A Comprehensive Study on Automation Testing using JUnit

Rakshith D C[1], Dr. Manjunath A E[2]

[1]Department of CSE, RVCE, Bengaluru, Karnataka, India
[2]Assistant Professor, Department of CSE, RVCE, Bengaluru, Karnataka, India

---***---

Abstract - *Testing is an important process of the software development life cycle. Testing can be either manual or automated. As the continuous development of many applications increase, there must be tests to run the developed components or to change the existing components to verify the given component is able to function correctly without any errors. Unit testing plays an important role to determine whether the developed code is fit for the use. A unit test is the smallest testing part of an application. In basic learning of unit testing, JUnit framework with interactive GUI techniques is suitable. In this paper, it presents the method to create unit testing code and how JUnit can give good experience in unit testing in software engineering. The study on automation testing using JUnit framework shows the advantages in unit testing and the features provided by the framework. This paper also provides the comparative study of JUnit 4 and JUnit 5.*

***Key Words*: JUnit, Unit testing, Automation framework, testing tools, Java**

.

1. INTRODUCTION

A software error is basically a bug in the programming of an application that is developed. The bugs incur huge costs, money, time and patience which may damage the product. Testing is a main process of software development life cycle. Automation testing saves a lot of time and money, also it increases the test coverage and improves accuracy. Creating and continuously executing test cases for the software to address the bugs is a standard and practical approach. Test cases are written by software quality engineers to make sure the code fits the design and runs as expected. A test case that is carried out to ensure a specific test, is called unit testing. This is the simplest form of testing.

In programming language, unit testing is method by which each unit of source code is tested to check whether the code is fit for use. In object-oriented programming, it is a method. In procedural programming, a unit may be an individual procedure or function. Figure 1 shows the Simplex process to perform Unit tests[1].

The JUnit testing is widely used for Java development that is having object-oriented framework. JUnit is also used for unit testing extensively in the case of Application Programming Interface(API). On comparing the JUnit framework with Sahi and Fitnesse[2], it is more simplified and easy to use. In JUnit, either one class can be tested as a single test case or a group of classes can be clubbed together into a suite for the purpose of integrated testing.

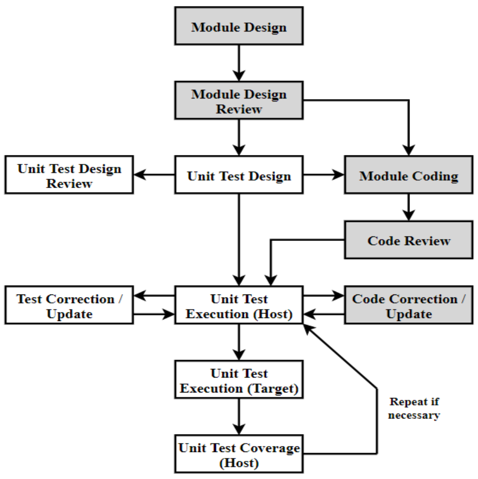


Figure 1: Simplex process to perform Unit tests (The highlighted boxes display the activities which must be performed in concurrent or finished prior to Unit Testing)

The test results will be provided accordingly for each class in the suite making easy to identify the failure test cases. Another major use of JUnit is to be the part of the build process in the development of the software product where it can execute the conglomeration of several test cases which verify functionality of every feature of the product during the development phases. The results of these test cases are captured for generating the colorful reports consisting of passed, failed or skipped test cases. The failure and erroneous test cases are distinguished by the JUnit framework. In this paper, the details about JUnit is extensively discussed and the features of the framework and the creation of test cases is shown.

2. JUNIT FRAMEWORK

To write repetitive automated test-cases for regression testing of the software product, JUnit will be the simplex framework for testing Java development code. It is open source and was developed by Kent Beck and Erich Gamma and hosted on Github. It is test driven framework and it uses annotations for identifying tests, which are written as methods. It is a part of xUnit, Unit testing family. The current version of JUnit is 5. The JUnit features include:

i. Textual and Graphical test runners
ii. To Test suites for quickly organize and run tests
iii. Testing expected results using assertions
iv. Sharing common test data using test fixtures

JUnit has seen rapid and wide-spread adoption since its inception and it is a simple framework for writing and running automated tests. As a political signal, it celebrates programmers testing their own software. A popular use for JUnit is to create a set of unit tests which can be run automatically when software is modified. The quality engineers ensure that the software is functioning properly as expected after every code change using these automated test cases. JUnit also renders a test runner that is capable to run the unit tests and cover on the failure or success of the tests.

The annotations which are used to write test cases are shown in the below table 1.

JUnit Annotations	Description of the JUnit annotation
@AfterEach	Gives the function that is to be executed after the "test" annotation in the current class; these methods are inherited in the next class.
@BeforeEach	Gives the annotation method that has to executed before the "test" annotation in the present class; these methods are inherited.
@DisplayName	Gives the customized name for the class that is to be tested. These annotations are not inherited.
@TestInstance	Used to configure the test instance lifecycle for the annotated test class. Such annotations are inherited.
@TestMethodOrder	Used to configure the test method execution order for the annotated test class; similar to JUnit 4's @FixMethodOrder. Such annotations are inherited.
@TestTemplate	Denotes that a method is a template for test cases designed to be invoked multiple times depending on the number of invocation contexts returned by the registered providers. Such method are inherited unless they are overridden.
@DisplayNameGeneration	Gives the customized name for the class that is to be tested.
@TestFactory	Used to denote a function in the dynamic tests is a factory of tests. There are inherited.
@RepeatedTest	Used to denote a method is a template in the test for which is repeated during the run. These are the methods that are inherited only if they not are overridden.
@ParameterizedTest	Used to denote a function that has parameters in the test is a parameterized test. These methods are inherited.
@Test	Used to denote that a function is a test function. This annotation is not lie JUnit's version 4. it does not declare any parameters since the extensions in the test in JUnit Eclipse operate based on their own devoted annotations.

2.1 Fundamental Operations

The class diagram from the Java package JUnit framework of the core framework is illustrated in the figure 2. This paper doe not consider other packages like extensions and runners[3].

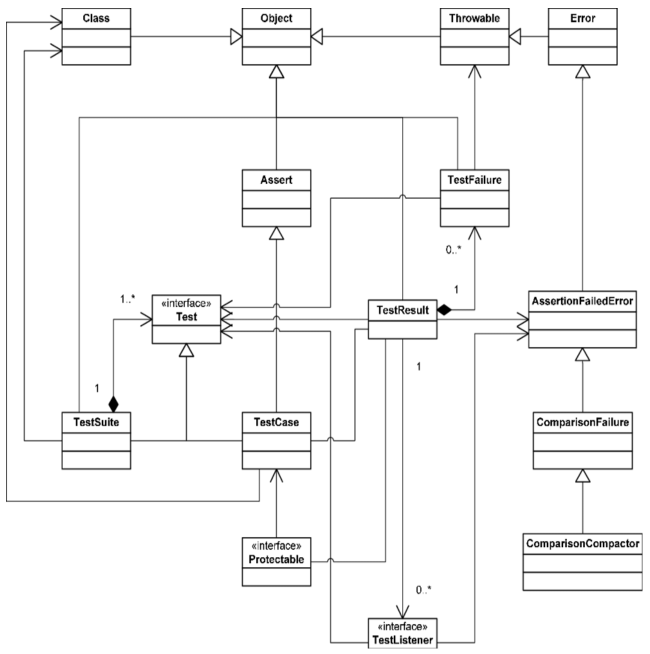


Figure 2: JUnit framework classe diagram

Test case configuration and test case execution are the two major phases in the complete run of the JUnit test cases. Further, the results and visual analysis of the passed and failed test cases are presented to the user. The test case's object hierarchy is built during the configuration phase in which each object represents a use-case. These sub-classes are implemented by programmers or developers and every method in the sub-class has to be annotated with the word "Test" which creates the object hierarchy. The application specific and the domain specific code is written inside the the "Test" methods. The test case sub-class form the leaf node whereas the instance of the test suite form the intermediate nodes and root node which basically provides the grouping functionality required for the tree structure. Test Interface is implemented by both test suite and test case to treat each node in the tree homogeneously[3].

Root object have the highest priority in the test hierarchy where the actual execution of the test starts upon calling it. Depth first traversal is used to traverse the tree and the node's order in which they were added into the tree are used for calculating the order of the nodes. Test Result Object which is a collecting parameter object records every test case execution results.

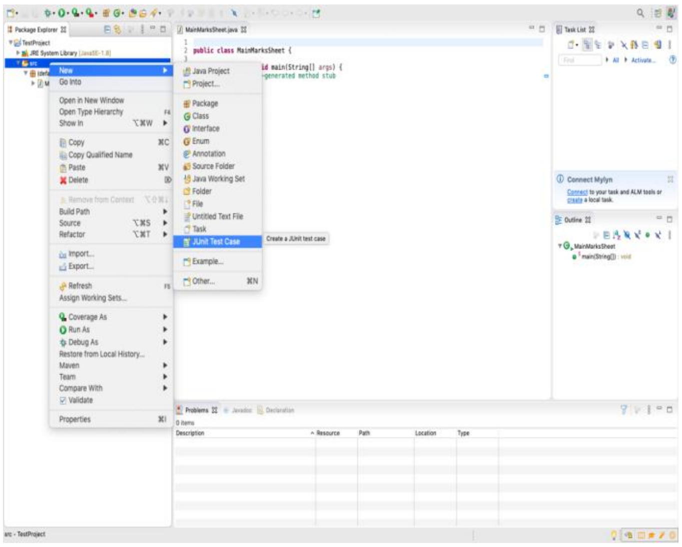
Errors and failures are differentiated by JUnit. A failure is when one of the assertions fail, that is the program written by the user does something wrong and JUnit reports that error. An error is something that occurs when there is an exception and that is not the test done by the user and did not expect that error such as, NullPointerException or ArrayIndexOutOfBoundsException.

2.2 Using JUnit to design Unit Testing

The followings steps and guidelines are used while running Unit test cases to make it more effective and easy. The primary step is to determine the way to test the mehtod in the manual scenario before designing the code. Once the manual scenario is understood, the implementation code is written simultaneously. The secondary step is to tun all the tests by grouping into suite which can performed quickly after successfully executing the unit test cases[4].

In order to perform this, the user have to create JUnit class by using any IDE in a standard format. Here, in this paper Eclipse IDE has been considered to build up the example. In Eclipse IDE, right click on the package in the project explorer area and click New, further selecting JUnit test case as illustrated in the below Figure 3. Every test class name will be followed with the word "Test" by convention. The passed or failure status is determined for every test case.

To verify the correct value is returned after the execution of the test case, assertions are ran to the methods that will be tested inside the unit testing. JUnit provides various types of assertions such as: assertFalse(Condition), assertTrue(Condition) and assertEquals(expected, actual). These assertions aids in comparing the actual and expected values and awaits for the pass condition to return the status false/true. The failed test case are displayed in red and the passed test cases are displayed in green in the JUnit view.



```
public class MainMarksSheet {
id main(String[] args) {
-generated method stub
```

Figure 3: Creation of JUnit test case(Eclipse menu options)

3. JUNIT EXPERIMENT

The following example provides information to write and execute the test cases in JUnit framework using Eclipse IDE. JUnit 5 test cases and test suites can be quickly and easily created with the help of the Eclipse IDE[5]. Creation of JUnit tests and test suites are shown in this experiment for a simple java class library project. The creation of tests using annotations is shown in the first part of the experiment. The next part of the example provides information to modify the existing test case created by annotation and to modify the output messages for making them functioning.

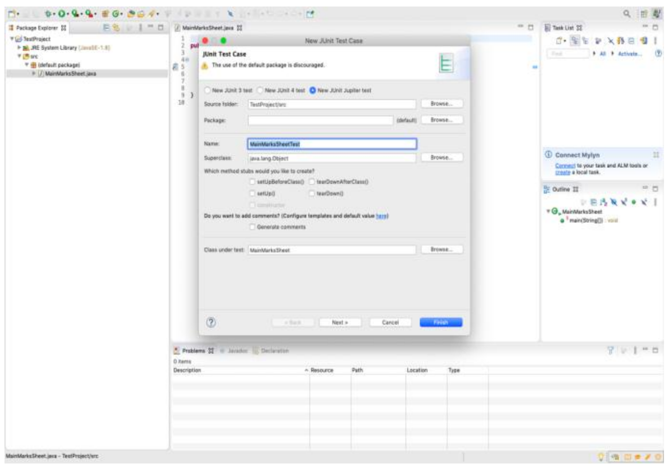
3.1 To write JUnit tests in Eclipse IDE

Eclipse IDE is used to create basic skeleton of the JUnit test class. Then it will be modified to add required test methods. Here in this step, a JUnit test case is created for the class MainMarksSheet.java.

3.2 To write JUnit tests in Eclipse IDE

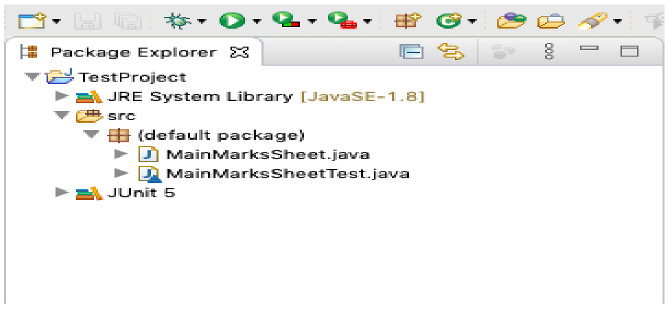
To create a test class for a Java method, right click on the class MainMarksSheet.java and create JUnit test in the project explorer area(left window of the Ecplise IDE is the project explorer). A prompt will be generated for selecting the JUnit version by the IDE. Then the JUnit version is selected according to project, currently there are three versions of the JUnit as illustrated in the Figure 4(a).

By convention, the test class is named after the Java class name that is created to perform testing. The word "test" is followed by the Java class name for naming the test class. Here in this example, MainMarksSheetTest.java will be the test class and it will be created under the test section packages as shown in the figure 4(b).



```
New JUnit Test Case
JUnit Test Case
The use of the default package is discouraged.
( ) New JUnit 3 test  ( ) New JUnit 4 test  (•) New JUnit Jupiter test
Source folder: TestProject/src                          Browse...
Package:                                     (default)  Browse...
Name: MainMarksSheetTest
Superclass: java.lang.Object                            Browse...
Which method stubs would you like to create?
  [ ] setUpBeforeClass()  [ ] tearDownAfterClass()
  [ ] setUp()             [ ] tearDown()
  [ ] constructor
Do you want to add comments? (Configure templates and default value here)
  [ ] Generate comments
Class under test: MainMarksSheet                        Browse...
? < Back  Next >  Cancel  Finish
```

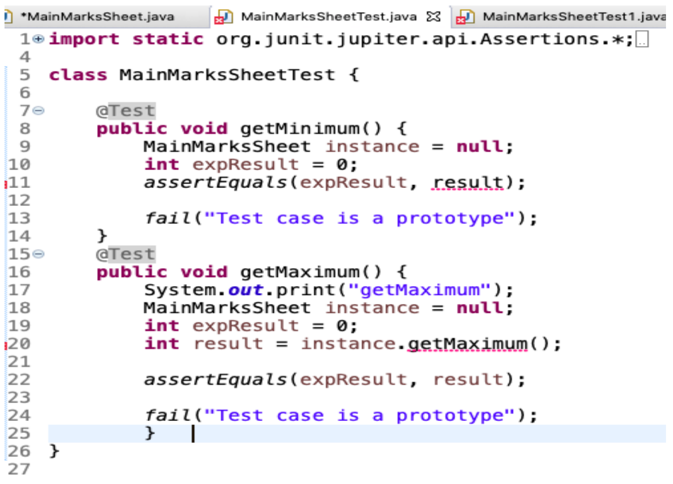
Figure 4(a): Dialog box for create test



```
Package Explorer
▼ TestProject
  ► JRE System Library [JavaSE-1.8]
  ▼ src
    ▼ (default package)
      ► MainMarksSheet.java
      ► MainMarksSheetTest.java
  ► JUnit 5
```

Figure 4(b): Project Window

In MainMarksSheetTest.java, each test method must be annotated with @Test. The IDE generated the names for the test methods based on the names of the method in MainMarksSheet.java. The default body of each generated test method is provided only as a guide and needs to be modified to be actual test cases (Figure 5).

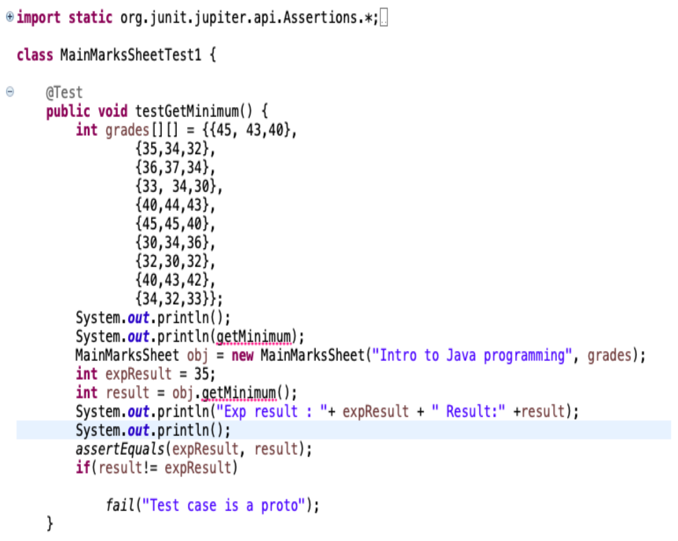


```
*MainMarksSheet.java   MainMarksSheetTest.java   MainMarksSheetTest1.java
import static org.junit.jupiter.api.Assertions.*;

class MainMarksSheetTest {

    @Test
    public void getMinimum() {
        MainMarksSheet instance = null;
        int expResult = 0;
        assertEquals(expResult, result);

        fail("Test case is a prototype");
    }
    @Test
    public void getMaximum() {
        System.out.print("getMaximum");
        MainMarksSheet instance = null;
        int expResult = 0;
        int result = instance.getMaximum();

        assertEquals(expResult, result);

        fail("Test case is a prototype");
        }
}
```

Figure 5: MainMarksSheetTest.java window

3.3 To write the Test Methods for Marks Sheet Test

The code enhancement and modification is done for the test methods that are generated for the above test methods in order fit the data of the test case. The JUnit assertion method is used for making comparison between real and the expected result. Here in this example the grades of 10 students in 3 subjects is fit into the array as the test data. The test method getsMinimum is renamed to testGetMinimum. To get a proper output view appropriate data for the test is provided in the array as illustrated in the Figure 6. The MainMarksSheet.java is called by feeding this data to it. Also, some println statements are used to improve the output view. In the below example, assertEquals method is used in the test method. In order to use this assertion, the expected value has to be provided in prior. The test method is considered to be passed only when the actual result is matched with the expected result. The below Figure 6 illustrates the test method.



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

class MainMarksSheetTest1 {

    @Test
    public void testGetMinimum() {
        int grades[][] = {{45, 43,40},
                {35,34,32},
                {36,37,34},
                {33, 34,30},
                {40,44,43},
                {45,45,40},
                {30,34,36},
                {32,30,32},
                {40,43,42},
                {34,32,33}};
        System.out.println();
        System.out.println(getMinimum);
        MainMarksSheet obj = new MainMarksSheet("Intro to Java programming", grades);
        int expResult = 35;
        int result = obj.getMinimum();
        System.out.println("Exp result : "+ expResult + " Result:" +result);
        System.out.println();
        assertEquals(expResult, result);
        if(result!= expResult)

            fail("Test case is a proto");
    }
```

Figure 6: MainMarksSheetTest.java window

3.4 To run the tests

In JUnit, either individual test case or the entire application can be run at once and the results can be viewed in the IDE. The execution of test case remains same as compared to the execution of the sample java program.

4. COMPARISON BETWEEN JUNIT4 AND JUNIT5

JUnit 5 aims to adapt Java 8 style of coding and to be more flexible and robust when compared to JUnit 4. In this section, the focus is on the comparison of JUnti 4 to JUnit 5, the changes in annotations used for test case creation, the addition of features in JUnit 5 which were not applicable to JUnit 4, assertions in JUnit 4 and 5 and test suites package information of JUnit 4 and 5.

4.1 Annotations

Most of the annotations used in JUnit 4 are same when compared to JUnit 5 but here are a few changes.

Feature	JUnit 5	JUnit 4
Tagging and filtering	@Tag	@Category
Nested tests	@Nested	NA
Test factory for dynamic tests	@TestFactory	NA
Register custom extensions	@ExtendWith	NA
Disable a test method/class	@Disabled	@Ignore
Execute after each test function	@AfterEach	@After
Execute before each test function	@BeforeEach	@Before
Execute after all test functions in the current class	@AfterAll	@AfterClass
Execute before all test functions in the current class	@BeforeAll	@BeforeClass
Test Method Declaration	@Test	@Test

4.2 Assertions

The resulted outcomes is matched with the expected results with the use of org.junit.Assert in JUnit 4. In order to display the error message, the string can be passed in the method signature as the first parameter.

Example: 1. public int assertEquals(int expected, int actual)
2. public int assertEquals(long message, Srting expected, String actual)
Comparatively, JUnit 5 consists all the assertions of the JUnit 4 along with assertAll() and assertThrows() methods. Some of the assertions in JUnit 5 are still in the experimental phase. The error message will be printed when test case fails by overloading the assertion methods in JUnit 5.
Example: 1. public int assertEquals(int expected, int actual)
2. public int assertEquals(long expected, int actual, Supplier messageSupplier)
3. public int assertEquals(int expected, int actual, String message)

4.3 Test Suites

Test suite is a group of multiple test cases which has to be executed at once. The below table provides the annotations to create test suite in Junit 5 and Junit 4.

JUnit 5	JUnit 4
@RunWith, @SelectPackages and @SelectClasses	@RunWith and @Suite

4.4 Assumptions

To state the assumptions in JUnit 4 there are some pre-defined methods in org.junit.assume. The 5 methods in org.junit.assume are listed below:
1. assumeTrue()
2. assumeThat()
3. assumeNotNull()
4. assumeNoException()
5. assumeFalse()
To state the assumptions in JUnit 5 there are few pre-defined methods in org.junit.jupiter.api.Assumptions. The below are the methods:
1. assumeTrue()
2. assumingThat()
3. assumeFalse()

5. CONCLUSIONS

JUnit is an open source framework for Java development. As per the above study, it is the most popular and widely used framework for creation and writing of test cases. It provides an alternative to the automation testing and it is simple to use. The automatic generation of test case and execution of the tests provides significant advantages for those who lack formal knowledge in unit testing. JUnit allows users to code and also to test during the development process. It provides a graphical user interface(GUI) which makes it possible to write and test the source code more quickly and efficiently.

Regarding complex and deep unit testing, users must use different tools that render more features for text editing and hence the errors can be eliminated. It is difficult to understand the IDE errors that is generated for those who lack technical knowledge and hence not suitable for customer-driven testing.

REFERENCES

[1] Michael Ellims & James Bridges & Darrel C. "The Economics of Unit Testing" Ince, Empir Software Eng (2006) 11: 5-31, Springer Science + Business Media, Inc. 2006
[2] ADempiere ERP, March 2011: http://www.adempiere.com/ADempiere-ERP
[3] Dirk Riehle, "JUnit 3.8 Documented Using Collaborations" ACM SIGSOFT Software Engineering Notes Page 1, New York, NY, USA, March 2008 Volume 33
[4] Andy Hunt and Dave Thomas "Pragmatic Unit Testing in Java with JUnit", 2004 http://www.pragmaticprogrammer.com/starter-kit
[5] "Eclipse IDE 4.15: The Eclipse IDE for Java Developers" Eclipse, 2011 : https://www.eclipse.org/ide/