CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

All Paths Coverage (AP)

- The strongest form of white-box testing based on “program structure”
- If you achieve AP, then you have also achieved SC and BC
- All the paths, from entry to exit of a block of code (usually a method; the test item), are executed during testing
- Developing all paths tests is complex and time consuming
- In practice it is seldom used (only considered for critical software)
- **However, as the strongest form of testing based on program-structure it is of theoretical importance, and is an important element in the body of knowledge for software testing**

Control Flow Graphs (CFGs)

- In statement and branch testing:
  - Control Flow Graphs (CFGs) are seldom used in practice
  - Tests can be more efficiently developed to supplement black-box tests
  - Using automated tools to measure the coverage
- However, in all paths testing, CFGs are essential
- Today we will see how to develop and use them

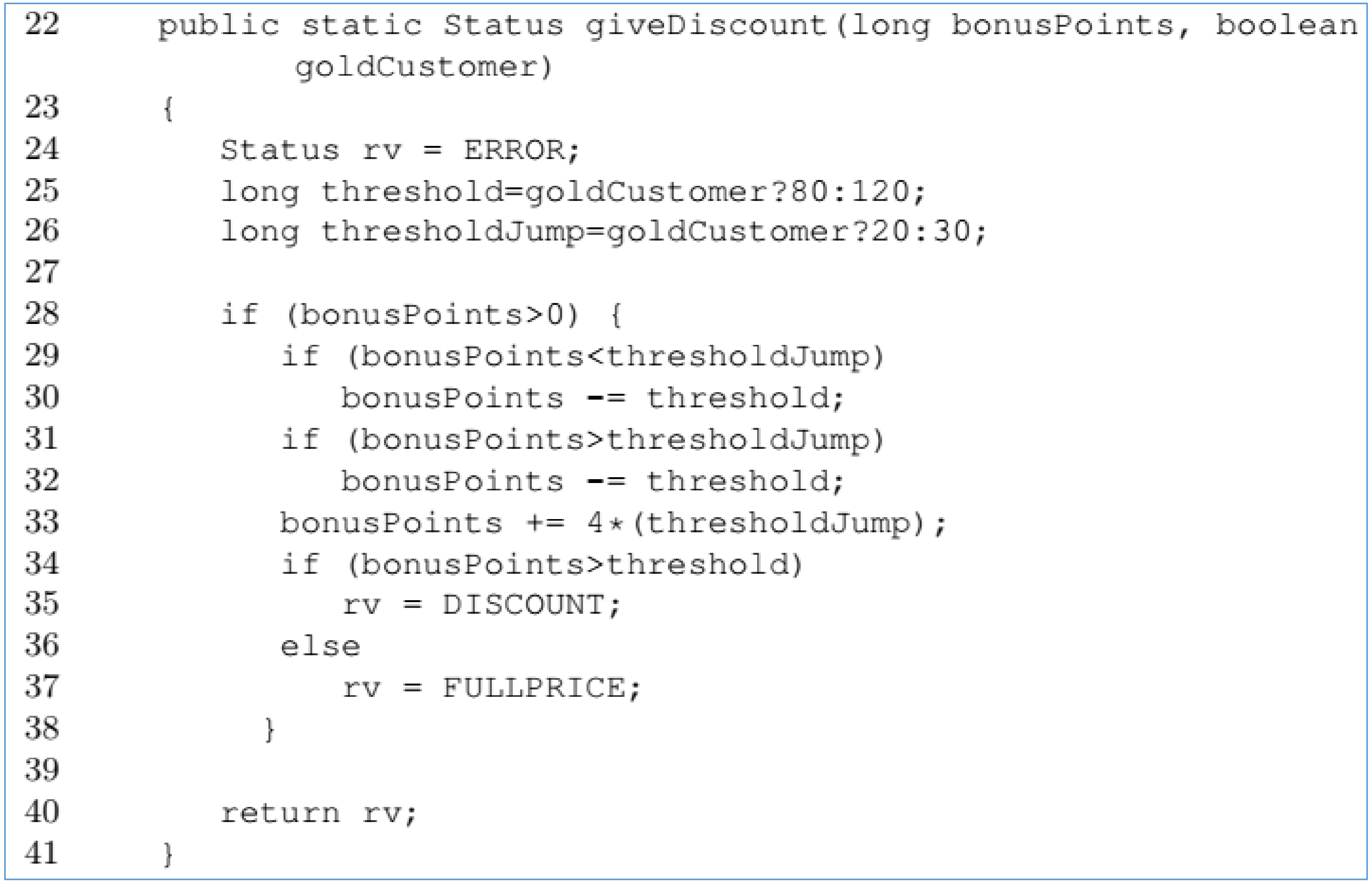
Testing with All Paths Coverage

- A path is a sequence of statements executed in a single run of a block of code
- All paths coverage refers to execution of all the paths that start on the first statement and end on the last statement of the block of code (“end-to-end” paths)
- The paths are usually identified at the source code level
- To identify these paths, a graph (CFG) is produced and provides a simplified representation of the code
- From this all the end-to-end paths are identified, and test data developed to cause these paths to be executed
- Identifying the paths directly from the source code takes experience, and is usually only achievable for smaller lengths of code

Definition

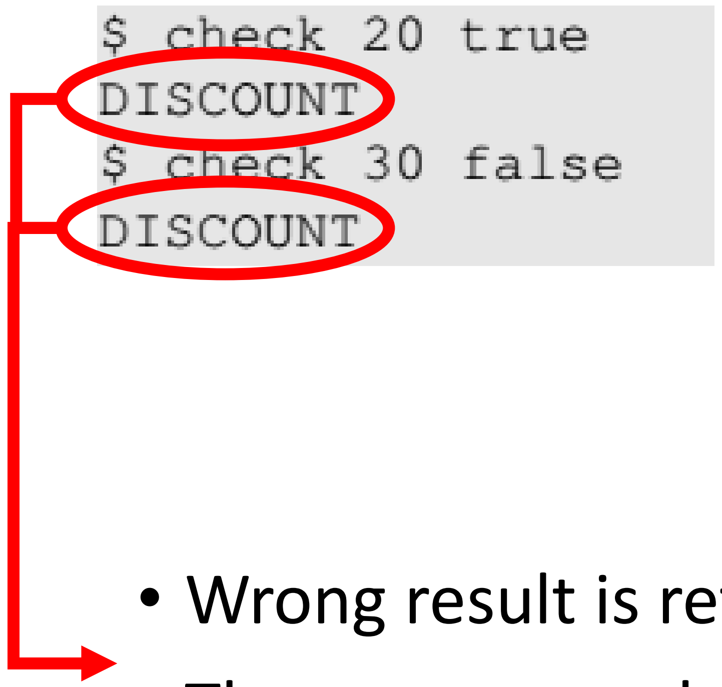
- Definition: an end-to-end path is a single flow of execution from the start to the end of a section of code
- This path may include multiple executions of any loops on the path

Example: giveDiscount() with Fault 6



```
public static Status giveDiscount(long bonusPoints, boolean
       goldCustomer)
{
   Status rv = ERROR;
   long threshold=goldCustomer?80:120;
   long thresholdJump=goldCustomer?20:30;

   if (bonusPoints>0) {
      if (bonusPoints<thresholdJump)
         bonusPoints -= threshold;
      if (bonusPoints>thresholdJump)
         bonusPoints -= threshold;
      bonusPoints += 4*(thresholdJump);
      if (bonusPoints>threshold)
         rv = DISCOUNT;
      else
         rv = FULLPRICE;
     }

   return rv;
}
```

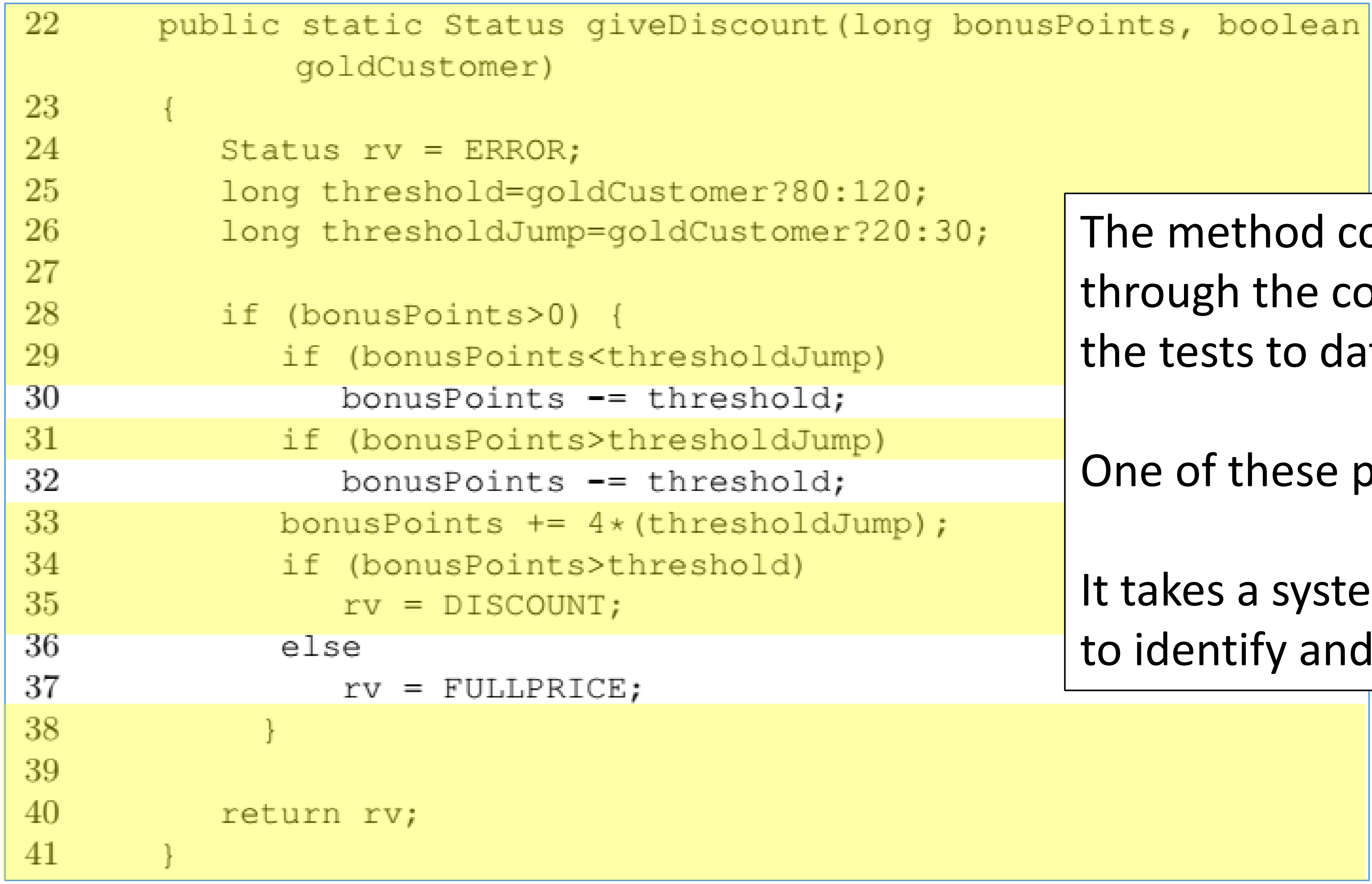
Demonstrating Fault 6



```
$ check 20 true
DISCOUNT
$ check 30 false
DISCOUNT
```

- Wrong result is returned for both the inputs (20,true) and (30,false)
- The correct result is **FULLPRICE** in both cases

Example: The Faulty Path



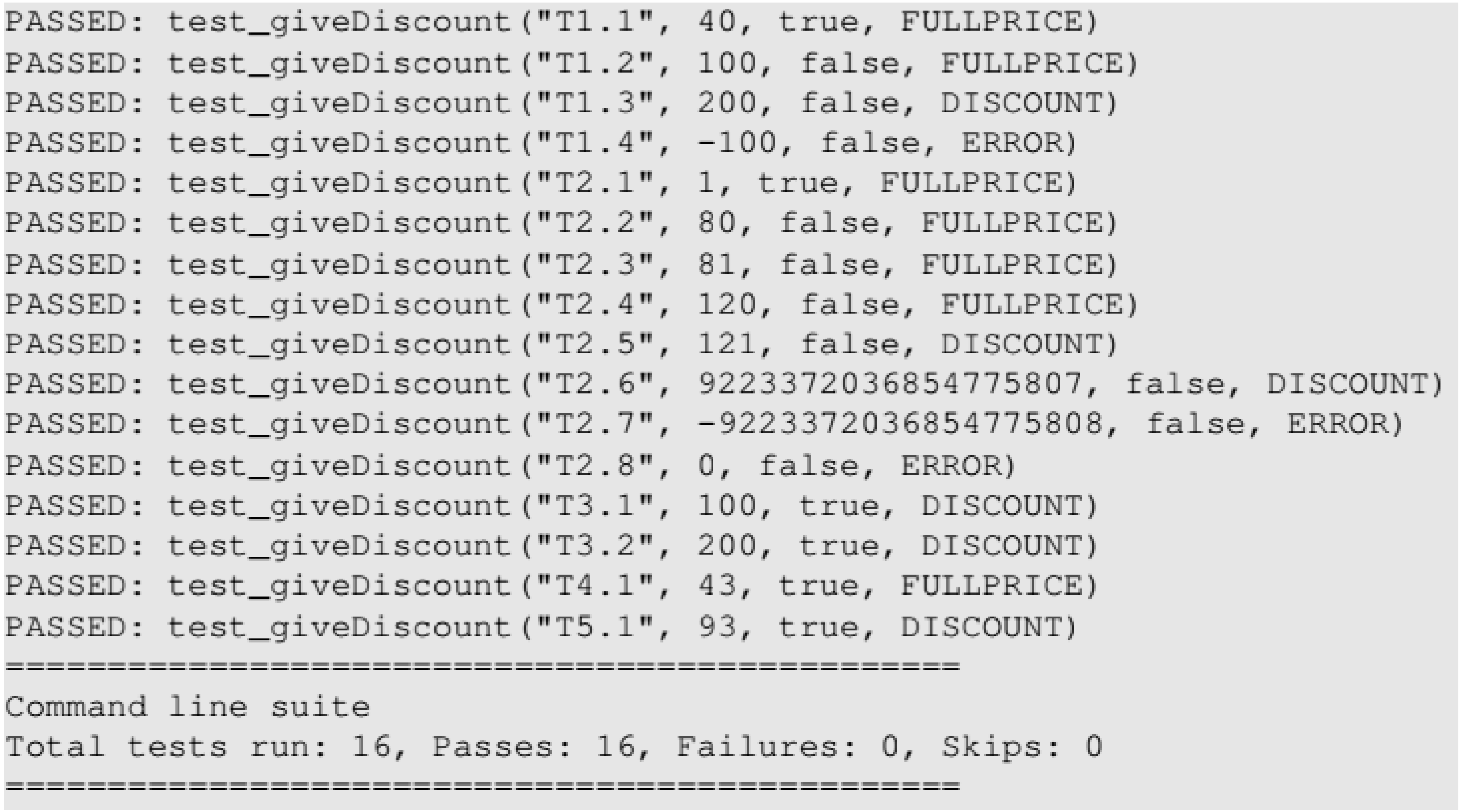
```
public static Status giveDiscount(long bonusPoints, boolean
        goldCustomer)
{
    Status rv = ERROR;
    long threshold=goldCustomer?80:120;
    long thresholdJump=goldCustomer?20:30;

    if (bonusPoints>0) {
        if (bonusPoints<thresholdJump)
            bonusPoints -= threshold;
        if (bonusPoints>thresholdJump)
            bonusPoints -= threshold;
        bonusPoints += 4*(thresholdJump);
        if (bonusPoints>threshold)
            rv = DISCOUNT;
        else
            rv = FULLPRICE;
    }

    return rv;
}
```

The method contains paths through the code not taken by the tests to date

One of these paths is faulty

It takes a systematic approach to identify and test every path

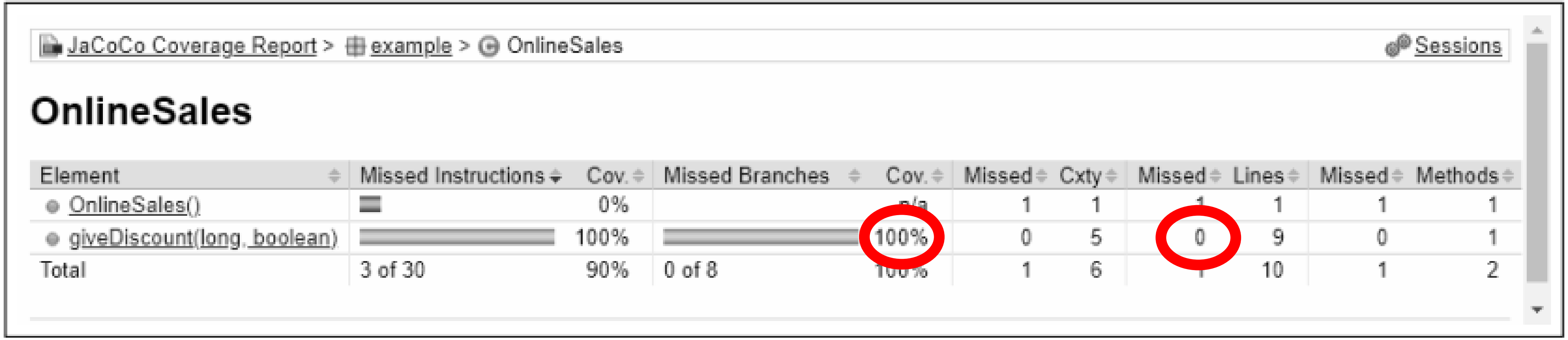
EP+BVA+DT+SC+BC Testing Against Fault 6



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
PASSED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
PASSED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
===============================================
Command line suite
Total tests run: 16, Passes: 16, Failures: 0, Skips: 0
===============================================
```

Branch Coverage of Fault 6

- Full statement and branch coverage achieved



JaCoCo Coverage Report > example > OnlineSales   Sessions

OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
OnlineSales()		0%		n/a	1	1	1	1	1	1
giveDiscount(long, boolean)		100%		100%	0	5	0	9	0	1
Total	3 of 30	90%	0 of 8	100%	1	6	1	10	1	2

- Tests 4.1 and 5.1 are in fact redundant for this version of the code
- The code has been changed, and these tests are no longer required to achieve statement or branch coverage

Code Analysis

- Unlike for statement and branch testing, no tools exist to identify path coverage in anything but the simplest of examples
- We will develop an abstract representation of the program in the form of a graph (CFG)
- And use that to identify the end-to-end paths

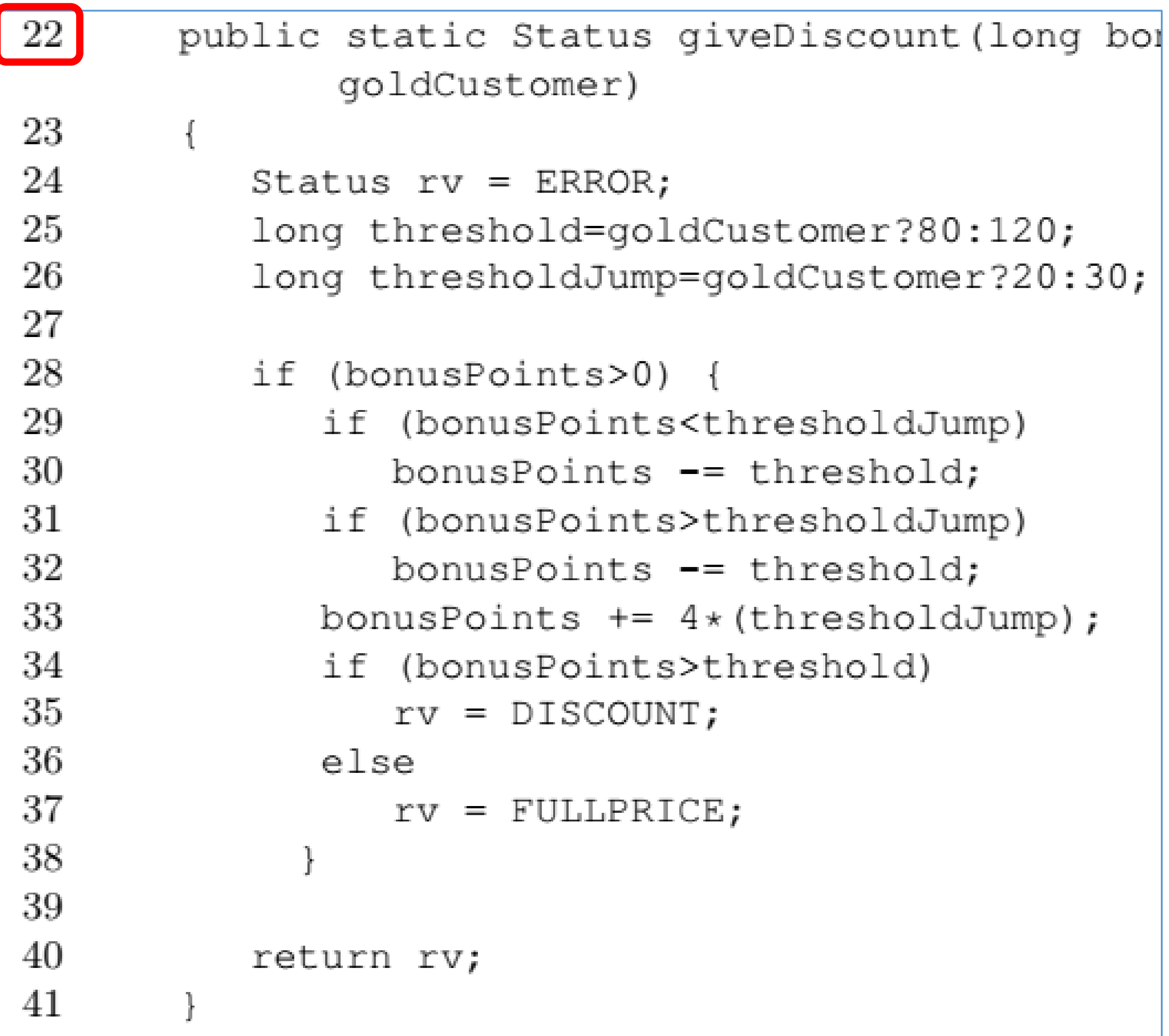
CFGs

- A control flow graph (CFG) is a directed graph showing the flow of control through a segment of code
- These are mainly used to represent code at the source code level
  - Each **node** in the graph represents one or more indivisible statements
  - Each **edge** represents a jump or branch in the flow of control
- A node with two exits represents a block of code terminating in a decision, with true and false exits (such as an if statement)
- The CFG is language independent – a simplified model of the code
- This allows sequential blocks of code, branches, and paths, to be easily identified

Development of the CFG

- CFG developed systematically, starting at the start of the code, and working down until all the code has been incorporated into the CFG
- Convention: where possible, to maintain consistency, true branches are placed on the right and false branches on the left

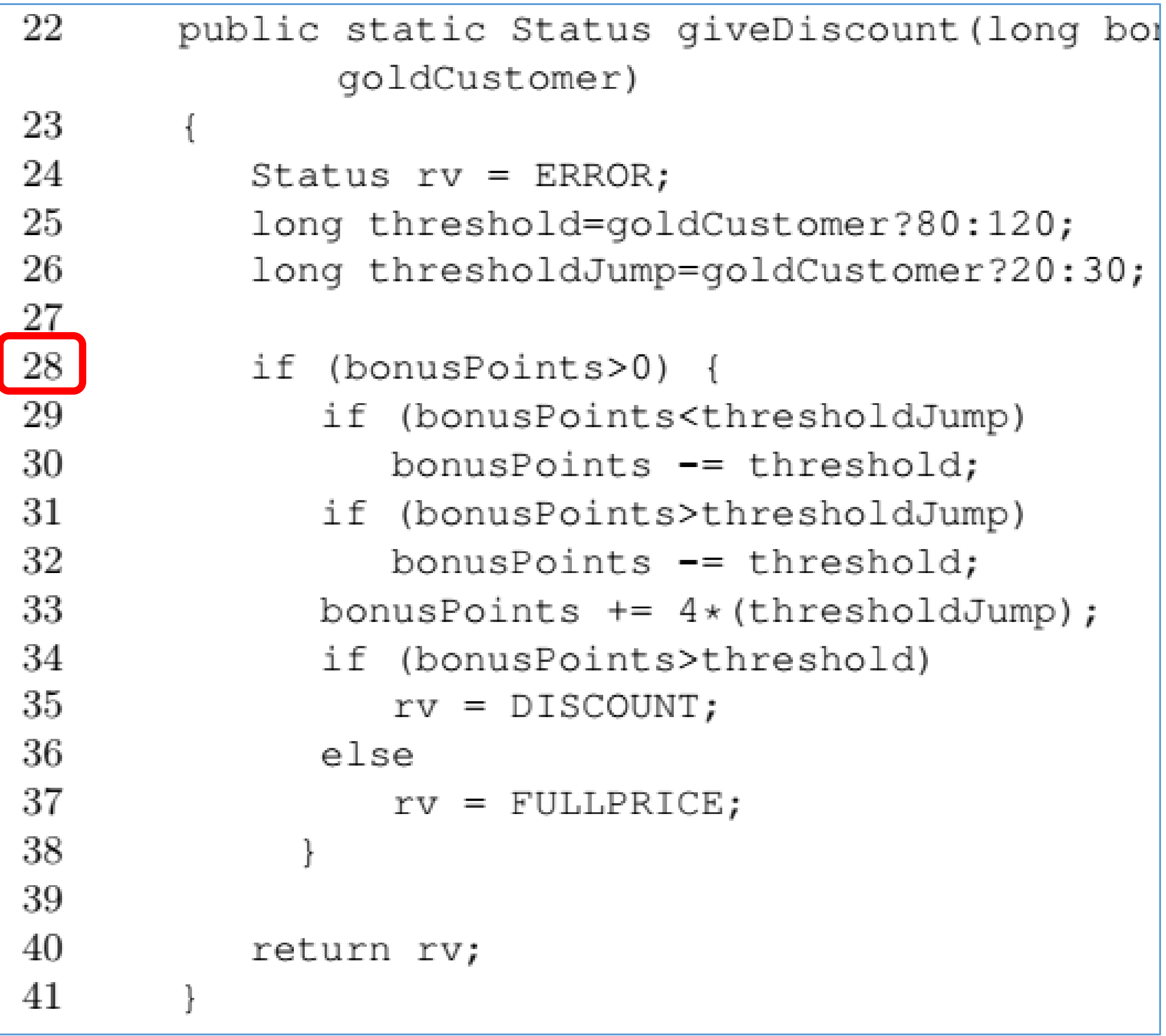
CFG – Stage 0



```
22    public static Status giveDiscount(long bo
             goldCustomer)
23    {
24       Status rv = ERROR;
25       long threshold=goldCustomer?80:120;
26       long thresholdJump=goldCustomer?20:30;
27
28       if (bonusPoints>0) {
29          if (bonusPoints<thresholdJump)
30             bonusPoints -= threshold;
31          if (bonusPoints>thresholdJump)
32             bonusPoints -= threshold;
33          bonusPoints += 4*(thresholdJump);
34          if (bonusPoints>threshold)
35             rv = DISCOUNT;
36          else
37             rv = FULLPRICE;
38       }
39
40       return rv;
41    }
```

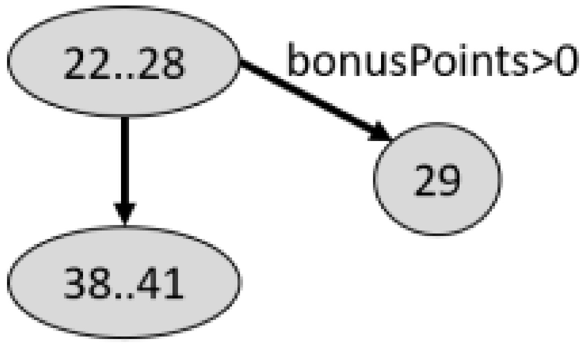
- Start with the entry point (line 22)
- From here, the code always executes to line 28
- Lines 22 to 28 form a block of indivisible statements
- This is represented by a single node, labelled (22..28)
- It is recommended that you label the nodes with the line numbers they contain – much more informative than just numbering the nodes 1, 2, 3, · · · , etc. as you will see in many books
- Each node represents an indivisible block of code (no branches)

CFG – Stage 1

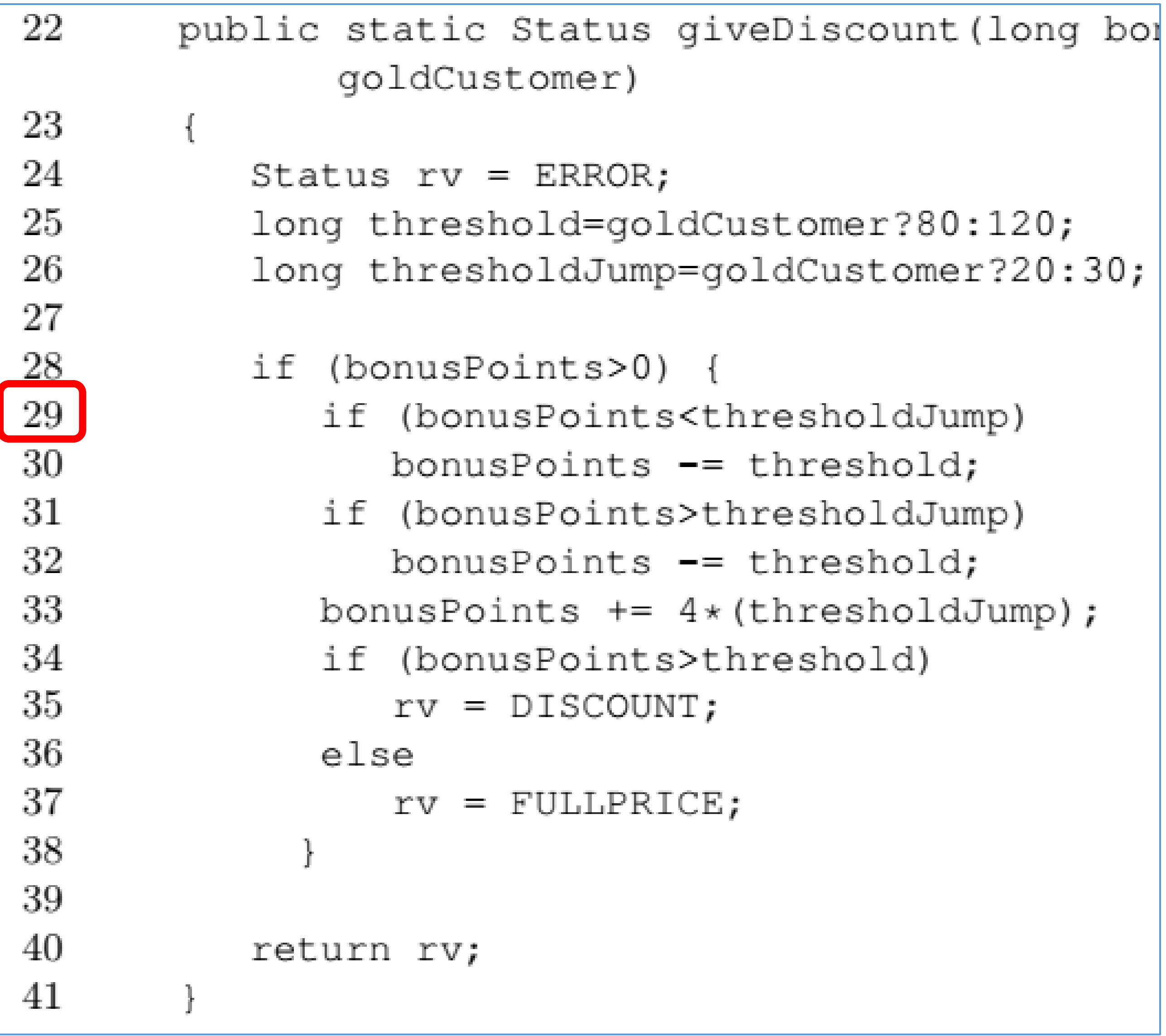


```
22      public static Status giveDiscount(long bo
                goldCustomer)
23      {
24          Status rv = ERROR;
25          long threshold=goldCustomer?80:120;
26          long thresholdJump=goldCustomer?20:30;
27
28          if (bonusPoints>0) {
29              if (bonusPoints<thresholdJump)
30                  bonusPoints -= threshold;
31              if (bonusPoints>thresholdJump)
32                  bonusPoints -= threshold;
33              bonusPoints += 4*(thresholdJump);
34              if (bonusPoints>threshold)
35                  rv = DISCOUNT;
36              else
37                  rv = FULLPRICE;
38          }
39
40          return rv;
41      }
```

- On line 28, if the decision (bonusPoints>0) is true, then control branches to line 29
- If not, control branches to line 38
- There are no jumps between lines 38 and 41, so this node is labelled 38..41
- The expression bonusPoints>0 shows the expression that must be true to take the right-hand branch

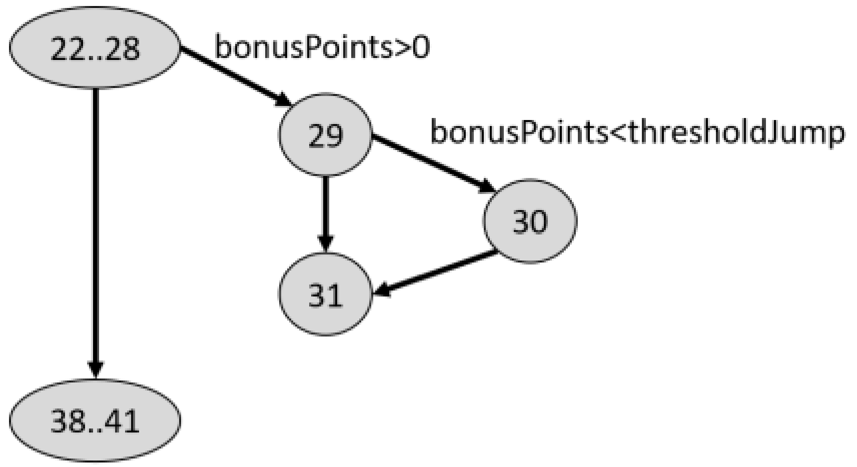


CFG – Stage 2

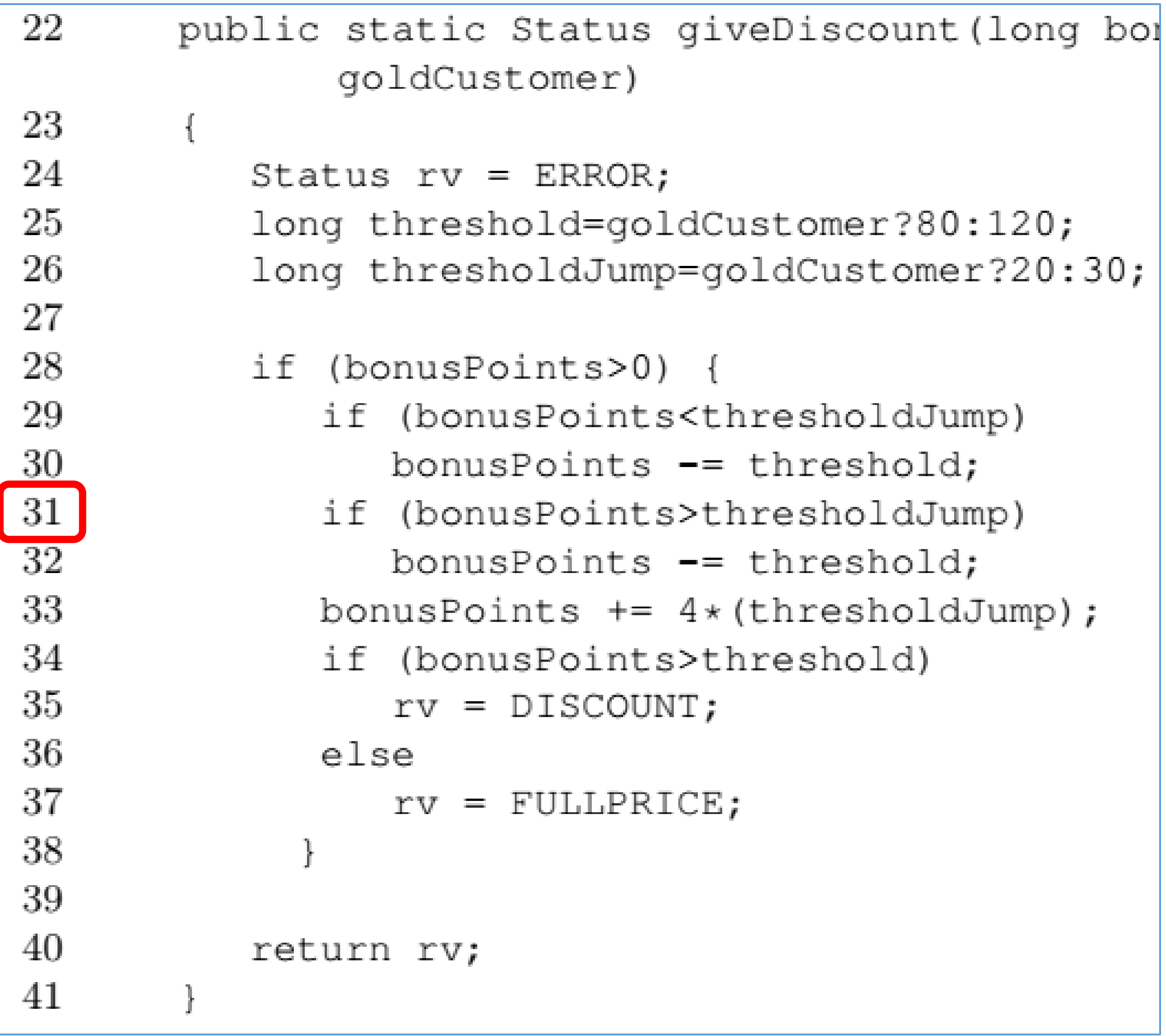


```
22      public static Status giveDiscount(long bo
                goldCustomer)
23      {
24          Status rv = ERROR;
25          long threshold=goldCustomer?80:120;
26          long thresholdJump=goldCustomer?20:30;
27
28          if (bonusPoints>0) {
29              if (bonusPoints<thresholdJump)
30                  bonusPoints -= threshold;
31              if (bonusPoints>thresholdJump)
32                  bonusPoints -= threshold;
33              bonusPoints += 4*(thresholdJump);
34              if (bonusPoints>threshold)
35                  rv = DISCOUNT;
36              else
37                  rv = FULLPRICE;
38          }
39
40          return rv;
41      }
```

- On line 29, if the decision (bonusPoints<thresholdJump) is true, then control branches to line 30
- Otherwise, control branches to line 31
- After line 30, control always goes to line 31



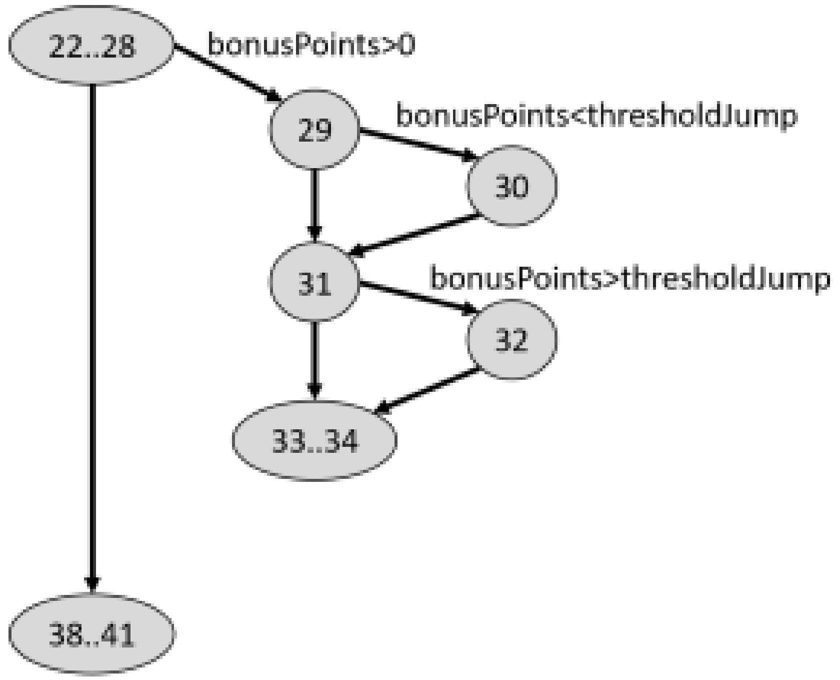
CFG – Stage 3



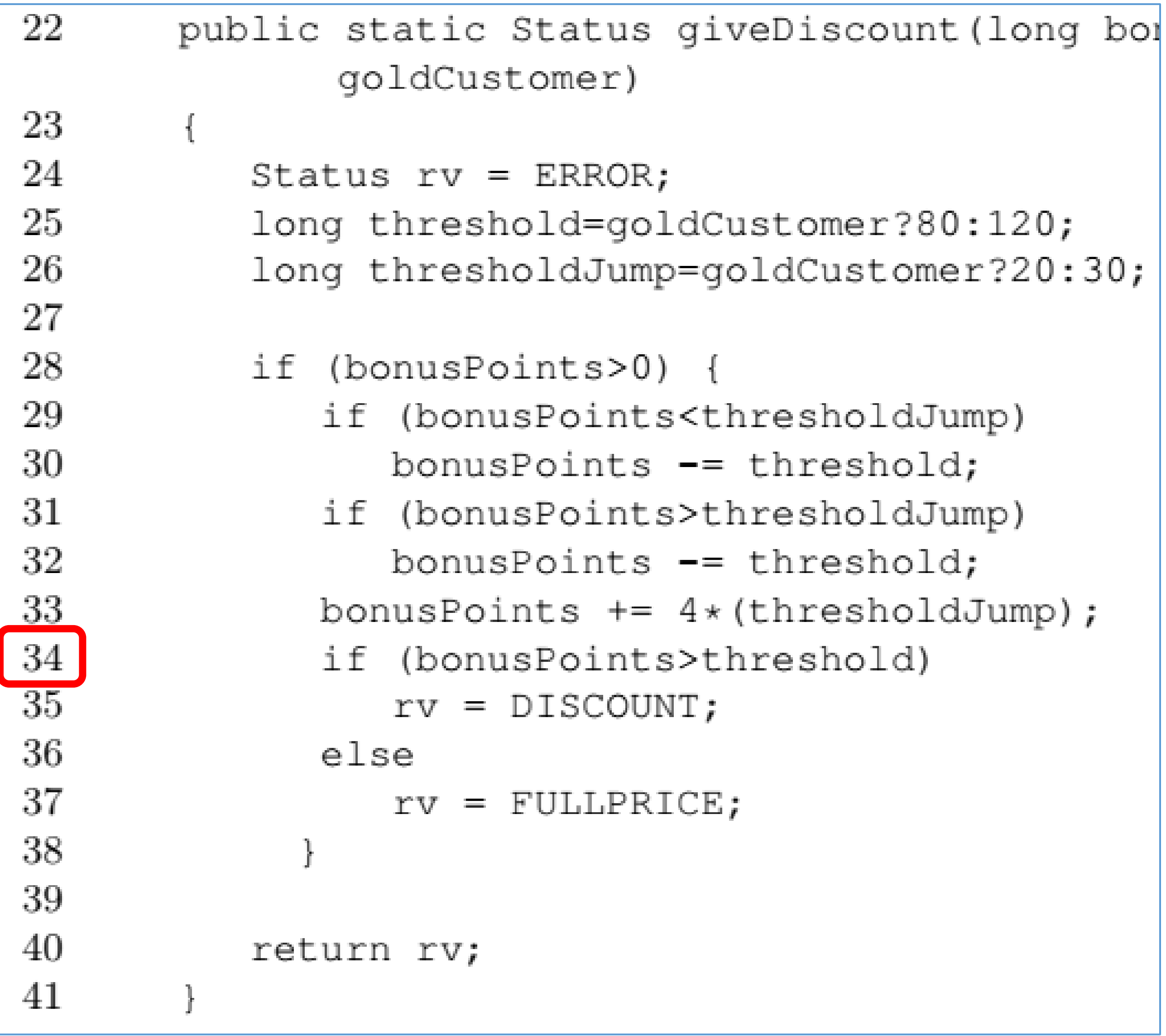
```
22      public static Status giveDiscount(long bo
                 goldCustomer)
23      {
24          Status rv = ERROR;
25          long threshold=goldCustomer?80:120;
26          long thresholdJump=goldCustomer?20:30;
27
28          if (bonusPoints>0) {
29              if (bonusPoints<thresholdJump)
30                  bonusPoints -= threshold;
31              if (bonusPoints>thresholdJump)
32                  bonusPoints -= threshold;
33              bonusPoints += 4*(thresholdJump);
34              if (bonusPoints>threshold)
35                  rv = DISCOUNT;
36              else
37                  rv = FULLPRICE;
38          }
39
40          return rv;
41      }
```

- Line 31, if the decision (bonusPoints>thresholdJump) is true, control branches to line 32
- Otherwise control branches to line 33 (first line in node 33..34)
- After line 32, control always goes to line 33

$$\begin{tikzcd}
22..28 \arrow[dr, "bonusPoints>0"] \arrow[dddd] & & \\
& 29 \arrow[r, "bonusPoints<thresholdJump"] \arrow[d] & 30 \arrow[dl] \\
& 31 \arrow[r, "bonusPoints>thresholdJump"] \arrow[d] & 32 \arrow[dl] \\
& 33..34 & \\
38..41 & &
\end{tikzcd}$$

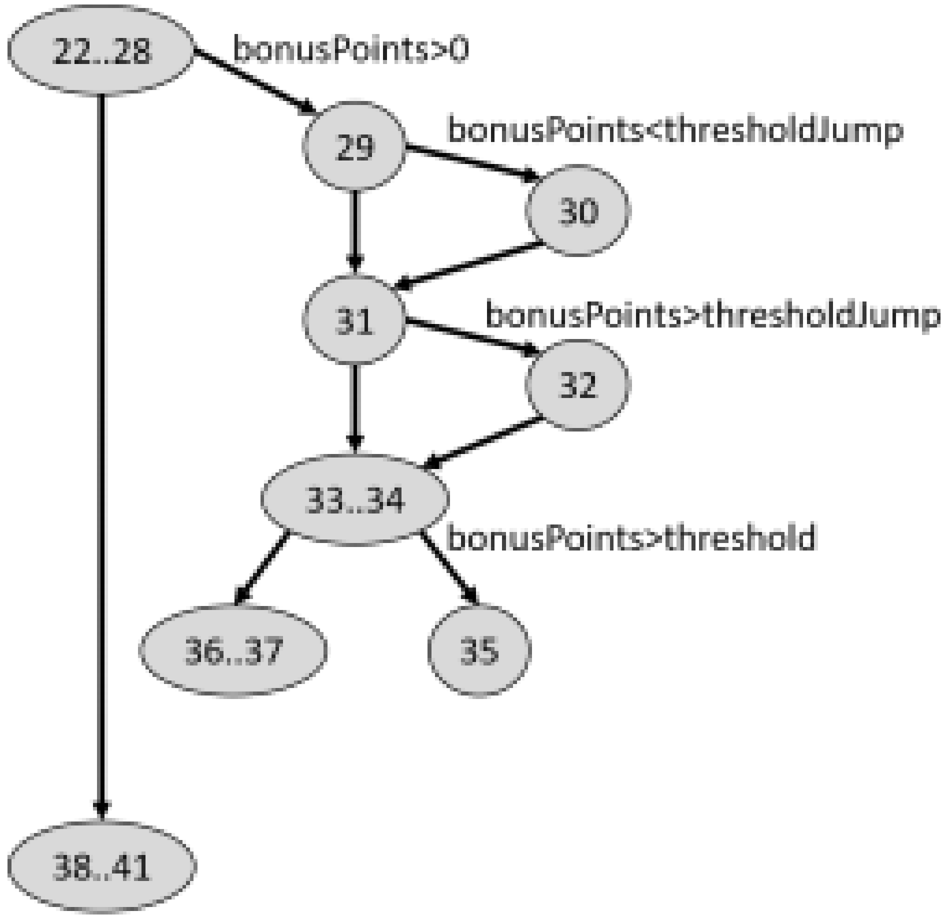


CFG – Stage 4



```
22      public static Status giveDiscount(long bo
                goldCustomer)
23      {
24          Status rv = ERROR;
25          long threshold=goldCustomer?80:120;
26          long thresholdJump=goldCustomer?20:30;
27
28          if (bonusPoints>0) {
29              if (bonusPoints<thresholdJump)
30                  bonusPoints -= threshold;
31              if (bonusPoints>thresholdJump)
32                  bonusPoints -= threshold;
33              bonusPoints += 4*(thresholdJump);
34              if (bonusPoints>threshold)
35                  rv = DISCOUNT;
36              else
37                  rv = FULLPRICE;
38          }
39
40          return rv;
41      }
```

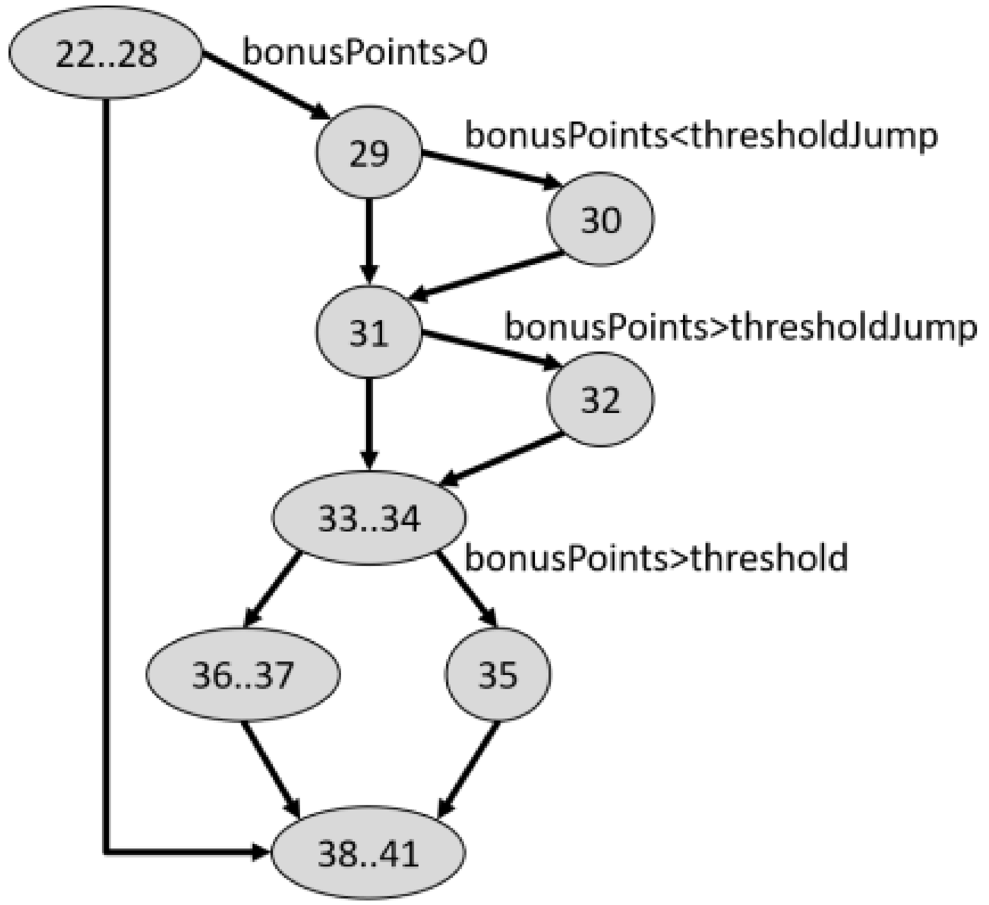
- On line 34, if the decision (bonusPoints>threshold) is true, then control branches to line 35
- Otherwise, control branches to line 36



CFG – Stage 5

```
22      public static Status giveDiscount(long bo
                goldCustomer)
23      {
24          Status rv = ERROR;
25          long threshold=goldCustomer?80:120;
26          long thresholdJump=goldCustomer?20:30;
27
28          if (bonusPoints>0) {
29              if (bonusPoints<thresholdJump)
30                  bonusPoints -= threshold;
31              if (bonusPoints>thresholdJump)
32                  bonusPoints -= threshold;
33              bonusPoints += 4*(thresholdJump);
34              if (bonusPoints>threshold)
35                  rv = DISCOUNT;
36              else
37                  rv = FULLPRICE;
38          }
39
40          return rv;
41      }
```

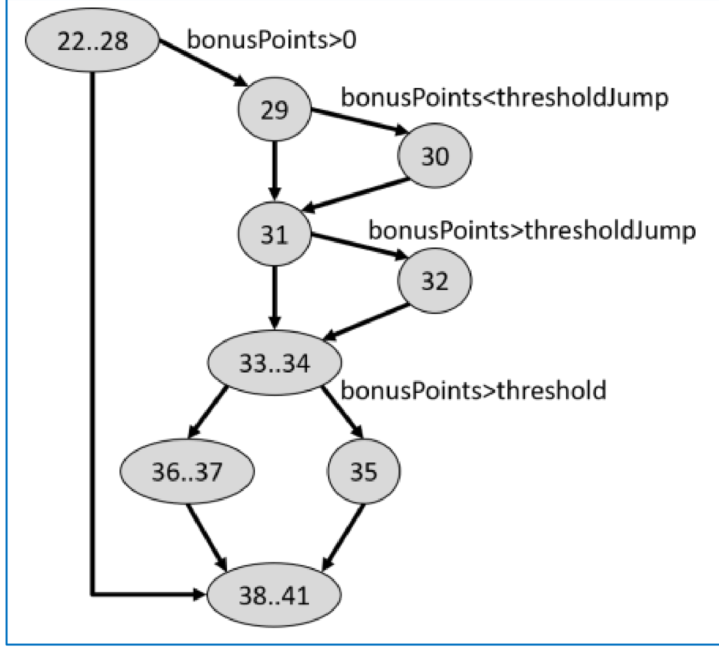
- After line 37, control always passes to line 38
- This is also the case for line 35
- This completes the CFG



Identifying the Candidate Paths

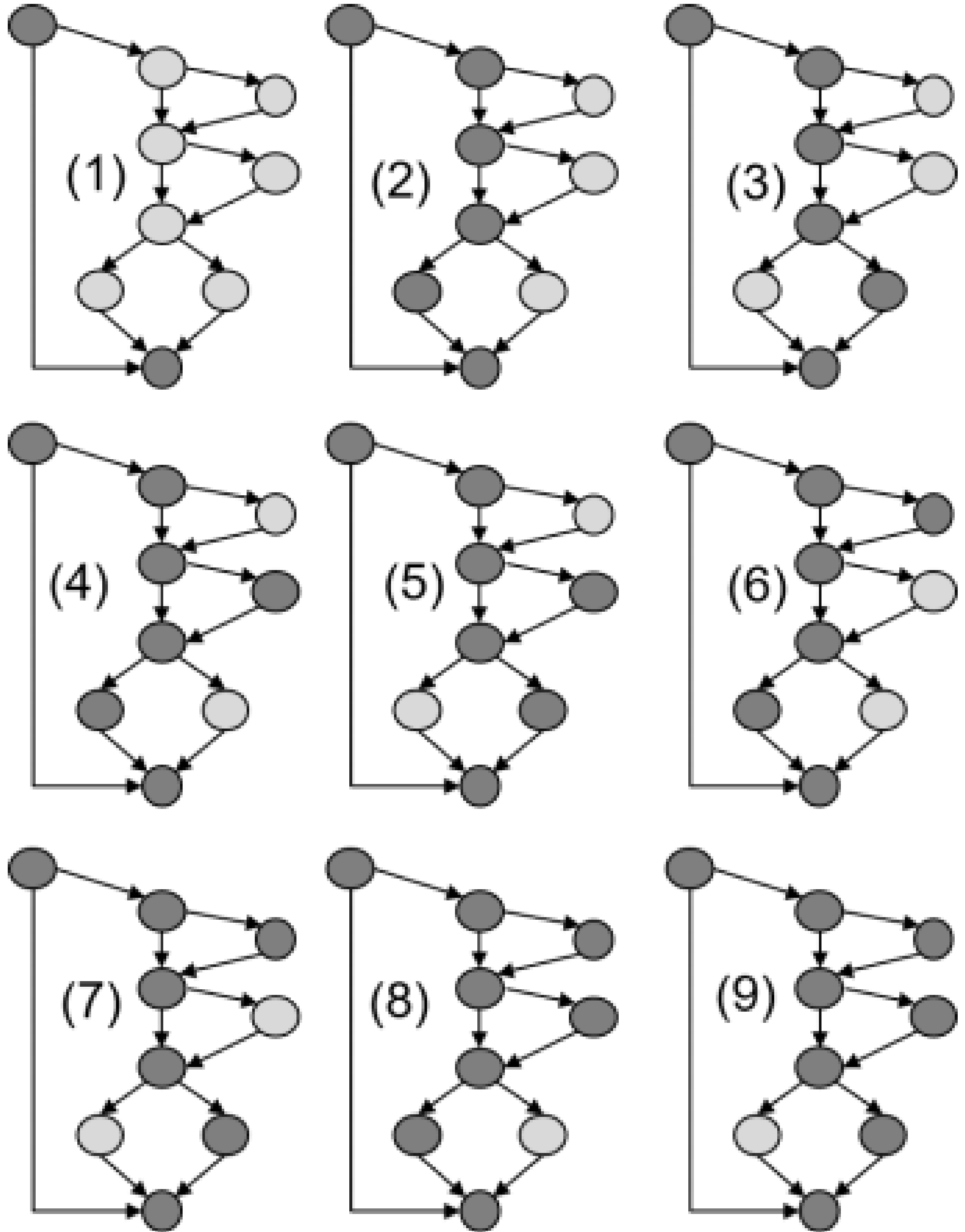
- By tracing all the start-to-end paths through the CFG, the candidate paths can be identified
- These are candidate paths: not of them are logically possible to take
- A systematic approach must be taken to avoid missing paths
- Work through the CFG starting at the top, and taking the left hand branches first

Candidate Paths



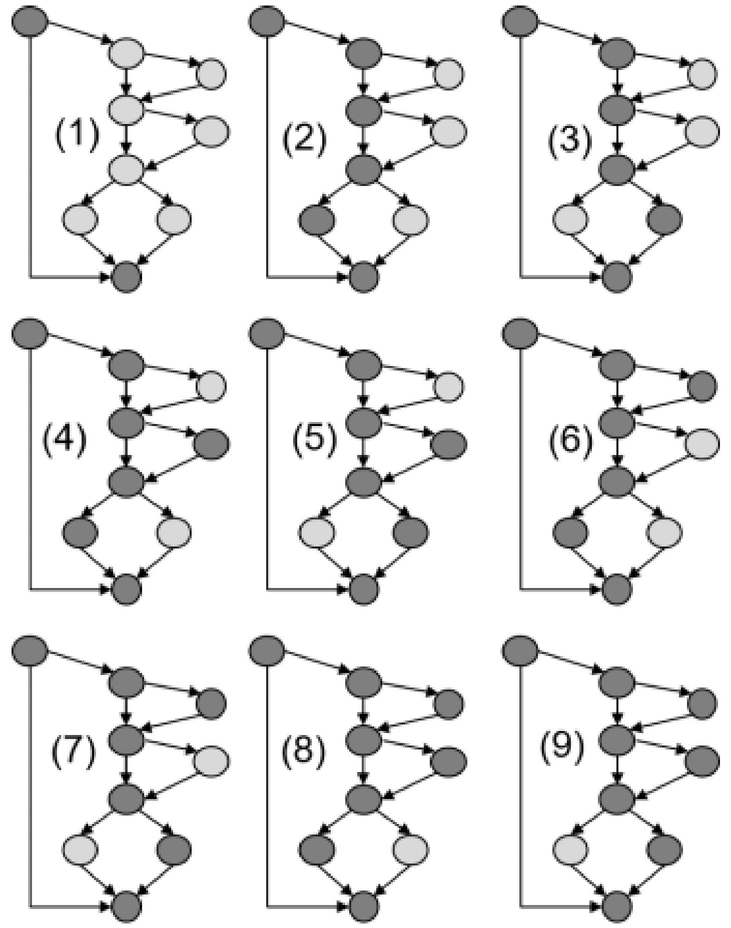
- Path 1, nodes (22..28)–(38..41)
- Path 2, nodes (22..28)–(29)–(31)–(33..34)–(36..37)–(38..41)
- Path 3, nodes (22..28)–(29)–(31)–(33..34)–(35)–(38..41)
- Path 4, nodes (22..28)–(29)–(31)–(32)–(33..34)–(36..37)–(38..41)
- Path 5, nodes (22..28)–(29)–(31)–(32)–(33..34)–(35)–(38..41)
- Path 6, nodes (22..28)–(29)–(30)–(31)–33..34)–(36..37)–(38..41)
- Path 7, nodes (22..28)–(29)–(30)–(31)–(33..34)–(35)–(38..41)
- Path 8, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(36..37)–(38..41)
- Path 9, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(35)–(38..41)

Candidate Paths



Identifying the Possible Paths

- Not all the paths are logically possible
- The possible paths are identified by mentally executing each path in the final CFG, following the code at the same time, and working out whether there is any possible input data that follows the path, starting using existing test data
- Shown in the following slides (and section 7.4.5 of the module textbook)



Identifying the Candidate Paths

By tracing all the start-to-end paths through the CFG, the candidate paths can be identified. These are candidate paths: not of them are logically possible to take. A systematic approach must be taken to avoid missing paths. Working through the CFG starting at the top, and taking the left hand branches first, identifies the following paths:

1. Path 1, nodes (22..28)–(38..41)
2. Path 2, nodes (22..28)–(29)–(31)–(33..34)–(36..37)–(38..41)
3. Path 3, nodes (22..28)–(29)–(31)–(33..34)–(35)–(38..41)
4. Path 4, nodes (22..28)–(29)–(31)–(32)–(33..34)–(36..37)–(38..41)
5. Path 5, nodes (22..28)–(29)–(31)–(32)–(33..34)–(35)–(38..41)
6. Path 6, nodes (22..28)–(29)–(30)–(31)–33..34)–(36..37)–(38..41)
7. Path 7, nodes (22..28)–(29)–(30)–(31)–(33..34)–(35)–(38..41)
8. Path 8, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(36..37)–(38..41)
9. Path 9, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(35)–(38..41)

DRAFT

7.4.5 The Possible Paths in Fault 6

The possible paths are identified by mentally executing each path in the final CFG (Figure 7.6), following the code in parallel to understand the processing, and identifying the input data constraints at each decision point required to follow the selected path.

If conflicting constraints are found, then that path is impossible. Otherwise, input data can be selected that matches the constraints. If possible, data values, and combinations of values, from previous tests are used (refer to Listing 3.1), so that duplicate tests can be easily identified and eliminated.

- **Path 1**

  Nodes: (22..28)→(38..41): note these must be in the correct sequence

  - Branch (22..29)→(38..41): requires (bonusPoints>0) to be false

  Therefore, bonusPoints must be zero or negative.

  Test T1.4 meets this condition with the input values (-100, false)

- **Path 2**

  Nodes: (22..28)→(29)→(31)→(33..34)→(36..37)→(38..41)

  - Branch (22..29)→(29): requires (bonusPoints>0)
  - Branch (29)→(31): requires (bonusPoints<thresholdJump) to be false

  - Branch (31)→(33..34): requires (bonusPoints>thresholdJump) to be false
  - Branch (33..34)→(36..37): requires (bonusPoints>threshold) to be false

  The input bonusPoints must be greater than or equal to zero on entry. At node (29) it must be greater than or equal to thresholdJump (20 or 30, depending on goldCustomer).

  At node (31) it must be less than or equal to threholdJump. This implies that bonusPoints must be equal to threholdJump.

  On node (33) the value is increased by 4*thresholdJump, that is by 80 or 120.

  So if the inputs are goldCustomer is true and bonusPoints equals 20, then on node (34) bonusPoints is modified to 100, which is greater than the threshold (80).

  And if the inputs are goldCustomer is false and bonusPoints equals 30, then on node (34) bonusPoints is modified to 150, which is greater than the threshold (120).

  In both cases, the decision at node (34) will be true. But the required value for this path is false. Thus, the path is impossible.

- **Path 3** Nodes: (22..28)→(29)→(31)→(33..34)→(35)→(38..41)
  - Branch (22..29)→(29): requires (bonusPoints>0)
  - Branch (29)→(31): requires (bonusPoints<thresholdJump) to be false
  - Branch (31)→(33..34): requires (bonusPoints>thresholdJump) to be false
  - Branch (33..34)→(35): requires (bonusPoints>threshold)

  The input bonusPoints must be greater than or equal to zero on entry. At node (29) it must be greater than or equal to thresholdJump (20 or 30, depending on goldCustomer).

  At node (31) it must be less than or equal to threholdJump. This implies that bonusPoints must be equal to threholdJump.

  On node (33) the value is increased by 4*thresholdJump, that is by 80 or 120.

  So if the inputs are goldCustomer is true and bonusPoints equals 20, then on node (34) bonusPoints is modified to 100, which is greater than the threshold (80).

  And if the inputs are goldCustomer is false and bonusPoints equals 30, then on node (34) bonusPoints is modified to 150, which is greater than the threshold (120).

  In both cases, the decision at node (34) will be true as required. None of the existsing tests have either 20 or 30 as input values for bonusPoints, so a new test is required. The input data may be either (20,true) or (30,false).

PREPRINT

- **Path 4** Nodes: (22..28)→(29)→(31)→(32)→(33..34)→(36..37)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(31): requires (bonusPoints<thresholdJump) to be false
  - Branch (31)→ (32): requires (bonusPoints>thresholdJump) – note node (32) modifies bonusPoints
  - Branch (33..34)→ (36..37): requires (modified bonusPoints>threshold) to be false

The input bonusPoints must be greater than or equal to zero on entry. At node (29) it must be greater than or equal to thresholdJump (20 or 30, depending on goldCustomer).

At node (31), bonusPoints must be greater than thresholdJump (20 or 30).

Therefore, bonusPoints must be greater than 0, greater or equal to than thresholdJump (20 or 30) at node(29), and greater than thresholdJump at node(31).

At node (32), the value of bonusPoints is reduced by thresholdJump, and at node (33..34) this value is increased by 4*thresholdJump, that is by 80 or 120.

So if the inputs are goldCustomer is true and bonusPoints is greater than 20, then on node (30), bonuspoints is reduced by 20, and at (33..34) bonusPoints increased by 80, giving an overall change of +60 to the value. The value must be less than or equal to 80 here, so the minimum value of bonusPoints is 1, and the maximum is 20.

And if the inputs are goldCustomer is false and bonusPoints equals 30, then on node (30) bonusPoints is reduced by 30, and at (33..34) bonuspoints increased by 120, giving an overall change of +90 to the value. The value must be less than or equal to 120 here, so the minimum value of bonusPoints is 1, and the maximum is 30.

Test T2.1 matches these constraints, with the input values (1,true).

- **Path 5** Nodes: (22..28)→(29)→(31)→(32)→(33..34)→(35)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(31): requires (bonusPoints>thresholdJump) to be false
  - Branch (31)→(32): requires (bonusPoints<thresholdJump) – note node (32) modifies bonusPoints
  - Branch (32)→(33..34) is always taken
  - Branch (33..34)→ (35): requires (modified bonusPoints>threshold)

  The input bonusPoints must be greater than or equal to zero on entry. At node (29) it must be greater than or equal to thresholdJump (20 or 30, depending on goldCustomer).

  At node (31), bonusPoints must be greater than thresholdJump (20 or 30).

  Therefore, bonusPoints must be greater than 0, greater or equal to than threshold-Jump (20 or 30) at node(29), and greater than thresholdJump at node(31).

  At node (32), the value of bonusPoints is reduced by thresholdJump, and at node (33..34) this value is increased by 4*thresholdJump, that is by 80 or 120.

  So if the inputs are goldCustomer is true and bonusPoints is greater than 20, then on node (30), bonuspoints is reduced by 20, and at (34) bonusPoints increased by 80, giving an overall change of +60 to the value at node (35). The value must be greater than 80 here, so the minimum value of bonusPoints is 21, and the maximum is large (Long.MAX_VALUE-60 to be exact).

  And if the inputs are goldCustomer is false and bonusPoints equals 30, then on node (30) bonusPoints is reduced by 30, and at (34) bonuspoints increased by 120, giving an overall change of +90 to the value at node (35). The value must be greater 120 here, so the minimum value of bonusPoints is 31, and the maximum again large (Long.MAX_VALUE-90).

  Test T1.1 matches these constraints, with the input values (40,true).

- **Path 6** Nodes: (22..28)→(29)→(30)→(31)→(33..34)→(36..37)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(30): requires (bonusPoints<thresholdJump) – note node (30) modifies bonusPoints
  - Branch (31)→(33..34): requires (modified bonusPoints>thresholdJump) to be false
  - Branch (33..34)→(36..37): requires (modified bonusPoints>threshold) to be false

  The input bonusPoints must be greater than or equal to zero on entry. At node (29) it must be less than thresholdJump (20 or 30, depending on goldCustomer).

  At node (31), bonusPoints must be less than or equal to thresholdJump (20 or 30).

  Therefore, bonusPoints must be greater than 0, less thresholdJump (20 or 30) at node(29), and less than or equal to thresholdJump at node(31).

  At node (30), the value of bonusPoints is reduced by thresholdJump, and at node (33..34) this value is increased by 4*thresholdJump, that is by 80 or 120.

  So if the inputs are goldCustomer is true and bonusPoints less than 20, then on node (30), bonuspoints is reduced by 20, and at (34) bonusPoints increased by 80, giving an overall change of +60 to the value at node (35). The value must be greater than 80 here, but the maximum value it can have is 59, so this is impossible.

  So if the inputs are goldCustomer is false and bonusPoints less than 30, then on node (30), bonuspoints is reduced by 30, and at (34) bonusPoints increased by 120, giving an overall change of +90 to the value at node (35). The value must be greater than 120 here, but the maximum value it can have is 119, so this is also impossible.

PREPRINT

- **Path 7** Nodes: (22..28)→(29)→(30)→(31)→(33..34)→(35)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(30): requires (bonusPoints<thresholdJump) – note node (30) modifies bonusPoints
  - Branch (31)→(33..34): requires (modified bonusPoints>thresholdJump) to be false
  - Branch (33..34)→ (35): requires (modified bonusPoints>threshold)

For any value of bonusPoints, if it is less than thresholdJump (20 or 30) at node(29), then it will always be less than threshold (80 or 120) at node(33..34), so this path is impossible.

- **Path 8** Nodes: (22..28)→(29)→(30)→(31)→(32)→(33..34)→(36..37)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(30): requires (bonusPoints<thresholdJump) – note node (30) modifies bonusPoints

  - Branch (31)→ (32): requires (bonusPoints>thresholdJump) – note node (32) modifies bonusPoints
  - Branch (33..34)→ (36..37): requires (modified bonusPoints>threshold) to be false

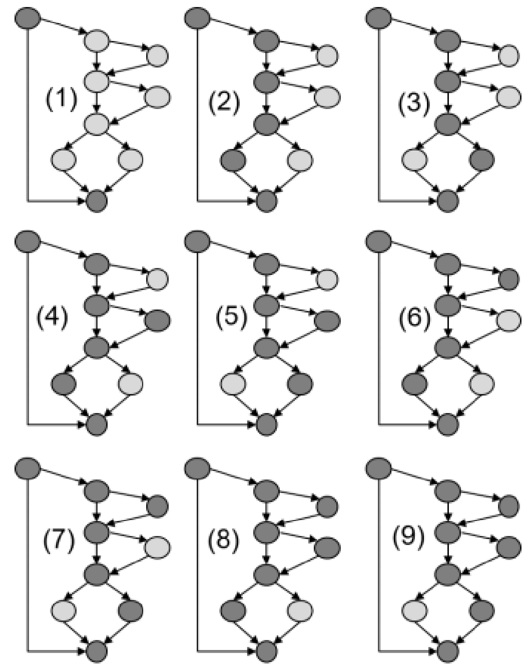
  For any value of bonusPoints, it is not possible to have bonusPoints less than thresholdJump at node 30, followed by the modified (reduced) value of bonusPoints greater than threasholdJump at node 31. So this path is impossible.

- **Path 9** Nodes: (22..28)→(29)→(30)→(31)→(32)→(33..34)→(35)→(38..41)
  - Branch (22..28)→(29): requires (bonusPoints>0)
  - Branch (29)→(30): requires (bonusPoints<thresholdJump) – note node (30) modifies bonusPoints
  - Branch (31)→ (32): requires (bonusPoints>thresholdJump) – note node (32) modifies bonusPoints
  - Branch (33..34)→ (35): requires (modified bonusPoints>threshold)

As for Path 8, for any value of bonusPoints, it is not possible to have bonusPoints less than thresholdJump at node 30, followed by the modified (reduced) value of bonusPoints greater than threasholdJump at node 31. So this path is impossible.

1. Path 1, nodes (22..28)–(38..41)
   covered by bonusPoints=-100, and goldCustomer=false (T1.4)
2. Path 2, nodes (22..28)–(29)–(31)–(33..34)–(36..37)–(38..41)
   not possible
3. Path 3, nodes (22..28)–(29)–(31)–(33..34)–(35)–(38..41)
   covered by 20, true or 30, false
4. Path 4, nodes (22..28)–(29)–(31)–(32)–(33..34)–(36..37)–(38..41)
   covered by 1, true (T2.1)
5. Path 5, nodes (22..28)–(29)–(31)–(32)–(33..34)–(35)–(38..41)
   covered by 40, true (T1.1)
6. Path 6, nodes (22..28)–(29)–(30)–(31)–33..34)–(36..37)–(38..41)
   not possible
7. Path 7, nodes (22..28)–(29)–(30)–(31)–(33..34)–(35)–(38..41)
   not possible
8. Path 8, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(36..37)–(38..41)
   not possible
9. Path 9, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(35)–(38..41)
   not possible

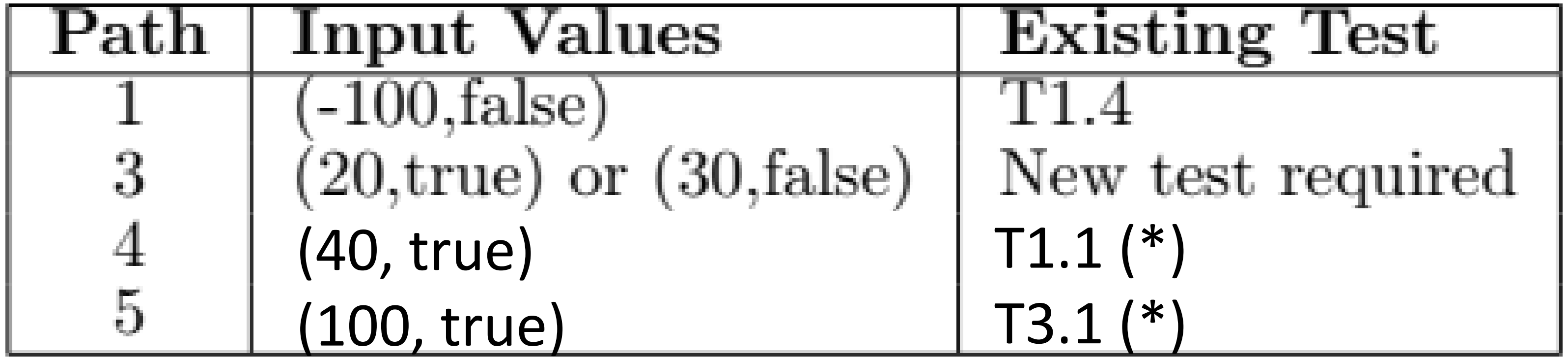
This is Hard Work!!!



- Path 1, nodes (22..28)–(38..41) covered by bonusPoints=-100, and goldCustomer=f
- Path 2, nodes (22..28)–(29)–(31)–(33..34)–(36..37)–(38..41) not possible
- Path 3, nodes (22..28)–(29)–(31)–(33..34)–(35)–(38..41) covered by 20, true or 30, false
- Path 4, nodes (22..28)–(29)–(31)–(32)–(33..34)–(36..37)–(38..41) covered by 40, true (T1.1) *
- Path 5, nodes (22..28)–(29)–(31)–(32)–(33..34)–(35)–(38..41) covered by 100,true (T3.1) *
- Path 6, nodes (22..28)–(29)–(30)–(31)–33..34)–(36..37)–(38..41) not possible
- Path 7, nodes (22..28)–(29)–(30)–(31)–(33..34)–(35)–(38..41) not possible
- Path 8, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(36..37)–(38..41) not possible
- Path 9, nodes (22..28)–(29)–(30)–(31)–(32)–(33..34)–(35)–(38..41) not possible

Analysis Results

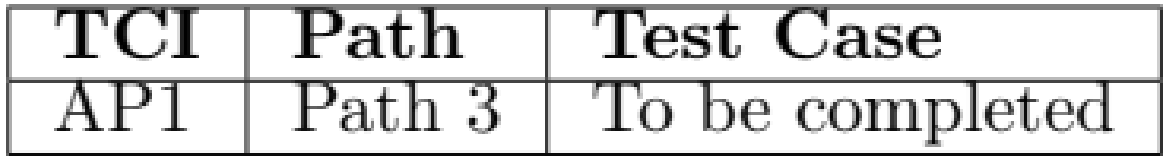
- This analysis of the CFG has produced three results:
    1. The possible end-to-end paths
    2. Where available, existing test data that causes a path to be taken
    3. Where no previous test causes a path to be taken, criteria or values for the new test data



Path	Input Values	Existing Test
1	(-100,false)	T1.4
3	(20,true) or (30,false)	New test required
4	(40, true)	T1.1 (*)
5	(100, true)	T3.1 (*)

Test Coverage Items for giveDiscount() with Fault 6

- Each uncovered path shown is a test coverage item



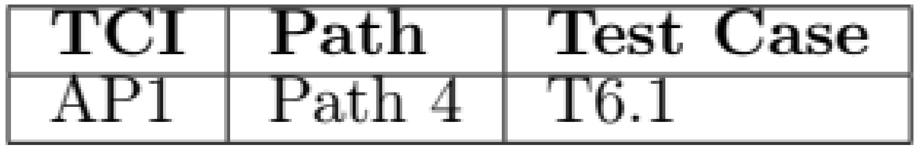
TCI	Path	Test Case
AP1	Path 3	To be completed

Test Cases

- Each path must be tested in a separate test case
- The analysis provides criteria for the necessary input data values for each
  - Path 3 requires (20, true)
- The data values for the expected results are derived from the specification
  - The input (20,true) produces the output FULLPRICE

Reviewing Your Work

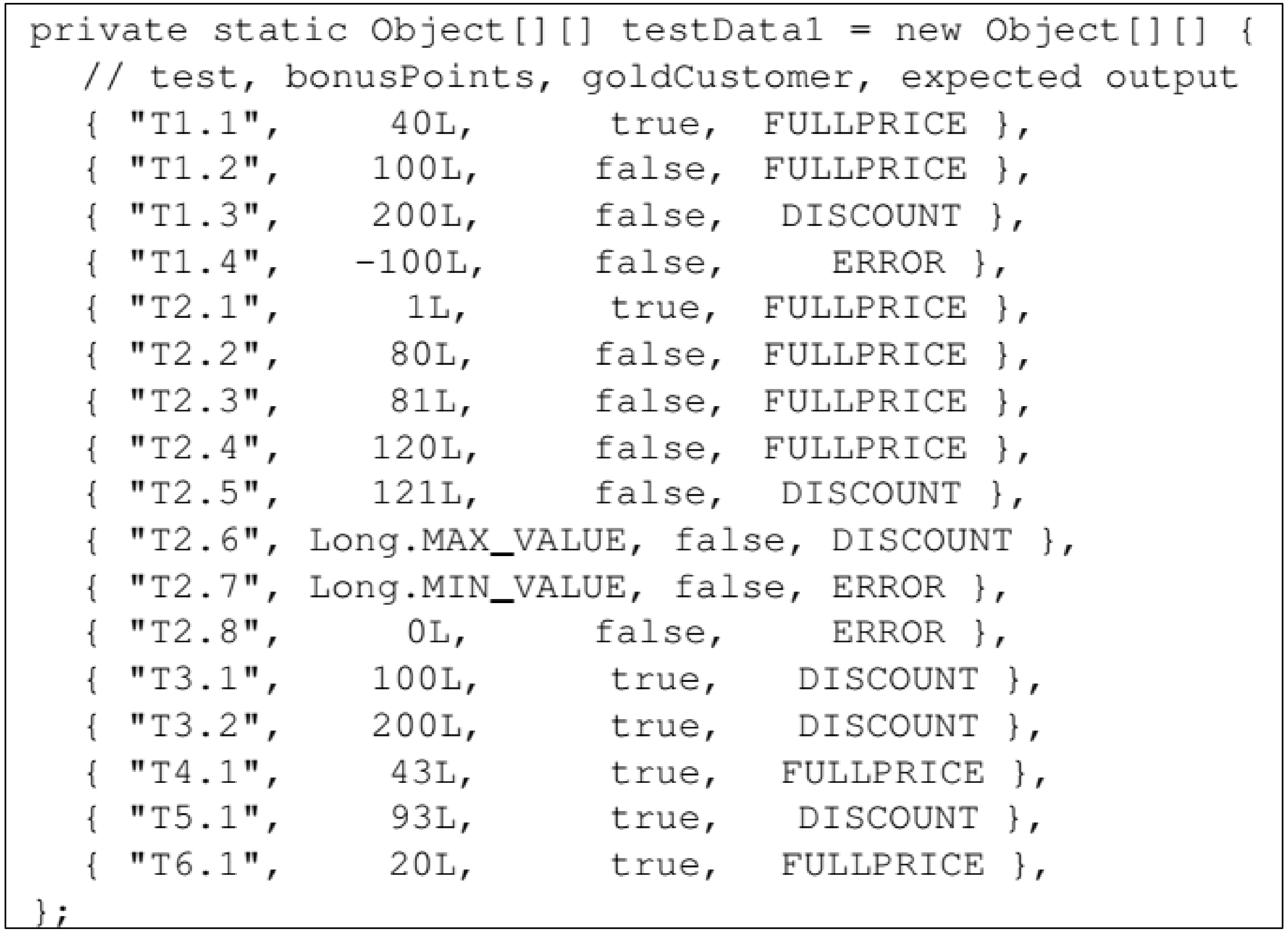
- Complete the TCI Table



TCI	Path	Test Case
AP1	Path 4	T6.1

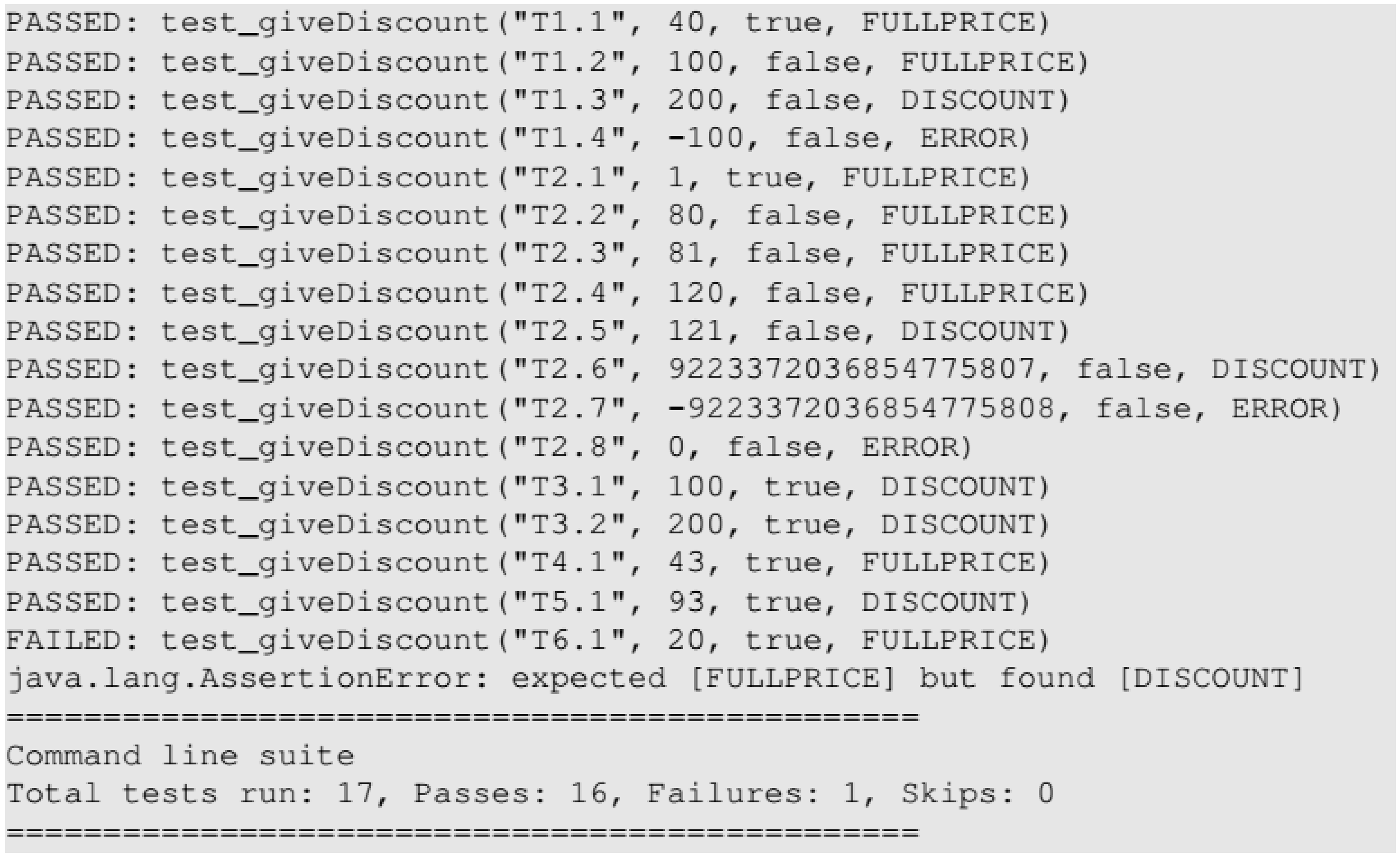
- The TCI Table shows that the single test coverage item is covered
- The Test Cases shows that test case T6.1 required to cover this test coverage item

Implementation



```
private static Object[][] testData1 = new Object[][] {
   // test, bonusPoints, goldCustomer, expected output
   { "T1.1",      40L,        true,  FULLPRICE },
   { "T1.2",     100L,       false,  FULLPRICE },
   { "T1.3",     200L,       false,   DISCOUNT },
   { "T1.4",    -100L,       false,      ERROR },
   { "T2.1",       1L,        true,  FULLPRICE },
   { "T2.2",      80L,       false,  FULLPRICE },
   { "T2.3",      81L,       false,  FULLPRICE },
   { "T2.4",     120L,       false,  FULLPRICE },
   { "T2.5",     121L,       false,   DISCOUNT },
   { "T2.6", Long.MAX_VALUE, false, DISCOUNT },
   { "T2.7", Long.MIN_VALUE, false, ERROR },
   { "T2.8",       0L,       false,      ERROR },
   { "T3.1",     100L,        true,    DISCOUNT },
   { "T3.2",     200L,        true,    DISCOUNT },
   { "T4.1",      43L,        true,   FULLPRICE },
   { "T5.1",      93L,        true,    DISCOUNT },
   { "T6.1",      20L,        true,   FULLPRICE },
};
```

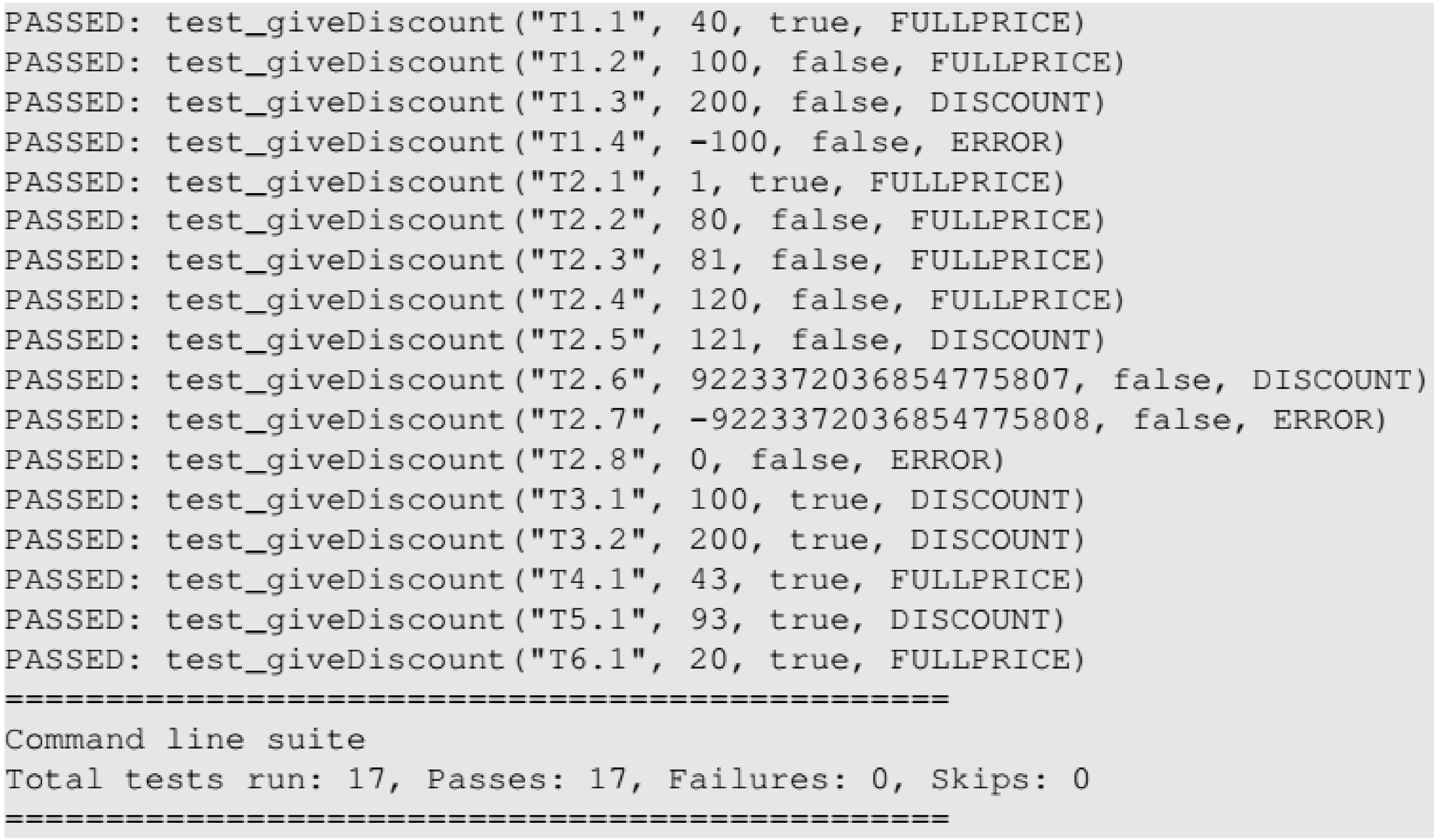
Test Results



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
PASSED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
PASSED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
FAILED: test_giveDiscount("T6.1", 20, true, FULLPRICE)
java.lang.AssertionError: expected [FULLPRICE] but found [DISCOUNT]
===============================================
Command line suite
Total tests run: 17, Passes: 16, Failures: 1, Skips: 0
===============================================
```

- Test T6.1 fails

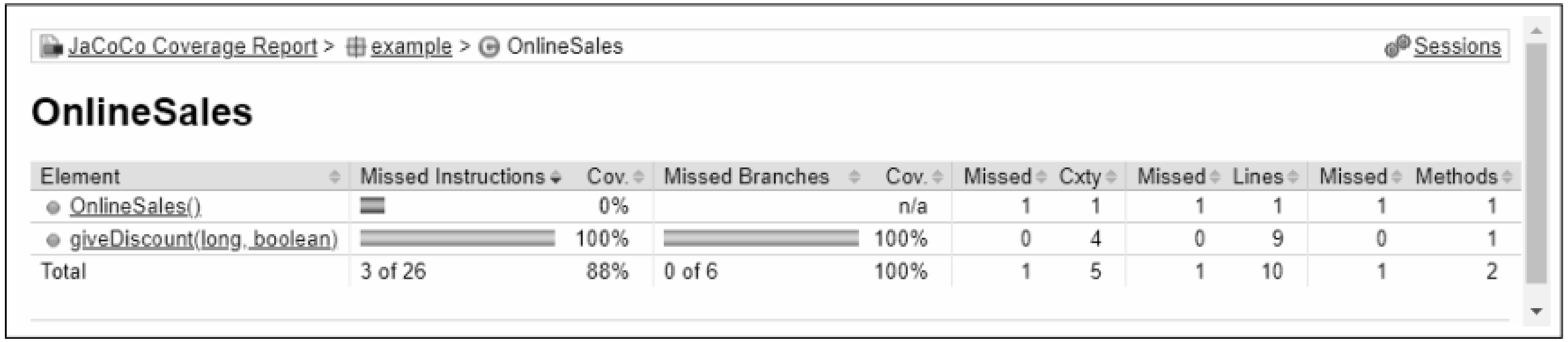
Test Results Against Correct Implementation



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
PASSED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
PASSED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
PASSED: test_giveDiscount("T6.1", 20, true, FULLPRICE)
===============================================
Command line suite
Total tests run: 17, Passes: 17, Failures: 0, Skips: 0
===============================================
```

Notes

- Note that the previous white-box tests (T4.1, T5.1) have been invalidated by changing the code
  - there is no guarantee that the statement coverage and branch coverage tests developed for Fault 5 provide full coverage for Fault 6
- However, the coverage tool confirms that full statement coverage and branch coverage have still been achieved by these tests



JaCoCo Coverage Report > example > OnlineSales Sessions

OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
OnlineSales()		0%		n/a	1	1	1	1	1	1
giveDiscount(long, boolean)		100%		100%	0	4	0	9	0	1
Total	3 of 26	88%	0 of 6	100%	1	5	1	10	1	2

- A manual analysis would be required to confirm that the four possible end to end paths have also been executed

Fault Model

- The all-paths fault model is where a particular sequence of operations, reflected by a particular path through the code, does not produce the correct result
- These faults are often associated with complex or deeply nested code structures where the wrong functionality is executed for a specific situation

Standard CFGs

```
1 int f()
2 {
3     int x,y;
4     x = 10;
5     y = x+3;
6     return y;
7 }
```

$$\begin{tikzcd} 1..7 \end{tikzcd}$$

Figure 7.11: CFG for Sequence

```
1 int f(int a)
2 {
3     int x=0;
4     if (a>10)
5         x=a;
6     return x;
7 }
```

$$\begin{tikzcd}
1..4 \arrow[dd, "\beta"'] \arrow[dr, "a>10\ \alpha"] & \\
& 5 \arrow[dl, "\gamma"] \\
6..7 &
\end{tikzcd}$$

Figure 7.12: CFG for Selection (if-then)

```
1 int f(int a)
2 {
3     int x=0;
4     if (a>10)
5         x=a;
6     else
7         x=-a;
8     return x;
9 }
```

$$\begin{tikzcd}
& 1..4 \arrow[dl] \arrow[dr, "a>10"] & \\
5 \arrow[dr] & & 6..7 \arrow[dl] \\
& 8..9 &
\end{tikzcd}$$

Figure 7.13: CFG for Selection (if-then-else)

```
    ...
 1 switch (a) {
 2     case 0:
 3         b=33;
 4         break
 5     case 1:
 6         b=-44;
 7         break;
 8     default:
 9         ok=false;
10         break;
11 }
    ...
```

$$\begin{tikzcd}
& 1 \arrow[dl, "a==0"'] \arrow[d, "a==1"] \arrow[dr, "default"] & \\
2..4 \arrow[dr] & 5..7 \arrow[d] & 8..10 \arrow[dl] \\
& 11 &
\end{tikzcd}$$

Figure 7.14: CFG for Selection (switch)

More Standard CFGs

```
1 int f(int a)
2 {
3     int x=a;
4     while (x>10)
5         x=x/2;
6     return x;
7 }
```

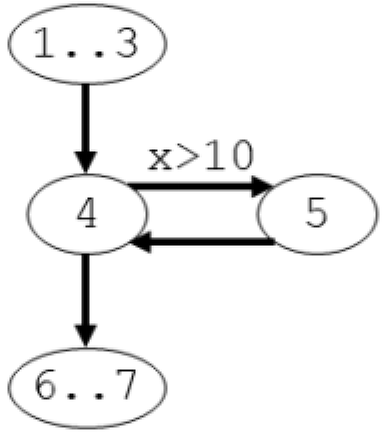


Figure 7.15: CFG for Iteration (while)

```
1 int f(int a)
2 {
3     int x=0;
4     for (int i=0; i<a; i++)
5         x = x+a+i;
6     return x;
7 }
```

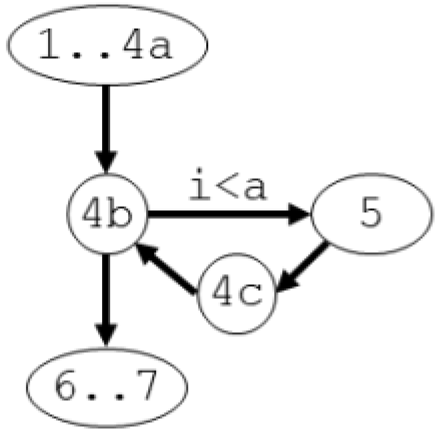


Figure 7.17: CFG for Iteration (for)

```
1 int f(int a)
2 {
3     int x=a;
4     do {
5         x=x/2;
6     } while (x>10);
7     return x;
8 }
```

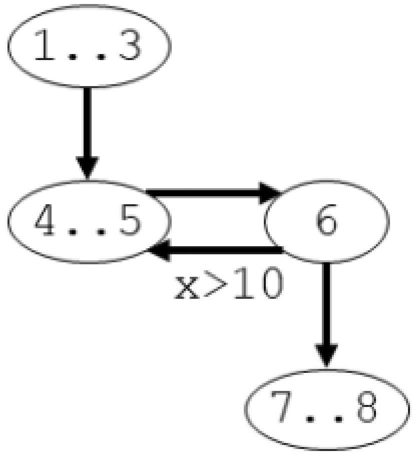


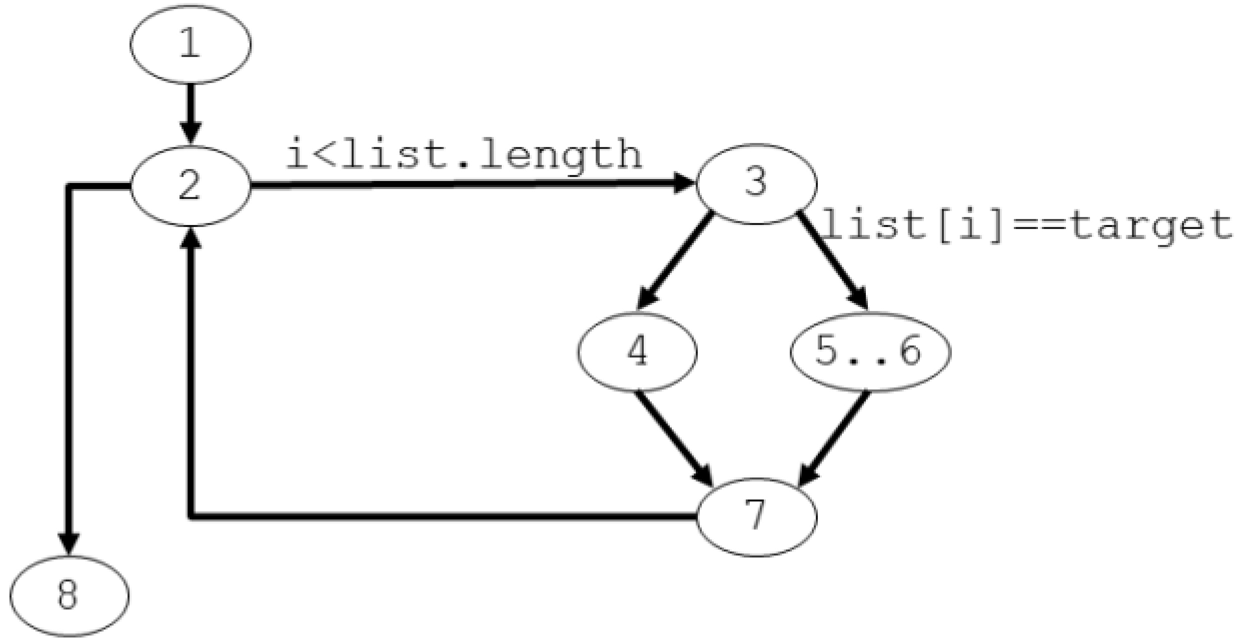
Figure 7.16: CFG for Iteration (do-while)

Basis Paths

- The flow of control through a CFG can be represented by a regular expression
- The expression is then evaluated to give the number of end-to-end paths
- Loops can be executed an indefinite number of times: this is resolved by treating them as code segments that can be executed exactly zero or one times
- There are three operators defined for the regular expression:
- . concatenation – this represents a sequence of nodes in the graph
- + selection – this represents a decision in the graph (if statement)
- $()*$ iteration – this represents a repetition in the graph (while statement)

Basis Paths Example

```
1    i=0;
2    while (i<list.length) {
3       if (list[i]==target)
4          match++;
5       else
6          mismatch++;
7       i++;
8    }
```



$$1 \cdot 2 \cdot (3 \cdot (4 + (5..6)) \cdot 7 \cdot 2)^{*} \cdot 8$$

Replace (x)* with (x+0)

$$1 \cdot 2 \cdot ((3 \cdot (4 + (5..6)) \cdot 7 \cdot 2) + 0) \cdot 8$$

Replace <n> with 1 and evaluate, using + and * as addition and multiplication

$$paths = 1 \cdot 1 \cdot ((1 \cdot (1 + 1) \cdot 1 \cdot 1) + 1) \cdot 1 = 3$$

Analysing The Possible Paths in Fault 6

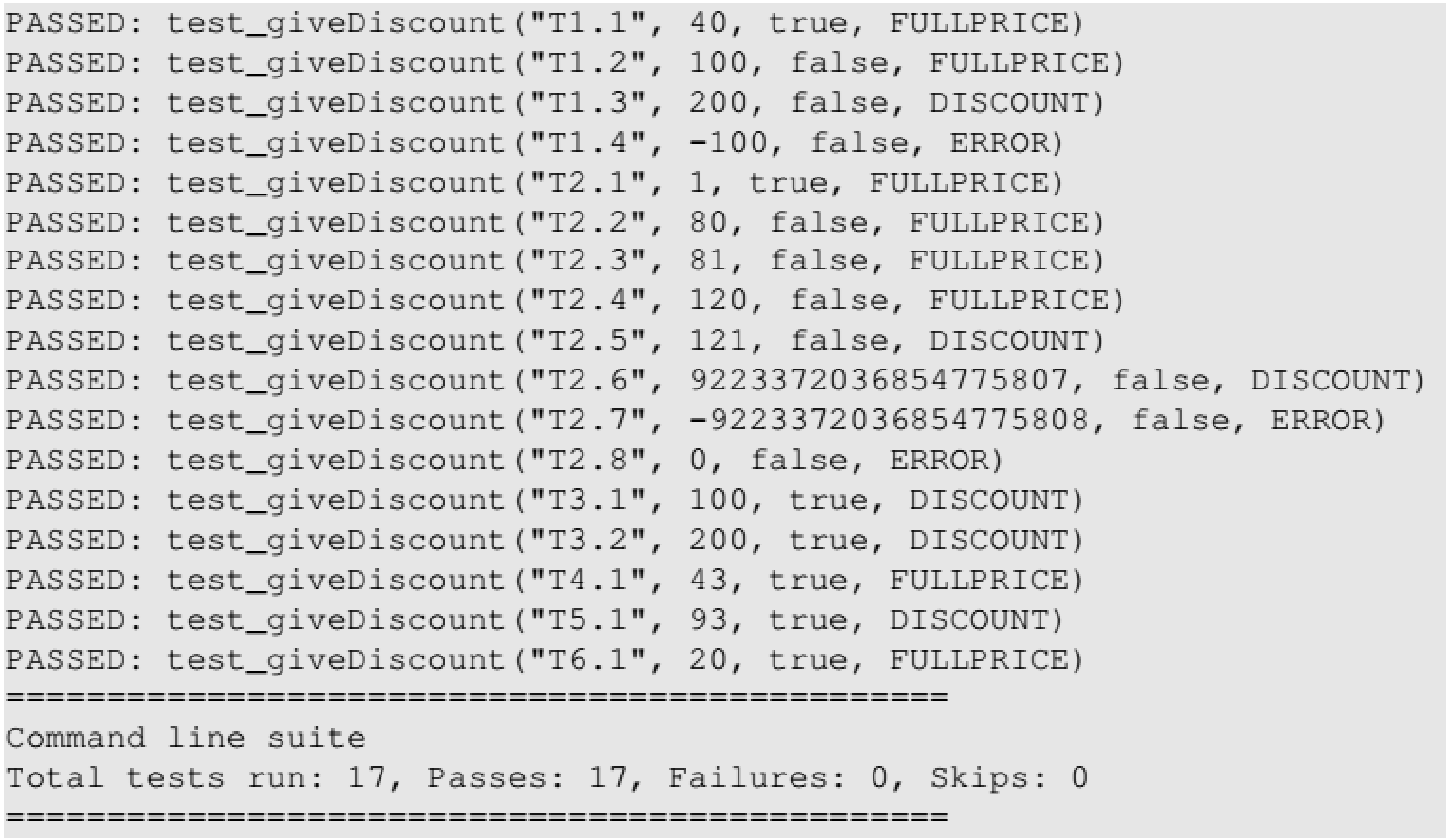
- Read the book: Section 7.4.5

Limitations

- Lines 26-33
  Fault 7
  lookup table contains a fault
- Line 37
  Fault 8
  faulty bitwise manipulation
- Line 45
  Fault 9
  Divide by zero fault in ‘dead code’

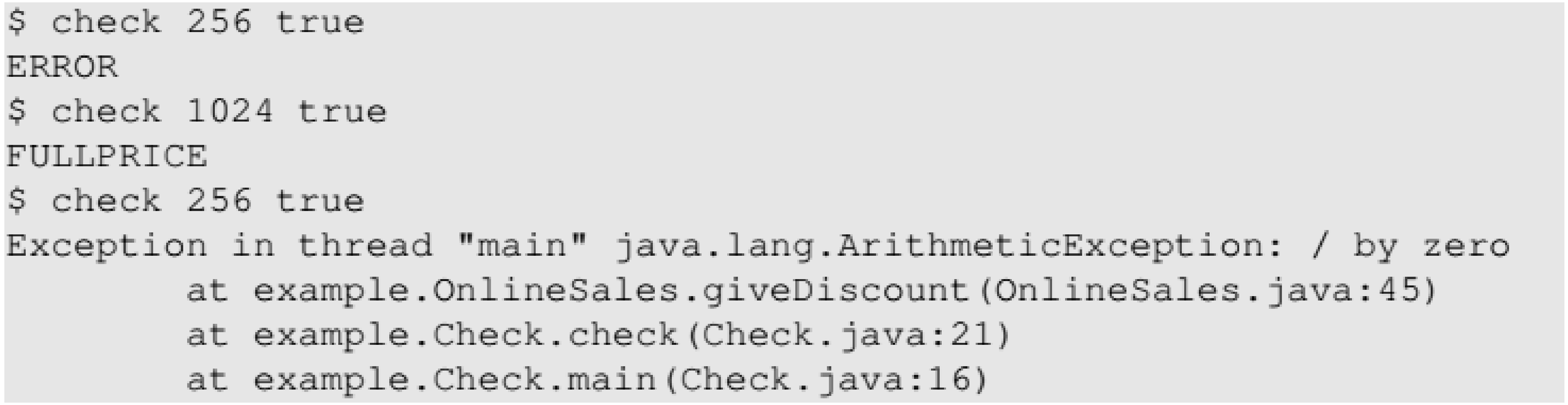
```
23    public static Status giveDiscount(long bonusPoints, boolean
              goldCustomer)
24    {
25
26       Object[][] lut=new Object[][] {
27          { Long.MIN_VALUE,              0L,  null, ERROR },
28          {             1L,             80L,  true, FULLPRICE },
29          {            81L, Long.MAX_VALUE,  true, DISCOUNT },
30          {             1L,            120L, false, FULLPRICE },
31          {           121L, Long.MAX_VALUE, false, DISCOUNT },
32          {          1024L,           1024L,  true, FULLPRICE },//Fault 7
33       };
34
35       Status rv = ERROR;
36
37       bonusPoints &=0xFFFFFFFFFFFFFEFFL; // Fault 8
38
39       for (Object[] row:lut)
40          if ( (bonusPoints>=(Long)row[0]) &&
41                (bonusPoints<=(Long)row[1]) &&
42                (((Boolean)row[2]==null)||((Boolean)row[2]==goldCustomer
                        )))
43             rv = (Status)row[3];
44
45       bonusPoints = 1/(bonusPoints-55); // Fault 9
46
47       return rv;
48    }
```

Test Results (Faults 7,8,9)



```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 80, false, FULLPRICE)
PASSED: test_giveDiscount("T2.3", 81, false, FULLPRICE)
PASSED: test_giveDiscount("T2.4", 120, false, FULLPRICE)
PASSED: test_giveDiscount("T2.5", 121, false, DISCOUNT)
PASSED: test_giveDiscount("T2.6", 9223372036854775807, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -9223372036854775808, false, ERROR)
PASSED: test_giveDiscount("T2.8", 0, false, ERROR)
PASSED: test_giveDiscount("T3.1", 100, true, DISCOUNT)
PASSED: test_giveDiscount("T3.2", 200, true, DISCOUNT)
PASSED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
PASSED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
PASSED: test_giveDiscount("T6.1", 20, true, FULLPRICE)
===============================================
Command line suite
Total tests run: 17, Passes: 17, Failures: 0, Skips: 0
===============================================
```

Demonstration of Faults 7,8,9



```
$ check 256 true
ERROR
$ check 1024 true
FULLPRICE
$ check 256 true
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at example.OnlineSales.giveDiscount(OnlineSales.java:45)
        at example.Check.check(Check.java:21)
        at example.Check.main(Check.java:16)
```

- The input (256,true) should return DISCOUNT
- The input (1024,true) should also return DISCOUNT
- The input (55,true) should return FULLPRICE, and not raise an exception

Strengths and Weaknesses

- **Strengths**
  - Covers all possible paths, which may have not been exercised using other methods
  - Guarantees statement coverage and branch coverage coverage
- **Weaknesses**
  - Difficult and time consuming
  - Must limit loops for practical reasons – weakens the testing
  - All-paths does not explicitly evaluate the boolean conditions in each decision
  - Does not explore faults related to incorrect data processing (e.g. bitwise manipulation or arithmetic errors)
  - Does not explore non-code faults (for example, faults in a lookup table)

Key Points

- All paths testing is used to augment black-box and white-box testing, by ensuring that every end-to-end path is executed
- It is the strongest form of testing based on a program’s structure
- Every unexecuted end-to-end path in the software is a test coverage item
- Input values for the test data are selected by analysis of the paths and decisions in the code

Notes for the Experienced Tester

- Identifying all the possible paths is a complex task
- Develop the CFG mentally
- Identify the required conditions mentally (or using the debugger)
- Even for the experienced tester, developing all-paths tests is a time consuming and difficult task