CptS 422 Project Deliverable 2

Dan, Juan, Brandon, Rebecca

Team Rebecca Project Report

Abstract

Our deliverable 2 consists of structural metric information through the use of checkstyle, a plug-in for Eclipse. Team Rebecca has developed these metric checks to generate numeric representations of information with regards to code, such as a class file. The checks utilize a recursion structure to traverse the tree of tokens parsed by checkstyle's built in treewalker module in the form of a DetailAST. The main purpose is to develop these checks accurately, and to accomplish that we have written a myriad of tests that will objectively conclude that our checks are made with speed, accuracy, and reliability. We are 95% confident that our program runs correctly and error-free. We ran both white-box and black-box analysis, as shown in the remainder of the document, paired with test suites. For white-box testing, we wanted to achieve full branch coverage, meaning we also achieve full statement coverage. We used CFGs to trace all DU-paths to accomplish this.

Checks

Our original checks were separated out, one check per file, which was a suboptimal approach once we started developing the more complex checks. Some checks remained disjoint as they had no coupling involved with any other tests. The checks we have maintained as disjoint are:

- Number of Casts
- External Method References
- Local Method References
- Number of Lines of Comments
- Number of Expressions
- Number of Comments
- Number of Looping Statement
- Number of Variable Declarations

Deliverable 1 involved having only one check per member made, so we were not completely aware of the level of involvement some tests on one another. Originally the operators and operands were separate checks individually, but after development of the Halstead metrics, our group quickly realized that those metrics required the use of counting operators and operands as the root of Halstead metrics. Once that fact came to light, we relegated to combining the operators and operands into a clustered Halstead metrics check, which checks all of the Halstead metrics at once, as many of them build upon each other. These checks include:

- Halstead Length
- Halstead Vocabulary

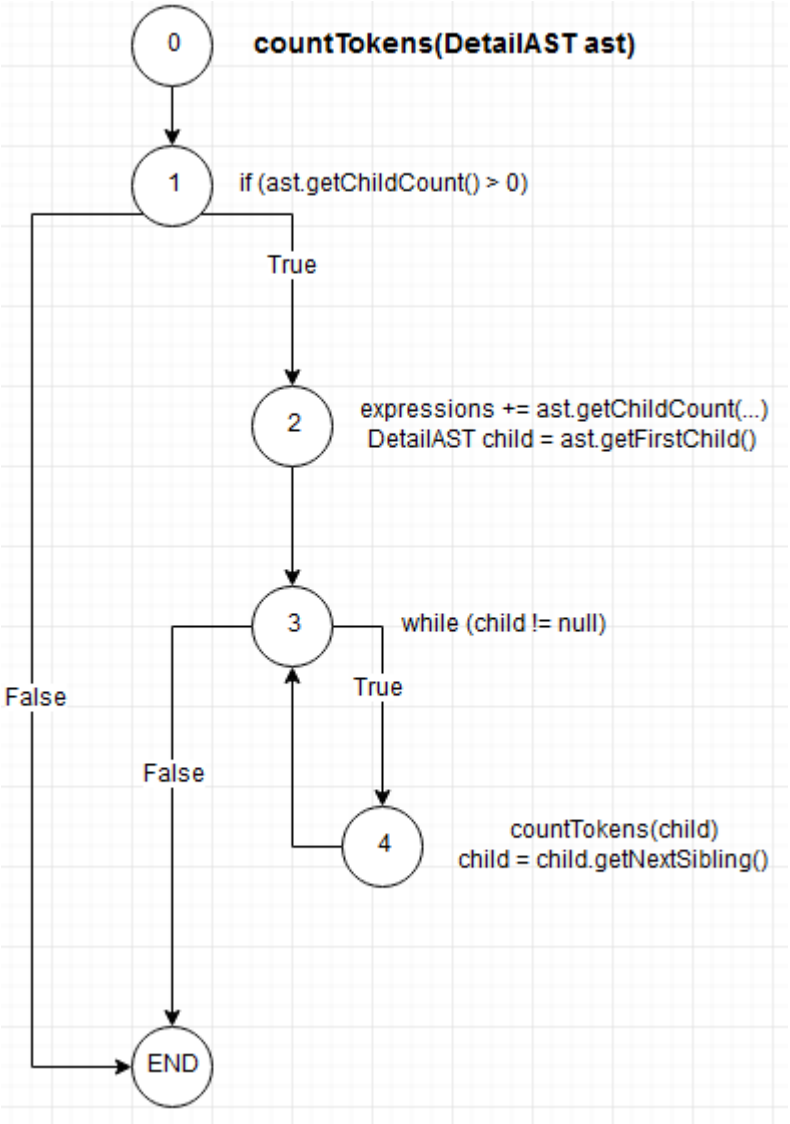
- Halstead Volume
- Halstead Effort
- Number of Operands
- Halstead Difficulty
- Number of Operators
- Maintainability Index

Number of operators and operands are necessary to calculate Halstead Length, Halstead Vocabulary, and Halstead Difficulty. The three resulting aforementioned checks are required in order to calculate Halstead Volume, Halstead Effort, and Maintainability Index. Due to this cascading reliance, the merging of checks seemed to be the most efficient course of action.

White Box Testing

ExpressionCheck:

For this check the only method that has actual logic is the countTokens() function which is called as the only statement in the visitToken() function. Here is the CFG for the countTokens() function:



Node(i)	Def(i)	C-Use(i)	Edge(I,j)	P-Use(I,j)
0	ast, expressions (class variable)		(0,1)	
1			(1,END) (1,2)	ast ast
2	expressions, child	ast	(2,3)	
3			(3,END) (3,4)	child child
4	child	child	(4,3)	

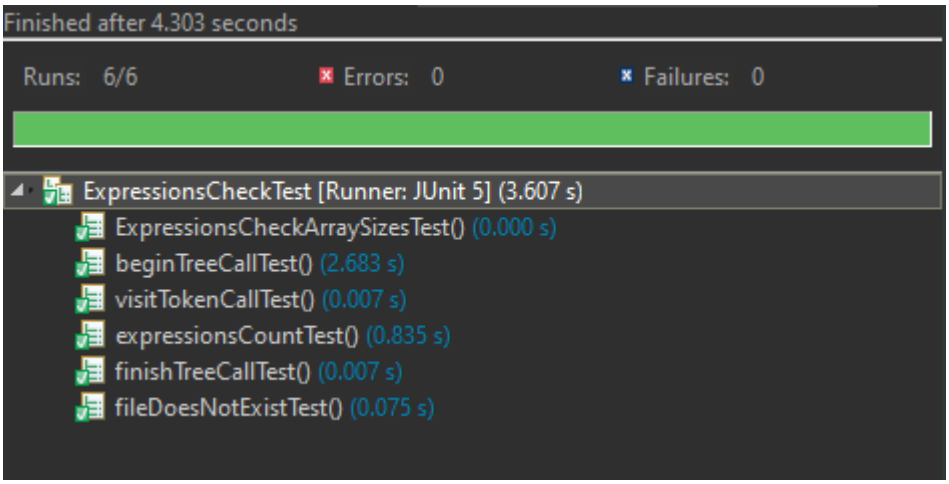
Node(i)	dcu(v,i)	dpu(v,i)
0	dcu(ast,0) = {2}	dpu(ast,0) = {(1,END), (1,2)}
2	dcu(child,2) = {4}	dpu(child,2) = {(3,END), (3,4)}
4	dcu(child,4) = {4}	dpu(child,4) = {(3,END), (3,4)}

From these tables above we can come up with a test suite that will satisfy coverage of all the dcu and dpu paths above. Note that because the only input is a DetailAST tree it's hard to provide a concrete test case for the tree, therefore I will give a more general testcase for the DetailAST.

Testcase #	DetailAST ast	Path(s) Covered
T1	A tree with no nodes (empty file)	dpu(ast,0) = (1,END)

T2	A tree with no expression nodes	dcu(ast,0) = {2} dpu(ast,0) = (1,2) dpu(child,2) = (3,END)
T3	A tree with a single expression node	dcu(child,2) = {4} dpu(child,2) = (3,4) dpu(child,4) = (3,END)
T4	A tree with multiple expression nodes	dcu(child,4) = {4} dpu(child,4) = (3,4)

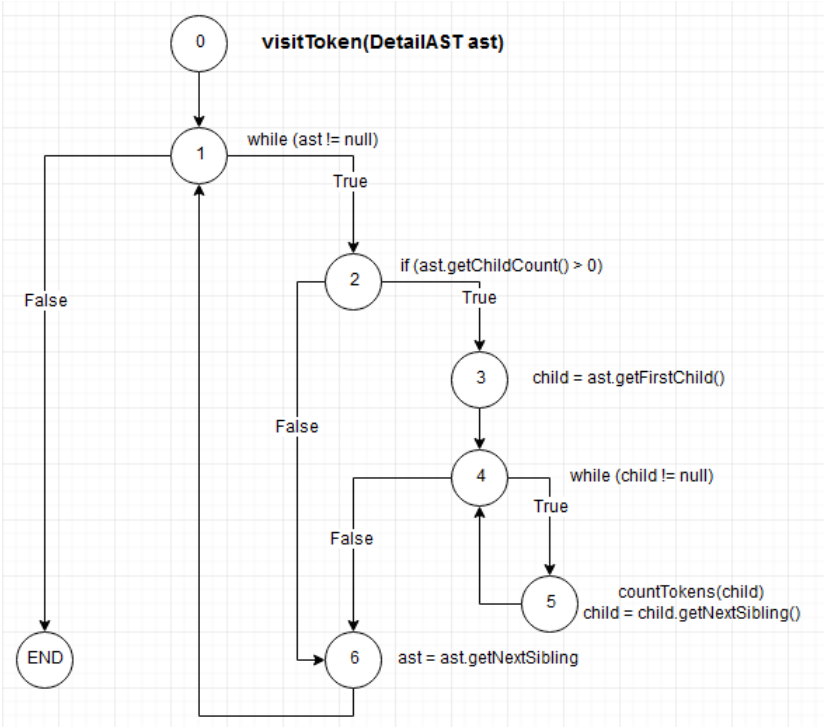
Again, because it's very difficult to come up with code that will create the specific DetailAST tree that we are looking for, we came up with tests that will "simulate" the test suite above. Here are the results of those tests and their subsequent coverage:



ExternalMethodsCheck:

This check is a little different than the Expressions check because we have two methods that contain actual logic. First we have visitToken() which will loop through the outer nodes of the DetailAST tree and call call our second method countTokens() on each of those outer nodes. countTokens() will recursively

go through all of the tree to count the number of external method calls. Here is the CFG for the visitToken() function:



Node(i)	Def(i)	C-Use(i)	Edge(i,j)	P-Use(i,j)
0	ast		(0,1)	
1			(1,END) (1,2)	ast ast
2			(2,6) (2,3)	ast ast
3	child	ast	(3,4)	
4			(4,6) (4,5)	child child

5	child	child	(5,4)	
6	ast	ast	(6,1)	

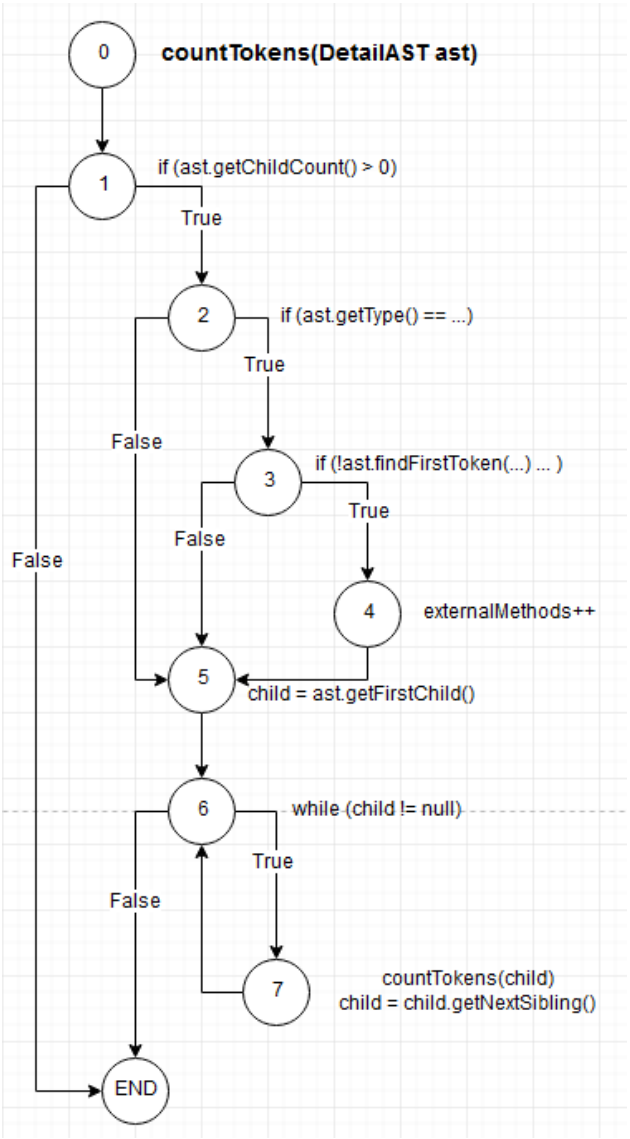
Node(i)	**dcu(v,i)**	**dpu(v,i)**
0	dcu(ast,0) = {3,6}	dpu(ast,0) = {(1,END), (1,2), (2,6), (2,3)}
3	dcu(child,3) = {5}	dpu(child,3) = {(4,6), (4,5)}
5	dcu(child,5) = {5}	dpu(child,5) = {(4,6), (4,5)}
6	dcu(ast,6) = {3,6}	dpu(ast,6) = {(1,END), (1,2), (2,6), (2,3)}

We can use the table above to find a partial Test Suite. I say partial because we still need to consider the countTokens() function which I will go over below.

TestCase #	**DetailAST ast**	**visitToken() Path(s) Covered**
T1	ast = null	dpu(ast,0) = (1,END)
T2	A tree with only a root node	dpu(ast,0) = (1,2) dpu(ast,0) = (2,6) dpu(ast,6) = (1,END)

T3	A tree with multiple children	dcu(ast,0) = {3} dpu(ast,0) = (2,3) dpu(child,3) = {(4,5), (4,6)} dcu(child,3) = {5} etc…

Here is the CFG for the countTokens() function:



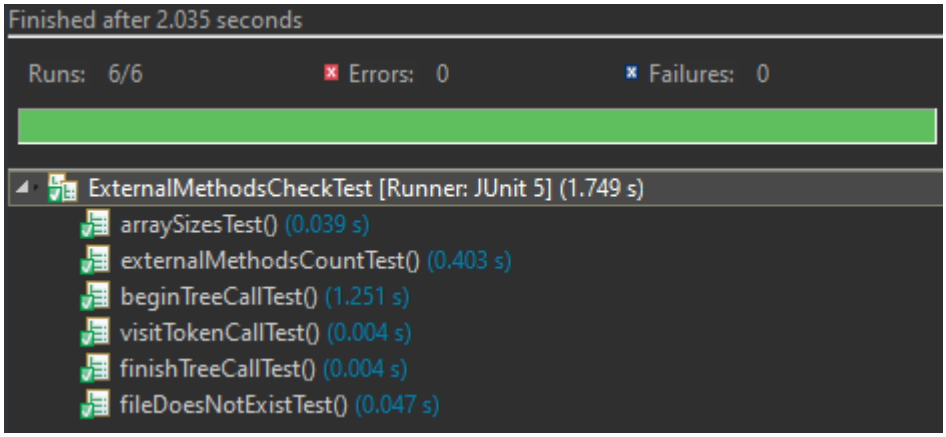
Node(i)	Def(i)	C-Use(i)	Edge(i,j)	P-Use(i,j)
0	ast, externalMethods (class variable)		(0,1)	
1			(1,END) (1,2)	ast ast
2			(2,5) (2,3)	ast ast
3			(3,5) (3,4)	ast ast
4		externalMethods	(4,5)	
5	child	ast		
6			(6,END) (6,7)	child child
7	child	child	(7,6)	

Node(i)	dcu(v,i)	dpu(v,i)
0	dcu(ast,0) = {5} dcu(externalMethods,0) = {4}	dpu(ast,0) = {(1,END), (1,2), (2,5), (2,3), (3,5), (3,4)}
5	dcu(child,5) = {7}	Dpu(child,5) = {(6,END), (6,7)}
7	dcu(child,7) = {7}	Dpu(child,7) = {6,end), (6,7)}

Using the path table above, we can create a Test Suite that will have coverage of the paths included in the table:

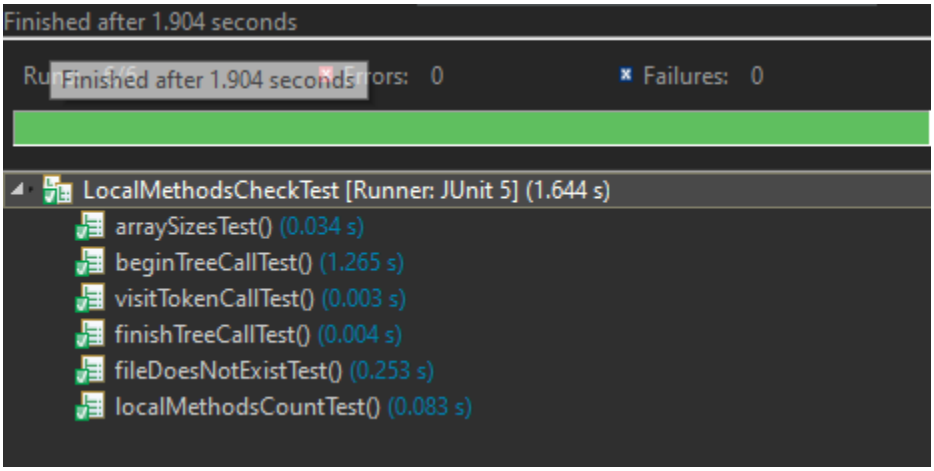
TestCase #	DetailAST ast	countTokens() Path(s) Covered
T1	A tree with no nodes	dpu(ast,0) = (1,END)
T2	A tree with no method calls	dpu(ast,0) = (2,5) dcu(child,5) = {7} Dpu(child,5) = {(6,END), (6,7)} dcu(child,7) = {7} Dpu(child,7) = {6,end), (6,7)}
T3	A tree with only internal method calls	dpu(ast,0) = (3,5)
T4	A tree with external method calls	dcu(externalMethods,0) = {4} dpu(ast,0) = (3,4)

If we combine the two test suites above we can come up with a “total test suite” used for our check. We came up with a code fragment that simulates this “total test suite” and the results are below:



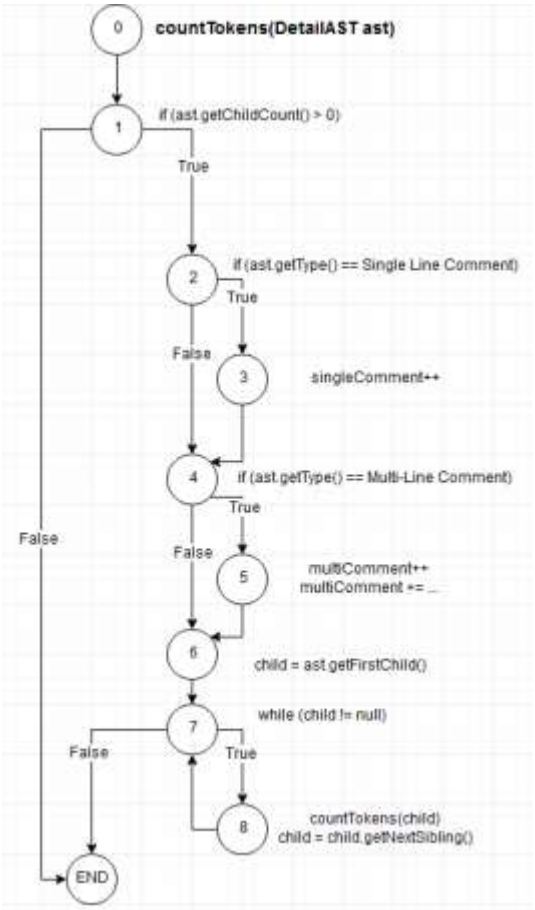
LocalMethodsCheck:

This check is exactly the same as the ExternalMethodsCheck that we have above. The only difference is the omission of a single ! (not) operator on one of the lines. As a result of this, we have an identical CFG and tables which means that we can use the same test suite except that we swap all references of "external methods" to be "local methods" and vice versa. Please see the tables and diagram directly above to avoid copying and pasting identical content. The results and coverage of the test suite is below:



LinesOfCommentsCheck.java	100.0 %	113	0	113
LinesOfCommentsCheck	100.0 %	113	0	113

LinesOfCommentsCheck:



This check is another that only takes advantage of the countTokens() function. Here is the CFG:

Node(i)	Def(i)	C-Use(i)	Edge(i,j)	P-Use(i,j)
0	ast singleComment (class variable) multiComment (class variable)		(0,1)	
1			(1,END) (1,2)	ast ast

2			(2,4) (2,3)	ast ast
3		singleComment	(3,4)	
4			(4,6) (4,5)	ast ast
5		multiComment	(5,6)	
6	child	ast	(6,7)	
7			(7,END) (7,8)	child child
8	child	child	(8,7)	

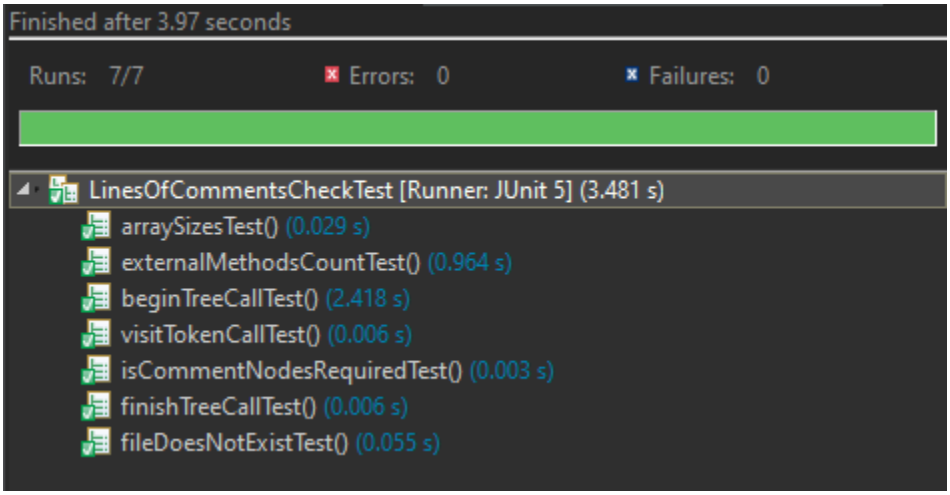
Node(i)	**dcu(v,i)**	**dpu(v,i)**
0	dcu(ast,0) = {6} dcu(singleComment,0) = {3} dcu(multiComment,0) = {5}	dpu(ast,0) = {(1,END), (1,2), (2,4), (2,3), (4,6), (4,5)}
6	dcu(child,6) = {8}	dpu(child,6) = {(7,END), (7,8)}
8	dcu(child, 8) = {8}	dpu(child,8) = {(7,END), (7,8)}

We now created a test suite to cover the paths above:

TestCase #	**DetailAST ast**	**Path(s) Covered**

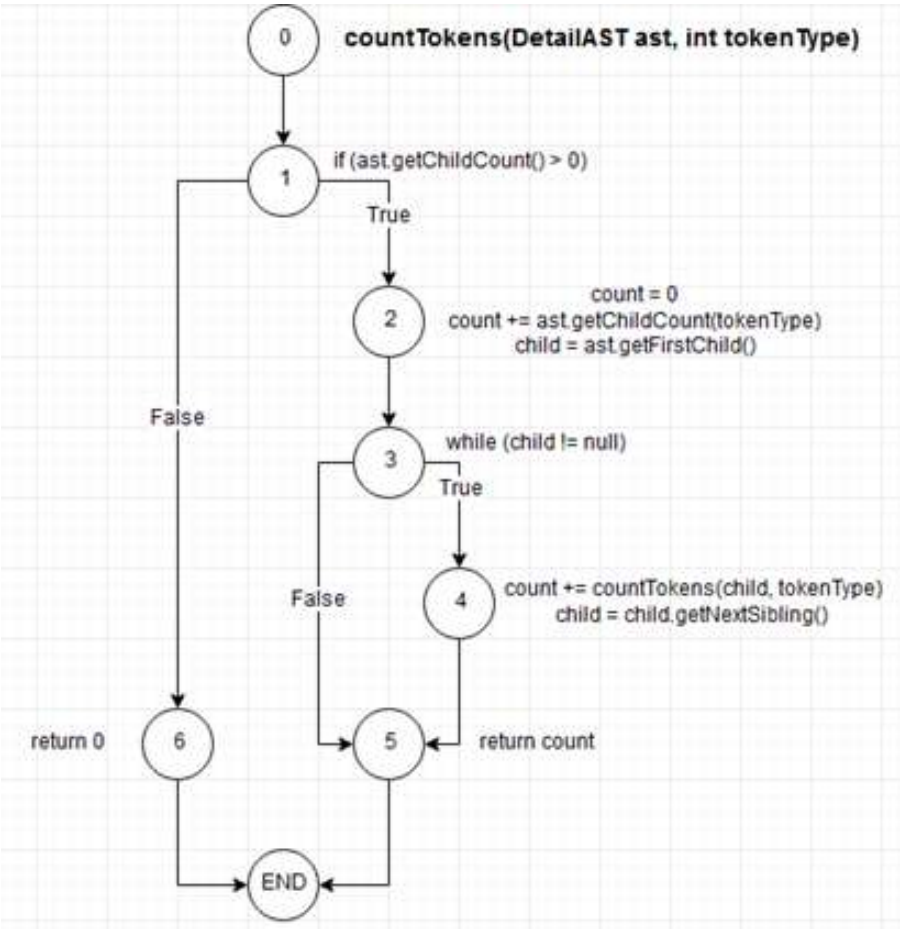
T1	Tree with no nodes (empty file)	dpu(ast,0) = (1,END)
T2	Tree with no comments	dpu(ast,0) = {(1,2), (2,4), (4,6)} dcu(child,6) = {8} dpu(child,6) = {(7,END), (7,8)} dcu(child, 8) = {8} dpu(child,8) = {(7,END), (7,8)}
T3	Tree with single and multi-line comment	dpu(ast,0) = { (2,3), (4,5)}

Results and coverage of this test suite are below:



LoopsCheck, CastsCheck, and VariablesCheck:

We decided to group these 3 checks together because they are all identical. They use a foreach loop in the visitToken() function which iterates over each tokenType and then recursively counts the occurrences of that type inside the countTokens() function. The only difference between the 3 checks are the tokenTypes that are included in the foreach loop Because the visitToken() function is just a foreach loop, we are only including the CFG of countTokens():



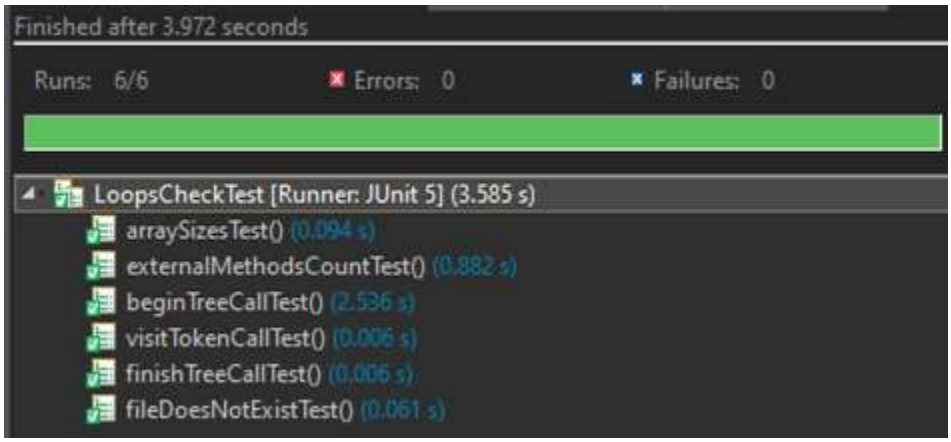
Node(i)	Def(i)	C-Use(i)	Edge(i,j)	P-Use(i,j)
0	ast, tokenType		(0,1)	
1			(1,6) (1,2)	ast ast
2	count, child	ast	(2,3)	
3			(3,5) (3,4)	child child
4	count, child	Child, tokenType	(4,3)	
5		count	(5,END)	
6			(6,END)	

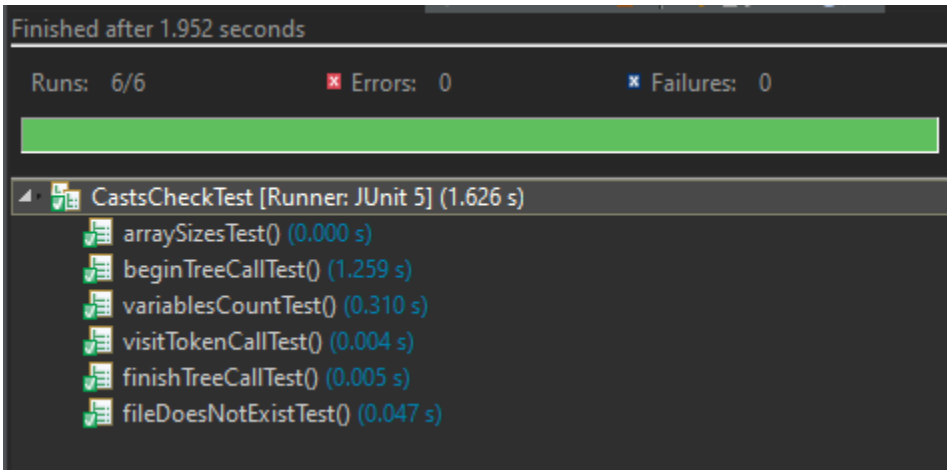
Node(i)	dcu(v,i)	dpu(v,i)
0	dcu(ast,0) = {2} dcu(tokenType,0) = {4}	dpu(ast,0) = {(1,6), (1,2)}
2	dcu(count,2) = {5} dcu(child,2) = {4}	dpu(child,2) = {(3,5), (3,4)}
4	dcu(count,4) = {5} dcu(child,4) = {4}	dpu(child,4) = {(3,5), (3,4)}

Now to create a test suite to cover the paths listed in the table above. Although tokenType is an input for the countTokens() function, it is iterated on from a static list so we know that they will all be covered so we are only going to worry about the DetailAST:

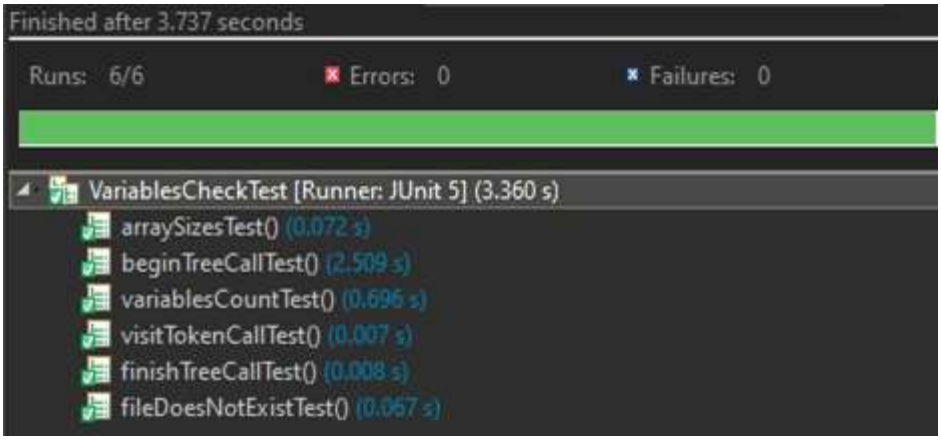
TestCase #	DetailAST ast	Path(s) Covered
T1	A tree with no nodes (empty file)	dpu(ast,0) = (1,6)
T2	A tree with no nodes of the tokenType	dpu(ast,0) = (1,2) dcu(ast,0) = {2} dcu(tokenType,0) = {4} dpu(child,2) = (3,5) dcu(count,2) = {5}
T3	A tree with nodes of the tokenType	dcu(child,2) = {4} dpu(child,2) = (3,4) dcu(count,4) = {5} dcu(child,4) = {4} dpu(child,4) = {(3,5), (3,4)}

Here are the results and coverage of the tests:





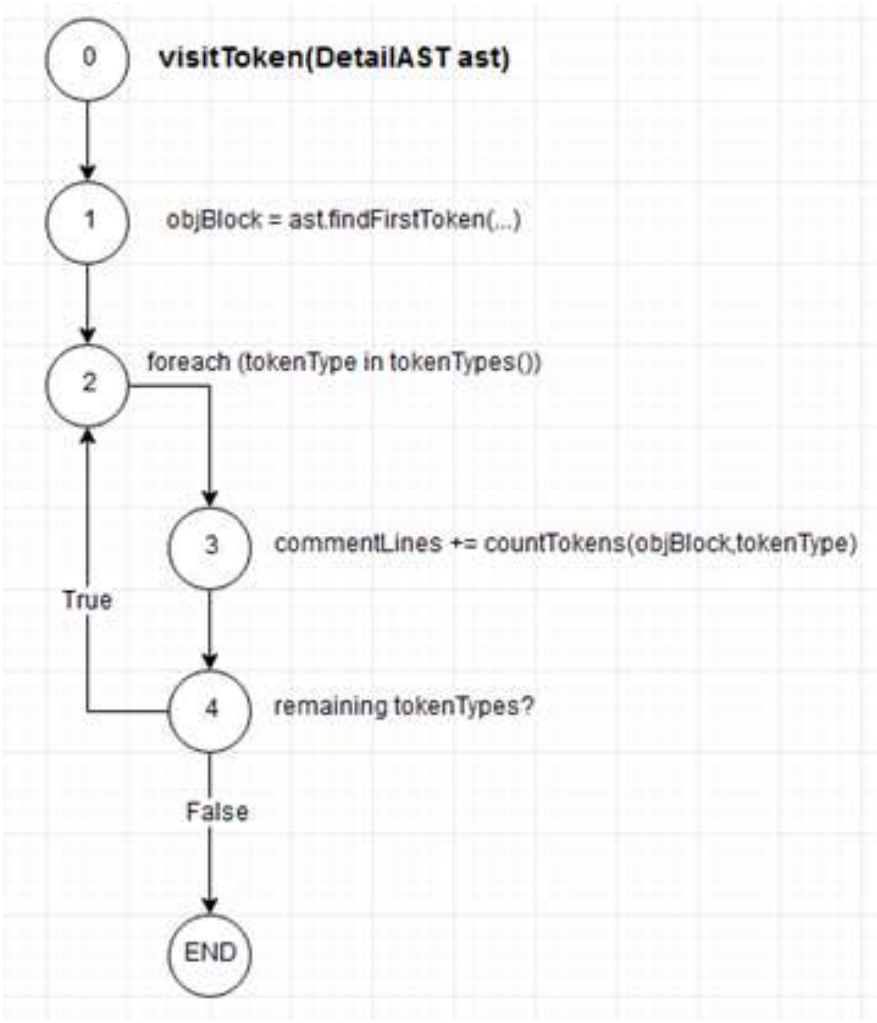
CastsCheck.java	100.0 %	114	0	114
CastsCheck	100.0 %	114	0	114



VariablesCheck.java	100.0 %	114	0	114
VariablesCheck	100.0 %	114	0	114

TotalCommentsCheck:

Unlike the other checks so far, this one doesn't take advantage of recursion like we were previously. Instead, it leverages the visitToken() function which is called on every node of the DetailAST in which it finds the first instance of a comment_content block (since we only care about the comment block as a whole, not how many lines it has) and then increments the counter every time one is found. If a match is not found, countTokens() isn't called as it has a null parameter. Here is the CFG for visitToken():



Node(i)	Def(i)	C-Use(i)	Edge(i,j)	P-Use(i,j)
0	ast, commentLines (class variable)		(0,1)	
1	objBlock	ast	(1,2)	
2	tokenType		(2,3)	
3	commentLines	commentLines, objBlock, tokenType	(3,4)	

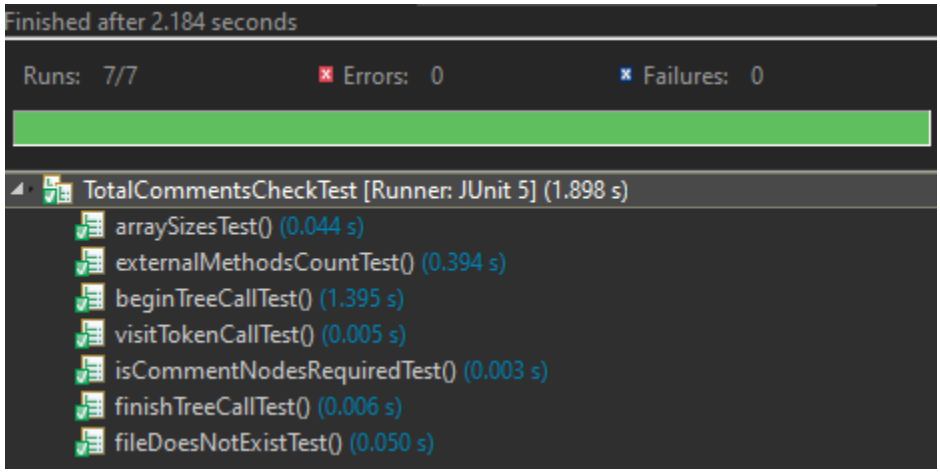
4			(4,2) (4,END)	tokenType tokenType

Node(i)	dcu(v,i)	dpu(v,i)
0	dcu(ast,0) = {1} dcu(commentLines,0) = {3}	
1	dcu(objBlock,1) = {3}	
2	dcu(tokenType,2) = {3}	dpu(tokenType,2) = {(4,2), (4,END)}
3	Dcu(commentLines,3) = {3}	

Note that the paths in red above are infeasible as there will always only be one tokenType, therefore the edge (4,2) will never be taken. Here is our test suite:

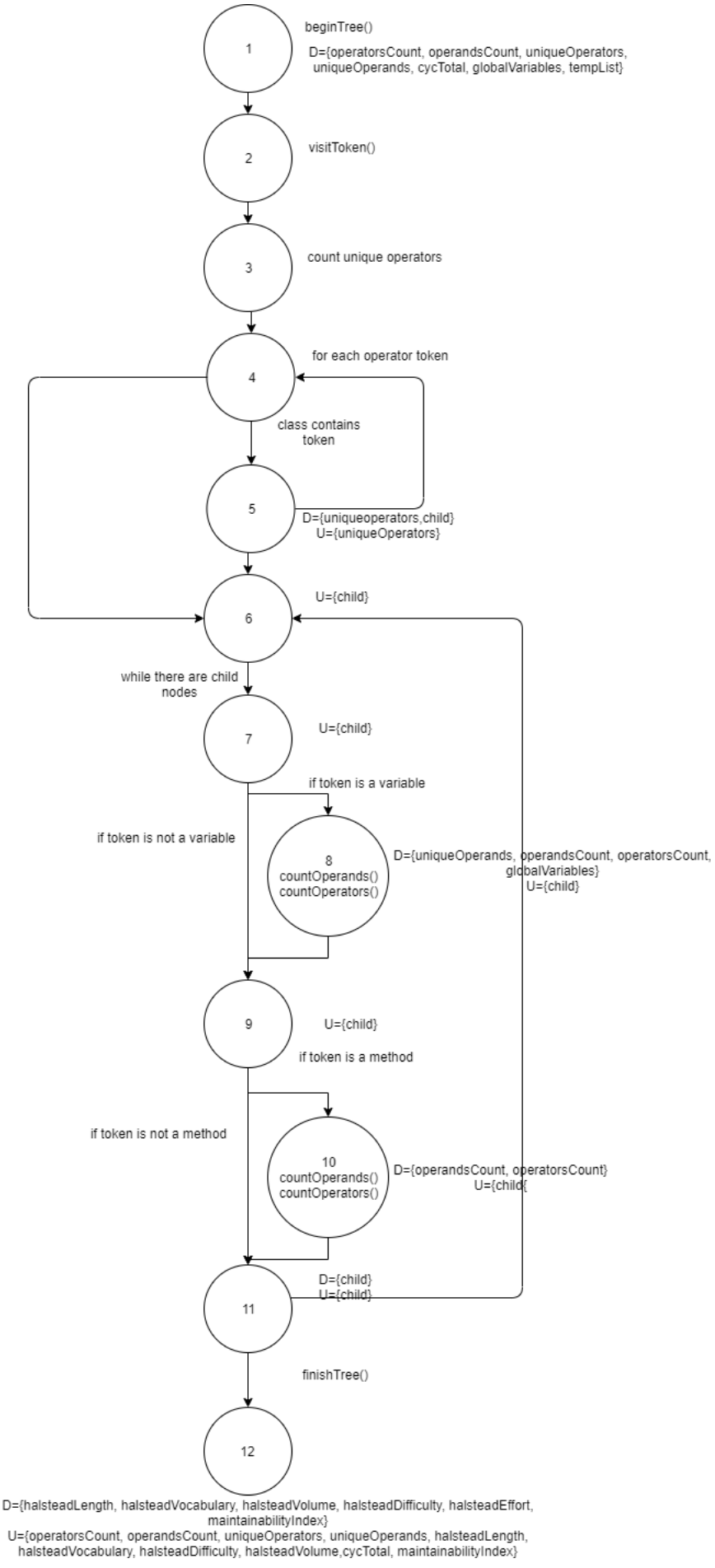
TestCase #	DetailAST ast	Path(s) Covered
T1	Tree with no comments	dcu(ast,0) = {1} dcu(tokenType,2) = {3} dpu(tokenType,2) = (4,END)
T2	Tree with comments	dcu(commentLines,0) = {3} dcu(objBlock,1) = {3} dcu(tokenType,2) = {3}

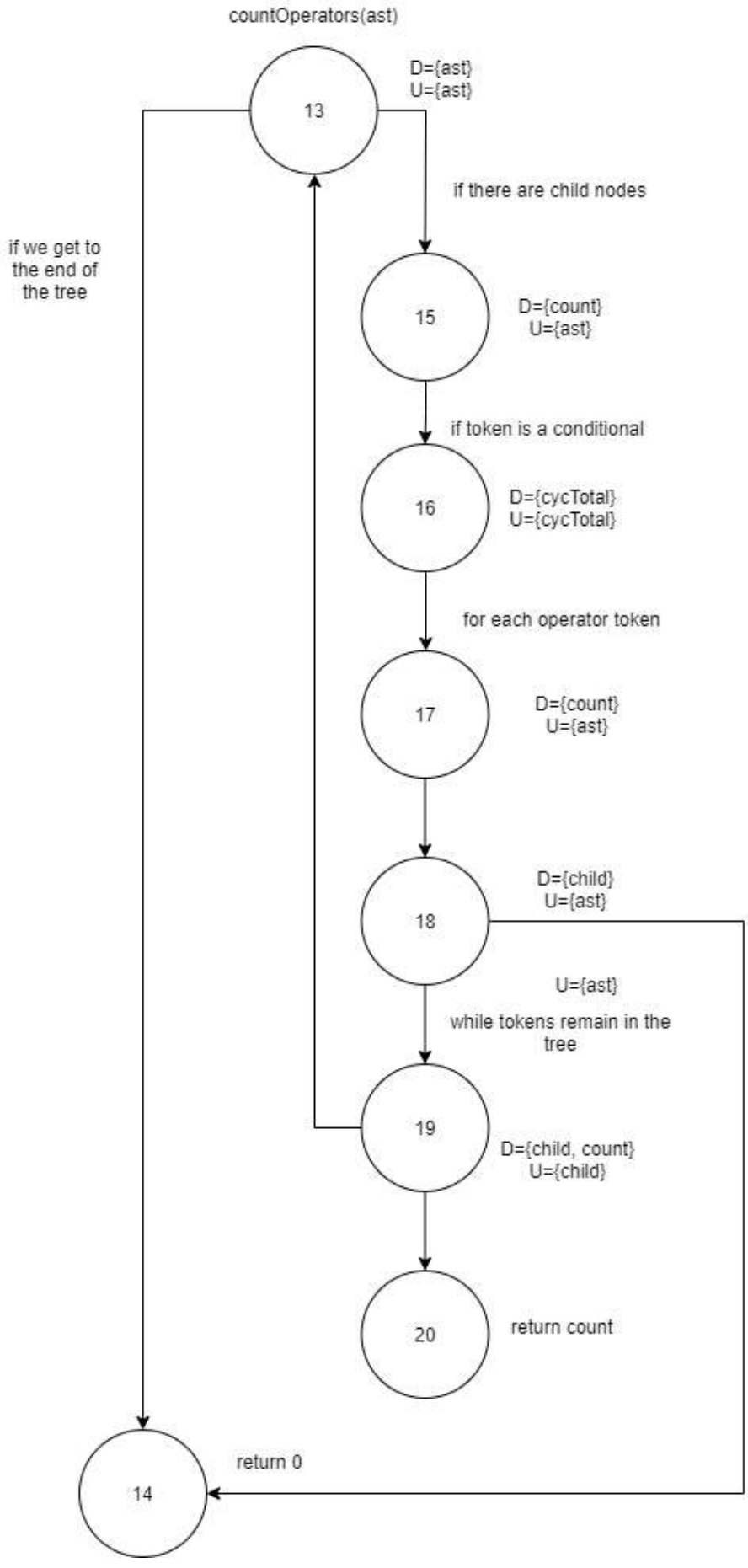
Results and Coverage:

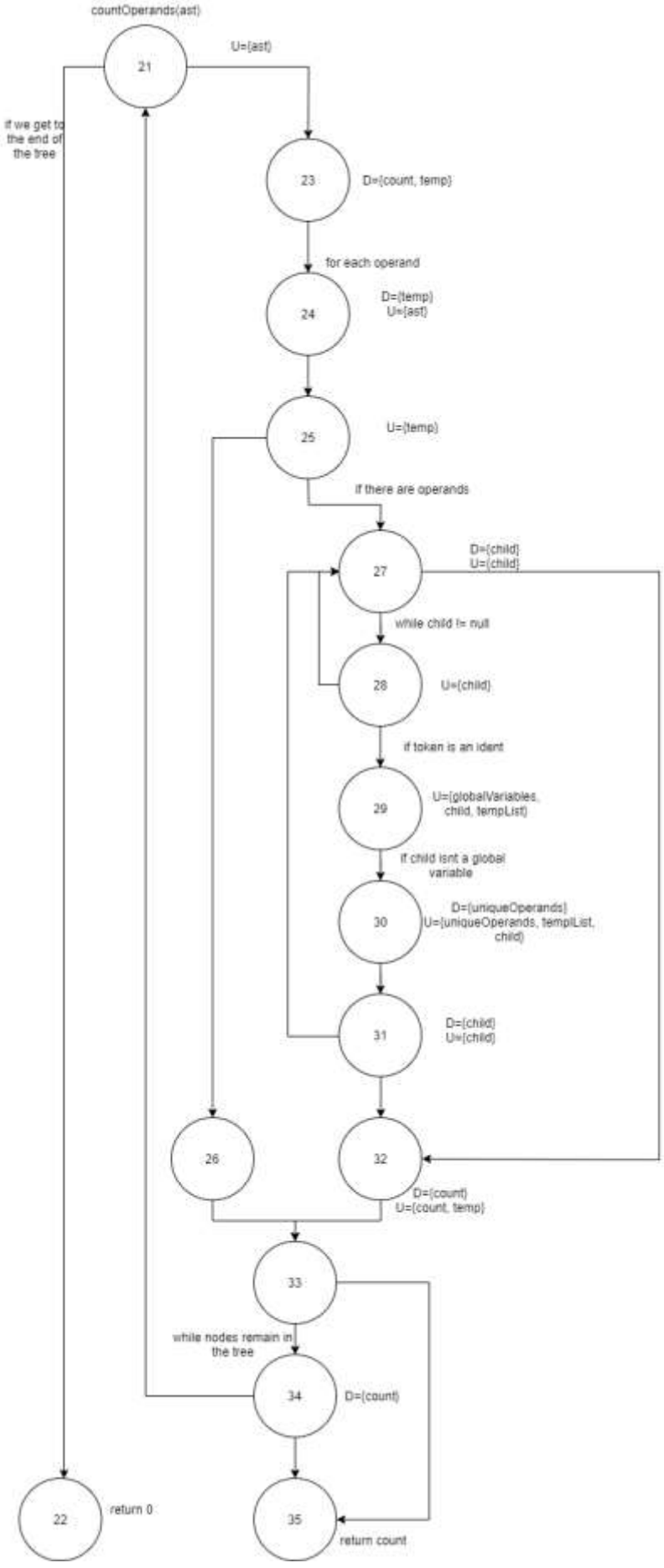


TotalCommentsCheck.java	100.0 %	79	0	79
TotalCommentsCheck	100.0 %	79	0	79

HalsteadMetricsCheck:


1
beginTree()
D={operatorsCount, operandsCount, uniqueOperators,
uniqueOperands, cycTotal, globalVariables, tempList}
2
visitToken()
3
count unique operators
4
for each operator token
class contains
token
5
D={uniqueoperators,child}
U={uniqueOperators}
6
U={child}
while there are child
nodes
7
U={child}
if token is a variable
if token is not a variable
8
countOperands()
countOperators()
D={uniqueOperands, operandsCount, operatorsCount,
globalVariables}
U={child}
9
U={child}
if token is a method
if token is not a method
10
countOperands()
countOperators()
D={operandsCount, operatorsCount}
U={child{
11
D={child}
U={child}
finishTree()
12
D={halsteadLength, halsteadVocabulary, halsteadVolume, halsteadDifficulty, halsteadEffort,
maintainabilityIndex}
U={operatorsCount, operandsCount, uniqueOperators, uniqueOperands, halsteadLength,
halsteadVocabulary, halsteadDifficulty, halsteadVolume,cycTotal, maintainabilityIndex}


countOperators(ast)
13
D={ast}
U={ast}
if there are child nodes
if we get to the end of the tree
15
D={count}
U={ast}
if token is a conditional
16
D={cycTotal}
U={cycTotal}
for each operator token
17
D={count}
U={ast}
18
D={child}
U={ast}
U={ast}
while tokens remain in the tree
19
D={child, count}
U={child}
20
return count
14
return 0


countOperands(ast)
21
U={ast}
if we get to the end of the tree
23
D={count, temp}
for each operand
24
D={temp}
U={ast}
25
U={temp}
if there are operands
27
D={child}
U={child}
while child != null
28
U={child}
if token is an ident
29
U={globalVariables, child, tempList}
if child isnt a global variable
30
D={uniqueOperands}
U={uniqueOperands, tempIList, child}
31
D={child}
U={child}
26
32
D={count}
U={count, temp}
33
while nodes remain in the tree
34
D={count}
22
return 0
35
return count

Node	Def	c-use	edge	p-use
1	operatorsCount, operandsCount, uniqueOperators, uniqueOperands, cycTotal, globalVariables, tempList		{1,2}	
2			{2,3}	
3			{3,4}	
4			{4,5} {4,6}	
5	uniqueOperators, child	uniqueOperators	{5,4} {5,6}	
6			{6,7}	child
7			{7,8} {7,9}	child
8	uniqueOperands, operandsCount, operatorsCount, globalVariables	Child, globalVariables	{8,9} {8,14} {8,21}	

9			{9,10} {9,11}	child
10	operatorsCount, operandsCount	child	{10,11} {10,13} {10,21}	
11	child	child	{11,6} {11,12}	
12	halsteadLength, halsteadVocabulary, halsteadVolume, halsteadDifficulty, halsteadEffort, maintainabilityIndex	operatorsCount, operandsCount, uniqueOperators, uniqueOperands, halsteadLength, halsteadEffort halsteadVocabulary, halsteadDifficulty, halsteadVolume,cycTotal , maintainabilityIndex	END	
13	ast		{13,14} {13,15}	ast
14			{14,8} {14,10}	
15	count		{15,16}	ast
16	cycTotal	cycTotal	{16,17}	

17	count	ast	{17,18}	
18	child	ast	{18,19} {18,14}	
19	count, child	child	{19,13} (19,20}	child
20			{20,8} {20,10}	
21	ast		{21,22} {21,23}	ast
22			{22,8} {22,10}	
23	count, temp		{23,24}	
24	temp	ast	{24,25}	
25			{25,26} {25,27}	temp
26			{26,33}	
27	child		{27,28} {27,32}	child

28			{28,27} {28,29}	child, globalVariables, tempList
29			{29,30}	tempList
30	uniqueOperands	uniqueOperands, tempList, child	{30,31}	
31	child	child	{31,27} {31,32}	
32	count	count, temp	{32,33}	
33			{33,34} {33,35}	
34	count		{34,21} {34,35}	
35			{35,8} {35,10}	

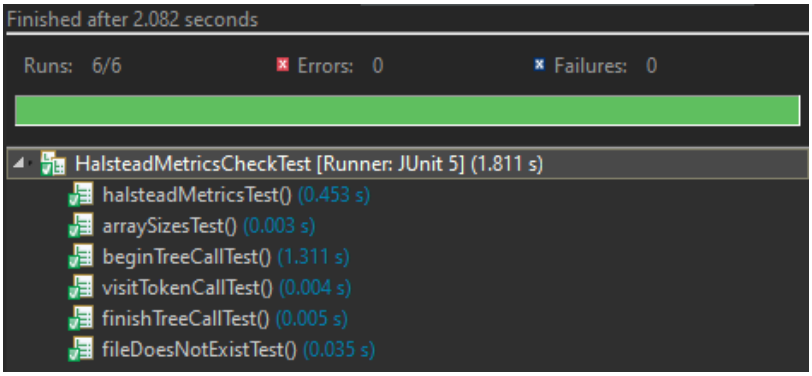
Node(i)	Dcu(v,node)	Dpu(v,node)
1	Dcu(tempList,1)={30}	Dpu(tempList,1)={28,29}

5	Dcu(child,5)={8, 10} Dcu(uniqueOperators,5)={5,12}	Dpu(child, 5)={6, 7, 9}
8	Dcu(uniqueOperands,8)={12} Dcu(globalVariables, 8)={8}	Dcu(globalVariables, 8)={28}
10	Dcu(operandsCount,10)={12} Dcu(operatorsCount,10)={12}	
11	Dcu(child, 11)={11}	
12	Dcu(halsteadLength, 12)={12} Dcu(halsteadVocabulary, 12)={12} Dcu(halsteadVolume, 12)={12} Dcu(halsteadDifficulty, 12)={12} Dcu(halsteadEffort, 12)={12} Dcu(maintainabilityIndex, 12)={12}	
13	Dcu(ast, 13)={17, 18}	Dpu(ast, 13)={13,15}
15	Dcu(count,15)={20}	
16	Dcu(cycTotal,16)={16}	
17	Dcu(count,17)={20}	
18		Dpu(child,18)={19}
19	Dcu(child,19)={19}	Dpu(child,19)={19}
21	Dcu(ast,21)={24}	Dpu(ast,21)={21}
23	Dcu(temp, 23)={32}	Dpu(temp, 23)={25}

24	Dcu(temp, 24)={32}	Dpu(temp, 24)={32}
27	Dcu(child, 27)={30}	Dpu(child,27)={27,28}
30	Dcu(uniqueOperands,30)={30}	
31	Dcu(child,31)={31}	
32	Dcu(count, 34)={35}	

For each test case, since we will not be testing the private methods, the path coverage shown in the tables above keep track of definitions and uses to make sure all variables are used appropriately in the private methods. For the public methods, the AST is the only thing being passed between methods. Since we cannot manipulate the AST itself directly, we will mainly be testing the public methods.

Test Case #	Description
1	Make sure the arrays for all available tokens in the class work properly
2	Make sure beginTree() is called
3	Make sure visitToken() is called
4	Make sure finishTree() is called
5	Call the check with full values
6	Handle the exception where no file is found



HalsteadMetricsCheck.java	100.0 %	751	0	751
HalsteadMetricsCheck	100.0 %	751	0	751

Black Box Testing

Each class's Fault Models are referenced in the sections below. To see the Fault Models, view the CheckFaultModels document.

Test Cases Based off Fault Models

For simplicity sake, when possible all faults in the respective fault model will be addressed in a single test case

Number of Casts

TC1		
Addresses	Fault	Code

FM1	1, 2, 3	`public static void main(String[] args) {`
		
		`int a = 2;`
		`double d = (double)a; // double d = (double)a;`
		`a = (int)d;`
		`/*`
		`double d = (double)a;`

		*/
		// implicit cast
		int i = 20;
		int j = 40;
		float k = i + j;
		// reflection cast
		Object o = "str";
		String tempStr = String.class.cast(o);
		}

Number of Expressions

TC2		
Addresses	Fault	Code

FM2	1, 2, 3	public static void main(String[] args) {
		
		int x = 2 + 3; // int x = 2 + 3
		int y = 3 + 4;
		/*
		int x = 2 + 3
		*/
		Boolean test = (2 != 5);
		int z = 4 + 5 % 4 - 1;
		}

External Method References

TC3		
Addresses	Fault	Code
FM3	1, 2	class ExternalMethodsCheckTestCode {

		@Override
		public String toString(){
		return "dumbo";
		}
		
		public void theMethod() {
		double d = Math.cos(Math.PI);
		// Math.cos(Math.PI);
		Main myMain = new Main();
		// Local method reference
		String test = this.toString();
		// External method reference
		System.out.println(test);
		}
		}

Number of Comments

TC4		
Addresses	Fault	Code

FM4	1, 2, 3, 4, 5	public static void main(String[] args) {
		// This is the most fun I've had in years
		int x = 75; // Grandma's age
		int y = 75;
		/*
		This program finds loops through each
		amazing number from 0 to Grandma's age.
		it multiplies her age by 2.
		*/
		
		for(int i = 0; /*must be less than her age*/ i < 75; i++){
		y++;
		}

		// The result is // her age * 2
		/*
		Sometimes I like
		/*
		To put comments within comments
		/*
		within comments
		*/
		}

Local Method References

TC5		
Addresses	Fault	Code

FM5	1, 2	class LocalMethodsCheckTestCode {
		
		@Override
		public String toString(){
		return "dumbo";
		}
		
		public static void main(String[] args) {
		double d = Math.cos(Math.PI);
		// Math.cos(Math.PI);
		Main myMain = new Main();
		// Local method reference
		String test = myMain.toString();
		// External method reference
		System.out.println(test);
		}
		}

Number of Looping Statements

TC6		
Addresses	Fault	Code
FM6	1, 2, 3, 4, 5, 6	

```
class LoopsCheckTestCode {
 public static void main(String[] args) {
   for(int i = 0; i < 1000; i++){
     do {
       i = i + 11;
       System.out.println("Heyo");
       for (int j = 0; j < 10; j++){
         i--;
       }
     } while (i < 800);
   }
   //for (int j = 0; j < 10; j++){
   //  System.out.println("Heyo");
   //}
   /*
   int test = 15;
   while (test >= 0){
     test--;
   }
   */
 }

  public static void forLoop(){
    for (int i = 0; i < 10; i++){
      System.out.println("Just existing for loop");
    }
  }

  public static void whileLoop(){
   // infinite loop
    while (true){
      System.out.println("While loop");
    }
  }
```

		public static void doWhileLoop(){
		do {
		System.out.println("Just existing do while loop");
		break;
		} while (true);
		}
		}

Number of Lines of Comments

TC7		
Addresses	Fault	Code

FM7	1, 2, 3, 4, 5	public static void main(String[] args) {
		// This is the most fun I've had in years
		int x = 75; // Grandma's age
		int y = 75;
		/*
		This program finds loops through each
		amazing number from 0 to Grandma's age.
		It multiplies her age by 2.
		*/
		
		for(int i = 0; /*must be less than her age*/ i < 75; i++){
		y++;
		}
		
		// The result is // her age * 2
		/*
		Sometimes I like
		/*
		To put comments within comments
		/*
		within comments

		*/ }

Number of Variable Declarations

TC8		
Addresses	Fault	Code

FM8	1, 2, 3, 4, 5	class VariablesCheckTestCode {
		
		public String name;
		private double multiplier = 4.5;
		
		public static void main(String[] args) {
		int x = 1;
		int y = 2;
		// int swag = 3000
		
		/*
		String coolCat = "Rebecca";
		*/
		
		int j, p = 80;
		String a, c = "";
		x = 6;
		y = x++;
		
		Main classInstance = new Main();
		}
		}

Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Number of Operators, Number of Operands, Maintainability Index

TC9		
Addresses	Fault	Code
FM9	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17	

```
class HalsteadMetricsCheckTestCode {
 private int globalVariable = 0;

 public static void main(String[] args) {
 }

  public void firstMethod() {
    int first = switchStatement(10);
    int second = switchStatement(20);
    int third = switchStatement(15);
  }
 /*
 expression = 5 + 5 + 5;
 I love writing stuff
 globalVariable = 47 % 5;
 */
  public void ifStatement() {
    globalVariable = 1;
    globalVariable = 2;
    if (globalVariable > 0) {
      int a = 2 + 2;
      int b = 10;
      b = a + 4;
      if (b > 5) {
        a = 0;
      }
    }
  }
 // number = number + 54 / 6;
  public int switchStatement(int number) {
    switch (number) {
    case 10:
      number++;
      break;
    case 20:
      number--;
      break;
    default:
     number = number + 3 / 35 % 3;
      break;
    }
```

```
    return number;
  }

  public void emptyMethod() {
  }

public void checkOperators(){
  int value = 10000;
  value = value + 1000;
  value = value - 1000;
  value = value * 2;
  value = value / 2;
  value++;
  value--;

  value += value;
  value -= value;
  value *= 2;
  value /= 2;
  value ^= 2;
}
/*
Testing lines of comment percentage
*/
public void checkComparisonOps(){
  int x = 3;
  int y = 4;
  if(x < y){
    System.out.println("1");
  }
  if(y > x){
    System.out.println("2");
  }
  if(y == (x+1) ){
    System.out.println("3");
  }
  if(x != y){
    System.out.println("4");
  }
  if(x <= y){
    System.out.println("5");
  }
  if(y >= x){
    System.out.println("6");
  }
  if(y >= x){
```

```
      System.out.println("6");
    }
    if(y > x && x < y){
      System.out.println("7");
    }
    if(y > x || x < y){
      System.out.println("8");
    }
    if(!(x > y)){
      System.out.println("9");
    }
  }
}
```

Detailed reports with tables and figures of the test cases, what they are testing, if the tests have passed

Test Case Details						
Test Case	Fault Model	Checks	Passed?	Result	Remarks if any	Coverage (%)
TC1	FM1	Number of Casts	Yes	2	Cast method invocation, commented casts, and implicit casts do not get counted.	100
TC2	FM2	Number of Expressions	Yes	4	Commented expressions are not added	100
TC3	FM3	External Method References	Yes	2	Overwritten external method references and commented references are not counted	100
TC4	FM4	Total Comments	Yes	6	Comments within comments are counted as 1	100
TC5	FM5	Local Method References	Yes	2	Overwritten methods are correctly identified	100
TC6	FM6	Number of Looping Statements	Yes	6	Did not count loops in comments	100

TC7	FM7	Lines of Comments	Yes	11	Did not count extra nested comments	100
TC8	FM8	Number of Variable Declarations	Yes	9	Class instantiations are counted, redefinitions are not	100
TC9	FM9	Halstead Length, Halstead Vocabulary, Halstead Volume, Halstead Difficulty, Halstead Effort, Number of Operators, Number of Operands, Maintainability Index	Yes	Difficulty: 17.4, Effort: 42171.834 Length: 343, Vocabulary: 134, Volume: 2423.67, Maintainability Index: 0.7652, Operands: 126, Operators: 217	Each type of operator is recognized. Operators and operands in comments are not counted	100

Notable Bugs:

The only notable bug we discovered involved our LocalMethodsCheck. While running the test, we discovered that there were two calls to traverse the tree with counting the number of times a local method was invoked. Despite this check functioning in the past, the test discovered the problem when the CheckStyle module would try to load the file's DetailAST, which was already completed from the initial loading of the file, so loading the tree a second time was not able to be done and threw an exception. Once we removed the second call to traverse the tree, the check was correctly running again.

Distribution of work:

Team Rebecca had the following base distribution:

- Daniel: Check Test Code compiling
- Juan: White Box analysis and diagrams for various tests
- Brandon: White Box analysis and diagrams for various tests
- Rebecca: Black Box analysis and diagrams for various tests

Our team was able to have a fair distribution of work and gave input on the other portions as we saw fit. We had numerous team meetings to discuss our situation and to make sure we were all up to date with regards to this milestone to be sure that we had ample time to put everything together and that the documentation was accurate, as well as have time to address any potential bugs or voids the documentation would reveal.