# CPTS 422 Documentation Report

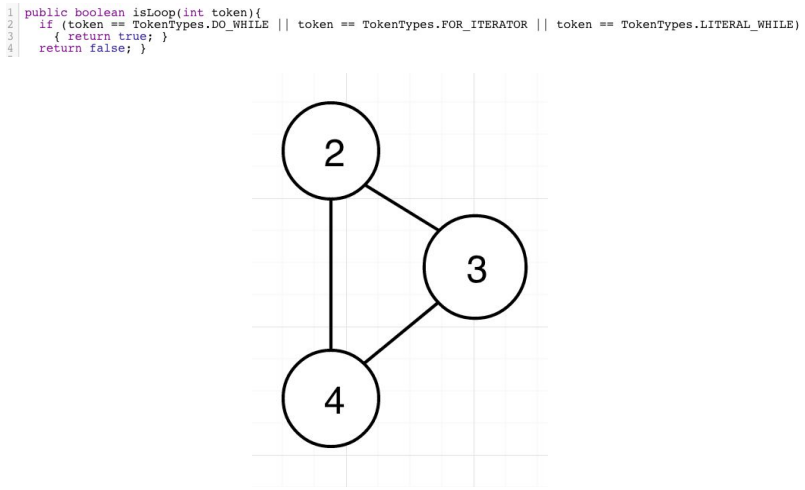
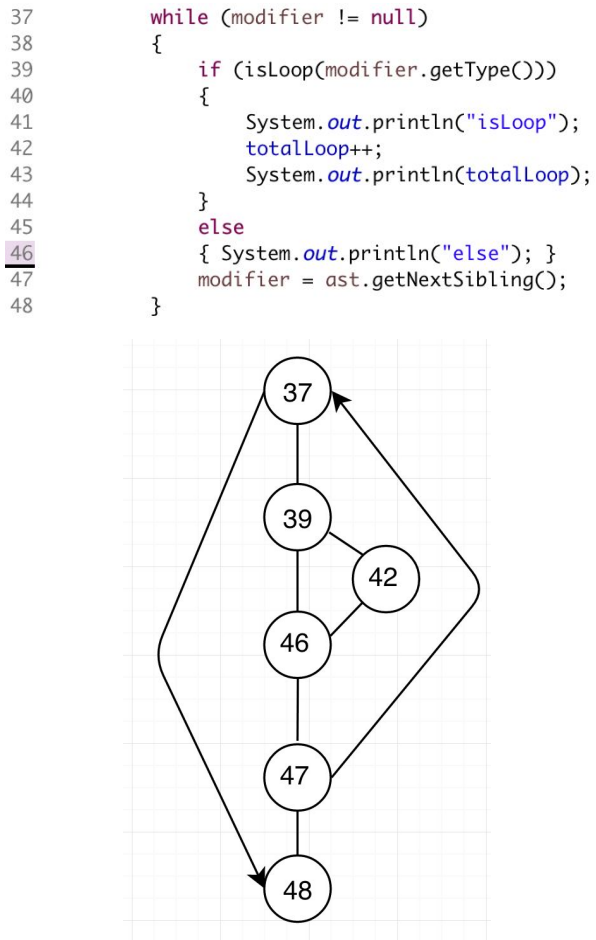
| | |
|---|---|
| Check Name | LoopCountCheck() |
| Check Description | This check returns the number of loops (for, do while, and while) by checking the token types in a DetailAST. |
| **Black-Box Testing** | For my Black-Box testing, I utilized Decision Table Testing. I thought this was fitting for both Loop Count and Distinct Operator Checks because (in the case of my Loop Count Check) for every loop indicator found, I increment the total loop count. I found that this my original testing from Deliverable 1 was very similar to Decision Table Testing, but I did not test my code deep enough. As for code coverage, I believe just using the Decision-Table testing was enough since I only needed to check if the token indicators were being picked up. Due to the fact that PowerMock disables EclEmma code coverage, I just used a lot of print statements everywhere to keep track of what each test was covering. With that in mind, I believe I reached 90%+ percent code coverage.<br>The main problems I found with my Black-box test was that my visitToken() wasn't being implemented correctly. I ran into a null exception error when I tried to access an empty AST. I also found out that my code was not truly looping through every token in the tree, and that it only checked the first child. I wasn't too sure how to implement adding sibling to my tree so I ended up Black-Box Testing my isLoop()which is called in visitToken(). My isLoop() function works perfectly fine and returned the appropriate value. I isolated the issue to be within my visitToken() function. |
| Decision Table for isLoop() |  |

```
@Test
    public void isLoopTest()
    {
        LoopCountCheck myCheck = spy(new LoopCountCheck());

        // checking if correct tokens return true
        assertTrue(myCheck.isLoop(TokenTypes.FOR_ITERATOR));
        assertTrue(myCheck.isLoop(TokenTypes.DO_WHILE));
        assertTrue(myCheck.isLoop(TokenTypes.LITERAL_WHILE));

        // checking if non loop tokens return false
        assertFalse(myCheck.isLoop(TokenTypes.ABSTRACT));
        assertFalse(myCheck.isLoop(TokenTypes.BOR));
        assertFalse(myCheck.isLoop(TokenTypes.ENUM_DEF));
    }
```

<table>
<tr><td></td><td>
<table>
<tr><th>Conditions</th><th>Actions</th></tr>
<tr><td>C1: For Loop Token Encountered</td><td>A1: Increment loop count</td></tr>
<tr><td>C2: Do While Loop Token Encountered</td><td>A2: Increment loop count</td></tr>
<tr><td>C3: While Loop Token Encountered</td><td>A3: Increment loop count</td></tr>
<tr><td>C4: Not a token</td><td>A4: Error</td></tr>
<tr><td>C5: Unacceptable Token</td><td>A5: Error</td></tr>
</table>
</td></tr>
<tr><td>visitToken()</td><td>

```
35          LoopCountCheck myCheck = spy(new LoopCountCheck());
36
37          // equals 0
38          assertEquals(0, myCheck.getTotalLoopCount());
39
40          DetailAST astMock = PowerMockito.mock(DetailAST.class);
41          DetailAST astChildMock = PowerMockito.mock(DetailAST.class);
42
43          // C1 -> A1
44          when(astMock.findFirstToken(TokenTypes.FOR_ITERATOR)).thenReturn(null);
45          myCheck.visitToken(astMock);
46          assertEquals(1, myCheck.getTotalLoopCount());
47
48          // C2 -> A2
49          when(astMock.findFirstToken(TokenTypes.DO_WHILE)).thenReturn(null);
50          myCheck.visitToken(astMock);
51          assertEquals(2, myCheck.getTotalLoopCount());
52
53          // C3 -> A3
54          when(astMock.findFirstToken(TokenTypes.LITERAL_WHILE)).thenReturn(null);
55          myCheck.visitToken(astMock);
56          assertEquals(3, myCheck.getTotalLoopCount());
57
58          // C4 -> A5
59          when(astMock.findFirstToken(TokenTypes.ASSIGN)).thenReturn(null);
60          myCheck.visitToken(astMock);
61          assertEquals(3, myCheck.getTotalLoopCount());
62
63      }
64
```

</td></tr>
<tr><td>White-Box Testing</td><td>For White-Box Testing, I didn’t really code as much seeing as though the code would be similar to the way I Black-Box tested. Instead, I made a Control Flow Graph (CFG) to see how data is manipulated throughout the program. After making the CFG’s, I can see that my tests cover all possible edges and all possible paths. I also made a Def-Use table to see all definition-use paths which helped me see all possible paths. I created my tests based off computational/predicate uses. As for code coverage, I’d say I covered about 90%+ of the code. From my Black-Box tests, I</td></tr>
</table>

| | |
|---|---|
| | assume I reached every possible branch and edge, so getting a higher coverage would not be possible. White-Box Testing only ensured my tests from the prior tests. |
| CGF for isLoop() |  |
| CFG for visitToken() |  |

Def-Use Table for isLoop()

| Variable | Definition Line | Use Line |
|---|---|---|
| token | 1 | 2 |

Def-Use Table for visitToken()

| Variable | Definition Line | Use Line |
|---|---|---|
| myCheck | 35 | 38, 46, 51, 56, 61 |
| astMock | 40 | 44, 45, 49, 50, 54, 55, 39, 60 |

P-use and C-use for isLoop()

Variable

| | token |
|---|---|
| Node i | P-use \| C-use |
| 1 | X \| |
| 2 | X \| |

P-use and C-use for visitToken()

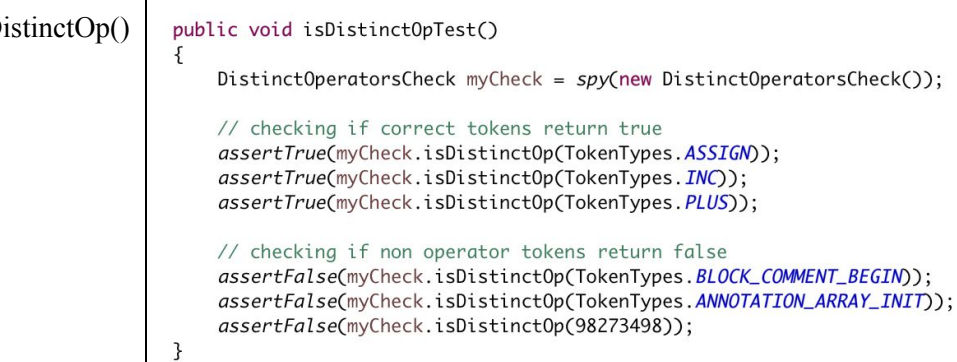
Variable

| | modifier | isLoop | totalLoop |
|---|---|---|---|
| Node i | P-use \| C-use | P-use \| C-use | P-use \| C-use |
| 37 | X \| | | |
| 39 | | X \| | |
| 42 | | | \| X |
| 48 | X \| | | |

Conclusion

I can say that I reached high coverage with both testing techniques. I believe both coverage criteria performed well. I do think Black-Box testing was enough coverage for these checks, and White-Box just ensured the outcomes. From Deliverable 1 to Deliverable to, code coverage increased dramatically, I would say

| | |
|---|---|
| | from 20% code coverage to 90%+. Adding the Black-Box test cases, I could see the that unit and integration testing could have been enough if I had more specific test cases which I did with Black-Box. Along the way, I was somewhat able to rewrite my checks to do what I wanted it to do, but I still ran into the problem of not knowing how to implement a "nextSibling". |

| | |
|---|---|
| Check Name | DistinctOperatorsCheck() |
| Check Description | This check returns the number of distinct operators by checking the token types in a DetailAST. |
| **Black-Box Testing** | As explained in the previous check, I used Decision Table Testing for my Black-Box technique. I found that this was the best way to test the code coverage since I just have to check token types. I did not test for all token types because I felt like it was unnecessary to test if all tokens fulfilled their actions. I tested 3, and I assumed the rest did the appropriate action. |
| Decision Table for isDistinctOp() |  |

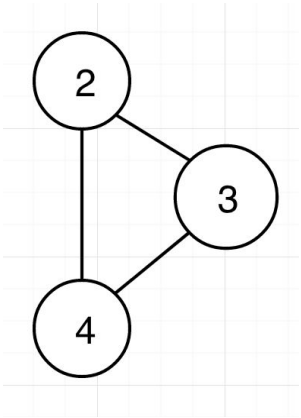
| Conditions | Actions |
|---|---|
| C1: Distinct Op Encountered | A1: Increment count |

<table>
<tr><td></td><td>
<table>
<tr><td>C2: ^^</td><td>A2: Increment count</td></tr>
<tr><td>C3: ^^</td><td>A3: Increment count</td></tr>
<tr><td>C4: Not a token</td><td>A4: Error</td></tr>
<tr><td>C5: Unacceptable Token</td><td>A5: Error</td></tr>
</table>
</td></tr>
<tr><td>visitToken()</td><td>

```
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());
```

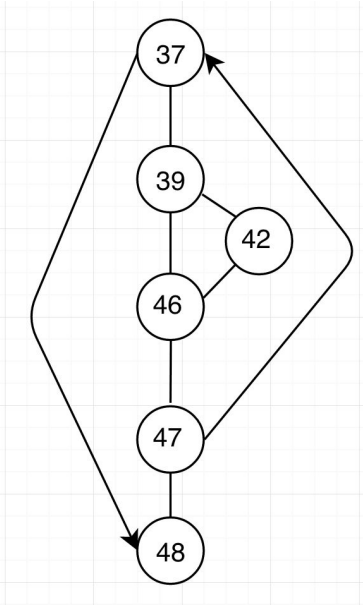
</td></tr>
<tr><td>White-Box Testing</td><td>The White-Box Technique for this check is the exact same thing for the previous check. Even though I did not test every single possible acceptable token, I still believe I reached 90%+ code coverage.</td></tr>
<tr><td>CGF for isDistinctOp()</td><td>Although the node numbers are off, the CFG is very similar to the previous check. We just want to check if the acceptable tokens types return true, otherwise return false. Therefore the basic CFG for this would be an if/else CFG.</td></tr>
</table>

```
64    public boolean isDistinctOp(int token)
65    {
66        System.out.println("inside isDistinctOp");
67        if (token == TokenTypes.ASSIGN || token == TokenTypes.BAND|| token == TokenTypes.BAND_ASSIGN || token == TokenTypes.BNOT ||
68            token == TokenTypes.BOR || token == TokenTypes.BOR_ASSIGN || token == TokenTypes.BSR || token == TokenTypes.BSR_ASSIGN || token == TokenTypes.BXOR ||
69            token == TokenTypes.BXOR_ASSIGN || token == TokenTypes.COLON || token == TokenTypes.DEC || token == TokenTypes.DIV || token == TokenTypes.DIV_ASSIGN ||
70            token == TokenTypes.DOT || token == TokenTypes.EQUAL || token == TokenTypes.GE || token == TokenTypes.GT || token == TokenTypes.INC || token == TokenTypes.INDEX_OP ||
71            token == TokenTypes.LAND || token == TokenTypes.LE || token == TokenTypes.LITERAL_INSTANCEOF || token == TokenTypes.LNOT || token == TokenTypes.LOR ||
72            token == TokenTypes.LT || token == TokenTypes.MINUS || token == TokenTypes.MINUS_ASSIGN || token == TokenTypes.MOD || token == TokenTypes.MOD_ASSIGN ||
73            token == TokenTypes.NOT_EQUAL || token == TokenTypes.PLUS || token == TokenTypes.PLUS_ASSIGN || token == TokenTypes.POST_DEC || token == TokenTypes.POST_INC ||
74            token ==TokenTypes.QUESTION || token == TokenTypes.SL || token == TokenTypes.SL_ASSIGN || token == TokenTypes.SR || token == TokenTypes.SR_ASSIGN ||
75            token == TokenTypes.STAR || token == TokenTypes.STAR_ASSIGN || token == TokenTypes.UNARY_MINUS || token == TokenTypes.UNARY_PLUS)
76        {
77            System.out.println("true");
78
79            return true;
80        }
81        System.out.println("false");
82
83        return false;
84    }
```



<table>
<tr>
<td>CFG for visitToken()</td>
<td>

```
DetailAST modifier = ast.getFirstChild();

while (modifier != null)
{
    if (isDistinctOp(modifier.getType()))
    {
        System.out.println("isDistinctOp");
        count++;
        System.out.println(count);
    }
    else
    { System.out.println("else"); }
    modifier = ast.getNextSibling();
}
```




The CFG for visitToken(), just like the visitToken() for the previous check is also implemented the exact same way, so the CFG would be the same. Although the node numbers are off, we can still see how the corresponding statements, definition, and uses are used throughout the function, which is also the use of a Def-Use table and P-use/C-use table. Since the checks are almost identical in implementation a lot of the tests, diagrams, and tables were either the same or very similar.
</td>
</tr>
<tr>
<td>Def-Use Table for isDistinctOp()</td>
<td>

| Variable | Definition Line | Use Line |
|---|---|---|
| token | 1 | 2 |

</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</table>

Def-Use Table for visitToken()

| Variable | Definition Line | Use Line |
|---|---|---|
| myCheck | 35 | 38, 46, 51, 56, 61 |
| astMock | 40 | 44, 45, 49, 50, 54, 55, 39, 60 |

P-use and C-use for isDistintOp()

| | Variable | |
|---|---|---|
| | token | |
| Node i | P-use | C-use |
| 1 | X | |
| 2 | X | |

P-use and C-use for visitToken()

| | Variable | | | | | |
|---|---|---|---|---|---|---|
| | modifier | | isLoop | | totalLoop | |
| Node i | P-use | C-use | P-use | C-use | P-use | C-use |
| 37 | X | | | | | |
| 39 | | | X | | | |
| 42 | | | | | | X |
| 48 | X | | | | | |

Conclusion

My conclusions for this check will be the exact same as the previous check --I reached high coverage with both testing techniques, both coverage criteria performed well, I still think Black-Box testing was enough coverage for these checks, and White-Box was not necessarily needed. From Deliverable 1 to Deliverable to, code coverage increased just as dramatically, I would say from 20% code coverage to 90%+.

Structural metrics: *Number of comments* [3]

## Black box testing - Cause-Effect Graphs Testing

The reason of choosing Cause-Effect Graphs Testing as black box testing is because this type of testing focus on visualizing decision tables. It will helps identify each decisions making and easier to track the entire functions coverage manually. The way to do it is to creating nodes as Cause on the left side and Effect on the right side to differentiate the reason and the result. Each Effect will at least connect with one Cause, and the relation in between are the decision for each test. Then after crafting the whole graphs, we are able to locate each decisions easily by following the Cause to the Effect.

The drawback of using Cause-Effect Graphs Testing is we are drawing the graph manually. It is like a double edge sword where you can create a graph easily, but it is easy to missing out Causes, Effects, or even decisions. Every Time we are missing out a component, it is a coverage we are not identifying, and it is also mean the graph is one step further from being truthful.

We are creating the test code right after we complete the graph, so it is dangerous to missing out any components. However because the scale of this project is small and mainly focus on learning the different type of testing, it would be ok to creating the graph manually.

While programmer working in the industry, they tend to use coverage tool and other checking tools over manually because of they are more reliable and more accurate. Some have the ability to automatically create test case out of thin air, some are able to check tests code in under 5 minutes, and some tools can even check all the errors and fix it for you. There are coverage tool called EclEmma, and it check the coverage over all the test code we created. It is very easy to use with their one click checking, and showing which path is cover or not in the code with colors. However, the limitation of Eclemma is it does not work well with powermock, which we are using in this project. It is a known issue on the internet that powermock would disable Eclemma and always return 0% coverage whenever we use powermock.

There are also some other alternative solution with using other coverage tools to check, but this time we are going to manually testing by printing out log onto the console to check whenever it goes through a path. This way we can know which path is being process and which path is not.

## Example: Counting the comment from DeatilAST

```
28    public ArrayList<DetailAST> findChildAST(DetailAST parent, int type)
29    {
30        ArrayList<DetailAST> children = new ArrayList<DetailAST>();
31        DetailAST first_child = parent.getFirstChild();
32        while(first_child != null)
33        {
34            if (first_child.getType() == type)
35            {
36                children.add(first_child);
37            }
38            else
39            {
40                children.addAll(findChildAST(first_child, type));
41            }
42            first_child = first_child.getNextSibling();
43        }
44        return children;
45    }

47    // count function for counting all the comments
48    public int Counts(DetailAST ast)
49    {
50        // set the counts to zero and ready to hold the result and return
51        int counts = 0;
52        ArrayList<DetailAST> temp;
53
54        // find the first token so it can start running
55        DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);
56
57        // running a loop until we count all the TokenTypes we need from the object
58        while(objBlock != null)
59        {
60            temp = findChildAST(objBlock, TokenTypes.COMMENT_CONTENT);
61
62            // counts set to holding all the child for the comments
63            counts = objBlock.getChildCount(TokenTypes.COMMENT_CONTENT);
64        }
65
66        // return counts for ending the method
67        return counts;
68    }
```

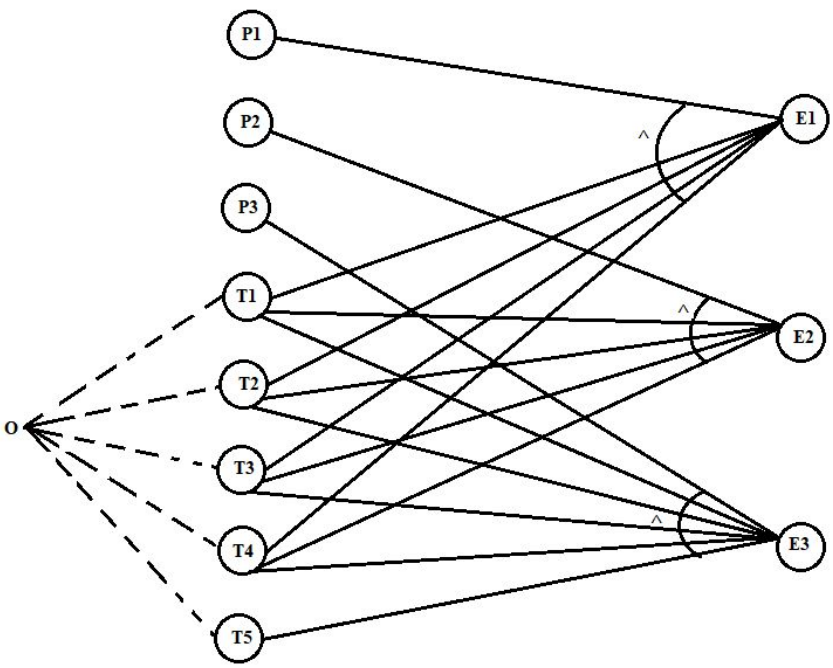
In the class, I have mainly two function/methods, findChildAST and Counts. The findChildAST is to help traverse the DetailAST to the child based on the the node that passed in. The Counts is to count every time we get a TokenTypes.COMMENT_CONTENT to keep track of the number of comment, which also the whole purpose of the check.

Now we know the class, and we are moving onto the test. The function/method Counts is nothing interesting because we are passing the DetailAST as the parameter which is a structural type, and it is going to either pass in a DetailAST type, null or other types that causing errors.

```java
@Test
public void testNumCom()
{
    NumberComment mockNumCom = mock(NumberComment.class);
    DetailAST mockAST = PowerMock.createMock(DetailAST.class);

    //PowerMock.mockStaticPartial(NumberComment.class, "Counts");
    //EasyMock.expect(NumberComment.Counts(mockAST));

    assertNull(mockNumCom.findChildAST(null, 0));
    assertNotNull(mockNumCom.findChildAST(mockAST, TokenTypes.COMMENT_CONTENT));
    assertNotNull(mockNumCom.findChildAST(mockAST, TokenTypes.BLOCK_COMMENT_BEGIN));
    assertNotNull(mockNumCom.findChildAST(mockAST, TokenTypes.BLOCK_COMMENT_END));
    assertNotNull(mockNumCom.findChildAST(mockAST, TokenTypes.SINGLE_LINE_COMMENT));

    assertEquals(0,mockNumCom.Counts(null));
    assertNotNull(mockNumCom.Counts(mockAST));

    //System.out.println("checking" );
}
```

## findChildAST

First, we test the findChildAST while passing in nothing, and it will return null as expected, since there are no node to check at the first place, of course there will be no child to be found.
Then, we pass in a mockAST with a mock NumberComment class and try to find TokenTypes.COMMENT_CONTENT, which in the result, it will not be null in this case, because of we are passing in a mockAST which can traverse and to be found. After that, we will try to pass in different TokenTypes, which they are all appear to be not null.




**Parameter**

**Cause:**
P1: mockAST
P2: null
P3: Invaild type (string, int, double, etc.)
T1: TokenTypes.COMMENT_CONTENT
T2: TokenTypes.BLOCK_COMMENT_BEGIN
T3: TokenTypes.BLOCK_COMMENT_END
T4: TokenTypes.SINGLE_LINE_COMMENT
T5: Invalid type (string, double, ArrayList, etc.)

**Effect:**
E1: not null
E2: null
E3: ERROR

## Counts

First, we are checking if passing in null, will the Counts to be 0, and not be surprise, it does return 0 since there are nothing to be count at the first place.
Then, we passing our mockAST to the Counts, and turn out the return type will not be null, which is exactly what had been implement above.


**Parameter**

**Cause:**
A1: mockAST
A2: null
A3: invalid type (string, int, double, etc.)

**Effect:**
E1: not null
E2: null
E3: ERROR

Since all the decision making are going to be on the findChildAST method, Counts method will only responsible for one to one directional Cause and Effect relationship.

## White box testing - Control Flow Testing

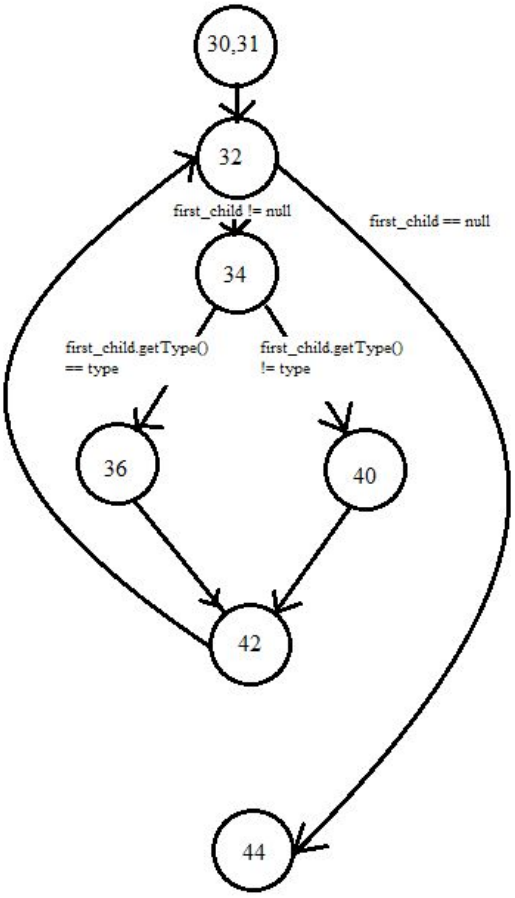
The reason of choosing Control Flow Testing over the other testing is because this type of testing visually shows all the nodes and edges. It also easier for other people to follow how the function flow through different decisions during the call. It is the similar reason of choose Cause-Effect Graphs Testing as black box testing because I believe it is easier to follow and realize if components are missing from the diagram.

Every nodes from the graph corresponds to the line number shown in the code. That way we can compare the line number to the graph to see where is the flow is going with every decision we make. By starting from the top, then we can create test case in our head into a piece of paper to find out what is the path we show be taking to complete the path coverage, by thinking what value we put into the function/method, we can find out what to pass in to get to different branch. Control flow graph will literally become the map to the function/method and show exactly what it takes to get to the result that we want.

Now we are going to create Control Flow Graph (CFG) base on the line number provide from the function/method as below.

## findChildAST

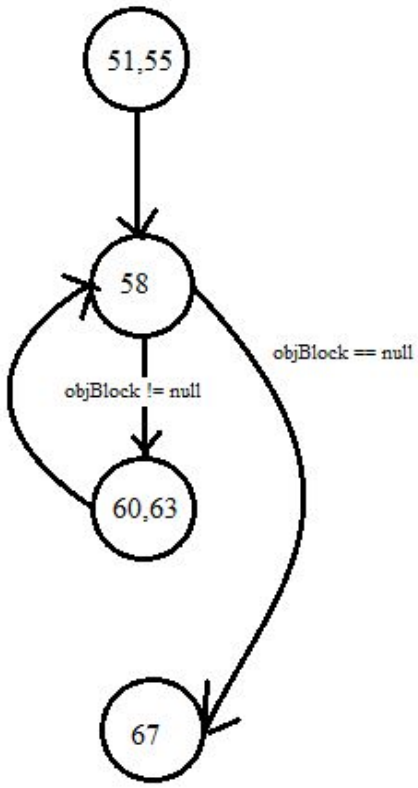
```
28      public ArrayList<DetailAST> findChildAST(DetailAST parent, int type)
29      {
30          ArrayList<DetailAST> children = new ArrayList<DetailAST>();
31          DetailAST first_child = parent.getFirstChild();
32          while(first_child != null)
33          {
34              if (first_child.getType() == type)
35              {
36                  children.add(first_child);
37              }
38              else
39              {
40                  children.addAll(findChildAST(first_child, type));
41              }
42              first_child = first_child.getNextSibling();
43          }
44          return children;
45      }
```




When we follow the CFG above, we can identify we need to take the paths <(30,31), 32>, <32,34>, <34,36>, <36,42>, <42,32>, <32,34>, <34,40>, <40,42>, <42,32>, <32,44> to complete the path coverage, and we need to use the clause of first_child != null, first_child.getType() == type, first_child.getType() != type, and first_child == null to complete the condition coverage.

## Counts

```
47      // count function for counting all the comments
48      public int Counts(DetailAST ast)
49      {
50          // set the counts to zero and ready to hold the result and return
51          int counts = 0;
52          ArrayList<DetailAST> temp;
53
54          // find the first token so it can start running
55          DetailAST objBlock = ast.findFirstToken(TokenTypes.OBJBLOCK);
56
57          // running a loop until we count all the TokenTypes we need from the object
58          while(objBlock != null)
59          {
60              temp = findChildAST(objBlock, TokenTypes.COMMENT_CONTENT);
61
62              // counts set to holding all the child for the comments
63              counts = objBlock.getChildCount(TokenTypes.COMMENT_CONTENT);
64          }
65
66          // return counts for ending the method
67          return counts;
68      }
```




We need to take <(51,55),58>, <58,(60,63)>, <58,67> to complete path coverage. We need to use the clause objBlock!= null, and objBlock == null to complete condition coverage.

By following the CFG from top down, it become easy to see what decision we need to make to complete all kind of coverage with absolute confidence, and with Control Flow Testing as white box testing, I am confident to say the correctness of white box testing are high enough and truthful enough to say it is a strong tactic to be used.

Now we finished one Structural metrics, and the other metrics will be almost identical, since my other test is testing for number line of comment where instead of counting for comment, we count up every time we see a comment, and the only difference is when there are a block comment, then we will count it up each line a time. All the graphs and diagrams are basically the same, and so will be the results.

Now we finish doing black box and white box testing on the function/method, we can compare which one is more useful, truthful, more confidence on saying it has the higher correctness. I would chose black box testing over white box testing because the Cause-Effect graph have more options and thought process to lay out all the input and the possible outcome, when the Control Flow Graph just going to give the possible path base on the existing code. At the end of the day we want to have the confidence to say one testing is better over another, in our case, Cause-Effect graph provide the paths the code didn't show, and show what could have happen if we decide to go through those path, when CFG only show what would happen when we go to existing path, which have less confidence to say it coverage more path than the black box testing. This is why I believe the Cause-Effect Testing as Black box is greater than Control Flow Testing as White box testing.

Kept in mind that we also have the options of using tools to help on the testing. However the tools I encounter have a lot of limitation and end up unable to be used, this is why I am doing the manually testing. If we are able to use proper tools, I believe there will be less problems on the missing out component part and be able to finish testing in a faster paced. In a real world scenario, we are most likely to be force to be using tools to increase to speed of testing and the accuracy, however because we working with a smaller scale, which using a manual test will be good enough and have the confidence to say the test is good enough.

<table>
<tr><td colspan="2">Black box testing for Number of Operand (NumberOfOperandCheck.cs)</td></tr>
<tr><td>Name of methods</td><td>Used testing method and explanation.</td></tr>
<tr><td>int[] getDefaultTokens()</td><td>

**Black box testing technique: Logic Function testing**

```
@Test
public void getDefaultTokensTest()
{
    //Logic Function Testing
    NumberOfOperandCheck myCheck = spy(new NumberOfOperandCheck());
    int[] defaultTokens = myCheck.getDefaultTokens();

    assertTrue(defaultTokens.length == 4);  //length check
    assertTrue(arraySearch(defaultTokens, TokenTypes.PLUS) ||
            arraySearch(defaultTokens, TokenTypes.MINUS)||
            arraySearch(defaultTokens, TokenTypes.STAR)||
            arraySearch(defaultTokens, TokenTypes.DIV));
}
```

The returning value for this method an integer array that containing TokenTypes of PLUS, MINUS, DIV, STAR. While doing blackbox text, the assumption is we cannot see inside of the code. It means I cannot guarantee the order of TokenTypes is also {PLUS, MINUS, DIV, STAR}. In another words we cannot test like:
assertTrue(myCheck == new int[]{TokenTypes.PLUS, TokenTypes.MINUS, TokenTypes.STAR, TokenTypes.DIV});

In order to pass the test, the return value must contain “ALL” four types of operand and the length of array must be 4. I used logic function testing to see if the return value contains all of PLUS, MINUS, DIV, and STAR types and has exactly length of 4.

Here is the logic function I set up:
Let P as array.contains(P),
M as array.contains(M),
D as array.contains(D),
S as array.contains(S)
Z stands for all 4 operand is including.

Then,
Z = PMDS

Below chart is the truth table

| **P(plus)** | **M(minus)** | **D(div)** | **S(star)** | **Z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0(false) |
| 0 | 0 | 0 | 1 | 0(false) |
| 0 | 0 | 1 | 0 | 0(false) |

</td></tr>
</table>

<table>
<tr>
<td></td>
<td>
<table>
<tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0(false)</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0(false)</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0(false)</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0(false)</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>1</td><td>0(false)</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td>0(false)</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0(false)</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0(false)</td></tr>
<tr><td>1</td><td>0</td><td>1</td><td>1</td><td>0(false)</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0(false)</td></tr>
<tr><td>1</td><td>1</td><td>0</td><td>1</td><td>0(false)</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0(false)</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1(true)</td></tr>
</table>
<br>
Finally, we need to check the counts. Let L as Legth<br>
Pass = ZL<br>
<br>
<table>
<tr><td>Z</td><td>L(length)</td><td>Pass</td></tr>
<tr><td>0</td><td>0</td><td>0 (fail)</td></tr>
<tr><td>1</td><td>0</td><td>0 (fail)</td></tr>
<tr><td>0</td><td>1</td><td>0 (fail)</td></tr>
<tr><td>1</td><td>1</td><td>1 (PASS)</td></tr>
</table>
</td>
</tr>
<tr>
<td>getRequiredTokens()</td>
<td>Black box testing technique: Logic Function testing<br>Same as int[] getDefaultTokens(), since the return value is similar to getDefaultTokens() method.</td>
</tr>
<tr>
<td>VisitToken(DetailAST ast)</td>
<td>technique: Equivalence class partitioning test</td>
</tr>
</table>

```
@Test
public void visitTokenTest()
{
    NumberOfOperandCheck myCheck = spy(new NumberOfOperandCheck());
    int[] allpossibleInputs = getAllPossibleTokens();

    //creating input ast
    DetailAST astMock = PowerMockito.mock(DetailAST.class);
    for(int i = 0; i< allpossibleInputs.length; i++)
    {
        DetailAST temp = new DetailAST();
        temp.setType(allpossibleInputs[i]);
        astMock.addChild(temp);
    }
    //The AST has all possible tokens
    myCheck.visitToken(astMock);
    assertTrue(myCheck.count == 4);
}
```

While visiting all tokens in AST, it increments the number of 'count' member variable. Since it is black box testing, we cannot look inside of the code. So just checking if the function "isOperand(int op)" is called is not a good idea. Instead, I made my custom DetailedAST with child ASTs that containing a distinct possible TokenTypes. Since the function will count when there is either {PLUS, MINUS, STAR, DIV}, the expected value of 'count' is 4.

isOperand(int token)

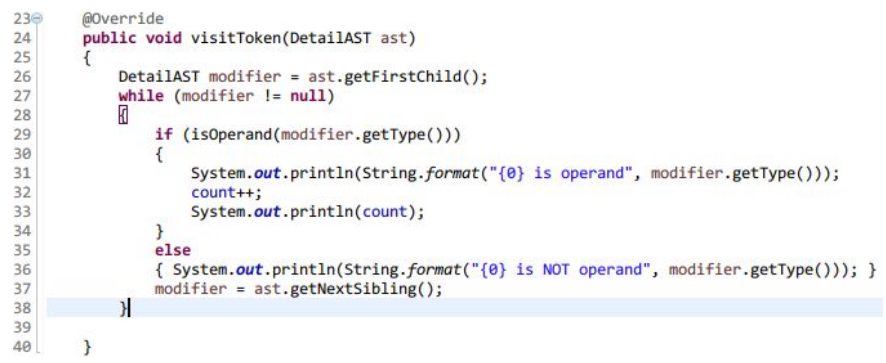
**Black box testing technique:** Equivalence class partitioning test

```
@Test
public void isOperandTest()
{
    NumberOfOperandCheck myCheck = spy(new NumberOfOperandCheck());
    assertTrue(myCheck.isOperand(TokenTypes.PLUS));
    assertTrue(myCheck.isOperand(TokenTypes.MINUS));
    assertTrue(myCheck.isOperand(TokenTypes.STAR));
    assertTrue(myCheck.isOperand(TokenTypes.DIV));
    assertFalse(myCheck.isOperand(TokenTypes.ANNOTATION));
    assertFalse(myCheck.isOperand(TokenTypes.COLON));
    assertFalse(myCheck.isOperand(TokenTypes.TYPE_PARAMETERS));
    assertFalse(myCheck.isOperand(TokenTypes.STAR_ASSIGN));
}
```

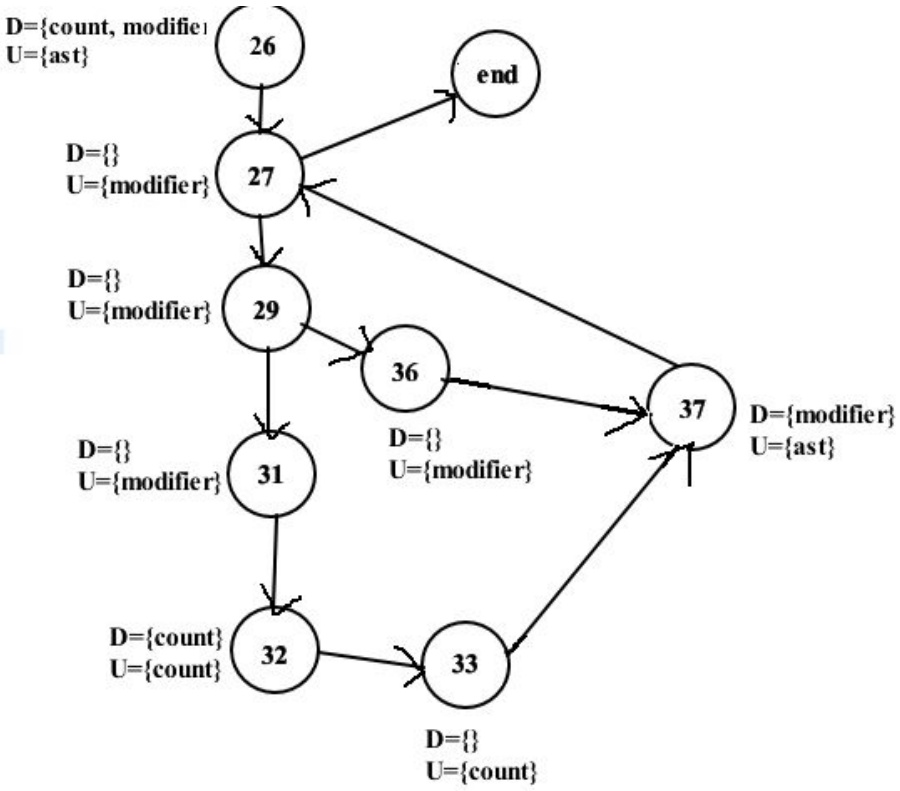
ECT1: token == TokenTypes.PLUS||MINUS||DIV||STAR : valid
ECT2: token != TokenTypes.PLUS||MINUS||DIV||STAR : invalid

| ECT | token | Valid \| invalid |
|---|---|---|
| ECT1 | PLUS | valid |
| ECT1 | MINUS | valid |
| ECT1 | DIV | valid |

| | | | |
|---|---|---|---|
| | ECT1 | STAR | valid |
| | ECT2 | ANNOTATION | invalid |
| | ECT2 | COLON | invalid |
| | ECT2 | TYPE_PARAMETERS | invalid |
| | ECT2 | STAR_ASSIGN | invalid |

| **White box** testing for **Number of Operand (NumberOfOperandCheck.cs)** | |
|---|---|
| getDefaultTokens() | The function returns array of int with entries of TokenTypes.PLUS, MINUS, DIV, and STAR. It is very simple function and it has 100% coverage whenever the function is called, so there is no need to run white box testing. |
| getRequiredTokens() | Same as getDefaultTokens() function. |
| void visitToken(DetailAST ast) |  |

```
23      @Override
24      public void visitToken(DetailAST ast)
25      {
26          DetailAST modifier = ast.getFirstChild();
27          while (modifier != null)
28          {
29              if (isOperand(modifier.getType()))
30              {
31                  System.out.println(String.format("{0} is operand", modifier.getType()));
32                  count++;
33                  System.out.println(count);
34              }
35              else
36              { System.out.println(String.format("{0} is NOT operand", modifier.getType())); }
37              modifier = ast.getNextSibling();
38          }
39
40      }
```




C-use/p-use chart for visitToken(DetailAST ast) function in NumberOfOperandCheck class.

| | modifier | | count | |
|---|---|---|---|---|
| Line | c-use | p-use | c-use | p-use |
| 26 | | x | x | |
| 27 | | x | | |
| 29 | | x | | |
| 31 | | x | | |
| 32 | | | x | |
| 33 | | | | x |
| 36 | | x | | |
| 37 | | x | | |

Variable modifier is used only for p-use. The value is changing to traverse. On the line 32, variable count is used for c-use. It increments the value of count.

Def-use

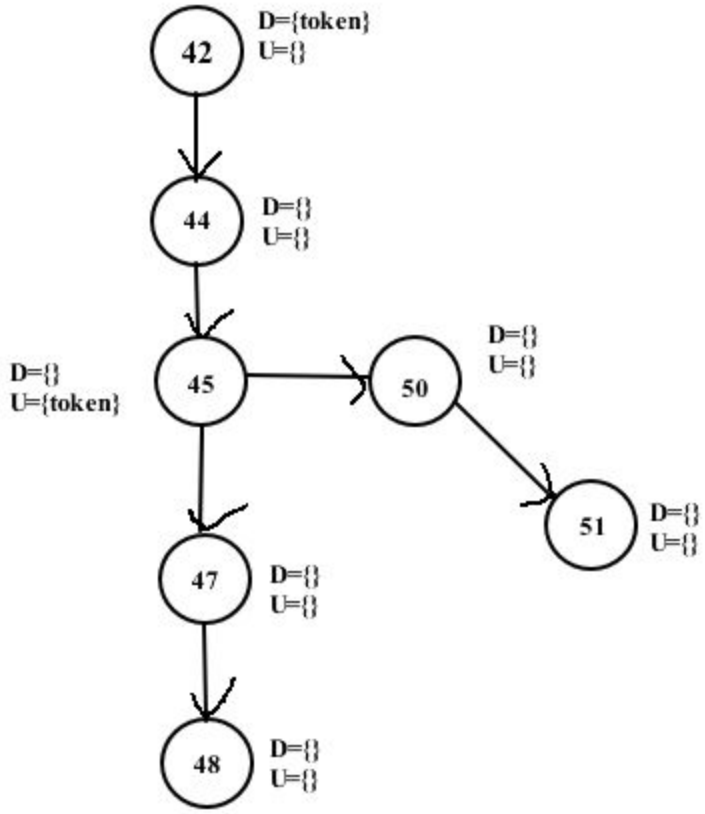
| def-> | 24 | 26 | 27 | 29 | 31 | 32 | 33 | 36 | 37 |
|---|---|---|---|---|---|---|---|---|---|
| modifier | | 27, 29, 31,36 | | | | | | | 27,29,31, 36 |
| count | 32, 33 | 32 | | | | 32, 33 | | | |
| ast | 26, 37 | | | | | | | | |

Find test cases that satisfy the all-uses criterion

| Test case | ast(TokenTypes) | DU covered | | |
|---|---|---|---|---|
| | | modifier | count | ast |
| 1 | {} | <26,27> | | <24,26> |
| 2 | {TokenType.PLUS} | <26,27><br><26,29><br><26,29><br><26,31> | <26,32> | <24,26> |
| 3 | {TokenType.PLUS, TokenType.XOR} | <26,27><br><26,27><br><26,29><br><26,31><br><26,36><br><37,27><br><37,29><br><37,31><br><37,36> | <24,32><br><24,33><br><26,32><br><32,32><br><32,33> | <24,26><br><24,37> |

boolean isOperand(int token)

```
42    public boolean isOperand(int token)
43    {
44        System.out.println("inside isOperand?");
45        if (token == TokenTypes.PLUS || token == TokenTypes.MINUS || token == TokenTypes.DIV || token == TokenTypes.STAR)
46        {
47            System.out.println("true");
48            return true;
49        }
50        System.out.println("false");
51        return false;
52    }
```




| variable | Token | |
|---|---|---|
| Line | C-use | P-use |
| 42 | x | |
| 44 | | |
| 45 | x | |
| 47 | | |
| 48 | | |
| 50 | | |
| 51 | | |

Def use

| def: | 42 | 44 | 45 | 47 | 48 | 50 | 51 |
|---|---|---|---|---|---|---|---|

<table>
<tr><td></td><td>
<table>
<tr><td>token</td><td>45</td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
<br>For this function, if the token is type of operands, it does not go below line 48, and if the token is not type of operands, it does not go to the line between 47 and 48.
</td></tr>
<tr><td></td><td></td></tr>
</table>

<table>
<tr><td colspan="2">Black box testing for Number of Variable Declaration Check (NumberOfVariableDeclarationCheck.cs)</td></tr>
<tr><td>Name of methods</td><td>Used testing method and explanation.</td></tr>
<tr><td>int[] getDefaultTokens()</td><td>Black box testing technique: Logic Function testing

```
@Test
public void getDefaultTokensTest()
{
    //Logic Function Testing
    NumberOfVariableDeclarationCheck myCheck = spy(new NumberOfVariableDeclarationCheck());
    int[] defaultTokens = myCheck.getDefaultTokens();
    assertTrue(defaultTokens.length == 1);  //length check
    assertTrue(defaultTokens[0] == TokenTypes.VARIABLE_DEF);
}
```

The returning value for this method an integer array that containing only one TokenTypes: VARIABLE_DEF. I checked if the length is equal to 1 and the value of defaultTokens[0] is exactly equal to TokenTypes.VARIABLE_DEF.<br><br>Here is the logic function I set up:<br>Let V as defaultTokens[0] == TokenTypes.VARIABLE_DEF<br>L as defaultTokens.length == 1;<br>Pass = LV<br><br>Below chart is the truth table
<table>
<tr><td>V</td><td>L</td><td>Pass</td></tr>
<tr><td>0</td><td>0</td><td>0 (fail)</td></tr>
<tr><td>0</td><td>1</td><td>0 (fail)</td></tr>
<tr><td>1</td><td>0</td><td>0 (fail)</td></tr>
<tr><td>1</td><td>1</td><td>1 (pass)</td></tr>
</table>
</td></tr>
<tr><td>getRequiredTokens()</td><td>Black box testing technique: Logic Function testing</td></tr>
</table>

<table>
<tr><td></td><td>Same as int[] getDefaultTokens(), since the return value is similar to getDefaultTokens() method.</td></tr>
<tr><td>VisitToken(DetailAST ast)</td><td>technique: Equivalence class partitioning test<br><br>

```
public void visitTokenTest()
{
    NumberOfVariableDeclarationCheck myCheck = spy(new NumberOfVariableDeclarationCheck());

    //creating input ast
    DetailAST astMock = PowerMockito.mock(DetailAST.class);

    //AST with 1 variable definition
    int[] myInputs = {};
    for(int i = 0; i< myInputs.length; i++)
    {
        DetailAST temp = new DetailAST();
        temp.setType(myInputs[i]);
        astMock.addChild(temp);
    }
    //The AST has all possible tokens
    myCheck.visitToken(astMock);
    assertTrue(myCheck.count == 0);

    //AST with 1 variable definition
    int[] myInputs2 = {TokenTypes.ASSIGN, TokenTypes.BOR, TokenTypes.LITERAL_THROWS, TokenTypes.VARIABLE_DEF};
    for(int i = 0; i< myInputs2.length; i++)
    {
        DetailAST temp = new DetailAST();
        temp.setType(myInputs2[i]);
        astMock.addChild(temp);
    }
    //The AST has all possible tokens
    myCheck.visitToken(astMock);
    assertTrue(myCheck.count == 1);


    //AST with 2 variable definition
    int[] myInputs3 = {TokenTypes.VARIABLE_DEF, TokenTypes.VARIABLE_DEF, TokenTypes.EOF};
    for(int i = 0; i< myInputs3.length; i++)
    {
        DetailAST temp = new DetailAST();
        temp.setType(myInputs3[i]);
        astMock.addChild(temp);
    }
    //The AST has all possible tokens
    myCheck.visitToken(astMock);
    assertTrue(myCheck.count == 2);
}
```

ECT1: AST: num of Variable_def = 0 : invalid<br>
ECT2: AST: num of Variable_def = 1 : valid<br>
ECT3: AST: num of Variable_def = 2 : valid<br><br>

<table>
<tr><td>ECT</td><td>AST</td><td>Valid | invalid</td></tr>
<tr><td>ECT1</td><td>{}</td><td>invalid</td></tr>
<tr><td>ECT2</td><td>{TokenTypes.ASSIGN,<br>TokenTypes.BOR,<br>TokenTypes.LITERAL_THROWS,<br>TokenTypes.VARIABLE_DEF};</td><td>valid</td></tr>
<tr><td>ECT3</td><td>{TokenTypes.VARIABLE_DEF,<br>TokenTypes.VARIABLE_DEF,<br>TokenTypes.EOF};</td><td>valid</td></tr>
</table>

There can be only three case of inputs. One is an input with empty AST, Other one is an input with more than one Variable_def tokens. The other one is an input with other types but no Variable_def token.</td></tr>
</table>

<table>
<tr>
<td>isVarDec(int token)</td>
<td>
Black box testing technique: Equivalence class partitioning test

```
@Test
public void isVarDecTest()
{
    NumberOfVariableDeclarationCheck myCheck = spy(new NumberOfVariableDeclarationCheck());
    assertTrue(myCheck.isVarDec(TokenTypes.VARIABLE_DEF));
    assertFalse(myCheck.isVarDec(TokenTypes.MINUS));
    assertFalse(myCheck.isVarDec(TokenTypes.STAR));
    assertFalse(myCheck.isVarDec(TokenTypes.DIV));
    assertFalse(myCheck.isVarDec(TokenTypes.ANNOTATION));
    assertFalse(myCheck.isVarDec(TokenTypes.COLON));
    assertFalse(myCheck.isVarDec(TokenTypes.TYPE_PARAMETERS));
    assertFalse(myCheck.isVarDec(TokenTypes.STAR_ASSIGN));
}
```

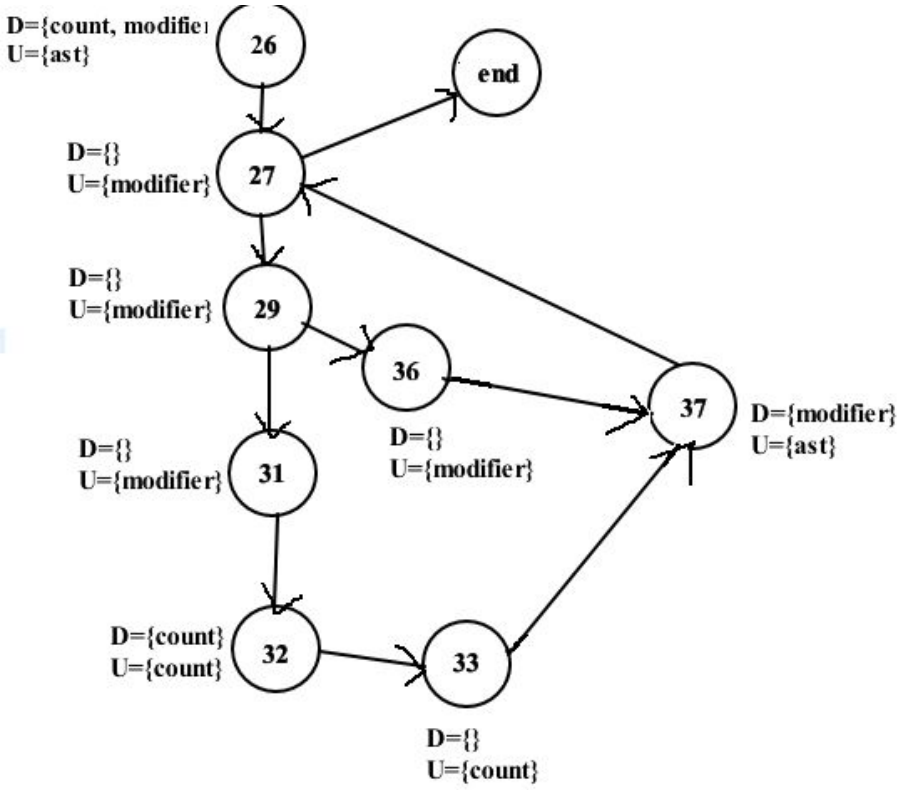
ECT1: token == TokenTypes.VARIABLE_DEF : valid<br>
ECT2: token != TokenTypes.VARIABLE_DEF : invalid

<table>
<tr><td>ECT</td><td>token</td><td>Valid | invalid</td></tr>
<tr><td>ECT1</td><td>VARIABLE_DEF</td><td>valid</td></tr>
<tr><td>ECT1</td><td>MINUS</td><td>invalid</td></tr>
<tr><td>ECT1</td><td>DIV</td><td>invalid</td></tr>
<tr><td>ECT1</td><td>STAR</td><td>invalid</td></tr>
<tr><td>ECT2</td><td>ANNOTATION</td><td>invalid</td></tr>
<tr><td>ECT2</td><td>COLON</td><td>invalid</td></tr>
<tr><td>ECT2</td><td>TYPE_PARAMETERS</td><td>invalid</td></tr>
<tr><td>ECT2</td><td>STAR_ASSIGN</td><td>invalid</td></tr>
</table>
</td>
</tr>
</table>

<table>
<tr><td colspan="2">White box testing for Number of Variable Declaration (NumberOfVariableDeclarationCheck.cs)</td></tr>
<tr><td>getDefaultTokens()</td><td>The function returns array of int with only one entry of TokenTypes.VARIABLE_DEF. It is very simple function and it has 100% coverage whenever the function is called, so there is no need to run complex white box testing.</td></tr>
<tr><td>getRequiredTokens()</td><td>Same as getDefaultTokens() function.</td></tr>
</table>

void visitToken(DetailAST ast)

```
@Override
public void visitToken(DetailAST ast)
{
    DetailAST modifier = ast.getFirstChild();
    while (modifier != null)
    {
        if (isVarDec(modifier.getType()))
        {
            System.out.println(String.format("{0} is declaration for variable", modifier.getType()));
            count++;
            System.out.println(count);
        }
        else
        { System.out.println(String.format("{0} is NOT declaration for variable", modifier.getType())); }
        modifier = ast.getNextSibling();
    }

}
```



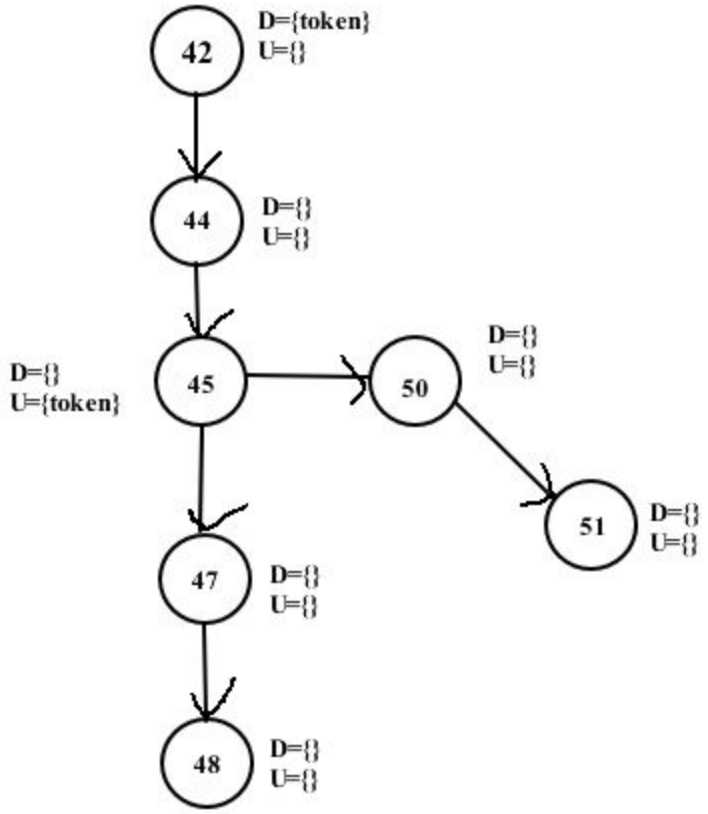

C-use/p-use chart for visitToken(DetailAST ast) function in NumberOfVariableDeclarationCheck.class.

| | modifier | | count | |
|---|---|---|---|---|
| Line | c-use | p-use | c-use | p-use |
| 26 | | x | x | |
| 27 | | x | | |
| 29 | | x | | |
| 31 | | x | | |
| 32 | | | x | |

<table>
<tr><td></td><td>
<table>
<tr><td>33</td><td></td><td></td><td></td><td>x</td></tr>
<tr><td>36</td><td></td><td>x</td><td></td><td></td></tr>
<tr><td>37</td><td></td><td>x</td><td></td><td></td></tr>
</table>
Variable modifier is used only for p-use. The value is changing to traverse. On the line 32, variable count is used for c-use. It increments the value of count.
</td></tr>
<tr><td>Def-use</td><td>
<table>
<tr><td>def-></td><td>24</td><td>26</td><td>27</td><td>29</td><td>31</td><td>32</td><td>33</td><td>36</td><td>37</td></tr>
<tr><td>modifier</td><td></td><td>27, 29, 31,36</td><td></td><td></td><td></td><td></td><td></td><td></td><td>27,29,31, 36</td></tr>
<tr><td>count</td><td>32, 33</td><td>32</td><td></td><td></td><td></td><td>32, 33</td><td></td><td></td><td></td></tr>
<tr><td>ast</td><td>26, 37</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>
</td></tr>
<tr><td>Find test cases that satisfy the all-uses criterion</td><td>
<table>
<tr><td rowspan="2">Test case</td><td rowspan="2">ast(TokenTypes)</td><td colspan="3">DU covered</td></tr>
<tr><td>modifier</td><td>count</td><td>ast</td></tr>
<tr><td>1</td><td>{}</td><td><26,27></td><td></td><td><24,26></td></tr>
<tr><td>2</td><td>{TokenType.VARIABLE_DEF}</td><td><26,27><br><26,29><br><26,29><br><26,31></td><td><26,32></td><td><24,26></td></tr>
<tr><td>3</td><td>{TokenType.VARIABLE_DEF, TokenType.PLUS}</td><td><26,27><br><26,27><br><26,29><br><26,31><br><26,36><br><37,27><br><37,29><br><37,31><br><37,36></td><td><24,32><br><24,33><br><26,32><br><32,32><br><32,33></td><td><24,26><br><24,37></td></tr>
</table>
</td></tr>
</table>

| | |
|---|---|
| boolean isVarDec(int token) | |

```
public boolean isVarDec(int token)
{
    System.out.println("isVarDec?");
    if (token == TokenTypes.VARIABLE_DEF)
    {
        System.out.println("true");
        return true;
    }
    System.out.println("false");
    return false;
}
```




| variable | Token | |
|---|---|---|
| Line | C-use | P-use |
| 42 | x | |
| 44 | | |
| 45 | x | |

| | | | |
|---|---|---|---|
| 47 | | | |
| 48 | | | |
| 50 | | | |
| 51 | | | |

Def use

| def: | 42 | 44 | 45 | 47 | 48 | 50 | 51 |
|---|---|---|---|---|---|---|---|
| token | 45 | | | | | | |

Compare to the tests from deliverable 1, since I could see my source code, I can reference it on the unit testing. While I was working on black-box testing, I did not reference the source code, to make high coverage. For instance, as I mentioned above, when I test a function getDefaultTokens() which returns an int[] collection of tokens, I should not compare the return value of it and a collection of integer array I made, since the order might be different.

From the white box testing, this was the first time I have been working on white box testing. During the process, I can determine the code flow and see any unnecessary codes. Since used algorithm is very simple, it was hard to make a logic mistake from coding. If the code is long and logic is very complicated, doing white box testing will help analyzing the code a lot.