Design Benefits of TDD (Parts I, II, & III)

- A well designed code is easier to test. It will also require fewer test. It will fail less so can be easily tested as well.
- Writing test cases forces the code to become more modular.
- If some code has a lot of dependencies, it becomes hard to write tests for it, because tests become brittle because of these dependencies.
- Code written through tests ends up become more cohesive/narrow/small/focused, has a single responsibility and also has less coupling.
- **High cohesion & low coupling is one example of good designed code.**

How does test influence design?

- “Unit Testing is an act of design more than an act of verification.” - Ken someone
- A good tested code is a good designed code, in general.
  - But that isn’t a guarantee. Just like we can write bad code, we can write bad tests.
- A good written test can influence a good design.

Types of Unit Test

1. Positive
   a. The code is doing what it’s supposed to do.
2. Negative
   a. The code is doing something that isn’t supposed to happen. Eg. money being deducted for a closed mortgage account.
3. Exception
   a. To make sure that the code is throwing the right kind of exceptions.

Performance Tests

- Test the performance of the code (time take to compute, for instance).
- JUnitPerf is a JUnit extension → gives time testing.

What's a Unit Test?

- A unit test is a test on a unit of code.
- What's a unit of code?
  - A unit of code is the smallest piece of code that does useful work.
    - Getters & Setters are the smallest functions — but **do not** write tests for getters & setters. They should come into the picture when we are writing other tests and other functions. So, no point in testing getters & setters.
  - A unit is often a method or a function.
- When writing unit tests, try to focus on behaviour rather than state.
  - If you start testing the state, you end up creating objects that are heavy on state.
  - If you start testing the behaviour, you move quickly toward the purpose of the application rather than polluting the objects with states.

How TDD changes that?

- First, put yourself into the shoes of the person who's going to use your code even before your write your code.
  - "What kind of method would they call?"
  - "How would they initialise this object?"
  - "What kind of parameters would they pass? What would the type of the parameters be?"
  - "Does this look lightweight or does it look verbose and heavy? Is this a burden on the user?"

Test first coding

- Many people write the code before writing the tests.
- The longer we have gone without writing tests, the harder it becomes to write effective and meaningful tests.
- How about we first write a test, and then come and write enough (bare minimum) code so that that test passes.

Benefits

- Reverts
  - It reverts the way we think about the code.
  - Rather than rushing eagerly to create the implementation, we grow the skin, the interface, and then when we have something nice/usable/practical, then we put minimum details in it.
    - It helps to keep things minimum and practical from the point of view.
- Robustness
  - The code becomes robust (Regression benefit).
  - Anytime we make a change to the code, these tests wake up and verify, and if anything were to deviate from expectations, they'd immediately tell us.
- Verification
  - It's good way to verify that everything that worked before works now.
- Confidence
  - It gives us, the programmers, confidence that our code actually works.
- Decoupling
  - It is hard to test code that has a lot of intricate coupling/dependencies, so we do dependency inversions and we decouple the code.
  - At times, we eliminate dependencies, because it becomes hard to test the code with dependencies.
- Form of Documentation

Influence of TDD

- Clean code that works
- Predictability
  - There are no mountains of error.
  - We make a change, we immediately get a feedback.

Canary Test

- The stupidest test you could ever write.
- `assertTrue(True)`
- Just to check if the everything's set up correctly.
- Write this only when you start a new test project.
- "Canary in a coal mine." — Canary bird is very sensitive to certain types of fumes.

Principles, Practices, and Tasks

- On a little paper.

▼ **Practices**

- Jot down test that come to your mind.
- Pick the test that is fairly easy and quick to write.
- Pick a test that is valuable.
- Write a failing test first, then write minimum code to make that pass.
- Jot down positive, negative, and exception test
- Listen to the whispers of your tests
- Strive for simplicity; reduce and remove accidental complexity
- Every test should have an assert
- Do not write more than one independent assert per test
- Keep tests isolated from each other

- Tests have to be FAIR
  - F → Fast
  - A → Automated
  - I → Isolated/Independent
  - R → Repeatable
- When tests pass, take a look at the code to see if it can be simpler, can be refactored
- Design decisions have to be documented through test cases

▼ **Tasks (tic tac toe player) — follow along in the video**

- ‘game not won by anyone when the game starts’
- pick the first player
- pick the second player
- first player places peg at some location
- second player places peg at some location
- place peg at an occupied position
- place peg out of row range
- place peg out of column range
- set first peg should not be other than X and O
- win by row match
- declare winner for row match
- win by column match
- declare winner for column match
- win by left diagonal match
- declare winner for left diagonal match
- win by right diagonal match
- declare winner for right diagonal match

- place peg after a game is won

▼ **Principles**

- YAGNI → You Aren’t Gonna Need It (yet)
- Postpone decisions until the last responsible moment
- Keep it DRY → Don’t Repeat Yourself
- SRP → Single Responsibility Principle
- SLAP → Single Level of Abstraction Principle

Benefits Revisited

- TDD is an act of design than an act of verification.
- We drive this based on feedback.
- It gives us sustainable development.
- The tests serve as a documentation.
- Tests serve as a safety net when it comes to refactoring.

Tenets of TDD

- Write/change code only if a test fails. If all tests are passing, only refactor — don’t add new code.
  - If you want to add more code, then first write the test which will force you to write the code.
- Minimum code that works.
- Seek pragmatic and simple design.
- Eliminate duplication.

Red/Green/Refactor

- The test is failing. You make the test pass. Then you refactor the test to make it work.

Untested code is unfinished code

- Don’t declare it completed unless you’ve fully tested it.

Test on each platforms

Don’t forget functional tests

- At higher levels, you will need functional tests.

Who should write unit tests?

- Unit tests should be written by programmers as they are sitting down to write code, not by some other programmer at some other time.

When should you write unit tests?

- When you are writing the code.

Unit test and legacy code

- “Put an end to legacy code by not writing more legacy code.”
- Michael Feathers → “Legacy code is a code with no tests.”
- Write tests for any new code you add to it. Don’t write tests for legacy code.