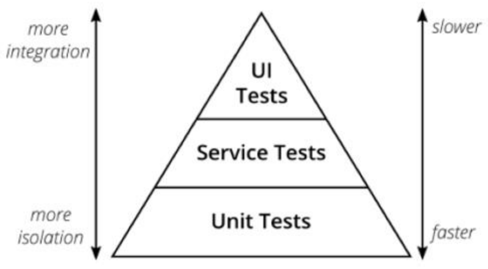
Should you Focus on Unit versus End-to-End Tests?

By Matthew Heusser matt@xndev.com and Subject7

The best part of a presentation might just be the Q&A. After all, that is the part where the material meets the actual life experiences of the audience. About a month ago, we hosted a panel discussion titled "Revisiting the Test Automation Pyramid - 20 years later." The panel attracted a lot of great questions, so we're delving deeper into some of the concepts explored by the panel. As a side note, if you missed the webinar, you can Find the Replay Here.



Question: "Let's talk about unit testing versus functional and E2E testing — how much of each type of testing should I do?"

This seems like the perfect follow-up question for a presentation on the test automation pyramid. How steep are the sides? What is the appropriate number of unit tests, API tests, and end-to-end tests?

We can provide guidance for figuring out how to shift testing, what to add more of, and what to do less of. To get there, I'll start at the very beginning, and go very fast.

The Measurement Problem

In order to answer how much, how many, or what ratios, we need a way to count and compare types of tests. A small unit test might take a minute or two to write and 10 milliseconds to run; a small end-to-end test might take a half-hour to create and a minute to run. In the time it takes a tester to create one end-to-end test, a programmer might create several unit tests.

Does that satisfy the pyramid's requirements of a broad base? Or are we just comparing apples to oranges?

Talking to the people who kicked around the original ideas, I've come to the conclusion that the intent was one of focus: The team should focus on the unit tests more than the higher-level tests. In other words, they should build a solid foundation first.

These ideas are conceptual advice, but they are not very practical. To figure out the ratio, we would need to measure the number of minutes people spend on activities at various levels, then compare high-functioning teams to low-functioning, and then eliminate other independent variables.

However, the reality is that no one does this sort of measurement at industry scale. In his landmark paper Software Engineering Metrics: What Do They Measure and How Do We Know?, Cem Kaner suggests we have so little agreement on what words mean - is the build-maintainer a developer? A tester? What about user experience specialists? Kaner goes on to argue that comparisons like figuring out a developer-to-tester ratio make little sense. Even with a survey of every company, the metrics would reflect an average of all the people who responded to the survey. Companies with an external API as a main product would certainly have a different ideal ratio from those that do electronic data interchange (EDI) as a business. Likewise, software as a service (SaaS) companies, like GitHub or Atlassian (who makes Jira), are going to have a different approach from insurance companies and banks that do data processing.

With that in mind, the question becomes less about the "right" industry-standard number of each type of test, or even the right number for your company. Instead, the question is, “should our team be doing more or less of each type of testing?”

Considering Context

Unit tests tend to find low-level, isolated problems, and also provide incredibly fast feedback to the programmer. Integration and API tests find problems gluing together components, often due to misunderstanding the protocol — how the information is transferred. End-to-end tests exercise the entire system, making sure the complete flow works for the customer.

The next question is: What kind of problems is your team experiencing *right now*?

It is tempting to get the data you need from a Scrum retrospective. But I find that people in Scrum retrospectives tend to focus on a single incident or something that has happened only a few times. What we want is an objective assessment that looks at what has happened over time, not what people experienced in the past two weeks.

Instead of a retrospective, I suggest looking at bugs found recently. That includes bugs found in testing and bugs that are found later by customers after they escape to production. This evaluation includes quantitative data, but also qualitative data from actually reading the bug reports, particularly for impact.

When I've done this, I've gone back somewhere between a month and six months and pulled the bugs that qualify into a spreadsheet, with a column for root cause and another for where the defect should have been found. Most recently I've just looked at the last hundred bugs. If your team does a "zero-known-bug" policy and doesn't write them up but just fixes them, you might need to write down what you find for two to four weeks.

One team I worked with recently had a great number of user interface bugs that were based on regressions — that is, a change in one piece of the code had unintended consequences in a different area of the code that seemed to be unrelated. It was a mobile application, and the screens used shared code libraries. The changes were visual and mostly concerned the look and feel of elements like text boxes and combo boxes.

Another team had mobile application problems that showed up mostly when testing with real devices, such as a sticky scroll on an iPhone, or changes in screen design that did not render properly when the screen resolution changed for various devices. This was an e-commerce application for a luxury brand, and as it turned out, iOS represented an extremely large percentage of actual dollars spent on checkout.

None of these bugs would have been caught by unit tests. In these cases, the focus clearly needed to shift to more user interface and end-to-end tests.

Another data point comes from looking at how often testers are waiting for a fix, how long that wait is, and if the problem could reasonably have been found by end-to-end tests. Waiting is one of the seven wastes of the Toyota Production System. In software, we tend to "paper over" waiting by having people work on other tasks, but the wait (and delay) is still there. Gathering data for this problem can be as simple as going to the daily standup and getting a list of stories (and bugs) the testers are waiting for, then working backwards to see if end-to-end tests would have found those defects or if unit tests could have found them.

Bugs that escape to production can be a little trickier. In some cases, the team needs additional end-to-end tests. In others, the problem was a lack of creativity and freedom within the testers — "Nobody thought to test that," or in some cases, "We thought to test that, but setup was hard, and the programmer said it could never happen."

The classic question "Why didn't QA find that bug?" is really unproductive here. Instead, the questions are "Can this bug reoccur?" and, if so, "How should we shift the test effort to find the defect?"

For one team I worked with, the bugs that escaped to production were relatively unique: a configuration flag was left pointing to test.companyname.com when an API was called, or a database indexer ran for too long and caused all requests from the database to block. These were big problems, but relatively rare, best addressed by policy and procedure. Certainly, a test running on test.companynname.com could never find that problem. In other cases, the problems in production were infrastructure-based and could have been found by making the change in a test environment, then running the end-to-end tests against that environment.

A final question to consider is how many of the important defects that are discovered are through exercising the entire system (often regressions), and how many are limited to new feature development. A great number of new feature bugs would imply the need for better exploratory testing, and perhaps, more unit tests. You would need to dig into the details.

End-to-end and GUI tests provide value when the workflow breaks. If testers spend a great deal of time dead in the water waiting for bug fixes, it's likely time for more end-to-end testing. On the other hand, I've worked with teams where the first build simply did not work at all, and the simplest combination of inputs rendered an error. In that case, it might be best to focus on unit tests, or at least programmer-testing.

From Theory to Action

Here's a place to start: Is the code quality on the first build delivered to test "good enough?" If not, then is the problem one of insufficient regression testing? Or do the issues stem from new features?

If the problem is insufficient regression testing, analyze whether the issues could be found by end-to-end testing. Test the login, search, creating a new page, creating a profile — errors along these lines that block the flow of work might be findable by end-to-end testing. Other problems, like a sticky scroll, might require more engineering work to build better code. If the problem is new feature development, examine whether the programmers should do more unit tests or better check their own work before passing it off.

These questions don't provide a percentage or ratio, but they can help set expectations for change.

On a full delivery team of 10 or more developers, analysts and testers, the next step is likely to take someone half-time to work on the area that needs emphasis. Often programmers feel like they need permission to do more unit tests (or write end to end tests themselves), while testers feel like they need permission to beef up the regression-check automation. Insisting on obtaining this permission will slow down the pace of delivery.

Still, if the heavier concentration on unit-testing was buggy, and those bugs delayed release, a heavier emphasis on end-to-end testing may result in less back-and-forth and better quality in less total time. Think of this as reducing the big, ugly delays by investing in a little more end-to-end testing. You may even end up going faster. That is a win.

Get winning.