TEST-DRIVEN DEVELOPMENT

Test driven development is an iterative development process. In TDD, developers write a test before they write just enough production code to fulfill that test and the subsequent refactoring. Developers use the specifications and first write test describing how the code should behave. It is a rapid cycle of testing, coding, and refactoring.

The key ingredient for being effective with test-driven development is understanding what it truly is. I find that there are a lot of misconceptions around how to do TDD properly. TDD is one of the practices that if you do it wrong, you often pay a hefty price.

TDD means letting your tests drive your development (and your design). You can do that with unit tests, functional tests, and acceptance tests. It leads you to create very different kinds of tests that tend to be more resilient to change in the future because you're verifying behaviors rather than testing pieces of code. Let's see how it's done by following three stages of test driven development.

After understanding what is TDD in software development, let's take a closer look at its benefits.

What are the advantages of TDD?

Code Quality

TDD framework encourages the development of simple, clean, and extensible code. The discipline of following TDD would naturally develop habits that lead to **better code** as part of developers' everyday practice. Developers become more focused on the system requirements by firstly asking themselves why a feature is needed before proceeding with the implementation. By this process, the developer can identify badly defined requirements as producing a unit test for them becomes taxing.

TDD helps developers towards simple designs; keeps things typically OO [Object Oriented] structured; pushes developers towards separated components.

Application Quality

Another advantage of writing breakpoint tests before production code is that developers spent more time designing the boundary cases needing to be covered by these tests, compared to the traditional approach. It results in more thorough testing and fewer bugs/defects at the end of the development cycle.

Increases Developers' Productivity

TDD improves the speed as developers spend less time in debugging. It may increase the time spent on developing tests and production code during early phases. But as the development progresses, adding and testing new functionality will be quicker and requires less rework. It's a lot cheaper in terms of resources to fix the issue immediately rather than months down the track when they may be discovered.

It is frustrating to write 15 tests but it's more frustrating to not be able to change something or not to know your changes are safe.

Higher Test Coverage

Higher test density and test coverage are by default advantages of TDD. In the traditional approach, there is a higher likelihood that testing would be left out or restricted to critical functionality, particularly if time was short. On the other hand, TDD institutes the discipline of all functionality being associated with a set of automated unit tests. This results in more tests and higher test coverage of the code.

Living Documentation

Tests can serve as documentation to a developer. If you're unsure of how a class or library works, go and have a read through the tests. With TDD, tests usually get written for different scenarios, one of which is probably how you want to use the class. So you can see the expected inputs a method requires and what you can expect as an outcome, all based on the assertions made in the test.

This can help increase developer understanding of parts of the system and therefore helps to support collective code ownership. As a result, changes to code can be made by any developer rather than the only developer who understands the code.

How to implement TDD: What are the 5 steps of Test Driven Development?

Test-Driven Development (TDD) process typically follows five main steps, often called the "TDD cycle." Let's understand what are the 5 steps of TDD:

Step 1: Write a Test
In this first step, you write a test for a specific functionality you want to implement. This test will initially fail because the functionality is not yet available.

Step 2: Run the Test
Once you have written the test, you run it to ensure it fails as expected. This confirms that the test works correctly and demonstrates that the feature is not yet implemented.

Step 3: Write the Code
In this step, you implement the minimum code necessary to pass the test. The goal is not to write the complete feature but to make the test pass successfully.

Step 4: Run All Tests
After writing the code, you run all the tests, not just the one you added in the first step. This ensures that the new code doesn't break any existing functionality.

Step 5: Refactor
In the last step, you refactor the code to improve its design and maintainability while keeping all tests passing. Refactoring is essential to keeping the codebase clean and easy to work with.

The cycle then repeats for the next functionality you want to add. By following this process, TDD helps ensure your code is thoroughly tested, maintainable, and less prone to bugs.

Test driven development (TDD) is one of the common practices of Agile core development. It is acquired from the Agile manifesto principles and Extreme programming. Let's see how TDD fits well in the agile development process.
What's agile development? In the simplest form, It is Feedback Driven Development.

And, Feedback is Critical.
The requirements you start with may change during the development cycle. If your objective is to build what your customers wanted, you will fail—you need to build what your customer still want, what's relevant. For you, below two types of feedback are equally important:

- You want rapid feedback
- You want to avoid Whack-a-mole software

Agile development is not about running fast... It is about running fast in the right direction at a sustainable pace.

And, TDD is a way to get rapid feedback.

"Test First," in which unit tests are written before the code, can mitigate bottlenecks that impede quality and delivery. The test helps to define what the code is meant to do, providing guidance for the developer in terms of user functions. This concept is a natural fit with Agile in two ways:

1. By developing the tests from the requirements, rather than the code, communication increases. The creator of the requirements, the developer, and the tester must collaborate

on the tests and the subsequent code, thereby increasing everyone’s understanding of the work at hand./li>

2. By having the test or test suite written first, there is no need to wait for the testing to be done. The code can be written and tested immediately, especially when automated testing is included in the process (considered a best practice). If the code fails, it can be pushed back onto the backlog, and if it succeeds, the next item can be started.

As you evolve the system based on feedback, bug fixes, and additional features, it tells you that the maintainable code worked and continues to work as expected.

After understanding what is TDD in agile, let’s take a look at some popular TDD tools.

Popular TDD Tools

Following are some common and most used unit testing frameworks/tools that support TDD approach.

- csUnit : An open source unit test tool that offers a TDD unit test framework for .Net projects
- DocTest: A very simple, easy to learn unit testing framework for Python
- JUnit: A Java TDD unit test framework.
- NUnit: This one again is used for .Net projects

- PHPUnit: This one is used for PHP projects
- PyUnit: A standard unit testing framework for Python
- TestNG: A testing framework for Java, which overrides the limitations of JUnit.
- RSpec: A framework for Ruby projects.

Limitations of TDD

While TDD has several benefits, it also has some limitations.

1. TDD slows down the development process

TDD initially requires writing tests before implementing the code, which can slow progress, especially in fast-paced startups where time to market is crucial. If an urgent product launch is needed, TDD may not be ideal as it involves creating tests before the code, causing delays.

2. Test cases may require a lot of upgrades

One major drawback of TDD is the need to modify tests when product requirements change. This can be costly and impact development schedules. Moreover, TDD leads to excessive unit tests, making code maintenance more time-consuming due to frequent updates and increasing developer overhead.

3. Learning TDD is not an easy

TDD is great for fast-paced projects but may not be ideal for complex projects with resource constraints. If your team has never used TDD, they will need time to understand and adapt the process.

4. UI testing with TDD is difficult

Using Test-Driven Development (TDD) for user interfaces is challenging due to their complexity and high interactivity. Additionally, the dynamic characteristics of modern applications contribute to the difficulty. Frequent changes to UI elements can lead to brittle UI tests, causing failures even when the core functionality of the application remains unaffected.

5. Applying TDD to legacy code is challenging

TDD is most efficient when implemented right from the start of a project. However, applying TDD to legacy code, which was not originally developed with this approach, can be more problematic because the code might not have been designed with testability in mind.

Test Driven Development (TDD) Examples

Here are some examples of Test Driven Development:

1. Calculator function

For a calculator application, you might start by writing a test to ensure that adding two numbers together works correctly. You'll define test cases for different scenarios, such as positive numbers, negative numbers, and decimals. Once the test cases are defined, you'll write the actual calculator code to make the tests pass.

2. Shopping cart functionality

For an eCommerce website, you’d follow TDD for the shopping cart functionality. Starting with tests, you would ensure that items can be added to the cart, the correct prices are calculated, and the cart updates as users add or remove items. By writing tests first, you’ll develop code that satisfies the requirements and behaves as expected.

3. User authentication system

When developing a user registration and login system, you’d first write tests to verify that new users can register successfully and that existing users can log in with the correct credentials. You’ll consider scenarios like invalid email addresses, password strength, and handling incorrect login attempts. After writing the tests, you’ll implement the registration and login functionality to meet the test requirements.

TDD vs. Traditional Testing

Test-Driven Development (TDD) and Traditional Testing are two different approaches to software testing, each with its own set of principles and benefits. Let’s explore the differences between them:

- **Scope of testing:** TDD centers on testing small code units individually, whereas conventional testing encompasses examining the system as a whole, including integration, functional, and acceptance testing.
- **Feedback loop:** TDD offers a quicker feedback loop for developers to know if their code passes the newly written test. In contrast, traditional testing may result in a longer feedback loop,

especially if tests are done after developing significant code portions or the entire application.

- **Documentation:** TDD documentation primarily focuses on test cases and their outcomes, whereas traditional testing documentation may contain more specific information about the testing methodology, test environment, and system under test.
- **Debugging:** TDD detects errors at the earliest stages of development, simplifying the process of debugging and rectification. On the flip side, conventional testing methods demand tremendous effort to troubleshoot errors in the development process.