Test Driven Development

27th April 2025

Mr. Suresh Fernando

Test Driven Development - Overview

- Introduction to Test Driven Development
- TDD Vs Traditional Testing
- Three Rules in TDD
- TDD Lifecycle
- SOLID Principles for TDD
- TDD Impact on Developers
- Benefits and Challenges
- TDD Myths

What is Test-***Driven*** Development?

- A software development technique which reiterates the importance of testing
- Promotes writing software requirements as tests as the initial step in developing a code
- Initially writes tests and then moves forward with the least amount of code needed to get through the tests.

Brief History of TDD

- Late 1990s: TDD as a formalized practice was popularized by **Kent Beck** during the development of Extreme Programming (XP).
- Kent Beck later published the book **"Test-Driven Development: By Example" (2002)** which became a foundational text.
- The origins of TDD are closely tied to the **Agile Movement,** where iterative, feedback-driven development became the norm.

TDD in Agile Context

- TDD fits Agile because it supports:
  - ***Short feedback loops*** (you know if your code works instantly).
  - ***Continuous integration*** (tests support rapid changes).
  - ***Customer collaboration*** (cleaner, testable code is easier to evolve).
- Agile methodologies like **Scrum** and **Extreme Programming** (XP) use TDD as a key technical practice to ensure quality and adaptability.

Test-Driven vs Traditional Testing

Aspect	TDD	Traditional Testing
Approach	Tests are written before code implementation	Testing is performed after code implementation
Test Focus	Emphasizes small, incremental tests for specific features	Focuses on comprehensive testing of the entire application
Test Creation	Tests are written by developers based on requirements	Tests are typically created by dedicated QA testers
Code Coverage	Encourages high code coverage with a focus on critical paths	May have varying levels of code coverage depending on test cases
Feedback Loop	Provides immediate feedback on code changes	Feedback may be delayed until the testing phase
Bug Detection	Helps identify bugs early in the development process	Bugs may be detected later in the development cycle

Steps in Test-Driven Development

1. Read and understand the feature request from clients.

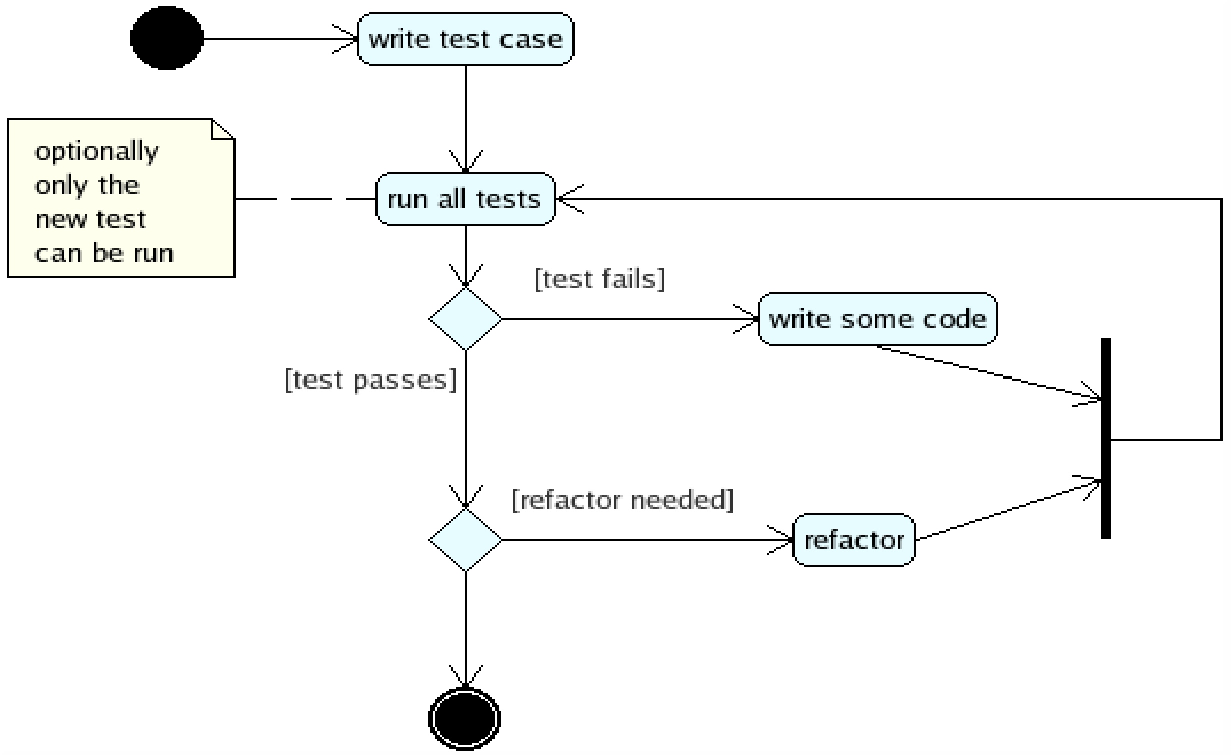
2. Translate them by creating a test. This test will fail when you run it because no code has been implemented so far.

3. Now, write and implement the code to pass the test. Then, run the test and now it should pass. If it doesn't pass yet, repeat the steps until it passes.

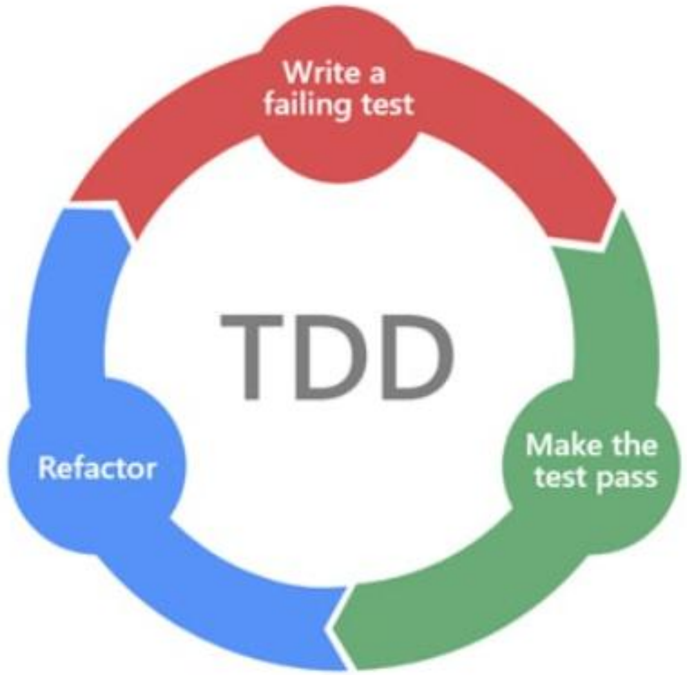
4. Once it passes, clean your code up. You can do it by refactoring.

5. Repeat above steps again for another requirement

Steps in Test-Driven Development



TDD Cycle (Red-Green-Refactor)



- Here, the **red** phase denotes that the code is not working.
- The **green** phase is an indication that everything is working. However, they do not work optimally.
- The **blue** phase denotes that the code is being refactored.

TDD Cycle – Red

- Goal:

Write a **test that fails** because the functionality you want **doesn't exist yet.**

- Why?

- Forces you to **think about the problem** and define **expected behavior** *before* implementation.
- Ensures your test is **valid** – if it doesn't fail when the feature is missing, something's wrong with the test.
- Prevents writing unnecessary code – you only write code to **make the test pass.**

TDD Cycle – Green

- Goal:

Write just enough code to make the failing test from the Red Phase pass – nothing more.

"Get it working first. Make it beautiful later."

- Why?

- Ensures your test was valid.
- Gives you confidence that the system behaves as expected.
- Helps you build functionality incrementally.
- Keeps you focused – no extra code means less chance of bugs.

TDD Cycle – Refactor

- Goal:

Improve the code’s structure, readability, and maintainability - without changing its behavior.
"Now that it works, let's make it nice."

- Why?
  - Keeps code clean and maintainable.
  - Prevents ***technical debt*** from accumulating.
- What?
  - Remove duplication.
  - Rename variables/methods for clarity.
  - Simplify logic or extract methods.

Activity

Discuss in Pairs – "What are the possible benefits and challenges you could expect from TDD?"

Three Rules of TDD

- **Rule #1 – No code without a test**
  - No new features or logic unless there's already a test failing because that logic is missing.

- **Rule #2 – No more test than needed**
  - Don't write multiple tests or long tests upfront. Just write *enough* to see the test fail.

- **Rule #3 – No more code than needed**
  - Only write code that's needed to make the current test pass. No extra logic, no predictions of future features.

#1- No code without a test

Code

```
def add(a, b):
    return a + b
```

You can't write this ***add*** function until you've written a test like mentioned below

Test

```
def test_addition():
    assert add(2, 3) == 5  # ❌ This test will fail initially.
```

Once the test fails, you write the minimal code to make it pass.

Test

```
def test_password_too_short():
    validator = PasswordValidator()
    assert not validator.is_valid("abc")  # Password must be at least 6 characters
```

Code

```
class PasswordValidator:
    def is_valid(self, password):
        return len(password) >= 6
```

#2- No more test than needed

Suppose you want to build a function that sums numbers from a comma-separate string input.

First Test : To check if an empty string will return zero

```
def test_add_empty_string_returns_zero():
    assert add("") == 0  # Fails because 'add' isn’t implemented yet.
```

Code: Then write ***minimal*** production code:

```
def add(numbers):
    return 0  # ✅ Now the test passes.
```

Here we are not jumping ahead and writing tests for comma-separated numbers yet but just focused on the first test

#3- No more code than needed

Next Test : To check if a string with single number will return the same number

```
def test_add_single_number():
    assert add("5") == 5
```

Code: Update code ***just enough*** to pass

```
def add(numbers):
    if numbers == "":
        return 0
    return int(numbers)
```

Here we don’t yet add codes to support for two numbers or handle delimiters. That will come with the future tests accordingly.

Benefits of TDD

Improve code quality
Code is written to pass specific tests

Immediate Feedback
Finds out immediately if something is broken.

Better Design & Maintainability
Encourages writing loosely coupled, highly cohesive code

Documentation Through Tests
Tests act as **live documentation** of how the system should behave

Confidence to Refactor
You can improve or change code structure without fear, because tests will catch regressions.

Facilitates Debugging
When something breaks, you already have unit tests that isolate logic

Challenges of TDD

Initial Learning Curve
Beginners may struggle with writing tests first or designing testable code

Slowdown Development at First
Writing tests before writing any code will consume time

Maintenance Overhead
A large suite of outdated or redundant tests becomes a burden.

UI, Legacy, or Non-Deterministic Code
TDD works best with deterministic, modular code

Over-Testing / Rigid Code
Writing too many detailed tests can make refactoring painful

Mindset Change
TDD requires a **fundamental shift** in how developers think about writing code.

SOLID Principles for TDD

- SOLID principles are key to object-oriented design and closely related to Test-Driven Development (TDD).
- SOLID is not exclusive to TDD, but practicing TDD tends to enforce or encourage adherence to these principles.

Letter	Principle	Purpose
S	Single Responsibility Principle (SRP)	One reason to change
O	Open/Closed Principle (OCP)	Open to extend, closed to modify
L	Liskov Substitution Principle (LSP)	Subclasses must be replaceable
I	Interface Segregation Principle (ISP)	Prefer many small interfaces
D	Dependency Inversion Principle (DIP)	Depend on abstractions, not concrete classes

S - Single Responsibility Principle (SRP)

Principle: A class should have **one and only one reason to change.**

TDD Implication: As you write unit tests first, you tend to **split responsibilities** to make the code more testable.

- Each module, class, or function should **do one thing and do it well.**
- If a class is responsible for more than one thing, those responsibilities become **coupled.**
- A change to one responsibility may impact the others.

S - Single Responsibility Principle (SRP)

Example: Suppose you're writing a *ReportService*.

Before (Violates SRP):

```
class ReportService:
    def generate_report(self):
        # gathers data, formats it, sends email
```

Too many roles in one place

TDD forces to separate these roles

After TDD forces separation:

```
class DataFetcher:
    def get_data(self): pass

class ReportFormatter:
    def format(self, data): pass

class EmailSender:
    def send(self, report): pass
```

O – Open/Closed Principle

Principle: Software entities should be **open for extension but closed for modification.**

TDD Implication: TDD encourages you to write code that can **evolve** via extension - because changing tested code is risky.

- You should be able to **add new functionality** to a class or module **without changing its existing code.**
- This protects existing behavior from bugs and makes the system easier to maintain and extend.

O – Open/Closed Principle

Example: You write tests for a tax calculator. Later you want to support a new tax strategy.

```
class TaxStrategy:
    def calculate(self, amount): pass

class FixedTax(TaxStrategy):
    def calculate(self, amount):
        return amount * 0.1

class ProgressiveTax(TaxStrategy):
    def calculate(self, amount):
       if amount > 1000
           return amount * 0.2
       else
          return amount * 0.1
```

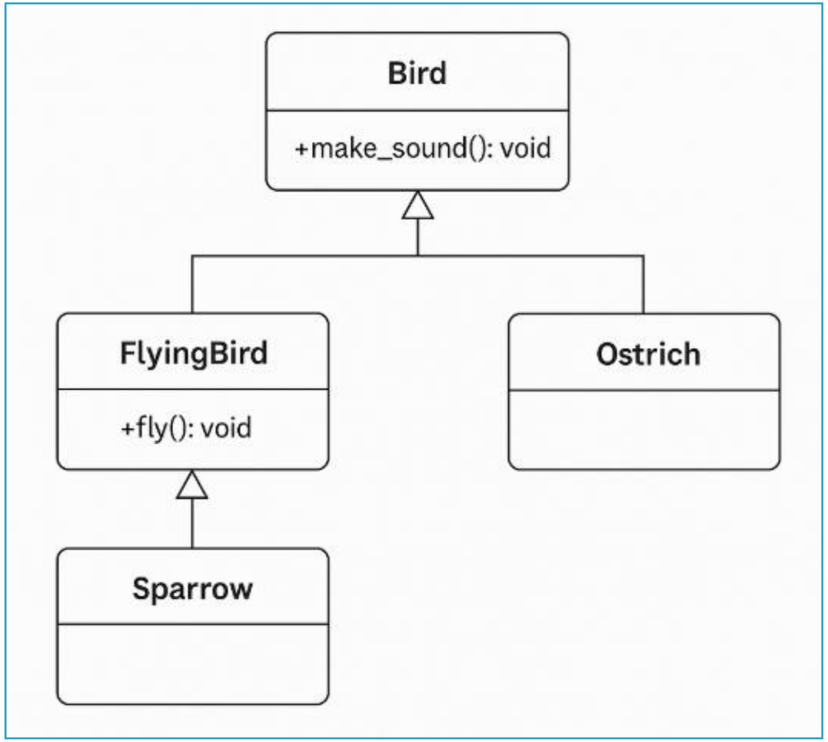
You don't modify existing logic — you **extend** with a new class so that you can run new tests for it.

L – Liskov Substitution Principle

Principle: Subclasses should be **substitutable** for their base classes without altering behavior.

TDD Implication: TDD helps spot LSP violations early - when a test for a base class fails with a subclass.

L – Liskov Substitution Principle



L – Liskov Substitution Principle

Example: Let’s say you’re testing *Bird.fly()* but your Ostrich subclass throws an error.

```
class Bird:
    def fly(self):
        print("Flying")

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostriches can't fly")
```

```
class Bird:
    def make_sound(self):
        print("Chirp")

class FlyingBird(Bird):
    def fly(self):
        print("Flying")

class Sparrow(FlyingBird):
    pass

class Ostrich(Bird):
    pass
```

I – Interface Segregation Principle

Principle: Clients shouldn’t be forced to depend on interfaces they don’t use.

TDD Implication: TDD naturally leads to **smaller, focused interfaces,** because large ones are hard to test.

I – Interface Segregation Principle

Example: You create a test for a SimplePrinter, but it doesn't scan. You're forced to ignore scan_document()

Before ISP Principle:

```
class Printer:
    def print_document(self): pass
    def scan_document(self): pass
```

Better design with ISP

```
class Printable:
    def print_document(self): pass

class Scannable:
    def scan_document(self): pass

class SimplePrinter(Printable): pass
class MultiFunctionPrinter(Printable, Scannable): pass
```

D – Dependency Inversion Principle

Principle: Depend on **abstractions**, not on concrete implementations.

TDD Implication: You often start writing tests by **mocking dependencies**, which encourages depending on interfaces, not classes..

D – Dependency Inversion Principle

Example: Imagine you're writing a **UserSignupService.** After signing up, the user should receive a welcome message.

In a tightly coupled system (no DIP)

** UserSignupService directly uses a concrete EmailService.*
** Your tests now:*

- *Send real emails (bad).*
- *Rely on infrastructure (fragile, slow).*
- *Are hard to isolate and mock.*

Better design with DIP

UserSignupService depends on a MessageSender interface.
Now:

- *Tests are isolated, fast, and reliable.*
- *You can verify that "a welcome message was sent" without sending real emails.*
- *Your tests stay green even if you later switch to SMS or in-app notifications.*

Using Mockups in TDD

- **Goal**: Focus your test on only the **behavior of the system under test (SUT)**, without involving the real implementations of its dependencies.
- **Why Use Mocks in TDD?**
  - To test only one unit at a time.
  - To avoid slow or unreliable dependencies (e.g., network, databases).
  - To simulate specific behaviors like errors, timeouts, or success cases.
  - To verify interaction between the SUT and its dependencies.

Using Mockups in TDD

- **System Under Test:: *OrderService* –** Responsible for placing an order.
- **Dependencies: *PaymentGateway*** and ***EmailNotifier***

Without Mocks:

- When you test ***OrderService***, it actually tries to charge a credit card and send an email.
- This means:
  - You need real accounts.
  - Tests are slow and can fail due to network/email issues.

With Mocks:

- You create mock objects for ***PaymentGateway*** and ***EmailNotifier***.
- You inject these mocks into ***OrderService***.
- In the test:
  - You simulate a successful payment.
  - You verify that the email notifier was called correctly.
  - Now you're truly testing just the behavior of ***OrderService***.

TDD Impact on Developer’s Life

Reversing the Usual Workflow

- Traditional approach: Write code → Then test it.
- TDD approach: Write test → Fail it → Write code → Pass the test → Refactor.
- This feels backward at first and can be uncomfortable for developers used to jumping straight into coding logic.

Thinking in Small, Testable Units

TDD forces developers to:

- Break problems into very small steps.
- Focus on testability, which often leads to better design – but takes more thought up front.

TDD Impact on Developer’s Life

- Letting Tests Drive Design
  - Instead of designing your code then writing tests around it you let the tests guide your architecture.
  - This feels strange, especially for developers used to object-oriented or top-down design habits.

- Immediate vs. Long-Term Payoff
  - TDD can feel slower in the short term.
  - But over time, it creates fewer bugs, cleaner code, and faster refactoring.
  - Convincing teams of this long-term ROI can be a tough cultural sell.

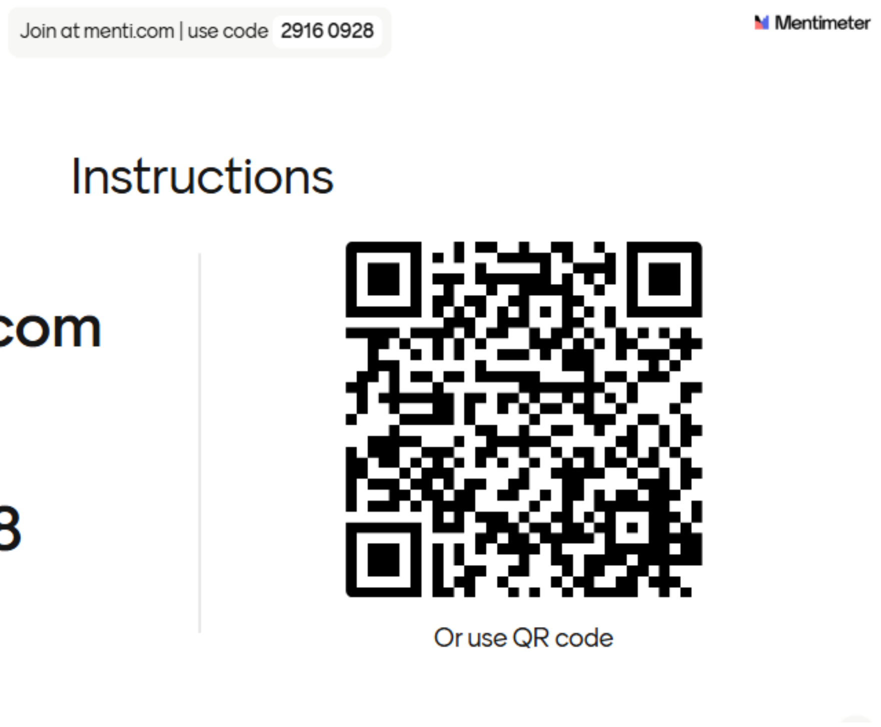
TDD Impact on Developer’s Life

- Team and Organizational Culture
  - If only one developer adopts TDD, it may not mesh well with the rest of the team’s workflow.
  - Organizational pressure to “deliver fast” often discourages writing tests first.

TDD Myths and Reality

Myth #1: TDD slows down development	**Reality:** While TDD may initially seem slower, it actually speeds up development in the long run.
Myth #2: TDD is just about writing tests and has no impact on the actual development process.	**Reality:** TDD is more than just testing. It is a design methodology that drives the development process.
Myth#3: TDD is only suitable for certain types of projects or specific programming languages.	**Reality:** TDD can be applied to almost any project, regardless of size, complexity, or technology stack.

Tutorial / Quiz



Join at menti.com | use code 2916 0928

Mentimeter

Instructions

Go to

www.menti.com

Enter the code

2916 0928

Or use QR code

Thank You

Q & A

suresh.n@sliit.lk | 755841849