Test Driven Development (TDD)

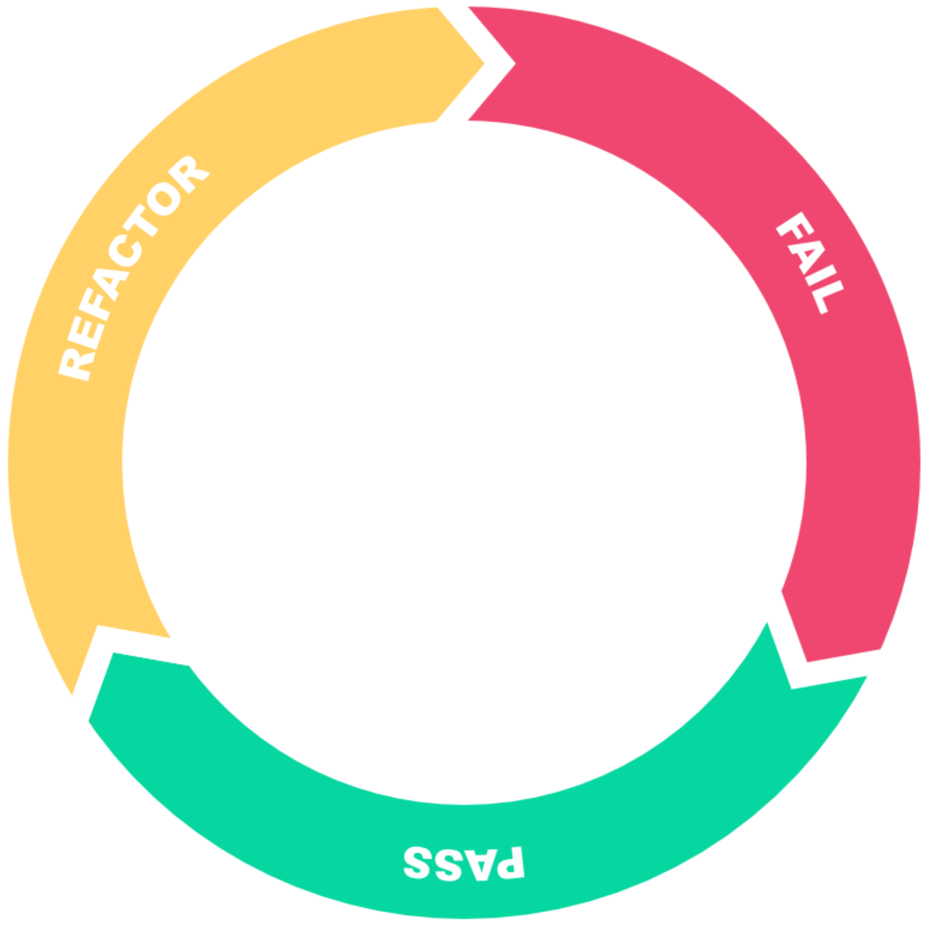
An Introduction by Kevin Denver

June 2022

Red-Green-Refactor: The Building Blocks of TDD

Test-driven development follows a three-phase process:

- **Red**. We write a failing test (including possible compilation failures). We run the test suite to verify the failing tests.
- **Green**. We write just enough production code to make the test green. We run the test suite to verify this.
- **Refactor**. We remove any code smells. These may be due to duplication, hardcoded values, or improper use of language idioms. If we break any tests during refactoring, we prioritize getting them back to green before exiting this phase.

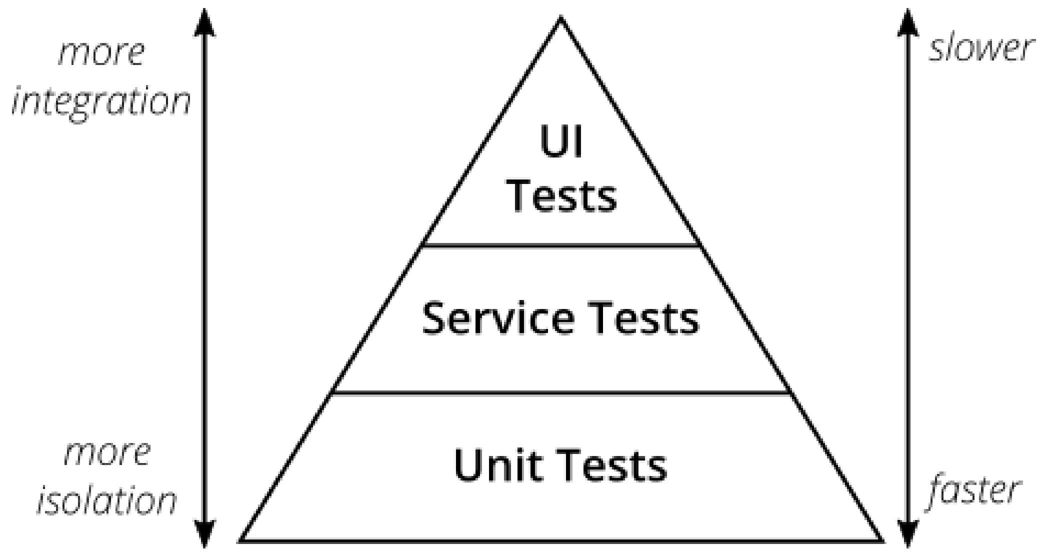


Benefits of TDD

- Writing the tests first requires you to really consider what do you want from the code
- Fast feedback
- Creates a detailed specification
- Tells you whether your last change (or refactoring) has broken previously working code
- Allows the design to evolve and adapt to your changing understanding of the problem
- Forces radical simplification of the code, you will only write codes in response to the requirements of the tests
- Shortens the development Time to Market
- Reduces the time it takes to refactor the code or fix bugs
- Cuts development costs
- Improves quality
- Test Driven Development gives programmers the confidence to change the larger architecture of an application when adding new functionality. Without the flexibility of TDD, developers frequently add new functionality by virtually bolting it to the existing application without true integration – clearly, this can cause problems down the road

Pitfalls of TDD

- Tests need to be treated with the same respect and care as production code (make them readable, refactor them)
- Excessive use of mocking can hinder your ability to refactor
    - See Mockists Are Dead. Long Live Classicists
- Testing the wrong thing or at the wrong level/abstraction



Mocking and Stubbing

- Mocks and Stubs are two different kinds of Test Doubles.
- You can use test doubles to replace objects you'd use in production with an implementation that helps you with testing.
- Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.
- Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in memory database is a good example).
- Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- Mocks objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

https://martinfowler.com/articles/mocksArentStubs.html
https://martinfowler.com/bliki/TestDouble.html

References

- Learning Test-Driven Development
- Mocks Aren't Stubs
- Scientific Research Into Pair Programming
- The Art of Agile Development, 2nd Edition
- Detroit and London Schools of Test-Driven Development
- Test Driven Development: By Example
- Mockists Are Dead. Long Live Classicists
- The Practical Test Pyramid