Hopefully, if the programmers forgot to test for something, the user acceptance testers would remember, and vice versa. Today, since the emergence of the Agile Manifesto, it is more common for people to actually talk to each other and brainstorm on test ideas. This is to create a shared understanding of what the software will be and how to test it. Most people consider that an improvement over what we did before, which was essentially that each party had a different interpretation of the specification, tested in different ways, argued at the end about what was "right," and probably rewrote the test in the end. That wasn't a great approach…

If you're in the second camp, by all means, stick with the redundant tests. If, however, the group as a whole mostly agrees on what the software would do, there can be a quick win: In addition to testing the logic (the pure API), the end-to-end tests should test the entire user experience, from login to whatever process ends the user flow.

You might slice and dice these steps into little pieces, but end-to-end tests still test whatever user interface elements need to be tested. What they do not have to do is test every *combination*. Once we know the user interface sends the contents of the textboxes and drop-downs to the API, we can likely skip every combination of the drop-downs. Instead, focus on the "happy path," error results from the API, and what the user interface does when the API is down. If you have pure front-end tests, they can check to make sure, for example, that the user cannot click Submit until the form is filled out.

Once we know the front end takes and receives API results that are properly formatted end to end, all the combinations and types of logically-different-but-need-to-be-done checks can go in the API tests. This will keep your end-to-end tests fast.

An even simpler approach, especially if your testers are less technical, is they can parameterize the tests and execute them in parallel which would get you to the same place, while further minimizing your test maintenance burden.

The Final Word on Retiring Tests

To review, there are two ways to decide what tests to retire. The first is to look at the pain the tests are causing (such as a delay of results or an increased effort to maintain) versus the value they provide, which could be bugs or the confidence that nothing huge broke. When tests become a liability that exceeds their value, it might be time to put them off to the side in a "deprecated tests" folder. After a year, if no one noticed they were missing, it might be time to let go.

Another approach is to look at the redundancy of tests, particularly if a test is checking logic, not wiring. In that case, it might be best to push it down in the API or an even lower level, where it runs faster and will be less brittle. If those lower tests already exist, it might be time to let the redundant tests go[2].

Retiring tests is not an easy decision. Done poorly, it could cause more harm than good. But letting them hang around could be harmful, too.

The only way to make the right decision is by gathering the data. What you find might surprise you.

Footnotes

[1] Running your tests in parallel can resolve this. It just turns out that many teams don't have the tooling to make that a reality. Even if the system can run in parallel, more automation generally leads to a higher maintenance burden. To be fair, some vendors have made it possible to design a test system that has a low maintenance burden and runs in parallel. That might mean you need a better technology stack. It is fair to say that if your tests take too long and break too often for the wrong reasons, and you can run them in parallel, then retiring them is less beneficial than *fixing* them.

[2] Here I speak of large, step-step-step-step-type-click-step-click-check results style tests that are repeated with different values. If you have the opportunity to combine them into a test set that looks like a table of inputs and expected results, then the only concern left is speed. That might be a clue that the user interface is too slow (human responses notice differences of longer than 0.25 seconds) - this could be a clue to work on the front-end API, not move the tests.