1. Introduction to EVOSUITE and SUT

Software testing is inevitable to search for defects in a software system. Test cases are needed that will execute the software systematically, and assessment of the correctness of the observed behavior while running the test cases are crucial in software development. EvoSuite is a perfect solution to solve this issue. EvoSuite is a tool that generates test cases automatically with assertions for classes written in Java programming language. EvoSuite achieves this by applying a hybrid approach that generates and optimizes the whole test suite towards satisfying a coverage criterion.

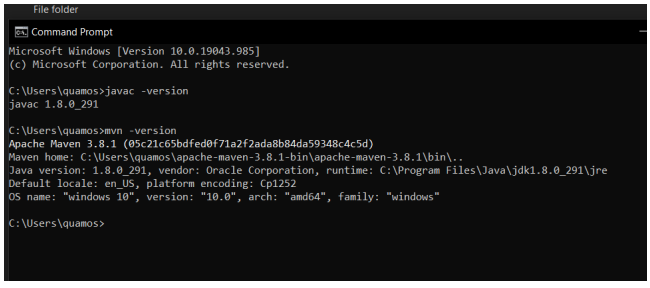
EvoSuite has several strengths and functions. It can be used to generate JUnit 4 tests for selected classes. It enables optimization of various coverage criteria like branches, outputs and mutation testing. Moreover, the tests are minimized and the tests that contribute to achieve coverage are retained. Also the tests run in a sandbox, thus preventing dangerous operations. Few other strengths include the virtual file system and virtual network of EvoSuite.

The Software Under Test(SUT) for this assignment will be a Calculator class that has been written in java code. The Calculator class serves as a basic calculator consisting of basic operations like addition, subtraction, multiplication and division. These are defined as methods(sum, minus, multiply, divide) inside the Calculator class. We also have a CalculatorTest class that serves as a test class for the Calculator class.

2. EVOSUITE Application Process

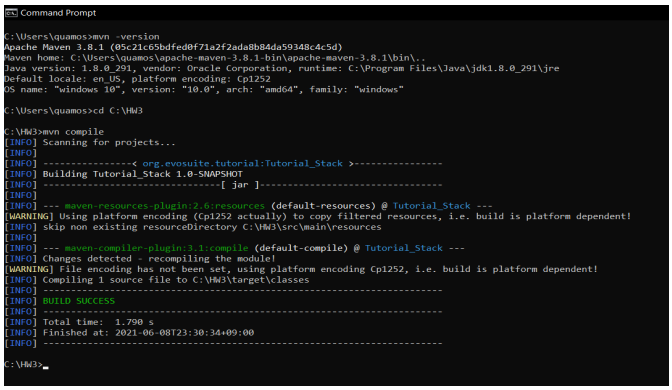
In this section, the steps that have been used to invoke EVOSUITE on my SUT will be briefly mentioned. Relevant screenshots are also added, to provide more clarifications about the output.

- We begin with opening the command prompt using and checking if we have JDK and maven installed. The following two snippets show the output obtained, indicating that we indeed have JDK and Maven in our system.
    - **javac -version**
    - **mvn -version**



- Then I changed my directory so that I am in the folder containing my class under test and the relevant files.
    - **cd C:\HW3**

- Then we invoke maven for compiling the project, and as a result, a file is produced which contains the bytecode of our Calculator class file
    - **mvn compile**



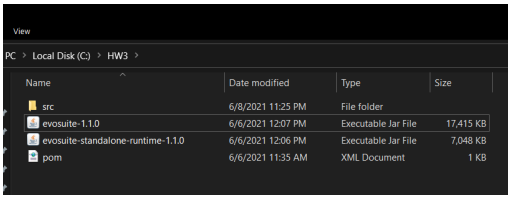


Fig: Before compiling

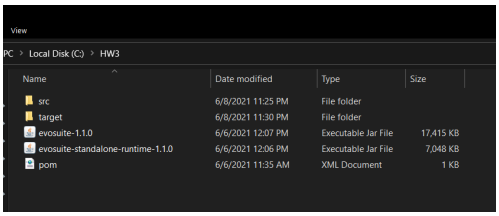
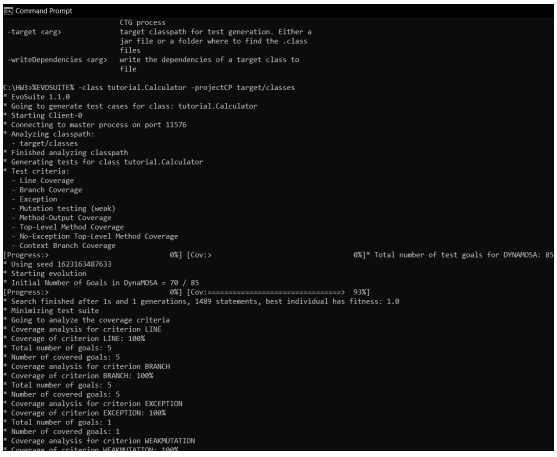


Fig: After compiling

- Since I have already downloaded the latest version of EVOSUITE, the next step is to execute the .jar file and create an environment variable to point to EvoSuite, so that Evosuite can be invoked much more easily.
    - **java -jar evosuite-1.1.0.jar**
    - **set EVOSUITE=java -jar "%CD%"\evosuite-1.1.0.jar**
    - **%EVOSUITE%** -> this is the new method to invoke evosuite

- Now we move on to generate tests and for that, we need to invoke EvoSuite on our example class. We need to provide 2 pieces of information to EvoSuite: The class under test and the classpath where the bytecode of the class under test and dependencies can be found.
  Then we run EvoSuite
    - **%EVOSUITE% -class tutorial.Calculator -projectCP target/classes**



We see that 2 new files have been created in a folder “evosuite-tests” in the same directory

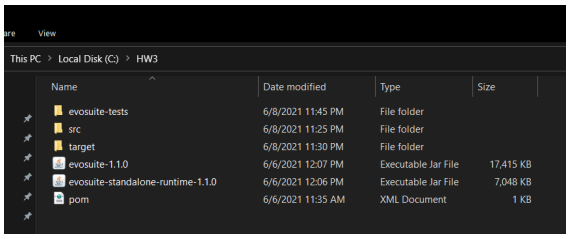


Fig: Creation of new folder

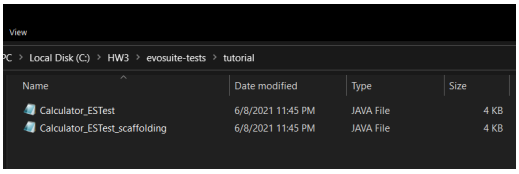
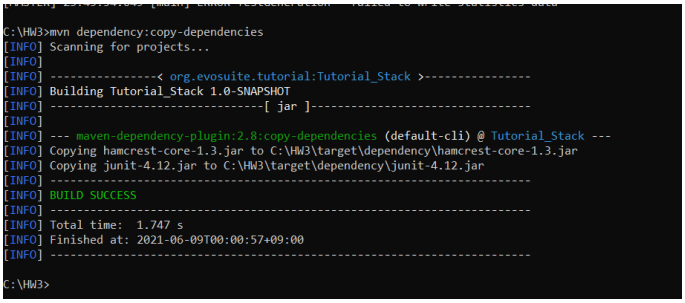
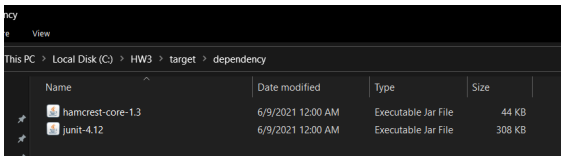


Fig: Creation of 2 new files

- Now we will need to compile the tests and for this, we will first need JUnit and Hamcrest dependencies which we obtain using Maven.
    - **mvn dependency:copy-dependencies**



Now we will have the following 2 .jar files in the dependency folder as shown below:



Now we need to tell the Java compiler where to find all these things, for which we set the CLASSPATH environment variable:

- **set CLASSPATH=target/classes;evosuite-standalone-runtime-1.1.0.jar;evosuite-tests;target/dependency/junit-4.12.jar;target/dependency/hamcrest-core-1.3.jar**

Now we compile the tests in place using the following command:

- **javac evosuite-tests/tutorial/*.java**

We will see that two new .class files have been created

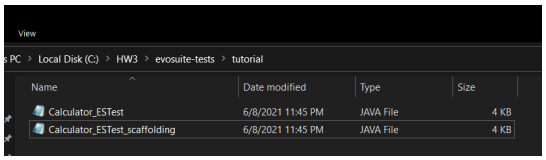


Fig: Before

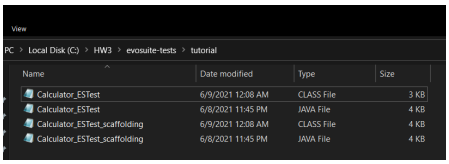


Fig: After

- Since the compilation was successful we can now run the test using the command
  - **java org.junit.runner.JUnitCore tutorial.Calculator_ESTest**

As we see in the snippet below, we have generated and executed the EvoSuite test suite.

```
C:\HW3>CLASSPATH=target/classes;evosuite-standalone-runtime-1.1.0.jar;evosuite-tests;target/dependency/junit-4.12.jar;target/dependency/hamcrest-core-1.3.jar
'CLASSPATH' is not recognized as an internal or external command,
operable program or batch file.

C:\HW3>set CLASSPATH=target/classes;evosuite-standalone-runtime-1.1.0.jar;evosuite-tests;target/dependency/junit-4.12.jar;target/dependency/hamcrest-core-1.3.jar

C:\HW3>javac evosuite-tests/tutorial/*.java

C:\HW3>java org.junit.runner.JUnitCore tutorial.Calculator_ESTest
JUnit version 4.12
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
.............
Time: 1.905

OK (13 tests)


C:\HW3>
```

- Now we will work with some Existing tests. We already have a file “CalculatorTest.java” which is a test suite with 4 tests inside including basic calculator operations like summation, subtraction, and division(It doesn’t have the operation multiplication). The snippet of the file is given below:

CalculatorTest - Notepad

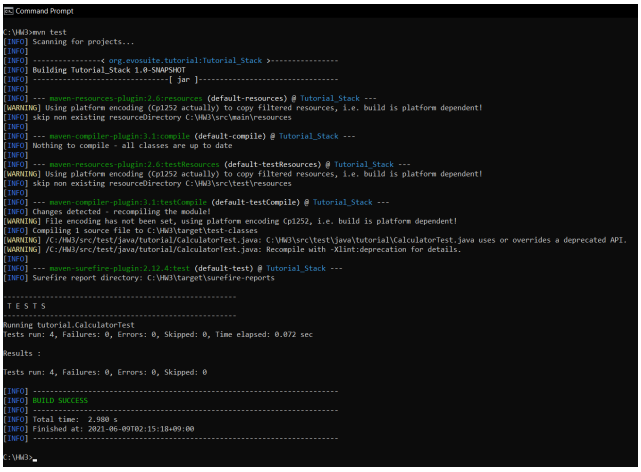
File Edit Format View Help

```
package tutorial;

import junit.framework.Assert;
import org.junit.Test;

public class CalculatorTest {

	@Test
	public void testSum() {
		// Given
		Calculator calculator = new Calculator();
		// When
		int result = calculator.sum(2, 2);
		// Then
		if (result != 4) {   // if 2 + 2 != 4
			Assert.fail();
		}
	}

	@Test
	public void testMinus() {
		Calculator calculator = new Calculator();
		Assert.assertEquals(0, calculator.minus(2, 2));
	}

	@Test
	public void testDivide() {
		Calculator calculator = new Calculator();
		Assert.assertEquals(2, calculator.divide(6, 3));
	}


	@Test(expected = ArithmeticException.class)
	public void testDivideWillThrowExceptionWhenDivideOnZero() {
		Calculator calculator = new Calculator();
		calculator.divide(6, 0);
	}
}
```

Fig: CalculatorTest.java

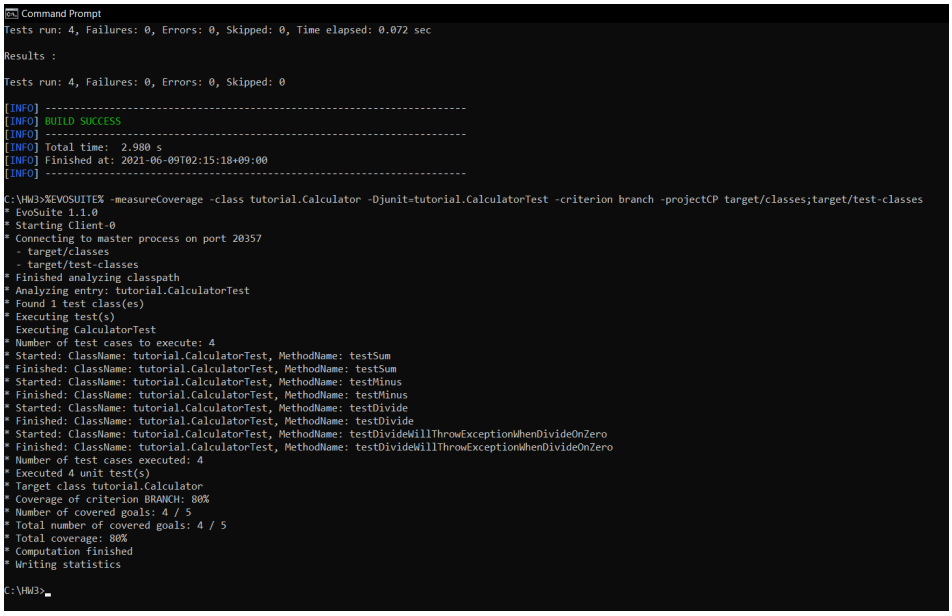
- Now we compile the test using Maven
  - **mvn test**

Then we can see that the following output is produced and it indicates that there were indeed 4 tests and it also tells us that all ran without any failures or errors.



- The bytecode of this test is also placed in the directory by Maven, and we can judge how good this suite is by using EvoSuite to measure the coverage, by using the following command
    - **%EVOSUITE% -measureCoverage -class tutorial.Calculator -Djunit=tutorial.CalculatorTest -criterion branch -projectCP target/classes;target/test-classes**

From the screenshot below, we see that **80 percent** has been covered, and it was expected because the “CalculatorTest.java” file didn’t cover the multiplication operation and so it couldn’t be tested.



- To have a test that covers the remaining one branch we need to invoke EvoSuite using the following command:

- **%EVOSUITE% -class tutorial.Calculator -Djunit=tutorial.CalculatorTest -projectCP target/classes;target/test-classes -criterion branch**

Now we check the file “Calculator_ESTest.java” and we see that a test is present there which tests the multiplication operation, and hence we will be able to cover all the operations now. The snippet of the file is given below:

` **Fig:Updated Calculator_ESTest.java**

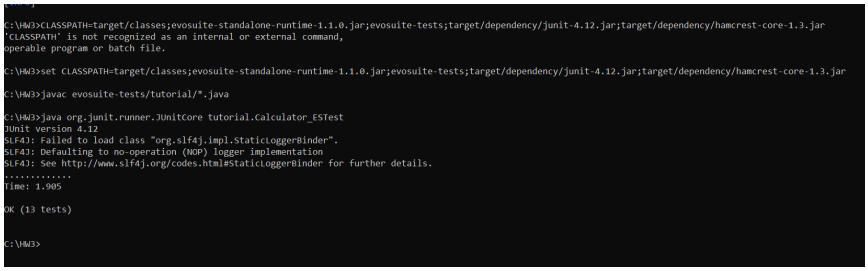
Thus we have been able to generate automated tests for “Calculator.java” and also we have worked with existing test suites to see the code coverage and also have been able to cover the remaining branches using EvoSuite.

3. Testing Result Analysis

We have already discussed how to use EvoSuite on our system and have also provided the outputs as snippets. In this section, we will discuss the outputs and analyze them.

- **# of tests Analysis and Test Case Analysis**

First of all, we invoked EvoSuite on the “Calculator.java” file, which resulted in the creation of 2 new files in a folder “evosuite-tests”. Then after compiling the files and running the tests on the command line we got the following output:

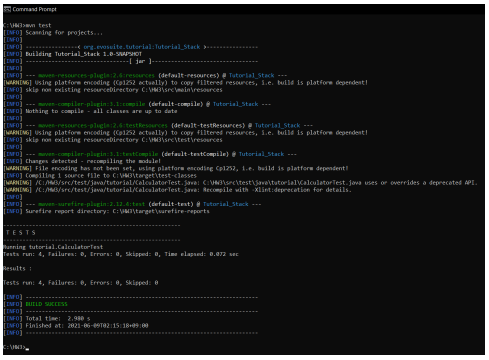


This tells us that there are 13 tests in the “Calculator_ESTest.java” file which is absolutely true because the file does contain 13 tests(The screenshot is given below). Hence The test result is consistent and **EvoSuite has generated 13 unit tests** for our “Calculator.java” file

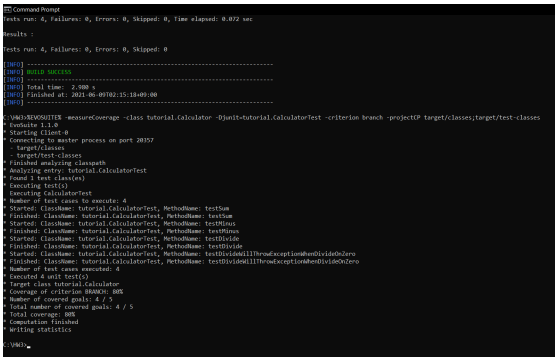
Fig: Calculator_ESTest.java

- **<u>Code Coverage Analysis:</u>**

Now we move onto the “CalculatorTest.java” which as mentioned before is a test suite for “Calculator.java” containing 4 tests(screenshot has been attached in Part 2). Since the file does not implement the multiplication operation we assume that the code coverage will not be 100 percent. After compiling and passing the tests using Maven we got the following output.



This is consistent with our statement that the file does contain 4 tests, and it is also evident that all of the tests pass without failures and errors. After this, we advance to analyze the code coverage of this test suite. After running the required commands the following output was obtained:



This tells us that the code coverage was 80 percent, and 4 out of the total 5 goals have been covered. This is consistent with our assumption that the coverage won't be 100 percent, which is obvious because the “CalclulatorTest.java” file did not implement the multiplication operation and so all the goals couldn’t be covered.

We have already mentioned in Part 2 how to use EvoSuite to cover the remaining one branch and after executing the commands we get a new updated “Calculator_ESTest.java” file containing the tests that have been missed(in our case, the multiplication operation).

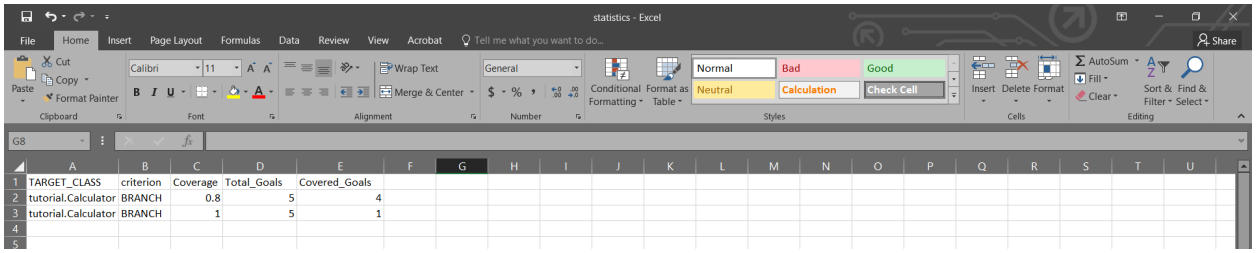
Calculator_ESTest - Notepad

File Edit Format View Help

```
/*
 * This file was automatically generated by EvoSuite
 * Tue Jun 08 17:22:38 GMT 2021
 */

package tutorial;

import org.junit.Test;
import static org.junit.Assert.*;
import org.evosuite.runtime.EvoRunner;
import org.evosuite.runtime.EvoRunnerParameters;
import org.junit.runner.RunWith;
import tutorial.Calculator;

@RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism = true, useVFS = true, useVNET = true, resetStaticState = true, separateClassLoader = true)
public class Calculator_ESTest extends Calculator_ESTest_scaffolding {

  @Test(timeout = 4000)
  public void test0()  throws Throwable  {
      Calculator calculator0 = new Calculator();
      int int0 = calculator0.multiply((-726), (-726));
      assertEquals(527076, int0);
  }
}
```

Fig: Updated Calculator_ESTest.java

Attached below is the overall Statistics file summarizing the coverages. It can be seen that the first row has a coverage of 0.8 whereas the second row has a coverage of 1. This is consistent with what we achieved before because we know that at first the multiplication operation was missed in the “CalculatorTest.java” file, which is why all the goals couldn’t be achieved. Later when we used EvoSuite to cover the remaining branch all the goals had been achieved and so the coverage became 1.



	A	B	C	D	E
1	TARGET_CLASS	criterion	Coverage	Total_Goals	Covered_Goals
2	tutorial.Calculator	BRANCH	0.8	5	4
3	tutorial.Calculator	BRANCH	1	5	1
4					
5					

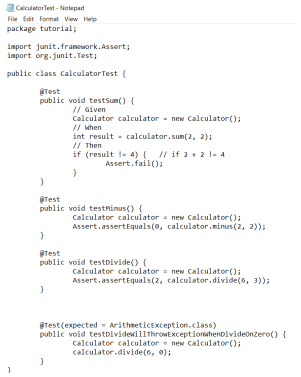
Fig: Statistics

- **<u>PASS/FAIL analysis:</u>**

Finally, we move on to discuss the Pass/Fail results of the test cases and also modify the “CalculatorTest.java” file to deliberately cause some errors. First, we will work with the original “CalculatorTest.java” file and explain why all the test cases passed. As seen in the screenshots below, the original file contains the following commands and outputs:

- Operation: 2+2, expected result: 4
- Operation: 2-2, expected result: 0
- Operation: 6/3, expected result: 2
- Operation: 6/0, expected result: raise exception when divide by zero

Hence it is clear that all of the results are consistent with the operations and hence we have all test cases as PASS.



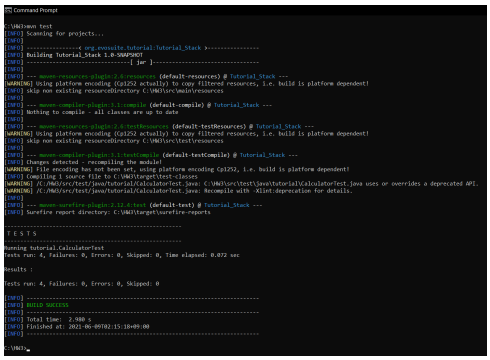
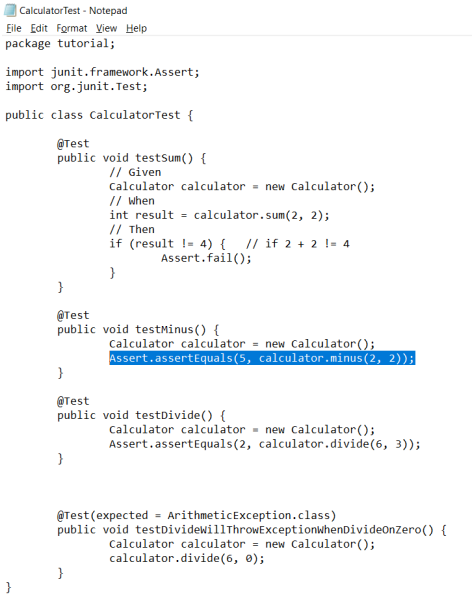


Fig: Original CalculatorTest.java **Fig: Test result**

Now let's intentionally change the codes to cause some errors in the CalculatorTest.java file. As seen in the screenshots below only one change was made in testMinus:

- Operation 2-2, expected result: 5

We already know that it is wrong, because 2-2 =0 and not 5, So we expect one test case to Fail and as seen from the second screenshot below, our assumption was correct, we have 3 test cases that passed and one test case that failed. The command line also tells us explicitly that the error was in testMinus. Hence, we showed how we can modify the test file and cause test cases to fail.



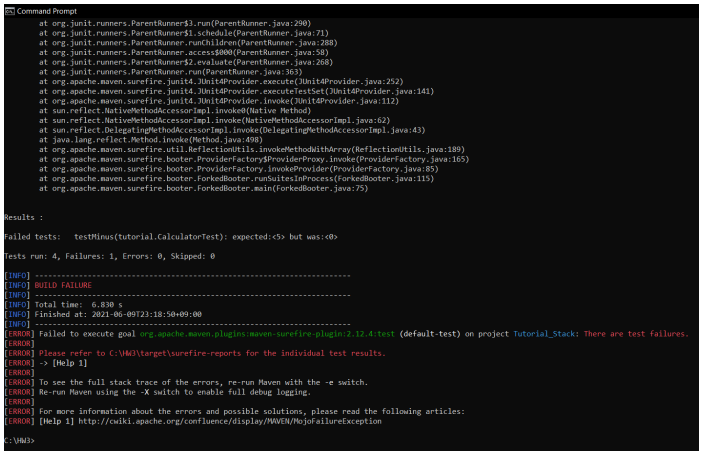


Fig: modified CalculatorTest.java **Fig: Test result**

4. Personal Evaluation

In this section, I will provide my personal evaluation of the EvoSuite tool. To be honest, when I first started using it, I was a bit confused about how it worked and just followed the tutorials, but after trying it for around 4 to 5 hours, I slowly started to understand how it works. And after I used it for a few days, I found it very easy to use and as a result could run commands at a faster pace. Hence I can say that if someone spends some time with it he will be able to figure it out, and so the usability is very good. In terms of efficiency, the command outputs were quick and didn't take much time to process and so it was not bothersome for me. And finally, to talk about the effectiveness, we saw how EvoSuite gave correct test results when we changed the files and introduced some errors intentionally and how EvoSuite could catch them.

So summing up, I was very pleased and satisfied with the EvoSuite tool. Despite being a novice in this field, I didn't take a long time to learn it and its functions and invoke it with my project.