Junit

pre-requisite :: Core Java + Eclipse IDE knowledge + (Adv.java basics) +maven tool

his

Entire Testing on the project will be done by testers.. Only unit testing and peer testing will be done by the Programmers

Unit Testing :: The test done by programmer on own piece of code is called unit testing.. Peer Testing:: The unit testing done on 1 programmer's code /task by his coleague programmer is called Peer testing

note:: testing matching expected results with actual results.

if matched then test result is positive (Test succedded)

it not matched then test result is -ve

(Test failed)

Development -->Unit/peer Testing should be done continuosly by programmer until

=>The task given to programmer is called Unit, so this programmer testing is called Unit testing

all test results are positive.

BA (Business Analyst)

by

testings done developers

=> unit testing

=> Integration testing

(Under monitoring of TL)

(talks about modules/

apps integration)

=> peer testing

Design Documents/ User stories

coding

Dev

Commits

Unit/Peer

Code Repository GIT/SVN

Testing

(when test

results are positive)

TL:: Team Leader

PL:: Project Leader

PM ::Project Manager

(QA Team) by

the

Testings done Testers (They test whole project)

=> Performence Test /Load Test

=> Navigation Test

=> User-Experience Test

=> System testing

=> Load Testing

=> Functional Testing

=> sanity Testing

and etc..

Before Developers committing the code to Code Repository we need they need to complete unit testing and peer testing on the code in

all permutations and combinations

Unit testing can be done in two ways

a) Manual Unit Testing

b) Automated Unit Testing

Limitations of Manual Unit Testing

=> No Productivity (takes time)

=> Writing test report manually is complex process (Excel sheet report)

=> Presenting Test plans/ test cases to TL or superior is complex process

=> Test Regression(repeating the tests) is very complex.

=> It is not indurstry standard..

of

To overcome these problems, take the support Unit testing automation tools like JUnit, HttpUnit(for web applicaitons), mockito, TestNG and etc..

tools

for HttpUnit, Mockito and TestNG Junit is the base tool. (java based Unit Testing tools)

In Test results we can see

PhpUnit (for php code)

with

a) success :: expected results are matched actual results

b) failures :: expected results are not matched actual results

c) errors ::

Unanticipated/unexpected exception has come while testing the code..

What is the differnce b/w failure and error?

with

failure :: actual code has given result.. but not mathing expected result. error :: actual code has not given

result..rather it has thrown exception..

While working with Junit we can see 3 main comps

==========

==========

a) Service class/Main class (Class to be tested)

(1 or more)

b) Test case class (The class that contains test methods) (1 or more)

c) TestSuite :: Allows to combine multiple test case classes to generate the test report (0 or 1 (optional)

estcase

if

note:: we can run each TestCase class manually.. to generate Test report.. But! want get test report of all the classes togather then take the support Test suite class..

=>Eclipse IDE gives built-in Support for Junit (i.e eclipse gives junit libraries as built-in libraries)

To add Junit Libraries eclipse Java Project

right click on project ---> buildpath --> cfg buidpath ---> libraries tab ---> classpath ---> add Libraries ----> Junt -->select Junit5.. (not a standard)

)

note:: Each test method in TestCase class represents one test plan or test case

note:: test plan /test case is a code /logic where the actual result will be matched with expetected result note: In Test case class, we can place 1 or more test case methods note: Taking TestSuite class in Junit Testing is optional, becoz we can run the test case classes directly

In maven archetype projects that

are created using 1.5 version we will

get junit 5 libraries as the default dependencies (Industry standard)

note:: It is recomaned to use Junit with Maven/Gradle Project becoz it gives the support to use built-in Decompiler to see the source code and other advantages..

category

Jnuit5 contains 3 runtime libraries

a) Junit Jupitor :: Junit5 libraries

b) Junit Vintage :: junit 3/4 libraries (for backword compitiblity)

c) Junit

Integrations :: To allow JUnit integration with TestNg, Mockito and etc..

Junit 5 architecture Diagram

your tests (Our Test cases calsses)

written

against

junit-4.12

discovers & runs

(junit 5) junit-jupiter-api

testng-api

matching tests

totally made-up example

junit-vintage-engine

junit-jupiter-engine

testng-engine

implements

Junit5 Jupitor api gives

a) Annotations

junit-platform-engine

discovers implementations

`via ServiceLoader`

orchestrates execution

junit-platform-launcher

This engine is reponsible

to load test case clases, to create object for them and to call test methods)

[Helps different IDEs, Tools like maven and etc..

to search and get Junit Platform Engine)

`IJ`

use

exclusively

we use them in the development of

b) Assertions API

Annotations

==========

etc

TestCase classes and test methods..

gives Assertions.assertXxx() methods (static methods) to match actual result. with expeected result and generatenfest report.

@Test.

@DisplayName....

@BeforeEach....

@AfterEach....

@BeforeAll....

JUnit 5

Assertion methods

fail

assertTrue

import org.junit.Assertion

(Normal package import statement)

To call method class name/method name is required Assertion.assertEqual(-,-);

import static org.junit.Assertion;

(static import feature of java 5

is used)

To call this method

assertEqual(-,-);

@AfterAll....

assertSame

@Tag.

These are static

@ParameterizedTest.

assertNull

methods of

@ValueSource

Assertions class.

@NullSource

@EmptySource

@NullAndEmptySource

@TestMethodOrder

@Order

and etc..

assertNotSame assertNotEquals assertNotNull assertFalse

assertEquals

assertArray Equals

assertAll

assertThrows

and etc..

Generally The Testcase class name starts or ends with Test word and all test methods geneally begings with "test" word (It is just recomandation / not a rule)

Example::

service class

BankService (main class)

|-->p float getBlance(-)

--> TestBankService or BankServiceTest (Test case class name)

|-->p float calcSimpleIntrest(-,-) b.methods

-->p v testcalcSimpleIntrest()

|->p v testGetBalnace()

we write multiple forms of these test methods

to test main method/service method in multiple angels with different varieties of inputs

note:: In TestCase classes, for each b.method/service method we need to write varitely of test methods not quanity of

test methods...

LoginApp Code (test methods/plans)

=========

->Possible test methods /test plans

success

or failure

ure

a) Test with Valid credentials

b) Test with Invalid crendentials

c) Test with No credentials

each test case each test plan = each @Test method

sum logic of two numbers

==========

possible test methods/plans

test with positives

error

test with negetives

success or

here

test with mixed values

failure will come

prefer writing good quantity of test method

having varieties of inputs in each test method

test with zeros

test with floating points

test with chars/strings (error)

First Example Application

=======

step1) create Maven Project in eclipse IDE as standalone Project by taking maven-archetype-quickstart as the Project areche type and chanage java version to 17

File --> maven project ---->next ---> select maven-archetype-quickstart ---> next --->

groupId :nit

artifactId :: JUnitTestProj1

package :: com.nt.service --->next -->finish.

to

open pom.xml and change java version 1/... (optional if we create the maven project using maven-archetype-quickstart 1.5)

<properties>

■

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<maven.compiler.source>1</maven.compiler.source>

<maven.compiler.target>1</maven.compiler.target>

</properties>

Right click on the Project --->maven ---> update the project.

step2) add junit5 jupitar in pom.xml as dependent.. by collecting from mvnrepository.com

in pom.xml under <dependencies> tag

====

==========

if the project is created by using maven-archetype-quickstart 1.5 then the junit5 libraries will come automatically as the default dependencies

<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-api -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-api</artifactId>

The LTS version of jdk are :: 8/11/17 LTS: Long Term Support

<version>5.7.0</version>

<scope>test</scope>

</dependency>

<!-- https://mvnrepository.com/artifact/org.junit.jupiter/junit-jupiter-params -->

<dependency>

<groupId>org.junit.jupiter</groupId>

<artifactId>junit-jupiter-params</artifactId>

<version>5.7.0</version>

<scope>test</scope>

</dependency>

step3) develop main class or service class in com.nt.service package of src/main/java folder

//Arthmetic.java

package com.nt.service;

```
public class Arithmetic {
}
public float sum(float x,float y) {
return x+y;
}
```

step4) develop Testcase class with with Test Methods in src/test/java folder having package com.nt.test.

src/main/java ---> to Place main soruce code src/test/java ---> To place unit testing code

(Test case classes/Test suite classes)

@Test:: to make the method of Testcase class as

the Test method

assertEquals()/assertNotEquals() ::

To check wheather expected result is equal or not with actual result

and to generate test report.

AssertThrows() :: To check expected exception has come or

not.

assertTimeout() :: To check wheather b.method execution is completeed in the specified time or not.

@BeforeEach :: To place common logic that shoud execute befor the each Test method execution.

```
@BeforeEach public void setUp() {
System.out.println("TestBankLoanService.setUp()");
service=new BankLoanService();
```

AppTest.java

==== ========

```
package com.nt.tests;
import static org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.Test;
import com.nt.service.Arithmetic;
public class AppTest
{
@Test
}
public void testWithPositives() {
// create Service class object
Arithmetic ar=new Arithmetic();
float expected=30.0f;
float actual=ar.sum(10.0f, 20.0f);
//perform testing
assertEquals(expected, actual);
```

```
}
@Test
public void testWithNegetives() {
// create Service class object
Arithmetic ar=new Arithmetic();
float expected=-30.0f;
float actual=ar.sum(-10.0f, -20.0f);
//perform testing
assertEquals(expected, actual);
}
@Test
public void testWithZeros() {
// create Service class object
Arithmetic ar=new Arithmetic();
float expected=0.0f;
float actual=ar.sum(0.0f, 0.0f);
//perform testing
assertEquals(expected, actual);
}
@Test
}
@AfterEach :: To place common logic that shoud execute after the
each Test method execution.
@AfterEach
public void clear() {
System.out.println("TestBankLoanService.clear()");
service=null;
}
```

To write common logic only for 1 time for all test methods.. then place in @BeforeAll method.. similarly place cleanup logic for all test methods in @AfterAll method.. These methods must be taken as static methods

```
@BeforeAll
public static void setUpOnce() {
System.out.println("TestBankLoanService.setUpOnce()");
service=new BankLoanService();
}
@AfterAll
public static void clearOnce() {
```

System.out.println("TestBankLoanService.clearOnce()");

service=null;

}

JUNIT 5 - LifeCycle

Call-back Annotation

@BeforeAll

@BeforeEach

@Test --- Testcase 1

@AfterEach

@BeforeEach

@Test

Testcase 2

@AfterEach

@AfterAll

Executes once before executing

all the test cases (@Test methods)

Executes before each

Test case

(@Test method)

executes after each

test case (@ Test method)

Executes once at the end of

the all the Test cases (for all @Test methods)

public void testWithMixedValues() {

// create Service class object

note: In junit 4, we use @Ignore to make one test method

not participating in the UnitTesting.. the same operation

be done using @Disabled annotation

Arithmetic ar=new Arithmetic();

float expected=-10.0f;

in junit 5

float actual-ar.sum(10.0f, -20.0f);

//perform testing

@Test

assertEquals(expected, actual);

@Disabled

public void testSumWithMixedValues() {

}

int val1=10;

@AfterAll and @BeforeAll methods are TestCase class level one tine executing

methods @BeforeEach and @AfterEach methods are each case level repeatedly executing blocks

as

if Testcase class is having 20 methods test methods

then

step5) Test the Test case class that is having test methods

Right click in AppTest.java ---> run as ---> JunitTest

Finished after 0.208 seconds

Runs: 4/4

* Errors: 0

AppTest [Runner: JUnit 5] (0.000 s)

testWithPositives() (0.000 s)

testWithMixedValues() (0.000 s)

testWithNegetives() (0.000 s)

testWithZeros() (0.000 s)

(or)

int val2=-20;

int expected=-10;

int actual=new ArthmeticOperations().sum(val1, val2); assertEquals(expected, actual);

}

Failures: 0

Failure Trace

Right click on the Project ----> run as --> maven test

[INFO] Results:

[INFO]

[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0

[INFO]

Instead of creating Service class object in every test method, it is better to create only for 1 time

in @BeforeAll method and we can uninitialize the same object using @AfterAll method

package com.nt.test;

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

import org.junit.jupiter.api.AfterAll;

import org.junit.jupiter.api.BeforeAll;

import org.junit.jupiter.api.Disabled;

import org.junit.jupiter.api.Test;

```
import com.nt.service.ArthmeticOperations;
public class TestArthmeticOperations {
private static ArthmeticOperations ar;
}
@BeforeAll
public static void setup() {
System.out.println("TestArthmeticOperations.setup()");
ar=new ArthmeticOperations();
}
@Test
public void testSum With Positives() {
System.out.println("TestArthmeticOperations.testSumWithPositives()");
int val1=10;
int val2=20;
int expected=30;
int actual=ar.sum(val1, val2);
assertEquals(expected, actual);
}
@Test
public void testSumWithNegatives() {
System.out.println("TestArthmeticOperations.testSumWithNegatives()");
int val1=-10;
int val2=-20;
int expected=-30;
int actual ar.sum(val1, val2);
assertEquals(expected, actual);
}
@Test
public void testSumWithZeros() {
System.out.println("TestArthmeticOperations.testSumWithZeros()");
int val1=0;
int val2=0;
int expected=0;
int actual=ar.sum(val1, val2);
assertEquals(expected, actual);
}
@Test
```

```
@Disabled
public void testSum With Mixed Values() {
System.out.println("TestArthmeticOperations.testSumWithMixedValues()");
int val1=10;
int val2=-20;
int expected=-10;
int actual=ar.sum(val1, val2);
assertEquals(expected, actual);
}
@AfterAll
public static void tearDown() {
System.out.println("TestArthmeticOperations.tearDown()");
ar=null;
}
```

Output on the console

```
<terminated> TestArthmeticOperations [JUnit] C:\Program Files\Java\jdk-1
TestArthmeticOperations.setup()
TestArthmeticOperations.testSumWithPositives()
TestArthmeticOperations.testSumWithNegative
zm
TestArthmeticOperations.testSumWithZeros()
MF
TestArthmeticOperations.tearDown()
BP
```

=>@BeforeAll, @AfterAll methods execute only for 1 time =>@BeForeEach, @AfterEach methods execute for 20 times on 1 per each test method basis