CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

Tutorial: Lab 5

- SC and BC

Running TestNG and JaCoCo

- JAR files:
  guice.jar
  jacocoagent.jar
  jacocoant.jar
  jacococli.jar
  jcommander.jar
  testng.jar
- Without gradle, you have to find and download the correction versions of all these files
- Gradle automatically downloads either specific versions or the latest versions of these files (specified in gradle\libs.versions.toml)

Running TestNG and JaCoCo

TestNG:

```
$ javac -d bin -cp libraries-win\* cs608\Insurance.java
$ java -cp libraries-win\*;bin org.testng.TestNG
      -testclass cs608.InsuranceTest
```

Note: labs\non-gradle-examples\lab5

Running TestNG and JaCoCo

Jacoco:

```
$ del jacoco.exec
$ java –ea
      -javaagent:libraries-win/jacocoagent.jar=append=false
      -cp libraries-win/*;bin org.testng.TestNG
      -testclass cs608.InsuranceTest
      -log 2
$ java -jar libraries-win/jacococli.jar report jacoco.exec
      --html coverage
      --classfiles ./bin
      --sourcefiles .
$ start coverage\index.html
```

Labs and Versions - see: libs.versions.toml

```
[versions]
v_slf4j = "1.7.36"
v_testng = "7.10.2"
v_jacoco = "0.8.12"
v_selenium = "4.27.0"

[libraries]
slf4jApi = { group="org.slf4j", name="slf4j-api", version.ref="v_slf4j" }
testngTestng = { group="org.testng", name="testng", version.ref="v_testng" }
jacocoCore = { group="org.jacoco", name="org.jacoco.core",
               version.ref="v_jacoco" }
seleniumJava = { group="org.seleniumhq.selenium", name="selenium-java",
                 version.ref="v_selenium" }
```

CS608

White-Box Testing
Statement Coverage and Branch Coverage
In More Detail – But Briefly!

(Essentials of Software Testing, Chapters 5.5-5.8 & 6.4-6.7)

giveDiscount()

Status giveDiscount(long bonusPoints, boolean goldCustomer)

Inputs

bonusPoints: the number of bonusPoints the customer has accumulated

goldCustomer: true for a Gold Customer

Outputs

return value:

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)

Status is defined as follows:

```
enum Status { FULLPRICE, DISCOUNT, ERROR };
```

PART I – STATEMENT COVERAGE

Test Coverage Items

- Each statement in the source code is a test coverage item
  - Every **executable** statement
- Normally, just ‘consider ‘extra’ TCIs (not already covered)
- Normally, a single line of source is regarded as being a statement
- Issues:
  - Multiple statements on one line
  - Multi-line statements
- As in other forms of testing, using unique identifiers for the test coverage items makes the task of reviewing the test design easier

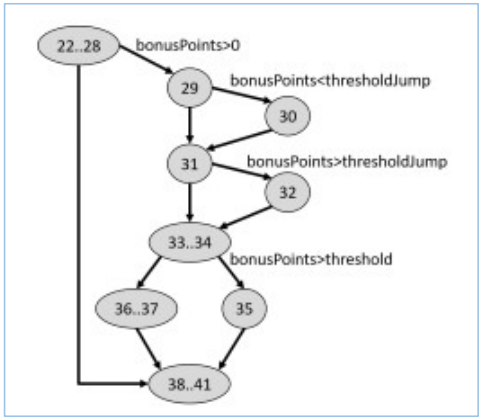
Fault Model

- The statement coverage fault model is where code that has not been executed in previous tests may contain a fault
- These unexecuted statements tend to be associated with edge cases, or other unusual circumstances
- Statement coverage tests with input values carefully selected to ensure that every statement is executed
- These tests attempt to find faults associated with individual lines in the source code

Analysis: Identifying Unexecuted Statements

- Using coverage results from previous tests (black-box tests), unexecuted statements easily identified
- For complex code, a Control-Flow Graph (CFG) may be developed first to help with understanding the code flow at a more abstract level, but these are seldom required for statement coverage
  - Quick look at CFGs later
- Statement coverage tests may be developed before black-box tests, though this is not usual practice
  - In this case, CFGs are traditionally used to assist in developing the tests
  - The experienced tester will probably not need to use them

```
22    public static Status giveDiscount(long bonusPoints, boolean
          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    }
```



Test Coverage Items

- Each statement in the source code is a test coverage item
  - Every **executable** statement
- Normally, just ‘consider ‘extra’ TCIs (not already covered)
- Normally, a single line of source is regarded as being a statement
- Issues:
  - Multiple statements on one line
  - Multi-line statements
- As in other forms of testing, using unique identifiers for the test coverage items makes the task of reviewing the test design easier

Evaluation

- SC Test Results for giveDiscount() with Fault 4 (reminder)

```
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
================================================
```

- Statement coverage has uncovered Fault 4 (not found by BBT)

```
34
35          if (bonusPoints==43) // fault 4
36              rv = DISCOUNT;
37
```

- Some benefits and limitations of Statement Coverage explored by inserting a new fault into the original code (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      }
```

Fault 5

Fault 5

```
31              if (goldCustomer && bonusPoints!=93) // fault5
32                 threshold = 80;
```

- Fault: modifying the if statement on line 31, incorrectly adding

  && bonusPoints!=93

- This creates an extra branch in the code
  - Not taken with the existing test data
  - The value 93 is never used for bonusPoints
  - When bonusPoints is equal to 93, line 32 will not be executed

SC Testing Against Fault 5

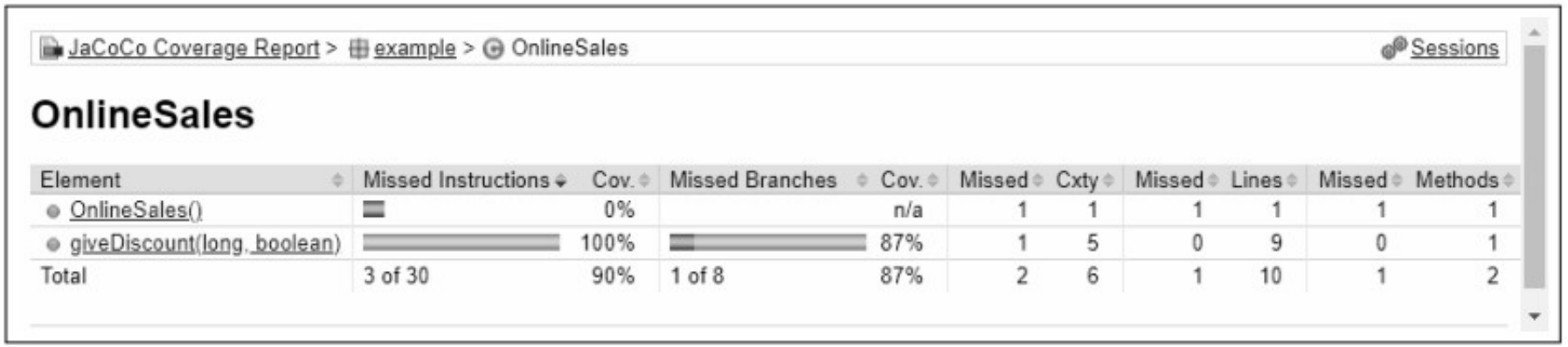
```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 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", 92233720368547758O7, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -92233720368547758O8, 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)
================================================
Command line suite
Total tests run: 15, Passes: 15, Failures: 0, Skips: 0
================================================
```

SC Testing Against Fault 5

```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, f                    SCOUNT)
PASSED: test                                                RROR)
PASSED: test                                                RICE)
PASSED: test                                                PRICE)
PASSED: test                                                PRICE)
PASSED: test                                                LPRICE)
PASSED: test                                                COUNT)
PASSED: test                                                5807, false, DISCOUNT)
PASSED: test                                                75808, false, ERROR)
PASSED: test
PASSED: test                                                OUNT)
PASSED: test                                                SCOUNT)
PASSED: test                                    43, true, FULLPRICE)
================================================
Command line suite
Total tests run: 15, Passes: 15, Failures: 0, Skips: 0
================================================
```

Fault is not found
This is expected
Fault bears no relationship to specification
Unlikely to be found by any black-box test technique
Or statement coverage

Statement Coverage of Fault 5



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%		87%	1	5	0	9	0	1
Total	3 of 30	90%	1 of 8	87%	2	6	1	10	1	2

- No missed lines in the method checkDiscount()
- No value in examining the detailed source code report

Demonstrating the Fault

```
$ check 93 true
FULLPRICE
```

- The wrong result is returned for the inputs (93,true)
- The correct result is DISCOUNT

Strengths

- Provides a minimum level of coverage by executing all the statements in the code at least once
- There is a significant risk in releasing software before every statement has been executed at least once during testing
  - its behaviour has not been verified, and may well be faulty
- Statement coverage can generally be achieved using only a small number of extra tests

Weaknesses

- Can be difficult to determine the required input parameter values
- Hard to test code only executed in unusual circumstances
- Does not provide coverage for the NULL else
  - `if (number < 3) number++;`
  - Statement coverage does not force a test case for number ≥ 3
- Not demanding of compound decisions
  - `if ( (a>1) || (b==0) ) then x = x/a;`
  - No test cases for the different boolean conditions that may cause the decision to be true or false
  - No test cases for the possible combinations of the boolean conditions

Usage

- Statement coverage is generally used to supplement black-box testing
- Mainly because it is easy to measure the coverage automatically
- If black-box testing does not result in the required coverage (normally 100%)
- Then this white-box technique can be used to increase the coverage to the required level

Key Points

- Statement coverage is used to augment black-box testing, by ensuring that every statement is executed
- Test coverage items are based on unexecuted statements
- Input values for the test cases are selected by analysis of the decisions in the code (and therefore are dependent on the specific version of the code being tested)
- Statement coverage can be used in unit testing as shown, and can also be used when testing object-oriented software in exactly the same way
- It can also be used in application testing, although for web applications, when the application is running on a remote server, setting up the server to produce coverage results and accessing those results can be challenging

Notes for Experienced Testers

- Do black-box tests first and measure their coverage
- Reviewing coverage results, and develop additional tests for full statement coverage
- Often use debugger to help work out the correct input values
  - set a breakpoint at the line of code directly before the first unexecuted line
  - examine the value of the relevant variables
- Probably develop the statement coverage test cases directly from the coverage results, without documenting the analysis or test coverage items
- Unlike in black-box testing, the test design work can be effectively reviewed by examining the coverage statistics generated by the test, without access to this documentation

PART II – BRANCH COVERAGE

Fault Model

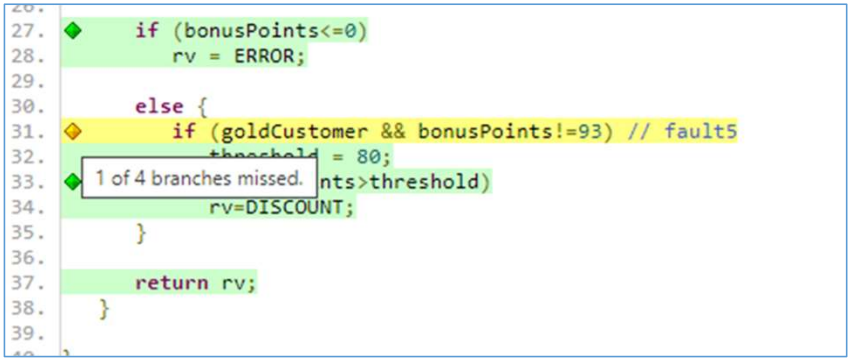
- Branches that has not been taken in previous tests (untaken branches) may contain a fault
- As for statement coverage, these tend to be associated with edge cases, or other unusual circumstances
- Branch coverage tests with input values carefully selected to ensure that every branch is taken during test execution
- These tests attempt to find faults associated with individual branches in the source code

Description

- As with many forms of testing, there is more than one approach
- We have used a tool to measure the branch coverage of the previously developed tests, and only develop new tests to complete the branch coverage
- This is the approach most usually used in practice

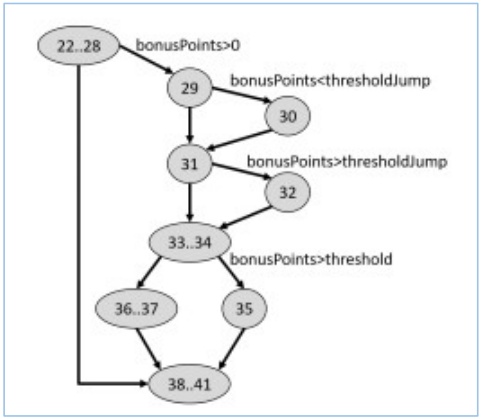
Using JaCoCo

- JaCoCo counts the outcome of each **boolean condition** as a branch
- The tester might select a different tool which counts the outcomes of each **decision** as branches instead



- This would reduce the number of branches, which leads to slightly reduced test effectiveness
- The tester must work with the available tools

“From Scratch”



- Develop a CFG – all edges are branches
- Decisions (not the boolean expressions) invariably used
- Seldom used in practice for two reasons
  1. Time consuming to develop CFG for significant code
  2. If the code is changed, either to fix a fault or add new features, the control flow graph will have to be reviewed possibly re-done. And the associated test implementation redeveloped
- The rapid change of code in a modern, Agile development environment makes this approach less realistic than tools-based/coverage measurement

Goal

- Make sure that every branch in the source code taken during testing
- Ideal test completion criteria is 100% branch coverage
- Note that a branch is based on the evaluation of a boolean expression, which can evaluate to true or false
  - A decision may be simple or compound
  - A simple decision contains a single boolean expression, or boolean condition, with no boolean operators
  - A compound decision contains multiple boolean conditions connected by boolean operators
- Let’s examine examples of each

Simple Decision

```
1    int f(int x, boolean special) {
2       int z=x;
3       if (x<0)
4          z = -1;
5       else if ( x>100 || (x>50 && special) )
6          z = 100;
7       return z;
8    }
```

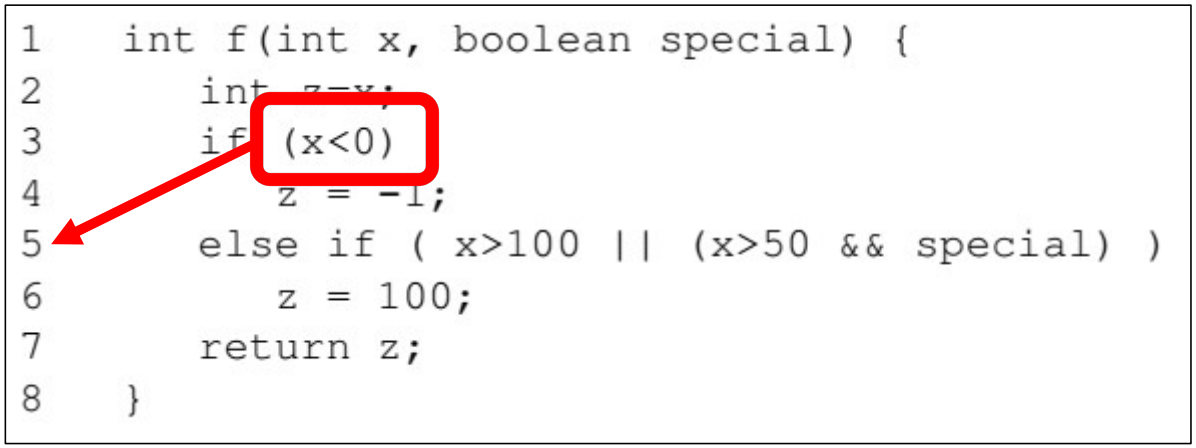
- Line 3 has a simple decision:
  - Single boolean condition “x< 0”
- Two associated branches:
  - From line 3 to line 4 when x is less than 0
  - From line 3 to line 5 when x is not less than 0

Simple Decision

```
1    int f(int x, boolean special) {
2        int z=x;
3        if (x<0)
4            z = -1;
5        else if ( x>100 || (x>50 && special) )
6            z = 100;
7        return z;
8    }
```

- Line 3 has a simple decision:
  - Single boolean condition “x< 0”
- Two associated branches:
  - **From line 3 to line 4 when x is less than 0 (decision true)**
  - From line 3 to line 5 when x is not less than 0

Simple Decision



- Line 3 has a simple decision:
  - Single boolean condition “x< 0”
- Two associated branches:
  - From line 3 to line 4 when x is less than 0
  - **From line 3 to line 5 when x is not less than 0 (decision false)**

Compound Decision

```
1    int f(int x, boolean special) {
2       int z=x;
3       if (x<0)
4          z = -1;
5       else if ( x>100 || (x>50 && special) )
6          z = 100;
7       return z;
8    }
```

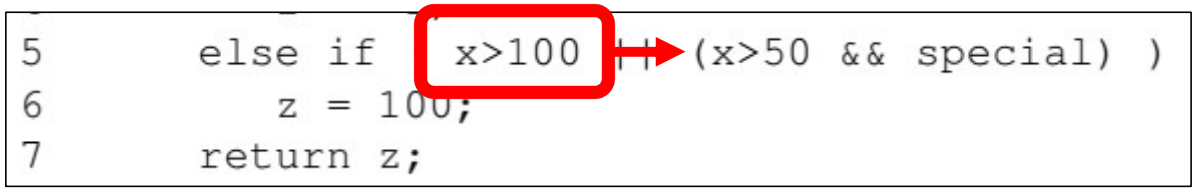
- Line 5 has a compound decision, with three boolean conditions:
  - x > 100
  - x < 50
  - special

6 Branches

- The true outcome of (x>100): branch from line 5 to line 6§

§ Short Circuit Evaluation: Short-circuit or lazy evaluation occurs when the evaluation of one boolean condition means that subsequent boolean conditions do not need to be evaluated – the result can be short-circuited

6 Branches



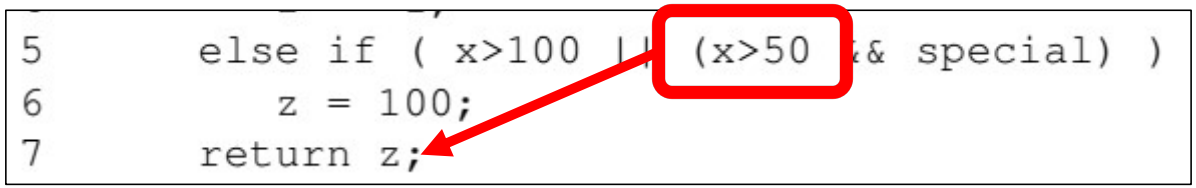
- The true outcome of (x>100): branch from line 5 to line 6[§]
- The false outcome of (x>100): branch to the next boolean condition (x>50)

6 Branches

```
5       else if ( x>100 || (x>50 && special) )
6          z = 100;
7       return z;
```

- The true outcome of (x>100): branch from line 5 to line 6§
- The false outcome of (x>100): branch to the next boolean condition (x>50)
- The true outcome of (x>50): branch to the next boolean condition (special)

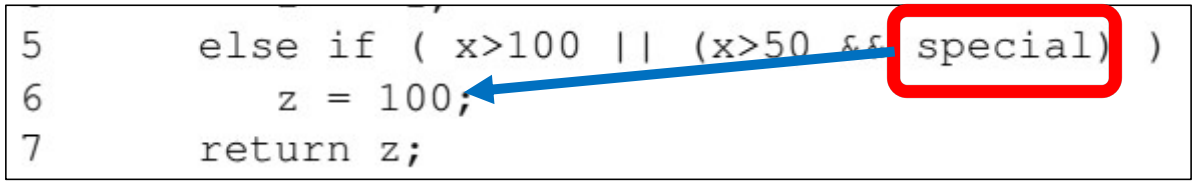
6 Branches



- The true outcome of (x>100): branch from line 5 to line 6§
- The false outcome of (x>100): branch to the next boolean condition (x>50)
- The true outcome of (x>50): branch to the next boolean condition (special)
- The false outcome of (x>50): branch from line 5 to line 7§

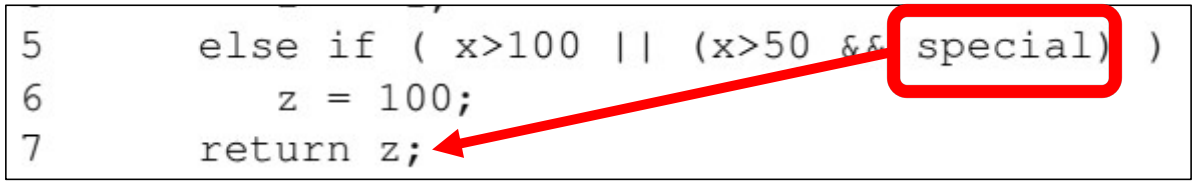
§ Short Circuit Evaluation

6 Branches



- The true outcome of (x>100): branch from line 5 to line 6§
- The false outcome of (x>100): branch to the next boolean condition (x>50)
- The true outcome of (x>50): branch to the next boolean condition (special)
- The false outcome of (x>50): branch from line 5 to line 7§
- The true outcome of (special): branch from line 5 to line 6

6 Branches



- The true outcome of (x>100): branch from line 5 to line 6§
- The false outcome of (x>100): branch to the next boolean condition (x>50)
- The true outcome of (x>50): branch to the next boolean condition (special)
- The false outcome of (x>50): branch from line 5 to line 7§
- The true outcome of (special): branch from line 5 to line 6
- The false outcome of (special): branch from line 5 to line 7

Short-Circuit Evaluation: Java Bytecode

```
30          else {
31              if (goldCustomer && bonusPoints!=93) // fault5
32                  threshold = 80;
33              if (bonusPoints>threshold)
34                  rv=DISCOUNT;
35          }
```

- Use javap –c –l to disassemble the .class file

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

LineNumberTable:

…
line 31: 22
line 32: 34
line 33: 39
…

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
LineNumberTable:
  …
  line 31: 22
  line 32: 34
  line 33: 39
  …
```

LocalVariableTable:

Start	Length	**Slot**	Name	Signature
0	52	**0**	bonusPoints	J
0	52	**2**	goldCustomer	Z
4	48	**3**	rv	Lexample/OnlineSales$Status;
9	43	**4**	threshold	J

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32              threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
public static example.OnlineSales$Status giveDiscount(long, boolean);
Code:
   22: iload_2          load slot 2: goldCustomer
   23: ifeq     39      line 33 - if goldCustomer==0 (false) jump to line 33
   26: lload_0          load slot 0: bonusPoints
   27: ldc2_w   #18     // long 93L
   30: lcmp             compare
   31: ifeq     39      line 33 - if bonusPoints!=93 jump to line 33
   34: ldc2_w   #20     // long 80L
   37: lstore    4
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
Pseudocode:
line 31:
   if (!goldCustomer) goto line 33
   if bonusPoints==93 goto line 33 : double negative !(bonusPoints!=93)
   threshold = 80
line 33:
   etc…
```

Short-Circuit Evaluation: Java Bytecode

```
31          if (goldCustomer && bonusPoints!=93) // fault5
32             threshold = 80;
33          if (bonusPoints>threshold)
```

```
Pseudocode:
line 31:
   if (!goldCustomer) goto line 33
   if bonusPoints==93 goto line 33 : double negative !(bonusPoints!=93)
   threshold = 80
line 33:
```

Most languages do this: quicker than evaluating all the conditions and using logical operators
In java single **&** forces all conditions to be evaluated

Forced Full Evaluation using Single &

- Viewing class Single
- Using javac to compile the Java source code
- Using javap –c –l to disassemble the class file
- Viewing the disassembled class
- Reading the line number table to find the bytecode
- Using the "**iand**" bytecode to perform a logical AND

Single.java

```
public class Single {
   public static boolean check(int a, int b) {
      if ((a==100)&(b==200))
         return true;
      return false;
   }
}
```

Single.bc

```
public class Single {
   public static boolean check(int a,
                               int b) {
      if ((a==100)&(b==200))
         return true;
      return false;
   }
}
```

```
 0: iload_0
 1: bipush        100
 3: if_icmpne     10
 6: iconst_1
 7: goto          11
10: iconst_0
11: iload_1
12: sipush        200
15: if_icmpne     22
18: iconst_1
19: goto          23
22: iconst_0
23: iand
24: ifeq          29
27: iconst_1
28: ireturn
29: iconst_0
30: ireturn
```

Single.bc

```
public class Single {
   public static boolean check(int a,
                               int b) {
      if ((a==100)&(b==200))
         return true;
      return false;
   }
}
```

```
 0: iload_0
 1: bipush        100
 3: if_icmpne     10
 6: iconst_1      true
 7: goto          11
10: iconst_0      false
11: iload_1
12: sipush        200
15: if_icmpne     22
18: iconst_1
19: goto          23
22: iconst_0
23: iand
24: ifeq          29
27: iconst_1
28: ireturn
29: iconst_0
30: ireturn
```

Single.bc

```
public class Single {
   public static boolean check(int a,
                               int b) {
      if ((a==100)&(b==200))
         return true;
      return false;
   }
}
```

```
 0: iload_0
 1: bipush        100
 3: if_icmpne     10
 6: iconst_1
 7: goto          11
10: iconst_0
11: iload_1
12: sipush        200
15: if_icmpne     22
18: iconst_1      true
19: goto          23
22: iconst_0      false
23: iand
24: ifeq          29
27: iconst_1
28: ireturn
29: iconst_0
30: ireturn
```

Single.bc

```
public class Single {
   public static boolean check(int a,
                               int b) {
      if ((a==100)&(b==200))
         return true;
      return false;
   }
}
```

```
 0: iload_0
 1: bipush        100
 3: if_icmpne     10
 6: iconst_1
 7: goto          11
10: iconst_0
11: iload_1
12: sipush        200
15: if_icmpne     22
18: iconst_1
19: goto          23
22: iconst_0
23: iand          AND
24: ifeq          29
27: iconst_1
28: ireturn
29: iconst_0
30: ireturn
```

Evaluation of Branch Coverage

Evaluation

```
31              if (goldCustomer && bonusPoints!=93) // fault5
32                 threshold = 80;
```

- Branch coverage has uncovered Fault 5 inserted into giveDiscount()
- Some benefits and limitations of branch coverage are now explored by injecting faults into the original (correct) source code

```
22    public static Status giveDiscount(long bonusPoints, boolean
          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    }
```

Fault 6

```
22    public static Status giveDiscount(long bonusPoints, boolean
          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    }
```

The entire processing of the method is rewritten
Lines 24 to 38
This creates a path through the code that is not taken with any of the existing branch coverage test data

Fault 6

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", 92233720368547758O7, false, DISCOUNT)
PASSED: test_giveDiscount("T2.7", -92233720368547758O8, 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
===============================================
```

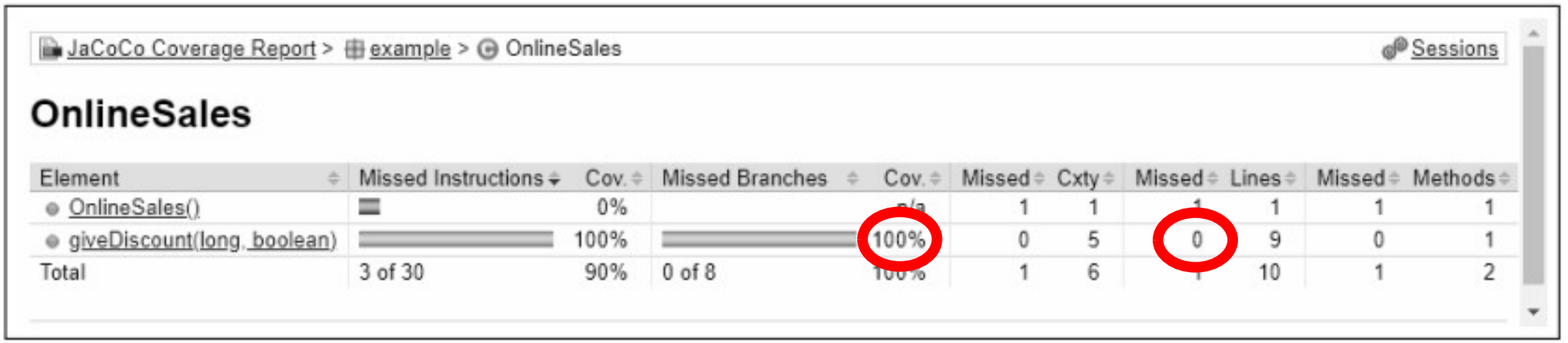
EP+BVA+DT+SC+BC Tes

```
PASSED: test_giveDiscount("T1.1", 40
PASSED: test_giveDiscount("T1.2", 10
PASSED: test_giveDiscount("T1.3", 20
PASSED: test_giveDiscount("T1.4", -1
PASSED: test_giveDiscount("T2.1", 1,
PASSED: test_giveDiscount("T2.2", 80
PASSED: test_giveDiscount("T2.3", 81
PASSED: test_giveDiscount("T2.4", 1
PASSED: test_giveDiscount("T2.5", 1
PASSED: test_giveDiscount("T2.6", 9
PASSED: test_giveDiscount("T2.7", -
PASSED: test_giveDiscount("T2.8", 0
PASSED: test_giveDiscount("T3.1",
PASSED: test_giveDiscount("T3.2",
PASSED: test_giveDiscount("T4.1",
PASSED: test_giveDiscount("T5.1",
===================================
Command line suite
Total tests run: 16, Passes: 16, Failures: 0, Skips: 0
===================================================
```

All the tests have passed
The fault has not been found
Expected:
(a) the inserted fault bears no relationship to the specification and is unlikely to be found by any black-box test technique, and
(b) the fault is not revealed by achieving either statement coverage or branch coverage of the code

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

Demonstrating the Fault

```
$ 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

Strengths and Weaknesses

- Strengths:
  - Branch coverage is a **stronger** form of testing than statement coverage: 100% branch coverage guarantees 100% statement coverage – but the test data is harder to generate
  - Resolves the NULL else problem
- Weaknesses:
  - Can be difficult to determine the required input parameter values
  - If the tool only counts decisions as branches, or if a control flow graph has been manually developed, then it is undemanding of compound decisions. In these cases it does not explore all the different reasons (i.e. the boolean conditions) for the decision evaluating as true or false

Usage

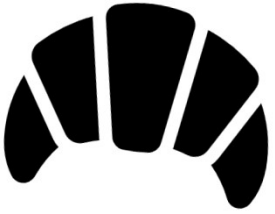
- Branch coverage, like statement coverage, is usually used as a supplementary measure of black-box testing – mainly because it is easy to measure automatically
- If black-box testing does not result in the required coverage (normally 100%) then this technique can be used to increase the coverage to the required level

Key Points

- Branch coverage is used to augment black-box and statement coverage testing, by ensuring that every branch is taken
- Test cases are based on untaken branches
- Input values for the test data are selected by analysis of the decisions/Boolean conditions in the code

Notes for Experienced Testers

- Use coverage measured by tools (decision-level or boolean condition level)
- Code analysis for data values usually be done in the tester's mind
- Perhaps add a comment to the test code for branches that cannot be taken
- Hard to review without audit trail
- Advanced testing, exception coverage as a form of branch coverage
  - Each exception raised and caught is regarded as a branch
- As with all white-box testing, tests that achieve full branch coverage will often become outdated by changes to the code. The experienced tester may, however, leave these tests in the code as extra tests
- Developing tests to achieve full branch coverage is practically impossible in code of any significant size. Focus these tests on critical code only



CHANGE OF TOPIC COMING UP
TAKE A BREAK