The Software Engineer in Test (SET)

writing feature code Or
writing test code

Thinking involved in writing feature code and writing test code

- Google SWE
  - **feature developer**
  - building components for customers
  - write feature code and test code
- Google SET
  - **test developer**
  - assist SWE with unit test and larger test frameworks in small and medium tests
- Google TE
  - **user developer**
  - automation for user scenarios

The Life of an SET

- Development and Test Workflow
- Early Phase of a Project
- Team Structure
- Design Docs
- Interfaces and Protocols
- Automation Planning
- Testability

Openness of codebase

- Most code at Google shares a single repository and common tool chain
- These tools and repository feed Google's build and release process
- All Google engineers, are familiar with this environment
  - checking in new code
  - submitting and executing tests,
  - launching a build

Guide for the shared infrastructure

- All engineers must reuse existing libraries
- All shared code written first
- Shared code must be reusable and self-contained
- Dependencies are surfaced and impossible to overlook
- Take code reviews seriously, especially with common code
- Code in shared repository is with a higher bar for testing

Simple and Uniform Google development platform

- a common linux distribution
  - engineering workstations
  - production deployment machines
  - centrally managed set of common, core libraries
  - common source build
  - test infrastructure
  - single compiler for each core programming language
  - language independent commmon build specification
  - **respect and reward the maintenance of these shared resources**

single platform, single repository, a unified build system

- A build specification language
  - independent of a project's language
  - share the same "build files"

The flow of a build (if TDD, step 3 proceeds Step 1 and 2)

1. Write a class or set of functions for a service
   a. make sure all the code compiles
2. Identify a library build target for this new service
3. Write a unit test importing the library, mocking out its nontrivial dependencis
   a. executes the most interesting code paths with interesting inputs
4. Create a test build target for the unit test
5. Build and run the test target
   a. making necessary changes until all the tests pass cleanly
6. Run all required static analysis tools
7. Send the code out for code review

What the roles SETs play ?

- SETs are the engineers involved in enabling testing at all levels of the Google development process
- Software Engineers in Test
  - 100% coding
- Test is just another feature of the application
  - SETs are the owner of the testing feature

Early Phase of a Project

- Common scenario for new project creation is from informal 20 percent effort
  - Gmail and Chrome OS are such examples
  - Quality is not important until the software is important
- Google doesn’t make specific attempts to get testers involved early in the project lifecycle

Early life of a project

- No project gets testing resources as some right of its existence

SWE and SET in a Team

- SWEs tend to make decisions optimized for local and narrow view of a product
- SETs take the opposite approach
  - assume not only a broad view of the entire product
  - consider all features over a product's lifetime
- SWEs may come and go
  - product will outlive the people who created it

Testing at the Speed and Scale of Google: Continuous integration systems

1. Get the lastest copy of the code
2. Run all tests
3. Report results
4. Repeat 1-3

Software Development at Google

- 20 changes per minute and 50% files change every month

Test Certified

- Level 1
  - Set up test coverage bundles
  - Set up a continuous build
  - Classify your tests as Small, Meduim, and Large
  - Identify nondeterministic tests
  - Create a smoke test suite

Level 2

- No releases with red tests
- Require a smoke test suite to pass before a submit
- Incremental coverage by all tests >= 50%
- Incremental coverage by small tests >= 10%
- At least one feature tested by an integration test

TDD

- Red-Green-Refactor
  - Write some skeleton code that compiles, has enough API to be testable.
  - Write tests which -- initially -- will be mostly failures. Red.
  - Finish the code. The tests pass. Green.

Level 3

- Require tests for all nontrivial changes
- Incremental coverage by small tests >= 50%
- New significant features are tested by integration tests

Level 4

- Automate running of smoke tests before submitting new code
- Smoke tests should take less than 30 minutes to run
- No nondeterministic tests
- Total test coverage should be at least 40%
- Test coverage from small tests alone should be at least 25%
- All significant features are tested by interation tests

Level 5

- Add a test for each nontrivial bug fix
- Actively use available analysis tools
- Total test coverage should be at least 60%
- Test coverage from small tests alone should be at least 40%