Test-Driven Development (TDD)

Definition and Principles of TDD

Test-Driven Development (TDD) is a software development approach where tests are written before the actual code. The primary principle of TDD is to ensure that the code is correct from the beginning by following a structured cycle of testing and development.

Principles of TDD

1. **Write a test first** – Before writing the actual code, create a test case that defines what the function or feature should do.
2. **Run the test (It should fail)** – Since the code hasn't been implemented yet, the test will fail. This failure confirms that the test is valid.
3. **Write the minimum code to pass the test** – Implement only enough code to make the test pass.
4. **Refactor the code** – Improve the code structure while keeping the functionality intact.
5. **Repeat the cycle** – Continuously repeat the process for new features and improvements.

Benefits of TDD in Software Development

- **Higher Code Quality**: Ensures well-structured, maintainable, and efficient code.
- **Fewer Bugs**: Reduces errors by catching issues early in the development cycle.
- **Improved Design**: Leads to modular and well-designed code.
- **Faster Debugging**: Since tests identify failing code quickly, debugging becomes easier.
- **Confidence in Code Changes**: Refactoring is safer because existing tests confirm correct behavior.
- **Better Collaboration**: Helps teams understand the intended functionality before coding starts.

TDD Lifecycle: Red-Green-Refactor

The TDD process follows three main stages, often called the **Red-Green-Refactor** cycle:

1. **Red (Write a Failing Test)**
   - Create a test case that specifies the desired functionality.
   - Run the test; it should fail because the code doesn't exist yet.
2. **Green (Write the Minimum Code to Pass the Test)**
   - Write just enough code to make the test pass.
   - Avoid unnecessary code at this stage.
3. **Refactor (Improve Code without Changing Behavior)**
   - Clean up the code while ensuring all tests still pass.
   - Optimize performance and structure without altering functionality.

TDD vs. Traditional Development

Feature	Test-Driven Development (TDD)	Traditional Development
Testing Approach	Write tests before writing code	Write code first, then test
Code Quality	Higher due to structured testing	Lower, as testing is often an afterthought
Bug Detection	Early detection before development	Found later, requiring more fixes
Development Speed	Initially slower but saves time later	Faster at first but debugging takes longer
Refactoring	Safe due to existing tests	Riskier as changes may break functionality
Confidence in Changes	High, since tests verify behavior	Lower, as manual testing is needed

TDD helps ensure that software is **reliable, well-structured, and maintainable** while reducing bugs and costly rework. Although it requires discipline, the long-term benefits make it a preferred method for modern software development.

Here's a simple example of **Test-Driven Development (TDD) in Java** using **JUnit**. We'll follow the **Red-Green-Refactor** cycle to implement a method that calculates the sum of two numbers.

Step 1: Red - Write a Failing Test

Before writing the actual function, we first create a test case.

Test Class (CalculatorTest.java)

```
import static org.junit.jupiter.api.Assertions.*;
     import org.junit.jupiter.api.Test;

     class CalculatorTest {

         @Test
         void testAddition() {
             Calculator calculator = new Calculator();
             int result = calculator.add(2, 3);
             assertEquals(5, result); // Expecting 2 + 3 =
5
         }
     }
```

Explanation

- We define a test case **testAddition()**.
- The test expects calculator.add(2, 3) to return 5.
- Since the Calculator class and add method don’t exist yet, this test **won’t compile (fails)**.

Step 2: Green - Write the Minimum Code to Pass the Test

Now, let's implement the minimal code needed to make the test pass.

Implementation Class (Calculator.java)

```
class Calculator {
    int add(int a, int b) {
        return a + b;
    }
}
```

Explanation

- We define the Calculator class.
- We add the add(int a, int b) method that simply returns a + b.
- Now, when we run our test, it **should pass**

Step 3: Refactor - Improve Code without Changing Behavior

In this simple case, there's not much to refactor. But if we were adding more methods, we could:

- Improve method structure.
- Ensure reusability and efficiency.
- Remove redundant code.

Running the Test

To run the test using **JUnit** (make sure you have JUnit 5 installed):

```
mvn test  # If using Maven
```

If using an IDE like **IntelliJ IDEA** or **Eclipse**, simply **right-click** the test file and select **Run Tests**.

TDD in Action: Summary

Step	Action	Status
Red	Write a failing test	□ (Fails)
Green	Write the simplest code to pass the test	□ (Passes)
Refactor	Optimize the code while keeping tests green	□ (Passes)

Final Thoughts

By following TDD: We ensured **correct functionality** from the start. We wrote **only necessary code** to meet requirements. We built **confidence in our code** with automated testing.

JUnit Framework:

1. Overview of JUnit Framework

JUnit is a **widely used testing framework** for Java applications. It is part of the **xUnit** family of unit testing frameworks and helps developers write repeatable, automated tests.

Key Features of JUnit:

- Simple and easy-to-use API for unit testing.

- Supports annotations for test execution control.
- Provides assertions to verify expected outcomes.
- Integrates well with IDEs like **Eclipse, IntelliJ IDEA** and build tools like **Maven, Gradle**.
- Supports test setup and teardown using fixture methods.

---

2. Writing and Running Simple JUnit Tests

Step 1: Adding JUnit Dependency

If using **Maven**, add the following dependency in pom.xml:

```
<dependencies>
  <dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.9.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Step 2: Writing a Simple JUnit Test

Here's a basic test using JUnit 5:

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class CalculatorTest {

    @Test
    void testAddition() {
        Calculator calculator = new Calculator();
        int result = calculator.add(5, 3);
        assertEquals(8, result, "5 + 3 should be 8");
    }
}
```

Step 3: Running the Test

- In **IntelliJ IDEA** or **Eclipse**, right-click the test file and select **Run Tests**.
- Using Maven:
- mvn test

---

3. Annotations in JUnit

JUnit provides various **annotations** to control test execution.

Annotation	Description
@Test	Marks a method as a test case.
@BeforeEach	Runs **before each** test method. Used for setup.
@AfterEach	Runs **after each** test method. Used for cleanup.
@BeforeAll	Runs **once before all** test methods. Used for global setup.
@AfterAll	Runs **once after all** test methods. Used for global cleanup.
@Disabled	Skips the test method or class.

Example Using Annotations

```
import org.junit.jupiter.api.*;

import static org.junit.jupiter.api.Assertions.*;

class ExampleTest {

    @BeforeAll
    static void setupAll() {
        System.out.println("Runs once before all tests.");
    }

    @BeforeEach
    void setup() {
```

```
        System.out.println("Runs before each test.");
    }

    @Test
    void testOne() {
        assertTrue(5 > 2);
    }

    @Test
    void testTwo() {
        assertEquals("Hello", "Hello");
    }

    @AfterEach
    void teardown() {
        System.out.println("Runs after each test.");
    }

    @AfterAll
    static void teardownAll() {
        System.out.println("Runs once after all tests.");
    }
}
```

Output:

Runs once before all tests.
Runs before each test.
Runs after each test.
Runs before each test.
Runs after each test.
Runs once after all tests.

---

4. Assertions for Verifying Expected Outcomes

Assertions help validate the expected behavior of code.

Assertion	Description
assertEquals(expected, actual)	Checks if two values are equal.
assertNotEquals(expected, actual)	Ensures two values are not equal.

assertTrue(condition)	Passes if the condition is true.
assertFalse(condition)	Passes if the condition is false.
assertNull(object)	Passes if the object is null.
assertNotNull(object)	Passes if the object is not null.
assertThrows(Exception.class, () -> method())	Checks if the code throws the expected exception.

Example: Using Assertions

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

class AssertionsExampleTest {

    @Test
    void testAssertions() {
        assertEquals(10, 5 + 5, "5 + 5 should be 10");
        assertTrue(10 > 5, "10 should be greater than 5");
        assertFalse(5 > 10, "5 is not greater than 10");
        assertNotNull("JUnit");
    }

    @Test
    void testException() {
        Exception exception =
assertThrows(ArithmeticException.class, () -> {
            int result = 10 / 0;
        });
        assertEquals("/ by zero", exception.getMessage());
    }
}
```

---

5. Test Fixture Setup and Teardown

A **test fixture** is a fixed state of data and resources required before running a test.

Example: Setup and Teardown

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.*;

class CalculatorTest {
    Calculator calculator;

    @BeforeEach
    void setup() {
        calculator = new Calculator(); // Create a new
instance before each test
        System.out.println("Setup before test");
    }

    @Test
    void testAddition() {
        assertEquals(7, calculator.add(3, 4));
    }

    @Test
    void testSubtraction() {
        assertEquals(2, calculator.subtract(5, 3));
    }

    @AfterEach
    void teardown() {
        calculator = null; // Clean up after test
        System.out.println("Cleanup after test");
    }
}

class Calculator {
    int add(int a, int b) {
        return a + b;
    }
    int subtract(int a, int b) {
        return a - b;
    }
}
```

Output:

Setup before test
Cleanup after test
Setup before test
Cleanup after test

Summary

Concept	Description
JUnit Overview	A testing framework for Java that automates unit testing.
Writing Tests	Use **@Test** methods to write test cases.
Annotations	Use **@BeforeEach, @AfterEach, @BeforeAll, @AfterAll** for setup and teardown.
Assertions	Use **assertEquals, assertTrue, assertThrows** to verify expected results.
Fixtures	Set up necessary objects before tests and clean up afterward.

Advanced JUnit Testing Features

This guide covers:
Parameterized Tests – Running a test multiple times with different inputs.

Test Suites and Categories – Grouping multiple tests together. **Test Execution Order** – Controlling the order of test execution. **Exception Testing** – Verifying that methods throw expected exceptions. **Timeout and Performance Testing** – Ensuring tests complete within a time limit.

---

1. Parameterized Tests (JUnit 5)

Parameterized tests allow running the same test logic **with multiple inputs**.

Example: Testing Addition with Different Inputs

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.params.ParameterizedTest;
import org.junit.jupiter.params.provider.CsvSource;

class ParameterizedTestExample {

    @ParameterizedTest
    @CsvSource({
        "2, 3, 5",
        "10, 5, 15",
        "7, 3, 10"
    })
    void testAddition(int a, int b, int expectedSum) {
        assertEquals(expectedSum, add(a, b));
    }

    int add(int x, int y) {
        return x + y;
    }
}
```

Explanation:

- **@ParameterizedTest** runs the test multiple times.
- **@CsvSource** provides different input sets (comma-separated values).

Other Ways to Provide Test Data

Annotation	Data Type Example
@ValueSource(ints = {1, 2, 3})	Single values
@CsvSource({"1,2", "3,4"})	Multiple values
@CsvFileSource(resources = "/test-data.csv")	From a CSV file
@MethodSource("methodName")	Dynamic test data

---

2. Test Suites and Categories

A **test suite** is a collection of test classes that can be executed together.

JUnit 5: Running Multiple Test Classes in a Suite

```
import org.junit.platform.suite.api.*;

@Suite
@SelectClasses({CalculatorTest.class, MathUtilsTest.class})
public class TestSuiteExample {}
```

Explanation:

- **@Suite** – Defines a test suite.
- **@SelectClasses({Class1, Class2})** – Specifies the test classes to run.

Grouping Tests Using Tags

```
import org.junit.jupiter.api.Tag;
import org.junit.jupiter.api.Test;

class TaggedTests {

    @Test
    @Tag("slow")
    void slowTest() {
    }
```

```
    @Test
    @Tag("fast")
    void fastTest() {
    }
}
```

---

3. Test Execution Order

JUnit **does not guarantee execution order** unless explicitly controlled.

Controlling Execution Order

```
import org.junit.jupiter.api.*;

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)
class OrderedTests {

    @Test
    @Order(1)
    void firstTest() {
        System.out.println("Running first test");
    }

    @Test
    @Order(2)
    void secondTest() {
        System.out.println("Running second test");
    }
}
```

Other Test Order Strategies

Order Strategy	Description
MethodOrderer.Random	Random execution order
MethodOrderer.Alphanumeric	Alphabetical order
MethodOrderer.OrderAnnotation	Uses @Order annotation

---

4. Exception Testing

Tests should verify that **expected exceptions** are thrown.

Example: Division by Zero Should Throw Exception

```
import static
org.junit.jupiter.api.Assertions.assertThrows;
import org.junit.jupiter.api.Test;

class ExceptionTest {

    @Test
    void testDivideByZero() {
        assertThrows(ArithmeticException.class, () ->
divide(10, 0));
    }

    int divide(int a, int b) {
        return a / b;
    }
}
```

Explanation:

- assertThrows(Exception.class, () -> methodCall()) ensures that the method throws the expected exception.

---

5. Timeout and Performance Testing

Sometimes, we need to ensure that tests complete **within a time limit**.

Setting a Timeout for a Test

```
import static
org.junit.jupiter.api.Assertions.assertTimeout;
import java.time.Duration;
import org.junit.jupiter.api.Test;
```

```
class TimeoutTest {

    @Test
    void testShouldCompleteWithin1Second() {
        assertTimeout(Duration.ofSeconds(1), () -> {
            Thread.sleep(500); // Simulating a delay
        });
    }
}
```

Explanation:

- assertTimeout(Duration.ofSeconds(1), () -> code) ensures that the test completes within 1 second.

Failing a Test If It Runs Too Long

```
import org.junit.jupiter.api.*;

class TimeoutFailureTest {

    @Test
    @Timeout(2) // Fails if it runs longer than 2 seconds
    void testTimeout() throws InterruptedException {
        Thread.sleep(3000); // Simulating a long operation
    }
}
```

Explanation:

- **@Timeout(2)** makes the test fail if it runs for more than **2 seconds**.

Summary

Feature	Description
Parameterized Tests	Run a test multiple times with different inputs.
Test Suites & Categories	Group tests together and execute based on tags.
Test Execution Order	Control the order of test execution with **@Order.**
Exception Testing	Verify that methods throw expected exceptions using **assertThrows().**
Timeout & Performance Testing	Ensure tests run within a given time limit using **@Timeout or assertTimeout().**