What is Test Driven Development (TDD) : Approach & Benefits?

What is Test Driven Development (TDD)?

In layman’s terms, Test Driven Development (TDD) is a software development practice that focuses on creating unit test cases before developing the actual code. It is an iterative approach that combines programming, the creation of unit tests, and refactoring.

The TDD approach derives its roots from the Agile manifesto principles and Extreme programming. As the name suggests, the test process drives software development. Moreover, it’s a structuring practice that enables developers and testers to obtain optimized code that proves to be resilient in the long term.

In TDD, developers start creating small test cases for every feature based on their initial understanding. The primary intention of this technique is to modify or write new code only if the tests fail. This prevents duplication of test scripts.

**Simple Example** 🡪

@Test

void test\_addition\_ofTwoNumbers(){

int expected\_output=20

int actual\_output= cal.addition(10,10);

assertThat(actual\_output).isEqualTo(expected\_output);

}

public void addition(int num1, int num2){

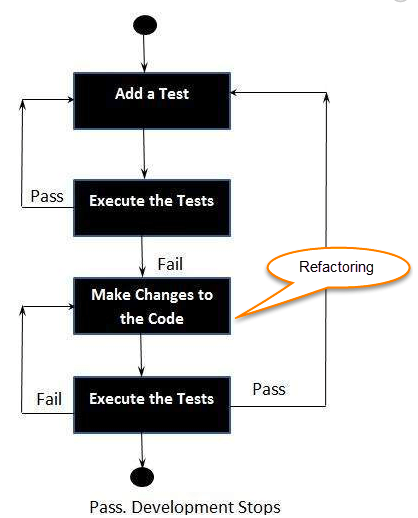
return num1+num2;

}

### Three phases of Test-Driven Development

1. **Create precise tests:** Developers need to create precise unit tests to verify the functionality of specific features. They must ensure that the test compiles so that it can execute. In most cases, the test is bound to fail. This is a meaningful failure as developers are creating compact tests based on their assumptions of how the feature will behave.
2. **Correcting the Code:** Once a test fails, developers need to make the minimal changes required to correct the code so that it can run successfully when re-executed.
3. **Refactor the Code:** Once the test runs successfully, check for redundancy or any possible code optimizations to enhance overall performance. Ensure that refactoring does not affect the external behavior of the program.

The image below represents a high-level TDD approach towards development:



# Application Under Test

Definition:

AUT is “Application under test”. After designing and coding section of development cycle, when the application(build) comes under testing then at that time application state is under test, so at that time period that application(build) is called “**Application under test**”.

## JUnit Test Fixture.

When there are multiple test cases in a JUnit class, there could be a common object or objects used by all the test cases. In this case, there could be specific functions that might be common throughout all the test cases.

**Term ‘Test Fixture’** is a fixed state in a code or a set of fixed steps in a code that is used as a precondition and few other sets of steps that are used as postcondition for all the tests.

Thus, in other words, we are identifying those sets of statements that will repeat for all tests and thereby, try setting a fixed environment for our test methods to run.

The **purpose** of using Test Fixture is to eliminate the duplication of the common code for all the testcases.

class testCalculator{

Calculator cal;

@Test

void test\_addition\_ofTwoNumbers(){

int expected\_output=20

int actual\_output= cal.addition(10,10);

assertThat(actual\_output).isEqualTo(expected\_output);

}

@Test

void test\_multiplication\_ofTwoNumbers(){

int expected\_output=100

int actual\_output= cal.addition(10,10);

assertThat(actual\_output).isEqualTo(expected\_output);

}

}

Calculator.java

Class Calculator{

public void addition(int num1, int num2){

return num1+num2;

}

public void multiplication(int num1, int num2){

return num1\*num2;

}

Let’s try to understand the practical implementation of the test fixture in a JUnit test.

### setUp() method (user defined method)

* There are tests that need initialization of certain objects (string, integer, or ArrayList or any object for that matter). You may create a method **public void setUp()** in which you could declare the instance variables for the common objects. Place this setUp() method under the annotation **@BeforeEach**. With the @BeforeEach annotation, the framework will run the method setUp() prior to every test case execution.
* The setUp() method could also be used in case, you wish to launch a browser with a specific URL as the first step you run a test case that follows login to the application with predefined credentials.

### tearDown() method (user defined method)

* If you have allocated external resources in a test, you should remember to free the resources too. The teardown() method could be added for the clean-up of the objects after the test case execution has been completed. In a similar fashion as the setUp() method, add a method **public void teardown()** under **@AfterEach** annotation. The JUnit framework makes sure that after each test case is run, the method under @AfterEach is surely executed. The objects used up in the test have to be set NULL in the teardown() method so that the garbage from the tests get collected.
* Another good candidate for the clean-up of the objects is killing a browser in action after the test case is completed and releasing the memory by destroying the used objects.
* **The method names setUp() and tearDown() are just a user-defined name. You may set any method name that you wish to. It is just the annotations used with the method which decides the sequence of the tests’ run.**

Calculator cal;

**@BeforeEach**

public void setup(){

cal = new Calculator();

}

**@AfterEach**

**public void teardown()** {

cal=null;

}

JUnit - Test Framework

JUnit is a **Regression Testing Framework** used by developers to implement unit testing in Java, and accelerate programming speed and increase the quality of code. JUnit Framework can be easily integrated with either of the following −

* Eclipse
* Ant
* Maven and with many more…

Features of JUnit Test Framework

JUnit test framework provides the following important features −

* Fixtures
* Test suites
* Test runners
* JUnit classes

Fixtures

**Fixtures** is a fixed state of a set of objects used as a baseline for running tests. The purpose of a test fixture is to ensure that there is a well-known and fixed environment in which tests are run so that results are repeatable. It includes −

* setUp() method, which runs before every test invocation.
* tearDown() method, which runs after every test method.

JUnit Classes

JUnit classes are important classes, used in writing and testing JUnits. Some of the important classes are −

* **Assert** − Contains a set of assert methods.
* **TestCase** − Contains a test case that defines the fixture to run multiple tests.
* **TestResult** − Contains methods to collect the results of executing a test case.

JUnit naming conventions

Naming convention for unit tests are very important. We have to make sure that is same naming convention is applied to all unit tests.

A good naming convention provide all information about that test.

There is a standard for naming strategy🡪

[UnitOfWork\_StateUnderTest\_ExpectedBehaviour]

UnitOfWork 🡪 is name of method being tested.

StateUnderTest 🡪 represent input values for methods

ExpectedBehavior 🡪 is method returns expected output

Test naming is important for teams on long term project as any other code style conventions. By applying code convention in tests you proclaim that each test name will be readable, understandable and will have a well known naming pattern for everyone on the project.

Before you choose the one naming convention you have to decide first why do you actually need it, what is the purpose of it?

There are few recommendations regarding test naming:

* **Test name should express a specific requirement**
* **Test name could include the expected input or state and the expected result for that input or state**
* **Test name should be presented as a statement or fact of life that expresses workflows and outputs**
* **Test name could include the name of the tested method or class**
* **MethodName\_StateUnderTest\_ExpectedBehavior**
  + cons: should be renamed if method change name
  + example: isAdult\_AgeLessThan18\_False
* **MethodName\_ExpectedBehavior\_StateUnderTest**
  + cons: should be renamed if method change name
  + example: isAdult\_False\_AgeLessThan18
* **testFeatureBeingTested**
  + cons: “test” prefix is redundant
  + example: testIsNotAnAdultIfAgeLessThan18
* **FeatureToBeTested**
  + cons: no clue what result is expected from name
  + example: IsNotAnAdultIfAgeLessThan18
* **Should\_ExpectedBehavior\_When\_StateUnderTest**
  + cons: duplicates `should` and `when`, long name
  + example: Should\_ThrowException\_When\_AgeLessThan18
* **When\_StateUnderTest\_Expect\_ExpectedBehavior**
  + cons: duplicates `when` and `expect`
  + example: When\_AgeLessThan18\_Expect\_isAdultAsFalse
* **Given\_Preconditions\_When\_StateUnderTest\_Then\_ExpectedBehavior — Behavior-Driven Development**(BDD)
  + *cons: duplicates `given`, `when`, `then`; really long names*
  + *example: Given\_UserIsAuthenticated\_When\_InvalidAccountNumberIsUsedToWithdrawMoney\_Then\_TransactionsWillFail.*

There are dozens of conventions and it is actually not important which of them you will choose it is important to choose one.

The Order of Tests in Junit ->