**What is Test Driven Development (TDD)?**

**What is Test Driven Development (TDD)?**

Test-Driven Development (TDD) is a method in software development where the focus is on writing an [**Automation Tests**](https://www.geeksforgeeks.org/automation-testing-software-testing/) before writing the actual code for any feature of an application or producrt

TDD simply means a method of coding in which you first write a test, and it fails, then write the code to pass the test of development, and clean up the code

 In other methods in which you write either all the code or all the tests first, TDD will combine and write tests and code together into one.

The process of Test-Driven Development (TDD) follows a repetitive cycle called **Red-Green-Refactor**.

Run all the test cases and make sure that the new test case fails.

* **Red -** Create a test case and make it fail, Run the test cases
* **Green -** Make the test case pass by any means.
* **Refactor -** Change the code to remove duplicate/redundancy and Refactor code - This is done to remove duplication of code.(improve code )

For each use case write the test case and it failes and write the production code

To make testing successful, it needs to be created at both small and big levels in test-driven development.

This means testing every part of the work, like methods in a class, input data values, log messages, and error codes.

**Advantages of Test Driven Development (TDD)**

* Unit test provides constant feedback about the functions.
* Quality of design increases which further helps in proper maintenance.
* Test driven development act as a safety net against the bugs.
* TDD ensures that your application actually meets requirements defined for it.
* TDD have very short development lifecycle.

**Disadvantages of Test Driven Development (TDD)**

* **Increased Code Volume**:Using TDD means writing extra code for tests cases , which can make the overall codebase larger and more Unstructured.
* **False Security from Tests**: Passing tests will make the developers think the code is safer only for assuming purpose.
* **Maintenance Overheads**: Keeping a lot of tests up-to-date can be difficult to maintain the information and its also time-consuming process.
* **Time-Consuming Test Processes**: Writing and maintaining the tests can take a long time.
* **Testing Environment Set-Up**: TDD needs to be a proper testing environment in which it will make effort to set up and maintain the codes and data.

## Conclusion

**Test-Driven Development (TDD)** is a coding methodology where tests are written before the actual code. This process verify the code reliability, quality, and maintainability through its **Red-Green-Refactor** cycle.

TDD offers several advantages, including constant feedback and improved design quality. However, it also presents challenges such as increased code volume and maintenance efforts. Despite these drawbacks, TDD remains a valuable approach for [**Developing High-Quality Software**,](https://www.geeksforgeeks.org/what-is-the-need-of-software-engineering/) promoting early bug detection, and aligning the code with [**Business Requirements**](https://www.geeksforgeeks.org/difference-between-brd-and-srs/).

**Unit Testing - Software Testing**

**Unit Testing** is a software testing technique in which individual units or components of a software application are tested in isolation. These units are the smallest pieces of code, typically functions or methods, ensuring they perform as expected.

vverify the code's correctness by running one by one

It's a key part of software development that improves code quality by testing each

You write unit tests for these code units and run them automatically every time you make changes

**Unit testing strategies**

To create effective unit tests, follow these basic techniques to ensure all scenarios are covered:

* **Logic checks**: Verify if the system performs correct calculations and follows the expected path with valid inputs. Check all possible paths through the code are tested.
* **Boundary checks**: Test how the system handles typical, edge case, and invalid inputs. For example, if an integer between 3 and 7 is expected, check how the system reacts to a 5 (normal), a 3 (edge case), and a 9 (invalid input).
* **Error handling**: Check the system properly handles errors. Does it prompt for a new input, or does it crash when something goes wrong?
* **Object-oriented checks**: If the code modifies objects, confirm that the object's state is correctly updated after running the code.
* [**DevOps**](https://www.geeksforgeeks.org/courses/devops-live)**and**[**CI/CD**](https://www.geeksforgeeks.org/what-is-ci-cd/)**:** In DevOps environments, Continuous Integration/Continuous Delivery (CI/CD) automatically runs unit tests whenever new code is added. This ensures that changes are integrated smoothly, tested thoroughly, and deployed efficiently, maintaining overall code quality.

**Difference Between Unit Testing and other Types of Testing**

There are several types of software testing methods, each having its specific role:

* [Integration Testing](https://www.geeksforgeeks.org/software-engineering-integration-testing/) will be check that different parts of the system work together as expected.
* [Functional Testing](https://www.geeksforgeeks.org/software-testing-functional-testing/) checks if the software meets the pre-planned requirements.
* [Performance Testing](https://www.geeksforgeeks.org/performance-testing-software-testing/) will make if the software runs smoothly, such as in the terms of speed and memory usage.
* [Acceptance Testing](https://www.geeksforgeeks.org/acceptance-testing-software-testing/) is performed manually by stakeholders to verify the software behaves as expected as we wants.
* [Security Testing](https://www.geeksforgeeks.org/security-testing/) examines the software for vulnerabilities, including third-party risks in the Software development process.

**Types of Unit Testing**

Unit testing can be performed manually or automatically:

**1. Manual unit testing**

[**Manual Testing**](https://www.geeksforgeeks.org/software-testing-manual-testing/)**is like checking each part of a project by hand, without using any special tools. People, like developers, do each step of the testing themselves. But manual unit testing isn't used much because there are better ways to do it and it has some problems:**

* **It costs more because workers have to be paid for the time they spend testing, especially if they're not permanent staff.**
* **It takes a lot of time because tests have to be done every time the code changes.**
* **It is hard to find and fix problems because it is tricky to test each part separately.**
* **Developers often do manual testing themselves to see if their code works correctly**

**. Automated unit testing:**

**We use special tools made by people to run these tests automatically**

**Unit Testing Techniques**

There are 3 types of Unit Testing Techniques. They are follows:

1. [Black Box Testing](https://www.geeksforgeeks.org/software-engineering-black-box-testing/): This testing technique is used in covering the unit tests for input, user interface, and output parts.
2. [White Box Testing](https://www.geeksforgeeks.org/software-engineering-white-box-testing/): This technique is used in testing the functional behavior of the system by giving the input and checking the functionality output including the internal design structure and code of the modules.
3. [Gray Box Testing](https://www.geeksforgeeks.org/gray-box-testing-software-testing/): This technique is used in executing the relevant test cases, test methods, and test functions, and analyzing the code performance for the modules.

* **JUnit**: A widely-used Java testing framework for creating and running unit tests.
* **NUnit**: A .NET framework for unit testing C# applications.
* **TestNG**: An advanced Java testing framework with features like parallel testing.

The **JUnit 5** version has three different modules for defining and performing different functionalities in testing. The components are:

* **JUnit Platform**
* **JUnit Jupiter**
* **JUnit Vintage**

These three components play an important role in writing test cases for a software application.

**JUnit Platform**

, In Java If we want to run the test cases, we need JVM. So, the JUnit Platform provides a launching mechanism for testing frameworks on the JVM.

**JUnit Jupiter**

This component provides new programming techniques for developing the test cases in JUnit 5.

**JUnit Vintage**

he Main functionality of JUnit Vintage is allowing JUnit 3 and JUnit 4 Test cases on the JUnit 5 Platform. Without change

**What is an Assertion in JUnit?**

**Assertions** in JUnit are statements that check if a certain condition is true. They are used to verify that your code behaves as expected during testing.

If the assertion fails, JUnit marks the test as failed and usually shows an error message indicating what went wrong.

Common assertions include:

* assertEquals(expected, actual): Checks if two values are equal.
* assertTrue(condition): Checks if a condition is true.
* assertFalse(condition): Checks if a condition is false.
* assertNotNull(object): Checks if an object is not null.

**Annotations**

The JUnit 5 framework uses different Annotations based on the test case design. Mostly in JUnit 5 **@Test, @BeforeEach, @AfterEach, @BeforeAll, @AfterAll, @DisplayName, @Disabled** these annotations are used. Basically, Annotations provides supplement information about the program in java. Annotations are always start with "@" symbol. ( [reference](https://www.geeksforgeeks.org/annotations-in-java) ).

* @Test: Marks a method as a test method.
* @BeforeEach: Indicates that the annotated method should be executed before each test.
* @AfterEach: Indicates that the annotated method should be executed after each test.
* @BeforeAll: Indicates that the annotated method should be executed before all tests in the test class.
* @AfterAll: Indicates that the annotated method should be executed after all tests in the test class.
* @DisplayName: Provides a custom name for the test class or test method.
* @Disabled: Disables the test method or class.

**What are Assumptions in JUnit?**

**Assumptions** are used to **skip tests or parts of tests** if certain conditions are **not met**. They act like preconditions that must be true for the test to run.

* If an assumption fails, JUnit **does not fail the test** — it **ignores or skips** the test.
* Useful when a test only makes sense in a specific environment or condition.

### Common Assumption Methods (JUnit 5)

| **Method** | **Description** | **Example** |
| --- | --- | --- |
| assumeTrue(condition) | Continue only if condition is true | assumeTrue(isDatabaseAvailable()); |
| assumeFalse(condition) | Continue only if condition is false | assumeFalse(isWindowsOS()); |
| assumingThat(condition, Executable) | Run the block only if condition is true | assumingThat(isLinux(), () -> { /\* Linux-specific test \*/ }); |

**What is a Parameterized Test?**

A **parameterized test** runs the **same test method multiple times** with different inputs (parameters). This helps you test multiple cases without writing separate test methods for each case.

 @ParameterizedTest: Marks the test method as parameterized.

 @ValueSource: Provides the source of input values (here, an array of ints).

import org.junit.jupiter.params.ParameterizedTest;

import org.junit.jupiter.params.provider.ValueSource;

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

public class MyParameterizedTest {

@ParameterizedTest

@ValueSource(ints = {1, 2, 3, 4, 5})

void testSquare(int value) {

int result = square(value);

assertEquals(value \* value, result, "Square calculation is incorrect");

}

private int square(int number) {

return number \* number;

}

}

* The test method takes a parameter value.
* JUnit runs this method **once for each value** provided by @ValueSource.

int result = square(value);

assertEquals(value \* value, result, "Square calculation is incorrect");

If any test fails, it shows the message "Square calculation is incorrect".

**Dynamic Tests**

JUnit 5 introduces the concept of dynamic tests, which can be generated at runtime. These are created using the **DynamicTest class**.

For creating Dynamic Tests in Run time by using **@TestFactory** annotation. This TestFactory provides a feature to create dynamic test case in the run time of the Application.

**Tagging and Filtering**

We can tag our test cases by using @tag annotation in the JUnit 5. Simply the Tags are labels for categorize the test cases. And Filter is used to filter and run the test cases by using the Tags.

**What is the AAA Pattern?**

**AAA** stands for:

* **Arrange** — Set up the objects, mocks, and prerequisites for your test.
* **Act** — Execute the action or method you want to test.
* **Assert** — Verify the result or behavior is as expected.

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

import org.junit.jupiter.api.Test;

public class CalculatorTest {

@Test

void testAddition() {

// Arrange

Calculator calc = new Calculator();

// Act

int result = calc.add(2, 3);

// Assert

assertEquals(5, result, "2 + 3 should equal 5");

}

}**Step-by-step**

1. **Arrange:**  
   Prepare everything needed for the test. Here, creating an instance of Calculator.
2. Creating instances of the classes to be tested
3. Initializing global or local variables with specific values
4. Setting up mock objects to simulate external services
5. Populating a sample database with test data
6. Configuring specific settings and configurations

**Act:**  
Call the method you want to test, calc.add(2, 3).

. The goal is to execute the action that will generate the outcome you intend to verify in the next step.

1. **Assert:**  
   Check if the result is what you expect, using assertEquals.
2. Verifying that a method returns the correct value
3. Checking that an object’s state has changed as anticipated
4. Guaranteeing that an exception is thrown under certain conditions

**Arrange**: Set up the test environment.

1. **Act**: Execute the code to test.
2. **Assert**: Verify the results.

Parameterized test case:

**Using @NullSource and @EmptySource**

We often need to test our code with null and empty values to be assured that we can process them correctly and avoid the most loved `NullPointerException`.

Parameterized test argument sources provide us the built-in support for such test data with  @NullSource and @EmptySource annotations

@ParameterizedTest

@NullSource

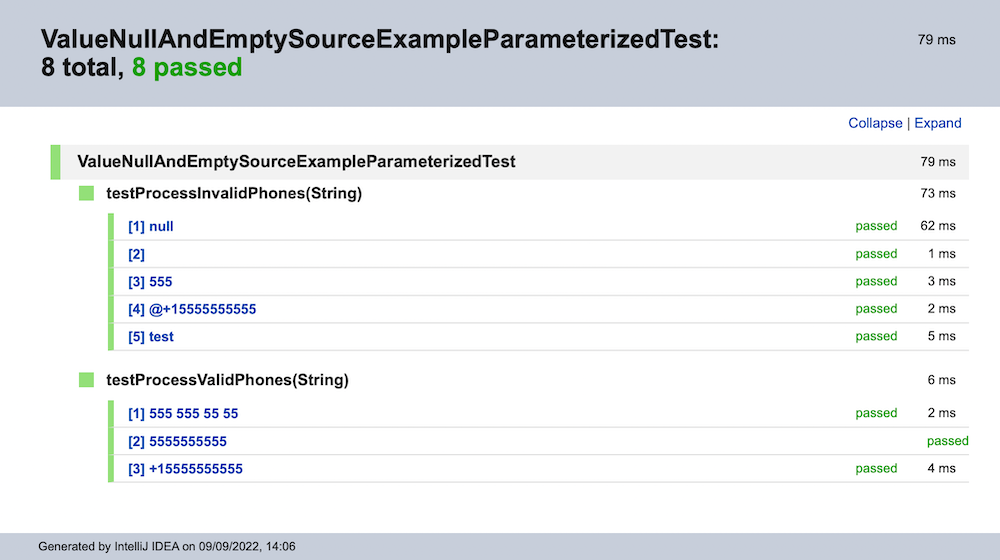
@EmptySource

@ValueSource(strings = {"555", "@+15555555555", "test"})

void testProcessInvalidPhones(String phone) {

assertFalse(phoneValidationService.validatePhone(phone));

}



**Test Suite**

n JUnit 5, a test suite is a way to combine multiple test classes and run them together as a single unit

 @Suite annotation on the test class is used to define a test suite

You can also include specific test classes in the suite using annotations like @SelectClasses, @SelectPackages, or @SelectClasspathRoots, that help to filter out the test classes you want to run together

[JUnit 5 - Test Suites with Example - GeeksforGeeks](https://www.geeksforgeeks.org/junit-5-test-suites-with-example/)

**JUnit 5 – Test Execution Order**

In **JUnit 5,** the **Test execution order** is not in the order by default. Every Test is randomly executed one after the other.

So, the order of execution of test cases is unpredictable.

Basically, in JUnit 5 we have 3 ways to control the order of test execution:

* By Annotation Order
* Alphanumeric Order
* Random Order

**1. Annotation Order**

The test execution order here will depend on the order number we give inside the **@order** annotation for the method we want to test. So, according to the value which is inside the **@order** annotation the execution of test cases will be taken place.

@TestMethodOrder(MethodOrderer.OrderAnnotation.class)

**2. Alphanumeric Order**

The test execution order here will be sorted in a lexicographic (alphanumerical) order. So, based on the method names the test cases will be executed. In order the test cases to execute in this way we must annotate the test class with **@TestMethodOrder(MethodOrderer.Alphanumeric.class).**

**3.Random Order**

The test execution order here is executed randomly. There will be no specific order that when a test case is executed. So here we us TestMethodOrder(MethodOrderer.Random.class) annotation to execute the test cases randomly when we run the test cases.

**What is Test Execution?**

T

est Execution is the process of executing the tests written by the tester to check whether the developed code or functions or modules are providing the expected result as per the client requirement or business requirement. Test Execution comes under one of the phases of the Software Testing Life Cycle (STLC).

**Activities for Test Execution**

 5 main activities

**Defect Finding and Reporting**

**Defect Mapping**

After the error has been detected and reported to the development team, the development team will work on those errors and fix them as per the requirement. Once the development team has done its job, the tester team will again map the test cases or test scripts to that developed module or code to run the entire tests to ensure the correct output.

**Re-Testing**

**Regression Testing:**Regression Testing is software testing that ensures that the newly made changes to the code or newly developed modules or functions should not affect the normal processing of the application or product.

**System Integration Testing:**SystemIntegration Testing is a type of testing technique that will be used to check the entire component or modules of the system in a single run. It ensures that the whole system will be checked in a single test environment instead of checking each module or function separately

**Ways to Perform Test Execution**

**Test Execution States**

The tester or the Quality Analyst team reports or notices the result of each test case and records it in their documentation or file. There are various results raised when executing the test cases. They are

* **Pass:**It tells that the test cases executed for the module or function are successful.
* **Fail:** It tells that the test cases executed for the module or function are not successful and resulted in different outputs.
* **Not Run:** It tells that the test cases are yet to be executed.
* **Partially Executed:** It tells that only a certain number of test cases are passed and others aren't met the given requirement.
* **Inconclusive:** It tells that the test cases are executed but it requires further analysis before the final submission.
* **In Progress:** It tells that the test cases are currently executed.
* **Unexpected Result:** It tells that all the test cases are executed successfully but provide different unexpected results.

**What is Exception Testing?**

**Exception testing checks whether your method or code block throws a specific exception when given invalid or edge-case input.**

**This is useful for:**

* **Validating error handling**
* **Confirming constraints (e.g., divide-by-zero, null input)**
* **Ensuring robust application behavior under bad conditions**

### 1. **Using** assertThrows()

The most common and preferred way.

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

**@Test**

**void testDivideByZero() {**

**ArithmeticException exception = assertThrows(**

**ArithmeticException.class,**

**() -> divide(10, 0)**

**);**

**assertEquals("/ by zero", exception.getMessage());**

**}**

 assertThrows() checks that the lambda expression throws ArithmeticException.

 It **passes** if the exception is thrown.

 It **fails** if:

* No exception is thrown
* A different exception is thrown

The **@Timeout**annotation in **JUnit**restricts the duration of test and lifecycle methods. It aids in ensuring that a test procedure is completed according to the planned timetable. **TimeoutExceptions** occur when the duration of the test method exceeds the specified limit.

**What is performance testing?**

Performance testing is a type of software testing that focuses on evaluating the performance and scalability of a system or application

 It is a testing method to determine the system's performance in terms of speed, reliability, and stability under varying workloads.

Performance testing is also known as ***Perf Testing***. The types of Performance testing are as follows: