Measuring the Effectiveness of Test Cases in to Java Program by Mutation Testing Tool

Mukesh Kumari*, Sudesh Jakhar*, Mandep Singh*

*PG student BRCM CET Bahal (Haryana), **Associate Professor in BRCM CET Bahal (Haryana)
& PG (M.Tech) student Manav Rachna International University, Faridabad

Abstract
Software Testing is a process of executing software in a controlled manner, in order to answer the question-does the software behave as specified? Testing process mostly used to execute with intention to find the bugs as well as it also examines the functions and performance behaviour of a program. Mutation analysis (testing) is a fault-based technique for determining the adequacy of test suite in terms of its test effectiveness. In this paper, we implement a tool or test case for java program by using the Mutation Testing. To check the effectiveness of test case we designed many mutant programs.

Keywords
Testing, Mutation testing, Test cases, Java operators, Killed mutant, alive mutant.

1. Introduction
Software testing is the process of exercising a program with the intention of finding errors in the code. It is the process of exercising or evaluating a system or system component by manual or automatic means to verify that it satisfies specified requirements or to identify differences between expected and the actual results. The objective of testing is to show incorrectness into the program. Testing is considered to succeed when an error or fault is detected

Objectives of software testing:
Although software testing improves the quality of software by detecting the absence of code into the program. Testing also helps the developer to optimize the functionality as well as behavior of the software. Software testing is often used in association with the terms Verification and Validation.

Verification: - Are we going on the way right?
Validation: - Are we going on the right way?

1.1 Test Case Design
The test case is the collection of set of inputs and a list of expected outputs.
The objective of test case design is to test all modules and then the whole software as completely as possible using a reasonably wide range of conditions.
Test cases should be designed in such a way as to uncover quickly and easily as many errors as possible. They should “Exercise” the program by using and producing inputs and outputs that are both correct and incorrect. A good test case having the higher capability of detecting the errors or bugs. A test suite is consisting of test cases. Earlier testing process was implemented by manually which takes the high cost, time consuming and less accurate practice. But mutation testing overcomes the all above problems. It also accepts as a good tool for automated testing.

2. Mutation testing
To estimate the number of fish of a certain species in a lake, one way to do it is letting some marked fish out in the lake (say, 30) and then catches some fish and count the marked ones. If we catch 50 fish and 5 of them are marked, then 1 out of 10 is marked and the population in the entire lake could be estimated to about 200. If we catch all marked fish, we would as a side-effect end up with almost the entire population in our nets.
Fault-based testing does something similar. We let some “marked” bugs loose in the code and try to catch them. If we catch them all, our “net” probably caught many of the other, fishier, fish. The unknown bugs, that is.
One of the fault-based testing strategies is mutation testing. Mutation testing is the fault based testing technique for determining the adequacy of a test suite in term of its test effectiveness. Mutation testing means slightly modified the original program and then executed or test it. After executed the both program the output is measure with respect to the same input test. The modified program is known as the mutant program.
To mutate a program, an error is put somewhere in the code. And just like the fish in the lake, we will try to catch it. A typical mutation would be to replace < with > in one and only one expression. Example: the program P =
1. If (x > 0)
2. goThis();
3. if (x > 10)
4. doThat();
A mutation of P would be (line 1)
1. if (x < 0)
2. doThis();
3. if (x > 10)
4. doThat();
Another mutation (line 3):
1. if (x > 0)
2. doThis();
3. if (x < 10)
4. doThat();
Now we have made several copies of P and introduced a single mutation into each copy. These copies are called mutants.

Types of mutant in program: - Mostly the mutants are categories in to two forms-
1. First order mutants
2. Higher order mutants
First order mutant are generated by applying mutation operators only once.
Higher order mutants are generated by applying mutation operators more than once. It is difficult to kill the higher order mutants compare to the FOM.

3. Implementation:
For implementation the concept of mutation testing we take a triangle program in java language and executed the program with the help of mu java tool.

3.1 Mu Java
Mutation operators make syntactic changes to the program under test. These syntactic changes depict usual syntactical mistakes made by programmers while writing code. Mu Java implements a ‘do faster’ approach to mutation testing to save compilation time. This approach has been adopted primarily for object-oriented programs.
The architecture of MuJava makes use of the Mutant Schemata Generation (MSG) approach. This approach works by creating one meta-mutant of all the mutant programs and requires only two compilations: compilation of the original program and compilation of the meta-mutant program. MuJava uses two types of mutant operators:
  - Operators that change the structure of the program.
  - Operators that change the behaviour of the program.

3.2 Operators categories into Mu java.
1. Arithmetic operators
2. Relational operators
3. Conditional operators
4. Shift operators
5. Logical operators
6. Assignment operators
Some of these operators have short-cut versions as well. Because of this, some operators are subdivided into binary, unary and short-cut versions. In all, there are 12 method-level operators in MuJava.
Since MuJava has been created for Java programs, only operators used in Java are considered for the creation of the tool.

<table>
<thead>
<tr>
<th>Category</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic</td>
<td>BAOR</td>
<td>Binary Arithmetic Operator</td>
</tr>
<tr>
<td></td>
<td>UAOR</td>
<td>Replacement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Unary Arithmetic Operator</td>
</tr>
<tr>
<td></td>
<td>UAOI</td>
<td>Arithmetic Operator Replacement</td>
</tr>
<tr>
<td></td>
<td>SAOI</td>
<td>Insertion (unary)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Insertion (short-cut)</td>
</tr>
<tr>
<td>Relational</td>
<td>ROR</td>
<td>Relational Operator Replacement</td>
</tr>
<tr>
<td>Conditional</td>
<td>COR</td>
<td>Conditional Operator Replacement</td>
</tr>
<tr>
<td></td>
<td>COI</td>
<td>Conditional Operator Insertion</td>
</tr>
<tr>
<td>Shift</td>
<td>SOR</td>
<td>Shift Operator Replacement</td>
</tr>
<tr>
<td>Logical</td>
<td>LOR</td>
<td>Logical Operator Replacement</td>
</tr>
<tr>
<td></td>
<td>LOI</td>
<td>Logical Operator</td>
</tr>
</tbody>
</table>
Table 1: Method-level operators in MuJava

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD</td>
<td>Logical Operator Deletion</td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>SASR</td>
<td>Assignment Operator Replacement (short-Cut)</td>
</tr>
</tbody>
</table>

The class-level operators in MuJava have been divided in four categories according to their usage in Object Oriented programming. The first 3 groups target features common to all OO languages. The last group includes features that are specific to Java. These groups are:
1. Encapsulation
2. Inheritance
3. Polymorphism
4. Java-specific Features

Like method-level operators, the class-level operators make changes to the program syntax by inserting, deleting or modifying the expressions under test. Operators have been defined for each category. In all, there are 29 class-level operators in MuJava. These operators are explained in detail below.

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>AMC</td>
<td>Access modifier change</td>
</tr>
<tr>
<td></td>
<td>IHD</td>
<td>Hiding variable deletion</td>
</tr>
<tr>
<td></td>
<td>IHI</td>
<td>Hiding variable insertion</td>
</tr>
<tr>
<td></td>
<td>IOD</td>
<td>Overriding method deletion</td>
</tr>
<tr>
<td></td>
<td>IOP</td>
<td>Overriding method calling position change</td>
</tr>
<tr>
<td></td>
<td>IOR</td>
<td>Overriding method rename</td>
</tr>
<tr>
<td>Polymorphism</td>
<td>PNC</td>
<td>New method call with child class type</td>
</tr>
<tr>
<td></td>
<td>PMD</td>
<td>Member variable declaration with parent class type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language Feature</th>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java-specific Features</td>
<td>PCD</td>
<td>Type cast operator deletion</td>
</tr>
<tr>
<td></td>
<td>PRV</td>
<td>Reference assignment with other compatible variable</td>
</tr>
<tr>
<td></td>
<td>JTI</td>
<td>this keyword insertion</td>
</tr>
<tr>
<td></td>
<td>JTD</td>
<td>this keyword deletion</td>
</tr>
<tr>
<td></td>
<td>JSI</td>
<td>static modifier insertion</td>
</tr>
<tr>
<td></td>
<td>JSD</td>
<td>static modifier deletion</td>
</tr>
<tr>
<td></td>
<td>JID</td>
<td>Member variable initialisation deletion</td>
</tr>
<tr>
<td></td>
<td>JDC</td>
<td>Java-supported default constructor creation</td>
</tr>
<tr>
<td></td>
<td>EOA</td>
<td>Reference assignment &amp; content assignment replacement</td>
</tr>
<tr>
<td></td>
<td>EOC</td>
<td>Reference comparison &amp; content assignment replacement</td>
</tr>
<tr>
<td></td>
<td>EAM</td>
<td>Accessor method change</td>
</tr>
<tr>
<td></td>
<td>EMM</td>
<td>Modifier method change</td>
</tr>
</tbody>
</table>

Table 2: Class-level operators in MuJava

3.3 Working with MuJava

MuJava requires 2 jar files and one config file. These files can be found at: [http://ise.gmu.edu/~ofut/mujava/](http://ise.gmu.edu/~ofut/mujava/)

These 2 files, (mujava.jar & adaptedOJ.jar) should be placed in the same directory. Let’s assume the directory is C:\MuJava\.

Before executing the software, the following steps should be taken:
1. Your computer must have a JRE installed and the Java path set. To learn how to do this refer to the java.sun.com website for details.
2. After this, first set the class path using the following command:
   SETCLASSPATH=%CLASSPATH%;C:\mujava\mujava.jar;C:\mujava\adaptedOJ.jar;C:\j2sdk1.4.0_01\lib\tools.jar;C:\mujava\classes
   This command allows you to run MuJava to create mutants. To run test cases a different path must be set which is explained later.
3. After this open the mujava.config file. You should see ‘MuJava_HOME=C:\ofut\mujava’. Change this to point to ‘MuJava_HOME=C:\MuJava_Home’. This command tells MuJava where to find the source files and the test programs when executing MuJava.
4. Now we need to make a directory structure for MuJava. All the source programs and test programs will be contained in this directory structure. This structure can be made either manually or by using the command:
   Java mujava.makeMuJavaStructure

The directory structure will now look like:
C:\MuJava_Home\classes
C:\MuJava_Home\testset
C:\MuJava_Home\src
C:\MuJava_Home\result

Now, you will have 2 different directories; one will be ‘C:\MuJava’ and the other will have ‘C:\MuJava_Home\’. The config file should be in the directory of MuJava.

Creating Mutants
1. Create your source file (say AccessorModifier.java) that you want to test in C:\MuJava_Home\src.
2. Compile this file using the DOS prompt and the command ‘javac AccessorModifier.java’. This command will create a class file in the folder C:\MuJava_Home\src\.
3. Remove this file from the C:\MuJava_Home\src\ and copy it to the folder C:\MuJava_Home\classes\.
4. Now change the path in the dos prompt to point to C:\MuJava\ instead of C:\MuJava_Home\src\.
5. Now set the class path using:
   Setclasspath=%classpath%;c:\MuJava\mujava.jar;c:\MuJava\adaptedOJ.jar;c:\J2ee\jdk\lib\tools.jar;c:\MuJava_Home\classes;
   6. Then use the following command to start MuJava:
      Java mujava.gui.GenMutantsMain
7. Now select the programs from the list shown for which you want to create the mutants. Also select the mutation operators for which you want to create mutants. Then press ‘Generate’. Whilst mutants are being generated, many messages will be sent to the command prompt in the background. The ‘Generate’ button will be disabled. When all the mutants have been generated, this button will turn yellow again and the different mutants that have been generated can then be viewed.

Creating Mutant with the help of Mujava

The MY TRIANGLE Program (in Java)
Public static int getType (int side1, int side2, int side3)
{
    return getType(new int[]{ side1, side2, side3 });
}
public static int getType (int[] sides)
{
    if (sides.length != 3)
    return ILLEGAL_ARGUMENTS;
    if (sides[0] < 0 || sides[1] < 0 || sides[2] < 0)
    return ILLEGAL_ARGUMENTS;
    int triang = 0;
    return ILLEGAL;
    if (triang == 0)
    return SCALENE;
    else if (triang == 1)
    return ISOSCELES;
    else if (triang == 2)
    return ISOSCELES;
    else if (triang == 3)
    return EQUILATERAL;
    else if (triang == 4)
    return SCALENE;
    }
else if (triang == 3 && sides[1] + sides[2] > sides[0])
    return ISOSCELES;
return ILLEGAL;
}

We design 13 test cases for the triangle program
1. A test case which represents a valid scalene triangle.
2. A test case which represents a valid equilateral triangle.
3. A test case which represents a valid isosceles triangle.
4. At least three test cases which represent valid isosceles triangles such that you have tried all three permutations of two equal sides.
5. A test case in which one side is zero.
6. A test case in which one side is negative.
7. A test case with three positive integers such that the sum of two of them is equal to the third.
8. At least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides.
9. A test case with the sum of two of the numbers less than the third.
10. At least three cases in category 9 such that you have tried all three permutations.
11. A test case with all side lengths equal to zero.
12. At least one test case specifying non-integer values.
13. At least one test case specifying the wrong number of values (two or four).

The total is the mutation score of the entire test set. The “statement coverage” column shows the percentage of executed statements of the program under test.

Test case Mutation score Statement coverage
1  36% 40%
2  23% 52%
3  32% 48%
4  49% 64%
5  3% 16%
6  3% 16%
7  34% 40%
8  44% 40%
9  19% 40%
10  26% 40%
11  0% 16%
13  5% 8%

6. Conclusion:
In this paper, we examine the utilities of mutation testing to check the effectiveness of given automated test case by creating mutants in the original program.

7. References:


Authors Profile

MUKESH KUMARI received the 3-years Diploma in Computer Science from Rani Jhashi Lexmi bai Govt polytechnic College Loharu. In 2008 received bachelor degree in Information Technology and Engineering from BRCM Engineering College, BAHAL (Haryana), India. She is pursuing her Master degree in Computer Science from BRCM Engineering College, BAHAL (Haryana).