White Box Testing

Mahtab EzzatiKarami

Bank.FeesCalculator

The focus of this class is to check the correct calculation of fees while performing withdrawal, deposit, and transfer.

Withdrawal

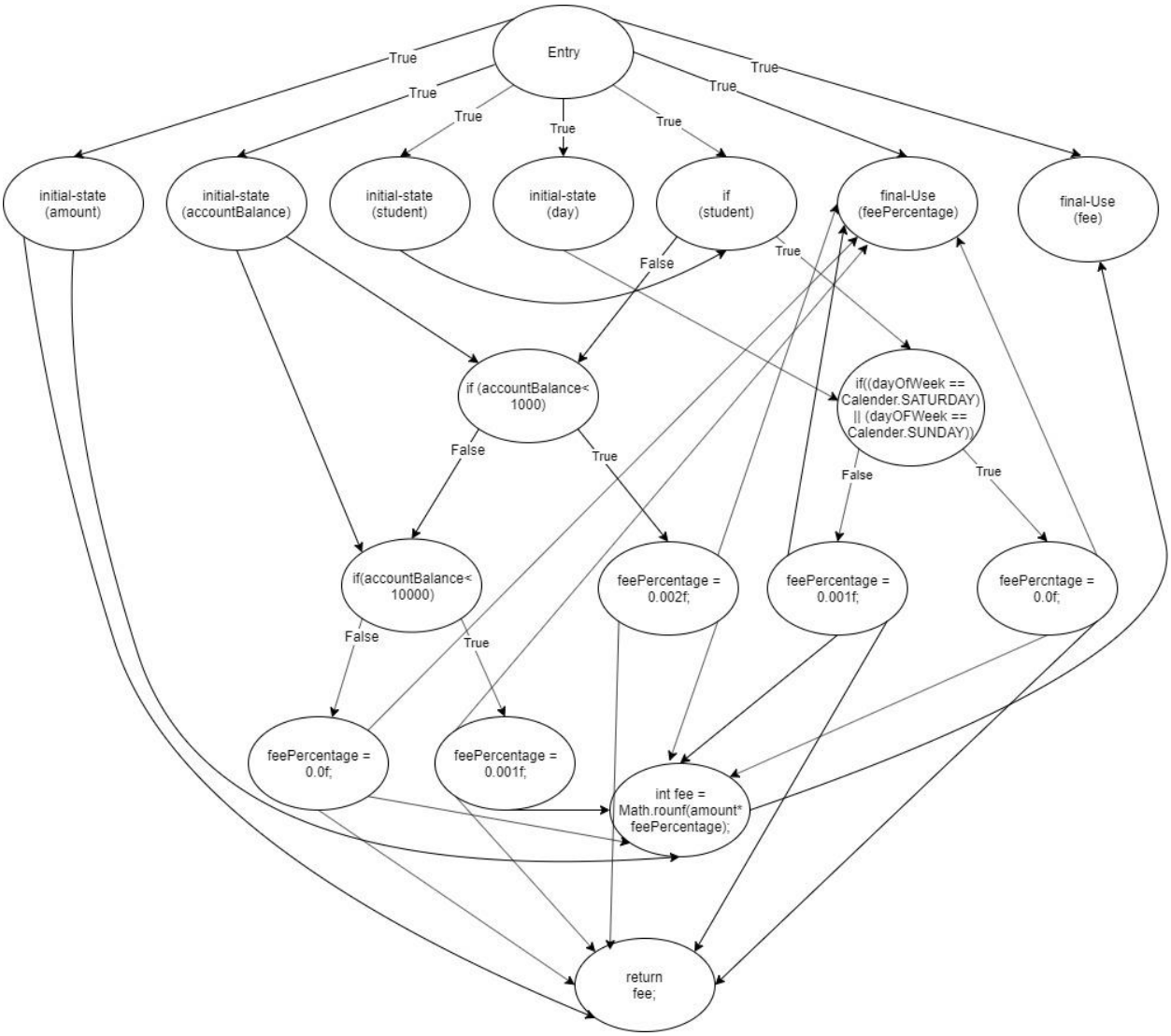
For withdrawal, we had to compute the slice based on the criterion *FinalUse(fee)* (i.e the statement *return(fee)*), and develop test cases that perform statement coverage on the resulting set of the statements in the computed slice. For our analysis, as we were told, we ignored the call to round function.
One way to compute a slice is by constructing a Program Dependence Graph (PDG) and then appropriately traverse this graph. As we know, Program Dependency Graphs are directed graphs. There are three types of vertices:

- Entry vertex
- Initial-state(x) for every variable x such that, there exists a path in the CFG on which x is used before it is defined. It represents as assignment to x from the initial state.
- Final-Use(x) for every variable x named in P's end statement. It represents an access to the final value of x computed by P.

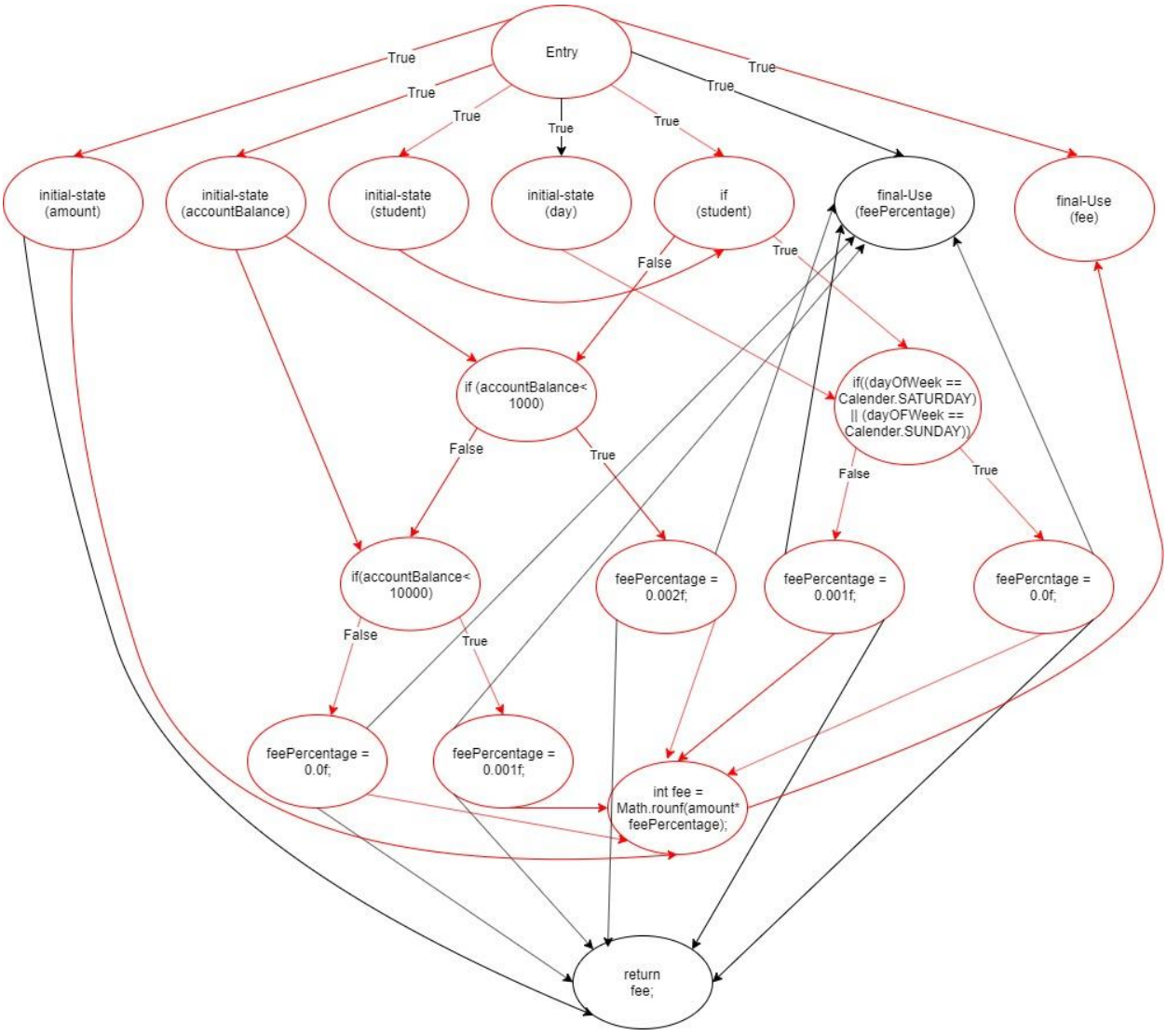
There are three types of edges:

- Control Dependence edges labeled as True or False. The source is either a predicate or the entry vertex. A control dependence means that during program execution, if the label of the edge matches the execution result of the source vertex predicate then, the target vertex will be eventually executed.
- Data Dependence between two nodes if one id defining a variable and the other is using the variable. Two types of data dependence edges:
    - Flow dependence
        - Loop carried
        - Loop independent
    - Def-order dependence

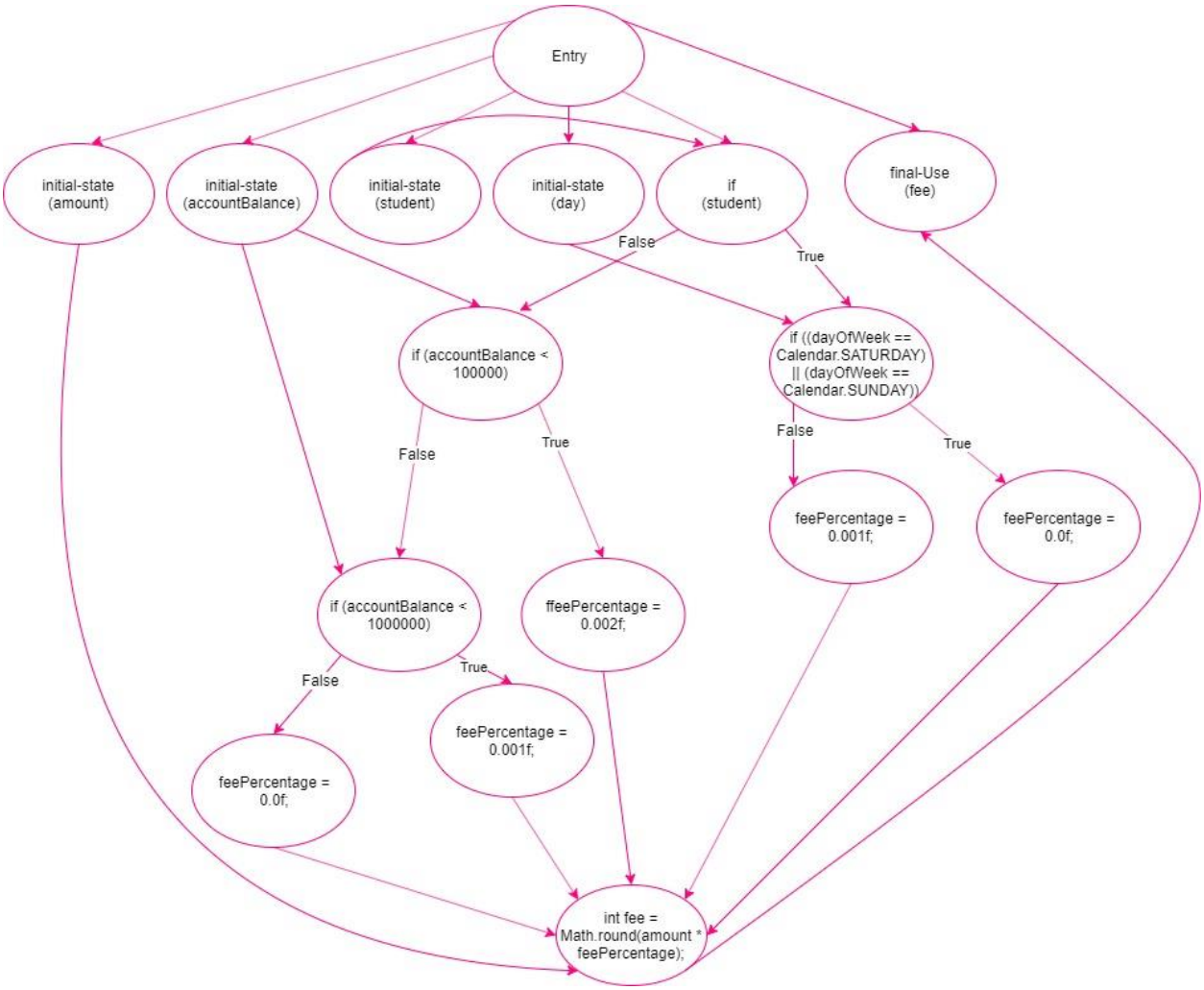
Program Dependency Graph for withdrawal is as below:


Entry
initial-state (amount)
initial-state (accountBalance)
initial-state (student)
initial-state (day)
if (student)
final-Use (feePercentage)
final-Use (fee)
if (accountBalance< 1000)
if((dayOfWeek == Calender.SATURDAY) || (dayOFWeek == Calender.SUNDAY))
if(accountBalance< 10000)
feePercentage = 0.002f;
feePercentage = 0.001f;
feePercntage = 0.0f;
feePercentage = 0.0f;
feePercentage = 0.001f;
int fee = Math.rounf(amount* feePercentage);
return fee;
True
False

Now we want to find the backward slice. We start from final-Use(fee) and trace it backward and each time when we go through a node or an edge we color it red.



The final slice:



These are the tests for statement coverage for the above slice:

Test Cases

The tests are as follows:

Test Case#	Amount	AccountBalance	Student	DayOfWeek
1	1000	1000	T	Saturday
2	1000	1000	T	Sunday
3	1000	1000	T	Wednesday
4	1000	900	F	Saturday
5	1000	9000	F	Saturday
6	1000	90000	F	Saturday

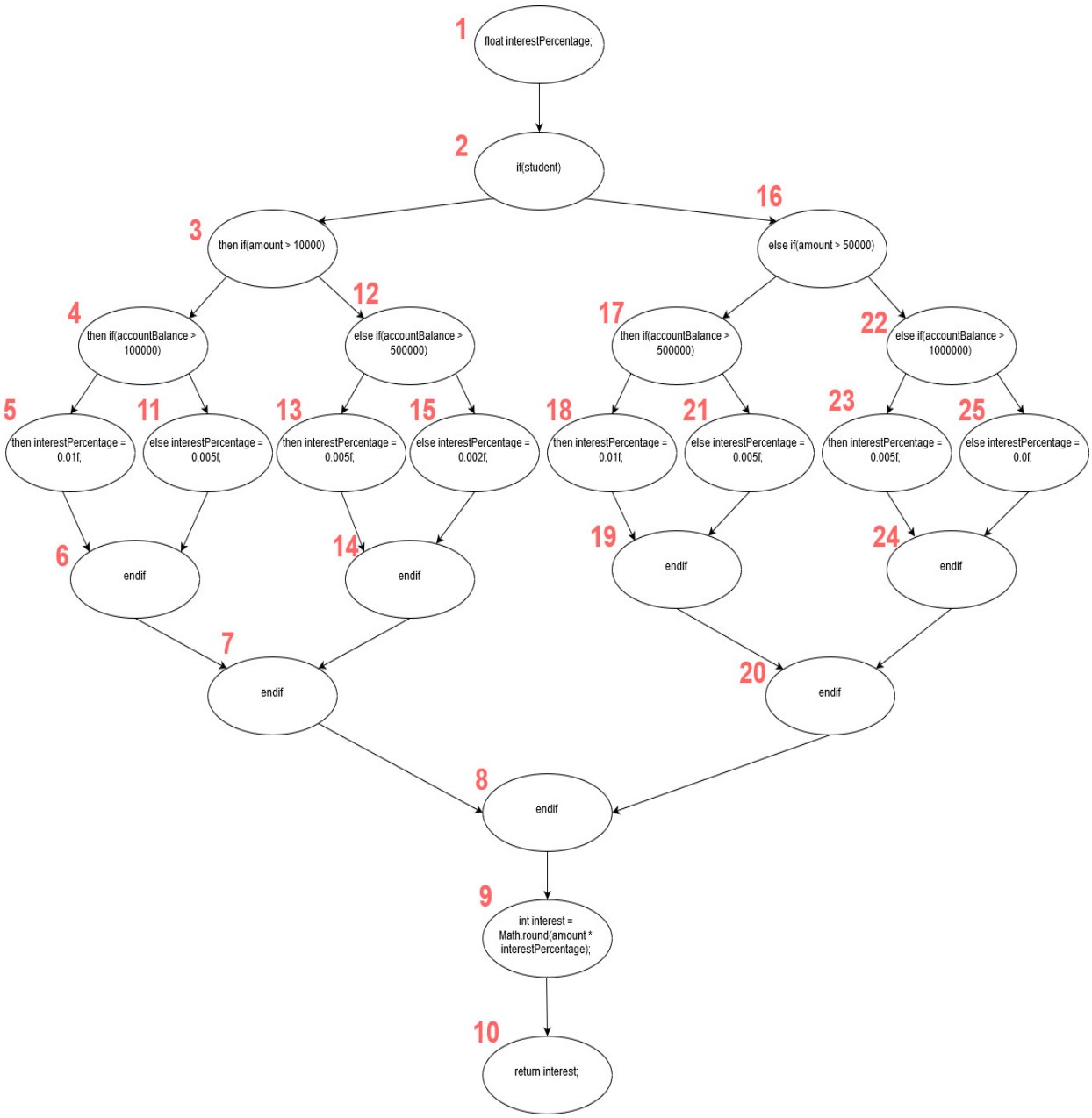
Results

The Junit test results are as follows:

Test Case#	Test Result
1	Passed
2	Passed
3	Passed
4	Passed
5	Failed
6	Failed

Deposit

The program graph of the calculateDepositInterest method is as follows:



In the program graph, nodes are taken to represent individual statements including conditional expressions, or fragments of statements such as the "endif" nodes.

The defining nodes of interestPercentage are nodes 5, 11, 13, 15, 18, 21, 23 and 25, and the usage node of interestPercentage is node 9. The DU-paths of interestPercentage are therefore:

Path number	Path nodes
1	1-2-3-4-5-6-7-8-9-10
2	1-2-3-4-11-6-7-8-9-10
3	1-2-3-12-13-14-7-8-9-10
4	1-2-3-12-15-14-7-8-9-10
5	1-2-16-17-18-19-20-8-9-10
6	1-2-16-17-21-19-20-8-9-10
7	1-2-16-22-23-24-20-8-9-10
8	1-2-16-22-25-24-20-8-9-10

As can be seen from the program graph, there is only one path which includes a given defining node of interestPercentage, and an execution which traverses this path will result in statement coverage for all statements in the path. Therefore, each path results in one test case. The domains of the input variables for each test case, which correspond to the DU-paths of the same number, are as follows:

Test case number	amount	accountBalance	student
1	>10000	>100000	true
2	>10000	<=100000	true
3	<=10000	>500000	true
4	<=10000	<=500000	true
5	>50000	>500000	false
6	>50000	<=500000	false
7	<=50000	>1000000	false
8	<=50000	<=1000000	false

The selected input values for each test case are as follows:

Test case number	amount	accountBalance	student
1	11000	110000	true
2	11000	90000	true
3	9000	510000	true
4	9000	490000	true
5	51000	510000	false
6	51000	490000	false
7	49000	1100000	false
8	49000	900000	false

The following are the test results for the FeesCalculatorDUPathTest.java class:

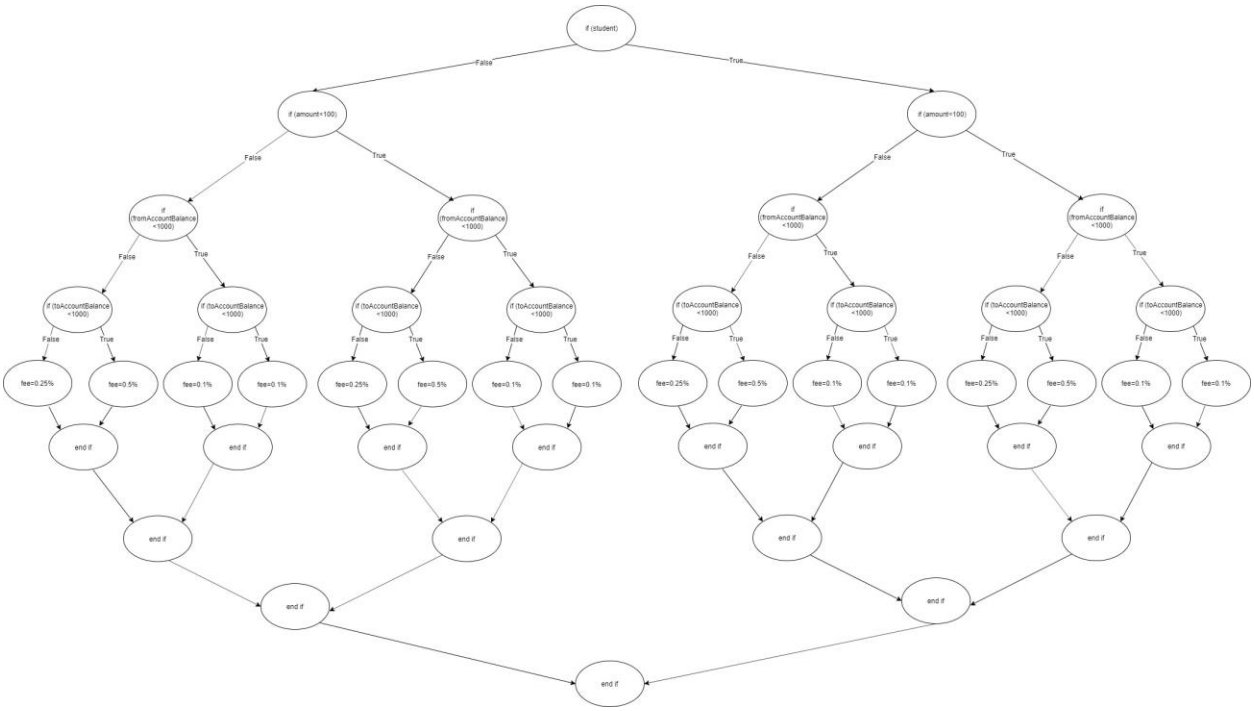
Test case number	Result
1	Passed
2	Passed
3	Passed
4	Failed
5	Passed
6	Passed
7	Passed
8	Passed

Test case 4 failed because the expected fee (per the program specification) is 0% but the fee implemented in the code is 2%.

Transfer

For this function, we are asked to compute and apply the test cases that result from basis path testing.

To perform basis path testing, first we are going to create CFG (Control Flow Graph) to determine different independent paths. Then we will calculate the Cyclomatic Complexity which is a metric to determine the number of independent logical routes. Then we will calculate a set of Basis set of paths. Afterwards, we create test cases to execute the program flow for each path.
Below, is the CDG for Transfer:
P.S. I know it is small but the quality of image is great. One way is to zoom in to check it out. I am also putting it in the main folder. So if it's easier, you can just check the image.

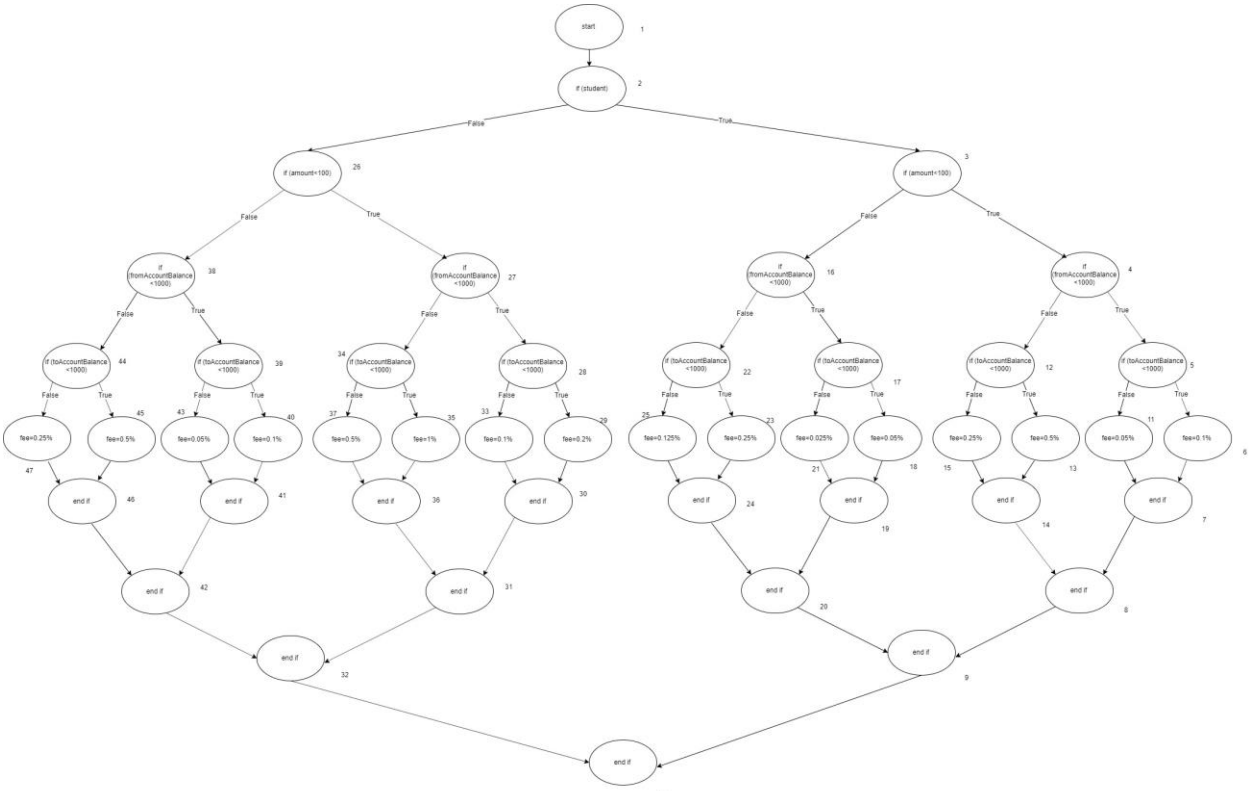


Now we want to calculate the Cyclomatic Complexity for this graph:

V(G) = E – N + 2 = 61 – 47 + 2 = 16

V(G) = P + 1 = 15 + 1 = 16

The image below, is the structure of CFG for Transfer. The numbers are for nodes in the graph.



There are 16 paths that we want to test:

1) 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
-checking first node of the node 5

2) 1, 2, 3, 4, 5,11,7, 8, 9, 10
-checking second node of the node 5

3) 1, 2, 3, 4, 12, 13, 14, 8, 9, 10
-checking first node of the node 12

4) 1, 2, 3, 4, 12, 15, 14, 8, 9, 10
-checking second node of the node 12

5) 1, 2, 3, 16, 17, 18, 19, 20, 9, 10
-checking first node of the node 17

6) 1, 2, 3, 16, 17, 21, 19, 20, 9, 10
-checking second node of the node 17

7) 1, 2, 3, 16, 22, 23, 24, 20, 9, 10

-checking first node of the node 22

8) 1, 2, 3, 16, 22, 23, 25, 20, 9, 10

-checking first node of the node 22

9) 1-2-26-27-28-29-30-31-32-10

-checking first node of the node 28

10) 1-2-26-27-28-33-30-31-32-10

-checking second node of the node 28

11) 1-2-26-27-34-35-36-31-32-10

-checking first node of the node 34

12) 1-2-26-27-34-37-36-31-32-10

-checking second node of the node 34

13) 1-2-26-38-39-40-41-42-32-10

-checking first node of the node 39

14) 1-2-26-38-39-43-41-42-32-10

-checking second node of the node 39

15) 1-2-26-38-44-45-48-42-32-10

-checking first node of the node 44

16) 1-2-26-38-44-47-48-42-32-10

-checking second node of the node 44

Test cases

The tests are as follows:

Test Case#	amount	fromAccountBalance	toAccountBalance	Student
1	90	900	900	T
2	90	900	1000	T
3	90	1000	900	T
4	90	1000	1000	T
5	100	900	900	T
6	100	900	1000	T
7	100	1000	900	T
8	100	1000	1000	T
9	90	900	900	F
10	90	900	1000	F
11	90	1000	900	F
12	90	1000	1000	F
13	100	900	900	F
14	100	900	1000	F
15	100	1000	900	F
16	100	1000	1000	F

Results

The Junit test results are as follows:

Test Case#	Test Result
1	Passed
2	Passed
3	Passed
4	Passed
5	Passed
6	Passed
7	Passed
8	Passed
9	Passed
10	Passed
11	Passed
12	Passed
13	Passed
14	Passed
15	Passed
16	Failed