**MIDTERM REPORT**

**Project Goals**

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

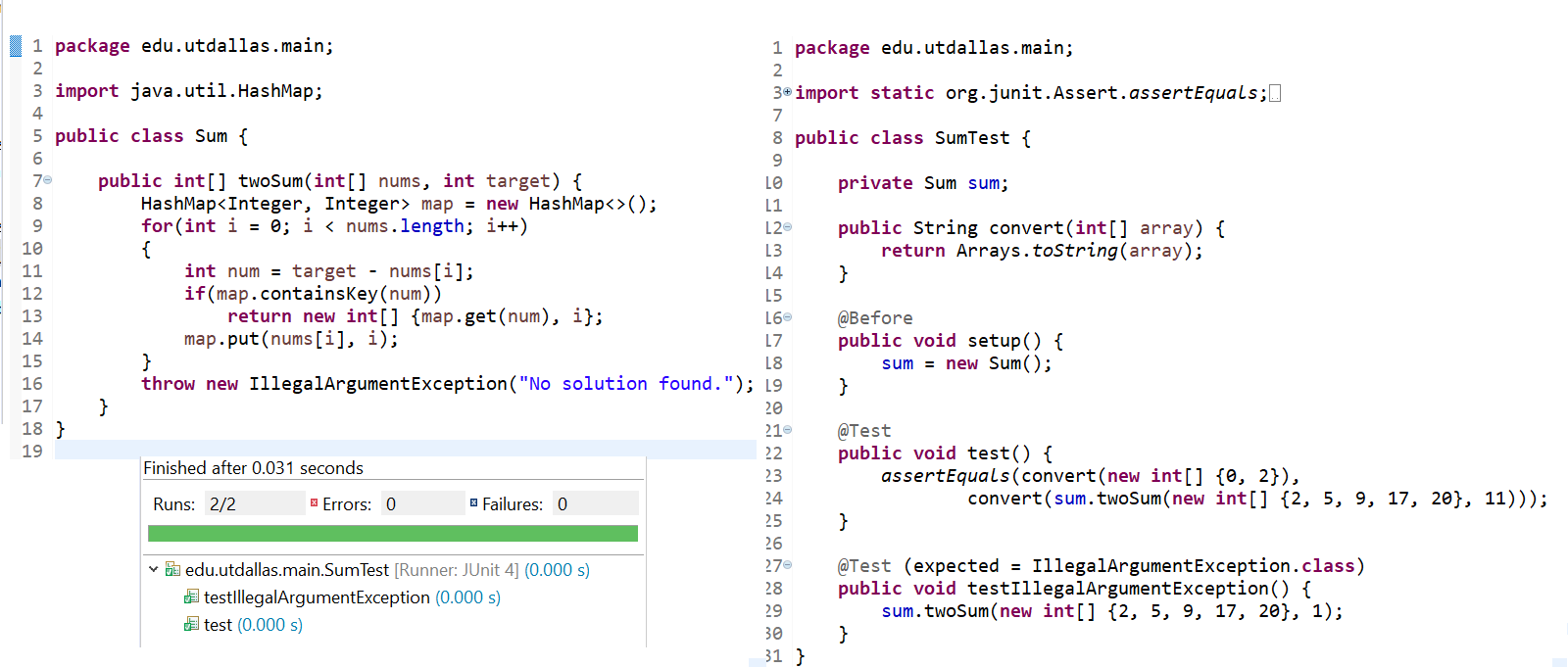
1. AOD (Arithmetic Operator Deletion): Replace an arithmetic expression by each one of the operand.
2. AOR (Arithmetic Operator Replacement): Replace an arithmetic expression by each of the other ones.
3. ROR (Relational Operator Replacement): Replace the relational operators with each of the other ones.

**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.



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**

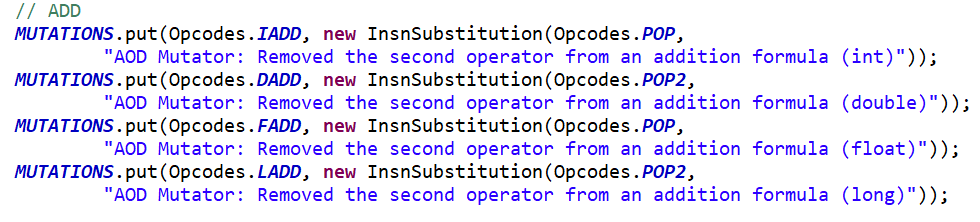
We decided to divide up the workload. Trung worked on implementing AOD, Chandrabo took on AOR’s implementation and Moustapha handled ROR’s.

**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.

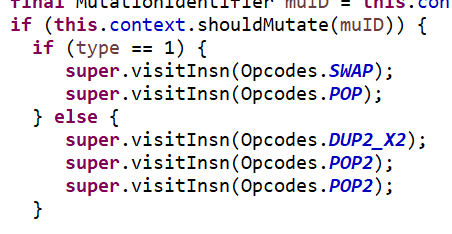
|  |  |  |  |
| --- | --- | --- | --- |
| Operator | Expression sample | Replaced by the first operand | Replaced by the second operand |
| + | a + b | a | b |
| - | a - b |
| \* | a \* b |
| / | a / b |
| % | a % b |

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 below 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 timesto 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:



**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.

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.

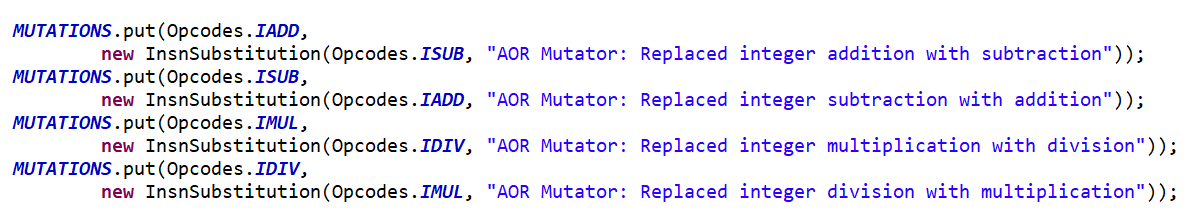
We also took into consideration the combinations of operators that can be replaced.

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

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 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):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| == | != | >= | > | <= | < |
| != | >= | > | <= | < | == |
| >= | > | <= | < | == | != |
| > | <= | < | == | != | >= |
| <= | < | == | != | >= | > |
| < | == | != | >= | > | <= |

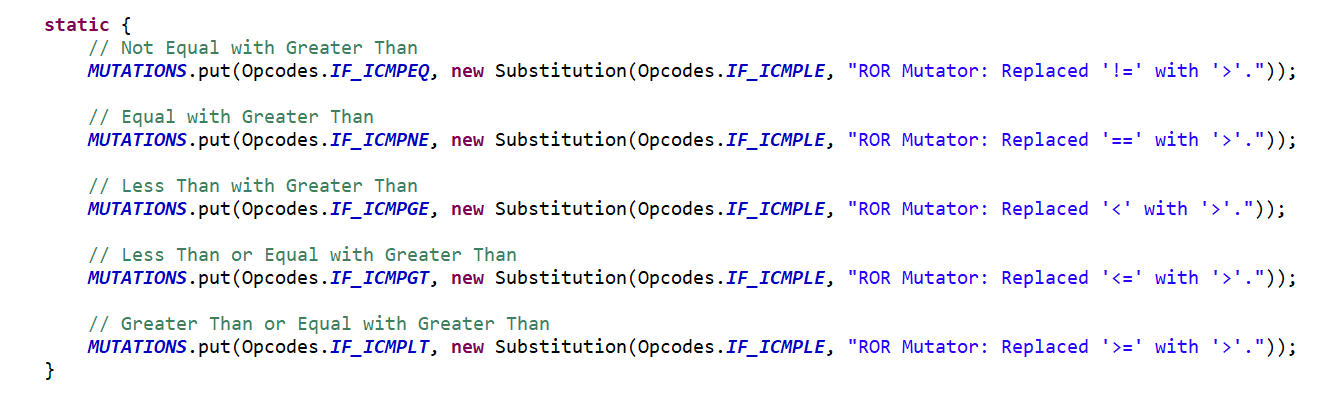
Approach

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.

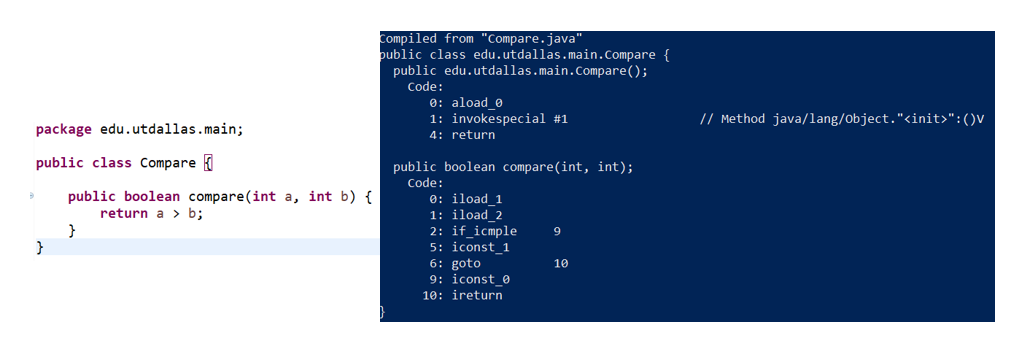
The flaw of this approach was that each mutator can only produce one mutant per mutation point. At first, we thought to change the way the mutations are stored by changing the map to a MultivaluedMap and the HashMap to a MultivaluedHashMap. However, that would require changing code outside the class 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 ‘>’.

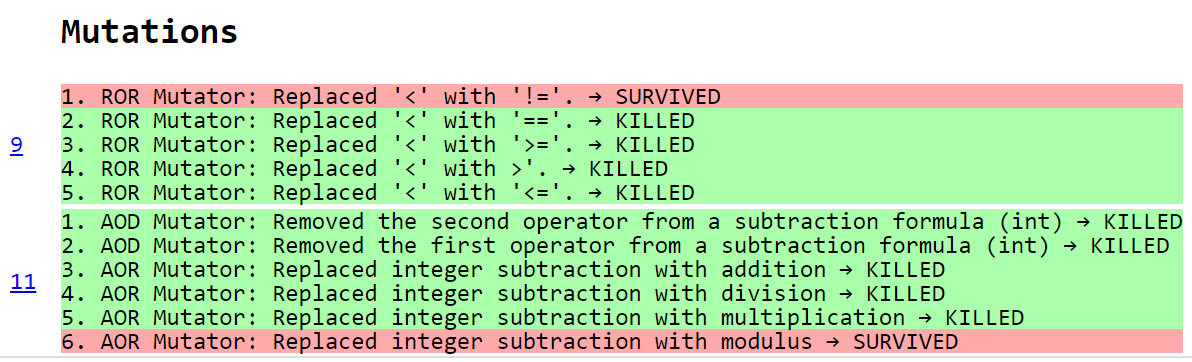


Note that the operators are not equivalent to the signs used. The reason is that JVM flips the signs of the operator with their complement. As you can see in the figure below ‘>’ becomes ‘if\_icmple’. JVM then simply compares operand 2 with 1 rather than 1 with 2.



**Results**

We first tested our mutators on the program we created and got the following results.



**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.

**UOR**

We implemented the unary operator replacement mutator to target every instance of the increment and decrement operators and replace them with the following:

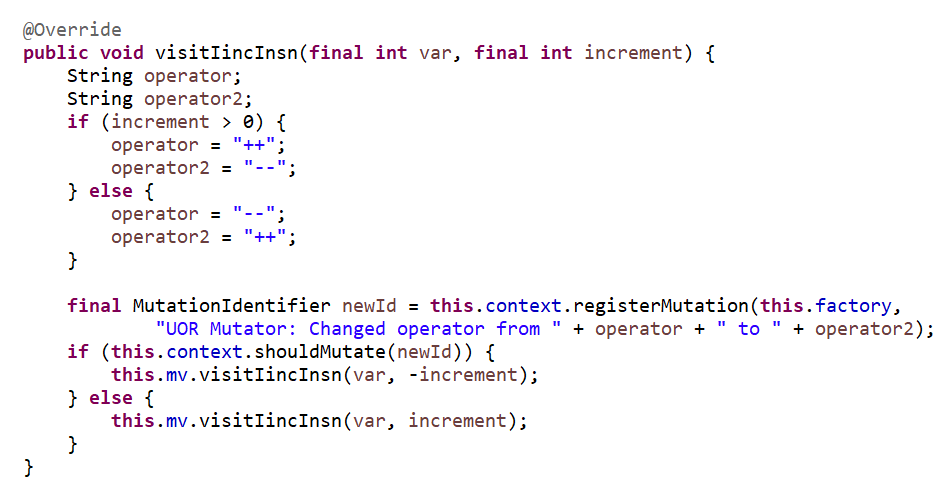
|  |  |  |  |
| --- | --- | --- | --- |
| **DECREMENT** | **INCREMENT** | **REMOVE** | **REVERSE** |

Special cases:

1. If the ‘INCREMENT’ operator is target by the ‘DECREMENT’ operator, it acts as a ‘REMOVE’.
2. If the ‘INCREMENT’ operator is target by the ‘INCREMENT’ operator, it acts as a ‘FURTHER INCREMENT’.
3. The reverse of the two cases listed above is true for the ‘DECREMENT’ operator.

Approach

The implementation of this mutator was based on the already existing NegateVariableMutator which essentially performs the ‘REVERSE’ mutation. It makes use of the visitIincInsn which takes the index of the local variable to be incremented and the amount to increment the local variable by as arguments.



In order to track the sign of the variable to be mutated, we check whether or not the increment is positive or negative. We also refined the way we create the mutator. Rather than creating one enum per mutator, we created a class that implements the MethodMutatorFactory and then defined the enums within the class.

