Little Languages

Lecture 07:

Test-Driven Development

Before lecture: **Start VM and pull 590 materials from upstream**.
Then…

```
$ cd comp590-material-<you>
$ git pull upstream master
$ cd comp590-material-<you>/lecture/07-regex
$ cp 590.vimrc ~/.vimrc
```

Test-Driven Development (TDD)

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

- There are a number of 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

Hands-on: Let's use TDD for **`index_of(&str, char)`**

- Let's setup a new Rust project in today's lecture directory:

```
$ cd ~/590-material-<you>/lecture/07-testing

$ cargo new --bin index_of
$ cd index_of
$ vim src/main.rs
```

- Add the code to the right below main...

- Test with vim command: **`:RustTest!`**

(press ZZ to close tests)

```rust
#![allow(unused)]

fn main() {
    println!("Hello, world!");
}

fn index_of(haystack: &str, needle: char) -> Option<usize> {
    None
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn none_match() {
        assert_eq!(None, index_of("a", 'b'));
    }
}
```

Rust's testing story is *built-in.*

```
#![allow(unused)]

fn index_of(haystack: &str, need                    ize
    None
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn none_match() {
        assert_eq!(None, index_of("", 'a'));
        assert_eq!(None, index_of("a", 'b'));
        assert_eq!(None, index_of("ab", 'c'));
    }
}
```

This attribute tells the Rust compiler the **tests** module should *only* be compiled in **test** configuration. No cost in debug/release builds.

Import all names in the outer module's scope. In this case, it allows us to call **index_of** directly.

The **#[test]** flag hints to the compiler this function is a test case.

The **assert_eq!** macro tests for equality between some *expected* result and an *actual* result.

When there is not equality, it panics and test fails.

Cargo has test running support built-in.

Run Tests in Command Line Shell

Run All Tests

```
$ cargo test
```

Run Specific Tests

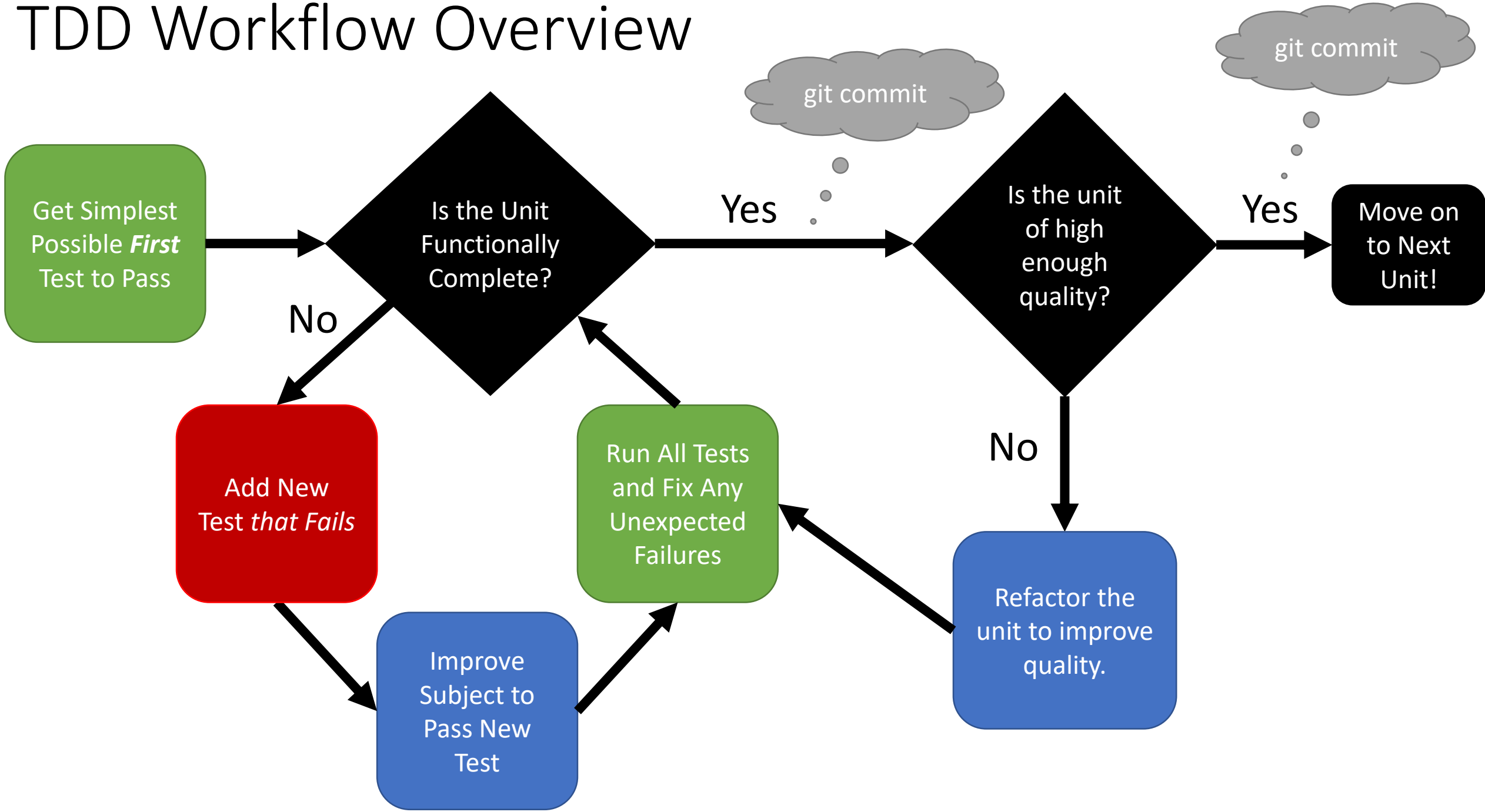
```
$ cargo test 'test_name_substr'
```

Run Tests in vim

All Tests **:RustTest!**

Test Under Cursor **:RustTest**

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.*

Aside: Improving Testing Ergonomics in **`vim`**

- One of the reasons many software engineers love **`vim`** is thanks to how customizable its workflows can be.

- Similar to how you can record macros on the fly, you can *remap* keyboard shortcuts to make the sequences of commands you commonly use in a projects into "one button" commands.

- For example: "Save all files *and then* run the Rust test under my cursor."
  - We can make a button for that.

- This is more possible in vim than most IDEs because it is driven by a little language.

Installing updated 590 `~/.vimrc` Shortcuts

```
$ cp ~/590-material-<you>/lecture/07-testing/590.vimrc ~/.vimrc
```

- Your ~/.vimrc file is loaded when vim starts and is responsible for:
  1. Loading custom plugins and extensions
  2. Non-default settings for built-in vim features (i.e. showing line numbers)
  3. Custom key mappings to carry out actions

- The file you just installed added a few new custom key maps that will help you this semester.

- Example mappings:

```
" Shift-T saves all files and runs rust test under cursor
nmap T :wa<CR>:RustTest<CR>

" Ctrl-t saves all files and runs all tests
nmap <C-t> :wa<CR>:RustTest!<CR>
```

590's ~/.vimrc Key Mappings

Ctrl-o - Open File Explorer

Shift-T - Run test case under cursor

Ctrl-T - Run all test cases

Ctrl-g - :RustRun
You can optionally add CLI arguments
You must press Enter.

Ctrl-x - Save current vim session, exit

$ vim -S Session.vim - Resume a vim session
Must be run from CLI *after* exiting with Ctrl-x
The flag must be a capital S

vim Window Control

:vsplit - Split the window vertically

:split - Split the window horizontally

Ctrl-ww - Cycle between windows

Ctrl-w(h|j|k|l) - Move to window
Move to left, down, up, right.

ZZ - Close split window (and save)

Follow-along: Add a Simple Failing Test

- This test feels easy to handle...

```
#[test]
fn exact_match() {
  assert_eq!(Some(0), index_of("a", 'a'));
  assert_eq!(Some(0), index_of("b", 'b'));
}
```

- Let's add a naive solution that passes this test...

```
fn index_of(haystack: &str, needle: char) -> Option<usize> {
    let haystack: Vec<char> = haystack.chars().collect();
    if haystack[0] == needle {
        return Some(0);
    }
    None
}
```

- Then, confirm all tests pass.

Hands-on: Add another test case that fails…

1. Add the next most challenging test case you expect will fail.
2. We've handled the cases of **None** and an **N** of **1**, try aiming for testing the **inductive** case of a match occurring at **N+1** positions.
3. Then, try to get your test to pass by improving **index_of**.
4. Need help? We'll be circulating around and you can work together!
5. Check in when you've got a failing test case and update your response when you've you're passing your failing test case. **pollev.com/compunc**

Is the unit of high enough code quality?

Remember: Before refactoring, run :RustFmt and make a `git` commit!

- Now that you've got tests to cover expected and edge cases, how can you improve the quality of your code? A few ideas to get started with:

1. Rename your variables to have more meaning. Turn magic, literal values into constants.

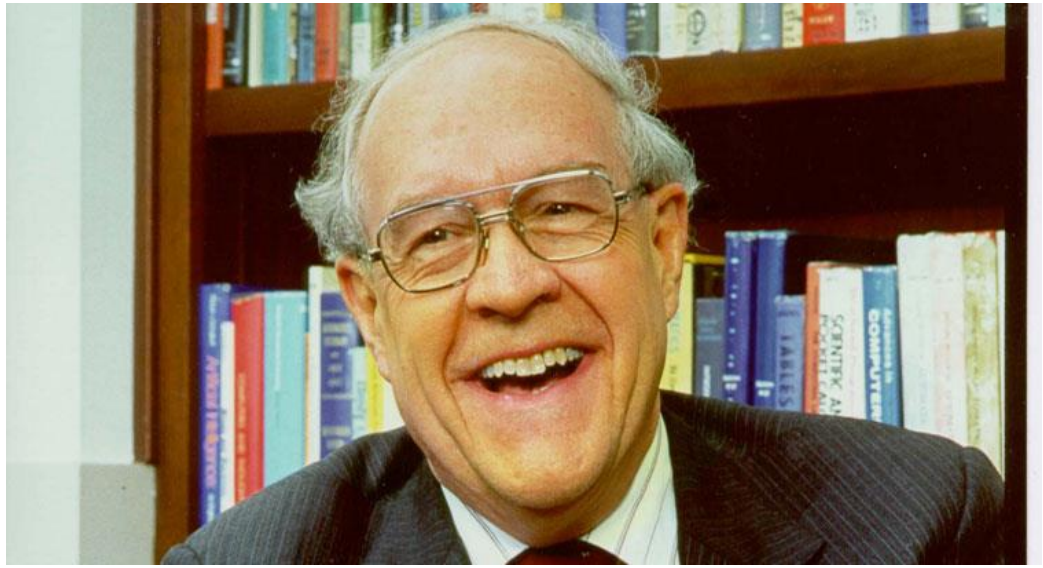
2. Look for complex logic, deeply nested conditionals, nested loops
   - How could you disentangle the mess and clean it up?

3. Are you being wasteful in either *time* (performing unnecessary work) or *space* (using memory unnecessarily)?
   - Example: Unless you absolutely must *look back* or *randomly access* data, you don't need to collect values from an iterator into a vector! Rely on the iterator alone.

- Now that you have solid tests, decide on something to improve, do it, run all tests (Ctrl-T, ZZ) and once back to passing all, repeat!

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 our a pure function that returns a String 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 failed system calls.
- 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