Deliverable 3 Report

Wai Lok Cheng, Jhenna Foronda, Sungsu Park

Evaluation of Black-box and White-box testing

Their black-box testing was robust and strong, so I feel like we don't need to add anything to their test. They did write out all 140 tests for their check. Their test cases have successfully proven the motivation for strong equivalence test cases in the way that there is a sense of complete testing and covering different classes definitely avoided redundancy. Their reasoning for utilizing Equivalence Class Technique was sound. I completely agree with their decision. They've covered all possible outcomes with all the classes they've found for operator and operand.

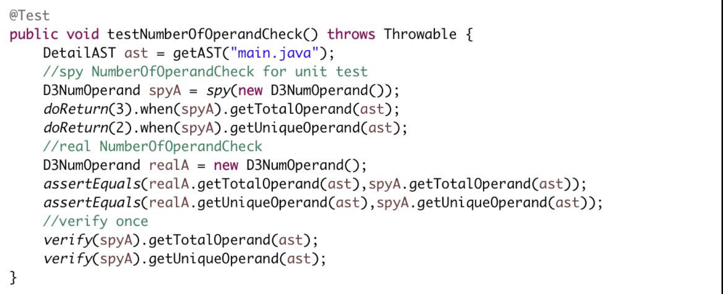
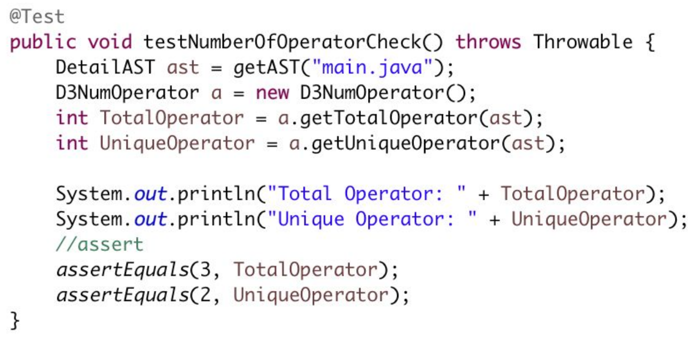
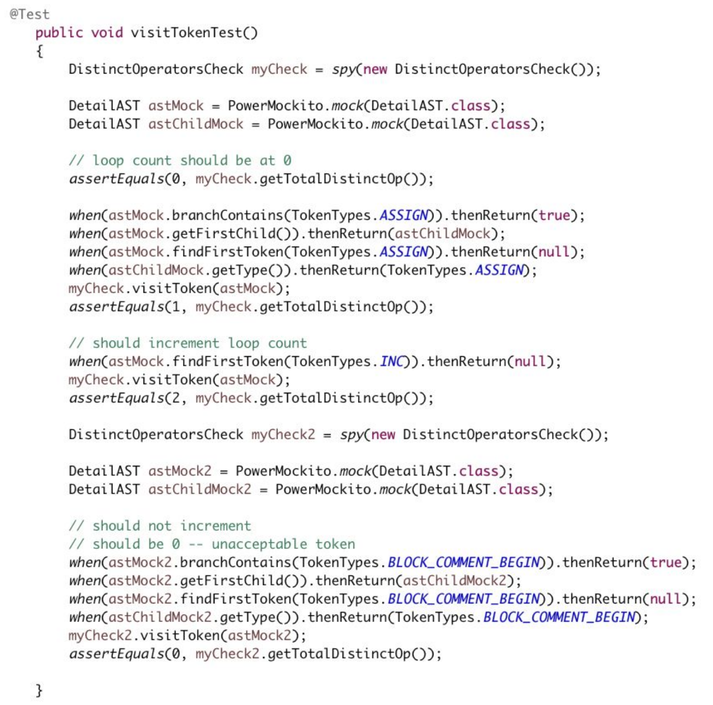
As for their white-box testing, this group did a really good job. Their control-flow diagram is correct. The c-uses and p-uses genuinely show the way the data is accorded to the way it's manipulated in their program. I believe their black-box was thorough enough to have full code coverage. Although they do have a decent amount of code coverage, it does not necessarily mean they have quality test cases. This is where mutation testing comes in.

After running the code on PitEclipse, there were several stretch cases that were not covered. Since the test is always incrementing per operator/operand found, the mutation that happened was replacing the integer addition with subtraction; that came back with no coverage. Similar to that, there was also no coverage for when we replace integer addition with subtraction.

Evaluation of unit test and integration test

<u>NumberOfOperandCheck.java and NumberOfOperatorCheck.java</u>

This group's unit and integration testing could be stronger. I added test cases where I check the count of operators before an acceptable token is found, after an acceptable token is found, and after an unacceptable token is found.

Original Test Cases	Added Test Cases
 	

Original Test Cases:

```
@Test
public void testNumberOfOperandCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    //spy NumberOfOperandCheck for unit test
    D3NumOperand spyA = spy(new D3NumOperand());
    doReturn(3).when(spyA).getTotalOperand(ast);
    doReturn(2).when(spyA).getUniqueOperand(ast);
    //real NumberOfOperandCheck
    D3NumOperand realA = new D3NumOperand();
    assertEquals(realA.getTotalOperand(ast),spyA.getTotalOperand(ast));
    assertEquals(realA.getUniqueOperand(ast),spyA.getUniqueOperand(ast));
    //verify once
    verify(spyA).getTotalOperand(ast);
    verify(spyA).getUniqueOperand(ast);
}
```

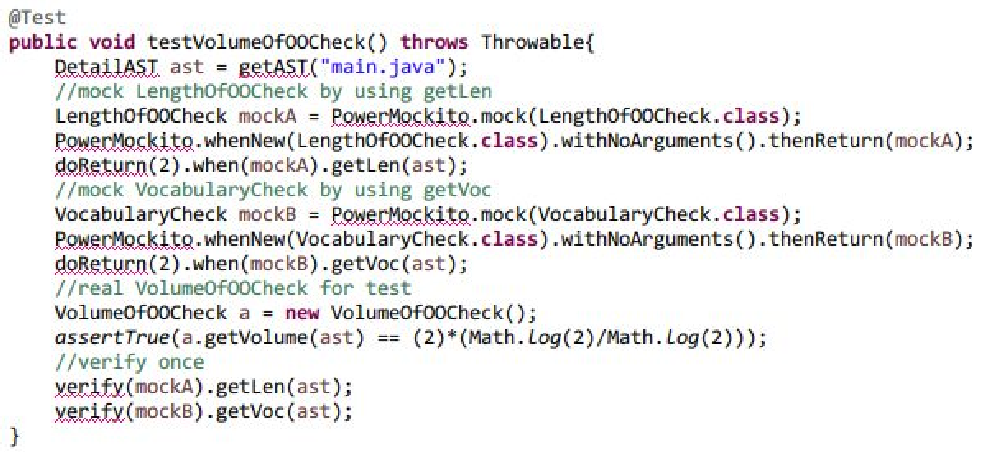
```
@Test
public void testNumberOfOperatorCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    D3NumOperator a = new D3NumOperator();
    int TotalOperator = a.getTotalOperator(ast);
    int UniqueOperator = a.getUniqueOperator(ast);

    System.out.println("Total Operator: " + TotalOperator);
    System.out.println("Unique Operator: " + UniqueOperator);
    //assert
    assertEquals(3, TotalOperator);
    assertEquals(2, UniqueOperator);
}
```

Added Test Cases:

```
@Test
   public void visitTokenTest()
   {
       DistinctOperatorsCheck myCheck = spy(new DistinctOperatorsCheck());

       DetailAST astMock = PowerMockito.mock(DetailAST.class);
       DetailAST astChildMock = PowerMockito.mock(DetailAST.class);

       // loop count should be at 0
       assertEquals(0, myCheck.getTotalDistinctOp());

       when(astMock.branchContains(TokenTypes.ASSIGN)).thenReturn(true);
       when(astMock.getFirstChild()).thenReturn(astChildMock);
       when(astMock.findFirstToken(TokenTypes.ASSIGN)).thenReturn(null);
       when(astChildMock.getType()).thenReturn(TokenTypes.ASSIGN);
       myCheck.visitToken(astMock);
       assertEquals(1, myCheck.getTotalDistinctOp());

       // should increment loop count
       when(astMock.findFirstToken(TokenTypes.INC)).thenReturn(null);
       myCheck.visitToken(astMock);
       assertEquals(2, myCheck.getTotalDistinctOp());

       DistinctOperatorsCheck myCheck2 = spy(new DistinctOperatorsCheck());

       DetailAST astMock2 = PowerMockito.mock(DetailAST.class);
       DetailAST astChildMock2 = PowerMockito.mock(DetailAST.class);

       // should not increment
       // should be 0 -- unacceptable token
       when(astMock2.branchContains(TokenTypes.BLOCK_COMMENT_BEGIN)).thenReturn(true);
       when(astMock2.getFirstChild()).thenReturn(astChildMock2);
       when(astMock2.findFirstToken(TokenTypes.BLOCK_COMMENT_BEGIN)).thenReturn(null);
       when(astChildMock2.getType()).thenReturn(TokenTypes.BLOCK_COMMENT_BEGIN);
       myCheck2.visitToken(astMock2);
       assertEquals(0, myCheck2.getTotalDistinctOp());

   }
```

Testing DiffuclutyCheck.java and VolumeOfOOCheck.java

Evaluation of the unit test on VolumeOfOOCheck.java:



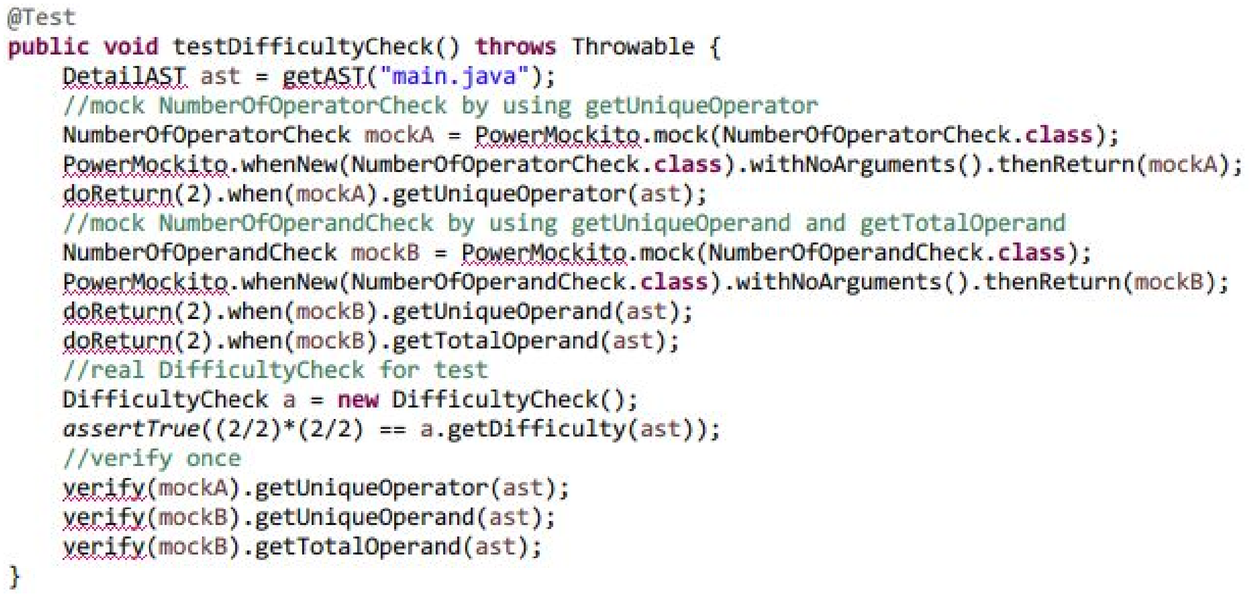
```
@Test
public void testVolumeOfOOCheck() throws Throwable{
    DetailAST ast = getAST("main.java");
    //mock LengthOfOOCheck by using getLen
    LengthOfOOCheck mockA = PowerMockito.mock(LengthOfOOCheck.class);
    PowerMockito.whenNew(LengthOfOOCheck.class).withNoArguments().thenReturn(mockA);
    doReturn(2).when(mockA).getLen(ast);
    //mock VocabularyCheck by using getVoc
    VocabularyCheck mockB = PowerMockito.mock(VocabularyCheck.class);
    PowerMockito.whenNew(VocabularyCheck.class).withNoArguments().thenReturn(mockB);
    doReturn(2).when(mockB).getVoc(ast);
    //real VolumeOfOOCheck for test
    VolumeOfOOCheck a = new VolumeOfOOCheck();
    assertTrue(a.getVolume(ast) == (2)*(Math.log(2)/Math.log(2)));
    //verify once
    verify(mockA).getLen(ast);
    verify(mockB).getVoc(ast);
}
```

The unit test for “VolumeOfOOCheck()” is missing three other cases. When either the value of genLen or getVoc is equal to zero, it must return zero. I would add on it to following piece of code:

```
//added case 1: when
doReturn(0).when(mockA).getLen(ast);
doReturn(2).when(mockB).getVoc(ast);
assertTrue(a.getVolume(ast) == 0);
//added case 2
doReturn(0).when(mockA).getLen(ast);
doReturn(0).when(mockB).getVoc(ast);
assertTrue(a.getVolume(ast) == 0);
//added case 3
doReturn(2).when(mockA).getLen(ast);
doReturn(0).when(mockB).getVoc(ast);
assertTrue(a.getVolume(ast) == 0);
```

Otherwise, the test case is correct since the function getVolume returns correct value which returns the computed equation “len * Math.log(voc) / Math.log(2);”

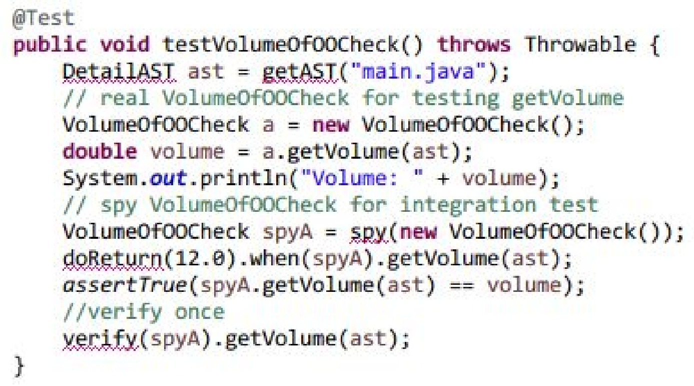
Evaluation of the unit test on “DifficultyCheck.java”:



```
@Test
public void testDifficultyCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    //mock NumberOfOperatorCheck by using getUniqueOperator
    NumberOfOperatorCheck mockA = PowerMockito.mock(NumberOfOperatorCheck.class);
    PowerMockito.whenNew(NumberOfOperatorCheck.class).withNoArguments().thenReturn(mockA);
    doReturn(2).when(mockA).getUniqueOperator(ast);
    //mock NumberOfOperandCheck by using getUniqueOperand and getTotalOperand
    NumberOfOperandCheck mockB = PowerMockito.mock(NumberOfOperandCheck.class);
    PowerMockito.whenNew(NumberOfOperandCheck.class).withNoArguments().thenReturn(mockB);
    doReturn(2).when(mockB).getUniqueOperand(ast);
    doReturn(2).when(mockB).getTotalOperand(ast);
    //real DifficultyCheck for test
    DifficultyCheck a = new DifficultyCheck();
    assertTrue((2/2)*(2/2) == a.getDifficulty(ast));
    //verify once
    verify(mockA).getUniqueOperator(ast);
    verify(mockB).getUniqueOperand(ast);
    verify(mockB).getTotalOperand(ast);
}
```

The unit test on “getDifficulty()” function seems strong enough since there is only one return value with an evaluation of “((double)n1 / 2) * ((double)N2 / (double)n2).” It contains doReturn functions to set all needed values to fixed value. Therefore, I would say this unit test is well done.

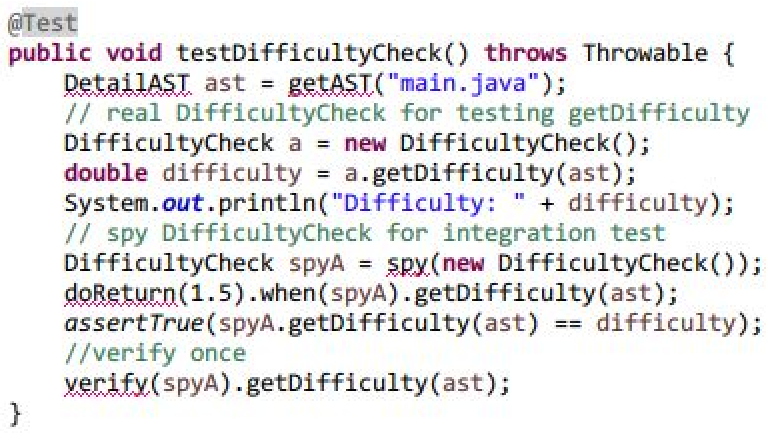
Evaluation of the integration test on “volumeOfOOCheck”:



```
@Test
public void testVolumeOfOOCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    // real VolumeOfOOCheck for testing getVolume
    VolumeOfOOCheck a = new VolumeOfOOCheck();
    double volume = a.getVolume(ast);
    System.out.println("Volume: " + volume);
    // spy VolumeOfOOCheck for integration test
    VolumeOfOOCheck spyA = spy(new VolumeOfOOCheck());
    doReturn(12.0).when(spyA).getVolume(ast);
    assertTrue(spyA.getVolume(ast) == volume);
    //verify once
    verify(spyA).getVolume(ast);
}
```

“getVolume()” returns “len * Math.log(voc) / Math.log(2).” From the test case, we could see that the value of getVoc() is 4, the value of getLen() is 6. So, 6*Log(4)/Log(2) must be 12. The integration test on “VolumeCheck” is correct.

Evaluation of the integration test on “DifficultyCheck”:



```
@Test
public void testDifficultyCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    // real DifficultyCheck for testing getDifficulty
    DifficultyCheck a = new DifficultyCheck();
    double difficulty = a.getDifficulty(ast);
    System.out.println("Difficulty: " + difficulty);
    // spy DifficultyCheck for integration test
    DifficultyCheck spyA = spy(new DifficultyCheck());
    doReturn(1.5).when(spyA).getDifficulty(ast);
    assertTrue(spyA.getDifficulty(ast) == difficulty);
    //verify once
    verify(spyA).getDifficulty(ast);
}
```

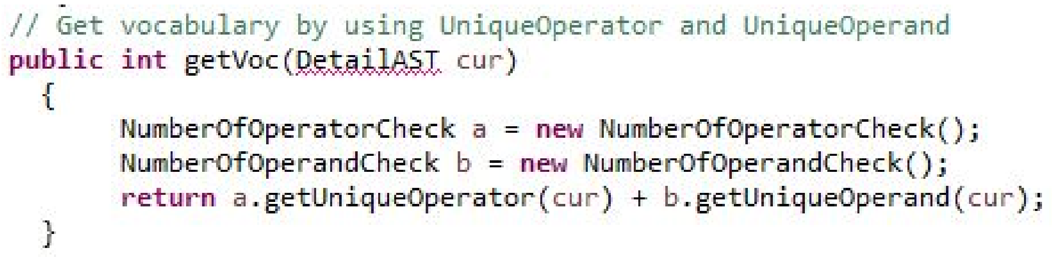
“getDifficulty()” returns the evaluation of (uniqueOperator/2) * (TotalOperand / UniqueOperand). According to the test case, we could know the number of uniqueOperand is 2, the value of totalOperand is 2, and the UniqueOperator is 3. By inserting the values into the evaluation, the output value is 1.5. The integration test on “DifficultyCheck” is correct.

<u>Testing VocabularyCheck.java and LengthOfOOCheck.java</u>

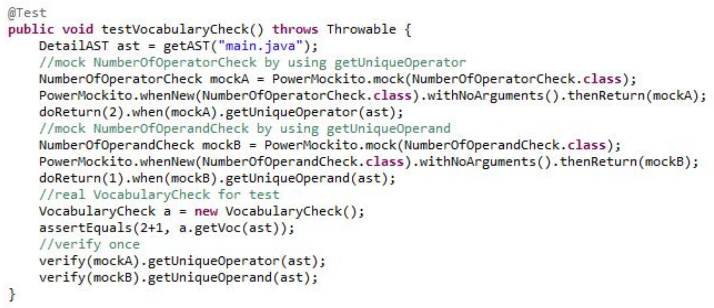
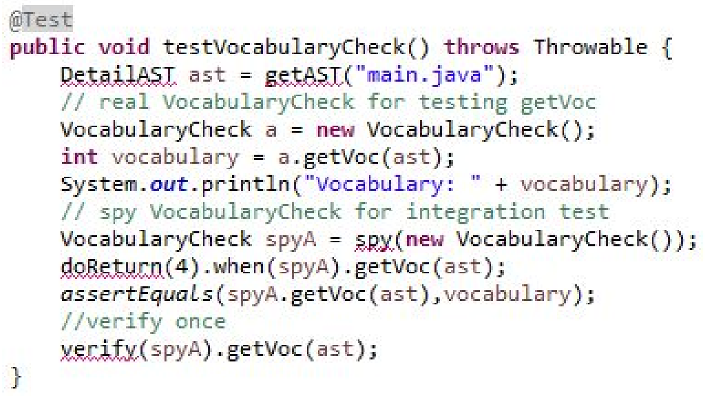
The checks I am responsible to test from the other team is the Halstead Length and Halstead Vocabulary. Halstead Length is the sum of the total number of operators and operand, and halstead Vocabulary is the sum of the number of unique operators and unique operands. The name of the checks in the program they go by is VocabularyCheck and LengthOfOOCheck.

When I get the deliverable 2 from the other team, I realize the project is not runnable. I changed all the build path jar file existed in my computer, open the project on a new workspace, use quick fix from eclipse, and none of it worked with all the errors still exist, so I decided to do the mutation test manually by adding tests on my own and judge by my experience I gain by doing white box and black box testing from my deliverable 2. It is the alternative solution I decided to go for.

VocabularyCheck's check:



```java
// Get vocabulary by using UniqueOperator and UniqueOperand
public int getVoc(DetailAST cur)
  {
      NumberOfOperatorCheck a = new NumberOfOperatorCheck();
      NumberOfOperandCheck b = new NumberOfOperandCheck();
      return a.getUniqueOperator(cur) + b.getUniqueOperand(cur);
  }
```

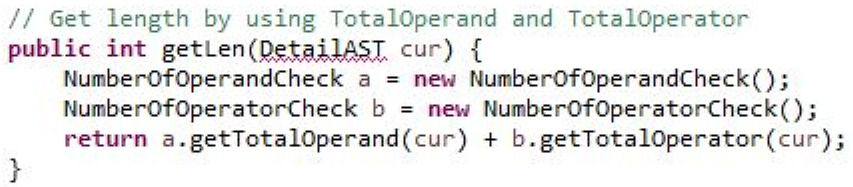
VocabularyCheck's Unit Test:	VocabularyCheck's Integration Test:
	

```java
@Test
public void testVocabularyCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    //mock NumberOfOperatorCheck by using getUniqueOperator
    NumberOfOperatorCheck mockA = PowerMockito.mock(NumberOfOperatorCheck.class);
    PowerMockito.whenNew(NumberOfOperatorCheck.class).withNoArguments().thenReturn(mockA);
    doReturn(2).when(mockA).getUniqueOperator(ast);
    //mock NumberOfOperandCheck by using getUniqueOperand
    NumberOfOperandCheck mockB = PowerMockito.mock(NumberOfOperandCheck.class);
    PowerMockito.whenNew(NumberOfOperandCheck.class).withNoArguments().thenReturn(mockB);
    doReturn(1).when(mockB).getUniqueOperand(ast);
    //real VocabularyCheck for test
    VocabularyCheck a = new VocabularyCheck();
    assertEquals(2+1, a.getVoc(ast));
    //verify once
    verify(mockA).getUniqueOperator(ast);
    verify(mockB).getUniqueOperand(ast);
}
```

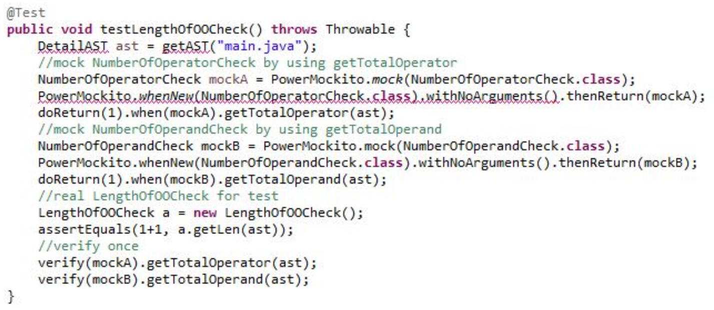
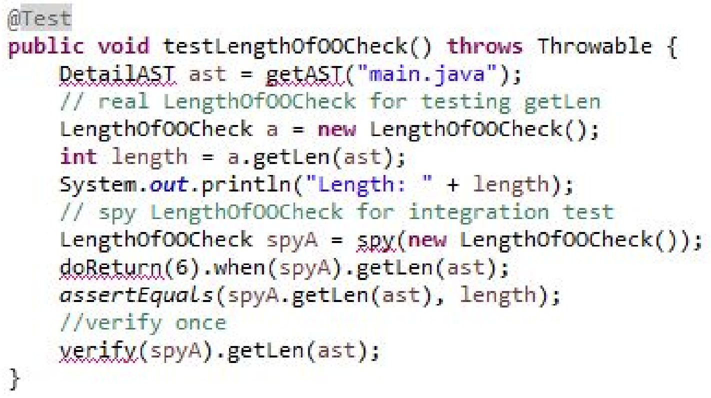
```java
@Test
public void testVocabularyCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    // real VocabularyCheck for testing getVoc
    VocabularyCheck a = new VocabularyCheck();
    int vocabulary = a.getVoc(ast);
    System.out.println("Vocabulary: " + vocabulary);
    // spy VocabularyCheck for integration test
    VocabularyCheck spyA = spy(new VocabularyCheck());
    doReturn(4).when(spyA).getVoc(ast);
    assertEquals(spyA.getVoc(ast),vocabulary);
    //verify once
    verify(spyA).getVoc(ast);
}
```

The test for VocabularyCheck from the original developers are pretty weak because the only thing they tested is the base case, where when the getter are able to get the one example. There are no test for if there are not able to create the object, or the corner cases. In result makes the unit test and the integration looks really weak.

LengthOfOOCheck's check:



```java
// Get length by using TotalOperand and TotalOperator
public int getLen(DetailAST cur) {
    NumberOfOperandCheck a = new NumberOfOperandCheck();
    NumberOfOperatorCheck b = new NumberOfOperatorCheck();
    return a.getTotalOperand(cur) + b.getTotalOperator(cur);
}
```

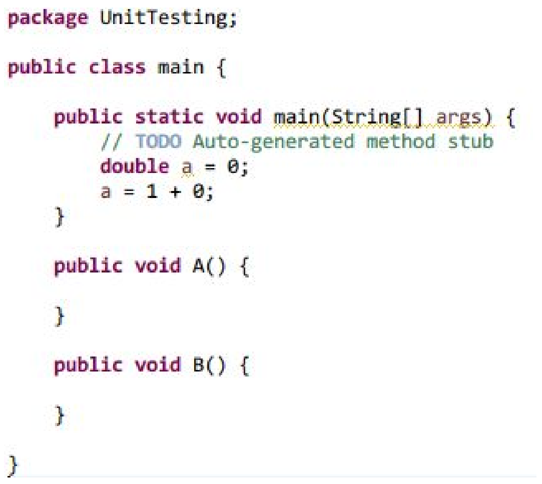
LengthOfOOCheck's Unit Test:	LengthOfOOCheck's Integration Test:
	

```java
@Test
public void testLengthOfOOCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    //mock NumberOfOperatorCheck by using getTotalOperator
    NumberOfOperatorCheck mockA = PowerMockito.mock(NumberOfOperatorCheck.class);
    PowerMockito.whenNew(NumberOfOperatorCheck.class).withNoArguments().thenReturn(mockA);
    doReturn(1).when(mockA).getTotalOperator(ast);
    //mock NumberOfOperandCheck by using getTotalOperand
    NumberOfOperandCheck mockB = PowerMockito.mock(NumberOfOperandCheck.class);
    PowerMockito.whenNew(NumberOfOperandCheck.class).withNoArguments().thenReturn(mockB);
    doReturn(1).when(mockB).getTotalOperand(ast);
    //real LengthOfOOCheck for test
    LengthOfOOCheck a = new LengthOfOOCheck();
    assertEquals(1+1, a.getLen(ast));
    //verify once
    verify(mockA).getTotalOperator(ast);
    verify(mockB).getTotalOperand(ast);
}
```

```java
@Test
public void testLengthOfOOCheck() throws Throwable {
    DetailAST ast = getAST("main.java");
    // real LengthOfOOCheck for testing getLen
    LengthOfOOCheck a = new LengthOfOOCheck();
    int length = a.getLen(ast);
    System.out.println("Length: " + length);
    // spy LengthOfOOCheck for integration test
    LengthOfOOCheck spyA = spy(new LengthOfOOCheck());
    doReturn(6).when(spyA).getLen(ast);
    assertEquals(spyA.getLen(ast), length);
    //verify once
    verify(spyA).getLen(ast);
}
```

The test for LengthOfOOCheck from the original developers are also weak just like the VocabularyCheck because the only thing they tested is the base case, where when the getter are able to get the one example. There are no test for the corner cases. In result makes the unit test and the integration looks really weak. Therefore, we should add more corner cases to make the test looks stronger.

The integration test and the test case they provided is correct, but we would not say that the integration testing is strong enough. There is a massive problem that they tested only one basic test case. The following code is the test case they have used and it seems very simple and weak.

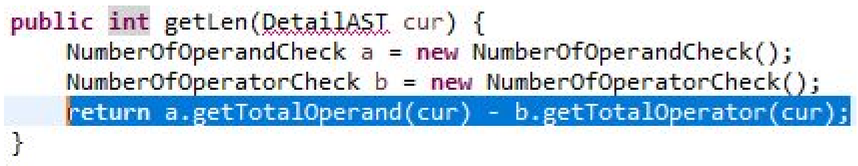


```
package UnitTesting;

public class main {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        double a = 0;
        a = 1 + 0;
    }

    public void A() {

    }

    public void B() {

    }

}
```

To complete the integration test, the developers should provide various and more complex test cases to check if the program does not make any failure of testing in all types of test cases. For example, the provided test case has only one unique operand and operator. They must check if it returns correct values when there are more than one unique operands and operators. Otherwise, the provided test case passed the integration test.

Evaluation of test cases by using Mutation Testing

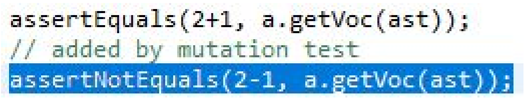
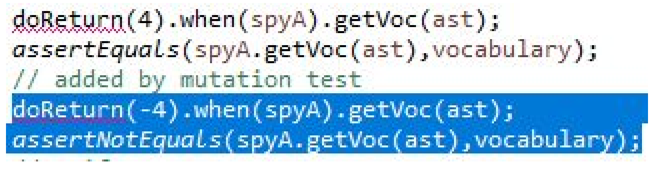
Mutation Testing on getLen() and getVoc()



```
public int getLen(DetailAST cur) {
    NumberOfOperandCheck a = new NumberOfOperandCheck();
    NumberOfOperatorCheck b = new NumberOfOperatorCheck();
    return a.getTotalOperand(cur) - b.getTotalOperator(cur);
}
```

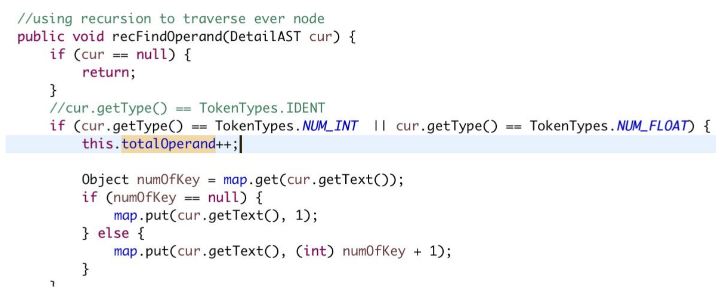
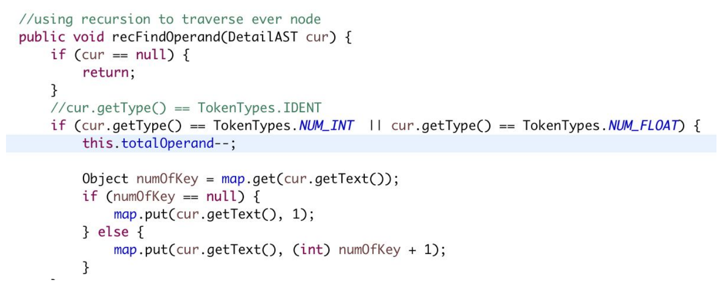
For the mutation testing, we change the plus to minus like the mutation description said, just like when we use the PIT tool and general automatically. It surely will make the unit test and integration test have a false value when b's value is greater than a's value. Therefore, we will need to add assertNotEqual to both testing to catch the error when it appear.

Unit test:	Integration test:

 `assertEquals(2+1, a.getVoc(ast));`
`// added by mutation test`
`assertNotEquals(2-1, a.getVoc(ast));`
	 `doReturn(4).when(spyA).getVoc(ast);`
	`assertEquals(spyA.getVoc(ast),vocabulary);`
	`// added by mutation test`
	`doReturn(-4).when(spyA).getVoc(ast);`
	`assertNotEquals(spyA.getVoc(ast),vocabulary);`

Now the test will have one more corner case after we did the mutation test, and it will have a higher mutation score, and have a better coverage and a higher quality.For a higher mutation score, we will have to cover more corner cases to raise the mutation score.

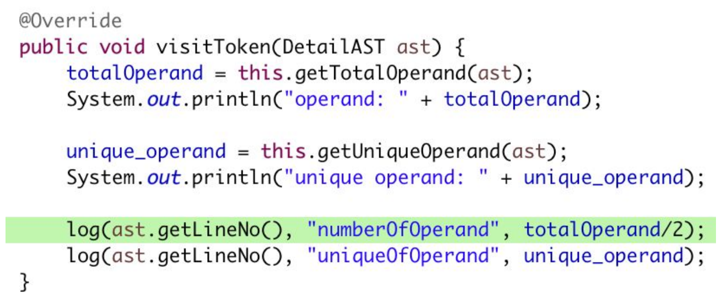
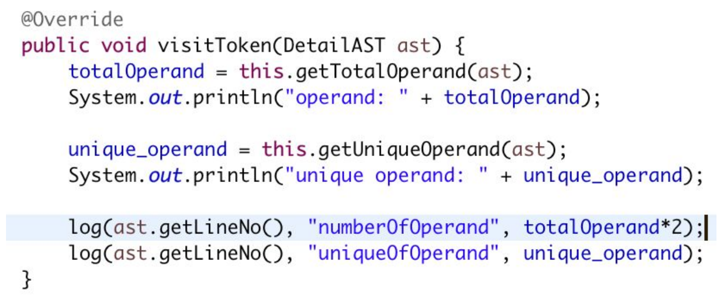
Mutation Testing on recFindOperand() and visitToken()

Original Code of recFindOperand:	Mutated Code of recFindOperand:
	

```java
//using recursion to traverse ever node
public void recFindOperand(DetailAST cur) {
    if (cur == null) {
        return;
    }
    //cur.getType() == TokenTypes.IDENT
    if (cur.getType() == TokenTypes.NUM_INT  || cur.getType() == TokenTypes.NUM_FLOAT) {
        this.totalOperand++;

        Object numOfKey = map.get(cur.getText());
        if (numOfKey == null) {
            map.put(cur.getText(), 1);
        } else {
            map.put(cur.getText(), (int) numOfKey + 1);
        }
```

```java
//using recursion to traverse ever node
public void recFindOperand(DetailAST cur) {
    if (cur == null) {
        return;
    }
    //cur.getType() == TokenTypes.IDENT
    if (cur.getType() == TokenTypes.NUM_INT  || cur.getType() == TokenTypes.NUM_FLOAT) {
        this.totalOperand--;

        Object numOfKey = map.get(cur.getText());
        if (numOfKey == null) {
            map.put(cur.getText(), 1);
        } else {
            map.put(cur.getText(), (int) numOfKey + 1);
        }
```

In their operand check, they divided a value, so the mutation resulted in no coverage if we switched that to multiplication.

Original Code of visitToken:	Mutated Code of visitToken:
	

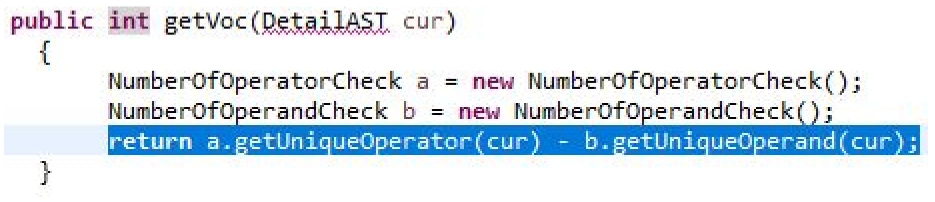
```java
@Override
public void visitToken(DetailAST ast) {
    totalOperand = this.getTotalOperand(ast);
    System.out.println("operand: " + totalOperand);

    unique_operand = this.getUniqueOperand(ast);
    System.out.println("unique operand: " + unique_operand);

    log(ast.getLineNo(), "numberOfOperand", totalOperand/2);
    log(ast.getLineNo(), "uniqueOfOperand", unique_operand);
}
```

```java
@Override
public void visitToken(DetailAST ast) {
    totalOperand = this.getTotalOperand(ast);
    System.out.println("operand: " + totalOperand);

    unique_operand = this.getUniqueOperand(ast);
    System.out.println("unique operand: " + unique_operand);

    log(ast.getLineNo(), "numberOfOperand", totalOperand*2);
    log(ast.getLineNo(), "uniqueOfOperand", unique_operand);
}
```

There was a lot of negated conditionals or printing statements that came back with no coverage, but I feel like those are minor. Before adding the test cases for these mutations, this group's mutation score was 36. After adding more test cases, I was able to get it up to 38. Even though I added more test cases, the mutation score didn't increase much, but in class we also discussed how difficult it was to increase your mutation score.

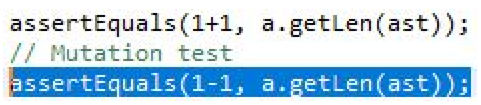
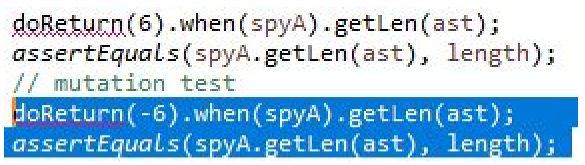
I think implementing their black-box technique would not only increase their code coverage percentage, but also gain confidence in their checks. Although they did 140 test cases, one from each class they had would have sufficed. Overall, the unit and integrations tests that were implemented were not as strong as they could have been. However, their black-box and white-box techniques made up for the lack of coverage. As for completeness, their black-box and white-box had robust coverage. For their structural coverage, this group exercised all proper elements in their software.

Mutation Testing on getVoc() and getLen():



```java
public int getVoc(DetailAST cur)
  {
      NumberOfOperatorCheck a = new NumberOfOperatorCheck();
      NumberOfOperandCheck b = new NumberOfOperandCheck();
      return a.getUniqueOperator(cur) - b.getUniqueOperand(cur);
  }
```

By using mutation test, the plus sign will become minus. Just like VocabularyCheck, errors will most likely to appear when b.getUniqueOperand is greater than a.getUniqueOperator. Therefore, we will have to create tests to cover the corner case.

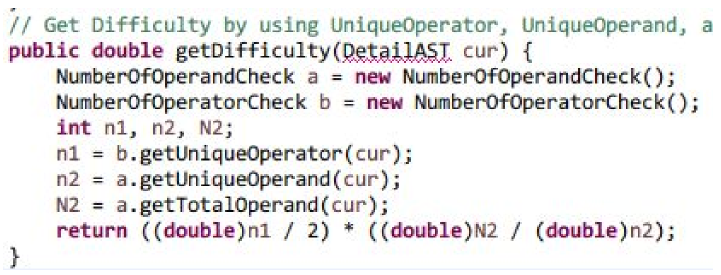
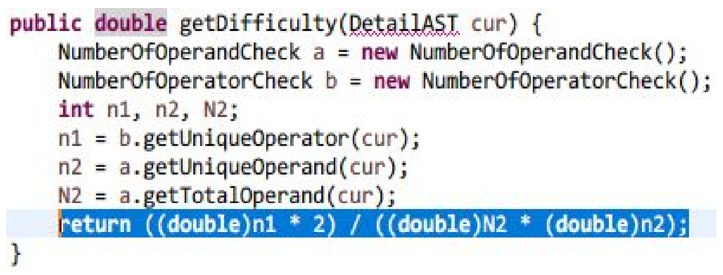
Unit test:	Integration test:
	

```
assertEquals(1+1, a.getLen(ast));
// Mutation test
assertEquals(1-1, a.getLen(ast));
```

```
doReturn(6).when(spyA).getLen(ast);
assertEquals(spyA.getLen(ast), length);
// mutation test
doReturn(-6).when(spyA).getLen(ast);
assertEquals(spyA.getLen(ast), length);
```

After the mutation testing and cover more corner cases, we are confidence to say we are now have a higher quality of the performed test. The more corner cases we can cover, the higher mutation score will get. The whole point for the mutation test is to increase mutation score close to 100%. By increasing mutation score, we can always confidence to say the quality of the testing will also be increased.

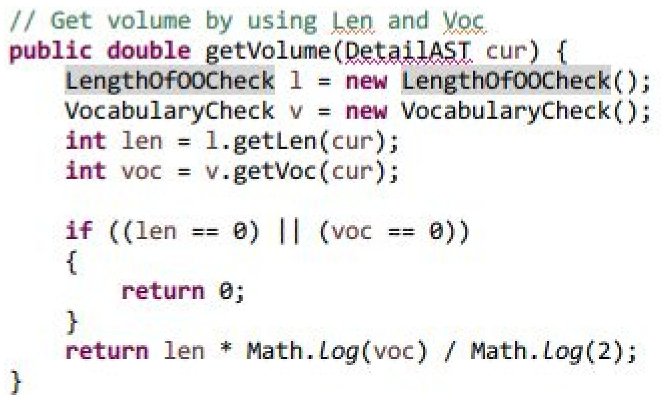
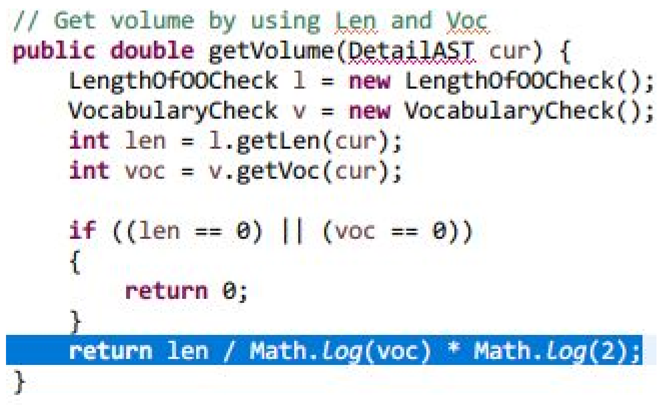
Mutation Testing on getDifficulty() and getVolume():

Original code for getDifficulty()	Mutated code for getDifficulty()
	

```
// Get Difficulty by using UniqueOperator, UniqueOperand, a
public double getDifficulty(DetailAST cur) {
    NumberOfOperandCheck a = new NumberOfOperandCheck();
    NumberOfOperatorCheck b = new NumberOfOperatorCheck();
    int n1, n2, N2;
    n1 = b.getUniqueOperator(cur);
    n2 = a.getUniqueOperand(cur);
    N2 = a.getTotalOperand(cur);
    return ((double)n1 / 2) * ((double)N2 / (double)n2);
}
```

```
public double getDifficulty(DetailAST cur) {
    NumberOfOperandCheck a = new NumberOfOperandCheck();
    NumberOfOperatorCheck b = new NumberOfOperatorCheck();
    int n1, n2, N2;
    n1 = b.getUniqueOperator(cur);
    n2 = a.getUniqueOperand(cur);
    N2 = a.getTotalOperand(cur);
    return ((double)n1 * 2) / ((double)N2 * (double)n2);
}
```

According to the result of the mutation testing, the unit test of getVolume() is not appropriate because if it is a good unit test, the coverage must be close to zero. On this case, the result didn’t change. From the unit test, the test case has input with value 2 on “len” and value 2 on “voc.” The return value of original code is 2*log(2)/log(2). The return value of mutation code is 2/log(2)*log(2). Both returns value of 2. Since the unit test cannot catch the logic changes, it is considered that the unit test is failed.

Original Code for getVolume():	Mutated code for getVolume():
	

```
// Get volume by using Len and Voc
public double getVolume(DetailAST cur) {
    LengthOfOOCheck l = new LengthOfOOCheck();
    VocabularyCheck v = new VocabularyCheck();
    int len = l.getLen(cur);
    int voc = v.getVoc(cur);

    if ((len == 0) || (voc == 0))
    {
        return 0;
    }
    return len * Math.Log(voc) / Math.Log(2);
}
```

```
// Get volume by using Len and Voc
public double getVolume(DetailAST cur) {
    LengthOfOOCheck l = new LengthOfOOCheck();
    VocabularyCheck v = new VocabularyCheck();
    int len = l.getLen(cur);
    int voc = v.getVoc(cur);

    if ((len == 0) || (voc == 0))
    {
        return 0;
    }
    return len / Math.Log(voc) * Math.Log(2);
}
```

The unit test of DifficultyCheck() also does not pass the mutation testing. The return value of original function getDifficulty() is 2/2*2/2 = 1. The return value of mutation code getDifficulty() is 2*2/2*2 = 1. Both returns value of 1. Since it cannot catch the logic changes, it is also considered that the unit test is failed.