MuJava: A Test Tool Analysis

Name: Róisín Ní Bhriain
Student Number: 23269640
Email: roisin.nibhriain3@mail.dcu.ie
Program: MCM Secure Software Engineering
Module Code: CA650
Submission: Test Tool Evaluation Project

Róisín Ní Bhriain - 23269640

I understand that the University regards breaches of academic integrity and plagiarism as grave and serious. I have read and understood the DCU Academic Integrity and Plagiarism Policy. I accept the penalties that may be imposed should I engage in practice or practices that breach this policy. I have identified and included the source of all facts, ideas, opinions, viewpoints of others in the assignment references. Direct quotations, paraphrasing, discussion of ideas from books, journal articles, internet sources, module text, or any other source whatsoever are acknowledged and the sources cited are identified in the assignment references. Any use of generative AI or search will be described in a one page appendix including prompt queries.

I declare that this material, which I now submit for assessment, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work. By signing this form or by submitting this material online I confirm that this assignment, or any part of it, has not been previously submitted by me or any other person for assessment on this or any other course of study. By signing this form or by submitting material for assessment online I confirm that I have read and understood DCU Academic Integrity and Plagiarism Policy.

Signed Name: Róisín Ní Bhriain
Date: 25-03-2024

Introduction

The tool I chose to analyse for this assignment was the MuJava mutation testing tool. It is an open-source testing tool used to test Java software and can be installed on Linux, Windows and Mac operating systems. Mutation testing is a method of generating faults in a piece of software to test the test suite (Ma et al., 2006). The mutants themselves are the given code with a small change. MuJava itself supports the mutation process for Java software. It generates mutants for the software and runs these mutants against a test suite. It displays test coverage results as a score in the interface (Ma et al., 2006). It offers a more reliable coverage score than line coverage as it tests the logic by changing it to see whether tests fail.

Setup

For the purpose of this section I will discuss the setup of the tool in Windows as this is the operating system that I use. Firstly three files must be downloaded and the two jar files are to be added to the CLASSPATH environment variable. The config file is used to point to a directory where the system will be run. Finally, a specific directory structure must be created which can be done using a single command:

```
java mujava.makeMuJavaStructure
```

Usage

You must place all the source files you wish to test under the /src directory in your directory structure. Then all the compiled classes must be placed in the /classes directory. You can then start the mutant generation GUI by using this command:

```
java mujava.gui.GenMutantsMain
```

After the mutants have been compiled with no errors you can view both the traditional mutants and the class mutants with the interface to see what changes have been made in each mutation. Finally, you must place all the test classes in the /testset directory and start the test GUI using this command:

```
java mujava.gui.RunTestMain
```

After starting this GUI with the mutants already generated you can choose to execute all mutants or choose between traditional or class mutants and you can also choose which methods you would like to execute the mutants or choose all methods. After the mutants have all been executed you will get a score displayed of how many mutants are “live”. If a mutant is live after execution then this means that the change has not been tested properly by the unit testing that you provided and the coverage is incomplete.

Sample Runs

For the purpose of this section I will be using a project I wrote a number of years ago of a class to check if points lie in the same line. I also include the code I wrote for the Queue object in the last assignment as code to test. I wrote tests for each of these objects which I will use to examine this testing tool. The tests were written using JUnit 4.

Collinear Code Used

```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.Reader;

class Collinear
{

	static int countCollinear(int[] a1, int[] a2, int[] a3)
	{
		int y1 = 1;
		int y2 = 2;
		int y3 = 3;
		int count = 0;
		for (int i = 0; i < a1.length; i++) {
			for (int j = 0; j < a2.length; j++) {
				for (int k = 0; k < a3.length; k++) {
					if (((a1[i]*(y2-y3)) + (a2[j]*(y3-y1)) + (a3[k]*(y1-y2))) == 0) {
						count++;
					}
				}
			}
		}
		return count;
	}

	static int countCollinearFast(int[] a1, int[] a2, int[] a3)
	{
		Collinear.sort(a3);
		int count = 0;
		for (int i = 0; i < a1.length; i++) {
			for (int j = 0; j < a2.length; j++) {
				int currentNumber = - ((a1[i]*(2-3)) + (a2[j]*(3-1))) / (1-2);
				if (Collinear.binarySearch(a3, currentNumber)) {
					count++;
				}
			}
		}
		return count;
	}

	static void sort(int[] a)
	{
		for ( int j = 1; j<a.length; j++)
		{
			int i = j - 1;
			while(i>=0 && a[i]>a[i+1])
			{
				int temp = a[i];
				a[i] = a[i+1];
```

```
                                        a[i+1] = temp;
                                        i--;
                                }
                        }
                }

                static boolean binarySearch(int[] a, int x)
                {
                        int low = 0, high = a.length-1;
                        while (low <= high)
                        {
                                int mid = low + (high - low) / 2;
                                if (x < a[mid]) high = mid - 1;
                                else if (x > a[mid]) low = mid + 1;
                                else return true;
                        }
                        return false;
                }
}
```

Code Snippet 2: Collinear Class implementation in Java (‘Algorithms-and-Data-Structures/Collinear/Src at Main · Rnibhriain/Algorithms-and-Data-Structures’, n.d.)

Collinear Test Class Used

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.Ignore;
import org.junit.runner.RunWith;
import org.junit.runners.JUnit4;
import static org.junit.Assert.*;
import java.io.*;
import java.util.Scanner;

import java.util.Arrays;

@RunWith(JUnit4.class)
public class CollinearTest
{



        //~ Constructor ........................................................
        @Test
        public void testConstructor()
        {
                new Collinear();
        }

        //~ Public Methods .....................................................

        // ----------------------------------------------------------
        /**
         * Check that the two methods work for empty arrays
         */
        @Test
        public void testEmpty()
        {
                int expectedResult = 0;
```

```
                assertEquals("countCollinear failed with 3 empty arrays",       expectedResult,
Collinear.countCollinear(new int[0], new int[0], new int[0]));
                assertEquals("countCollinearFast failed with 3 empty arrays", expectedResult,
Collinear.countCollinearFast(new int[0], new int[0], new int[0]));
        }

        // ---------------------------------------------------------
        /**
         * Check for no false positives in a single-element array
         */
        @Test
        public void testSingleFalse()
        {
                int[] a3 = { 15 };
                int[] a2 = { 5 };
                int[] a1 = { 10 };

                int expectedResult = 0;

                assertEquals("countCollinear({10}, {5}, {15})",       expectedResult, Collinear.countCollinear(a1, a2, a3) );
                assertEquals("countCollinearFast({10}, {5}, {15})", expectedResult, Collinear.countCollinearFast(a1, a2,
a3) );
        }

        // ---------------------------------------------------------
        /**
         * Check for no false positives in a single-element array
         */
        @Test
        public void testSingleTrue()
        {
                int[] a3 = { 15, 5 };       int[] a2 = { 5 };       int[] a1 = { 10, 15, 5 };

                int expectedResult = 1;

                assertEquals("countCollinear(" + Arrays.toString(a1) + "," + Arrays.toString(a2) + "," +
Arrays.toString(a3) + ")",     expectedResult, Collinear.countCollinear(a1, a2, a3));
                assertEquals("countCollinearFast(" + Arrays.toString(a1) + "," + Arrays.toString(a2) + "," +
Arrays.toString(a3) + ")", expectedResult, Collinear.countCollinearFast(a1, a2, a3));
        }

        @Test

        // tests to see if the binary search gives a correct false

        public void testFalseBinary () {
                int [] array = {5, 10, 15, 20};

                int number = 11;

                boolean expectedResult = false;

                assertEquals("binarySearch(" + Arrays.toString(array) + number + ")", expectedResult,
Collinear.binarySearch(array, number));
        }

        // tests to see if the binary search gives a correct true
        @Test

        public void testTrueBinary () {
                int [] array = {5, 10, 15, 20};

                int number = 10;
```

```
                boolean expectedResult = true;

                assertEquals("binarySearch(" + Arrays.toString(array) + number + ")", expectedResult,
Collinear.binarySearch(array, number));
        }

        // tests to see if sort works

        @Test

        public void testSorting () {
                int [] array = { 1, 6, 3, 7, 9, 2};

                int [] expectedResult = {1, 2, 3, 6, 7, 9};
                Collinear.sort(array);
                assertArrayEquals(expectedResult, array);

        }

}
```

Code Snippet 2: Test code used to test the Collinear Class using JUnit 4 (*Algorithms-and-Data-Structures/Collinear/Src at Main · Rnibhriain/Algorithms-and-Data-Structures*, n.d.-b).

Queue Code Used

```
public class Queue {

        final int DEFAULT_SIZE = 10;
        private int tail = -1;
        private Object [] list;

        Queue () {
                create( DEFAULT_SIZE );
        }

        public void create ( int maxNum ) {
                list = new Object[ maxNum ];
                tail = -1;
        }

        public int maxSize () {
                return list.length;
        }

        public void add ( Object newObj ) {

                if ( tail + 1 < maxSize() ) {
                        tail++;

                        list[ tail ] = newObj;
                } else {
                        System.out.print( "Queue is full!" );
                }
        }

        public Object remove () {

                if ( !isEmpty() ) {
```

```
                                Object current = list[ 0 ];
                                for ( int i = 0; i < tail ; i++ ) {
                                        list[ i ] = list[ i + 1 ];
                                }

                                tail--;

                                return current;
                        }

                        System.out.print("Queue was empty!");


                        return null;

                }

                public int howMany () {
                        return tail + 1;
                }

                public Boolean isEmpty () {

                        if ( howMany() == 0   ) {

                                return true;

                        }
                        return false;
                }

}
```

Code Snippet 3: Queue class from previous assignment.

Queue Test Class Used

```
import static org.junit.Assert.*;
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.After;
import org.junit.Before;
import java.io.ByteArrayOutputStream;
import java.io.PrintStream;


public class QueueTest {
        Queue queue;

        @Before
        public void setUp () {
                queue = new Queue();
        }

        @Test
        public void testInitialisation() {
                assertEquals(   "Length is equal to default size", queue.maxSize(), 10 );
                assertEquals( "Make sure there is null returned with remove function", queue.remove(), null );
                assertEquals( "Make sure there is nothing in the list", queue.howMany(), 0 );
                assertEquals( "Make sure list returns empty", queue.isEmpty(), true );

        }
```

```
// This test sets the list max num to 0
@Test
public void testNullList () {
	queue.create( 0 );

	assertEquals(  "Length is equal to 0", queue.maxSize(), 0 );
	assertEquals( "Make sure there is null returned with remove function", queue.remove(), null );
	assertEquals( "Make sure there is nothing in the list", queue.howMany(), 0 );
	assertEquals( "Make sure list returns empty", queue.isEmpty(), true );

}

@Test
public void addToNullList () {
	queue.create( 0 );

	ByteArrayOutputStream outContent = new ByteArrayOutputStream();
	System.setOut(new PrintStream(outContent));

	Object obj = new Object();
	queue.add( obj );

	String expectedOutput  = "Queue is full!" ;

	// Do the actual assertion.
	assertEquals( expectedOutput, outContent.toString() );

}

@Test
public void addToDefaultList () {

	Object obj = new Object();
	queue.add( obj );

	assertEquals( "ensure object added is same as object removed", queue.remove(), obj );

}

@Test
public void testLengthValues () {


	queue.create( 20 );

	assertEquals( "Length is equal to 20", queue.maxSize(), 20 );
	assertEquals( "Make sure there is null returned with remove function", queue.remove(), null );
	assertEquals( "Make sure there is nothing in the list", queue.howMany(), 0 );
	assertEquals( "Make sure list returns empty", queue.isEmpty(), true );
}

@Test
public void testAdding () {
	queue.create( 5 );

	for (int i = 0; i < 5; i++ ) {
		Object obj = new Object();
		queue.add( obj );
		System.out.println("hello");
		assertEquals(  "Make sure list returns correct number of elements", queue.howMany(), i + 1 );
	}

	assertEquals( "Make sure queue does not return empty", queue.isEmpty(), false );
```

```
		Object obj = new Object();
		queue.add( obj );

		ByteArrayOutputStream outContent = new ByteArrayOutputStream();
		System.setOut(new PrintStream(outContent));

		queue.add( obj );

		String expectedOutput  = "Queue is full!" ;

		// Do the actual assertion.
		assertEquals( expectedOutput, outContent.toString() );
	}

	@Test
	public void testRemoving () {
		queue.create( 3 );

		queue.add("1");
		queue.add("2");
		queue.add("3");

		assertEquals( "Make sure first object is removed", queue.remove(), "1" );
		assertEquals(  "make sure correct number in queue", queue.howMany(), 2 );
		assertEquals( "Make sure first object is removed", queue.remove(), "2" );
		assertEquals( "make sure correct number in queue", queue.howMany(),  1 );
		assertEquals( "Make sure first object is removed", queue.remove(), "3" );
		assertEquals( "make sure correct number in queue", queue.howMany(), 0 );
		assertEquals( "Make sure queue is empty", queue.isEmpty(), true );

		ByteArrayOutputStream outContent = new ByteArrayOutputStream();
		System.setOut(new PrintStream(outContent));

		queue.remove();

		String expectedOutput  = "Queue was empty!" ;

		assertEquals(expectedOutput, outContent.toString());

	}

	@Test
	public void differentTypes () {
		queue.create( 3 );

		Object obj = new Object();
		queue.add(obj);
		queue.add("1");
		queue.add(14);

		assertEquals( "Make sure first object is removed", queue.remove(), obj );
		assertEquals( "Make sure first object is removed", queue.remove(), "1" );
		assertEquals( "Make sure first object is removed", queue.remove(), 14  );

	}

                                                @Test
	public void removeAndAdd () {
		queue.create( 3 );

		ByteArrayOutputStream outContent = new ByteArrayOutputStream();
		System.setOut(new PrintStream(outContent));

		assertEquals( "Make sure first object is null", queue.remove(), null  );
```

```
                String expectedOutput  = "Queue was empty!" ;

                assertEquals(expectedOutput, outContent.toString());

                queue.add(1);
                assertEquals( "Make sure first object has been added correctly", queue.remove(), 1 );

        }

        @After
        public void tearDown () {
                queue = null;
        }

}
```

Code Snippet 4: Queue testing class adapted from JUnit 5 to JUnit 4.

Run 1

In this run, I used the Collinear class from code snippets 1 and 2. For the first run of this using the MuJava testing, I used all method operators and class operators to generate the mutants as shown below in Figure 1. I also used all of the tests in the above test class provided in code snippet 2.

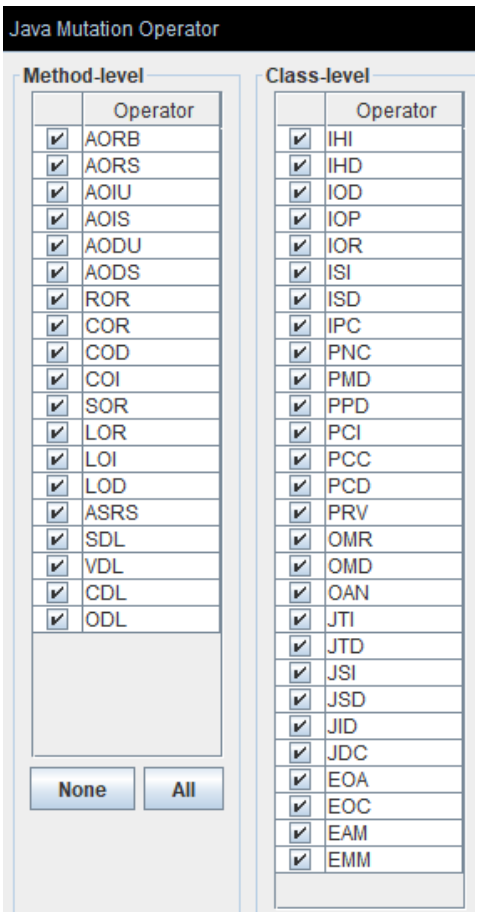

Figure 1: All the mutation operators used in the first run.

After clicking the generate button the generated mutants were displayed in the other two windows. For this particular class, there were no class mutants generated as there were no class variables but there were some traditional mutants as shown in Figure 2 below. There were 547 mutants in total generated.

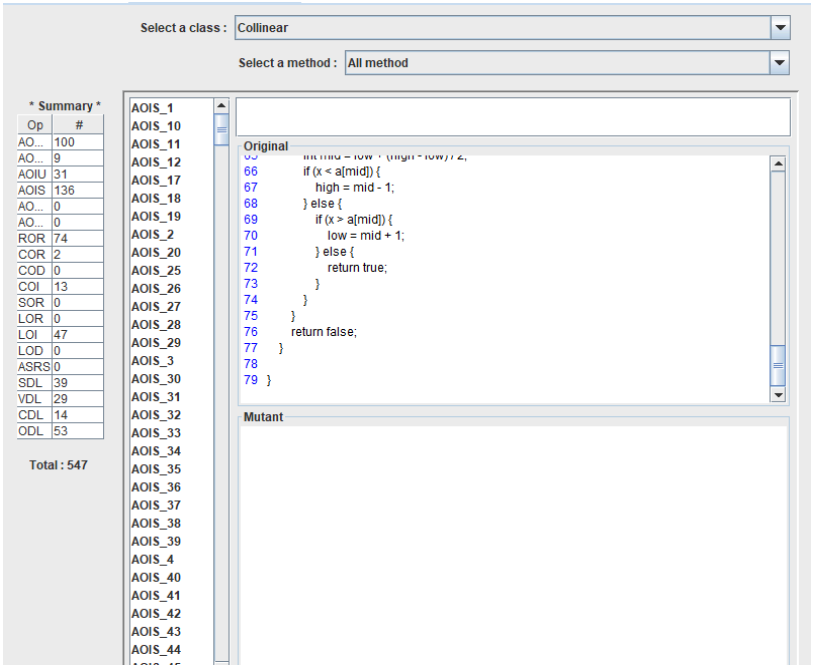

Figure 2: All generated mutants for class Collinear.

Finally, after executing all of the mutants in the test GUI the final score of 85% for the traditional mutants was given in Figure 3 below. There were 82 live mutants left out of all 547 mutants generated as shown in Figure 2.

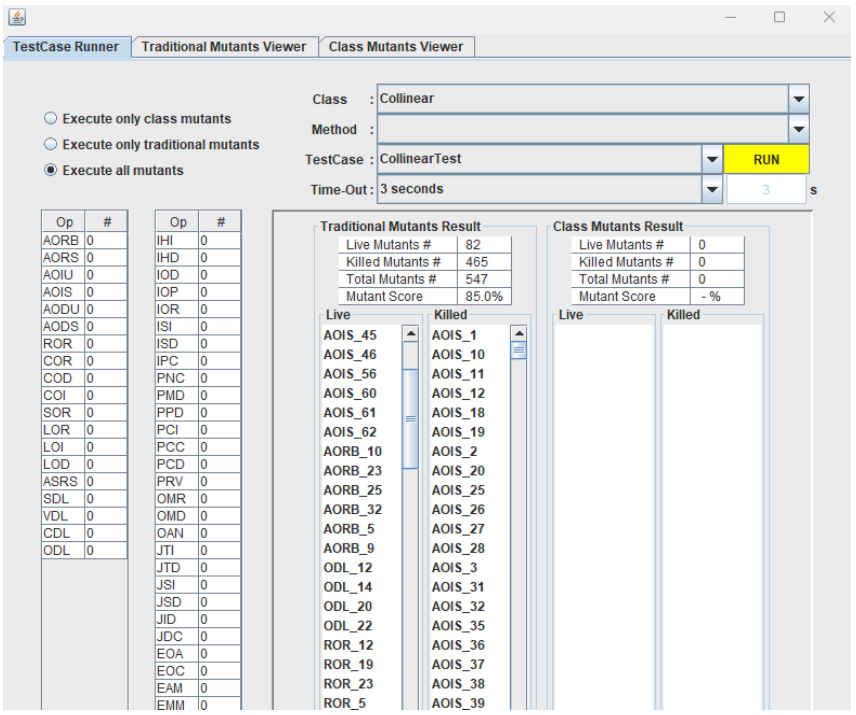

Figure 3: Results from running all the mutants on the collinear test.

Run 2

For this run, I decided to remove all of the test methods other than the constructor test and the testEmpty from the testing class above to see if I would get a different score. After completing all the steps above with all of the mutant options generated. I got a final result of 2% as shown below in figure 4. This score indicates that there were more live mutants that were not killed by the new unit test with less testing that I provided to the MuJava system.

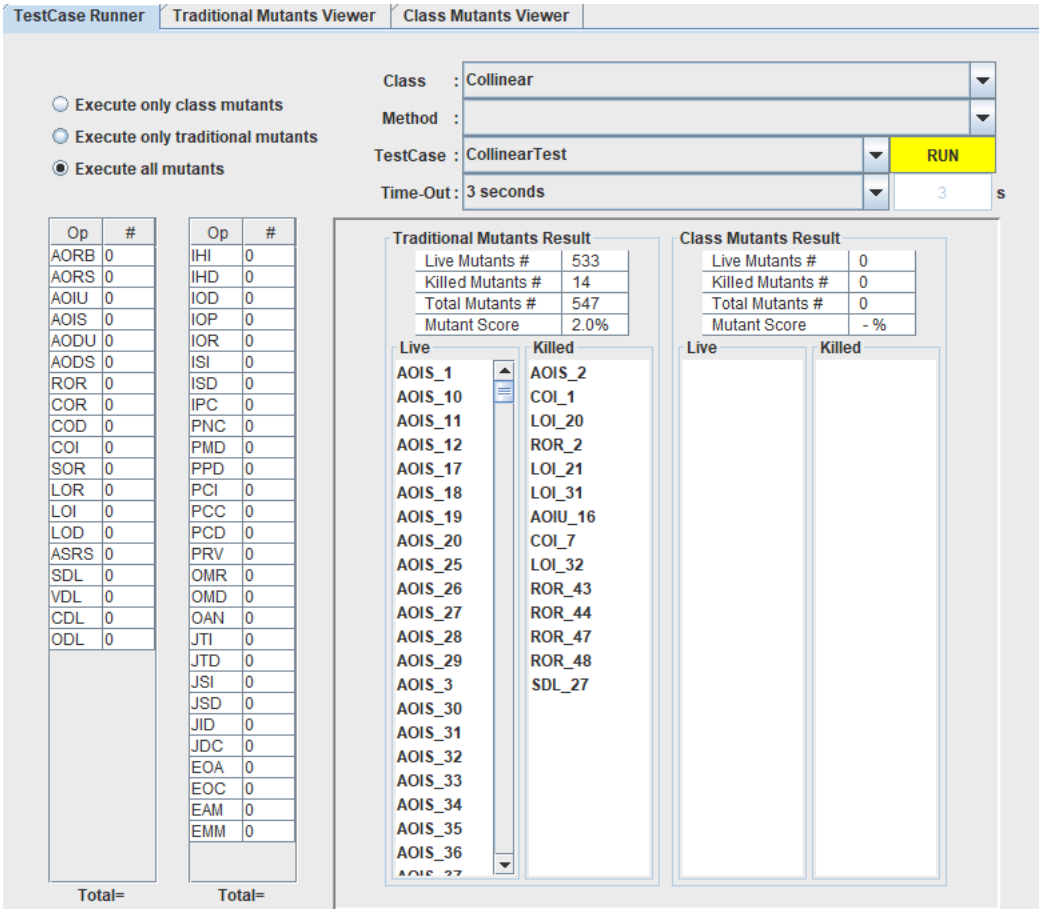

Figure 4: Results after executing the mutants with only two test methods.

Run 3

For the final test, I decided to switch and run the mutation generator on the Queue class as shown above in code snippets 3 and 4 which I used for the previous assignment. Again as before I generated all of the mutant options both traditional and class. Figure 5 below shows the generated mutants for the Queue class where there were both traditional and class mutants generated. In this figure, I show an example of one of the class mutants. There are only five class mutants generated.

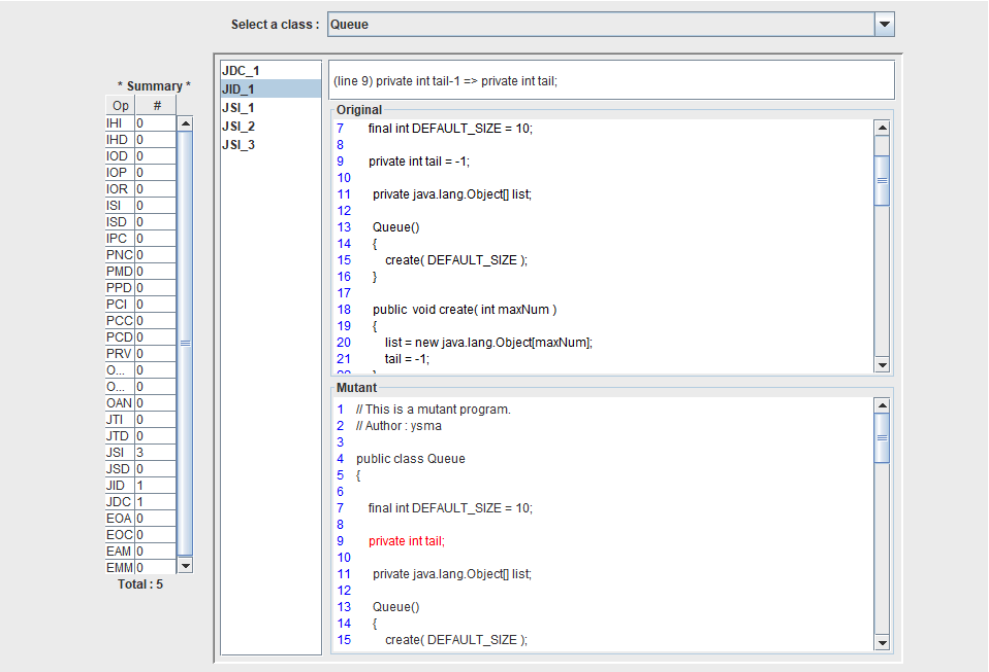

Figure 5: The class mutants generated based on the Queue class.

Figure 6 shows the traditional mutants generated based on the Queue class. There are 117 traditional mutants generated for this class.

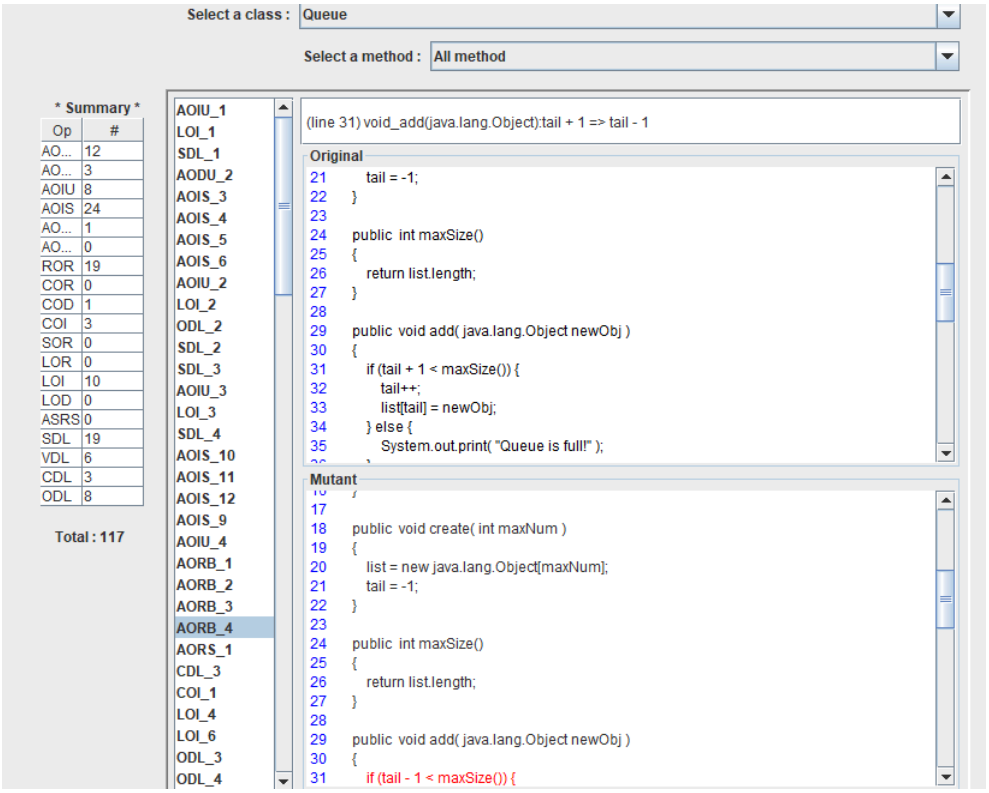

Figure 6: Traditional mutants generated for the Queue class.

Figure 7 below shows the results of the executed mutants for the Queue class. The traditional mutant final score based on the Queue testing class is 94% as there are only 6

live traditional mutants left. The class mutant score was lower and was only 20% as there were 4 out of five mutants live after the execution.

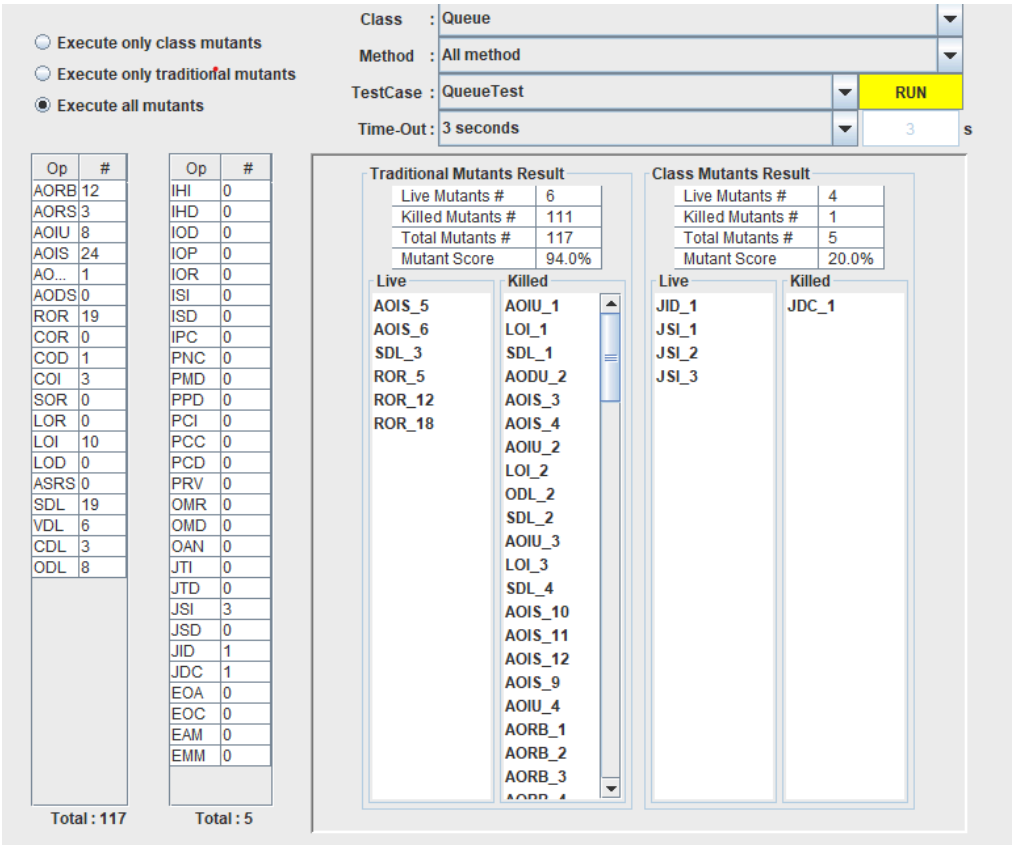

Figure 7: All mutants executed based on the Queue class with a score of 94% for traditional mutants.

Analysis of Coverage Methods

In this section, I will discuss the coverage methods available using the MuJava tool. As mutant testing is a different type of testing in comparison to graph, logic, input space and syntax testing I will do a comparison of mutant testing with each of these coverage methods.

Graph

MuJava was used in (Li et al., 2009) to compare mutant testing against edge-pair, prime path and all-uses graph testing. In this paper, they discover that despite fewer tests being needed by mutation testing it had more requirements. It also ended up discovering a significant number of faults in the classes used to test. Mutant testing also required a lot of extra work to kill the last few mutants. The mutant tests in this paper discovered all of the faults that the graph testing discovered as well as several other faults that they missed. It is worth noting that not all faults were discovered by these coverage methods. In comparison with the other graph testing methods, however, the mutant testing did discover a significant number of faults that the graph testing methods didn't.

Logic

Logic-based testing is probably the most similar testing when compared to mutation testing. Logic-based testing is similar to mutation testing as some of the mutations include changing the logic in statements. The relational operator replacement operator used in MuJava changes the definition of the relational operators to each of the other options as well as the entire expression to true and false. To improve logic-based testing one can use mutation analysis (Kaminski et al., 2013). This paper also claims that even the strongest logic coverage criteria is not as strong as mutation testing. When compared with combinatorial coverage the relational operator replacement operator detects more faults. As combinatorial coverage subsumes all other logic coverage criteria this means that mutation testing also subsumes logic-based testing even those such as MCDC and clause coverage.

Input Space

When using partition testing one separates the input domain into natural classes whose points are the same. It often uses functional testing involving specifications and requires that the division of input is a partition. In (D. Hamlet & R. Taylor, 1990) they explore the viability of partition testing. They also compare partition testing with mutation testing and claim that many types of partition testing involve a division of the input domain that is not proper partitioning as the subdomains overlap. Mutation testing should mean that any grouped inputs should kill the same mutants. In essence, mutation testing indicates all of the inputs that have been tested by the test cases. The mutants that have not been killed indicate the input groups that have not been tested. Mutation testing using MuJava would be a good way of creating partitions in testing.

Syntax

Logic mutation testing is inefficient as often the same mutants are generated more than once and any mutations that have been generated may be killed by a test that kills some other mutant. In (G. K. Kaminski & P. Ammann, 2009) they claim that weak mutation testing is almost as effective as strong mutation testing with a significant computational saving. They aim to reduce mutant set size while increasing detection with predicates in minimal DNF. MuJava does not have mutation operators which cover the LRF and LIF faults in the hierarchy. In general, mutation operators cover the rest of the faults in the Lau and Yu fault hierarchy.

Conclusion

Overall, I think mutation testing is a very good way to find faults in testing and therefore faults in software. When testing code it may be beneficial to use one of the other coverage methods to create tests and then to apply mutation analysis to determine whether all faults have been discovered. While the MuJava software was difficult to set up and is an older piece of software in comparison to other software that is available, it was a very useful method of ensuring that the testing was sufficient for discovering faults. Mutation testing is widely considered to be more expensive when compared to the other testing methods above it is clear that it is a very useful tool in testing to ensure a greater number of faults are

discovered in software. In order to reduce the computation expense there is the option to use selective sets of operators to reduce costs (Kaminski et al., 2013).

References

Algorithms-and-data-structures/Collinear/src at main · rnibhriain/algorithms-and-data-structures. (n.d.-a). GitHub. Retrieved 22 March 2024, from https://github.com/rnibhriain/algorithms-and-data-structures/tree/main/Collinear/src

Algorithms-and-data-structures/Collinear/src at main · rnibhriain/algorithms-and-data-structures. (n.d.-b). GitHub. Retrieved 22 March 2024, from https://github.com/rnibhriain/algorithms-and-data-structures/tree/main/Collinear/src

D. Hamlet & R. Taylor. (1990). Partition testing does not inspire confidence (program testing). *IEEE Transactions on Software Engineering*, *16*(12), 1402–1411. https://doi.org/10.1109/32.62448

G. K. Kaminski & P. Ammann. (2009). Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing. *2009 International Conference on Software Testing Verification and Validation*, 386–395. https://doi.org/10.1109/ICST.2009.13

Kaminski, G., Ammann, P., & Offutt, J. (2013). Improving logic-based testing. *Journal of Systems and Software*, *86*(8), 2002–2012. https://doi.org/10.1016/j.jss.2012.08.024

Li, N., Praphamontripong, U., & Offutt, J. (2009). An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage. In *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009* (p. 229). https://doi.org/10.1109/ICSTW.2009.30

Ma, Y.-S., Offutt, J., & Kwon, Y.-R. (2006). MuJava: A mutation system for java. *Proceedings of the 28th International Conference on Software Engineering*, 827–830. https://doi.org/10.1145/1134285.1134425