CPTS_422 Deliverable 3
Pit Testing TeamRebecca's code

10 December 2019

Kenzo Banaag

Drew Kelly

Isaac Schulz

Sean Wallace

Repository:

https://github.com/IsaacMSchultz/CptS_422_Project

All project files can be found on the master branch.
Tests conducted for deliverable 3 are contained within \PluginProject\tests\Deliverable3Tests

Mocking Difficulties

TreeWalker

We had a lot of trouble mocking the re-implementations of the TreeWalker that were written to run TeamRebecca's Checks. Since their code contained a private function that was responsible for walking the entire tree, and counting metrics related to that check's expected output, writing a whitebox test for this function was near impossible without creating nearly an entire tree.

Since the functions are recursive, it does not lend itself to easy mocking of functions to enter specific parts of the loop. For example, examine this sections of the HalsteadMetricsCheck.java file responsible for counting the number of operands. First, we can look at a snippet from the visitToken function:

```
DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);
/*removed for simplicity*/
DetailAST child = objBlock.getFirstChild();
```

We can see that the getFirstChild() function must be overridden with mockito to return another DetailAST. And further, the same function is called in the recursive countOperands() function:

```
for (int n : operandTokens()) {
    temp = ast.getChildCount(n);
    if (temp > 0) {
  /* removed in this snippet to take up less space */
    }
}

DetailAST child = ast.getFirstChild();

while (child != null) {
    count += countOperands(child);
    child = child.getNextSibling();
}
```

Since the function getFirstChild() does not take a parameter, we essentially have to mock an entire different portion of a DetailAST tree for each of the different possible checks that were written by TeamRebecca to run whitebox tests that get good coverage.

AbstractCheck

In addition, there are functions(such as getLines() for line count) being utilized within the checks that make calls to functions within the AbstractCheck class.This complicates accurately calculating code coverage, since mocking these functions requires the use of a *spy*, which does not allow for code coverage to be calculated accurately since the code we wish to measure coverage for is not actually running in the tests.

Conclusion

The team has decided that the time cost to implement the complex mocking of TeamRebecca's Re-Implementation of the TreeWalker class, and use of functions that require the use of a *spy*, we would be modifying their source code to insert getter functions for all of this data. And instead of mocking the complex treewalker, we would simply mock the getter function associated with it for most of their checks.

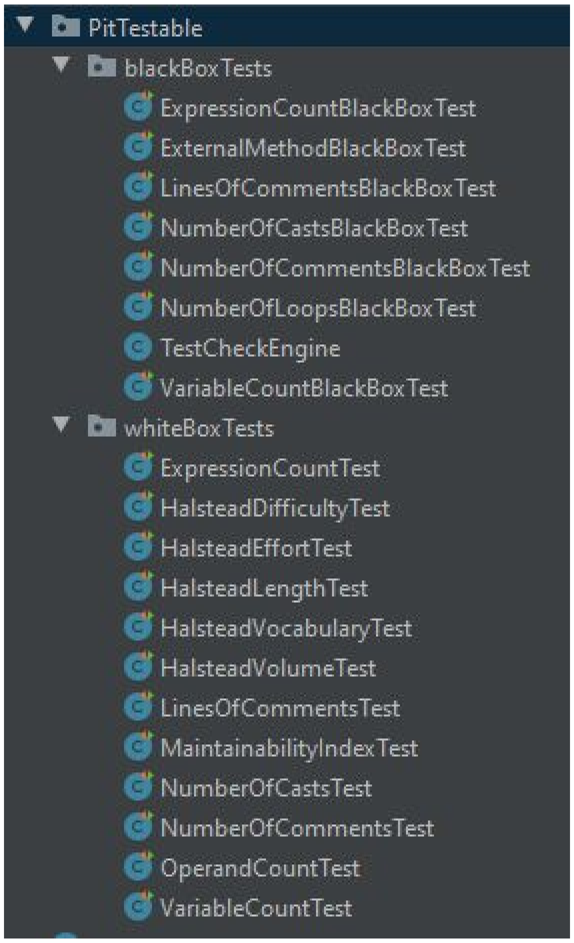
The only Check that we did partially mock to test for was the HalsteadMetricsCheck. This is what we started on, and is the reason we decided not to mock going forward. The amount of mocking and tree hierarchy needed to test properly is large. With at least 6 DetailAST's needing to be mocked. Below is an example of a function that mocked the DetailAST tree.

```
HalsteadMetricsCheck test = spy(new HalsteadMetricsCheck());
doReturn(2).when(test).getLOC(); // need to mock the lines of code cause its run each time.

DetailAST ast = PowerMockito.mock(DetailAST.class); //parent
DetailAST objBlock = PowerMockito.mock(DetailAST.class); //child
DetailAST child = PowerMockito.mock(DetailAST.class); //grandchild
DetailAST GrandChild = PowerMockito.mock(DetailAST.class); // great grandchild
DetailAST GreatGrandChild = PowerMockito.mock(DetailAST.class);

DetailAST textAST = PowerMockito.mock(DetailAST.class);

//additional mocking to deal with Dan's Treewalker
doReturn(objBlock).when(ast).findFirstToken(anyInt()); // mock ObjBlock creation
doReturn(child).when(objBlock).getFirstChild(); // Mock Child creation
doReturn(GrandChild).when(child).getFirstChild(); //mock the great great great granchild
doReturn(GreatGrandChild).when(GrandChild).getFirstChild(); //mock the great great great granchild

doReturn(textAST).when(child).findFirstToken(TokenTypes.IDENT); //mock getting text from child
doReturn(GrandChild).when(child).findFirstToken(TokenTypes.SLIST); //mock getting operators from method
definition

doReturn(1).when(child).getChildCount(); //mock 1 child
doReturn(1).when(GrandChild).getChildCount(); //stop the loop when reaching here
doReturn(0).when(GreatGrandChild).getChildCount(); //stop the loop when reaching here

test.beginTree(ast); // begin the tree

doReturn(TokenTypes.VARIABLE_DEF).when(child).getType(); // operand (with implied operator)
doReturn(TokenTypes.NUM_DOUBLE).when(GreatGrandChild).getType(); //operand
doReturn(1).when(GrandChild).getChildCount(TokenTypes.NUM_DOUBLE); //stop the loop when reaching here
doReturn(false).when(objBlock).branchContains(TokenTypes.NUM_DOUBLE); // not an operator
doReturn("double").when(textAST).getText(); //mock the name that the treewalker needs
test.visitToken(ast);
```

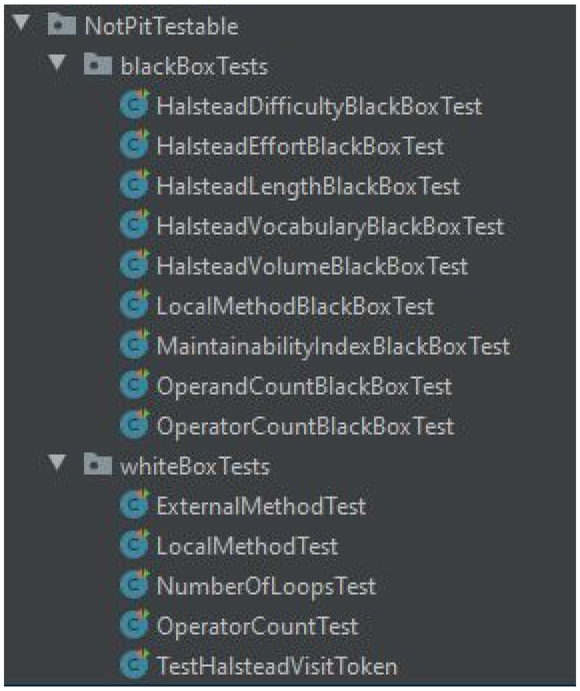
PIT Testing Difficulties

Due to significant design differences in approach White Box testing frequently needed to be radically restructured, with a heavy dependency on mocking that reduced true code coverage. Although Black Box testing should not, in theory, have been dependent on internal code structure a single significant failure ultimately cascaded throughout the testing, which rendered the PIT testing ineffective. The TeamRebecca definition of operands did not include the same set of elements, and the operand count function did not accurately count the operands in the sample code. This miscount then propagated a series of failures throughout the Halstead Checks, which were dependent on the operand count. As discussed above in the Mocking Difficulties section, although tests were written for these parts of the code, the structure of the code was such that significant portions of the program's behavior required extensive mocking, which in turn obscured the natural behavior of the code.

In the end, the team decided to break up all the tests into two groups, one of those tests that passed and were testable with PIT, and the other with failures or parts of the code that were not mocked entirely. The team will only be PIT testing the tests that ran



```
PitTestable
    blackBoxTests
        ExpressionCountBlackBoxTest
        ExternalMethodBlackBoxTest
        LinesOfCommentsBlackBoxTest
        NumberOfCastsBlackBoxTest
        NumberOfCommentsBlackBoxTest
        NumberOfLoopsBlackBoxTest
        TestCheckEngine
        VariableCountBlackBoxTest
    whiteBoxTests
        ExpressionCountTest
        HalsteadDifficultyTest
        HalsteadEffortTest
        HalsteadLengthTest
        HalsteadVocabularyTest
        HalsteadVolumeTest
        LinesOfCommentsTest
        MaintainabilityIndexTest
        NumberOfCastsTest
        NumberOfCommentsTest
        OperandCountTest
        VariableCountTest
```

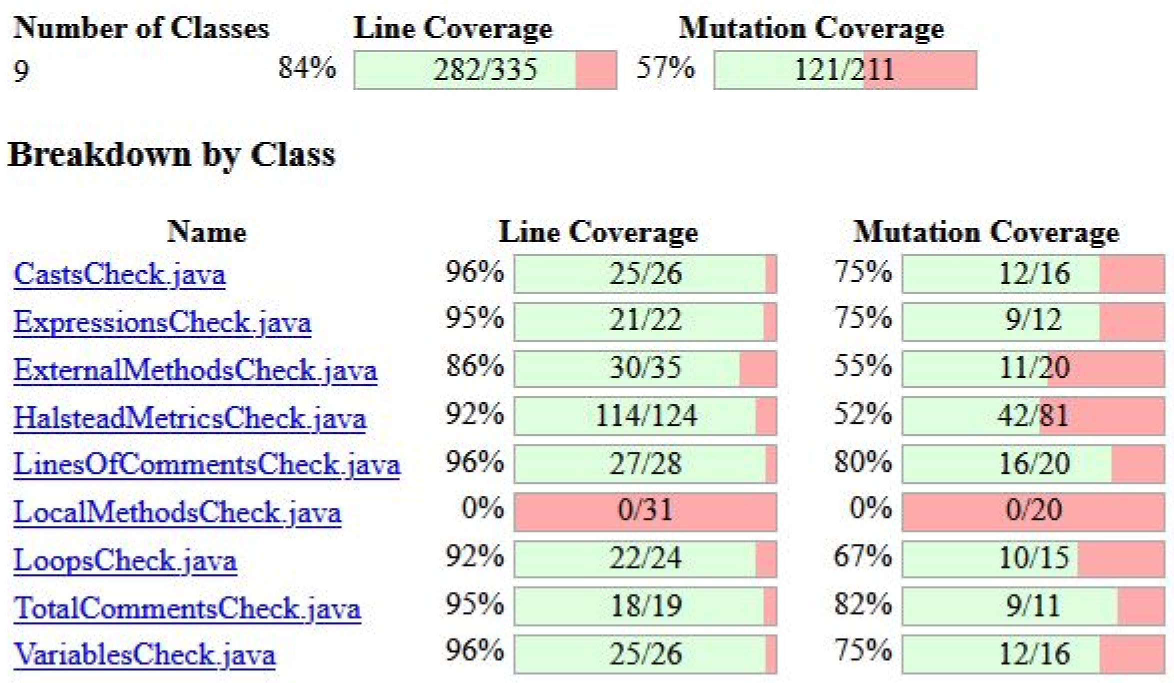


```
NotPitTestable
    blackBoxTests
        HalsteadDifficultyBlackBoxTest
        HalsteadEffortBlackBoxTest
        HalsteadLengthBlackBoxTest
        HalsteadVocabularyBlackBoxTest
        HalsteadVolumeBlackBoxTest
        LocalMethodBlackBoxTest
        MaintainabilityIndexBlackBoxTest
        OperandCountBlackBoxTest
        OperatorCountBlackBoxTest
    whiteBoxTests
        ExternalMethodTest
        LocalMethodTest
        NumberOfLoopsTest
        OperatorCountTest
        TestHalsteadVisitToken
```

Pit Testing Conclusions

After restructuring the project into a format that allows for PIT testing all the testable code, we ran the original PIT tests and got the following results:

Original Coverage

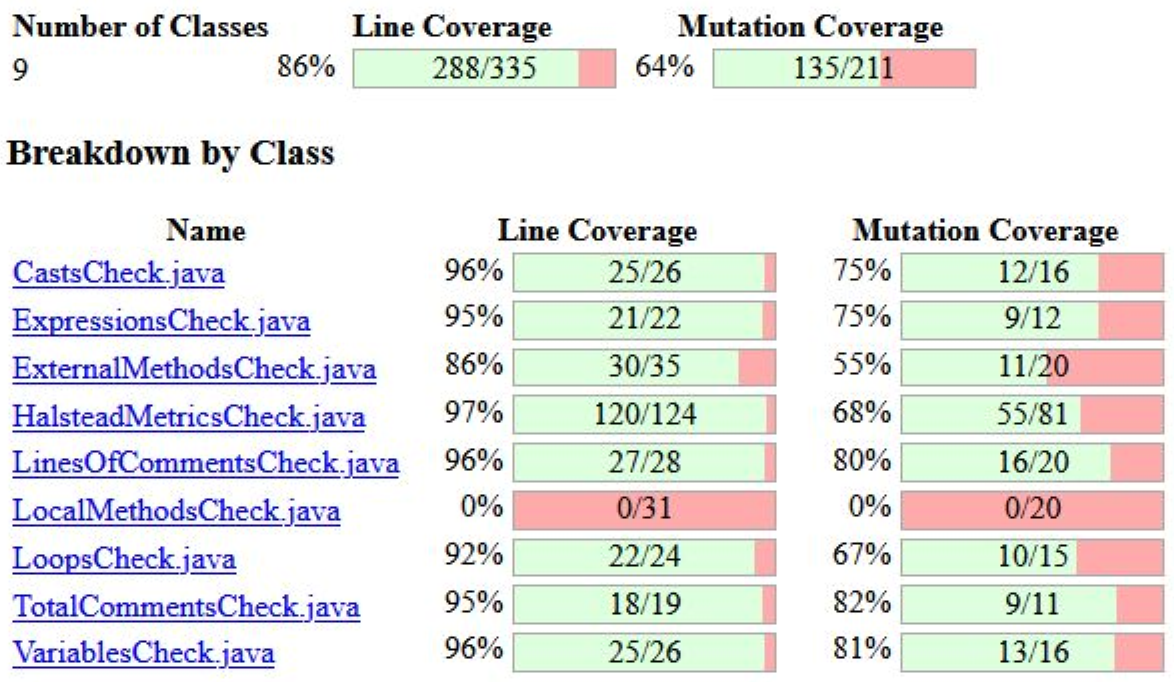


Number of Classes	Line Coverage		Mutation Coverage	
9	84%	282/335	57%	121/211

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CastsCheck.java	96%	25/26	75%	12/16
ExpressionsCheck.java	95%	21/22	75%	9/12
ExternalMethodsCheck.java	86%	30/35	55%	11/20
HalsteadMetricsCheck.java	92%	114/124	52%	42/81
LinesOfCommentsCheck.java	96%	27/28	80%	16/20
LocalMethodsCheck.java	0%	0/31	0%	0/20
LoopsCheck.java	92%	22/24	67%	10/15
TotalCommentsCheck.java	95%	18/19	82%	9/11
VariablesCheck.java	96%	25/26	75%	12/16

Final Coverage



Number of Classes	Line Coverage		Mutation Coverage	
9	86%	288/335	64%	135/211

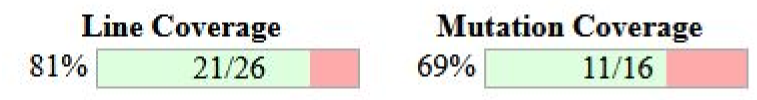
Breakdown by Class

Name	Line Coverage		Mutation Coverage	
CastsCheck.java	96%	25/26	75%	12/16
ExpressionsCheck.java	95%	21/22	75%	9/12
ExternalMethodsCheck.java	86%	30/35	55%	11/20
HalsteadMetricsCheck.java	97%	120/124	68%	55/81
LinesOfCommentsCheck.java	96%	27/28	80%	16/20
LocalMethodsCheck.java	0%	0/31	0%	0/20
LoopsCheck.java	92%	22/24	67%	10/15
TotalCommentsCheck.java	95%	18/19	82%	9/11
VariablesCheck.java	96%	25/26	81%	13/16

Improvements

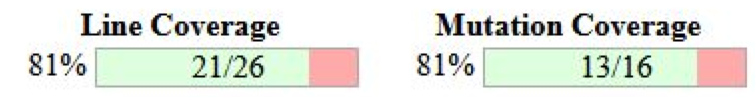
VariableCount WhiteBox tests

One area that we noticed was especially poor on mutation testing was the variable count whitebox tests. As it did not have a test for the TreeWalker function. Isaac decided to try to translate some of the mocking from the Halstead Metrics into here, and was able to increase the mutant coverage from 69%



Line Coverage		Mutation Coverage	
81%	21/26	69%	11/16

To 81% by adding 4 (absolutely massive) tests mocking the traversal of the TreeWalker.

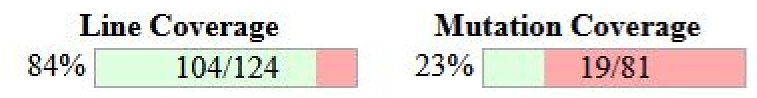


Line Coverage		Mutation Coverage	
81%	21/26	81%	13/16

We expected this to increase the coverage significantly more than it did, since there is only a few logic paths through the entire check file that the added tests cover rather thoroughly. The only conclusion we can reach is that the logic that is getting changed, is not really effecting the traversal of the tree, and thus the mutants are not getting caught.

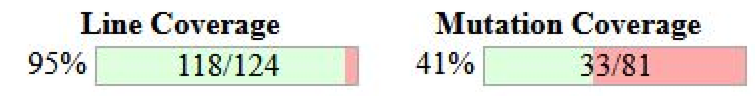
HalsteadVisitToken WhiteBox tests

Unlike with Variable count, the Treewalker was actually being mocked somewhat in the original tests. To improve mutant coverage, we expanded the test suite for this file from the original 20% coverage.



Line Coverage		Mutation Coverage	
84%	104/124	23%	19/81

To just 41% by adding tests that mock an entire other branch of the treewalker tree, as well as coverage for nodes with siblings.

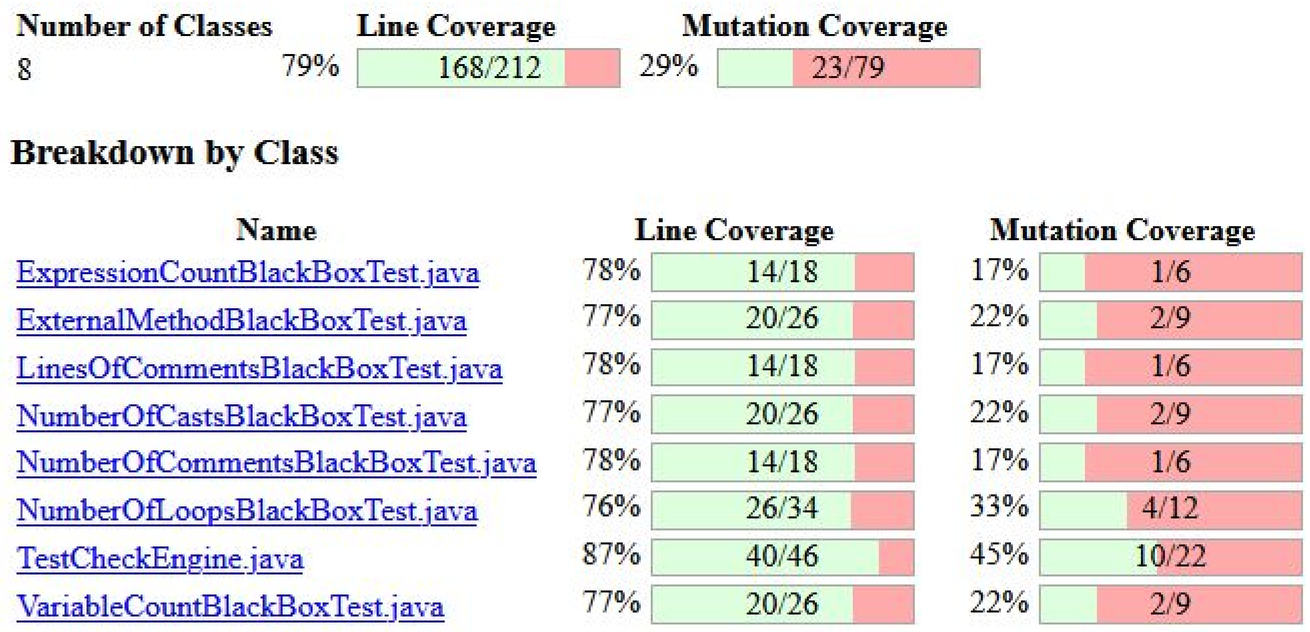


Line Coverage		Mutation Coverage	
95%	118/124	41%	33/81

This is an improvement, but still nowhere near what we were expecting for the increase in test code written.

BlackBox Tests

We also decided to add some more black box tests to increase coverage further. As a baseline we got:

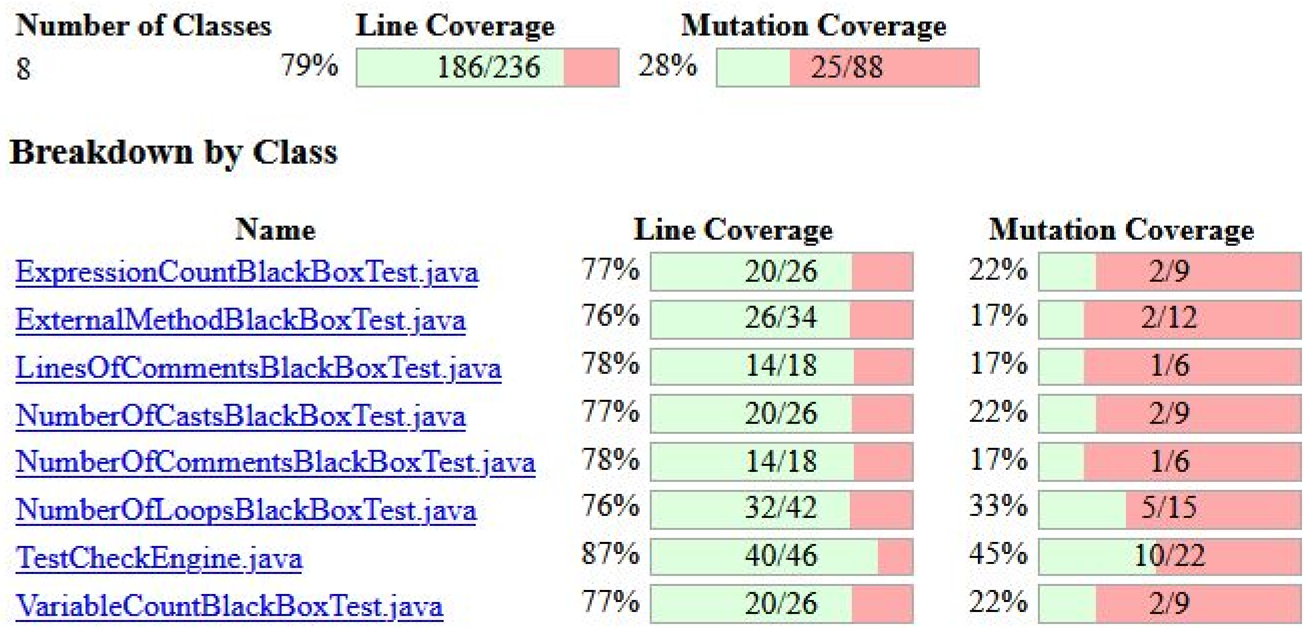


Number of Classes	Line Coverage		Mutation Coverage	
8	79%	168/212	29%	23/79

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ExpressionCountBlackBoxTest.java	78%	14/18	17%	1/6
ExternalMethodBlackBoxTest.java	77%	20/26	22%	2/9
LinesOfCommentsBlackBoxTest.java	78%	14/18	17%	1/6
NumberOfCastsBlackBoxTest.java	77%	20/26	22%	2/9
NumberOfCommentsBlackBoxTest.java	78%	14/18	17%	1/6
NumberOfLoopsBlackBoxTest.java	76%	26/34	33%	4/12
TestCheckEngine.java	87%	40/46	45%	10/22
VariableCountBlackBoxTest.java	77%	20/26	22%	2/9

With 29% coverage as our original results, so we added a few more test cases to get:



Number of Classes	Line Coverage		Mutation Coverage	
8	79%	186/236	28%	25/88

Breakdown by Class

Name	Line Coverage		Mutation Coverage	
ExpressionCountBlackBoxTest.java	77%	20/26	22%	2/9
ExternalMethodBlackBoxTest.java	76%	26/34	17%	2/12
LinesOfCommentsBlackBoxTest.java	78%	14/18	17%	1/6
NumberOfCastsBlackBoxTest.java	77%	20/26	22%	2/9
NumberOfCommentsBlackBoxTest.java	78%	14/18	17%	1/6
NumberOfLoopsBlackBoxTest.java	76%	32/42	33%	5/15
TestCheckEngine.java	87%	40/46	45%	10/22
VariableCountBlackBoxTest.java	77%	20/26	22%	2/9

A decrease to 28%. This is because despite catching more mutants, we also increased the total code covered by the tests, and thus decreased our total coverage as a result. This is the opposite to what we would have expected, as we figured that a change somewhere in the source file is much more likely to break one of the blackbox tests that runs a check on an entire code file, than just a single function in a white box test. The problem with this is it is difficult to add a new test file that you know is accurate, as the bigger it gets, the more likely we are to have messed up typing something in it. To get around this we needed to have added significantly more tests to actually increase our coverage

Final comments

In the end, most of the time we spent on this last Deliverable was on translating our tests to the format of TeamRebecca’s. Since Their code essentially re-wrote the treewalker for each check, it was incredibly difficult for us to adapt out tests to theirs, and we ended up having to scrap many of our tests because of it.

We did still increase our mutant coverage after some code modification, but we weren't able to spend as much time on it as we wanted to. We are still . We were able to achieve a 7% mutation coverage increase in white box tests solely from adding more mutator killer tests from halstead metrics and variable count checks.