

# CS608

# Software Testing

Dr. Stephen Brown

Room Eolas 116

[stephen.brown@mu.ie](mailto:stephen.brown@mu.ie)

# CS608

## White-Box Testing

### Statement Coverage and Branch Coverage

(Essentials of Software Testing, Chapters 5 & 6)

# PART I – STATEMENT COVERAGE

# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered

# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered

# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered

# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered

# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered



# Introduction to Statement Coverage

- Previously, we have used basic black-box test techniques (EP, BVA, DT)
- These provide a basic level of testing against the specification
- However, faults may remain in unexecuted components of the code
- In order to try and find these, tests based on the implementation (code) instead of the specification are used – these techniques are referred to as white-box testing or structure-based testing
- These test techniques identify particular parts of the code, and test that the correct result is produced when these are exercised
- In practice, the coverage of the code components achieved by executing the black-box tests is usually measured automatically, and then white-box tests are developed only for components not already covered

# White Box: Expected Results

- Test coverage items are derived from the **implementation**
- Expected results must always come from the **specification**
- It is easy to make the mistake of using the source code when it is in front of you, and the output is obvious from the code
- As we have discussed, using the source code to derive the expected results only verifies that you have understood the code properly, and not that the code works

# White Box: Expected Results

- Test coverage items are derived from the **implementation**
- Expected results must always come from the **specification**
- It is easy to make the mistake of using the source code when it is in front of you, and the output is obvious from the code
- As we have discussed, using the source code to derive the expected results only verifies that you have understood the code properly, and not that the code works

# Testing with Statement Coverage

- Statement coverage (SC) testing (or more formally testing with statement coverage criteria) ensures that every line, or statement, of the source code is executed during tests, while verifying that the output is correct

## **Definition:**

a statement is a line in the source code that can be executed  
(Not all lines can be executed: comments, blank lines, {}'s, etc.)

# Statement Coverage Measurement

- The lines in the source code which have been executed can be measured automatically for most programming languages
  - We will use “JaCoCo” – see [www.jacoco.org](http://www.jacoco.org)
- This supports the measurement of statement coverage during testing
- We will develop tests for our example to ensure that all the statements are executed during testing, helping to find faults related to unexecuted code

# Example

- Continue testing `OnlineSales.giveDiscount()`
- Summary – the method returns:

FULLPRICE if `bonusPoints ≤ 120` and not a goldCustomer

FULLPRICE if `bonusPoints ≤ 80` and a goldCustomer

DISCOUNT if `bonusPoints > 120`

DISCOUNT if `bonusPoints > 80` and a goldCustomer

ERROR if any inputs are invalid (`bonusPoints < 1`)

# Step 1. Analysis

## Identifying Unexecuted Statements


- For code that has already been tested using black-box techniques, the first step in achieving full statement coverage is to examine the coverage results of these existing tests

# Coverage Report from Black-Box Tests

- Running the full set of existing tests (equivalence partition, boundary value analysis and decision tables) against Fault 4 with code coverage measurement enabled, produces:
  - the test report
  - a coverage summary report
  - a source code coverage report








# BBT Coverage Summary for giveDiscount()

 [JaCoCo Coverage Report](#) >  [example](#) >  [OnlineSales](#)

 [Sessions](#)

## OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
 <a href="#">OnlineSales()</a>		0%		n/a	1	1	1	1	1	1
 <a href="#">giveDiscount(long, boolean)</a>		93%		87%	1	5	1	11	0	1
Total	5 of 32	84%	1 of 8	87%	2	6	2	12	1	2

# Coverage Summary Report

JaCoCo Coverage Report > example > OnlineSales

Sessions

OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
OnlineSales()	<div><div></div></div>	0%		n/a	1	1	1	1	1	1
giveDiscount(long, boolean)	<div><div></div></div>	93%	<div><div></div></div>	87%	1	5	1	11	0	1
Total	5 of 32	84%	1 of 8	87%	2	6	2	12	1	2

- For statement coverage, see:
  - The **Lines** column and the associated **Missed** column
  - Only interested in the method under test: giveDiscount(long, boolean)
- Lines shows the number of lines in the method: 11
- Missed shows the number of missed (unexecuted) lines in the method: 1
- The tests have missed 1 out of 11 lines in the method, and have not achieved full statement coverage

# Identifying Unexecuted Statements

- The next step is to determine which lines have not been executed
- Examine the source code coverage report

## OnlineSales.java

```
1. package example;
2. // Note: this version contains Fault 4
3. import static example.OnlineSales.Status.*;
4.
5. public class OnlineSales {
6.
7.     public static enum Status { FULLPRICE, DISCOUNT, ERROR };
8.
9.     /**
10.      * Determine whether to give a discount for online sales.
11.      * Gold customers get a discount above 80 bonus points.
12.      * Other customers get a discount above 120 bonus points.
13.      *
14.      * @param bonusPoints How many bonus points the customer has accumulated
15.      * @param goldCustomer Whether the customer is a Gold Customer
16.      *
17.      * @return
18.      * DISCOUNT - give a discount<br>
19.      * FULLPRICE - charge the full price<br>
20.      * ERROR - invalid inputs
21.      */
22.     public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.     {
24.         Status rv = FULLPRICE;
25.         long threshold=120;
26.
27.         if (bonusPoints<=0)
28.             rv = ERROR;
29.
30.         else {
31.             if (goldCustomer)
32.                 threshold = 80;
33.             if (bonusPoints>threshold)
34.                 rv=DISCOUNT;
35.         }
36.
37.         if (bonusPoints==43) // fault 4
38.             rv = DISCOUNT;
39.
40.         return rv;
41.     }
42.
43. }
```

# Coverage Report

- Green lines:
  - executed
- Yellow lines:
  - partially executed
- Red lines:
  - not executed
- Only interested in the lines in the method giveDiscount() on lines 22-41

```
21.  */
22.  public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.  {
24.      Status rv = FULLPRICE;
25.      long threshold=120;
26.
27.      ◆ if (bonusPoints<=0)
28.          rv = ERROR;
29.
30.      else {
31.          ◆ if (goldCustomer)
32.              threshold = 80;
33.          ◆ if (bonusPoints>threshold)
34.              rv=DISCOUNT;
35.      }
36.
37.      ◆ if (bonusPoints==43) // fault 4
38.          rv = DISCOUNT;
39.
40.      return rv;
41.  }
```

# Notes

- Remember, we are testing the method `giveDiscount()` and not the entire class
- The coverage shows that some of the compiler generated code for the class has not been executed
- This is of relevance when we consider object-oriented testing

## OnlineSales.java

```
1. package example;
2. // Note: this version contains Fault 4
3. import static example.OnlineSales.Status.*;
4.
5. public class OnlineSales {
6.
7.     public static enum Status { FULLPRICE, DISCOUNT, ERROR };
8.
9.     /**
10.      * Determine whether to give a discount for online sales.
11.      * Gold customers get a discount above 80 bonus points.
12.      * Other customers get a discount above 120 bonus points.
13.      *
14.      * @param bonusPoints How many bonus points the customer has accumulated
15.      * @param goldCustomer Whether the customer is a Gold Customer
16.      *
17.      * @return
18.      * DISCOUNT - give a discount<br>
19.      * FULLPRICE - charge the full price<br>
20.      * ERROR - invalid inputs
21.      */
22.     public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.     {
24.         Status rv = FULLPRICE;
25.         long threshold=120;
26.
27.         if (bonusPoints<=0)
28.             rv = ERROR;
29.
30.         else {
31.             if (goldCustomer)
32.                 threshold = 80;
33.             if (bonusPoints>threshold)
34.                 rv=DISCOUNT;
35.         }
36.
37.         if (bonusPoints==43) // fault 4
38.             rv = DISCOUNT;
39.
40.         return rv;
41.     }
42.
43. }
```

# Statement Coverage Analysis

- Line 38 – red – not executed
- Line 37 – yellow – partially executed
- Lines 24-34, 40 – green - executed


```
21.  */
22.  public static Status giveDiscount(long bonusPoints) {
23.  {
24.      Status rv = FULLPRICE;
25.      long threshold=120;
26.
27.      ◆ if (bonusPoints<=0)
28.          rv = ERROR;
29.
30.      else {
31.      ◆ if (goldCustomer)
32.          threshold = 80;
33.      ◆ if (bonusPoints>threshold)
34.          rv=DISCOUNT;
35.      }
36.
37.      ◆ if (bonusPoints==43) // fault 4
38.          rv = DISCOUNT;
39.
40.      return rv;
41.  }
```

# Uncovered Statements and Conditions

- By examining the source code and the coverage results, identify:
  - unexecuted lines of code
  - conditions that must be met to cause these lines to be executed
  - required input values to meet these conditions
- In this example, we can identify the unexecuted statements: line 38

ID	Line Number	Condition
1	38	

# Uncovered Statements and Conditions

```
37.   if (bonusPoints==43) // fault 4  
38.  rv = DISCOUNT;
```

- By a careful examination of the source code, we can identify the condition that will cause line 38 to execute:
  - bonusPoints==43 must be true on line 37, which means that
  - bonusPoints must have the value 43 at this point in the code execution



# Uncovered Statements and Conditions

- As bonusPoints is an input parameter, and is not modified in the code, it is easy to identify the required input condition:
  - The input bonusPoints must have the value 43
- When the condition is the result of a calculation in the code, more detailed analysis may be required!!

<b>ID</b>	<b>Line Number</b>	<b>Condition</b>
1	38	bonusPoints==43

## 2. Extra Test Coverage Items

- Each unexecuted statement is a TCI

TCI	Line Number	Test Case
SC1	38	To be completed

- The test coverage items and test cases are marked as being **extra** to indicate that on their own they do not provide full statement coverage
- The complementary tests required to provide full statement coverage must be documented: in this example it is all the equivalence partition, boundary value analysis, and decision tables tests as developed previously
- Little value in identifying which existing BB test covers which statement...

### 3. Developing the Test Cases

- Test cases are developed to execute all the previously unexecuted statements
- Experience allows a minimum set of test cases – but emphasis is on full coverage
- The analysis has identified the condition required for each uncovered statement to be executed, and this assists in developing the test data
- In this example as we have already worked out that to execute the code on line 38, the parameter bonusPoints must have the value 43
- According to the specification, if bonusPoints is 43, then the result will always be FULLPRICE, irrespective of the value of the goldCustomer parameter
- This allows any value to be picked for goldCustomer, be systematic as before

# Extra Test Cases

ID	TCI	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T4.1	SC1	43	false	FULLPRICE

- Unlike in black-box testing, there is no differentiation between error and non-error cases for white-box testing

## 4. Test Design Verification

- Two parts:
  1. Complete the test coverage item table
  2. Review your work

# Completed Extra Test Coverage Item Table

TCI	Line	Test Case
SC1	38	T4.1

# Reviewing Your Work

1. Every TCI is covered by a test case
  - Shown in the completed TCI table
2. Every decision table test case covers a different TCI
  - Shown in the test case table
3. No duplicate tests (including previously developed tests)
  - Shown in the test case table (and reviewing TC Tables for EP, BVA and DT)

## 5. Implementation

- The full test implementation includes the previously developed equivalence partition and boundary value analysis tests
- Full statement testing is not achieved unless these previously developed tests are included!!

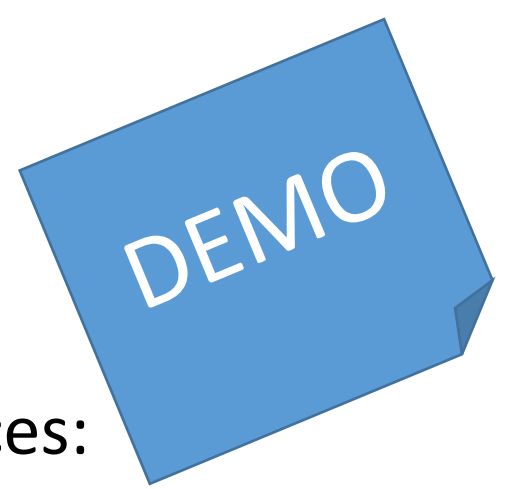


# OnlineSalesTest with SC Coverage

testData1[] is extended

```
1 public class OnlineSalesTest {
2
3     private static Object[][] testData1 = new Object[][] {
4         // test, bonuspoints, goldCustomer, expected output
5         { "T1.1",      40L,      true,  FULLPRICE },
6         { "T1.2",     100L,     false,  FULLPRICE },
7         { "T1.3",     200L,     false,  DISCOUNT },
8         { "T1.4",    -100L,     false,   ERROR },
9         { "T2.1",      1L,      true,  FULLPRICE },
10        { "T2.2",     80L,     false,  FULLPRICE },
11        { "T2.3",     81L,     false,  FULLPRICE },
12        { "T2.4",    120L,     false,  FULLPRICE },
13        { "T2.5",    121L,     false,  DISCOUNT },
14        { "T2.6", Long.MAX_VALUE, false,  DISCOUNT },
15        { "T2.7", Long.MIN_VALUE, false,  ERROR },
16        { "T2.8",      0L,     false,   ERROR },
17        { "T3.1",    100L,     true,   DISCOUNT },
18        { "T3.2",    200L,     true,   DISCOUNT },
19        { "T4.1",     43L,     true,  FULLPRICE },
20    };
```

## 6. Test Execution (Fault 4)



- Running these tests against the class OnlineSales produces:

```
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)
FAILED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
java.lang.AssertionError: expected [FULLPRICE] but found [DISCOUNT]
=====
Command line suite
Total tests run: 15, Passes: 14, Failures: 1, Skips: 0
=====
```

## 7. Interpreting the Test Results




```
FAILED: test_giveDiscount("T4.1", 43, true, FULLPRICE)
java.lang.AssertionError: expected [FULLPRICE] but found [DISCOUNT]
=====
Command line suite
Total tests run: 15, Passes: 14, Failures: 1, Skips: 0
=====
```

- The statement coverage test T4.1 fails indicating a fault in the code: Fault 4 has been found by this test
- There is no value in running these tests against any other versions of the code, unless it has been verified that the extra test continues to provide 100% code coverage
- The test data has been specifically developed to ensure statement coverage against the code version that contains Fault 4

# Checking the Coverage

[JaCoCo Coverage Report](#) > [example](#) > [OnlineSales](#) [Sessions](#)

## OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
<a href="#">OnlineSales()</a>		0%		n/a	1	1	1	1	1	1
<a href="#">giveDiscount(long, boolean)</a>		100%		100%	0	5	0	11	0	1
Total	3 of 32	90%	0 of 8	100%	1	6	1	12	1	2

- Full statement coverage has been achieved for giveDiscount(long, Boolean) – there are no missed lines for this method
- There is no value in viewing the annotated source code once 100% coverage has been achieved

## 2 Things to Check

- Did the tests pass
- Did you achieve 100% statement coverage

# PART II – BRANCH COVERAGE

# Branch Coverage

- Essentially the same, except that we consider branches instead of statements
- Note that there are different ways to measure branches:
  - Every statement with a decision in being true or false
  - Every binary condition in a decision being true or false
- JaCoCo measures the outcome of every Boolean condition in a decision as a separate branch
  - Based on the Java bytecode

# Branches Explained

```
1  if (x && y)
2      z = 10;
3  else
4      z = 20;
```

## Decisions-Based

Two branches:

1. Line 1 to line 2 if x and y are true
2. Line 1 to line 4 otherwise

## Boolean Condition-Based (JaCoCo) (Short-Circuit Evaluation)

Four branches:

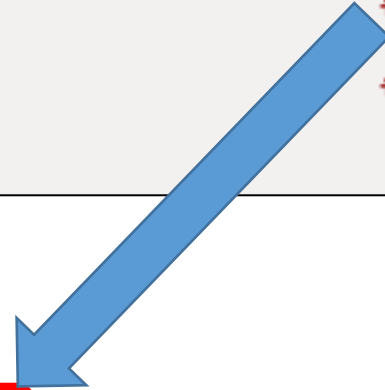
1. Condition x to y if x is true
2. Condition x to line 4 if x is false
3. Condition y to line 2 if y is true
4. Condition y to line 4 if y is false



# Add Fault 5

```
22 public static Status giveDiscount(long bonusPoints, boolean
    goldCustomer)
23 {
24     Status rv = FULLPRICE;
25     long threshold=120;
26
27     if (bonusPoints<=0)
28         rv = ERROR;
29
30     else {
31         if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33         if (bonusPoints>threshold)
34             rv=DISCOUNT;
35     }
36
37     return rv;
38 }
```

```
30 else {
31     if (goldCustomer)
32         threshold = 80;
33     if (bonusPoints>threshold)
34         rv=DISCOUNT;
35 }
```



# Fault 5 Test Results (SC)



```
=====
Command line suite
Total tests run: 15, Passes: 15, Failures: 0, Skips: 0
=====
```

Note: not found by statement coverage

# Analysis







- View the branch coverage of `giveDiscount()`
- Identify untaken branches
- Add tests for these

# Counting Untaken Branches

JaCoCo Coverage Report > example > OnlineSales



 [Sessions](#)

## OnlineSales

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
 <a href="#">OnlineSales()</a>		0%		n/a	1	1	1	1	1	1
 <a href="#">giveDiscount(long, boolean)</a>		100%		87%	1	5	0	9	0	1
Total	3 of 30	90%	1 of 8	87%	2	6	1	10	1	2

# Is the Test Valid against Fault 5?

- It is important to note that the statement coverage tests were developed to provide full statement coverage of `giveDiscount()` with Fault 4, and not with Fault 5
- The tests are therefore not guaranteed to provide statement coverage for the version with Fault 5
- However, in this example, we can see from the coverage summary that 100% statement coverage has been achieved (Lines Missed is 0)

● <a href="#">giveDiscount(long, boolean)</a>		100%		87%	1	5	0	9	0	1
-----------------------------------------------	-------------------------------------------------------------------------------------	------	---------------------------------------------------------------------------------------	-----	---	---	---	---	---	---

# Identifying Untaken Branches/Coverage Report

```
22. public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23. {
24.     Status rv = FULLPRICE;
25.     long threshold=120;
26.
27.     ◆ if (bonusPoints<=0)
28.         rv = ERROR;
29.
30.     else {
31.     ◆ if (goldCustomer && bonusPoints!=93) // fault5
32.         threshold = 80;
33.     ◆ if (bonusPoints>threshold)
34.         rv=DISCOUNT;
35.     }
36.
37.     return rv;
38. }
```

Line 31 – yellow – not fully executed – one or more untaken branches



# Identifying Untaken Branches/Coverage Report

## Hover the Cursor (on the Yellow Diamond)

```
22. public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23. {
24.     Status rv = FULLPRICE;
25.     long threshold=120;
26.
27.     if (bonusPoints<=0)
28.         rv = ERROR;
29.
30.     else {
31.         if (goldCustomer && bonusPoints!=93) // fault5
32.             threshold = 80;
33.         if (bonusPoints>threshold)
34.             rv=DISCOUNT;
35.     }
36.
37.     return rv;
38. }
```

Line 31 – yellow – not fully executed – one or more untaken branches

# Hover the Cursor (on the Yellow Diamond)

```
30.      else {  
31.       if (goldCustomer && bonusPoints!=93) /  
32.      threshold = 80;  
33.       if (bonusPoints>threshold)  
34.      rv=DISCOUNT;  
35.      }
```

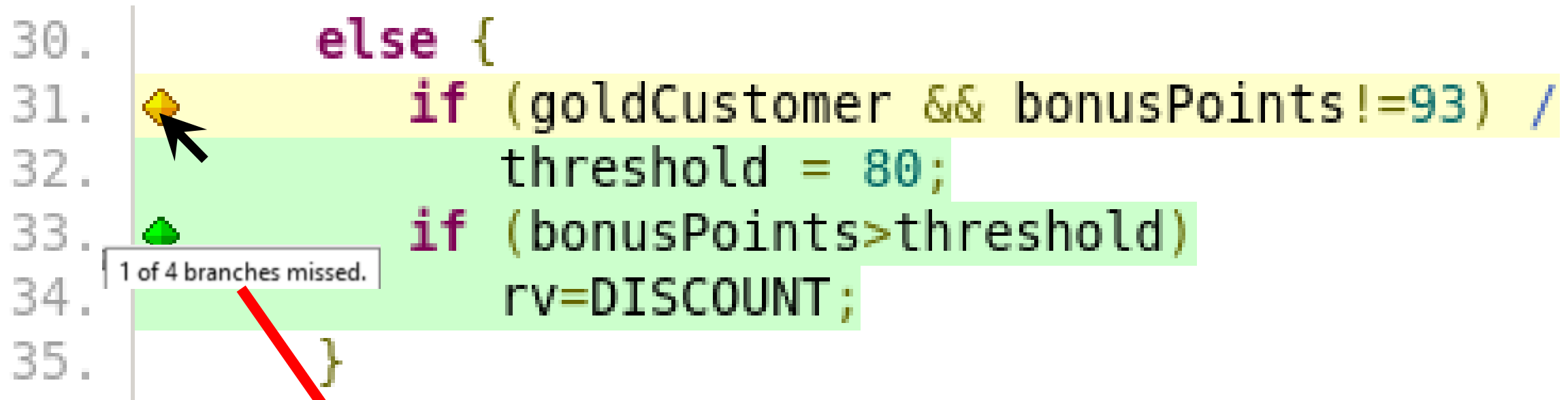
1 of 4 branches missed.

The report 1 of 4 branches missed. indicates that the line contains four branches, and that one of these branches has not been taken



# Hover the Cursor (on the Yellow Diamond)


```
30.      else {  
31.          if (goldCustomer && bonusPoints!=93) /  
32.              threshold = 80;  
33.          if (bonusPoints>threshold)  
34.              rv=DISCOUNT;  
35.      }
```



The report **1 of 4 branches missed.** indicates that the line contains four branches, and that one of these branches has not been taken


# Understanding the Branches

- Recall that JaCoCo is typical of Java branch coverage tools in that it counts branches for the outcome of each boolean condition and not for the decision itself
- We can now examine the branch coverage report in more detail

```
31.      if (goldCustomer && bonusPoints!=93)
```

- There is one decision on line 31 (a compound decision):
  - Decision 1: goldCustomer && bonusPoints!=93
- There are two boolean conditions in the decision on line 31:
  - Condition 1: goldCustomer
  - Condition 2: bonusPoints!=93

# Understanding the Branches

```
31.   if (goldCustomer && bonusPoints!=93)
```

- There are 4 branches in the code, two for each boolean condition: one for the true outcome, and one for the false outcome:
  - Branch 1: goldCustomer is false
  - Branch 2: goldCustomer is true
  - Branch 3: bonusPoints==93
  - Branch 4: bonusPoints!=93
- We obviously only reach branches 3 and 4 if goldCustomer is true
- This is called 'short circuit' evaluation

# Review the Test Data

- A review of the test data shows that the value 93 has not been used in the tests, and therefore the untaken branch is the one identified as branch 3 in the list above

```
31.        if (goldCustomer && bonusPoints!=93)
```

- The requirement for this branch to be taken is that goldCustomer is true and bonusPoints is equal to 93 (examine the code)

Branch	Start Line	End Line	Condition
B1	31 (branch 3)	33	goldCustomer && bonusPoints==93

## 2. (Extra) Test Coverage Items

<b>Test Coverage Item</b>	<b>Branch</b>	<b>Test Case</b>
BC1	B1	To be completed

### 3. (Extra) Test Cases

ID	TCI	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T5.1	BC1	93	true	DISCOUNT

## 4. Review: Complete the TCI Table

<b>Test Coverage Item</b>	<b>Branch</b>	<b>Test</b>
BC1	B1	T5.1

## 4. Complete the Review

- In the Extra TCI Table 6.4 that every test case is covered
- In the Extra TC Table, every branch coverage test covers a new branch coverage TCI
- Comparing the test data for the equivalence partition, boundary value analysis, decision tables, statement coverage, and branch coverage tests we can see that there are no duplicate tests
- This confirms that:
  - there are no missing tests
  - there are no unnecessary tests



## 5. Test Implementation

```
1 public class OnlineSalesTest {
2
3     private static Object[][] testData1 = new Object[][] {
4         // test, bonuspoints, goldCustomer, expected output
5         { "T1.1",      40L,      true,  FULLPRICE },
6         { "T1.2",     100L,     false,  FULLPRICE },
7         { "T1.3",     200L,     false,  DISCOUNT },
8         { "T1.4",    -100L,     false,   ERROR },
9         { "T2.1",       1L,      true,  FULLPRICE },
10        { "T2.2",      80L,     false,  FULLPRICE },
11        { "T2.3",      81L,     false,  FULLPRICE },
12        { "T2.4",     120L,     false,  FULLPRICE },
13        { "T2.5",     121L,     false,  DISCOUNT },
14        { "T2.6", Long.MAX_VALUE, false,  DISCOUNT },
15        { "T2.7", Long.MIN_VALUE, false,  ERROR },
16        { "T2.8",       0L,     false,   ERROR },
17        { "T3.1",     100L,     true,   DISCOUNT },
18        { "T3.2",     200L,     true,   DISCOUNT },
19        { "T4.1",      43L,     true,  FULLPRICE },
20        { "T5.1",      93L,     true,   DISCOUNT }
```

# 6 Test Execution




```
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)
FAILED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
java.lang.AssertionError: expected [DISCOUNT] but found [FULLPRICE]
=====
Command line suite
Total tests run: 16, Passes: 15, Failures: 1, Skips: 0
=====
```

# Test Results






```
FAILED: test_giveDiscount("T5.1", 93, true, DISCOUNT)
java.lang.AssertionError: expected [DISCOUNT] but found [FULLPRICE]
=====
Command line suite
Total tests run: 16, Passes: 15, Failures: 1, Skips: 0
=====
```

- Test T5.1 fails indicating that the fault in the code (Fault 5) has been found by the full branch coverage tests

# Test Coverage Results

 [JaCoCo Coverage Report](#) >  [example](#) >  OnlineSales

## OnlineSales

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

- Full branch coverage has been achieved – there are no missed branches (100% coverage shown)

## 2 Things to Check

- Did the tests pass
- Did you achieve 100% branch coverage

# Next Week

- SC & BC in More Detail
  - Fault Model
  - Description, Analysis, Test Coverage Items, Test Cases
  - Pitfalls
- Evaluation
  - Limitations: injected faults into the code
    - Faults 4, 5 & 6
  - Strengths & Weaknesses
- Key Points & Notes for Experienced Testers

# This Afternoon

- Lab 5:
  - SC & BC: Analysis, Test Coverage Items, Test Cases, Review, Implement – should be quick
  - Deadline: next Monday evening (but try and get it done today)
- Submit via Moodle:
  - Work the problems on paper first (better than using word/excel)
- Submit for manual checking:
  - Collate into a single .pdf
  - Make sure to include BEFORE and AFTER coverage screenshots for SC and BC and the test results in each case:
    - Before SC
    - After SC but before BC
    - After BC
  - Submit your test code as a separate .java file

# Independent Study

- Read Chapter 4 (DT in more detail)
- Read Chapters 5&6 (SC & BC)
- If you're interested, read up on 'short circuit evaluation' in Java