Teoria 4 - AGILE

Unit 4.3. Writing Code in TDD (Autoestudi)

Test Driven Development

TDD is an approach that drive the design of software.

First Step: Think.

Think of a small increment that will require fewer than five lines of code and think of a test
that will fail unless that behavior is present.

Second Step: Red bar.

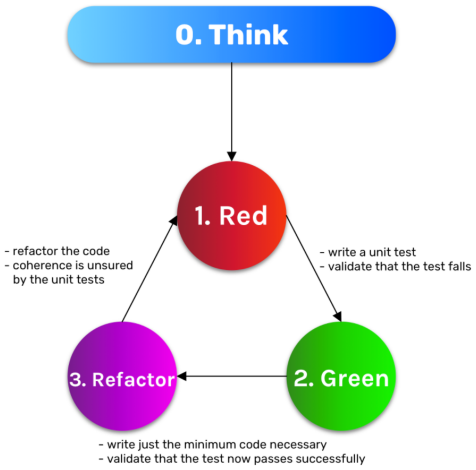
Write the test in terms of the class' behavior and its public interface, run it and watch
the new fail.

Third Step: Greenbar.

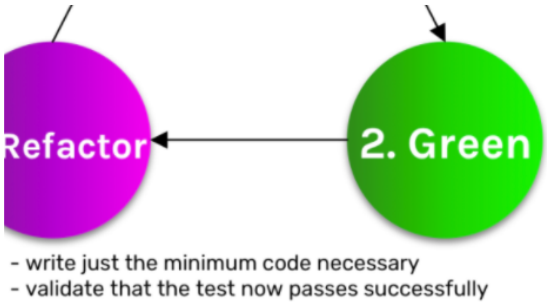
Write just the enough production code to get the test to pass. Run again and watch all tests pass.

Fourth Step: Refactor.

Review the code for improvements and apply small refactorings. Run again and watch all tests pass.



Third Step: Coding



Third Step: Greenbar.

> Write just the enough production code to get the test to pass. Run again and watch all tests pass.

In this step, we focus on designing and writing the code to get the test pass. But, what are the practices to keep in mind to design and write the code?

Two general practices have to take into account when we design and write the code to get a test to pass:

- **SimpleDesign**:

  allows the design and code to change to support any feature request, no matter how surprising.

- **Incremental Designand Architecture**:

  allows programmers to work on features in parallel with technical infrastructure.

Simple Design

When writing code, agile developers often stop to ask themselves:

> "`What is the simplest thing that could possibly work?`".

Rather anticipating changes and providing extensibility hooks and plug-in points, they create a simple design that anticipates as little as possible, as cleanly as possible.

This results in designs that are ready for any change, anticipated or not.

Simple design is described by Kent Beck as code that passes its tests and meets the four following guidelines:

- **Appropriate for the intended audience:** If people who need to work with it don't understand it, it isn't simple for them.
- **Factored:** Duplication of logic or structure makes code hard to understand and modify
- **Communicative:** Every idea that needs to be communicated is represented in the system.
- **Minimal:** The system should have the fewest elements possible. Fewer elements means less to test, document and communicate.

Design Principles for Simple Design

You Aren’t Gonna Need It (YAGNI).

Avoid speculative coding. Add something to your design only if it supported by the stories and features. If not, well... you aren’t gonna need it.

Once and Only Once.

Express every concept once. Every piece of knowledge must have a single and an unambiguous representation within a system.

- Example: *Rather than representing dollar amounts with a decimal data type, create a Dollar class.*

Self-Documenting Code.

Be sure to use names that clearly reflect the intent of your variables, methods, classes, and other entities. Pair programming will help you create simple code.

Isolate Third Party Components.

Isolate your third- party components behind an interface that you control. Consider using the Adapter pattern and create it incrementally.

Limit Published Interfaces.

Published interfaces reduce your ability to make changes. Internal publication assumes that, once defined, a public interface should never change. This is a bad idea. A better approach is to change your nonpublished interfaces whenever you need, updating callers accordingly.

Fail Fast.

Code your software such that, when there is a problem, the software fails as soon as and as visibly as possible, rather than trying to proceed in a possibly unstable state.

Incremental Design and Architecture

Incremental design.

Allows us to build technical infrastructure (such as domain models and persistence frameworks) incrementally, in small pieces.

Start by creating the simplest design that could possibly work (for methods, classes, ...), incrementally add to it as the needs of the software evolve, and continuously, improve the design.

Incremental Architecture:

> While your software grows, be conservative in introducing new architectural patterns: introduce just what you need at the moment. Before introducing a new pattern, ask yourself if it is really necessary.
>
> There are standard conventions to follow. For example, in a three-layer architecture, every business logic class will probably be part of a “business logic” namespace and probably interfaces with its persistence layer counterpart in a standard way.