Test-Driven Design (TDD) Primer

This document is designed to help you better understand how to use the red-green-refactor cycle to do test-driven design (TDD).

This document is a “short” overview of the **automated unit tests** that software developers create as **an integral part of writing code**.

TDD is likely unlike anything you have done before. It takes some time to get used to, but provides huge (yes, I mean this big of hyperbole) benefits.

Table of Contents

First steps

My experience is that students and professionals do NOT understand what TDD is about until the SEE it in action.

Before you read the theory below, watch the first episode of James Shore’s Let’s Play TDD screencasts in which he creates a real project via TDD:

#1: How Does this Thing Work, Again? [6 mins]

Watching the next three will also help you SEE how TDD actually works in practice:

#2: Peering Dimly Into the Future [11 mins]
#3: Cleaning Up My Mess [15 mins]
#4: Gaining on Capital Gains [15 mins]

Key things to know about unit testing

Unit tests are **small**. Often just a few lines of code.

Unit tests **test code**. Not data. Data tests are different from unit tests, even if they happen to use the same xUnit testing framework. Testing code and testing data use different methods and test different types of considerations.

Each unit test **checks a single code path** in the **production code**.

A **code path** is a particular sequence of instructions executed by a computer. Consider the following code snippet:

```
1:  If (condition_A) {
2:      Foo = 1;
3:      redoTheTing(Foo);
3:  } else
4:      Foo = 2;
```

That code has two code paths, and thus two unit tests:

- One unit test would have condition_A be true and thus execute lines 1, 2, and 3
- A second unit test would have condition_A be false and thus execute lines 1 and 5

Thus, there should be a minimum of two unit tests for the above code.

Production code is code that runs on a “server” that provides the needed functionality to actual end-users. Production code effects the end-user directly. Test code is not put on the production computer. Unit tests are for use by software developers and the build pipeline to use to check their system before deploying the working code to a server.

Two unit tests should not test the same code path, unless they are testing the edge cases within a partition (see Equivalence Partitioning and Boundary Testing)**.**

Test-Driven Development (TDD) (see below) is a well-known and extensively practiced way to write code. In TDD, developers follow the red-green-refactor pattern to create unit tests in order to **guide the design of the code they are writing**. Each test helps the developer design the next small part of code and gain confidence that the code is working appropriately.

Unit testing is just one form of testing. Software systems require a wide range of types of tests to ensure their quality.

Writing automated unit tests has become a standard part being a software engineer. Why? Because automated unit tests enable continuous integration / continuous deployment (CI/CD) to work. And CI/CD pipelines provide huge value in software development. For instance, the 2017 State of DevOps Report notes that compared with lower-performing teams,

high performing teams use DevOps to deploy code much more quickly with much higher quality; high performing teams have (see p. 21):

- 46 times more frequent code deployments
- 440 times faster lead time from commit to deploy
- 96 times faster mean time to recover from downtime
- 5 times lower change failure rate (changes are 1/5 as likely to fail)

What is a “unit test”? What is “unit testing”?

There are many, many “types” of testing. The Guru99 site, for instance, lists 100 types of software testing:

- acceptance
- accessibility
- active
- agile
- age
- ad-hoc
- alpha
- assertion
- API
- all-pairs
- automated
- basic path
- backward compatibility
- beta
- benchmark
- big bang integration
- binary portability
- boundary value
- bottom up
- branch
- breadth
- black box
- code-driven
- compatibility
- comparison
- component
- configuration
- condition coverage
- compliance
- concurrency
- conformance
- context driven
- conversion
- decision coverage
- destructive
- dependency
- endemic
- domain
- error-handling
- end-to-end
- endurance
- exploratory
- equivalence partitioning
- fault injection
- formal verification
- functional
- fuzz
- guerrilla
- gray box
- glass box
- GUI software
- globalization
- hybrid integration
- integration
- interface
- install/uninstall
- internationalization
- inter-systems
- keyword-driven
- load
- localization
- loop
- manual scripted
- manual-supported
- model-based
- mutation
- modularity-driven
- non-functional
- negative
- operational
- orthogonal array
- pair
- passive
- parallel
- past
- attrition
- performance
- qualification
- ramp
- regression
- recovery
- requirements
- security
- sanity
- scenario
- scalability
- statement
- static
- stability
- smoke
- storage
- stress
- structural
- system
- system integration
- top-down integration
- unit
- user interface
- usability
- volume
- vulnerability
- white box
- workflow

Unit testing is just one of those types.

Before we get to the process of creating unit tests, let’s look into the what a “unit test” is.

A “unit test” is a test that covers a unit. Here’s an example from Vladimir Khorikov’s book *Unit Testing Principles, Practices, and Patterns*, annotated to illustrates several key features:

Unit tests follow the Arrange Act Assert (AAA) pattern:

Arrange	Create the test inputs, variables and objects needed to do the test.
Act	Use those inputs to run a part of the “production code”. **Production code** is code that will be run on a production computer that provides the needed functionality to actual users. Unit tests are NOT production code; while they are run by developers and as part of the automated CI/CD build process to help check that the code is working appropriately, unit tests in and of themselves are not “run” by users and thus do not belong on production machines. Project repositories often have production code in one folder, and test code in a different folder.
Assert	Check the output of the act step to see whether the state is as expected. The above example first checks the returned output of the customer.Purchase method. It then calls a method on the store to check whether the inventory is as expected.

Each unit test is **self-contained**. It contains all of the coded need to create the inputs it needs, run the production code directly, and check the resultant state.

Each unit test is **independent** of other unit tests. Unit tests should not use information from another unit test. That would create dependencies between unit tests, which would mean that changing one unit test might break another unit test - not what we want. Having each unit test

be self-contained dramatically simplifies each unit test. It allows one to add, modify, or delete a unit test without worrying about whether it will impact any other unit test.

Each unit test typically **tests exactly one code path** of one method. If another code path needs to be tested, created another unit test. By code path, we mean the set of production code statements that are executed via the Act section.

Unit testing “code smells”

“Code smells” is a term of art that refers to things about how code is written that “smell bad”, that suggest that the code design *might* have a problem. When one detects a code smell, it’s good to investigate to see whether there is an issue or not.

Here are some unit testing code smells indicating things to explore:

- Two unit tests test the same code path. Probably want to delete one of those unit tests.
- The Act section contains multiple lines of code. That might be okay, but does suggest that the abstraction (aka “API”) for the part of the production code might not be as clean as one wants.
- A unit test includes a conditional.
- A unit test tests multiple “similar” values, such as by looping over values stored in an array. Ideally, each value is tested by its own unit test, so that the full set of problematic inputs will be shown in a failed build.

How big is a unit test?

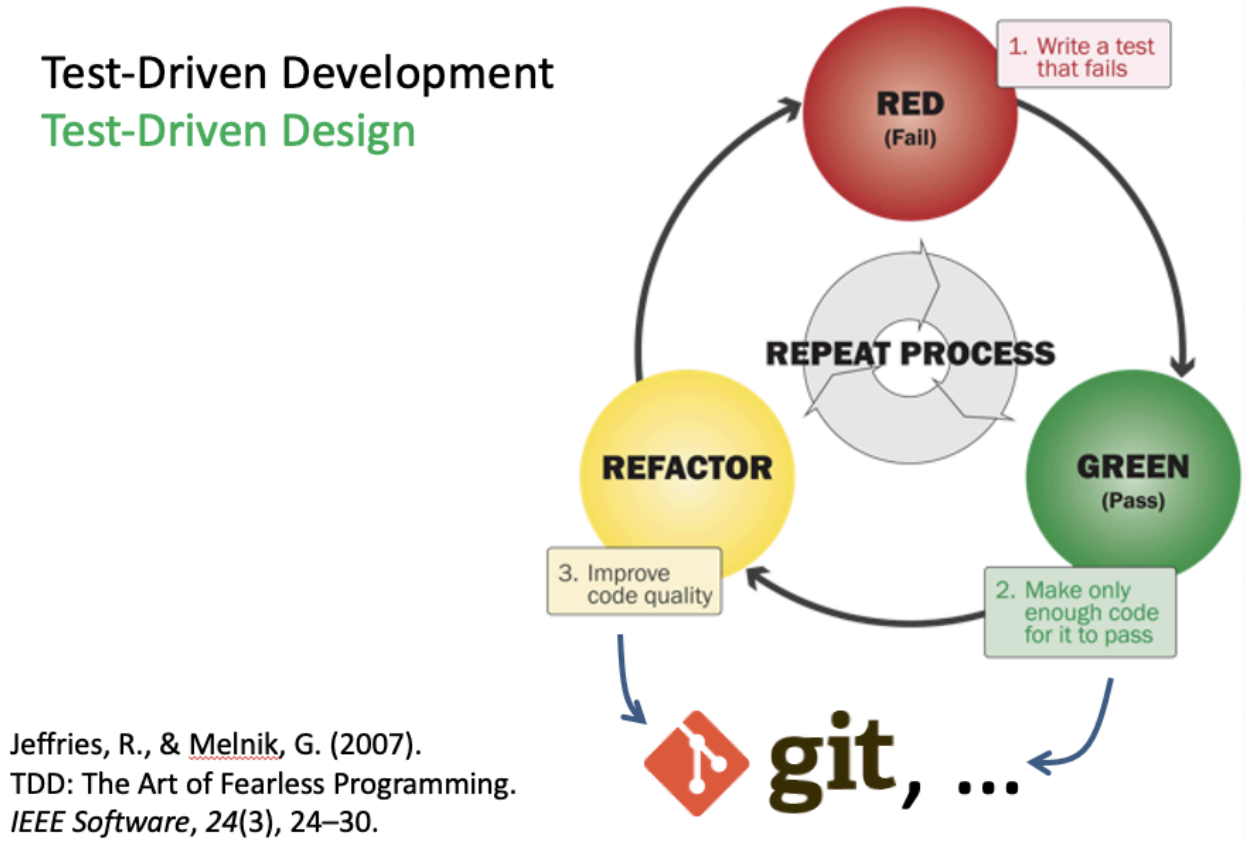
Martin Fowler, an author of much wisdom about software development, notes in his entry on UnitTest that the term “unit test” is very ill-defined. In general, “unit tests are low-level, focusing on a small part of the software system” but people disagree about what is “low-level” or “small”. In other words, “unit test” is a relative term. A unit tends to be smaller than a component, which tends to be smaller than a module, which tends to be smaller than a subsystem, which tends to be smaller than a system, and so on.

We use the term “**unit testing**” to refer to the process of software developers creating and using **automated** unit tests to check whether small units of code works as they expect. Because these tests are automated, they can be used again and again to help ensure that the code is working appropriately.

Red-Green-Refactor Pattern of Unit testing via Test-Driven Development (TDD)

TDD uses unit tests to help you produce cleaner code. See Clean code: a handbook of agile software craftsmanship, Robert Martin, 2009.

Test-Driven Development (TDD) [also known as Test-Driven Design] is a well-respected and commonly used method for creating unit tests. In TDD, the software developer creates code by following the **red-green-refactor pattern** (see TestDrivenDevelopment):



Jeffries, R., & Melnik, G. (2007).
TDD: The Art of Fearless Programming.
IEEE Software, *24*(3), 24–30.

TDD steps

1. Plan your work.
    a. Come up with an understanding of the **external behavior** you are trying to implement. Why are you changing the code? How do you expect people will use the new code you are about to produce? What constraints does the new code need to conform to? What are quality measures by which you can assess the new code?
    b. Don't think too much about the internal design of the code. The internal design will emerge from using the red-green-refactor cycle.

2. Prepare to do TDD.
    a. Create a new branch from your repository. Each feature should have its own branch. When the feature is done, the branch should be merged into main and then deleted.
    b. Make sure that all of the automated tests pass. You want to make sure that any issues that arise during your TDD are because of the changes you made, not because the code you downloaded was broken before you started.

    c. Do any other manual sanity checks you need to ensure that the code you have works as intended.
    d. If you find errors, fix them. If the errors are about the code, use the the red-green-refactor cycle below to fix the errors (write a test that fails because of the bug; fix the code; make sure that new test and all other tests pass). When the errors are fixed, merge your fixes back into the main repository so that your team members won't run into those same bugs.

3. Use the red-green-refactor cycle to implement the new functionality. The steps are:
    a. Write exactly one test that fails because the desired functionality is not written yet
        i. Figure out what is the next smallest unit of functionality you want to add to the system.
        ii. Write ONE (1) test for that next small bit of functionality.
        iii. Run that ONE (1) test and make sure it fails. This ensures that you are actually testing what you are trying to achieve. (I've heard several experienced developers talk about how sometime a new test passes … even though the code it was testing did not have worked that way! Ether the test was incorrect, or the code did not do what the developer thought it did.)
    b. Write just enough code for that new test to pass
        i. Write JUST ENOUGH production code for that ONE (1) test to pass. It's likely only a few lines of code. See the TINY steps that James Shore takes in his Let's Play TDD videos.
        ii. Run all of the other unit tests to make sure you haven't broken anything else in the system.
        iii. Commit the new improvements to your development branch. Committing the code and tests before the next step allows you to easily abandon any changes that go poorly.
    c. Improve code quality.
        i. If needed, refactor the new and/or old code to make it well structured. Separating adding changing functionality from refactoring reduces complexity, since you are only doing one type of work. Changing new functionality can be difficult. Refactoring can be difficult. Doing both at the same time is even more difficult. So, only do one at a time.
        ii. Run all of the other unit tests to make sure you haven't broken anything else in the system.
        iii. Commit the new improvements to your development branch.
    d. Start the next red-green-refactor cycle.

4. Get the new functionality into main.
    a. Once you have completed the functionality, merge your branch into main. Branches are intended to be short lived, typically at most a couple of days. Merging frequently back into main is critical to reduce merge conflicts.

b. Once the merge is done, delete the branch. The next piece of functionality will use a new branch.

Depending upon the context, it may be helpful to merge your branch into main multiple times during the above process.

Seeing TDD in practice

Reading the above steps cannot convey the actual way in which unit testing is done in TDD. To understand that, you need to see unit testing in practice. Do the following:

> Watch at least the first four episodes of James Shore's Let's Play TDD screencasts in which he creates a real project via TDD (scroll down to the bottom to start with episode #1).

TDD ⊆ Test-first development

Test Driven Development is NOT about writing multiple unit tests cases first, and then writing the code so that all the tests pass. That approach is useful in some contexts, but is not TDD. TDD is a particular way of doing test-first development.

TDD is about writing ONE test at a time, then writing just enough code so that ONE test works, making sure that the new code hasn't broken any of the other automated unit tests, and then repeating the process. TDD follows the red-green-refactor pattern.

Doing unit testing

Read the following sections of the Topic - Unit Testing document:

- What unit tests to write
- How to name unit tests
- How many tests to write? Using code coverage

Summary

Terminology

- Unit test - automated test, typically created during TDD
- Unit testing - the process of software developers creating and using automated unit tests to check that small units of code works as they expect

Qualities of unit tests

- Are automated
- Check that a small part of the software system works correctly
- Are small (typically less than 20 lines of code)

- Run quickly (milliseconds per test)
- Are typically written by developers
- Use test dummies to simulate any external dependencies (e.g., database)
- Are committed to the code base along with the code changes they are testing
- One (or very few) tests per unit test method

Additional information

For more information on unit testing, see:

- The Topic - Unit Testing
- James Shore's Let's Play: Test-Driven Development videos show him going through the process of using TDD to create an application from scratch. Even watching first several videos will show you how the tests help inform the design of the code, and the very small size of tests created.
- Chapter 9: Unit Tests of Robert C. Martin's book *Clean Code: A Handbook of Agile Software Craftsmanship*.