

Lab 7 Group 4

Kara King, Dibyajyoti Mukherjee, Eric Weyant, Shane Regel, Andreas Landgrebe
Department of Computer Science
Allegheny College

November 13, 2013

Abstract

A document containing each of the eight deliverables for laboratory 7.

1 Member Contributions and Tool Support

1.1 Member Contributions

Andreas Landgrebe: Printed the output for Deliverable 5. Description of the output made from running the run.sh script for Deliverable 6.

Kara King: Description of the the target in the build.xml and order for mutation for Deliverable 2. Listing of the steps for the Demonstration of MAJOR for Deliverable 8.

Dibyajyoti Mukherjee: Extra Credit to explain how to use MAJOR to perform mutation analysis on ReleasePlanner project to make sure that this demo of MAJOR will work properly. Maintained build.xml file in order to run with the main method and also run MAJOR with ReleasePlanner.

Eric Weyant: Write a main method in Triangle.java for demonstration purposes for Deliverable 3. Print the output from running Triangle.java with the modified main version for Deliverable 4.

Shane Regel: Thoughtful commentary on one live and one killed source code mutant for Deliverable 7. Converted deliverables into LaTeX.

1.2 Tools

Gvim: Gvim was used to create the main method and also analyze the Triangle.java program.

LaTeX(Texmaker): Texmaker was used to compile LaTeX code for each deliverable.

BitBucket: Bitbucket was used via Git commands so we were able to maintain version control throughout the lab.

Major Mutation System: Major was used for mutation analysis of the Triangle project in conjunction with the Ant build system.

Apache Ant: Ant is used with our groups build.xml file to run our entire project alongside in conjunction with other tools such as MAJOR.

2 Description of targets in the build.xml and the order in which they are run

There are many targets within the build.xml file for this project. The first target is clean. This target simply cleans up the files by eliminating the bin directory, which is the directory that holds all of the compiled code, along with the .csv and .log files. The next target is init. This target simply initializes the build for the project by creating the bin directory. The next target is compile. This target compiles all of the source files in the src directory and then puts the compiled versions in the bin directory. This target depends on the init target. Also, this target adapts to use Major's compiler and it must pass the mml script to Major's compiler. It then generates mutants in the projects code and outputs a .log file that shows a description of all the mutants created. The next target is compile.tests. This target compiles all of the source code in the test directory and puts all the compiled versions in the bin directory. This target depends on the compile target. Also, this target executes the javac file executable. The next target is mutation.test. This target is the adapted mutation test target that uses Major's junit task. Essentially, this target is used to run the mutation analysis for the the test cases in our project. The results of this target go into the kill.csv file. This file tells the programmer how many mutants were alive and how many failed after this target is run. The main output from this target being run is put into the results.csv file which shows a summary of the time took to run coverage, how much our tests covered, how many mutants were kills versus how many were not, and a few other details. This target does this since the mutationAnalysis attribute is set to true. The next target is run. This target simply runs the Triangle classes main method and produces the output the main method lays out. The final target is test. This target runs all of the test files and prints out the summary of the tests such as whether they passed, failed, how long they took, and how many tests there were that ran.

The order of commands that are executed during mutation are as followed. The first step is ant -Dmutation="=../mml/tutorial.mml.bin" which sets the path the the mml file we are using. Then, an ant clean command is done in order to get rid of the already existing bin directory from the previous trial run. Next, the build is initialized from the ant init command, however, if one simply runs ant compile or ant compile.tests after a clean, this will build the bin directory. Hence, if those commands are used, the ant init command does not have to be typed out. Once the bin directory is made, or in other words the build is initialized, the ant compile command is run which puts mutants into the code and compiles the src directory. Then, ant compile.tests is ran which compiles all the tests and puts their compile code into the bin directory. Then, during the process, ant test is run to see how good the tests do without the mutation analysis. The final step is running

ant mutation.test which runs the tests using mutation. These commands should be run in this order for mutation to ensure that the major compiler is bound to the java compiler to ensure that mutants are added into the code before the actual testing of the test suite using mutation analysis is done. We must set the path to the mml file before anything is done so the program knows what compiler and jar file to use. Mutation analysis is the last step because the programmer needs to make sure their test suite compiles correctly before they can run mutation analysis on the tests.

3 Modified Main

See attached programs

4 Output from running Triangle.java with main

Listing 1: Main output

```
izfweyante2@aldenv185:~/cs290/lab7/cs290f2013-lab7-team1/example/ant$ ant run
Buildfile: /home/w/weyante2/cs290/lab7/cs290f2013-lab7-team1/example/ant/build.xml

run:
    [java] The type of triangle is EQUILATERAL
    [java] The type of triangle is INVALID
    [java] The type of triangle is INVALID
    [java] The type of triangle is SCALENE
    [java] The type of triangle is ISOSCELES

BUILD SUCCESSFUL
Total time: 1 second
```

5 Output from executing run.sh

```
izfkingk2@aldenv186:~/cs290/lab7/cs290f2013-lab7-team1/
example/ant$ ./run.sh
```

Compiling and mutating project
(ant -Dmutation="=../mml/tutorial.mml.bin" clean compile)

```
Buildfile: /home/k/kingk2/cs290/lab7/cs290f2013-lab7-team1/
example/ant/build.xml
```

clean:

```
[delete] Deleting directory /home/k/kingk2/cs290/lab7/
         cs290f2013-lab7-team1/example/ant/bin
```

init:

```
[mkdir] Created dir: /home/k/kingk2/cs290/lab7/
         cs290f2013-lab7-team1/example/ant/bin
```

compile:

```
[javac] Compiling 1 source file to /home/k/kingk2/cs290/lab7/
         cs290f2013-lab7-team1/example/ant/bin
[javac] #Generated Mutants: 79 (22 ms)
```

BUILD SUCCESSFUL

Total time: 1 second

Compiling tests

(ant compile.tests)

Buildfile: /home/k/kingk2/cs290/lab7/cs290f2013-lab7-team1/example/
ant/build.xml

init:

compile:

compile.tests:

```
[javac] Compiling 1 source file to /home/k/kingk2/cs290/lab7/
         cs290f2013-lab7-team1/example/ant/bin
```

BUILD SUCCESSFUL

Total time: 1 second

Run tests without mutation analysis

(ant test)

Buildfile: /home/k/kingk2/cs290/lab7/cs290f2013-lab7-team1/
example/ant/build.xml

test:

```
[echo] Running unit tests ...
[junit] Running triangle.test.TestSuite
[junit] Testsuite: triangle.test.TestSuite
```

```
[junit] Tests run: 33, Failures: 0, Errors: 0, Time elapsed: 0.01 sec
[junit] Tests run: 33, Failures: 0, Errors: 0, Time elapsed: 0.01 sec
[junit]
[junit] Testcase: test6 took 0.005 sec
[junit] Testcase: test7 took 0.001 sec
[junit] Testcase: test8 took 0 sec
[junit] Testcase: test1 took 0 sec
[junit] Testcase: test24 took 0 sec
[junit] Testcase: test25 took 0 sec
[junit] Testcase: test26 took 0 sec
[junit] Testcase: test27 took 0 sec
[junit] Testcase: test28 took 0 sec
[junit] Testcase: test29 took 0 sec
[junit] Testcase: test30 took 0 sec
[junit] Testcase: test31 took 0 sec
[junit] Testcase: test32 took 0 sec
[junit] Testcase: test33 took 0 sec
[junit] Testcase: test16 took 0 sec
[junit] Testcase: test17 took 0 sec
[junit] Testcase: test18 took 0 sec
[junit] Testcase: test19 took 0 sec
[junit] Testcase: test20 took 0 sec
[junit] Testcase: test21 took 0 sec
[junit] Testcase: test22 took 0 sec
[junit] Testcase: test23 took 0 sec
[junit] Testcase: test4 took 0 sec
[junit] Testcase: test5 took 0 sec
[junit] Testcase: test2 took 0 sec
[junit] Testcase: test3 took 0 sec
[junit] Testcase: test9 took 0 sec
[junit] Testcase: test10 took 0 sec
[junit] Testcase: test11 took 0 sec
[junit] Testcase: test12 took 0 sec
[junit] Testcase: test13 took 0 sec
[junit] Testcase: test14 took 0 sec
[junit] Testcase: test15 took 0 sec
```

BUILD SUCCESSFUL

Total time: 0 seconds

Run tests with mutation analysis
(ant mutation.test)

```
Buildfile: /home/k/kingk2/cs290/lab7/cs290f2013-lab7-team1/
           example/ant/build.xml
```

```
mutation.test:
```

```
[echo] Running mutation analysis ...
[junit] MAJOR: Mutation analysis enabled
[junit] MAJOR: -----
[junit] MAJOR: Run 1 ordered test to verify independency
[junit] MAJOR: -----
[junit] MAJOR: Preprocessing took 0.09 seconds
[junit] MAJOR: -----
[junit] MAJOR: Mutants covered: 79
[junit] MAJOR: -----
[junit] MAJOR: Run mutation analysis with 1 individual test
[junit] MAJOR: -----
[junit] MAJOR: 1/1 - triangle.test.TestSuite$null (14ms / 79):
[junit] MAJOR: 1780 (70 / 79 / 79) -> AVG-RTPM: 22ms
[junit] MAJOR: Mutants killed / live: 70 (70-0-0) / 9
[junit] MAJOR: -----
[junit] MAJOR: Summary:
[junit] MAJOR:
[junit] MAJOR: Total runtime: 1.8 seconds
[junit] MAJOR: Mutation score: 88.61%
[junit] MAJOR: Mutant executions: 79
[junit] MAJOR: Mutants killed / live: 70 (70-0-0) / 9
[junit] MAJOR: -----
[junit] MAJOR: Export runtime results (to results.csv)
[junit] MAJOR: Export mutant kill details (to kill.csv)
```

6 Analysis of Output from run.sh

In the output produced by the run.sh script, all of the four compilations of ant commands built successfully. These four compilation of ant commands are in the run.sh file. On the first part of the run.sh script, the project is being tested if it is compiling and mutating. In this example.

During this first part, the bin directory is being cleared and deleted, then it will create another bin directory where all of the compiled source code is stored. After this, it will compile the source code and check if the source code builds correctly. Then it will generate the mutants and put them into the byte code of the source code along with creating a mutants.log files that gives a representation of all the mutants that were created

and put into the source code. This part also outputs into the terminal that there were 79 mutants made with 21 ms. This first set of commands in the script has a command that also specifies that the compiling process should include mutation using the MML script in our project.

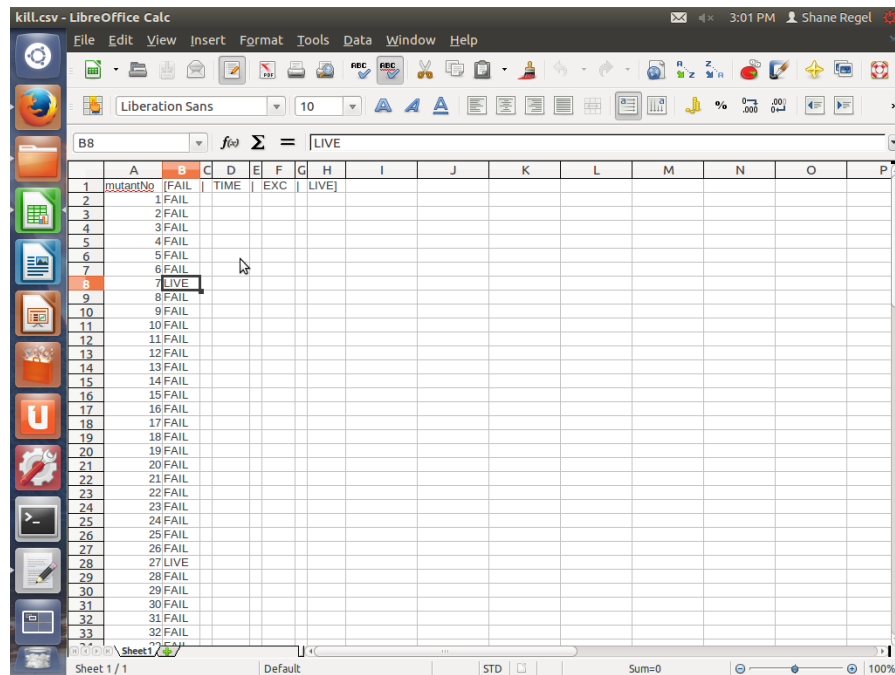
The next set of ant commands in the script file compiles the test source code to see if they build correctly. From the output shown, the test cases were successfully compiled and the build was successful which took 1 second in total. This set of commands puts all of the compiled source code from the tests folder in the bin directory.

After this, the script file will run the tests without mutation analysis, which is the same as running the command ant test. From this, it will now go through the build file and go through each junit test that was written. In the output, each junit test, will display the test cases and show how much time it took for the test cases to be run. At the top of the output from the run.sh file, it shows the number of tests being run, the number of failures, the number of errors, and how long it took for all of the test to be run. At the bottom of the screen, it will show that the build was successful and total time was 0 second.

The last set of ant commands in the script runs the tests with mutation analysis. As shown in the output, the junit testing will have the mutation analysis enabled and this analysis will show a number of key information. The first number will show how many ordered tests will run to verify independency. After this, the mutation analysis will display the amount of time it took to preprocess the mutation analysis, which in our case was 0.09 seconds. After that, the next number being displayed is the number of mutants that is being covered, which was 79. After this, it will display how long it takes to run the test suites, which in our case took 22 milliseconds. Then it will show the total time it has taken so far to run the tests cases along with the number of mutants that were covered in total. From this, it will show the number of mutants that was killed and how many that was currently alive after running the test cases. As shown in the output, there are 70 mutants that are killed and 9 that are currently alive in the source code. After this, it displays the total summary from running the test cases using the mutation analysis. This summary includes the total run time with the mutation analysis, the percentage of mutants that were killed over the amount that were still alive (the mutation score in this case was 88.61 percent). After this, the result of the test cases will now export the runtime results to results.csv and the mutant kill details to kill.csv.

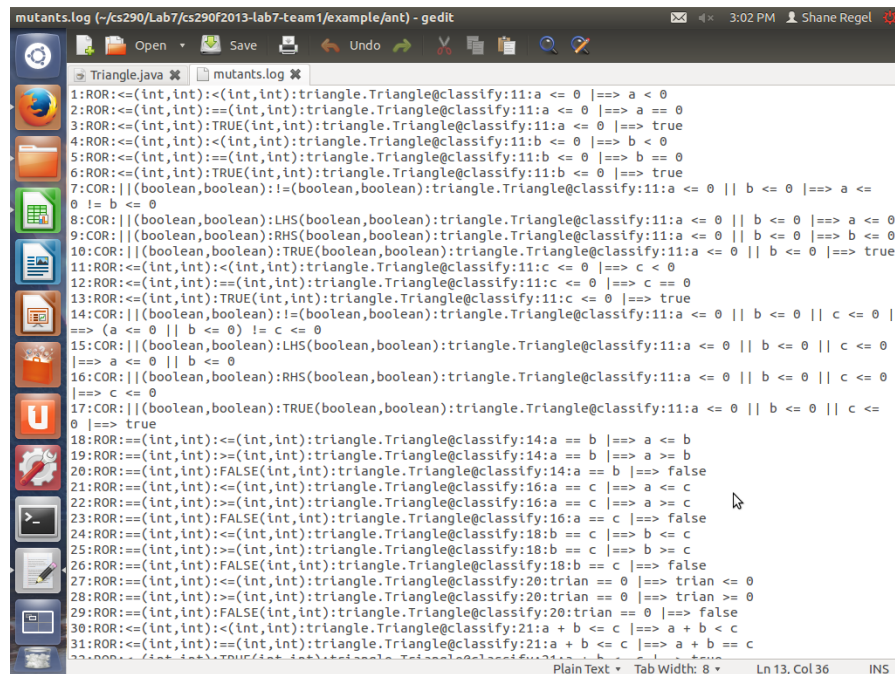
7 Live vs Killed Commentary

The kill.csv file provides information on whether or not the mutant was killed by the test cases. A mutant with the FAIL annotation is considered killed while one with the LIVE annotation remained alive after running the test cases. Figure 1 provides an example



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	mutantNo	I	TIME		EXC											
2	1	FAIL														
3	2	FAIL														
4	3	FAIL														
5	4	FAIL														
6	5	FAIL														
7	6	FAIL														
8	7	LIVE														
9	8	FAIL														
10	9	FAIL														
11	10	FAIL														
12	11	FAIL														
13	12	FAIL														
14	13	FAIL														
15	14	FAIL														
16	15	FAIL														
17	16	FAIL														
18	17	FAIL														
19	18	FAIL														
20	19	FAIL														
21	20	FAIL														
22	21	FAIL														
23	22	FAIL														
24	23	FAIL														
25	24	FAIL														
26	25	FAIL														
27	26	FAIL														
28	27	LIVE														
29	28	FAIL														
30	29	FAIL														
31	30	FAIL														
32	31	FAIL														
33	32	FAIL														

Figure 1: kill.csv File



```
1:ROR:<=(int,int):<(int,int):triangle.Triangle@classify:11:a <= 0 |==> a < 0
2:ROR:<=(int,int):==(int,int):triangle.Triangle@classify:11:a <= 0 |==> a == 0
3:ROR:<=(int,int):TRUE(int,int):triangle.Triangle@classify:11:a <= 0 |==> true
4:ROR:<=(int,int):<(int,int):triangle.Triangle@classify:11:b <= 0 |==> b < 0
5:ROR:<=(int,int):==(int,int):triangle.Triangle@classify:11:b <= 0 |==> b == 0
6:ROR:<=(int,int):TRUE(int,int):triangle.Triangle@classify:11:b <= 0 |==> true
7:COR:|| (boolean,boolean):!=(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 |==> a <= 0 != b <= 0
8:COR:|| (boolean,boolean):LHS(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 |==> a <= 0
9:COR:|| (boolean,boolean):RHS(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 |==> b <= 0
10:COR:|| (boolean,boolean):TRUE(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 |==> true
11:ROR:<=(int,int):<(int,int):triangle.Triangle@classify:11:c <= 0 |==> c < 0
12:ROR:<=(int,int):==(int,int):triangle.Triangle@classify:11:c <= 0 |==> c == 0
13:ROR:<=(int,int):TRUE(int,int):triangle.Triangle@classify:11:c <= 0 |==> true
14:COR:|| (boolean,boolean):!=(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 || c <= 0 |==> (a <= 0 || b <= 0) != c <= 0
15:COR:|| (boolean,boolean):LHS(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 || c <= 0 |==> a <= 0 || b <= 0
16:COR:|| (boolean,boolean):RHS(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 || c <= 0 |==> c <= 0
17:COR:|| (boolean,boolean):TRUE(boolean,boolean):triangle.Triangle@classify:11:a <= 0 || b <= 0 || c <= 0 |==> true
18:ROR:<=(int,int):<=(int,int):triangle.Triangle@classify:14:a == b |==> a <= b
19:ROR:<=(int,int):>=(int,int):triangle.Triangle@classify:14:a == b |==> a >= b
20:ROR:<=(int,int):FALSE(int,int):triangle.Triangle@classify:14:a == b |==> false
21:ROR:<=(int,int):<=(int,int):triangle.Triangle@classify:16:a == c |==> a <= c
22:ROR:<=(int,int):>=(int,int):triangle.Triangle@classify:16:a == c |==> a >= c
23:ROR:<=(int,int):FALSE(int,int):triangle.Triangle@classify:16:a == c |==> false
24:ROR:<=(int,int):<=(int,int):triangle.Triangle@classify:18:b == c |==> b <= c
25:ROR:<=(int,int):>=(int,int):triangle.Triangle@classify:18:b == c |==> b >= c
26:ROR:<=(int,int):FALSE(int,int):triangle.Triangle@classify:18:b == c |==> false
27:ROR:<=(int,int):<=(int,int):triangle.Triangle@classify:20:trian == 0 |==> trian <= 0
28:ROR:<=(int,int):>=(int,int):triangle.Triangle@classify:20:trian == 0 |==> trian >= 0
29:ROR:<=(int,int):FALSE(int,int):triangle.Triangle@classify:20:trian == 0 |==> false
30:ROR:<=(int,int):<=(int,int):triangle.Triangle@classify:21:a + b <= c |==> a + b <= c
31:ROR:<=(int,int):>=(int,int):triangle.Triangle@classify:21:a + b <= c |==> a + b <= c
32:ROR:<=(int,int):TRUE(int,int):triangle.Triangle@classify:21:a + b <= c |==> true
```

Figure 2: mutants.log File

where mutant 6 was killed while mutant 7 remains alive. The log file, demonstrated in Figure 2, provides detailed information on each generated and embedded mutant. The file contains seven columns separated by a :, each of which represents a different piece of information on the mutant. The columns, in order, provide information on:

1. Mutants unique number (id)
2. Name of the applied mutation operator
3. Original operator symbol
4. Replacement operator symbol
5. Fully qualified name of the mutated method
6. Line number in the original source file
7. Visualization of the applied transformation

For example mutant 6 (shown as two lines):

```
6:ROR:<=(int , int ):TRUE(int , int ): triangle . Triangle@classify :  
11:b <= 0 |==> true
```

The mutant id is 6 and the mutation operator is ROR, which stands for relational operator replacement. The original operator was (int,int) and has been changed to TRUE(int,int). This mutation occurs in the classify method of Triangle on line 11. The operation `b <= 0` has been replaced with `true`. This mutation was killed so the test cases were able to identify the replaced relational operator.

This mutation can be detected with a psuedo test case:

```
b = 2;  
expected = if (b>0) => false  
           else => true  
actual = return value from method  
check (expected == actual)
```

The psuedo code goes through the logic of a test case that finds the mutation. Initially `b` is set to 2. Then the case checks if `b` is greater than 0, which 2 is, so `expected` would be set to false. Actual finds the return value of the method in question. The mutation sets this value to true. Then the `expected` and `actual` values are compared, since they are not equal the test case fails and discovers the mutant.

The test cases were not able to identify mutant 7 (shown as two lines):

```
7:COR:||( boolean , boolean ) : != ( boolean , boolean ) :
```

```
triangle . Triangle @ classify : 11 : a <= 0 || b <= 0 ==> a <= 0 != b <= 0
```

The id number is seven and the mutation operator is COR, which stands for conditional operator replacement. The original operator was (boolean,boolean) but has been changed to !=(boolean,boolean). This mutation occurs in the classify method of Triangle on line 11. The mutation changed the source code from $a \leq 0 \parallel b \leq 0$ (a less than or equal to zero and b less than or equal to zero) to $a \leq 0 \neq b \leq 0$ (a less than or equal to zero not equal to b less than or equal to zero). The \parallel (or) is changed to \neq (not equal to).

The test cases do not catch this error because the two statements are semantically equivalent. The difference between the two statements though can be seen in Truth Tables 1 and 2.

	\parallel	
	T	F
T	T	T
F	T	F

Table 1: OR

	\neq	
	T	F
T	F	T
F	T	F

Table 2: NOT EQUAL

The instance that would pass the test case but be incorrect would be the true true instance for the \neq truth table. Specifically it avoids the statement:

```
if ( a <= 0 || b <= 0 || c <= 0 )
    return Type.INVALID;
```

The mutant changes the statement to:

```
if ( a <= 0 != b <= 0 || c <= 0 )
    return Type.INVALID;
```

In this case with the input (0, 0, 4) $a = 0$ and $b = 0$ so both statements evaluate to true, however since they are joined by a \neq they evaluate to false. Since $c = 4$ which is \leq

0 the entire if returns false and the return invalid is never invoked. The issue is, (0, 0, 4) would not print out that the triangle is invalid due to having a side of 0. The error is eventually discovered in the code with the statement:

```
if ( a + b <= c || a + c <= b || b + c <= a )  
    return Type.INVALID;
```

Since $0+0 \leq 4$ the statement returns true and the return invalid is invoked.

8 Steps For Demonstration

1. Go into correct directory `../lab7/cs290f2013-lab7-team1/example/ant`
2. Run the command `./run.sh`
3. Explain the commands that `run.sh` file calls and in the order they are called.
 - (a) `ant -Dmutation="=../mml/tutorial.mml.bin" clean compile`
 - (b) `ant compile.tests`
 - (c) `ant test`
 - (d) `ant mutation.test`
4. Explain what each command produced
 - (a) created bin directory and compiled the source code and put them into the bin directory along with adding in 79 mutants and a .log file of all the mutants.
 - (b) compiles all of the tests and puts them into the bin directory
 - (c) This runs all of the tests and outputs how many tests ran, the amount that failed and errors that occurred. It also shows the total time it took for the tests along with the time it took for each test to complete.
 - (d) Runs the mutation analysis on the project and outputs the results into the terminal along with creating a `results.csv` and `kill.csv` file
5. Go into the result files along with the terminal output and explain what each means.