TDD - External

What is TDD?

TDD stands for *Test Driven Development.* In brief, it means writing tests *before* you write your implementation code. You use your tests to prescribe how a unit of code or system *should* function from an outside perspective, and then you write implementation code to match that prescription.

Why do we TDD?

We TDD for two reasons, to **tighten feedback loops** and **de-risk change.**

Let's get one thing out of the way, everyone tests their code. Whether you're running a local environment to check if a button works, spinning up a local copy of an API, deploying to a dev environment for QA to regression, or deploying straight to production, your code gets tested. What makes TDD stand apart is that the feedback on functioning code is much *faster*.

A good test suite gives you the confidence that your new change doesn't break existing functionality. It reduces the amount of regressions you'll see on deployed software. TDD and CI/CD go hand in hand. If you want the ability to move fast and incrementally, you need good tests. Note that **de-risking change** isn't the same as *eliminating* it. You will have production fires and bugs even if you use TDD. But after you fix them, you'll write a test and never see them again.

Code As Living Documentation

It's a cliche for sure, but your test suite truly acts as living documentation for your code. New developers can use test code to understand how it functions. When presented with a challenging piece of implementation code, good tests empower you to edit that code and see what breaks. You can use the test code to explore the development of a domain over time, or follow along with an entire business flow. Good tests give new developers confidence.

TDD is Good For Developer Morale

Remember learning to code for the first time? Writing something like a hello world program? Programming is fun! Especially when you can see the positive effects you're effecting as you go. In a non-tested or poorly tested codebase with long startup and lots of dependencies, that visibility disappears pretty fast. It could be minutes or hours before you see your software work, and programming becomes a slog. TDD lets you see it work (almost) immediately, and you get a little dopamine on every green run.

Common Counterpoints

Why not write tests after?

Writing tests before implementation means our frame of reference for code is always coming from how that software needs to be used and interacted, rather than how it works internally.

A key component of a well written test suite is that it is *implementation-unaware.* This is important because implementation aware tests can easily devolve into a brittle test suite that does not serve its primary purpose, which is to protect the *business* functionality of software, not to enforce a particular design choice. By writing tests before your implementation, you cannot write implementation aware tests. A well written unit test should allow you to change the implementation of the code (make it faster, change the pattern, etc) without altering the test.

It's not impossible to write good unit tests after the fact, it's just harder. You're fighting your own bias that way, and you don't get the feedback as you code.

Doesn't writing tests mean you take twice as long?

Not really. Remember, everyone tests their code. What's faster, writing a ten line unit test to check that button you just implemented, or restarting your app and then trying it by hand? Honestly it's probably faster to test it by hand once, but what about when you make a change and have to test again? What about when you want to deploy that artifact to production? What about six months down the line, how do you verify that button still works in each of three deployed environments? Are you going to check them all? What about 80 buttons?

Look, for small projects or small teams, you can probably get away without testing to a point. If you've got smart people who code carefully, it *can* work. You're not going to see a group of college students doing TDD on a class project. But in the enterprise software world (where we make our money) code will stick around for years, be deployed thousands of times, and worked on by tens or hundreds of people. Once the reliability of a piece of software comes into question, it adds a thousand pound weight to the team that maintains it.

How do we advocate for TDD?

Testing Pyramid

Red Green Refactor

Arrange Act Assert

Strong Opinions To Be Justified Later Or Not At All

- All your tests should run in the pipeline before deploys.
- You should never mock library code.
- You should never make something public in a class just so you can see it in a test.
- You should avoid inheritance in tests.
- If you can remove a piece of code, and your tests still pass, delete that code or update your tests.
- Tests should be named in short English sentences. `this_should_be_ThrownInTheTrash`
- Your test suite should optimize single-test readability.
- Code coverage tools are an anti-pattern.
- If your codebase is on fire, and you can only save the tests or the implementation, save the tests.