TDD :- Test Driven Development

Software development is a process where error occurs very frequently.

In traditional development approaches errors are found at the build stage or when the s/w is being used by the end user, which is the drawback.

TDD refers to a style of programming in which 3 activities are are tightly interconnected:

Coding, Testing and Design. TDD follows “Red”, “Green”, “Refactor” cycle.

Steps for TDD:

Red :- Writing a failing(automated) test before writing production code.

Green :- make a minimal change to production code to fix the failing test.

Refactor :- when there are enough tests,…

Is an Iterative development process. Every iteration start with a set of tests written for a new functionality.

Test cases are created before code is written.

TDD instructs us to write new code only if an automated test has failed.

Testing phase

Coding phase

TDD

increment

1. TDD focuses on Unit testing
   1. Unit Testing is not the same thing as TDD.
2. If you write your logic first, then your test- this is considered as Unit testing, but not TDD.

Main benefits of TDD

1. Reduction in bugs – Test is done
2. Make code clearer, simpler and bug free.
3. Avoids duplication of code.

Disadvanages of TDD

1. Requirement changes, test case need to change.
2. Process is slow

Sequence of steps

1. Add a test case Requiremet
2. Run test if fails Executing
3. Write code to fix it
4. Run test
5. Refactor code

Add a test

1. Repeat

Run

pass

fails

Make a little change

Run the test

Refactor

TDD way

1. Write a failing test first
2. Change the minimal amount of code to pass the test
3. Refactor the logic to get a standard output.

Verification and Validation

Verfication is the process of evaluationg whether the product meets the requirements of a particular phase

Ex:- Am I building the right product?

Validation is the process of evaluating the product at the end phase to determine whether it satisfies the specific requirements.

Ex:- Am I building the product right?

Scrum Principle :- Inspect and Adapt

**Junit testing – Mockito / TDD**

**Jsp servlets jstl**

**Spring web mvc**

**Spring boot**

**Rest controller**

**Setup Junit in Eclipse**

1. **Download junit jar files – External jar files**
2. **Junit Library – if not found, you can install an external plugin from Eclipse market space.**

**JUnit**

**Junit is a testing framework for Java programming language. It is an open source testing framework it is important in the development of test-driven frameworks. It is used to verify a small chunk of code by creating a path function or a method. Junit works on the idea of tests first and code later.**

**Junit is a Regression Testing framework.**

**Advantages**

1. **It is included in many application developers IDE’s.**
2. **Provides testing report for swing-based applications**
3. **JUnit Enterprise Edition enables to test the cases within the application’s server container.**
4. **Efficient testing process.**

**Features**

**Faster , increases quality**

**Less complex – very simple**

**Provides annotations to identify test methods.**

**It provides test runners for running the test.**

**Junit checks with their results and provide immediate feedback**

**Tests can be organized into Test suite containing all the tests.**

**Junit framework is built on annotations.**

**\*\*\*\* An Annotation is a special form of pre-compiled class or metadata that can be added to the java source code for better code readability.**

**These annotations provide the information regarding the methods that are going to be run before and after the test methods, the methods that are run before and after all the methods are completed.**

**@Test , @Before, @After, @BeforeClass , @AfterClass, @Ignore.**

**Assert is the method which is used to determine pass or fail status of a test case. In JUnit all the assertions are in the assert class, so this class basically provides a set of assertion methods, these methods are useful in writing the test case.**

1. **Void assertEquals(Boolean exp, Boolean actual) :- checks two primitive/objects are equal.**
2. **Void assertTrue(Boolean condition) :- that a condition is true**
3. **Void assertFalse(boolean condition) :- that a condition is false.**
4. **assertNotNull(Object object):- that the object isn’t null**
5. **assertNull(Object object):- that an object is null**
6. **assertSame(Object obj1, Object obj2) :- if two Object references point to the same object.**
7. **assertNotSame(Object obj1, Object obj2):- do no point**
8. **assertArrayEquals(expArray,resArray):- whether two arrays are equal to each other.**

**Test Runner :- is used for executing the test cases.**

1. **Create a TestRunner java class**
2. **Use runclasses method of JUnitCore class of JUnit to run the test case**
3. **Get the result of test cases in the Result Object.**
4. **Get failure(s) using the getFailures() of Result object.**
5. **Get success result using the wasSuccessful() of Result object.**

**TestCase class :- it defines the multiple tests to be fixed for execution.**

**Methods:-**

**TestResult run() :- very simple and convienent method to run the test and collect the results**

**void run(TestResult result) :- same**

**void setup() :- for starting up the environment**

**void teardown() :- close the environment**

**Parameterized Test :- allows us to use/run the same test again and again using different values.**

**Steps**

* 1. **Annotate the class with @RunWith(Parameterized.class)**
  2. **Create a static method annotated with @Parameters that returns a Collection of Object(Arrays) as test result.**
  3. **Create a constructor – row set**
  4. **Create an instance variable for each column set.**
  5. **Run the test**

**Example**

@RunWith(Parameterized.**class**)

**public** **class** ArithmeticTest {

**private** **int** a;

**private** **int** b;

**private** **int** expected;

// row

**public** ArithmeticTest(**int** a, **int** b, **int** expected) {

**this**.a=a;

**this**.b=b;

**this**.expected=expected;

}

@Test

**public** **void** test() {

ArithmeticDemo ad=**new** ArithmeticDemo();

**int** r=ad.add(a, b);

*assertEquals*(expected, r);

}

//instance variable for each column

@Parameters

**public** **static** Collection parameters(){

**return** Arrays.*asList*(**new** Object[][]

{

{10,20,30},

{1,1,2},

{10,2,12}

}

);

}

}

**Exceptions :- uncertainity occurring at run time**

**JUnit provides an option of tracing the exceptions that occur in the code. The expected parameter is used along with @Test annotation. While testing for an exception we need to ensure that exception class that we’re providing in the optional parameter of the test annotation is same.**

**Example**

**public** **class** ExceptionDemo {

**private** String message;

**public** ExceptionDemo(String m){

**this**.message=m;

}

**public** **void** divide(**int** number){

**int** result=12/number;

}

**public** String printMessage(){

message="Hello " + message;

System.***out***.println(message);

**return** message;

}

}

**public** **class** ExceptionTest {

**public** String message="Cannot divide by zero";

ExceptionDemo demo=**new** ExceptionDemo(message);

@Test(expected=ArithmeticException.**class**)

**public** **void** test() {

System.***out***.println(message);

demo.divide(0);

//demo.printMessage();

}

**==============================================================**

**Timeout :- if any test does not complete its execution in given time limit then its execution will stop by JUnit.**

**Why Timout ? :- measure the performance of a test case**

**public** **class** PerformanceTest {

@Test(timeout=10)

**public** **void** test() {

PerformanceDemo demo=**new** PerformanceDemo();

demo.performance();

}

}

**JUnit - Test Suite**

**It is used to group a few unit test cases and run them altogether. @RunWith and @Suite annotations.**