Alex Puga

011425121

MS2 & MS3 Report

**MS2 Report**

Commit Hash: 8792df6b7f095c8f353f5930005c35faa21eb1bf & e6223ba72e030dcce132646a947d7681de5aa0ba

**Overall Results**

All tests pass with overall line coverage on the StructuralMetrics Project came to be 100% and branch coverage to be 97% as shown below in Figure 1. I will go over the test cases, test results, and coverage percentage.

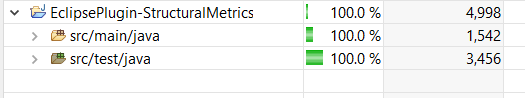
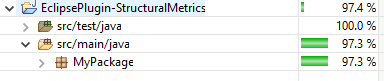
 

Figure 1: Test Coverage Results for Line and Branch

**Tests**

The test cases implemented are based on using Statement and Edge Coverage Criterion for the given classes. For the test cases below, I will be focusing on the visitToken and any other function used within visitToken as these will be the core of checking that specified metric. I will also be skipping the cases for MetricsSingleton and StructuralMetricsCheck. For more information on why I will be skipping these two classes please reference below.

**TestMetricsSingleton**

Line Coverage: 100%

Branch Coverage: 100%

Test Results: Pass

Description: The MetricsSingleton is responsible for storing, setting, resetting, getting, and utilizing operators, operands, expressions, loops, unique operators, unique operands, Halstead length, Halstead vocab, Halstead volume, Halstead difficulty, and Halstead effort. This singleton computes and retrieves all the Halstead metrics when needed. That being said the test cases performed here are very straight forward unit tests so I will be skipping these test cases.

**StructuralMetricsCheckTest**

Line Coverage: 100%

Branch Coverage: 100%

Test Results: Pass

Description: The StructuralMetricsCheck is responsible only for outputting the Halstead metrics. Unlike the other classes, this one does not visit any tokens as it utilizes the MetricsSingleton to compute and retrieve the data. I will also be skipping the test cases for this class.

**TestExprCheck**

Line Coverage: 100%

Branch Coverage: 100%

Test Results: Pass

Test Cases:

|  |  |  |
| --- | --- | --- |
| **visitToken** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| EXPR | Check expression count to be increased by 1 with valid data | Expressions count + 1 |
| COMMENT\_CONTENT | Check expression count to remain the same with invalid data | Expressions count to remain the same |

|  |  |  |
| --- | --- | --- |
| **checkExpression** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| EXPR | Check if type is an expression with valid data | True |
| BAND | Check if type is an expression with invalid data | False |

**TestStructuralMetricsCommentsCheck**

Line Coverage: 100%

Branch Coverage: 87%

Test Results: Pass

Test Cases:

|  |  |  |  |
| --- | --- | --- | --- |
| **visitToken** |  |  |  |
| **Mock ast Type** | **Mock ast parent type** | **Description** | **Expected Output** |
| SINGLE\_LINE\_COMMENT | SINGLE\_LINE\_COMMENT | Check if comment count is increased with valid type and parent type | numComments count + 1 |
| BLOCK\_COMMENT\_BEGIN | SINGLE\_LINE\_COMMENT | Check if comment count is increased with valid type and parent type. Check if bcls gets updated with valid data | numComments count + 1  bcls set to line number of mock |
| BLOCK\_COMMENT\_BEGIN | BLOCK\_COMMENT\_BEGIN | Check if comment count is remained the same with invalid parent type | numComments to remain the same |
| BLOCK\_COMMENT\_END | None | Check if the comment count and number of lines of comments are increased. Makes sure bcle and bcls is reset. | numComments count + 1  numLinesComments increased by line difference in bcls and bcle  bcls and bcle reset to -1 |

**TestStructuralMetricsLoopsCheck**

Line Coverage: 100%

Branch Coverage: 100%

Test Results: Pass

Test Cases:

|  |  |  |
| --- | --- | --- |
| **visitToken** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| LITERAL\_WHILE | Check loop count to be increased by 1 for while loop | Loop count + 1 |
| LITERAL\_FOR | Check loop count to be increased by 1 for for loop | Loop count + 1 |
| DO\_WHILE | Check loop count to be increased by 1 for do while loop | Loop count + 1 |
| NUM\_DOUBLE | Check loop count to remain the same for invalid mock type | Loop count to remain the same |

|  |  |  |
| --- | --- | --- |
| **isLoop** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| LITERAL\_WHILE | Check if isLoop returns true for while loop mock type | True |
| LITERAL\_FOR | Check if isLoop returns true for for loop | True |
| DO\_WHILE | Check if isLoop returns true for do while loop | True |
| BSR\_ASSIGN | Check if isLoop returns false for invalid mock type | False |

**TestStructuralMetricsOperandsCheck**

Line Coverage: 100%

Branch Coverage: 96%

Test Results: Pass

Test Cases:

|  |  |  |  |
| --- | --- | --- | --- |
| **visitToken** |  |  |  |
| **Returns** | **Parent Type** | **Description** | **Expected Output** |
| checkNum: True  checkOperator: False  isValidIdent: False | EXPR | Checks if operands count increases by 1 when mock is a num variable and parent’s type is EXPR. Unique operands should remain the same as isValidIdent is returned false. | Operands count + 1  Unique Operands remain the same |
| checkNum: True  checkOperator: False  isValidIdent: False | PLUS | Checks if operands count increases by 1 when mock is a num variable and parent’s type is an operator. Unique operands should remain the same as isValidIdent is returned false. | Operands count remains the same  Unique Operands remain the same |
| checkNum: False  checkOperator: False  isValidIdent: False | PLUS | Checks if operands count remains the same when mock is not a num variable and parent’s type is an operator. Unique operands should remain the same as isValidIdent is returned false. | Operands count + 1  Unique Operands remain the same |
| checkNum: False  checkOperator: False  isValidIdent: True | PLUS | Checks if a unique operator is added when isValidIdent returns true. Operand count should remain the same since checkNum and checkOperator is returning false. | Operands count remains the same  Unique Operands + 1 |
| checkNum: False  checkOperator: False  isValidIdent: True | PLUS | Checks if a unique operator remains the same when isValidIdent returns true but the operator text is the same as previous test. Operand count should remain the same since checkNum and checkOperator is returning false. | Operands count remains the same  Unique Operands remain the same |

|  |  |  |
| --- | --- | --- |
| **checkNum** |  |  |
| **Return Type** | **Description** | **Expected Output** |
| All number token types including IDENT | Check if checkNum returns true for all number token types and IDENT | True |
| BXOR | Check if checkNum returns false for a non number token type or IDENT | False |

|  |  |  |
| --- | --- | --- |
| **checkOperator** |  |  |
| **Return Type** | **Description** | **Expected Output** |
| All operator token types | Check if checkOperator returns true for operator token types | True |

|  |  |  |
| --- | --- | --- |
| **checkIdentVar** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| Mock: NUM\_DOUBLE parent: PLUS | Check if checkIdentVar returns true when parent is an operator and mock is not an operator, identity, or expression token. | True |
| Mock: NUM\_DOUBLE parent: EXPR | Check if checkIdentVar returns true when parent is an expression and mock is not an operator, identity, or expression token | True |
| Mock: IDENT parent: EXPR | Check if checkIdentVar returns false when parent is an expression and mock is an identity | False |

|  |  |  |
| --- | --- | --- |
| **isValidIdent** |  |  |
| **Return Type** | **Description** | **Expected Output** |
| checkIdent: True  checkIdentVar: False | Check if isValidIdent returns true when checkIdent is true and checkIdentVar is false | True |
| checkIdent: False  checkIdentVar: True | Check if isValidIdent returns true when checkIdent is false and checkIdentVar is true | True |
| checkIdent: False  checkIdentVar: False | Check if isValidIdent returns false when checkIdent is false and checkIdentVar is false | False |

**TestStructuralMetricsOperatorsCheck**

Line Coverage: 100%

Branch Coverage: 100%

Test Results: Pass

Test Cases:

|  |  |  |
| --- | --- | --- |
| **visitToken** |  |  |
| **checkOperator return** | **Description** | **Expected Output** |
| True | Check if operators and unique operator count is increased by 1 when checkOperator returns true | Operator count + 1  Unique Operator count + 1 |
| False | Check if operators and unique operator count is remained the same checkOperator returns false | Operators and Unique Operator count is remained the same |

|  |  |  |
| --- | --- | --- |
| **checkOperator** |  |  |
| **Mock ast Type** | **Description** | **Expected Output** |
| All operator token types | Check if all operators token types are returned true | True |
| CHAR\_LITERAL | Check if a non operator token type is returned false | False |

|  |  |  |
| --- | --- | --- |
| **addUniqueOps** |  |  |
| **convertUniqueOp return** | **Description** | **Expected Output** |
| MINUS | Check if unique operator count is increased by 1 with valid return type of convertUniqueOp | Unique Operator count + 1 |
| MINUS | Check if unique operator count is remained the same as MINUS has already been counted as a unique operator | Unique Operator count to remain the same |
| -1 | Check if unique operator count is remained the same from an invalid token type | Unique Operator count to remain the same |

|  |  |  |
| --- | --- | --- |
| **convertUniqeOp** |  |  |
| **checkOperator return** | **Description** | **Expected Output** |
| True | Check if convertUniqueOp returns token type (in this case DIV) when checkOperator returns true | DIV |
| False | Check if convertUniqueOp returns -1 when checkOperator returns false | -1 |

**MS3 Report**

This document will focus on evaluating the white box tests, results of mutation testing with Pitclipse, and making fault models for black box testing.

Commit Hash: c8f38bbb9987eb82138548f0f9151a184ec548b9

**Assumptions**

Implemented operators, operands, unique operators, unique operands, number of comments, number of lines of comments, number of looping statements, number of expressions, Halsted length, Halsted vocab, Halsted volume, Halsted difficulty, Halsted effort. Operators are assumed to be any +,/,\*,-,%,+=, -=, etc... when defining or assigning a variable. The trailing \* from imports are not considered to be an operator. Operands are any number or variable used to determine a variable. When determining unique operands, there is an assumption that class definitions and imports are not considered to be operands. Any nested function calls within an expression should be considered as one operand. i.e. int i = foo.getData().size() should be considered one operand. Any variables passed into a function call should also be counted as operands.

**Overview of Code**

This eclipse plugin is made to go over the Structural Metrics of a given file and or project. The metrics determined are:

* Number of Operators
* Number of Operands
* Number of Unique Operators
* Number of Unique Operands
* Number of Comments
* Number of Lines of Comments
* Number of Looping Statements
* Number of Expressions
* Halstead
  + Length
  + Vocabulary
  + Volume
  + Difficulty
  + Effort

**Overview of Tests**

The implementation to determine all the metrics above is already complete with test cases that have 100% line coverage and 97% branch coverage.

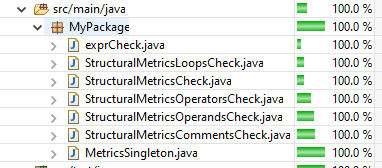
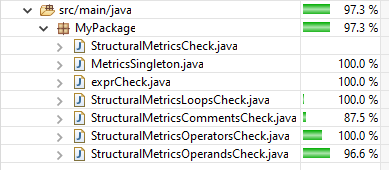
 

Figure 2: Test Coverage Results for Line and Branch

The only tests that did not complete full branch coverage are TestStrucutralMetricsComments and TestStructuralMetricsOperandsCheck. A description of the missing branch coverage is described below in the Before of Pitclipse Mutation Test Results.

**Fault Models**

**MetricsSingleton**

**Summary:**  This class is designated to help share information among all the other classes that require some metric to determine another metric. Since this class is not extended from the AbstractCheck and is solely used as a helper class, I will not be doing a fault model for this class.

**StructuralMetricsOperandsCheck**

**Summary:**  Visits a group of tokens to determine the total number of operands and unique operands. Uses MetricsSingelton to store the operands and unique operands.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **Operand** | * Nested function calls in expression should be one operand * Parameters in function calls not being counted as operands * Different types of variables in expressions |
| **Unique Operands** | * Nested function calls in expression not being counted as operand * Creation of variables counting as operands * Function calls with parameters not being considered as operands |

**StructuralMetricsOperatorsCheck**

**Summary:**  Visits all operator tokens to determine the total number of operators and unique operators. Uses MetricsSingelton to store the operators and unique operators.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **Operator** | * A mistake in counting the trailing star in an import as an operator |
| **Unique Operator** | * A function call being considered as operator |

**StructuralMetricsCommentsCheck**

**Summary:**  Visits all comment type tokens to determine the total number of comments and lines of comments.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **Comments** | * If a line of comment contains additional “//” within the comment itself * A long one line comment |
| **Lines of Comments** | * A block comment containing “//” inside * A block comment with a long one line comment |

**StructuralMetricsLoopsCheck**

**Summary:**  Visits all loop type tokens to determine the total number of loops.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **Loops** | * Missing loop count for a foreach loop * Multiple counts of loops if loop within a loop |

**exprCheck**

**Summary:**  Visits all expression tokens to determine total number of expressions. Uses MetricsSingleton to store the operators and unique operators.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **Expressions** | * When performing an expression within a function call * When performing an expression within a if statement * Not counting the if statement or else statement as expressions * A return statement from a function * Parameters not being counted as expressions |

**StructuralMetricsCheck**

**Summary:**  Utilizes the MetricsSingleton to compute and display all the Halstead Metrics results mentioned above. No tokens are visited.

|  |  |
| --- | --- |
| **Metric** | **Possible Faults** |
| **All Halstead Metrics** | * For all the Halstead Metrics the only way for them to be incorrect is when Operands, Unique Operands, Operators, or Unique Operators do not accurately compute the correct result. This relies on the metrics mentioned to be accurate since StructuralMetricsCheck itself does not visit any tokens but computes and reports the Halstead Metrics. |

**Black Box Testing Results**

After executing the black box tests, there were 3 detected failures. This had shown me a great importance in why we should not only choose one or the other when testing but both. A description on these failures is listed below:

* BlackBoxOperandsTest
  + When creating variables, counted the variable being defined as a unique operand
  + Failed to count characters and strings as operands
  + Failed to count a variable’s function call inside an expression as unique & operand
  + All function calls where considered to be one unique operand
* BlackBoxOperatorsTest
  + Failed to exclude trailing \* from imports as operand
  + Failed to exclude trailing \* from imports as unique operand
* BlackBoxMetricsTest
  + Since Operands, Operators, Unique Operands, and Unique Operators were miscalculated, this led to the Halstead Metrics to be miscalculated

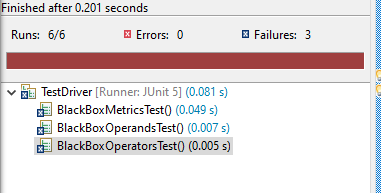


Figure 3: Test Results after Running Black Box Test cases

Because of the failing results, I decided it was best to go back and fix the checks and write new white box test cases in order to successfully pass the black box test cases.

**Pitclipse Mutation Test Results**

**Before:**

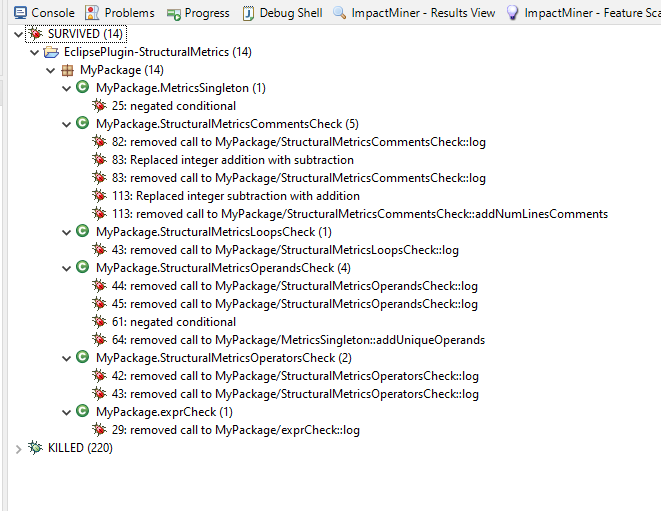


Figure 4: Mutation Testing Before Results

When executing the mutation tests after milestone 2, there were 14 bugs that had survived. Majority of these bugs are from the finishTree() function in all of my checks, specifically the log function. Besides the logs, there are a few of mutation bugs that can potentially be fixed. The remaining mututation bugs, 25, 113, 61, and 64 are due to poor quality testing that can potentially be fixed by writing better test cases to eliminate them. A test plan is listed below:

|  |  |
| --- | --- |
| **Bug** | **Potential Fix** |
| 25 | Since the singleton is determined to check whether an instance is already created, the mutation bug that survived is within the getInstance() function in which it checks if the singleton is null. If it is it returns a new singleton otherwise it returns the current on. The check if(singleton == null) is negated to if(singleton != null). To combat this I will create a singleton, add a random value, attempt to create a new singleton, assert the random values are equal. |
| 61 | For this one, I must write test cases for all branches within visitToken() |
| 64 | This is a similar situation for bug 61, in which I must write test cases for all branch coverage within visitToken() |
| 113 | For this bug, I had originally missed a small function that is used to compute the number of line counts. This function was partially covered from other test cases making it look it was tested but also why my branch coverage wasn’t 100%. |
| logs | This will be one of the most challenging mutations to kill. Since my log outputs are not returning any sort of data, it will be difficult finding a way to check the output of the logs. |

**After:**

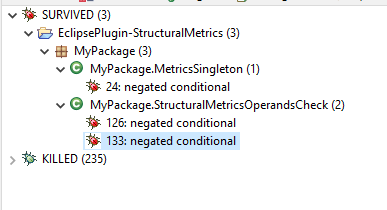


Figure 5: Mutation Testing After Results

My first attempt to kill all mutation bugs within my log function calls within the finishTree() was to store all expected messages output in an array. Using this array I can then test each individual message with the expected output. A sample snippet is show bellow.

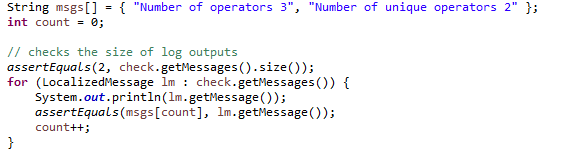


Figure 6: Code sample for Removing Remaining Mutations

I was soon to realize that I had already mocked the spy to do nothing when the log was called, it would fail. Since I had generated the black box tests I then moved these checks into their respective black box tests and was able to bring down all living mutation bugs to 3. The remaining 2 mutation bugs are because I wasn’t able to fully cover all branches within my StructuralMetricsOperandsCheck.

**Results and Quality**

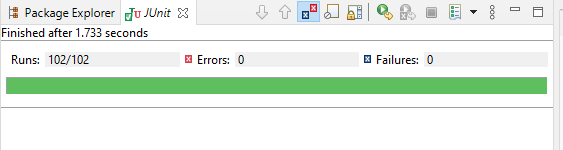


Figure 7: Passing Test Cases after Completion

When executing the all white box and black box test cases, I was able to successfully pass all tests. The only mutation bugs remaining was the negated conditional within my getInstance() in my MetricsSingleton and my getName() in StructuralMetricsOperandsCheck. I had attempted to add some default data within the singleton, calling the getInstance(), and then asserting that default data was not reset. Unfortunately, this was not able to able to catch the mutation. For fixing the mutation in my getName() function I had originally tried to mock all data being passed but because of the complexity of traversing the data I was unable to achieve this portion.

Before the additional white box testing, my overall testing quality was lacking in several different aspects such as missing a test on a function, not fully covering all branches, and not checking the log messages of all my finishTree calls. In the end, I was able to significantly improve these results by removing all mutation bugs except 3. In the end I was able to achieve a 99.6% branch coverage with 100% statement coverage.

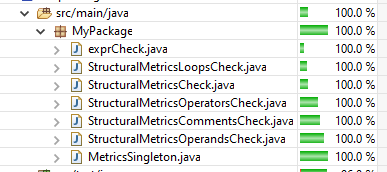
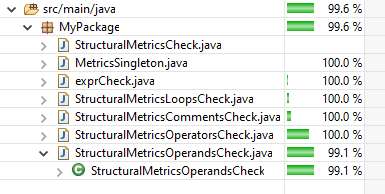


Figure 8: Branch and Statement Coverage after Completion

**Class Testing**

When considering Class testing for this project there are several differences that can be overviewed. A list of these differences are shown below:

* All attributes in a certain state are tested in correlation of input and output
* The class objects themselves are tested
* Testing interactions among classes
* All inherited classes must have changed methods tested
* All classes and subclasses must be retested if superclass is modified

While there are differences among class testing, there are still some similarities such as unit testing within the Class testing. Overall, for Class testing much more effort and test cases would need to take place but in return you get assure or uncover faults within your system that would have not been covered in regular testing.