TDDBUDDY.COM

TDD Gears

Into

TDD is a complex and challenging discipline filled with many practices and principles. It can be hard to understand the reasoning behind some practices such as the rule of 3 or triangulation while solving seemingly simple katas.

To explain the alignment between all the elements the idea of changing gears in a car works well. Low gear is to get going and build context, medium gear is to enhance design and apply advanced patterns, high gear is used to build functionality by following existing architectural patterns and practices of the system. Reverse is used to get back to green after a back refactor or to help take a new approach to find the next test.

Katas should be done in low and medium gear to learn and embed the practices and principles.
Production can be done in any gear you choose.

TDD Practices and Principles

A listing of all practices and principles involved with TDD.

Core TDD Practices

- Rule of 3
  - Allow duplication to appear 3 times before you abstract it.
- Triangulation
  - Implement for 2 specific test before moving towards a generic solution for the 3rd
- 3 Laws of TDD
- 3 Stages of Naming
  - Move from meaningless to specific to meaningful names
- Equivalence Partitions and Boundaries
- Test Factory Methods
  - Abstract common test setup per file
- Mocking
  - Received Asserting -> Did a method get called

Advanced TDD Practices

- Test Factory Methods
  - Avoid default arguments in method
  - Avoid optional parameters in method
- Test Data Builders
  - Abstract common test setup per solution
  - Uses fluent syntax
- Mocking
  - Content Asserting -> Save the contents of the method call and assert it is correct

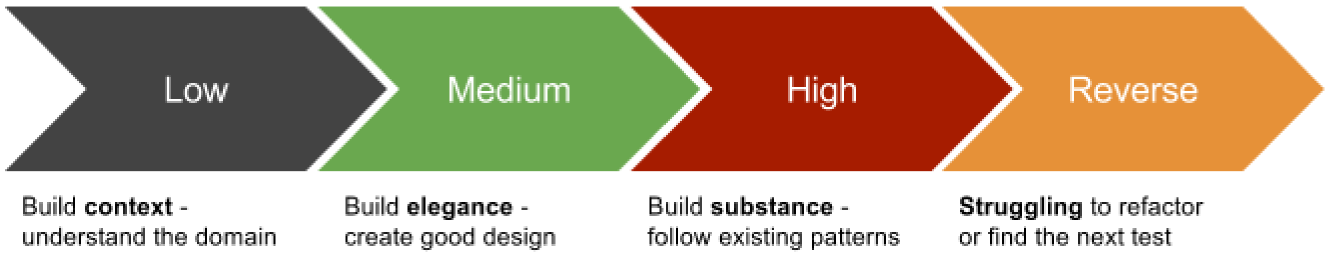
TDD Principles

- FIRST Principles
- SOLID Principles

TDDBUDDY.COM

The Gears

TDD contains a Red-Green-Refactor cycle at its core with a collection of practices and principles. TDD gears is a model to explain how it all fits together. There are three forward gears and one reverse gear in TDD because sometime you just need to back up and try a different approach.


Low
Medium
High
Reverse
Build context - understand the domain
Build elegance - create good design
Build substance - follow existing patterns
Struggling to refactor or find the next test

- **Low**
    - Unsure of Domain or algorithm being implemented
        - Evolve slowly to **build context -** understand the problem
    - Strict following all core practices
    - Little to no advanced practices
    - Low risk of reversing
- **Medium**
    - Some familiarity with domain or algorithm being implemented
        - Evolved to **build elegance** - create good design
    - Short circuit some core practices
        - Relaxed need to use Rule of 3
            - You may refactor after 2 instances
        - Relaxed need to use Triangulation
            - You may write the generic implementation if it is less code or clearly visible.
    - Applies mainly advanced practices
    - Moderate risk of reversing
- **High**
    - Familiarity with domain or algorithm being implemented
        - Evolve to **build structure** - follow existing patterns
    - Ignore some core practices
        - No need to follow Rule of 3
            - You may make abstractions straight away, be mindful of the increased risk of reversing.
        - No need to use Triangulation
            - You may write simple generic implementation straight away
    - Mixes advanced and core practices
    - Moderate / High risk of reversing
- **Reverse**
    - Like getting stuck in the mud - reverse and try a different angle.
    - When to use:
        - Struggle to complete a refactoring
        - Struggling to find next test
    - How to use:
        - Revert to a green test run
            - Revert both production and test classes - commenting out works best.
        - Restart in a **Low gear**
            - Find a new point of attack
            - Use your test to take small steps to build a “virtual stack trace” validating your understanding of the code.