ACE Inspiration

Professional Development Center

JUNIT TUTORIRAL

JUnit

Contents

Chapter 1
Software Testing Introduction

1.1 What is software testing?

Software testing is a process of identifying the correctness of software by considering its all attributes (Reliability, Scalability, Portability, Re-usability, Usability) and evaluating the execution of software components to find the software bugs or errors or defects.

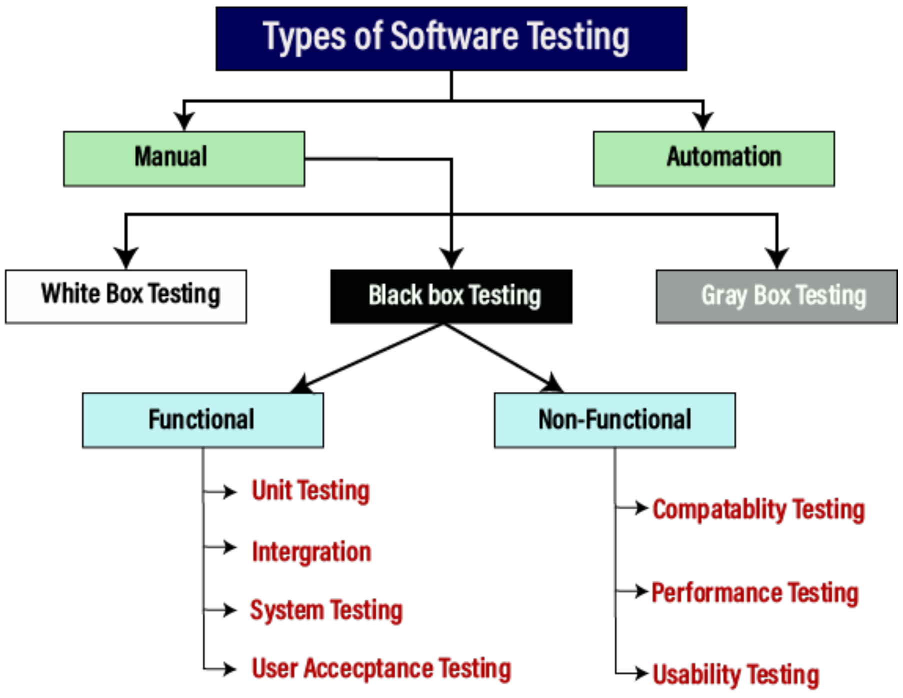
Software testing provides an independent view and objective of the software and gives surety of fitness of the software. It involves testing of all components under the required services to confirm that whether it is satisfying the specified requirements or not. The process is also providing the client with information about the quality of the software.

Testing is mandatory because it will be a dangerous situation if the software fails any of time due to lack of testing. So, without testing software cannot be deployed to the end user.

1.1 Types of software testing?

We have various types of testing available in the market, which are used to test the application or the software.

With the help of below image, we can easily understand the type of software testing:



Chapter 2
Unit testing introduction

2.1 What is unit testing?

A unit can be a function, a class, a package, or a subsystem. So, the term unit testing refers to the practice of testing such small units of your code, so as to ensure that they work as expected. For example, we can test whether an output is what we expected to see given some inputs or if a condition is true or false.

This practice helps developers to discover failures in their logic behind their code and improve the quality of their code. Also, unit testing can be used so as to ensure that the code will work as expected in case of future changes.

2.2 Test coverage

In general, the development community has different opinion regarding the percentage of code that should be tested (test cover-age). Some developers believe that the code should have 100% test coverage, while others are comprised with a test coverage of 50% or less. In any case, you should write tests for complex or critical parts of your code.

2.3 Unit testing in Java

The most popular testing framework in Java is JUnit. As this guide is focused to JUnit, more details for this testing framework will presented in the next sections. Another popular testing framework in Java is TestNG.

Chapter 3
JUnit introduction

JUnit is an open source testing framework which is used to write and run repeatable automated tests, so that we can be ensured that our code works as expected. JUnit is widely used in industry and can be used as stand alone Java program (from the command line) or within an IDE such as Eclipse.

Features of JUnit

- JUnit is an open source framework, which is used for writing and running tests.
- Provides annotations to identify test methods.
- Provides assertions for testing expected results.
- Provides test runners for running tests.
- JUnit tests allow you to write codes faster, which increases quality.
- JUnit is elegantly simple. It is less complex and takes less time.
- JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
- JUnit tests can be organized into test suites containing test cases and even other test suites.
- JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

3.1 JUnit Simple Example using Eclipse

In this section we will see a simple JUnit example. First we will present the class we would like to test:

Example 1

Calculate.java

```
package junittest;

public class Calculate {
     public int sum(int var1, int var2) {
          System.out.println("Adding values: " + var1 + " + "
+ var2); return var1 + var2;
          }

}
```

CalculateTest.java

```
package junittest.test;

import static org.junit.Assert.assertEquals;

import org.junit.*;

import junittest.Calculate;

public class CalculateTest {
	Calculate calculation = new Calculate();
	int sum = calculation.sum(2, 5);
	int testSum = 7;

	@Test
	public void testSum() {
		System.out.println("@Test sum(): " + sum + " = " + 
testSum);
		assertEquals(testSum, sum);
	}
}
```

Example 2

MessageUtil.java

```
package junittest;

public class MessageUtil {
	private String message;

	   //Constructor
	   //@param message to be printed

	   public MessageUtil(String message){
	      this.message = message;
	   }

	   // prints the message
	   public String printMessage(){
	      System.out.println(message);
	      return message;
	   }
}
```

MessageUtilTest.java

```
package junittest.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import junittest.MessageUtil;

public class MessageUtilTest {
     String message = "Hello World";
        MessageUtil messageUtil = new MessageUtil(message);

        @Test
        public void testPrintMessage() {
           assertEquals(message,messageUtil.printMessage());
        }

}
```

Example 3

CalculateMaximum.java

```
package junittest;

public class CalculateMaximum {
       public static int findMax(int arr[]){
             int max=0;
             for(int i=1;i<arr.length;i++){
                 if(max<arr[i])
                     max=arr[i];
             }
             return max;
         }
}
```

CalculateMaximumTest.java

```
package junittest.test;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

import junittest.CalculateMaximum;
```

```
public class CalculateMaximumTest {
     @Test
    public void testFindMax(){
        assertEquals(4,CalculateMaximum.findMax(new
int[]{1,3,4,2}));
        assertEquals(-1,CalculateMaximum.findMax(new int[]{-
12,-1,-3,-4,-2}));
    }
}
```

Let's see the output displayed in eclipse IDE.

As you can see, when we pass the negative values, it throws AssertionError because second time findMax() method returns 0 instead of -1. It means our program logic is incorrect.

!!! Write correct programming logic.

3.2 JUnit Execution Procedure

In this section the execution procedure of methods in JUnit, which defines the order of the methods called. Discussed below is the execution procedure of the JUnit test API methods with example.

ExecutionProcedureJunit
.java

```
package junittest.test;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class ExecutionProcedureJunit {

   //execute only once, in the starting
   @BeforeClass
   public static void beforeClass() {
      System.out.println("in before class");
   }

   //execute only once, in the end
   @AfterClass
   public static void  afterClass() {
      System.out.println("in after class");
   }
   //execute for each test, before executing test
   @Before
   public void before() {
      System.out.println("in before");
   }

   //execute for each test, after executing test
   @After
   public void after() {
      System.out.println("in after");
   }

   //test case 1
   @Test
   public void testCase1() {
      System.out.println("in test case 1");
   }

   //test case 2
   @Test
   public void testCase2() {
      System.out.println("in test case 2");
   }
}
```

Console Result

```
in before class
in before
in test case 1
in after
in before
in test case 2
in after
in after class
```

3.3 JUnit Assertions

In this section we will present a number of assertion methods. All those methods are provided by the Assert class which extends the class java.lang.Object and they are useful for writing tests so as to detect failures. In the table below there is a more detailed explanation of the most commonly used assertion methods.

Sr.No.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Checks that two primitives/objects are equal.
2	**void assertTrue(boolean condition)** Checks that a condition is true.
3	**void assertFalse(boolean condition)** Checks that a condition is false.
4	**void assertNotNull(Object object)** Checks that an object isn't null.
5	**void assertNull(Object object)** Checks that an object is null.
6	**void assertSame(object1, object2)** The assertSame() method tests if two object references point to the same object.
7	**void assertNotSame(object1, object2)** The assertNotSame() method tests if two object references do not point to the same object.
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.

AssertionsTest.java

```java
package junittest.test;

import org.junit.Test;
import static org.junit.Assert.*;

public class AssertionsTest {
     @Test
        public void testAssertions() {
           //test data
           String str1 = new String ("abc");
           String str2 = new String ("abc");
           String str3 = null;
           String str4 = "abc";
           String str5 = "abc";

           int val1 = 5;
           int val2 = 6;

           String[] expectedArray = {"one", "two", "three"};
           String[] resultArray =  {"one", "two", "three"};

           //Check that two objects are equal
           assertEquals(str1, str2);

           //Check that a condition is true
           assertTrue (val1 < val2);

           //Check that a condition is false
           assertFalse(val1 > val2);

           //Check that an object isn't null
           assertNotNull(str1);

           //Check that an object is null
           assertNull(str3);

           //Check if two object references point to the same
object
           assertSame(str4,str5);

           //Check if two object references not point to the
same object
           assertNotSame(str1,str3);

           //Check whether two arrays are equal to each other.
           assertArrayEquals(expectedArray, resultArray);
        }
}
```

3.4 Using @Ignore annotation

FirstDayAtSchool.java

```
package junittest;

import java.util.Arrays;

public class FirstDayAtSchool {
     public String[] prepareMyBag() {
          String[] schoolbag = { "Books", "Notebooks",
"Pens" };
          System.out.println("My school bag contains: " +
Arrays.toString(schoolbag));
          return schoolbag;
     }

     public String[] addPencils() {
          String[] schoolbag = { "Books", "Notebooks", "Pens",
"Pencils" };
          System.out.println("Now my school bag contains: " +
Arrays.toString(schoolbag));
          return schoolbag;
     }
}
```

FirstDayAtShoolTest.java

```
package junittest.test;

import static org.junit.Assert.assertArrayEquals;

import org.junit.Test;

import junittest.FirstDayAtSchool;

public class FirstDayAtSchoolTest {
     FirstDayAtSchool school = new FirstDayAtSchool();
     String[] bag1 = { "Books", "Notebooks", "Pens" };
     String[] bag2 = { "Books", "Notebooks", "Pens",
"Pencils" };

     @Test
     public void testPrepareMyBag() {
          System.out.println("Inside testPrepareMyBag()");
          assertArrayEquals(bag1, school.prepareMyBag());
```

```
	}

	@Test
	public void testAddPencils() {
		System.out.println("Inside testAddPencils()");
		assertArrayEquals(bag2, school.addPencils());
	}
}
```

Now we can run the test case by right-clicking on the test class and select Run As ! JUnit Test. The program output will look like that:

```
Inside testPrepareMyBag()
My school bag contains: [Books, Notebooks, Pens]
Inside testAddPencils()
Now my school bag contains: [Books, Notebooks, Pens, Pencils
```

and in the JUnit view will be no failures or erros. If we change one of the arrays, so that it contains more than the expected elements:

```
String[] bag2 = { "Books", "Notebooks", "Pens", "Pencils", "Rulers"};
```
and we run again the test class, the JUnit view will contain a failure:

Else, if we change again one of the arrays, so that it contains a different element than the expected:

```
String[] bag1 = { "Books", "Notebooks", "Rulers" };
```
and we run again the test class, the JUnit view will contain once again a failure:

Let's see in the above example how can we use the @Ignore annotation. In the test class FirstDayAtSchoolTest we will add the @Ignore annotation to the testAddPencils() method. In that way, we expect that this testing method will be ignored and won't be executed.

```
	@Ignore
	@Test
	public void testAddPencils() {
		System.out.println("Inside testAddPencils()");
		assertArrayEquals(bag2, school.addPencils());

	}
```

Now, we will remove the @Ignore annotation from the testAddPencils() method and we will annotate the whole class instead.

3.5 Using Expected Test

JUnit provides an option of tracing the exception handling of code. You can test whether the code throws a desired exception or not. The **expected** parameter is used along with @Test annotation. Let us see @Test(expected) in action.

Expected.java

```
package junittest;

public class Expected {

        public void divided(){
           int a = 0;
           int b = 1/a;
        }
}
```

ExpectedTest.java

```
package junittest.test;

import org.junit.Test;

import junittest.Expected;

public class ExpectedTest {
     Expected expected=new Expected();
      @Test(expected = ArithmeticException.class)
        public void testDivided() {
           expected.divided();
        }
}
```

3.6 Using Parameterized Test

JUnit 4 has introduced a new feature called **parameterized tests**. Parameterized tests allow a developer to run the same test over and over again using different values. There are five steps that you need to follow to create a parameterized test.

- Annotate test class with @RunWith(Parameterized.class).
- Create a public static method annotated with @Parameters that returns a Collection of Objects (as Array) as test data set.
- Create a public constructor that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Create your test case(s) using the instance variables as the source of the test data.

The test case will be invoked once for each row of data. Let us see parameterized tests in action.

Example 1

CalculateSum.java

```
package junittest;

public class CalculateSum {
     public int sum(int first,int second) {
          return first+second;
     }
}
```

CalculateSumTest.java

```
package junittest.test;


import static org.junit.Assert.assertEquals;

import java.util.Arrays;

import java.util.Collection;

import org.junit.Test;

import org.junit.runner.RunWith;

import org.junit.runners.Parameterized;

import org.junit.runners.Parameterized.Parameters;
import junittest.CalculateSum;

@RunWith(Parameterized.class)
public class CalculateSumTest {
        private int expected;
        private int first;
        private int second;

        public CalculateSumTest(int expectedResult, int firstNumber, int
secondNumber) {
                this.expected = expectedResult;
                this.first = firstNumber;
                this.second = secondNumber;
        }

        @Parameters
        public static Collection addedNumbers() {
                return Arrays.asList(new Integer[][] {
                        { 3, 1, 2 },
                        { 5, 2, 3 },
                        { 7, 3, 4 },
                        { 9, 4, 5 }, });
        }

        @Test
        public void sum() {
                CalculateSum add = new CalculateSum();
                System.out.println("Addition with parameters : " + first + " and " +
second);
                assertEquals(expected, add.sum(first, second));
        }
}
```

Example 2

PrimeNumberChecker.java

```java
package junittest;

public class PrimeNumberChecker {
     public Boolean validate(int primeNumber) {
           for (int i = 2; i < (primeNumber / 2); i++) {
              if (primeNumber % i == 0) {
                 return false;
              }
           }
           return true;
        }
}
```

PrimeNumberCheckerTest.java

```java
package junittest.test;

import java.util.Arrays;
import java.util.Collection;
import org.junit.Test;
import org.junit.Before;
import org.junit.runners.Parameterized;
import junittest.PrimeNumberChecker;
import org.junit.runner.RunWith;
import static org.junit.Assert.assertEquals;

@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
     private int inputNumber;
     private boolean expectedResult;
     private PrimeNumberChecker primeNumberChecker;

     @Before
     public void initialize() {
          primeNumberChecker = new PrimeNumberChecker();
     }

     // Each parameter should be placed as an argument here
     // Every time runner triggers, it will pass the arguments
     // from parameters we defined in primeNumbers() method

     public PrimeNumberCheckerTest(int inputNumber, boolean
expectedResult) {
          this.inputNumber = inputNumber;
```

```
            this.expectedResult = expectedResult;
        }

        @Parameterized.Parameters
        public static Collection primeNumbers() {
            return Arrays.asList(new Object[][] {
                { 2, true },
                { 6, false },
                { 19, true },
                { 22, false },
                { 23, true } });
        }

        // This test will run 4 times since we have 5 parameters
defined
        @Test
        public void testPrimeNumberChecker() {
            System.out.println("Parameterized Number is : " +
inputNumber);
            assertEquals(expectedResult,
primeNumberChecker.validate(inputNumber));
        }
}
```

3.7 Using Suit Test

Test suite is used to bundle a few unit test cases and run them together. In JUnit, both **@RunWith** and **@Suite** annotations are used to run the suite tests. This chapter takes an example having two test classes, **CalculateTest** & **CalculateMaximumTest**, that run together using Test Suite.

SuiteTest.java

```
package junittest.test;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
   CalculateTest.class,
   CalculateMaximumTest.class
})

public class SuiteTest {

}
```