Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 18 — March 4, 2019

Patrick Lam *version 1*

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 haven't written tests, then there's 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 you've 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.

Test design principles

Now that I've sold you on why you should have tests, let's talk about how your tests should be structured.

Some more references:

- Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.*
- Kent Beck. *Test Driven Development: By Example.*
- Roy Osherove. *The Art of Unit Testing: with examples in C#.*

Once again, I'm summarizing the article by Kat Busch, "A beginner's guide to automated testing." `https://hackernoon.com/treat-yourself-e55a7c522f71`

Many small tests, not one big test. My Assignment 1 Question 1 solution had 14 test cases. Each test case verified a different part of the behaviour. Out of 232 solutions, only 21 had fewer than 5 test cases. So most of you intuitively know this already. But why is this good?

- easier to deal with failures: you get a specific failed test case, ideally with a descriptive name. This is easier to debug.
- easier to understand what's being tested: again, because you used descriptive names, it's easier for future you, or someone else, to understand whether they need to add a new test case or not. Example: `test_integer_addition`, `test_integer_subtraction`, etc.

Make it easy to add new tests. I think that was the case for you on A1Q1 eventually, based on my `testEmpty` method. You just had to emulate that test case: create inputs; create a `FormattedCommandAlias`; and assert against the result. Using @Before and @After setup and teardown methods can help too (but was overkill for our case). "You should always make it easy for people to do the right thing."

Unit versus integration tests. Unit tests are more low-level and focus on one particular "class, module, or function". They should execute quickly. Sometimes you need to create fake inputs (or mocks) for unit tests; we'll talk about that too. You should generally not use an entire real input for a unit test.

As we discussed before, many interesting things happen when different units interact. Integration tests verify end-to-end functionality. They're slower, flakier, and harder both to write and use. Focus on unit tests while developing code. But you eventually need integration tests to make sure the user sees the right hting.

"TODO: write tests." Should you write the tests first? Test-Driven Development says, well, tests first, and only write as much code as needed to pass the tests that you write. A middle ground might be good: write some code, write some tests, repeat.

> You'll find that writing tests as you go makes your interfaces better and makes your code more testable. If you find yourself writing something hard to test, you'll notice it early on when there's still time to improve the design.

Flaky tests are terrible. Although your A1Q1 tests should have been deterministic, not all tests are deterministic. Some tests fail a small percentage of the time.

- timeouts can fail when something takes surprisingly long;
- iterators can return items in random order;
- random number generators are, well, random.

Try really hard to make your tests not flaky. There are ways of dealing with them, but they're not good.

Looking inside the system under test. Even in A1Q1 I had to expose some of the internal state, namely the command that actually got executed by the `commandSender`. As much as possible, try to avoid testing internal state, but rather only what is externally visible. This makes your tests more resilient to rewrites, and also focuses the test on what actually matters.