Software Testing, Quality Assurance and Maintenance Winter 2017

Lecture 17 — February 10, 2017

Patrick Lam *version 1*

Engineering Test Suites

We are going to move onto the second part of the course. In the first part of the course, you learned ways to make sure that your test suites are exhaustive enough. In this part of the course, you will learn about making your test suites better as engineered artifacts.

Why tests? Let's start by talking about what test suites can do for you (as a developer).

Reference: Kat Busch. "A beginner's guide to automated testing."
`https://hackernoon.com/treat-yourself-e55a7c522f71`

Anecdote: "TODO: write tests." We're all busy, right? Surely tests are less important than writing actual code solve problems. And it can be hard to set up the test.

When you're writing code, you clearly want to know that it can at least work. So you may have a development setup. Kat Busch describes writing a Java server to interface with an Android app. Her development setup involved manual testing:

- set up test server on dev machine;
- install app on test phone;
- manually create a test case on test phone.

She writes: "Obviously I didn't test very many code paths because it was just so tedious."

Since Dropbox (her employer at the time) used code review, she actually had to write tests to pass code review, even if it was annoying to do so.

"Lo and behold, I soon needed to fix a small bug." But, of course, it's easy to introduce even more bugs when fixing something. Fortunately, she had some tests.

> I ran the tests. Within a few seconds, I knew that everything still worked! Not just a single code path (as in a manual test), but all code paths for which Id written tests! It was magical. It was so much faster than my manual testing. And I knew I didnt forget to test any edge cases, since they were all still covered in the automated tests.

Not writing tests is incurring technical debt. You'll pay for it later, when you have to maintain the code. Having tests allows you to move faster later, without worrying about breaking your code.

Another reason for tests: avoiding regressions. Code often lives in a web of dependencies; often, code isn't right or wrong on its own, but rather in terms of how it's used. "If you havent written tests, then theres no reliable way for other coders to know that their commit has impacted yours." The biggest codebases we deal in your courses are tens of thousands of lines, but industrial codebases are millions of lines. Even though you may have worked with them on co-op, you were only there for a short time, not years.

> If your code is still in the codebase a year (or five) after youve committed it and there are no tests for it, bugs will creep in and nobody will notice for a long time.

She continues with an anecdote about a user-facing feature broken due to a seemingly-unrelated change. Test cases can help prevent this kind of brokenness.

> **If it matters that the code works you should write a test for it.** There is no other way you can guarantee it will work.