Software Testing, Quality Assurance and Maintenance Winter 2019

Lecture 25 — March 20, 2019

Patrick Lam *version 2*

More on Flaky Tests

Google also has a blog post about their experience with flaky tests. It provides a different perspective on flaky tests.

`https://testing.googleblog.com/2016/05/flaky-tests-at-google-and-how-we.html`

Here are some useful facts from that post:

- Rate of flakiness: 1.5% of all test runs. Over time, their flaky test insertion rate is about the same as the flaky test removal rate. The number of *tests* which are occasionally flaky is 16%.
- Code can only be submitted after it passes tests. There are also pre-release test suites which must succeed before the project can be released. This is presumably after correcting for flakiness.
- The vast majority (84%) of post-submit test failures in continuous integration are associated with flaky tests.
- The flaky test rate means that if you have 1000 pre-release tests, you should expect 15 failing tests which require manual investigation. If you ignore them, you might well be missing an actual problem.
- Mitigation: in addition to what I mentioned last time (automatic re-running), Google also allows re-running only flaky tests. Known flaky tests get automatically re-run 3 times before reporting a fail. This slows down notification of test failures, too. Also, tests that are seen to be flaky are automatically quarantined (not run every time; bug reported).

Case Study: Mocks/Stubs

Next, we'll summarize another Google case study, this time about mocks/stubs.

`https://testing.googleblog.com/2016/11/what-test-engineers-do-at-google.html`

This blog entry describes the work of a Google Test Engineer in improving test infrastructure.

Situation. Legacy system. Flaky tests: large and brittle. End-to-end tests were difficult to introduce fakes into. System used many external dependencies.

First false start. Splitting the end-to-end tests. Didn't work: would have needed refactoring for entire legacy system, not just the part the author's team was working on.

Second false start. Mock services that were not actually required. Not viable: dependencies changed often. Imposed test maintenance cost.

Actual solution. Replace the client code (which calls depended-on services) by unit tests of calls to RPC stubs. Stub implemented with mock objects. Then, in another test, send the data to the actual service (i.e. test the tests).

The benefits are much faster tests (10× speedup: from 30 minutes to 3 minutes) which are not at all flaky and which can run on developer machines.

Continuous Integration

This is not a complicated concept. Literally, continuous integration requires you to use a single shared master branch with your development teams. That is integration because you're merging your changes into master, and continuous because you're doing it all the time.

Why CI Is Awesome. Before CI, people could be stuck integrating changes for months (or longer!) after each team finishes developing their change. This is not good. Instead, your software always stays in a working state.

Necessary CI Practices. You can't just do CI, though. It's not new, but there was a time before CI, because the infrastructure didn't exist yet. You need *continuous builds* and *test automation* to make CI work (and, of course, a source control repository). You can then use CI to do Continous Deployment. Here's how CI works.

1. You clone the repo (which works).
2. You make your changes.
3. You commit and push your changes (often!)
4. A machine pulls the changes, compiles them, and runs automated tests.
5. Everyone knows whether your changes passed tests or not.

Continuous Deployment is a minor variant to CI where the production machines also pull changes as soon as the tests pass, and deploy them.

Key Details. To make Continuous Integration work, you'll probably have to follow these practices as well:

- Fix broken builds immediately! (Commits shouldn't even be accepted if they don't compile; test cases should start immediately and fixing them is a high priority task).
- Keep the build fast (minutes): parallelize the build and tests. Deploy tiered tests: fast tests run first, then slower, more detailed tests.
- Test in a production-like environment. Virtual machines are helpful here.

References about Continuous Integration.
Bullet points from Gitlab: `about.gitlab.com/2015/02/03/7-reasons-why-you-should-be-using-ci/`

Mid-length article from Atlassian:
`www.atlassian.com/agile/continuous-integration`

Longer article by Martin Fowler:
`martinfowler.com/articles/continuousIntegration.html`

Serverless CI:
`medium.com/@hichaelmart/lambci-4c3e29d6599b`

Airbnb Testing Infrastructure

Let's change it up here and talk about Airbnb instead.

`http://nerds.airbnb.com/testing-at-airbnb/`

This describes how the author worked with colleagues to introduce a (developer-based) testing culture at Airbnb.

Running tests locally. As I mentioned earlier, running tests in prod-like environments helps a lot. Airbnb did so using Vagrant boxes ("ready to run tests out of the box.") Airbnb runs on Ruby on Rails and they use Zeus to allow developers to start up the Rails environment super quickly.

Continuous Integration. They also use Solano for Continuous Integration (as described in the L24 notes). Tests run at Airbnb (not in the public cloud), in parallel for improved throughput.

Furthermore, their Github displays the build/test status of every pull request, and presumably developers only merge pull requests that pass the tests.