Software testing verification and validation

MIDTERM REPORT

Moustapha Sy Dieng

mxd152830

Ponmalar Silambaram Chandrabose

pxs162030

Trung Hieu Tran

txt171930

1. The problem statement

In 1970, Edsger W. Dijkstra said that "program testing can be used to show the presence of bugs, but never to show their absence" [1].

In fact, it can be easily demonstrated that even achieving a hundred percent code coverage does not guarantee bug-free software. For this reason, we can't effectively rely on code coverage as a metric for software quality. Our aim is to therefore utilize mutation testing to measure unit tests fiability. Mutation testing provides the ability to not only detect if code is executed but also evaluate the test suite ability to uncover regressions. It accomplishes this by introducing changes to the code, called mutations, and running the test suite against the newly modified code. In the next section, we will discuss some techniques used to implement it.

2. The basics of existing techniques and/or the design of your technique

2.1 Overview of PIT Mutation Testing:

Mutation testing is a concept where faults (or mutations) are automatically seeded into the code, tests are run and if any of the tests fail, then the mutation is killed. Otherwise the mutation lived. The percentage of mutations gauge the quality of the tests performed.

Specifically PIT Mutation Testing, an open source mutation testing tool is selected. Here, PIT runs your unit test against versions of application code that have been automatically modified. When the tests are run, since the application code has been changed it should produce different results. If a unit test does not fail in this situation, then there is some problem in the test suite [2].

2.2 Why use PIT?:

PIT Mutation Testing is fast because it operates on bytecodes and optimizes the executions of mutants. It can also be integrated into a wide range of development tools and also invoked through command line interfaces like Ant or Maven.

2.3 Existing Mutators:

Some of the mutators that are already available in the PIT Mutation Testing include

Conditionals Boundary Mutator : They replace the relational operators

Increments Mutator : They mutate the increments, decrements and assignment increments and decrements of local variables

Math Mutator : Replaces binary arithmetic operations for either integer or floating - point arithmetic

Invert Negatives Mutator : It inverts negation of integer and floating point numbers.

Return values mutator : It mutates the return values of the method calls depending on the type

Void Method calls Mutator : It removes method calls to void methods

3. Your implementation/study plan

Project Goals

The goal of the project was to augment the PIT mutation testing tool by adding three additional mutators:

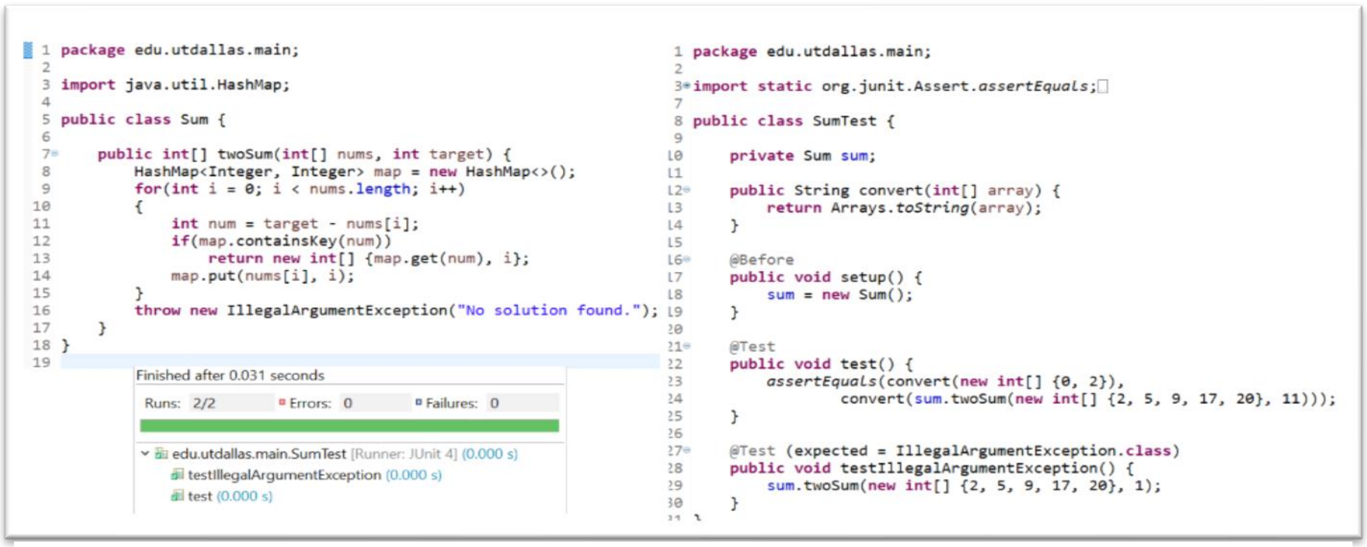
1. AOD (Arithmetic Operator Deletion): Replace an arithmetic expression by each one of the operand.
2. AOR (Arithmetic Operator Replacement): Replace an arithmetic expression by each of the other ones.
3. ROR (Relational Operator Replacement): Replace the relational operators with each of the other ones.

Preparations

Our first task was to setup our environment. We opted to use the following configuration:

- IDE: Eclipse Oxygen
- Java version: JDK 1.8
- Build system: Apache Maven 3.5.3
- Version control: GitHub

Once we setup the environment, we created a small program along with a couple of test cases to ensure we were able to run unit tests properly. (Figure 1)



```java
package edu.utdallas.main;

import java.util.HashMap;

public class Sum {

    public int[] twoSum(int[] nums, int target) {
        HashMap<Integer, Integer> map = new HashMap<>();
        for(int i = 0; i < nums.length; i++)
        {
            int num = target - nums[i];
            if(map.containsKey(num))
                return new int[] {map.get(num), i};
            map.put(nums[i], i);
        }
        throw new IllegalArgumentException("No solution found.");
    }
}
```

```
Finished after 0.031 seconds
Runs: 2/2    Errors: 0    Failures: 0
edu.utdallas.main.SumTest [Runner: JUnit 4] (0.000 s)
    testIllegalArgumentException (0.000 s)
    test (0.000 s)
```

```java
package edu.utdallas.main;

import static org.junit.Assert.assertEquals;

public class SumTest {

    private Sum sum;

    public String convert(int[] array) {
        return Arrays.toString(array);
    }

    @Before
    public void setup() {
        sum = new Sum();
    }

    @Test
    public void test() {
        assertEquals(convert(new int[] {0, 2}),
                convert(sum.twoSum(new int[] {2, 5, 9, 17, 20}, 11)));
    }

    @Test (expected = IllegalArgumentException.class)
    public void testIllegalArgumentException() {
        sum.twoSum(new int[] {2, 5, 9, 17, 20}, 1);
    }
}
```

Figure 1

Next, we cloned the pitest repository from Github. We ran into our first obstacle when attempting to run mutation testing on our program. We were mistakenly attempting to build the project we created with the command 'mvn clean install' command. After much troubleshooting, we figured out that the pitest project needed to be built in this manner. After successfully building the project, we were able to run the default pitest mutators on our program successfully.

Strategy

We decided to divide up the workload. Trung worked on implementing AOD, Ponmalar Silambaram Chandrabose took on AOR's implementation and Moustapha handled ROR's.

AOD

AOD replaces an arithmetic expression by each one of the operand. Below is the replacement table which is applied in the scope of the program.

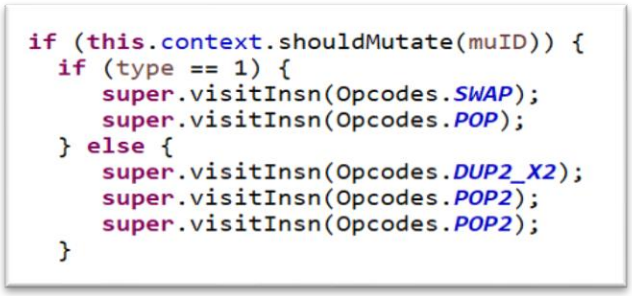
Operator	Expression sample	Replaced by the first operand	Replaced by the second operand
+	a + b	a	b
-	a - b		
*	a * b		
/	a / b		
%	a % b		

First operand replacement

To replace an expression by the first operand, we had to remove the first parameter from the stack of java virtual machine which is associated the second operand in the expression. However, we needed to take into account the way JVM stores the different data types. Data types with sizes betweem 8-bits and 32-bits occupy 1 stack slot, while 64-bits data occupies 2 stack slots. Therefore, it would take 1 stack slot for *int* and *float* and this figure is 2 stack slots for *double* and *long.* Thus, we used the operand stack *POP* to remove item from the stack for *int* and *float* operators and the *POP2* for *double* and *long* operators. The code snippet in figure 3 shows the example of implementation for addition operator

Second operand replacement

To replace an expression by the second operand, there are 2 scenarios. For the int and float operators, we use operand *SWAP* to swap two items on the stack. Then, the order of operand in the expression is changed and the problem becomes the replace the expression by the first operand. In case of long and double operators, there doesn't exist any operand to swap first two slots with next two slots in the stack. So that we use *DUP2_X2* to duplicate the first two slots in the stack and push them after the 4-th slot in the stack. Then we use *POP2* two times to pop first 4-slots on the top of the stack and the remainder is the second operand of the expression. Below is the code snippet for those above scenarios.



```java
if (this.context.shouldMutate(muID)) {
  if (type == 1) {
     super.visitInsn(Opcodes.SWAP);
     super.visitInsn(Opcodes.POP);
  } else {
     super.visitInsn(Opcodes.DUP2_X2);
     super.visitInsn(Opcodes.POP2);
     super.visitInsn(Opcodes.POP2);
  }
```

Figure 2

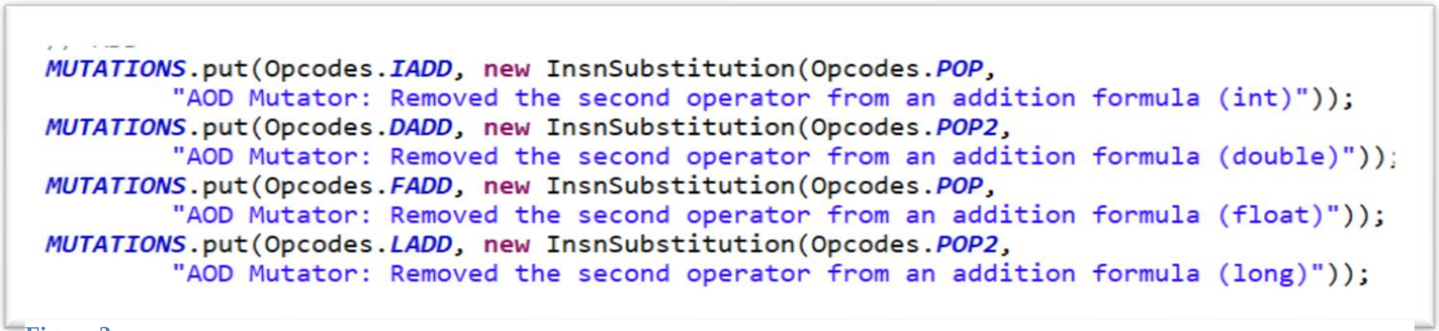
AOR

Definition

Arithmetic Operator Replacement is defined as the mutation process where one specific operator in an expression is replaced by all other operators.

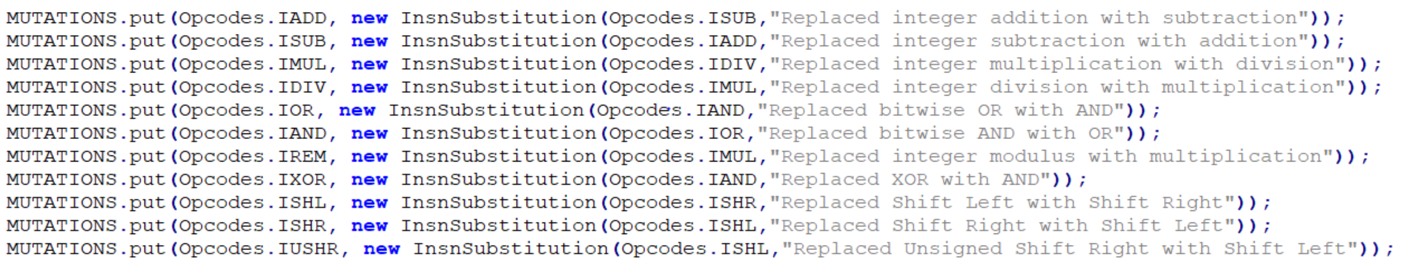
For example: Let us consider a+b as an expression. This mutation operator should replace the ‘+’ operator by subtraction, multiplication, division and modulus operator.

We used java byte code manipulation to replace these operators directly in the byte code of the program rather than change them manually in the system.



```
MUTATIONS.put(Opcodes.IADD, new InsnSubstitution(Opcodes.POP,
        "AOD Mutator: Removed the second operator from an addition formula (int)"));
MUTATIONS.put(Opcodes.DADD, new InsnSubstitution(Opcodes.POP2,
        "AOD Mutator: Removed the second operator from an addition formula (double)"));
MUTATIONS.put(Opcodes.FADD, new InsnSubstitution(Opcodes.POP,
        "AOD Mutator: Removed the second operator from an addition formula (float)"));
MUTATIONS.put(Opcodes.LADD, new InsnSubstitution(Opcodes.POP2,
        "AOD Mutator: Removed the second operator from an addition formula (long)"));
```

Figure 3



```
MUTATIONS.put(Opcodes.IADD, new InsnSubstitution(Opcodes.ISUB,"Replaced integer addition with subtraction"));
MUTATIONS.put(Opcodes.ISUB, new InsnSubstitution(Opcodes.IADD,"Replaced integer subtraction with addition"));
MUTATIONS.put(Opcodes.IMUL, new InsnSubstitution(Opcodes.IDIV,"Replaced integer multiplication with division"));
MUTATIONS.put(Opcodes.IDIV, new InsnSubstitution(Opcodes.IMUL,"Replaced integer division with multiplication"));
MUTATIONS.put(Opcodes.IOR, new InsnSubstitution(Opcodes.IAND,"Replaced bitwise OR with AND"));
MUTATIONS.put(Opcodes.IAND, new InsnSubstitution(Opcodes.IOR,"Replaced bitwise AND with OR"));
MUTATIONS.put(Opcodes.IREM, new InsnSubstitution(Opcodes.IMUL,"Replaced integer modulus with multiplication"));
MUTATIONS.put(Opcodes.IXOR, new InsnSubstitution(Opcodes.IAND,"Replaced XOR with AND"));
MUTATIONS.put(Opcodes.ISHL, new InsnSubstitution(Opcodes.ISHR,"Replaced Shift Left with Shift Right"));
MUTATIONS.put(Opcodes.ISHR, new InsnSubstitution(Opcodes.ISHL,"Replaced Shift Right with Shift Left"));
MUTATIONS.put(Opcodes.IUSHR, new InsnSubstitution(Opcodes.ISHL,"Replaced Unsigned Shift Right with Shift Left"));
```

Figure 4

The possible combinations are:

+	-	*	/	%
-	*	/	%	+
*	/	%	+	-
/	%	+	-	*
%	+	-	*	/

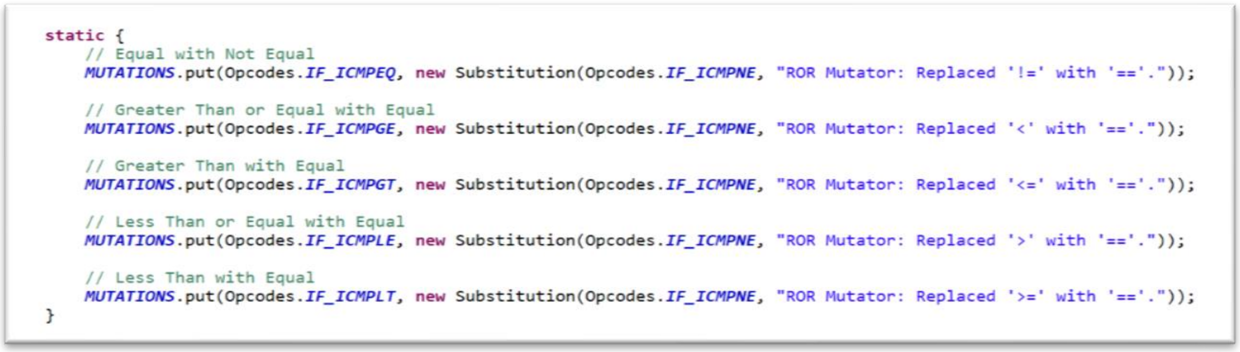
Thought Process
Solution 1:
We need to implement java byte code manipulation to replace these arithmetic operators in the byte code of the program directly.
We also need to take into consideration the various combinations of operator exchange possible in the system.
Solution 2: Manually go through the program and replace all of them, which becomes tedious if the program to be tested is really long.
Roadblocks:
Learning how to access the Java Byte code for that particular method.
Solution/Implementation:
We took into consideration the data types such as integer, long, double and float for arithmetic operators because each of them are stored differently in the stack.
We also took into consideration the combinations of operators that can be replaced.

We used the MethodVisitor function to look through the system and visit every method in the program.
In each method we look for the ADD, SUB, MUL, DIV, REM, as well as bitwise operation functions for each respective data type represented in the byte code and replace them with each other using the put function in java
We maintain HashMap values of each of these and create separate functions for each combination possible to make it faster to implement the test function.

ROR

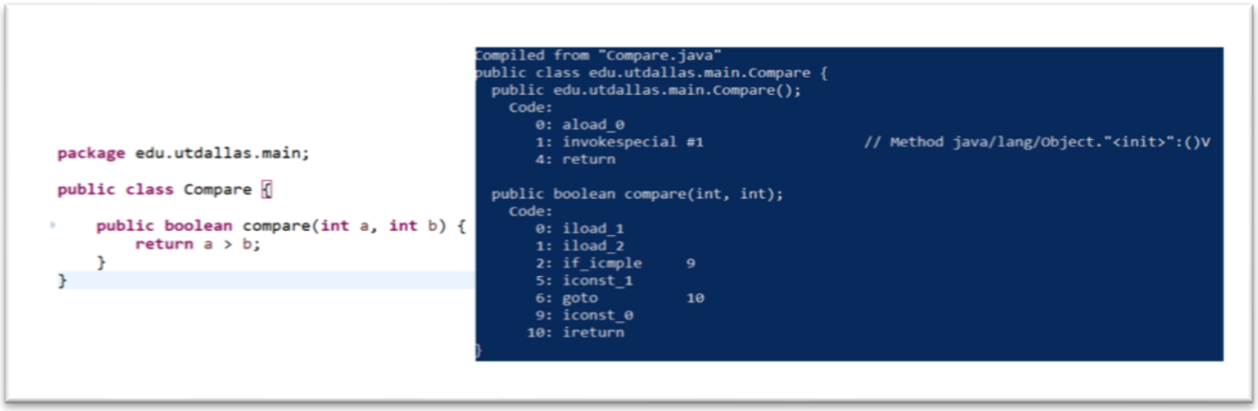
The implementation of the relational operator replacement required for each relational operator to be replaced by every other one. The table below shows the all the possible replacements we implemented (note: We only handled the cases listed on the project guideline):

==	!=	>=	>	<=	<
!=	>=	>	<=	<	==
>=	>	<=	<	==	!=
>	<=	<	==	!=	>=
<=	<	==	!=	>=	>
<	==	!=	>=	>	<=



```
static {
    // Equal with Not Equal
    MUTATIONS.put(Opcodes.IF_ICMPEQ, new Substitution(Opcodes.IF_ICMPNE, "ROR Mutator: Replaced '!=' with '=='."));

    // Greater Than or Equal with Equal
    MUTATIONS.put(Opcodes.IF_ICMPGE, new Substitution(Opcodes.IF_ICMPNE, "ROR Mutator: Replaced '<' with '=='."));

    // Greater Than with Equal
    MUTATIONS.put(Opcodes.IF_ICMPGT, new Substitution(Opcodes.IF_ICMPNE, "ROR Mutator: Replaced '<=' with '=='."));

    // Less Than or Equal with Equal
    MUTATIONS.put(Opcodes.IF_ICMPLE, new Substitution(Opcodes.IF_ICMPNE, "ROR Mutator: Replaced '>' with '=='."));

    // Less Than with Equal
    MUTATIONS.put(Opcodes.IF_ICMPLT, new Substitution(Opcodes.IF_ICMPNE, "ROR Mutator: Replaced '>=' with '=='."));
}
```

Figure 5



```
package edu.utdallas.main;

public class Compare {

    public boolean compare(int a, int b) {
        return a > b;
    }
}
```

```
Compiled from "Compare.java"
public class edu.utdallas.main.Compare {
  public edu.utdallas.main.Compare();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public boolean compare(int, int);
    Code:
       0: iload_1
       1: iload_2
       2: if_icmple     9
       5: iconst_1
       6: goto          10
       9: iconst_0
      10: ireturn
}
```

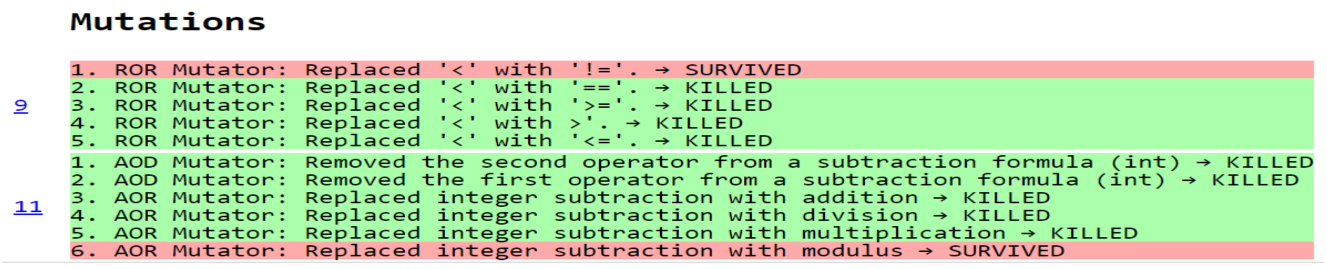
Figure 6

Approach
We followed the same approach used by the existing mutators that perform similar replacement such as the Conditional Boundary Mutator. We created a single mutator through an enum that implemented the MethodMutatorFactory which is responsible for creating the MethodVisitor method used to find mutation points and apply the mutators. Each method visitor extends AbstractJumpMutator since it contains the Substitution class which can replace one operator with another.
The flaw of this approach was that each mutator can only produce one mutant per mutation point. At first, we thought to change the way the mutations are stored by changing the map to a MultivaluedMap and the HashMap to a MultivaluedHashMap. However, that would require changing code outside the class we were creating.

Solution
We ended up creating a class containing several mutators defined through enums. We also defined method visitor for each of the cases we were going to handle. Below is a snippet of the method visitor responsible for replacing operators with the '!='.
Note that the operators are not equivalent to the signs used. The reason is that JVM flips the signs of the operator with their complement. As you can see in the figure below '>' becomes 'if_icmple'.

Results
We first tested our mutators on the program we created.

Future plans
For the second phase, we are planning to implement the four mutators listed on the project guidelines and possibly some additional ones as time permits

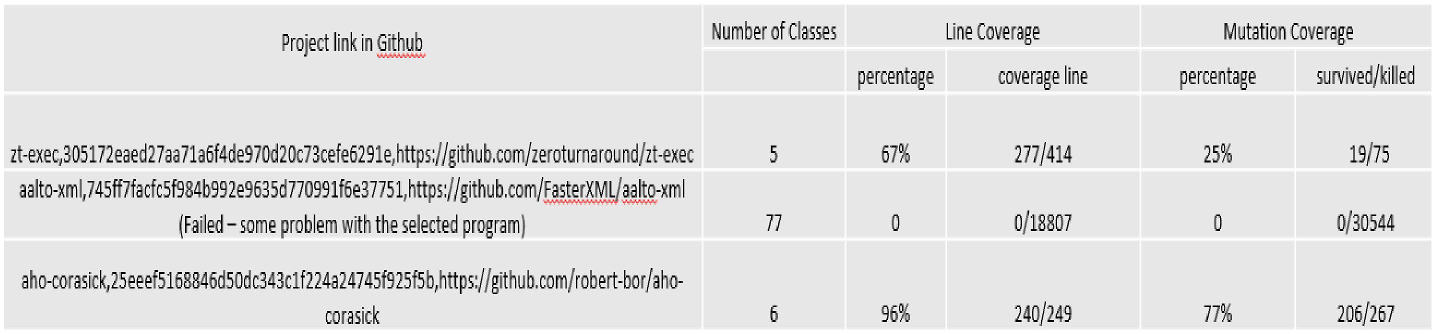


```
Mutations

    1. ROR Mutator: Replaced '<' with '!='. → SURVIVED
    2. ROR Mutator: Replaced '<' with '=='. → KILLED
9   3. ROR Mutator: Replaced '<' with '>='. → KILLED
    4. ROR Mutator: Replaced '<' with >'. → KILLED
    5. ROR Mutator: Replaced '<' with '<='. → KILLED
    1. AOD Mutator: Removed the second operator from a subtraction formula (int) → KILLED
    2. AOD Mutator: Removed the first operator from a subtraction formula (int) → KILLED
11  3. AOR Mutator: Replaced integer subtraction with addition → KILLED
    4. AOR Mutator: Replaced integer subtraction with division → KILLED
    5. AOR Mutator: Replaced integer subtraction with multiplication → KILLED
    6. AOR Mutator: Replaced integer subtraction with modulus → SURVIVED
```

Figure 7

Results of Tested projects:

The following table represents the Testing results of three random projects from github on our mutation tester program.



Project link in Github	Number of Classes	Line Coverage		Mutation Coverage	
		percentage	coverage line	percentage	survived/killed
zt-exec,305172eaed27aa71a6f4de970d20c73cefe6291e,https://github.com/zeroturnaround/zt-exec	5	67%	277/414	25%	19/75
aalto-xml,745ff7facfc5f984b992e9635d770991f6e37751,https://github.com/FasterXML/aalto-xml (Failed – some problem with the selected program)	77	0	0/18807	0	0/30544
aho-corasick,25eeef5168846d50dc343c1f224a24745f925f5b,https://github.com/robert-bor/aho-corasick	6	96%	240/249	77%	206/267

References:

[1] E. W. Dijkstra, Notes On Structured Programming, Section 3 ("On The Reliability of Mechanisms"), 1970.
[2] http://pitest.org/