Testing!

Test-Driven Development (TDD)

- Test-Driven Development implies *writing tests first, then write code!*

- There are several benefits to TDD, including but not limited to:

    1. Well tested code leads to *more robust code resilient to regressions*
        - As bodies of code evolve over months and years, having confidence you're not breaking something you forgot about, or even worse you didn't even know existed because it's someone else's code, is important.

    2. Writing testable code *improves the design of your code*
        - Promotes smaller functions with single purposes
        - Promotes pure functions (given the same arguments always result in same return value)

    3. Writing good tests speeds up your ability to complete a project
        - Minimizes the feedback loop between need to add feature and being feature complete
        - Counter intuitive because it can slow down your "out of the gate" progress

Two Important Kinds of Tests

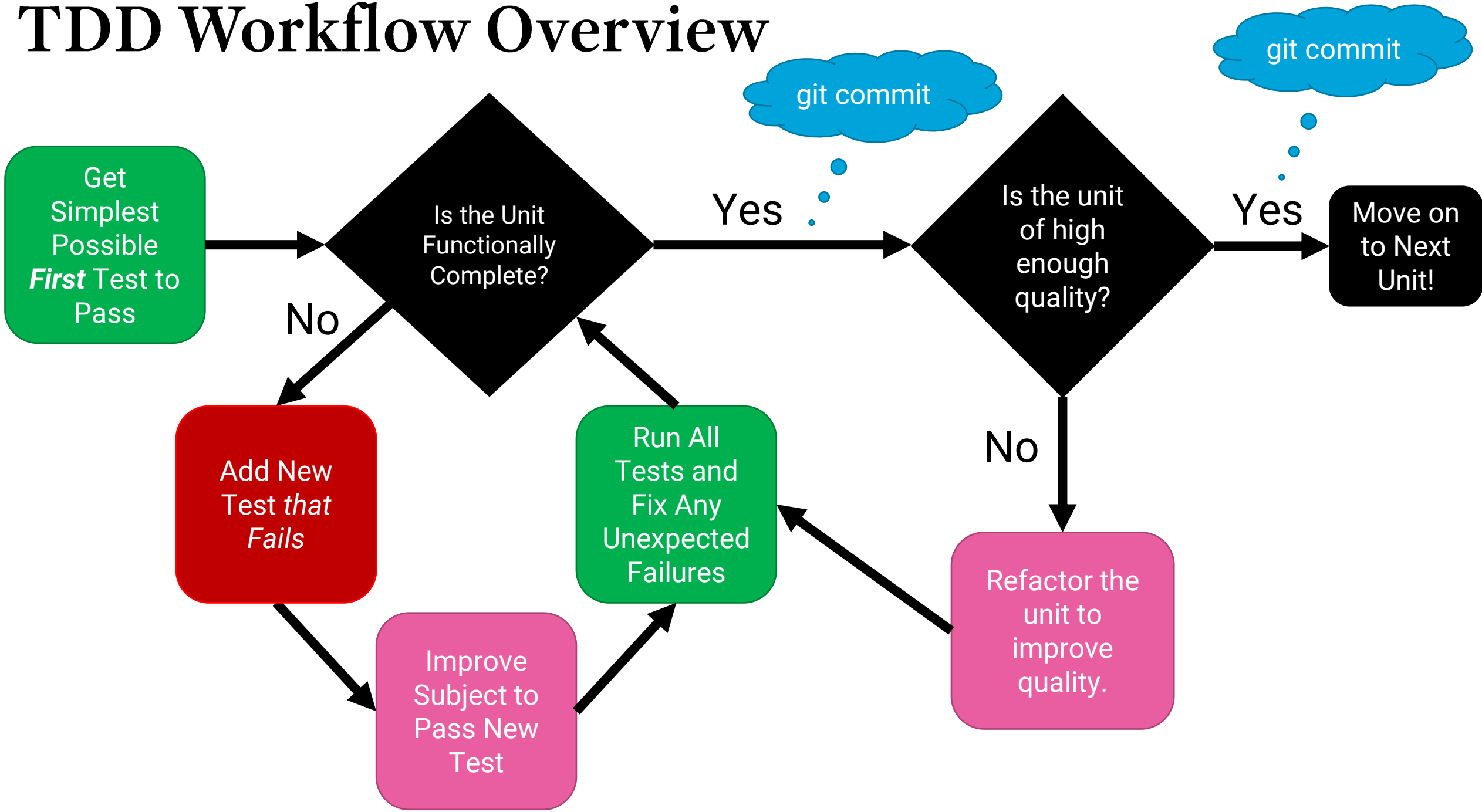
- Unit Tests
  - The word *unit* refers to the test subject's *unit of code*
  - Units of code that are well suited for unit testing: *functions and methods*
  - Unit tests should test the subject with as much *isolation* as possible
    - If you cannot easily isolate the test subject, it's often a hint your design needs work.
    - How can you break your solution down into more testable units?

- Integration Tests
  - Tests the entire system or subsystems of many units composed
  - Typically these *run the program from the user's vantage point*
  - Generally much more challenging to write and more fragile than unit tests
    - *Especially* for applications that involve graphical user interfaces!

Starting work on a unit of functionality in TDD

- Test-Driven Development implies *writing tests first, then fixing!*
- When writing the first test case for a new unit of functionality, focus on:
    1. The **simplest test case possible** (often an edge case that requires 0 effort to pass)
    2. The **stub of your test subject**
        - i.e. A working function definition that always returns a dummy value of the correct type.
- Start with ***just enough*** of those two items for the ***test to actually run***
    - In compiled languages, like Rust & Java, this means your code must actually compile
    - In interpreted languages, like Python & JavaScript, this means you shouldn't get "function not defined" exceptions when your tests are running

TDD Workflow Overview



TDD Workflow Observations

- **There are *two feedback cycles* in this process**
  1. Getting to a good, complete suite of tests (that barely pass)
  2. Getting to a good, complete unit of code that's well implemented

- **How do you know when you've got enough unit tests?**
  1. When you've exhausted ideas of tests for unique cases to cover
     - The edge cases of a problem as well as
     - Expected use cases with emphasis on cases that involve inductive steps
  2. When you've exhausted ideas for coming up with the next test that fails your current implementation

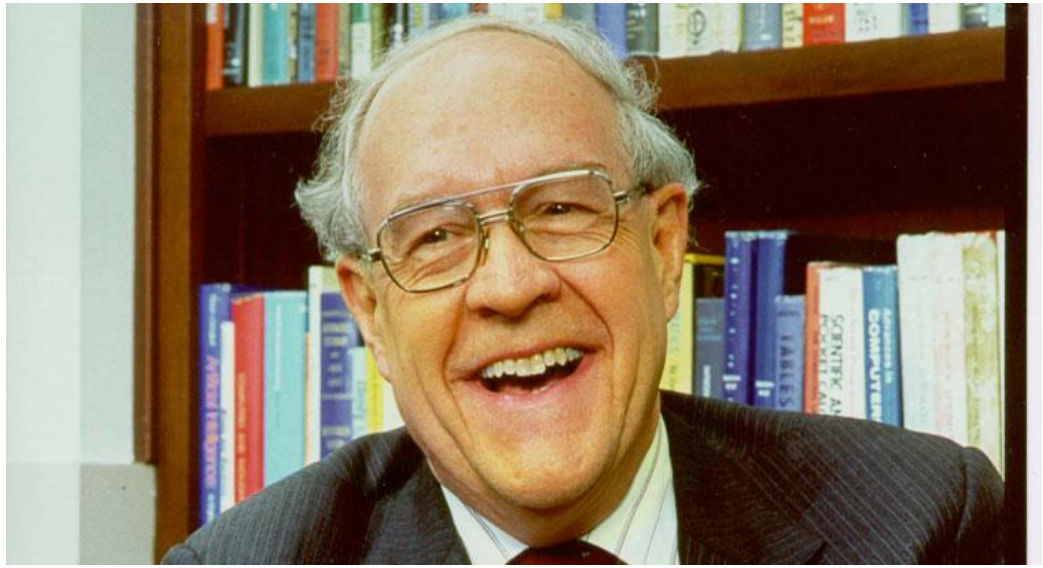
- **The tighter and faster you can make complete trips around each of these two feedback cycles the more joy you'll experience in the process!**

Aside: Software Engineering Ergonomics

- A common differentiator between average teams and great teams is how well thought out their development workflows are.
- If the experience of writing and running tests is painful for a project, software engineers will tend to do a poorer and slower job of testing.
- Investing time in a project's workflow and tooling is a force multiplier.
- Great development environments should be designed to make the *common tasks fast, effortless,* and *joyful*.

Some disclaiming notes on TDD

- Tests have quickly diminishing returns once you've covered the important cases
  - Redundant tests can become a net negative
- Testing for printed output in unit tests is nontrivial. Best to refactor out a pure function that returns what is to be output and test the pure function.
- Some code paths are *really difficult* to test and unless it's mission critical to handle those cases gracefully it's generally OK to skip:
  - For example: test cases that cover out of memory paths.
- When TDD is done right it should *reduce frustration* and *increase confidence*. *Not vice-versa!*
- In real world projects there's a danger in jumping to testing before you actually know the "shape" of what it is you're trying to implement. Prototyping without testing is OK, but be sure to throw away your prototype.



The management question, therefore, is not *whether* to build a pilot system and throw it away. You *will* do that. […] Hence *plan to throw one away; you will, anyhow.*
- Fred Brooks

Dan Abramov
@dan_abramov

Follow

TDD paralyzes me. I'm all for writing tests early in the process — especially in library code. But I can't write them before I *play*. I need to write a shitty draft and play with the behavior to understand what I really want. Then rewrite guided by tests.

7:23 PM - 18 Jan 2019

1,304 Retweets **6,878** Likes

319 1.3K 6.9K

Tweet your reply

Dan Abramov @dan_abramov · Jan 18
Writing tests too early creates inertia. I get attached to them because they help verify correctness. Throwing them away or disabling feels like a step back, or giving up.

But during the design process, a step back often *is* what I want. I might be building the wrong thing.

16 55 652

Dan Abramov @dan_abramov · Jan 18
Tests help me verify it works right. Playing lets me verify it *feels* right.

16 37 531