Volume 2, Issue 7, July 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Enhancing Component Based Testing Using JUnit Tool in Net Beans Environment

Ravinder Kumar
Student: *CSE Department*
U.I.E.T. kurukshetra university
Kurukshetra, India

Mr. Karambir Singh
Asst. Proff. in *CSE Department*
U.I.E.T. Kurukshetra University
Kurukshetra, India

Abstract— ***Component-based software engineering, software systems are mainly constructed based on reusable components, such as third-party components and built-in components. From this approach, system quality depends on the quality of the involved components. Any change in the component, it must be changes at the unit level. Unit testing is a common step in software development. There is variety of unit testing tools available today. The main purpose is to use of a variety of automatic unit test generation tools, so that they could produce and execute large no of test inputs that are under test. In this paper, we implemented a java calculator program on the Net beans platform and test its main components under JUnit4 testing tool. Because Net Beans platform has Inbuilt support for JUnit4 framework. It helps in reducing the cost and times of the software under test. JUnit4 testing tools help in creating robust and reliable software's. This paper also reports this tool and its features.***

***Keywords*—Net beans platform, unit testing, JUnit4 tool.**

I. INTRODUCTION

Any programmer knows that testing is essential in the development of software applications. Component based software systems are mainly constructed based on reusable components such as third party components and built in components.

JUnit is a unit testing framework, by this we mean that JUnit works by taking the smallest possible part of testable software (a unit), isolating this part from the rest of the software and in turn validating that it works as it is required to work. The testing procedure uses verification and validation methods which may differ from framework to framework but will all essentially test if a specific unit is fit for use. In JUnit the testable units will be the methods within a class and the validation methods are dependent on the JUnit framework. Some of the benefits gained when performing unit tests:

- Unit tests enable refactoring – When changes are made to the code within a unit, tests are readily available to check if the changes produce a fault. Units can be checked at all times to make sure that functionality is maintained.
- Unit tests allow collective ownership – The code is not owned by an individual, changes may be made by all relevant parties. This is because unit tests guard the validity of the code so that after changes are made the unit must be tested to assure that all functionality still remains.

Tao Xie and David Not Kin [6] presented the operational violation approach for unit-test generation and selection, a black-box approach without requiring a priori specifications. Their approach dynamically generated the operational abstractions from executions of the existing unit test suite. These operational abstractions guide test generation tools to generate tests to violate them. Their approaches select those generated tests violating operational abstractions for inspection. These selected tests exercise some new behavior that has not been exercised by the existing tests. They implemented this approach by integrating the use of Daikon (a dynamic invariant detection tool) and Para soft Jtest (a commercial Java unit testing tool), and conducted several experiments to assess the approach.

Gao *et* al. [8] focused on component API based changes and impacts, and proposed a systematic re-test method for software components based on a component API based test model. The proposed method has been implemented in a component test tool, known as COMP Test. It could be used to automatically identify component-based API changes and impacts, as well as reusable test cases in a component test suite. A systematic method is provided to support test change and impact analysis for a given component black-box test suite.

In this paper we implemented a java calculator program on Net Beans Rich client platform and tested it main components. For testing of these components we used JUnit testing tool in Net Beans environment. Net Beans provide inbuilt support for JUnit framework. Net Beans automatically generate JUnit test for java programming code because it provide inbuilt support for JUnit framework.

In this paper our first main purpose is to check the functionality of all the components that they are working properly. Secondly we want to check the time consumed by each JUnit test for each component and also the total time consumed by the entire component we have tested by JUnit tool.

The next section presents background information on the unit-test generations and tools used for unit testing. Section III describes the unit testing supporting tool and its features. Section IV describes the implementation work done. Then Section V conclusion and VI section future work.

II. RELATED WORK

In the past, many papers have been published to address different unit test generations and issues related to component based software engineering.

We are giving some examples of unit testing which uses the testing tools for different problems.

Stott's et al. [2] proposed a method for systematically creating complete and consistent test classes for JUnit Called JAX. The method is based on Guttags algebraic specification of abstract data types. They demonstrated an informal use of ADT semantics for guiding JUnit test method generation; the programmer uses no formal notation other than Java, and the procedure meshes with XP test-as-design principles. Preliminary experiments show that informal JAX-based testing finds more errors than an ad hoc form of JUnit testing. Motivation and background Regression testing has long been recognized as necessary for having confidence in the correctness of evolving software. Programmers generally did not practice thorough tool-supported regression testing, however, unless they work within a significant industrial framework. JUnit was developed to support the test first principle of the XP development process; it has had the side effect of bringing the benefits of regression testing to the average programmer, including independent developers and students. JUnit is small, free, easy to learn and use, and has obtained a large user base in the brief time since its introduction in the XP community. JUnit and its supporting documentation are available at http://www.junit.org. The basic JUnit testing methodology is simple and effective.

Xie et al. [7] proposed Rostra, a framework for detecting redundant unit tests, and presented five fully automatic techniques within this framework. They used Rostra to assess and minimize test suites generated by test-generation tools. They also presented how Rostra could be added to these tools to avoid generation of redundant tests. They have implemented the five Rostra techniques and evaluated them on 11 subjects taken from a variety of sources. The experimental results show that Jtest and JCrasher generate a high percentage of redundant tests and that Rostra can remove these redundant tests without decreasing the quality of test suites.

There we purposes a unit testing technique for component based software in Net Beans environment. This paper focusing on testing of components using JUnit4 testing tool under Net Beans platform.

III. SUPPORTING TOOLS AND ITS FEATURES

Net Beans 7.0.1 refers to both a platform framework for java desktop environment (IDE) for developing with Java, Java script, PHP, C++, Scala and others. The Net Beans IDE is written in Java and can run anywhere a compatible JVM is installed. The Net Beans platform allows application to be developed from a set of modular software components called modules. Application based on the Net Beans platform (including the Net Beans IDE) can be extended by third party developers.

The growing popularity of test-driven development is partly due to the fact that powerful testing tools are available. These tools are known under the name *xUnit*, where x stands for a programming-language initial: JUnit is the tool for Java, SUnit for Smalltalk, CUnit for C, CppUnit for C++, NUnit for the .NET languages etc.

A common feature of this tool family is that testing code is separated from the actual implementation code. The tester writes testing code with xUnit annotations regarding the code unit to be tested. xUnit tools process the annotations and create a compact testing protocol which is outside the unit under testing. This is a definite advantage over conventional module testing where testing code is usually embedded in the actual implementation code.

Although JUnit is the most popular tool from the xUnit family, the origin of the tool family was a pattern and a framework written by Kent Beck for Smalltalk (SUnit). Ported to Java by Erich Gamma and Kent Beck, JUnit is an open-source framework to write and run repeatable unit tests.

Its features include:

– Assertions for testing expected results

– Test fixtures for sharing common test data

– Test runners for running tests

Beck and Gamma encourage the use of JUnit due to the obvious advantages of automation: The test runs as such can be automated; many tests can be run at the same time; and the test results can be interpreted automatically, without human interference. In their "JUnit cookbook", Beck and Gamma introduce JUnit in the following way [5]:

"JUnit tests does not require human judgment to interpret, and it is easy to run many of them at the same time. When you need to test something, here is what you do:

1. Annotate a method with @org.junit.Test.
2. When you want to check a value, import org.junit.Assert.*

statically, call assertEqual() and pass a boolean that is true if the test succeeds.

The JUnit testing process from writing a relevant test class to obtaining the final test results is quite simple. Below we can see how this process might work.

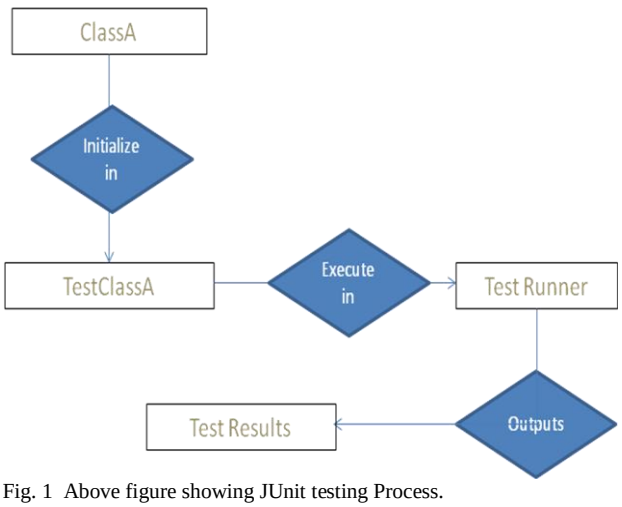

Fig. 1 Above figure showing JUnit testing Process.

Suppose we have a class called 'ClassA'and we want to test units within this class, in order to do so we must create a test class with our test cases, let's called this 'TestClassA'. TestClassA will import the relevent classes from the JUnit library and be compiled with the JUnit framework. After this we will use the JUnit tool runner provided by the framework to run the class and check for faults. The tool runner will check if a unit fails or passes the test, it will then output the results of the test showing how and where a test has failed. The developer must then check the failure and change ClassA where appropiate. After this the test class must be run with tool runner again to ensure that the changes made have enabled functionality to the unit.

IV. IMPLEMENTATIONS

Net Beans is Rich client platforms which provide an environment for writing java program. Net Beans has great feature that it can automatically generates java programming necessary code for the project under construction.

We implemented a calculator program in java on the Net Beans platform. First of all we have created a project name MyCalculator under package name JCalculator. After creating the project name we have created a class called Java Calculator, which is in the JCalculator package, which is in the MyCalculator project. When we open it in Net Beans environment a source code window opens. Then we create the necessary codes for all its components. After completion the code we run the program from the tool menu Run project. Net Beans automatically generated the result window which shows that the program is build successfully and also show the time in building the project. After building the project successfully we test it in JUnit4 tool to test the Main function whether it is working properly or not.

For creating the JUnit test by right clicking on the JavaCalculator class and click the tools and under tools choose create JUnit test. A wizard window will open in which we have to select JUnit3 or JUnit4. We select JUnit4 testing tool and Net Beans platform automatically generates the JUnit4 test codes for Java Calculator. After making some suitable changes in the test and Run the test it passes 100% successfully.

After testing the Main function, we are satisfied that the calculator program is working without any failures and passes the result 100%. Because green line shows the passing and red line means failure of test.

This is an example of JUnit4 test code for Main class javacalculator:

```
package jCalculator;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
/**
*
* @author ravi
*/
public class javaCalculatorTest {
public javaCalculatorTest() {
}
@BeforeClass
public static void setUpClass() throws Exception
{
}
@AfterClass
public static void tearDownClass() throws
Exception {
}
@Before
public void setUp() {
}
@After
public void tearDown() {
}
/**
* Test of main method, of class javaCalculator.
*/
@Test
public void testMain() {
System.out.println ("main");
String[] args = null;
javaCalculator.main (args);
}
}
```

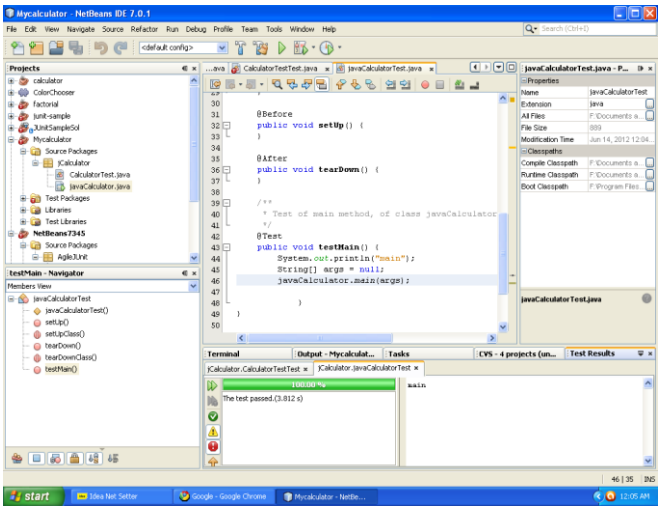

Fig.1 below is the window for successful of Main function under JUnit4 test.

After testing the main function, we want to test the components of calculator (e.g. multiply, divide, minus, plus) and test all these components that they are giving the output as we have expected from these components.

To test these components we create a Calculator Test class under Java Calculator class. When we open this calculator class in Net Beans a code window appears. We make some changes in the code to test the each component.

Writing test in JUnit has changed significantly as of JUnit4 compared to its predecessors. JUnit4 has never been more easy to use or more flexible and writing a test for smaller methods can be extremely simple. In addition the JUnit4 library offers a variety of methods that can be used for more complex applications that require finer tuning and more attention in detail.

The current release of JUnit relies on annotations for indentifying test methods as well as other methods that perform test related operations. These annotations are attached to the start of every relevant method that requires it. This differs from previous versions which required specific naming procedures of each method in order to identify the function it will perform.
Due to these annotations test cases are more organized and easier to understand, developers have more freedom in naming their tests and it's easier to implement collective ownership.
When we create a test class in Net Beans platform it imports all the relevant classes from the JUnit library that is required. For example importing 'org.junit.Test' or 'org.junit.*' enables the use of '@Test' before every test method. The same applies for methods used from the JUnit library such as assertEquals, this method takes a Boolean condition and asserts if it is equals to expected or false.

Here is an example of JUnit4 test code for calculator test class:

```
package jCalculator;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import static org.junit.Assert.*;
/**
 * @author ravi
 */
      public class CalculatorTestTest {
       public CalculatorTestTest() {
   }
   @BeforeClass
   public static void setUpClass() throws
Exception {
   }
   @AfterClass
   public static void tearDownClass() throws
Exception {
   }
   @Before
   public void setUp() {
   }
      @After
   public void tearDown() {
   }
   @Test
    public void testMultiply() {
      System.out.println ("Multiply");
      String s1 = "20";
      String s2 = "20";
      CalculatorTest instance = new
CalculatorTest ();
      double expResult = 400;
      }
   @Test
    public void testDivide() {
      System.out.println ("Divide");
      String s1 = "8";
      String s2 = "2";
      CalculatorTest instance = new
CalculatorTest ();
      double expResult = 4;
            }
   @Test
   public void testPlus() {
      System.out.println ("plus");
      String s1 = "20";
      String s2 = "25";
      CalculatorTest instance = new
CalculatorTest();
      double expResult = 45;
      double result = instance.plus (s1,s2);
         }
      @Test
      public void testMinus() {
      System.out.println("minus");
      String s1 = "50";
      String s2 = "10";
      CalculatorTest instance = new
CalculatorTest();
      double expResult = 40;
      double result = instance.minus(s1, s2);
      assertEquals (expResult , result , 40);
            }
}
```

From this code we can see the structure of a JUnit4 test class and how all the @Test annotation and assertEquals method is put into practice. This test is used to check whether certain units within the class 'CalculatorTest' function correctly and give the correct output. Apart from the JUnit4 specific methods and annotations the rest of the code uses normal java syntax.
When we tested the above calculator test class in the Net Beans environment the test passed successfully 100%.

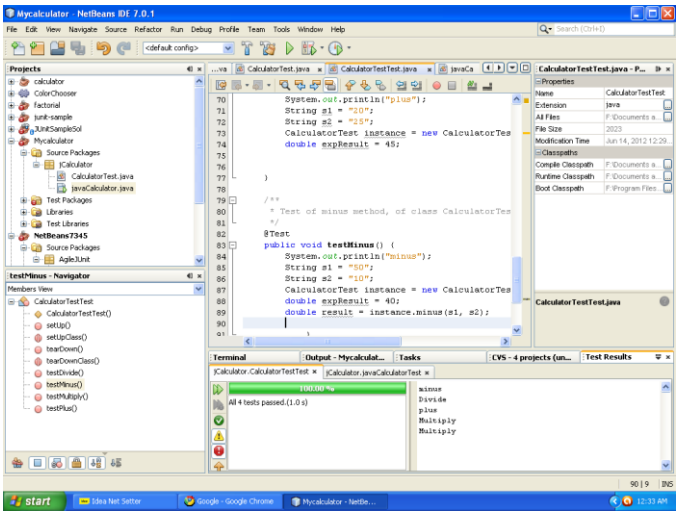

Fig. 2 above is the window for passing of CalculatorTest class under JUnit4 test.

From the above test results we can draw a graph between the no of test performed and time taken by each test. From testing of the components we can observe the time consumed by the components.

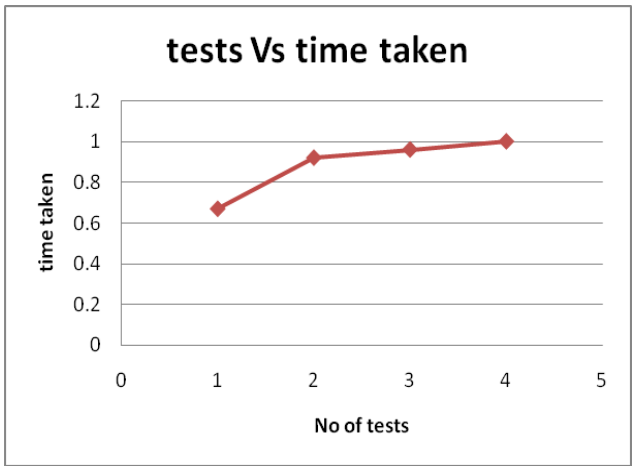

Fig. 3 A simple line graph between test vs time taken.

From the above graph it can be seen that the time taken by the first and second test component increases rapidly. While the time in the graph between first, second and third test components changes slightly. From the above graph we have observed that all the tests completed with in a minute.

Therefore from the above graph we concludes that JUnit4 testing tool helps in testing the components without consuming so much time. Hence JUnit4 testing tool helps the developer to test the components at unit level with in a less time and helps the developers to develop a reliable and bug free software.

V. CONCLUSIONS

This paper presents a systematic unit testing approach which allows engineers to identify the component standard in component – based software engineering. This paper also identifies the role of Net Beans Rich client platform in the development of component based software's. Net Beans also helps in automatic generation of Java code for Java based software's.

Another important features of Net Beans platform is that it has inbuilt support for JUnit4 tool which helps in the testing of Java based software components and reduce the testing cost. From our results we observed that JUnit4 test tool helps in testing the component at unit level and with in a less time and effort. By assuring at unit level that a component is reliable and bug free, the developers can construct software with in time.

JUnit4 tools have strong standing in terms of unit testing software in the Java world. It is one of the most used third party software in java and has helped to greatly improve java applications by making the code more reliable and bug-free.

There are also many other testing frameworks that are rising quite quickly and wish to overtake JUnit as the leading Java unit testing framework, e.g. TestNG. But we believe that JUnit will stand strong and continue to help software developers and companies create more robust and reliable software.

VI. FUTURE SCOPE

In this paper we have discussed the features of JUnit4 testing tool under Net Beans Rich Client Platform. We have created small scale software program in Net Beans environment and observes that how JUnit4 testing tool can help in testing of the component at unit level and assured the developer for the construction of software with in time.

In future we can imply Net Beans environment for large scale software and can use JUnit4 testing tool for component based software's and can helps in reducing the testing time and cost and we can observe the effect of JUnit4 testing tool on the large software without consuming time.

VII. REFERENCES

[1] David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. *In Proc. the 2002 XP/Agile Universe*, pages 131–143, 2002.

[2] JUnit Homepage - http://www.junit.org/

[3] Net beans homepage-http: // www.netbeans.org/

[4] JUnitcookbook- http://junit.sourceforge.net/doc/cookbook/cookbook.htm

[5] Tao Xie and David Notkin," Tool assisted Unit –Test Generation and election Based on operational Abstraction", ASE, vol. 13 issue 3, pp.345-371, Jul. 2006.

[6] Tao Xie, Darko Marinov, David Notkin, "Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests", *19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pp.196-205, Sept. 2004.

[7] Jerry Gao,Deepa Gopinathan and Quan Mai Jingsha He, "A Systematic Regression Testing Method and Tool for Software components", *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, **vol.** 1 ,**pp.** 455 - 466 , 17-21 Sept. 2006.

[8] In pursuit of code quality: JUnit 4 vs. TestNG by Andrew Glover - http://www.ibm.com/developerworks/java/library/j-cq08296/

