CS608
Software Testing

Dr. Stephen Brown

Room Eolas 116

stephen.brown@mu.ie

Tutorial: Lab 10

- Random Testing
    1. Develop automated random tests for giveDiscount() using DT testing as a basis
    2. Essentially, randomise (where possible) the DT test data
    3. Using the ‘causes’ as criteria

CS608

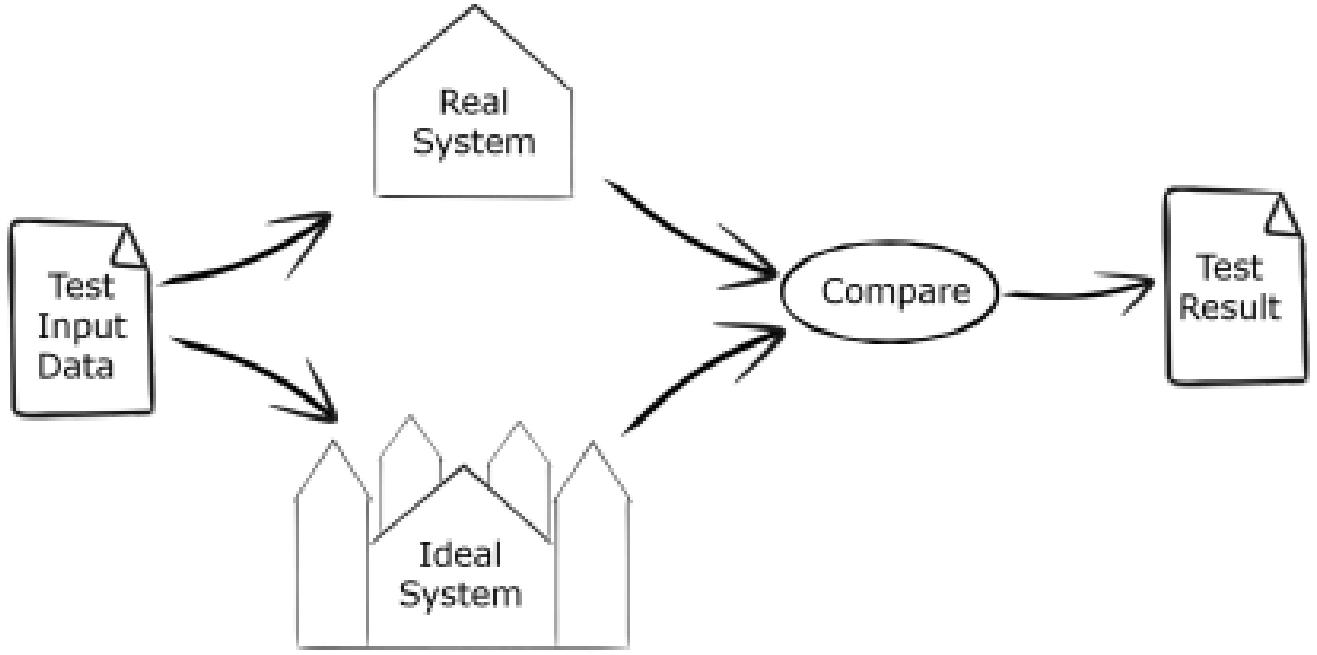
Black-Box and White-Box Testing

(Essentials of Software Testing, Chapter 8)

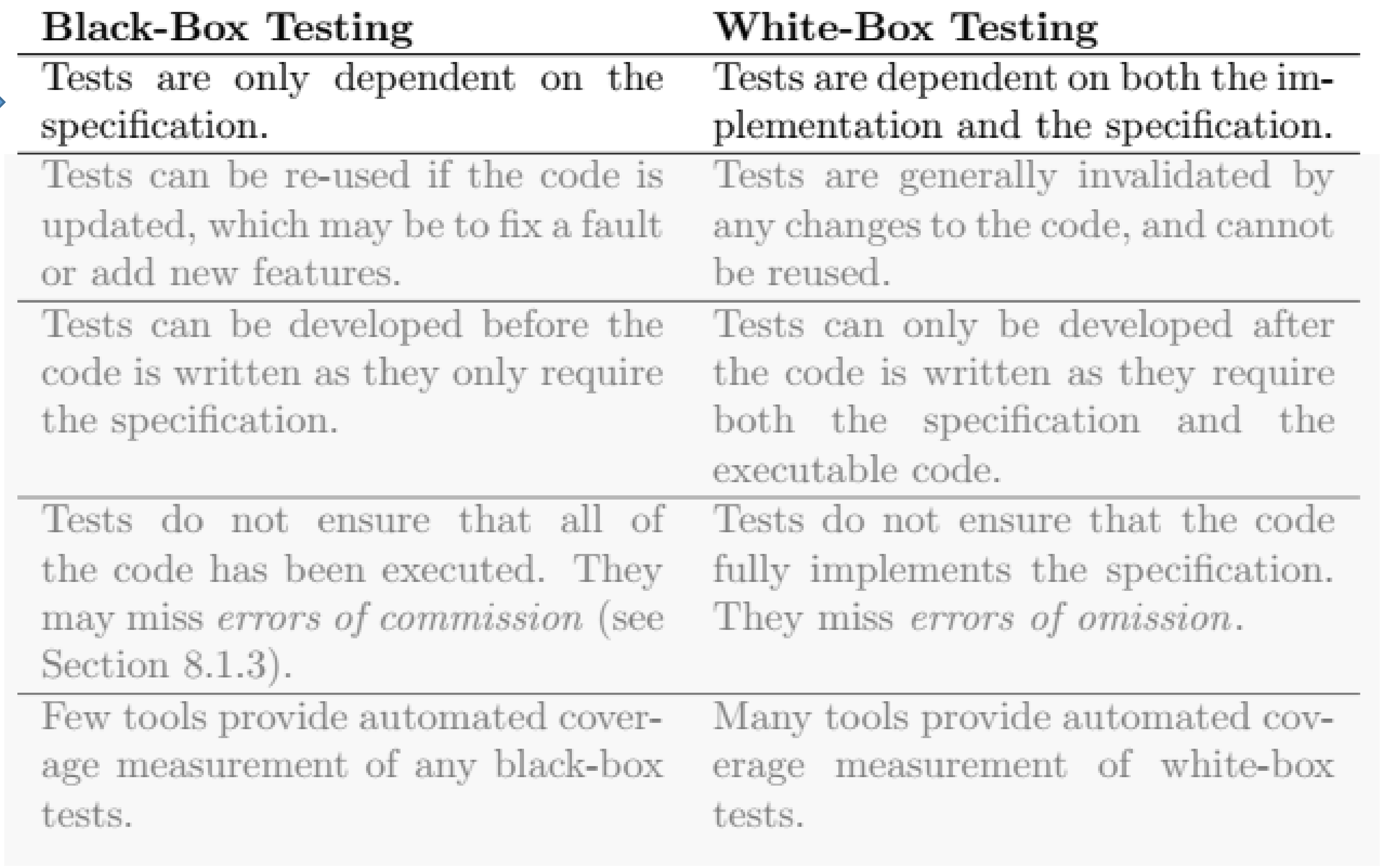
Black-Box and White-Box Testing

- Dynamic testing confirms the correct operation of a program ("test item") by executing it

- Test process modelled as comparison of the outputs of a real system with those of an ideal system
  - Ideal system: specification
  - Real system: test item

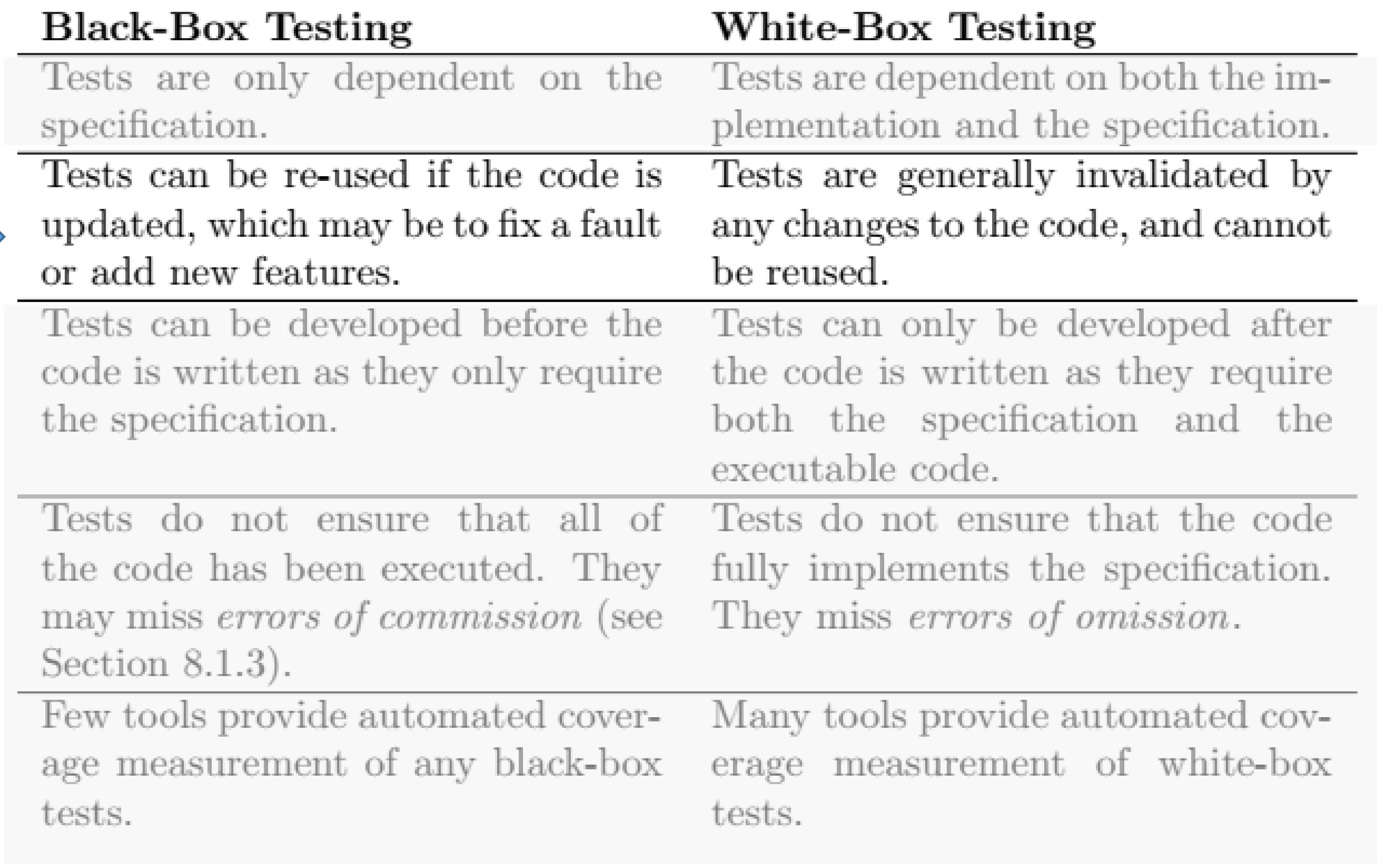


Comparison



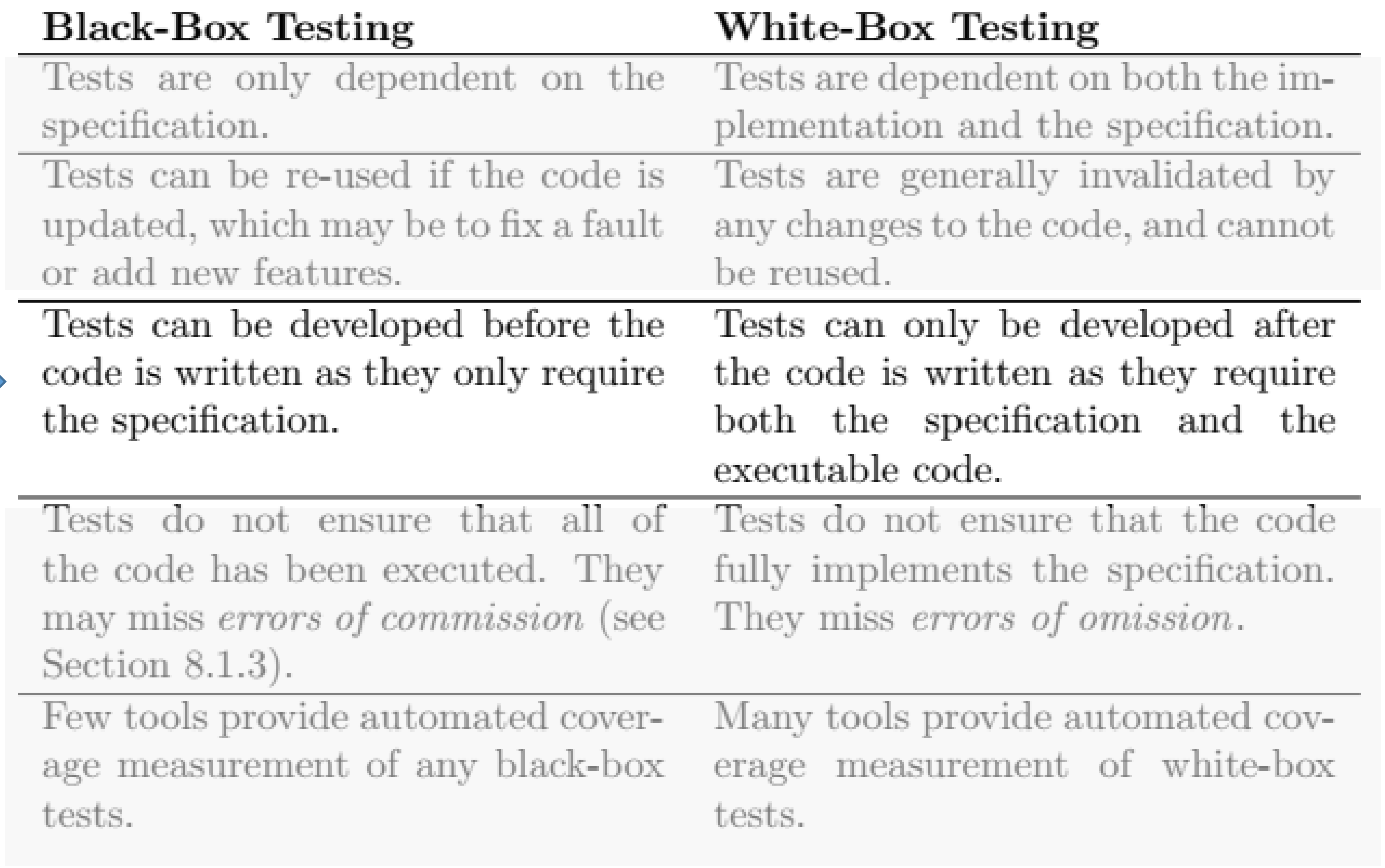
Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Comparison



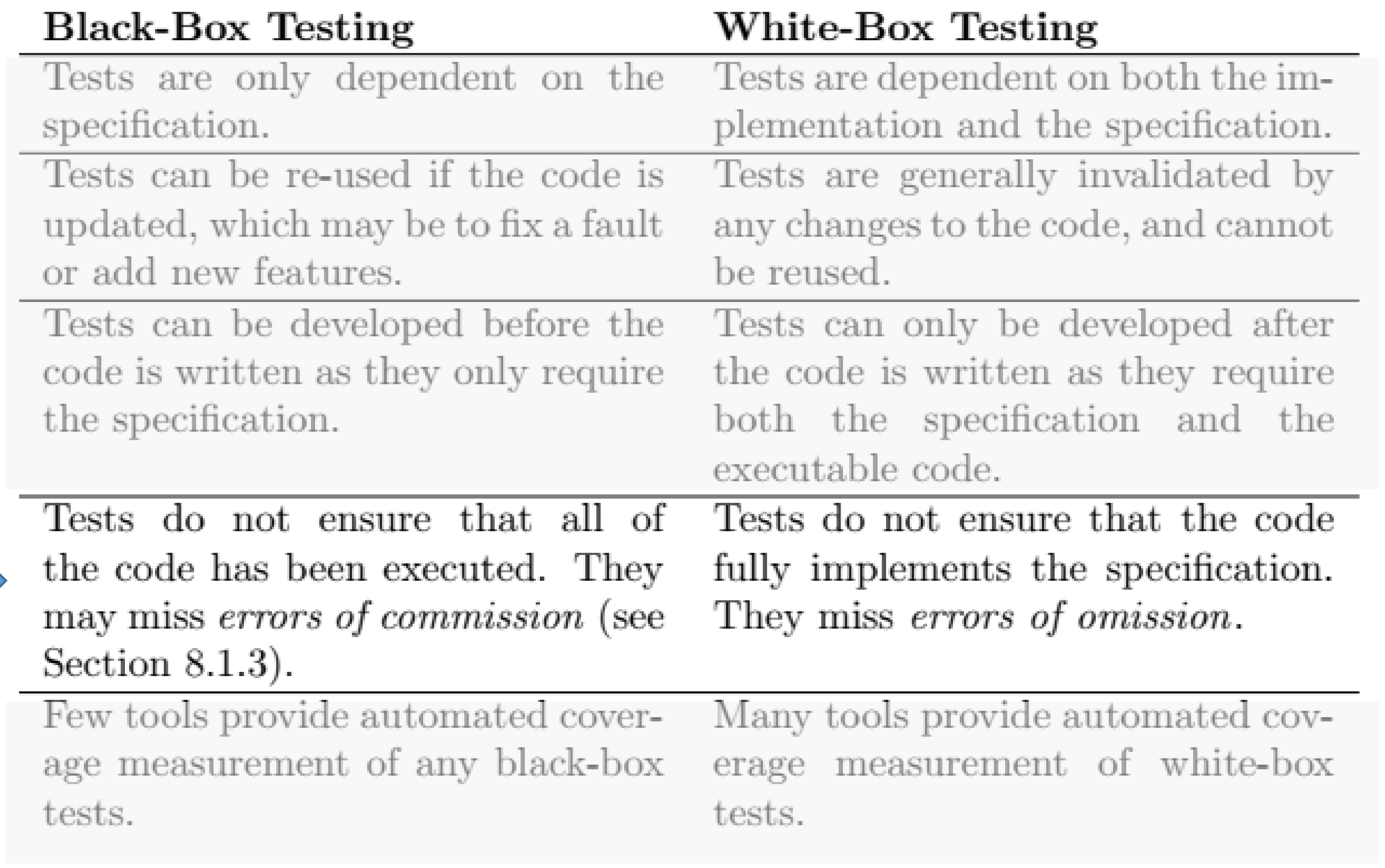
Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Comparison



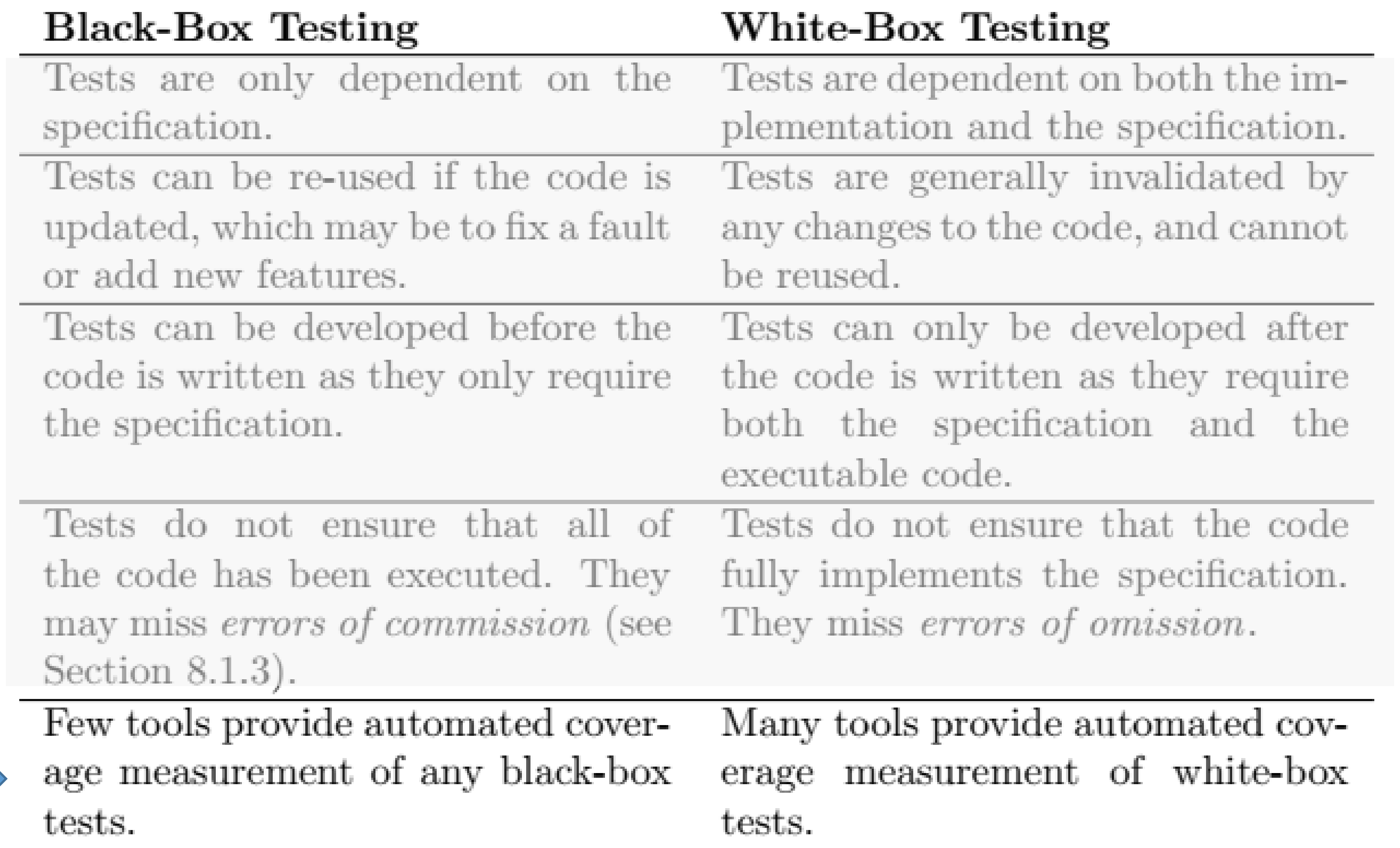
Black-Box Testing	**White-Box Testing**
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Comparison



Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Comparison

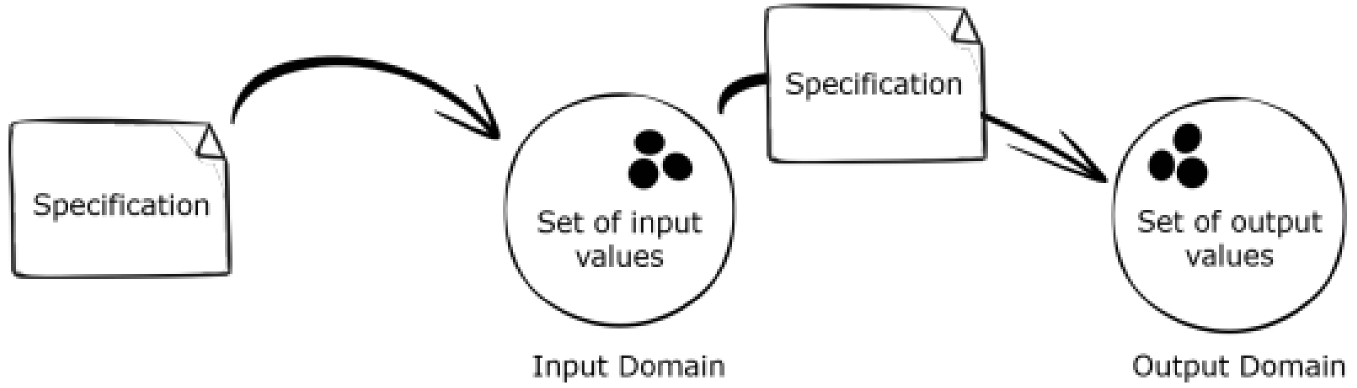


Black-Box Testing	White-Box Testing
Tests are only dependent on the specification.	Tests are dependent on both the implementation and the specification.
Tests can be re-used if the code is updated, which may be to fix a fault or add new features.	Tests are generally invalidated by any changes to the code, and cannot be reused.
Tests can be developed before the code is written as they only require the specification.	Tests can only be developed after the code is written as they require both the specification and the executable code.
Tests do not ensure that all of the code has been executed. They may miss *errors of commission* (see Section 8.1.3).	Tests do not ensure that the code fully implements the specification. They miss *errors of omission*.
Few tools provide automated coverage measurement of any black-box tests.	Many tools provide automated coverage measurement of white-box tests.

Black-Box Testing (BBT)

- Specification used for test cases:
    - Testing against the specification
    - Using test coverage criteria based on the specification
    - Developing test cases derived from the specification
    - Exercising the specification
- Measurement:
    - Difficult to automatically measure the degree of coverage of the specification that black-box testing has achieved
    - Relies heavily on the quality of the tester’s work

Black-Box Testing (BBT)

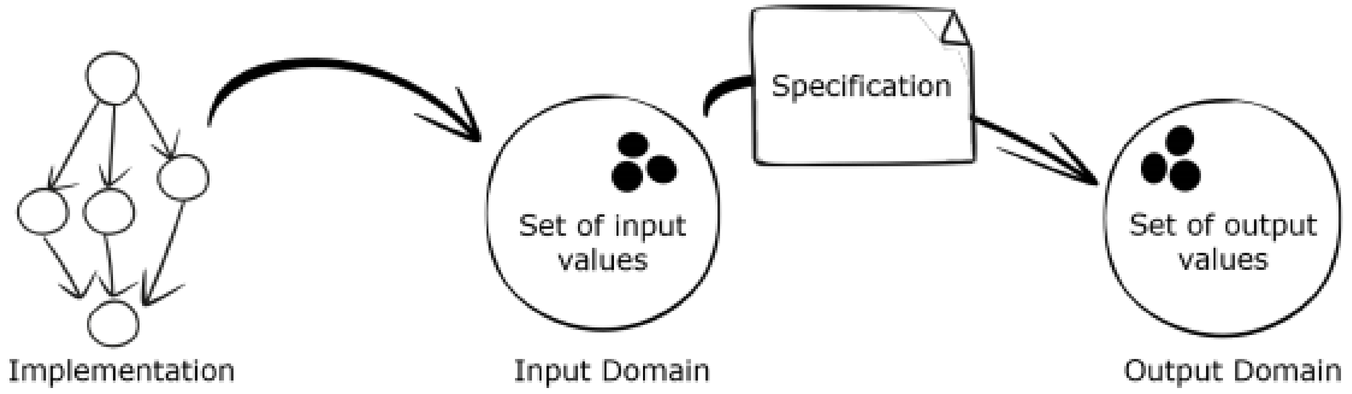


- Software is a mapping from input values to output values
- Purpose of black-box testing: ensure that every value in the input domain maps to the correct value in the output domain
- This is ideally achieved by exercising all the mappings in the specification
- Exhaustive testing is seldom feasible, a subset of possible inputs selected to cover key mappings between the input and output domains

White-Box Testing (WBT)

- Both specification and implementation used for test cases:
  - Testing against the implementation
  - Using test coverage criteria based on the implementation
  - Developing test cases derived from the implementation
  - Exercising the implementation
- Measurement:
  - Tools measure what was executed in the code during testing
  - Recording which instructions have been executed ("instrumentation")
  - Coverage expressed as a percentage: e.g. 100% coverage
  - Less than 100% indicates untested code
  - 100% may not be achievable or worth the effort: concentrate on critical code?

White-Box Testing (WBT)



- Software is set of components that create output values from input values
- Purpose of white-box testing: ensure that executing the components (e.g. statements, branches, etc.) always results in the *correct* output values
- The specification is still needed to ensure that the output values are correct
- Exhaustive testing of the components is seldom feasible (e.g. all paths), subset of possible inputs selected to exercise components

Errors of Omission & Commission

- BBT provides coverage of specification not implementation
  - There may be code in the implementation that incorrectly produces results not stated in the specification
  - Does not typically find extra functionality (errors of commission)
- WBT provides coverage of implementation not specification
  - There may be behaviour stated in the specification for which there is no code in the implementation
  - Does not typically find missing functionality (errors of omission)
- It is for these reasons that black-box testing is done first, to make sure the code is complete. This is then augmented by white-box testing, to make sure the code does not add anything extra.

Usage

- BBT and WBT generally used in succession to maximise coverage
- The degree of coverage is usually based on the quality requirements of the test item. For example, software that decides whether to sound an alarm for a hospital patient will have higher quality requirements than software that recommends movies to watch
- In order:
  - Basic BBT
  - Augment with extra BBT
  - Basic WBT
  - Augment with extra WBT
- And use repair-based testing after fixing software faults

We have Covered Basic BBT

- **EP**: Use equivalence partitions to verify the basic operation of the software by ensuring that one representative of each partition is executed
- **BVA**: f the specification contains boundary values, use boundary value analysis to verify correct operation at the edges
- **DT**: If the specification states different processing for different combinations of inputs, use decision tables to verify correct behaviour for each combination

Extra BBT

- **STATE**: If the specification contains state-based behaviour, or different behaviour for different sequences of inputs, then use state-based/sequential testing to verify this behaviour
- **EXPERT OPINION**: If there are reasons to suspect that there are faults in the code, perhaps based on past experience, then use error guessing/expert opinion to try and expose them
- **RANDOM**: If the typical usage of the software is known, or to achieve a large number of tests, then use random test data to verify the correct operation under these usage patterns

We have Covered Basic WBT

- **Measure** the statement and branch coverage for black-box tests
  - If required coverage not achieved, add white-box testing
  - To enhance quality of testing
- **SC**: Use statement coverage to ensure that 100% of the statements have been executed
- **BC**: Use branch coverage to ensure that 100% of the branches have been taken

Extra WBT

- **DU-Pairs**: if code contains complex data usage patterns, then use du pair testing
  - Described later in this lecture
- **CC, DC, DCC, MCC, MCDC**: If code contains complex decisions, then use these forms of testing to ensure coverage of these
- **AP**: If code contains complex end-to-end paths, use all paths testing

BBT: Additional Matters

- Strings and Arrays
- Discontinuous input partitions and overlapping output partitions
- Inband error reporting
- Handling relative values
- Classic triangle problem
- Testing sequences of inputs (state-based testing)
- Floating point
- Numeric processing

Strings and Arrays

- Complex to handle
  - they contain a large number of data values
  - may have no specific length limit
- There is no standard approach for handling these
- Guidelines on applying black-box techniques to these data structures
- First we consider error cases, and then normal cases

General Categories of Error Cases

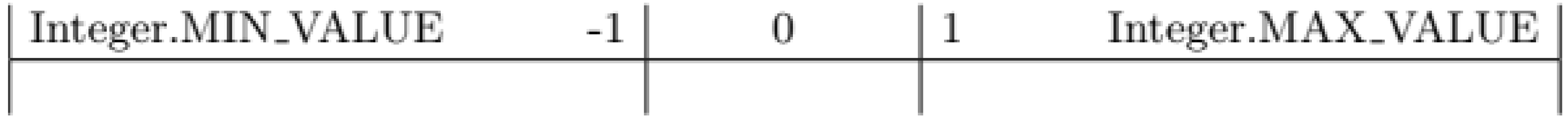
- A **null** reference
- A string or array of **length 0**
- A string or array which is **too long** by exceeding its specified maximum length (e.g. 13 digits for a modern ISBN number)
- **Invalid data:** a String may be specified to only include printable characters; an integer array may be specified to contain only positive values
- **Invalid relationships:** a String may be specified to only contain characters in alphabetical order; an integer array may be specified to be in descending order

Normal Cases

- More difficult to categorise
  - **Multiple** EP values
  - Boundary values of both the **length** of the data structure and of the **data**
  - Multiple tests per combination, with multiple values for each cause
- Test data complexity depends on complexity of the specification
  - Select one data set that includes the same value in each location in the String or array
  - For Strings representing contact information (such as names, addresses, or phone numbers) one might use a telephone directory to find short, long, and of typical values
  - For arrays representing ints, such as a list of numbers to be subject to statistical tests, one might select a few small data sets and one large data sets with known characteristics
- Important to ensure that the output cases are all covered: these often provide additional guidance on selecting input values

Discontinuous Input Partitions

- boolean isZero(int x)

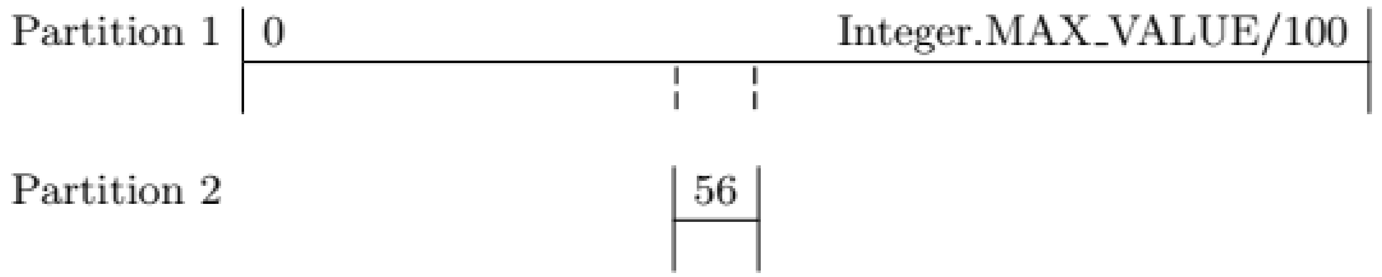


- One input partition with the single value 0
- Another input partition with all the values except 0
- Treat as three partitions (-1 and +1 are special: boundary values):

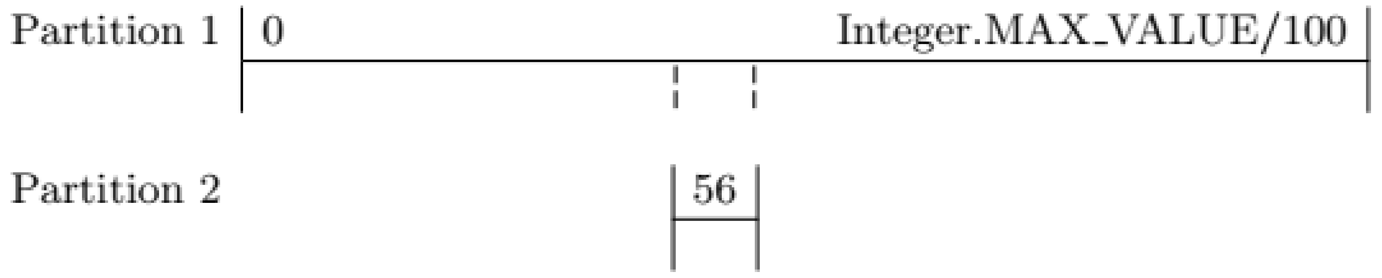
x.EP 1	Integer.MIN_VALUE..-1
x.EP 2	0
x.EP 3	1..Integer.MAX_VALUE

Overlapping Output Partitions

- int tax(int x, boolean fixed):
  - -1 indicating an error, if amount < 0
  - 56, if fixed is true (fixed tax)
  - x=100, which is 1% of the amount if fixed is false (variable tax)
- Non-error partitions:



Overlapping Output Partitions (contd)



- Treat as two independent partitions

x.EP 1 0..Integer.MAX_VALUE/100

x.EP 2 56

- The values 55 and 57 are not special: they are not values at the boundary of a partition, and are not likely to be associated with faults in the software

Inband Error Reporting

- Errors reported via the same mechanism as for normal processing
- Sometimes even using the same values
- Typical examples:
    - boolean value false
    - integer value -1
- The same value can be used both to indicate that some requirement is not met and also to indicate errors
- The alternative, raising an exception, is much easier to test
    - but the tester must be able to test both types of error reporting

Inband Error Reporting Example

- boolean inRange(int x, int low, int high):
  - true, representing that x is in range
  - false, representing that x is not in range
  - false, representing that the inputs are invalid
- Two non-error cases, and one error case, for the return value:
  - true (non-error)
  - false (non-error) – not in range
  - false (error) – invalid inputs
- Need to test for both causes of a false output

Handling relative values

```
int largest(int x, int y)
```

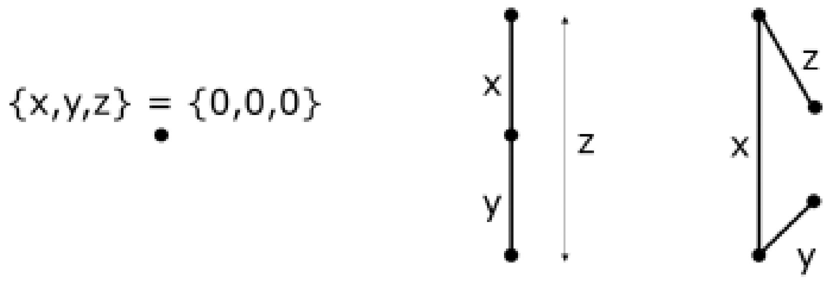
- Cases for x (treat y as fixed):
  - x<y
  - x==y
  - x>y
- Partitions for x:
  - Integer.MIN_VALUE..y-1
  - y
  - y+1..Integer.MAX_VALUE

- Cases for y (treat x as fixed):
  - y<x
  - y==x
  - y>x
- Partitions for y:
  - Integer.MIN_VALUE..x-1
  - x
  - x+1..Integer.MAX_VALUE

Classic triangle problem

- String classify(int x, int y, int z):
  - Equilateral, Isosceles, Scalene or Invalid

- Invalid triangles:

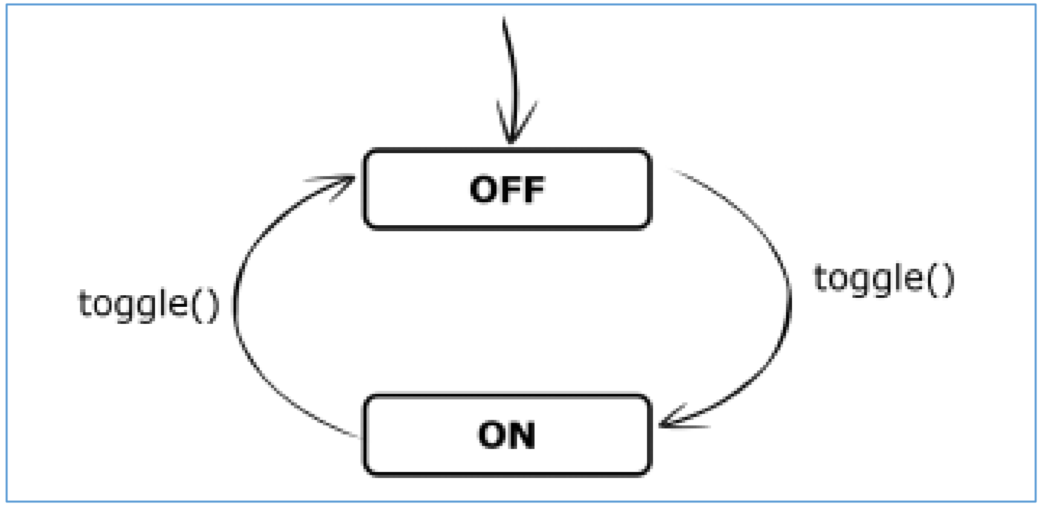


- Partitions for x

x.EP 1	Integer.MIN_VALUE..0
x.EP 2	1..y-1
x.EP 3	y
x.EP 4	y+1..y+z-1
x.EP 5	1..z-1
x.EP 6	z
x.EP 7	z+1..y+z-1
x.EP 8	y+z..Integer.MAX_VALUE

- Similar for y and z

Testing sequences of inputs (state-based testing)



- Lamp.toggle()
  - When off, a touch toggles the lamp to on
  - When on, a touch toggles the lamp to off
- Two states: ON and OFF
- State-based testing raises the events that should cause the software to transition between states by calling the methods on the state diagram
- Tests check (a) that the correct state transitions have taken place, and (b) that the correct actions have occurred

State-Based Testing Strategies

- **Piecewise** (test some transitions)
- **All Transitions** (test every transition at least once)
- **All Round-trip Paths** (test all paths starting and ending at the same state)
- **M-length Signatures** (sequences of M events with unique responses)
- **All n-event transition sequences** (test all sequences of transitions for a particular value of n).
- **Exhaustive Testing** (testing all possible sequences of transitions)

How to Test Transitions

- Verify that the software is in the required starting state for the test
- Set values, if required, to ensure that the transition guard is true
- Raise the event (call a function or method) that should cause the transition
- Check that the software is now in the expected end state
- Check that any required actions have occurred

Floating Point

- Floating point is significantly more difficult to handle than int
- IEEE 754 – sign bit – no wrapping
- Floating point best regarded as an approximation:
  - If a small value is added to a large value it may have no effect
  - The difference between two large, unequal values may be returned as 0.0
  - Many fractions and decimals (such as 1/3, or 0.1) cannot be represented exactly in floating point
  - Example: printing 0.1 to 17 decimal places
    returns 0.10000000149011612
    and not 0.10000000000000000 as you might expect

FP Issues

- **Comparing to a Constant**
  - Compare to a range of values, representing the minimum and maximum allowable values (e.g. TestNG)
  - Answer must be correct to 6 decimal places, compare to expected value ±0:0000005
  - Need an **accuracy margin** to test properly
- **Boundary Values**
  - Java.lang.Math.nextUp() and Java.lang.Math.nextAfter()
- **Handling Cumulative Errors**
  - Approximation means errors accumulate in code. Need to know allowable error. And compare within a range
- **Input and Output Conversion**
  - It is easy to make a mistake when reading or writing floating point numbers
  - For example, the visible format of the number may change depending on its value (0.000001 vs 1.0000E-6)
  - Test input and output with very large and very small floating point numbers to make sure they are handled correctly
- **Special Values**
  - Floating point has a number of special values: NaN (not a number), POSITIVE INFINITY (Infinity), and NEGATIVE INFINITY (-Infinity)
  - These are usually error values, and should be included as extra input partitions

Numeric processing

- The tests we have considered focus on **logic** processing: they do not produce a **calculated** result
- For numeric processing, which returns the result of a numeric calculation:
    - EP and BVA input partitions should be handled the same way
    - Output partitions require more analysis if not all outputs are possible
    - If any output is possible, back-calculate required input values for:
        - Output EP values (example: Integer.MIN_VALUE/2 and Integer.MAX_VALUE/2)
        - Output BVA values (example: Integer.MIN_VALUE, -1, 0, 1, Integer.MAX_VALUE)

Example

- int add10(int x)
  - adds 10 to the input value if it is between 0 and 90
  - raises an IllegalArgumentException otherwise
- Input partitions

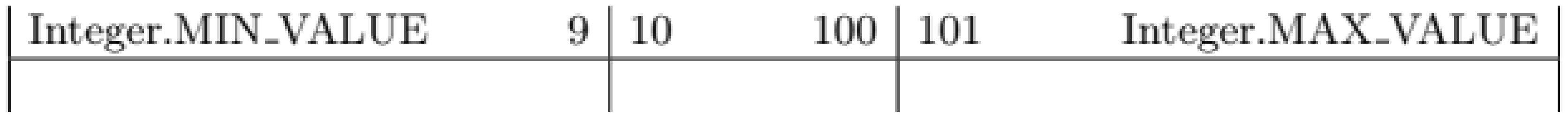
Integer.MIN_VALUE -1	0 90	91 Integer.MAX_VALUE

- Output partitions (require calculations)

Integer.MIN_VALUE 9	10 100	101 Integer.MAX_VALUE

Example (contd)

- Note additional output: IllegalArgumentException (input x not in [0..90])
- Return value analysis:



- Range [Integer..MIN VALUE..9] is **not possible** – there is no input value that produces an output in this range
- The value 10 is produced as an output for input x=0
- The value 100 is produced as an output for input x=90
- A good EP value would be x=45
- The range [101..Integer.MAX VALUE] is **not possible** – there is no input value that produces an output in this range

WBT: Some More Techniques

- Condition Coverage (CC)
- Decision Coverage (DC)
- Decision/Condition Coverage (DCC)
- Multiple Condition Coverage (MCC)
- Modified Condition/Decision Coverage (MCDC)
- Dataflow coverage/Definition-Use pairs (D-UP)
- All-Paths Testing (AP)
- Test ranking

Condition Coverage (CC)

- Each boolean condition within a decision is tested for its true and false values
- TCI for “if (a || (b && c))”
  - a and !a
  - b and !b
  - c and !c
- Does not ensure decision is true and false
- Can be difficult to determine the required input parameter values

Decision Coverage (DC)

- In decision coverage (often abbreviated to DC), every decision is evaluated to true and false
- This is equivalent to branch coverage if a coverage measurement tool is used that identifies decisions as branches (or if a control flow graph has been used)

Decision/Condition Coverage (DCC)

- In decision condition coverage (often abbreviated to DCC) , tests are generated that both cause every decision to be taken at least once (decision coverage), and also every boolean condition to be true and false at least once (condition coverage)
- TCIs
  - Each value of each conditions (true and false)
  - Each value of each decision (true and false)
- Doesn’t test every combination of conditions
- Can be difficult to determine required input parameter values

Multiple Condition Coverage (MCC)

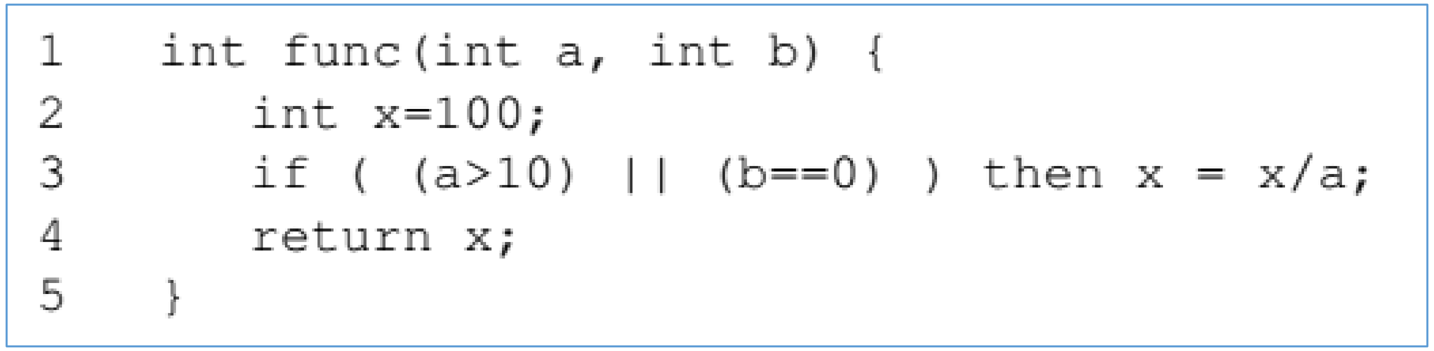
- Tests are generated to cause every possible combination of boolean conditions for every decision to be tested
- Use a Decision Table
- Each decision with n independent boolean conditions has 2^n test coverage items
- A large number of TCIs
- Can be difficult to determine required input parameter values

Modified Condition/Decision Coverage (MCDC)

- MCC generates a very large number of tests
- Reduced by only considering those combinations of Boolean conditions that cause a discernible effect on the output
- The test conditions:
  - decision condition coverage, plus
  - additional conditions to verify the independent effect of each boolean condition on the output
- Each decision has two test coverage items (true and false)
- Every boolean condition in each decision has two test coverage items
- In addition, test coverage items must be created that show the effect on the output of changing each of the boolean conditions independently

NASA: *A Practical Tutorial on Modified Condition/Decision Coverage* (Hayhurst)

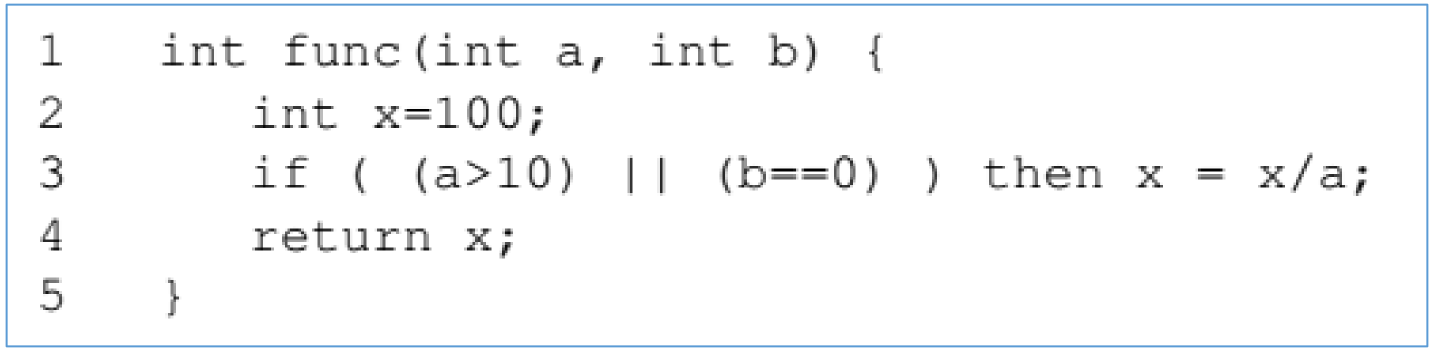
MCDC Example



```
int func(int a, int b) {
   int x=100;
   if ( (a>10) || (b==0) ) then x = x/a;
   return x;
}
```

- func() returns:
  - 100/a if a>10 or b==0
  - 100 if a<=10 and b!=0

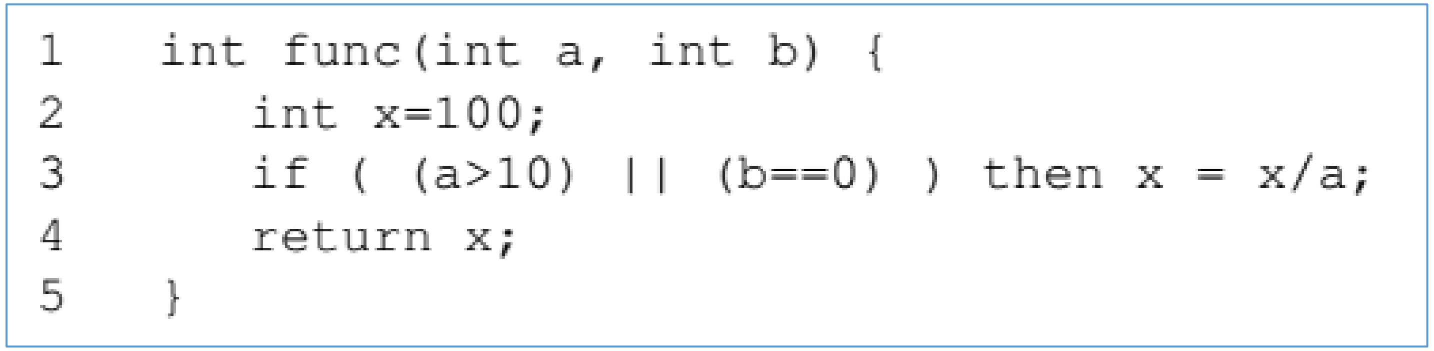
MCDC Example



```
int func(int a, int b) {
    int x=100;
    if ( (a>10) || (b==0) ) then x = x/a;
    return x;
}
```

- Tests must ensure each condition true and false:
  - a>10
  - b==0
- Tests must ensure each decision true and false:
  - (a>10) || (b==0)
- Tests must ensure the effect on the output value of changing the value of every boolean condition is shown

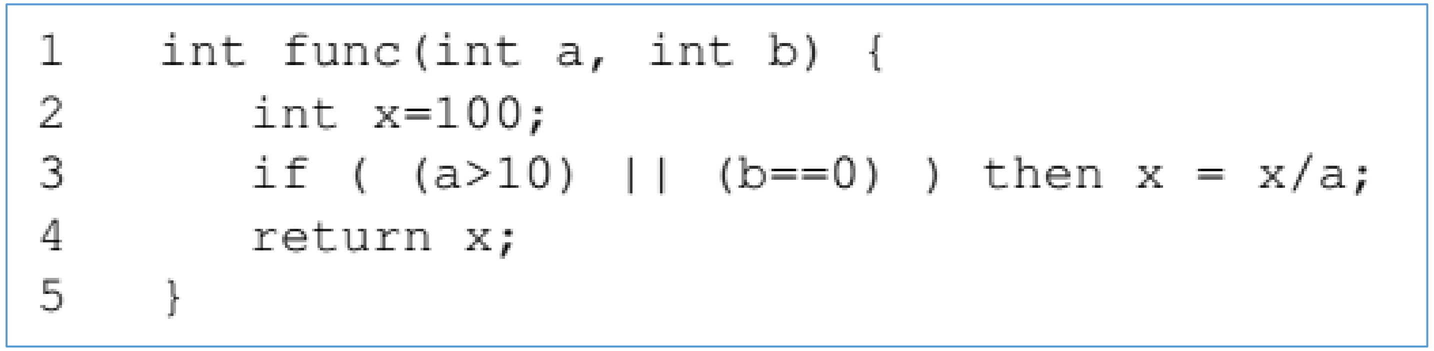
MCDC Example/Test Data



```
int func(int a, int b) {
   int x=100;
   if ( (a>10) || (b==0) ) then x = x/a;
   return x;
}
```

- Inputs (a=50,b=1) should result in the output value: 2

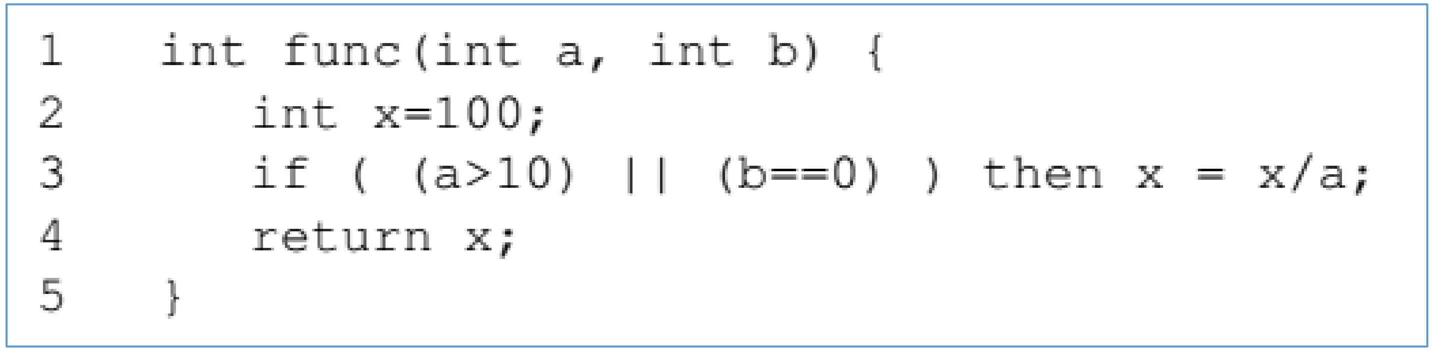
MCDC Example/Test Data



```
int func(int a, int b) {
   int x=100;
   if ( (a>10) || (b==0) ) then x = x/a;
   return x;
}
```

- Inputs (a=50,b=1) should result in the output value: 2
- (a=5,b=1) should result in the output value: 100
  - This shows the independent effect of changing the boolean condition (a>10)

MCDC Example/Test Data



```
int func(int a, int b) {
   int x=100;
   if ( (a>10) || (b==0) ) then x = x/a;
   return x;
}
```

- Inputs (a=50,b=1) should result in the output value: 2
- Inputs (a=5,b=1) should result in the output value: 100
  - This shows the independent effect of changing the boolean condition (a>10)
- Inputs (a=5,b=0) should result in the output value: 20
  - This shows the independent effect of changing the boolean condition (b==0)

Dataflow coverage/Definition-Use pairs

- Each path between the writing of a value to a variable (definition) and its subsequent reading (use) is executed during testing
  - Definition: **x=10**
  - Use: **y=x+20** or **if (x>100)**
- Every possible du pair is a test coverage item
  - Test must cause execution of the D followed by the U
- Strengths
  - This is a strong form of testing.
  - It generates test data in the pattern that data is manipulated in the program
- Weaknesses
  - The number of test cases can be very large
  - Hard to handle object references (C/C++ pointers) and arrays

AP Testing

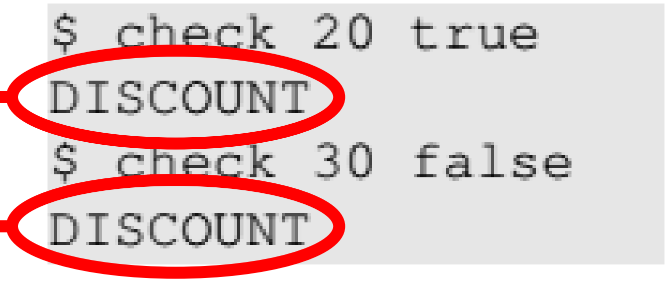
- All the "end to end paths"
- And end to end path:
  - starts at the beginning of the method
  - and exits at the end - return statement or final "}"

- Short example (fault 6)

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;
}
```

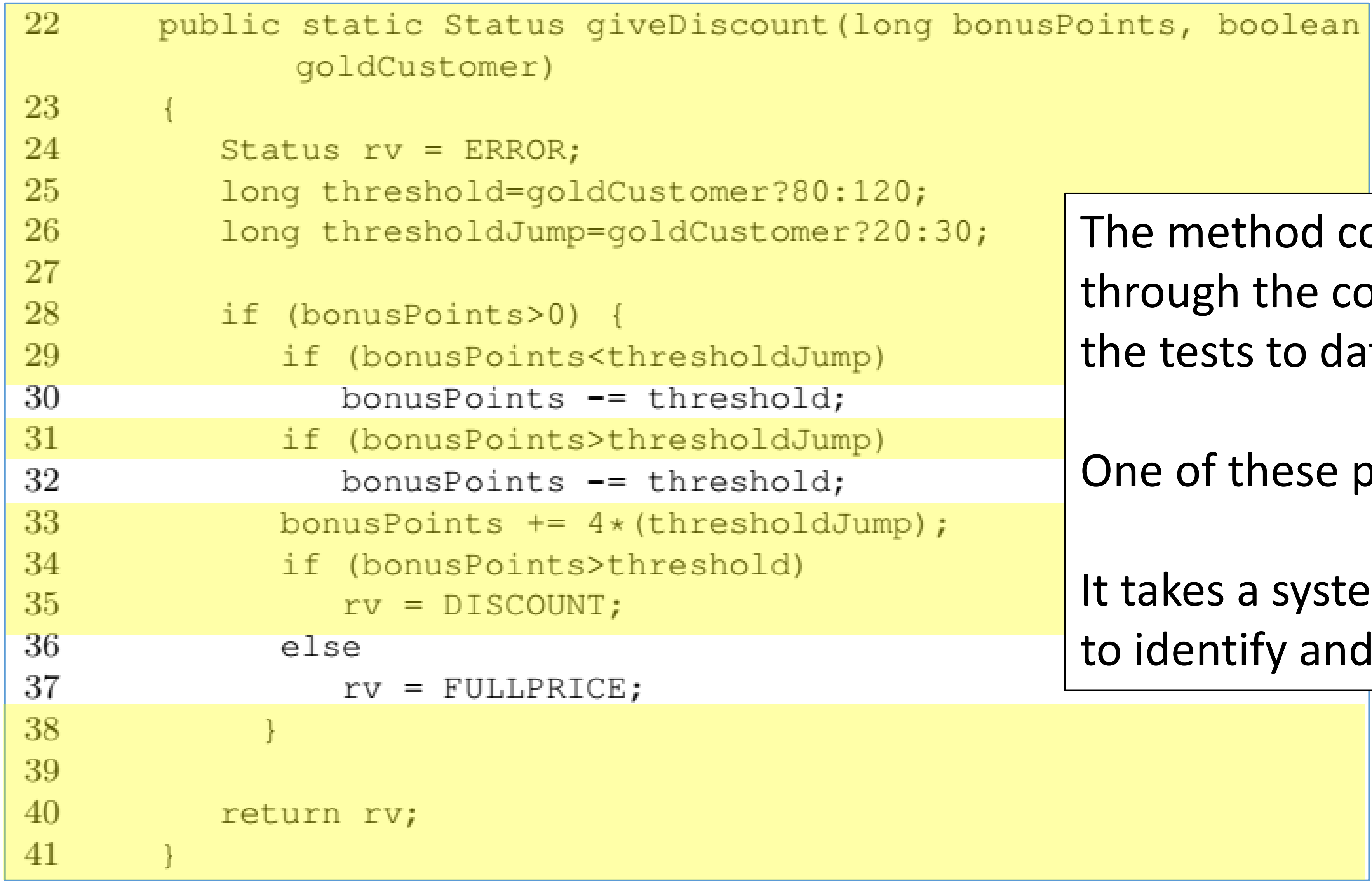
Demonstrating Fault 6



```
$ 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



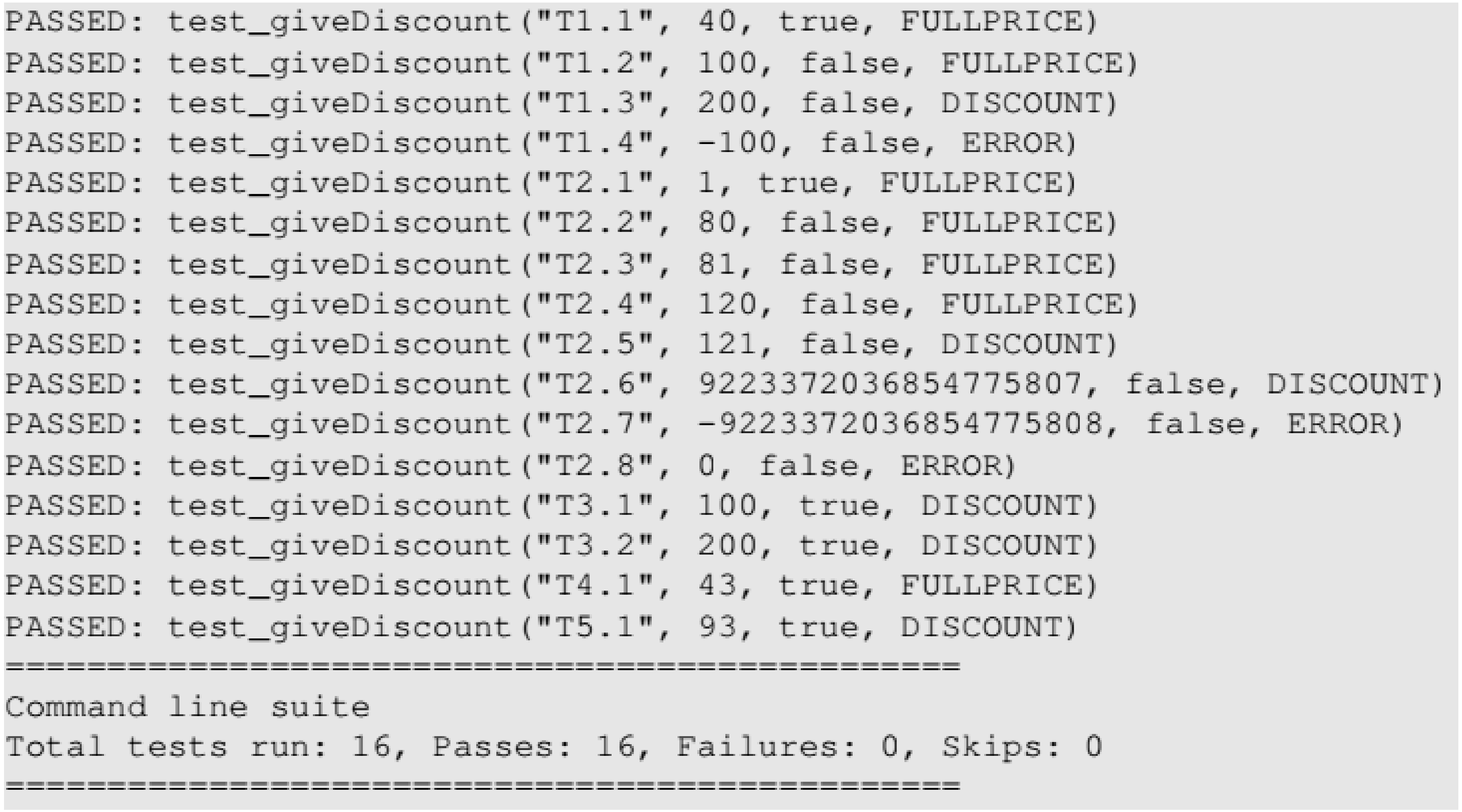
```
public static Status giveDiscount(long bonusPoints, boolean
       goldCustomer)
{
   Status rv = ERROR;
   long threshold=goldCustomer?80:120;
   long thresholdJump=goldCustomer?20:30;

   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

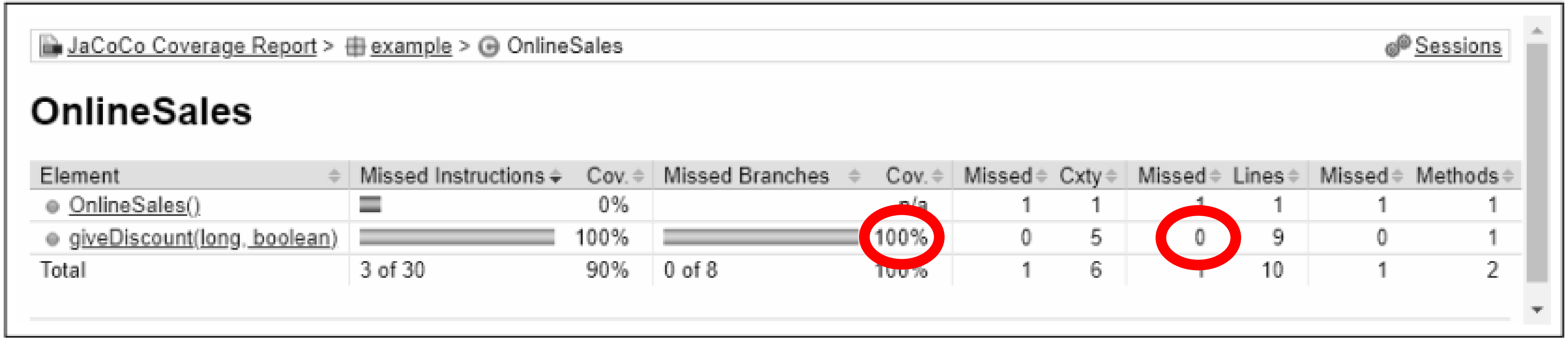
EP+BVA+DT+SC+BC Testing Against Fault 6



```
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



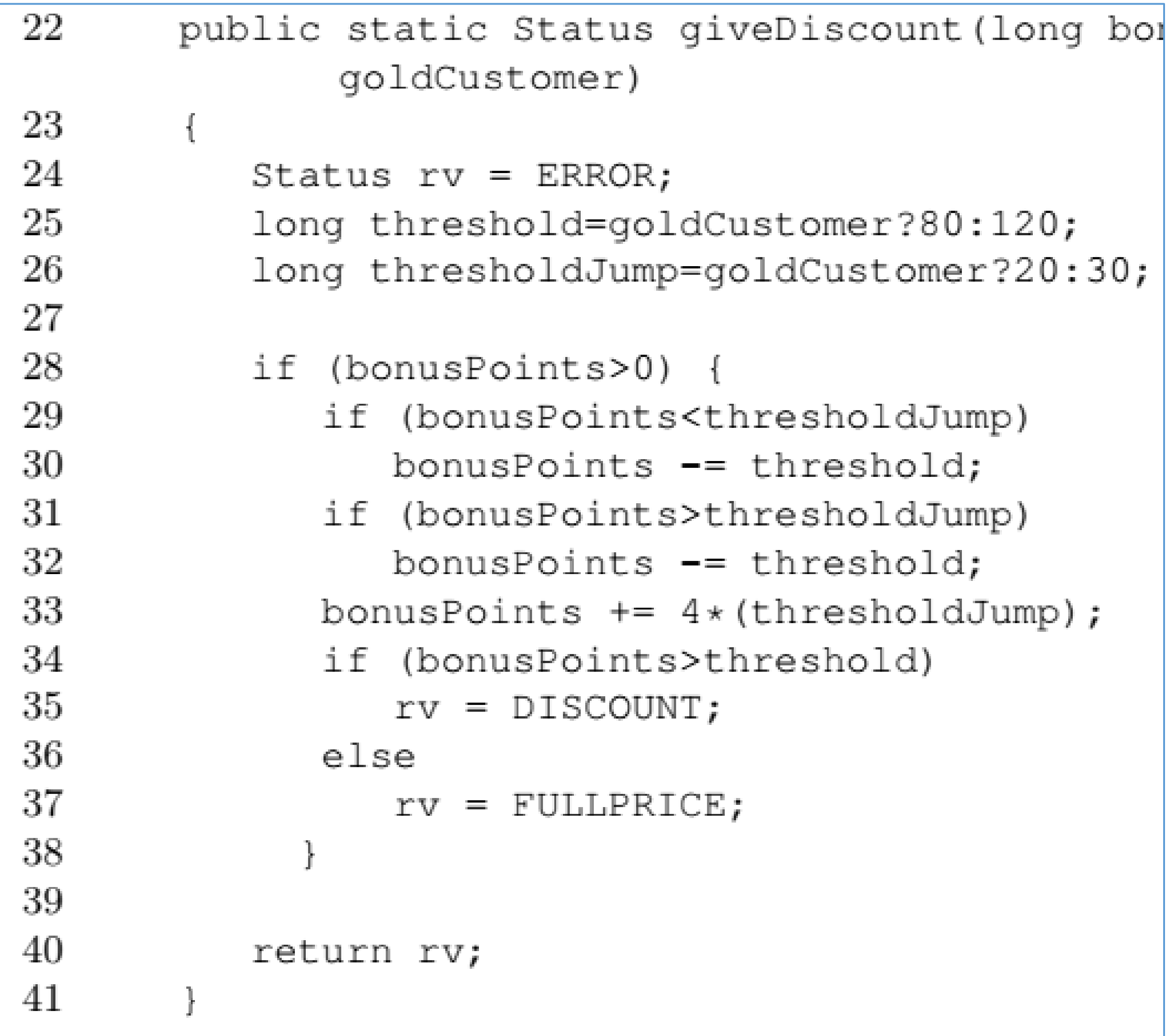
JaCoCo Coverage Report > example > OnlineSales Sessions

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

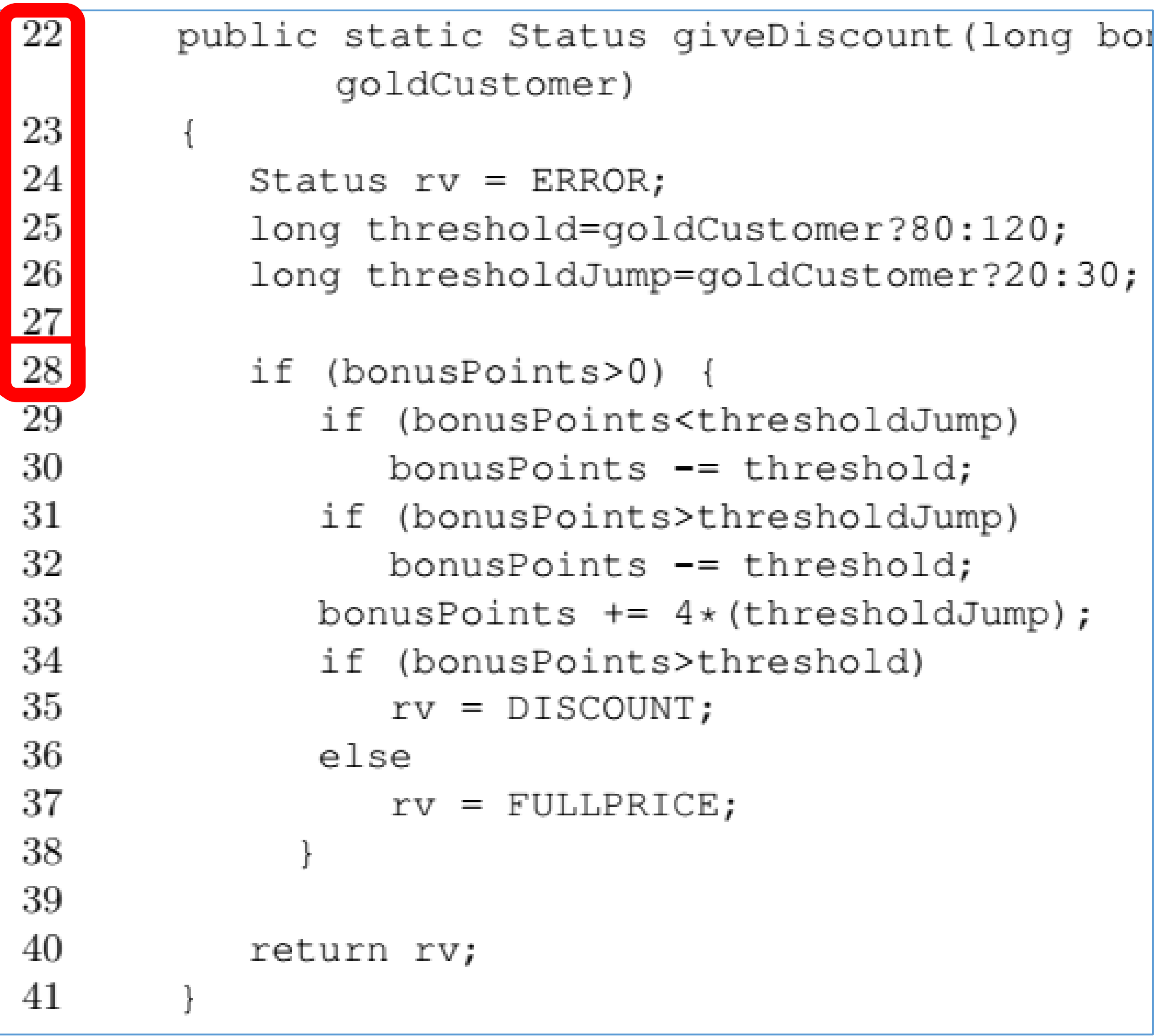
- Tests 4.1 and 5.1 are in fact redundant for this version of the code
- The code has been changed, and these tests are no longer required to achieve statement or branch coverage

AP - Code



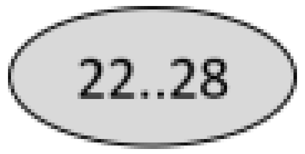
```
public static Status giveDiscount(long bo
        goldCustomer)
{
   Status rv = ERROR;
   long threshold=goldCustomer?80:120;
   long thresholdJump=goldCustomer?20:30;

   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;
}
```

CFG – Example (step 1)

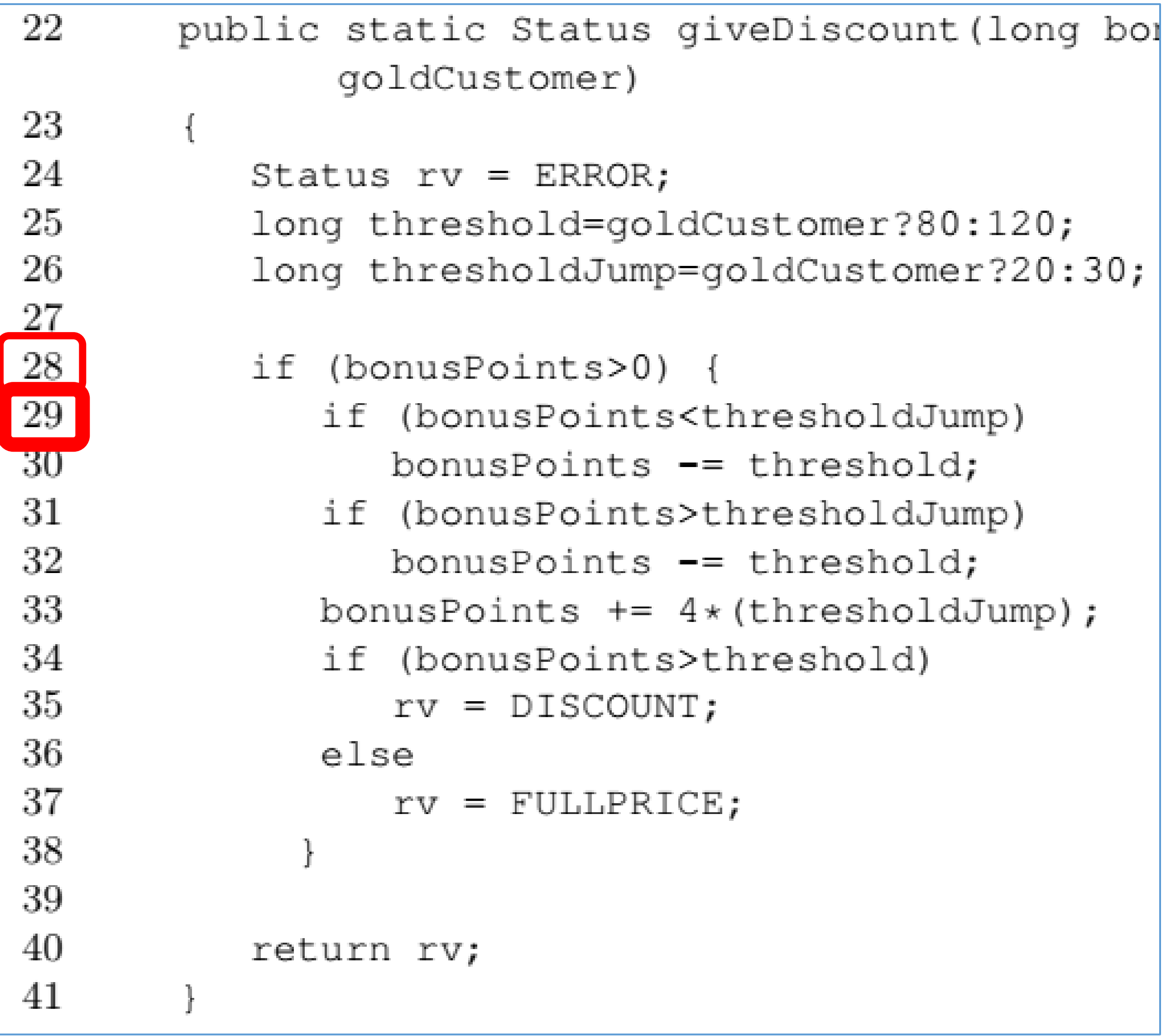


```
22      public static Status giveDiscount(long bo
               goldCustomer)
23      {
24         Status rv = ERROR;
25         long threshold=goldCustomer?80:120;
26         long thresholdJump=goldCustomer?20:30;
27
28         if (bonusPoints>0) {
29            if (bonusPoints<thresholdJump)
30               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      }
```

- Lines 22-28 are indivisible
- Represented by a single node in the CFG

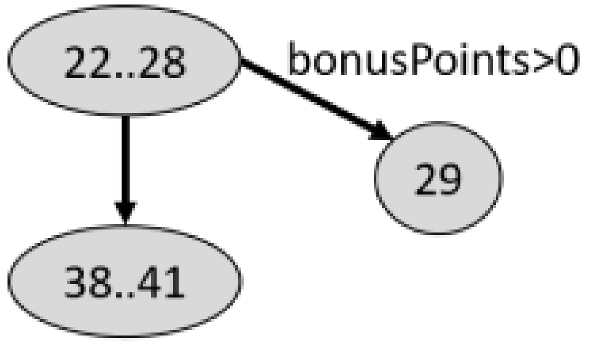


CFG – Example (step 2)

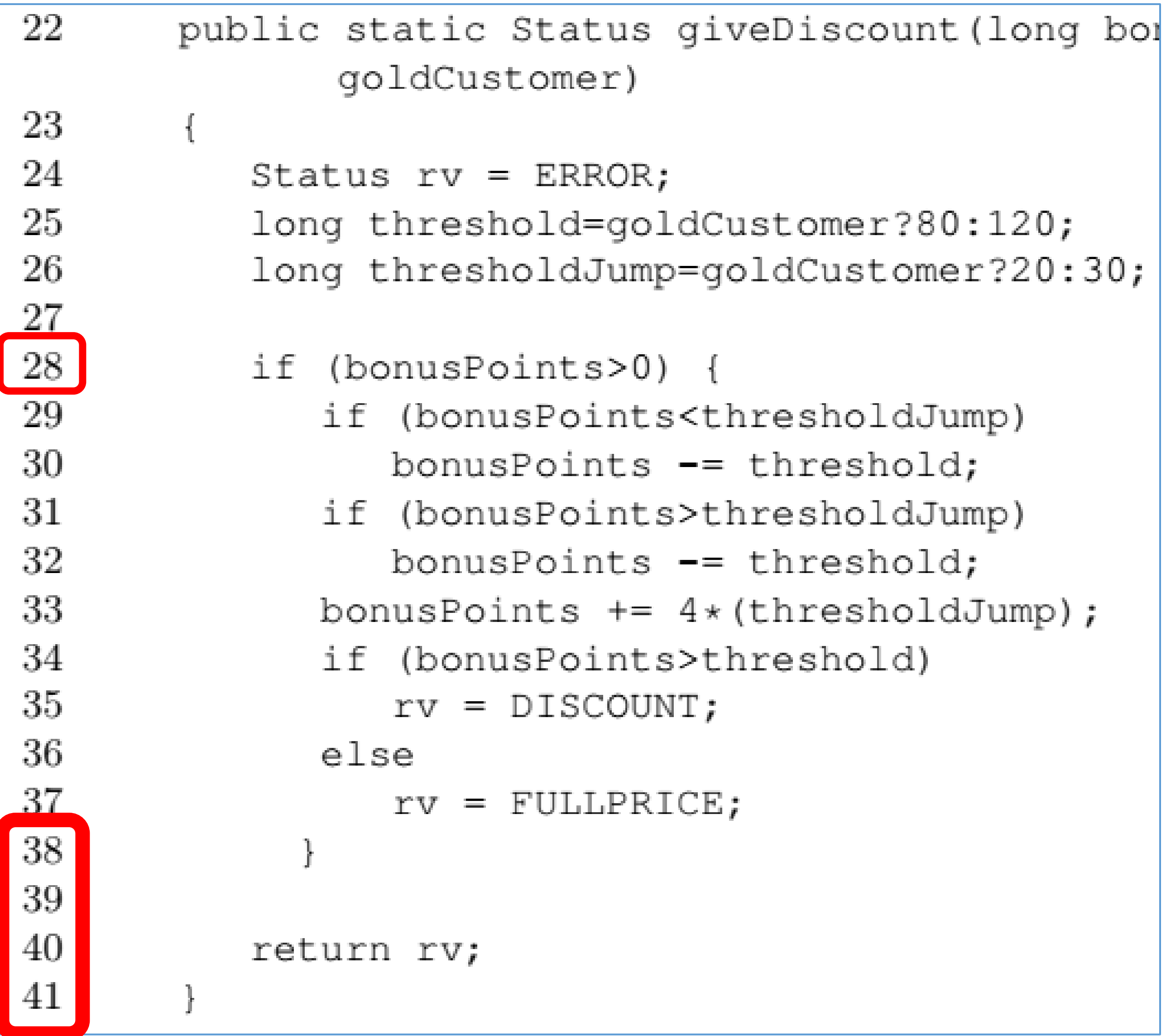


```
22   public static Status giveDiscount(long bo
              goldCustomer)
23   {
24      Status rv = ERROR;
25      long threshold=goldCustomer?80:120;
26      long thresholdJump=goldCustomer?20:30;
27
28      if (bonusPoints>0) {
29         if (bonusPoints<thresholdJump)
30            bonusPoints -= 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 28, if the decision (bonusPoints>0) is true, then control branches to line 29

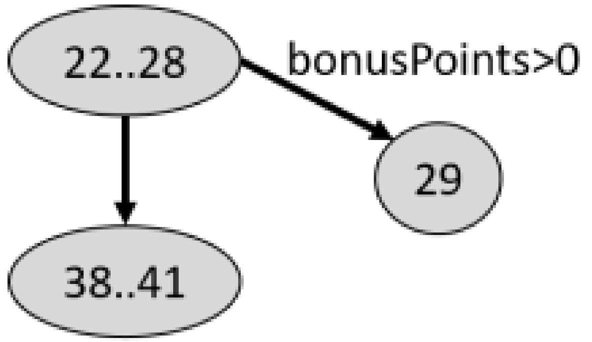


CFG – Example (step 2)



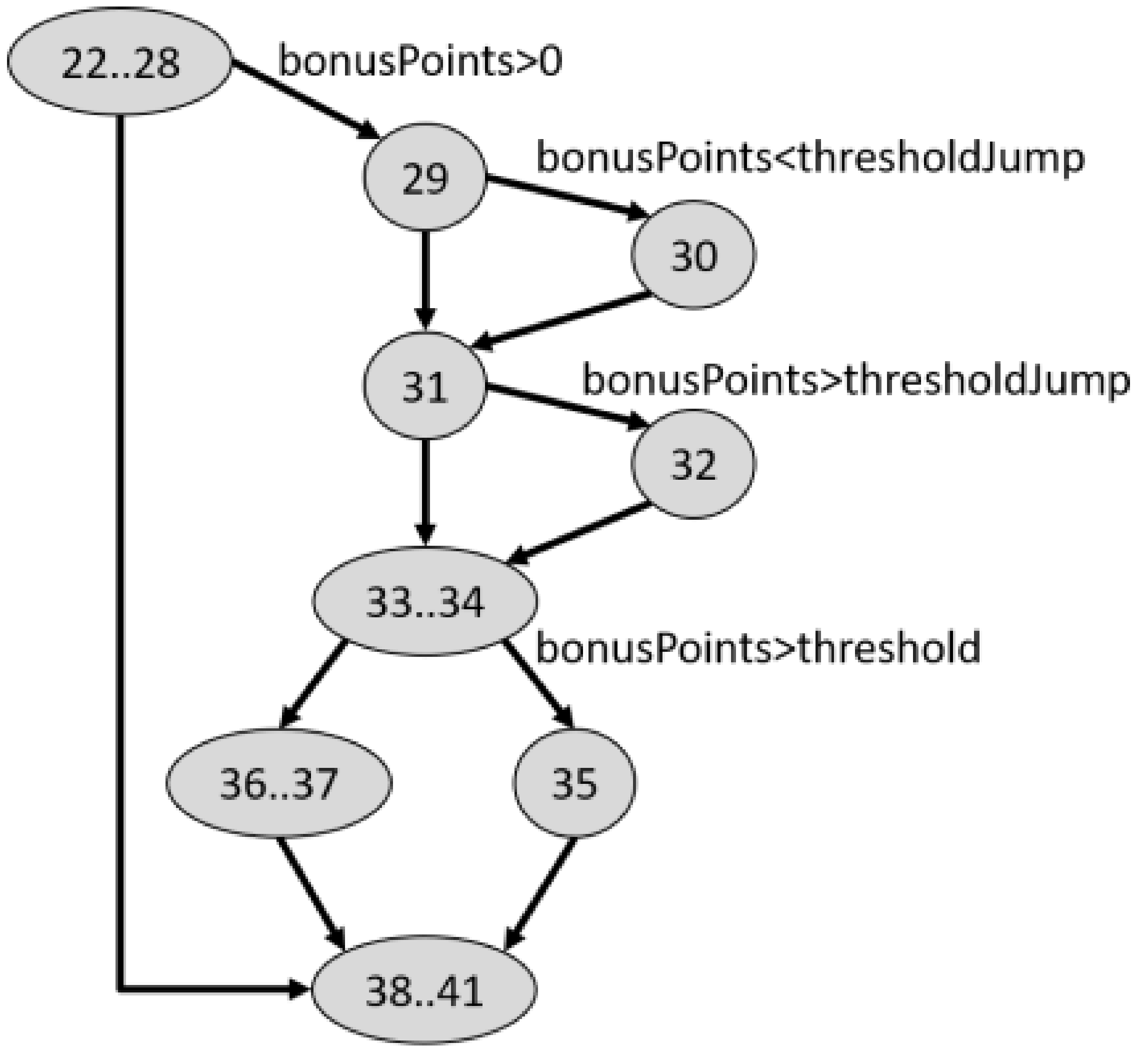
```
22      public static Status giveDiscount(long bo
                goldCustomer)
23      {
24         Status rv = ERROR;
25         long threshold=goldCustomer?80:120;
26         long thresholdJump=goldCustomer?20:30;
27
28         if (bonusPoints>0) {
29            if (bonusPoints<thresholdJump)
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 28, if the decision (bonusPoints>0) is true, then control branches to line 29
- If not, control branches to line 38
- Lines 38-41 are indivisible



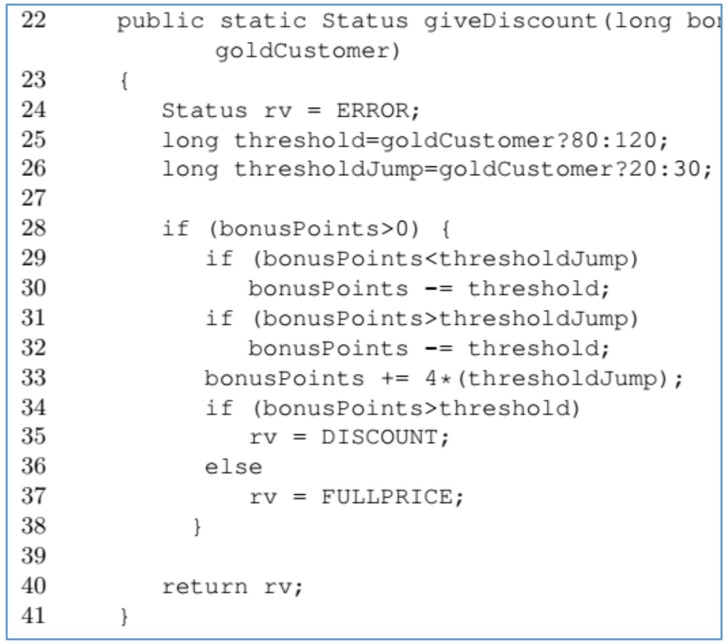
AP – Code and Full CFG

```
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      }
```

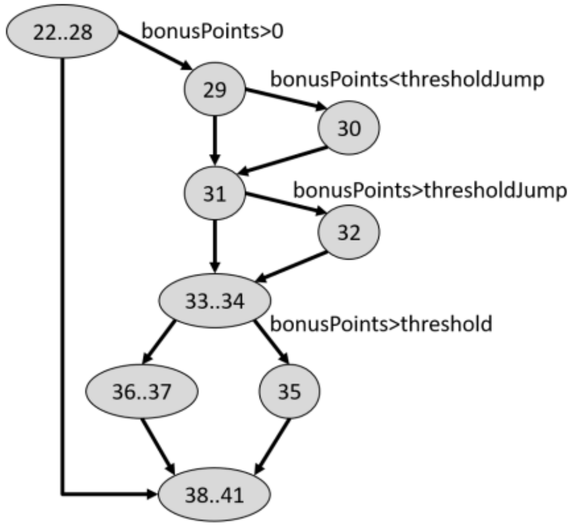


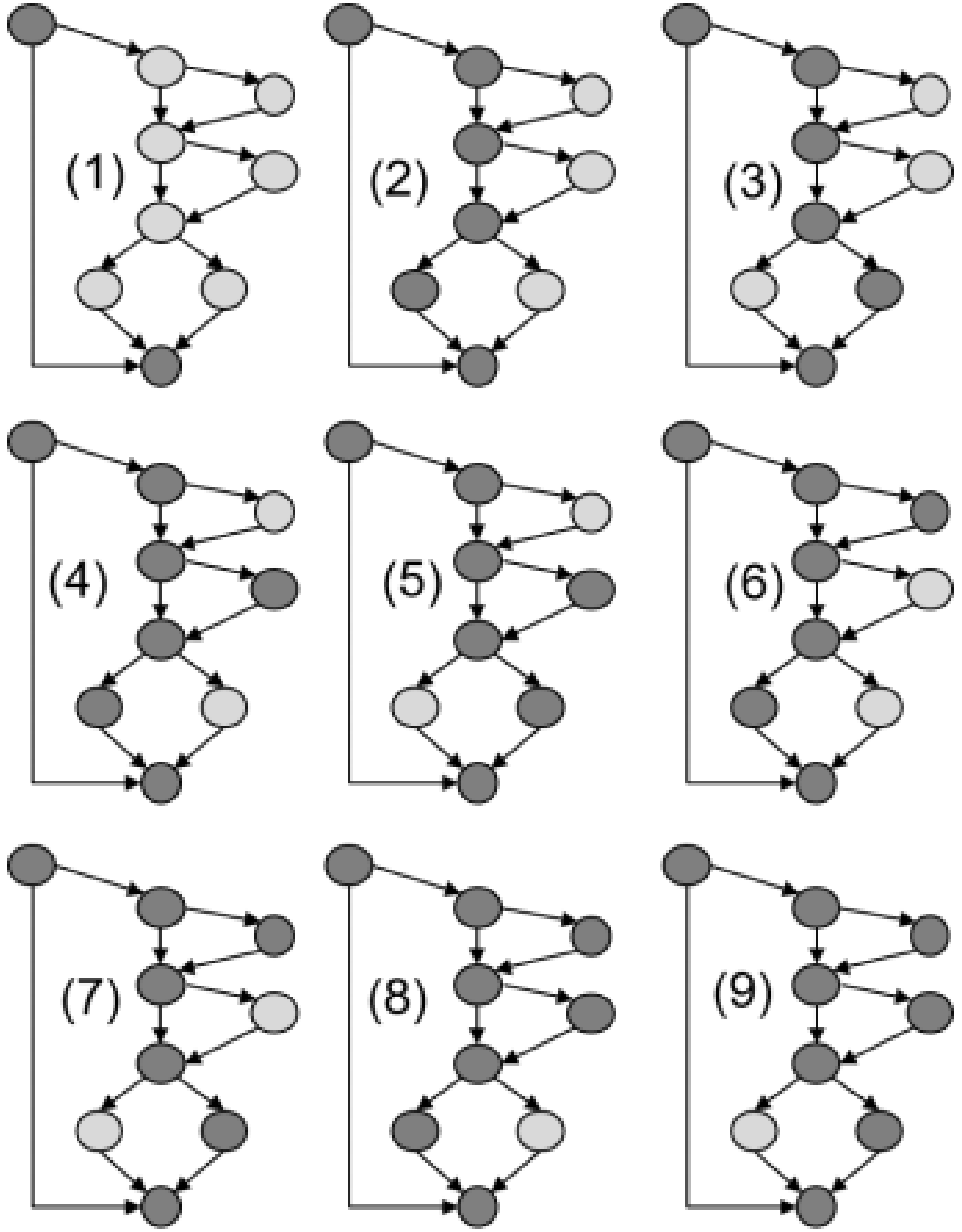
$$\begin{tikzcd}
22..28 \arrow[drr, "\text{bonusPoints>0}"] \arrow[dddddddrr] & & & \\
& & 29 \arrow[dr, "\text{bonusPoints<thresholdJump}"] \arrow[dd] & \\
& & & 30 \arrow[dl] \\
& & 31 \arrow[dr, "\text{bonusPoints>thresholdJump}"] \arrow[dd] & \\
& & & 32 \arrow[dl] \\
& & 33..34 \arrow[dl] \arrow[dr, "\text{bonusPoints>threshold}"] & \\
& 36..37 \arrow[dr] & & 35 \arrow[dl] \\
& & 38..41 &
\end{tikzcd}$$

AP – Code and CFG and end-to-end paths

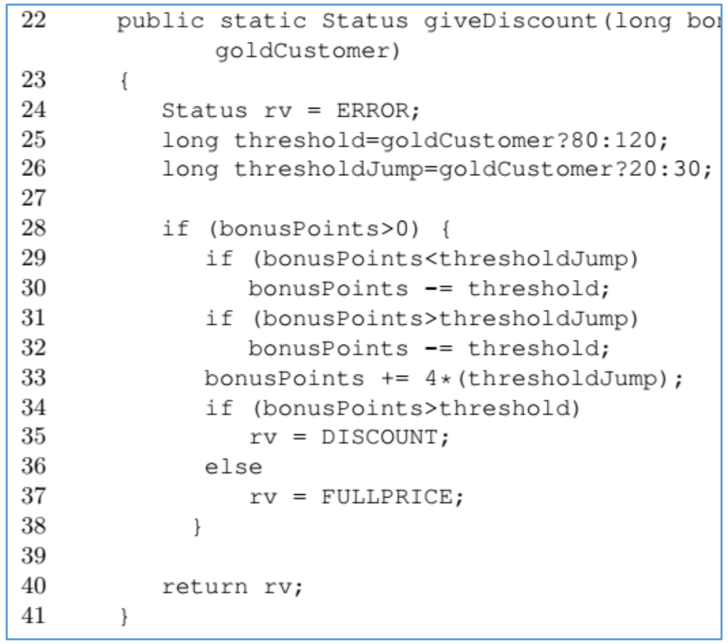


```
22      public static Status giveDiscount(long bo
               goldCustomer)
23      {
24         Status rv = ERROR;
25         long threshold=goldCustomer?80:120;
26         long thresholdJump=goldCustomer?20:30;
27
28         if (bonusPoints>0) {
29            if (bonusPoints<thresholdJump)
30               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      }
```

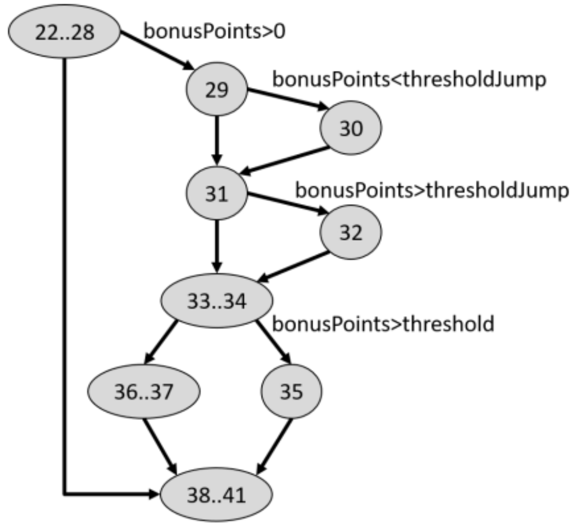


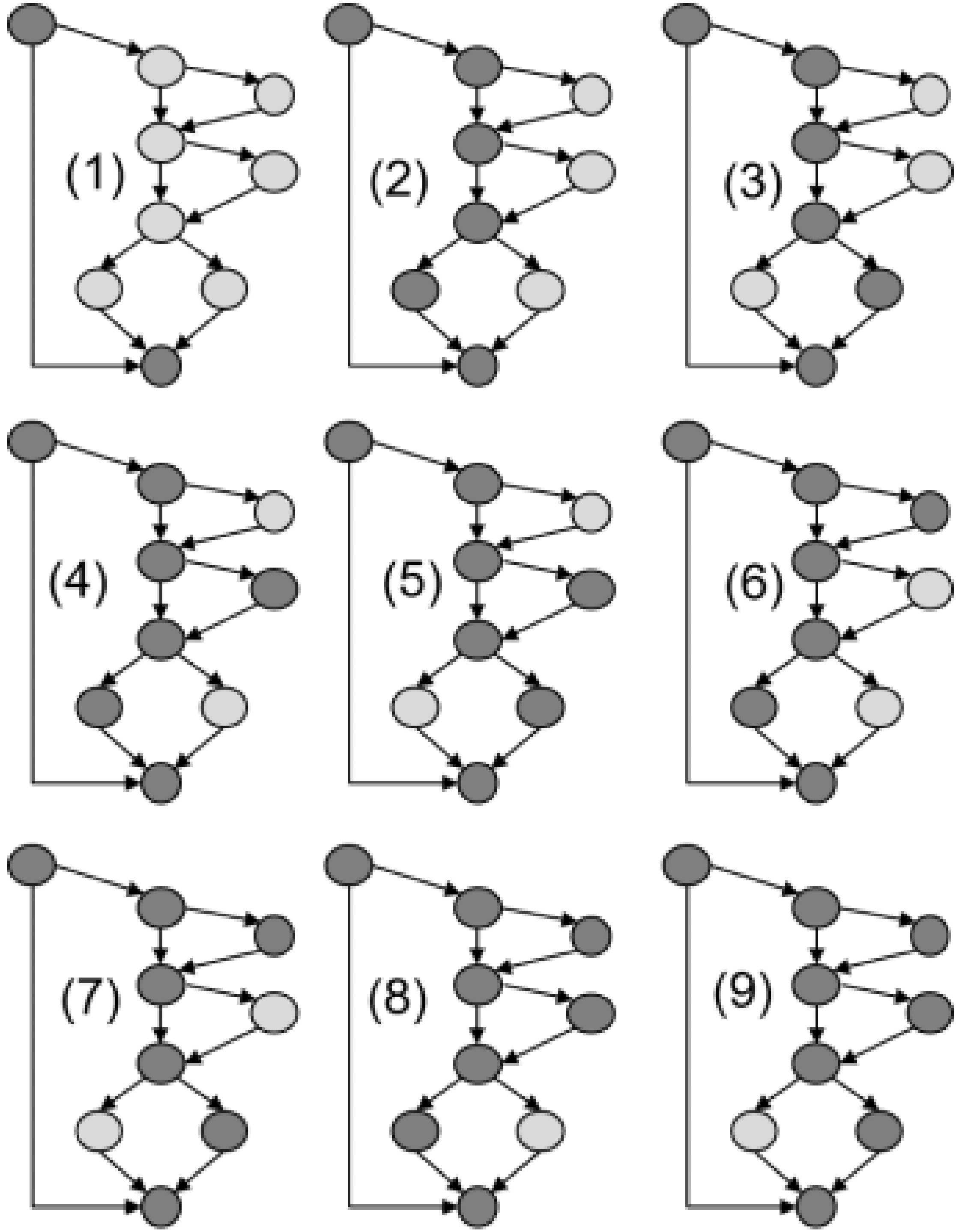


AP – Code and CFG and end-to-end paths



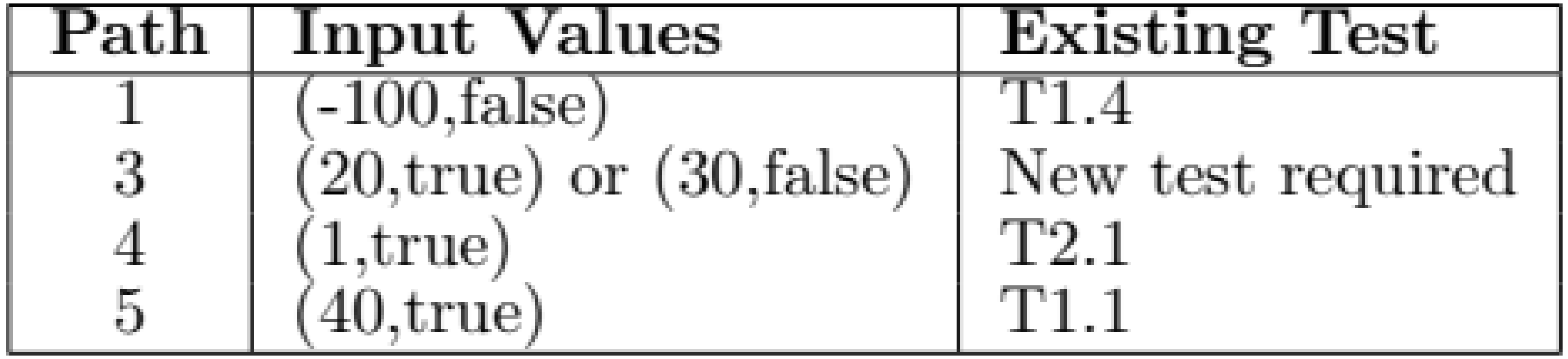
```
22     public static Status giveDiscount(long bor
              goldCustomer)
23     {
24        Status rv = ERROR;
25        long threshold=goldCustomer?80:120;
26        long thresholdJump=goldCustomer?20:30;
27
28        if (bonusPoints>0) {
29           if (bonusPoints<thresholdJump)
30              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     }
```





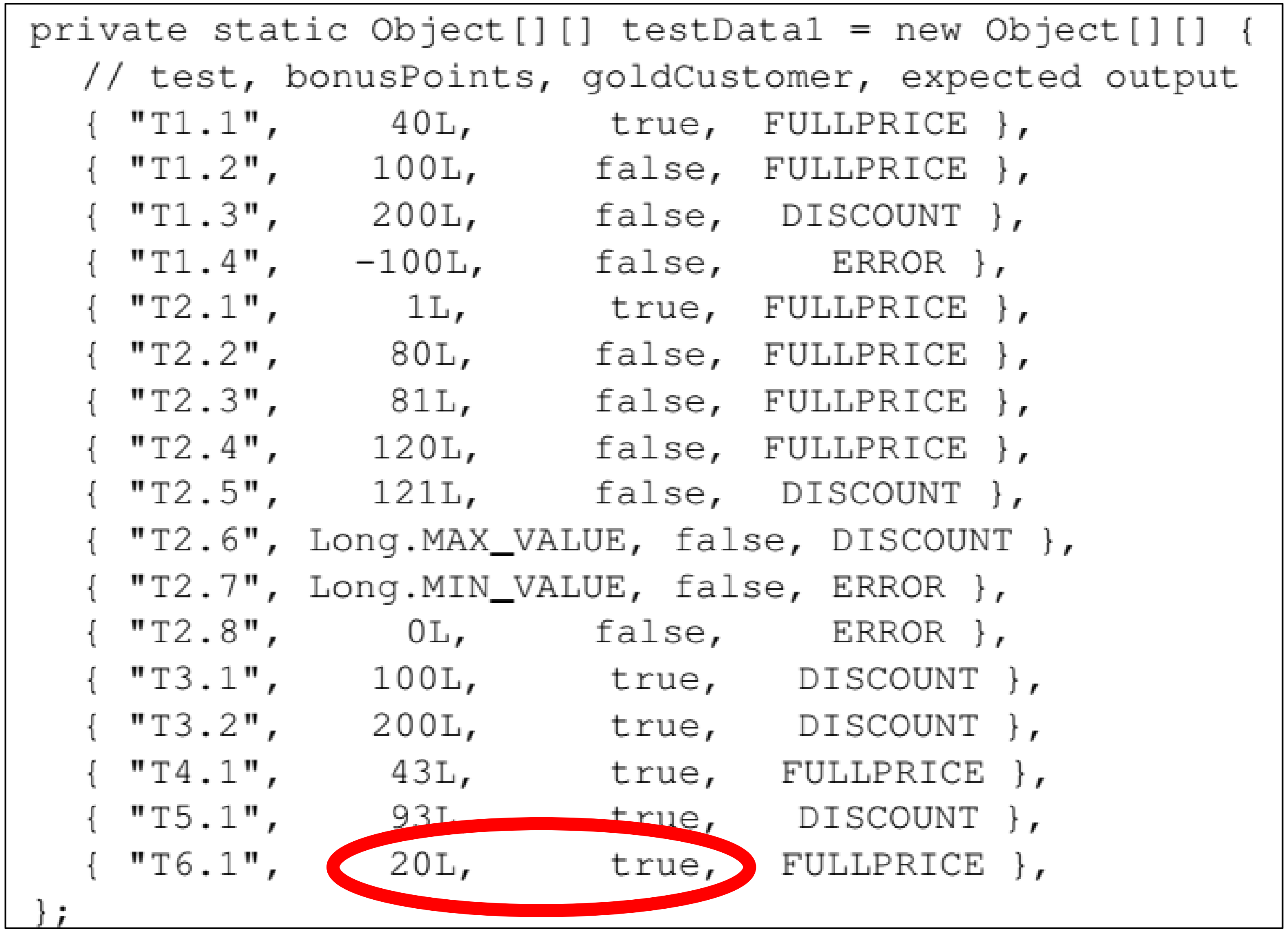
Note: not all these paths are logically possible
Long and complex procedure to follow each path
I will put a full explanation up on Moodle

Results of Analysis
Possible Paths and Associated Input Values



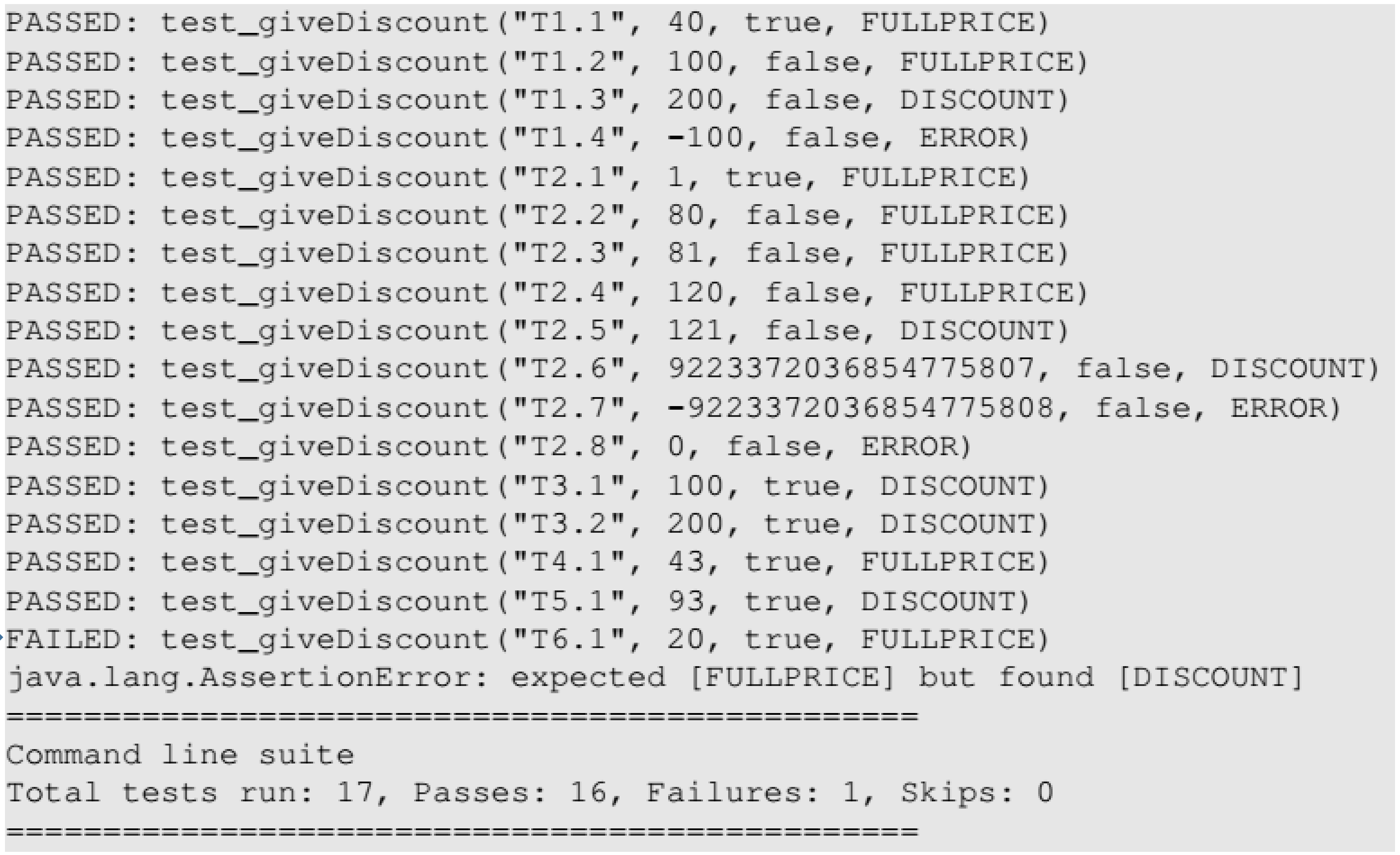
Path	Input Values	Existing Test
1	(-100,false)	T1.4
3	(20,true) or (30,false)	New test required
4	(1,true)	T2.1
5	(40,true)	T1.1

Test 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

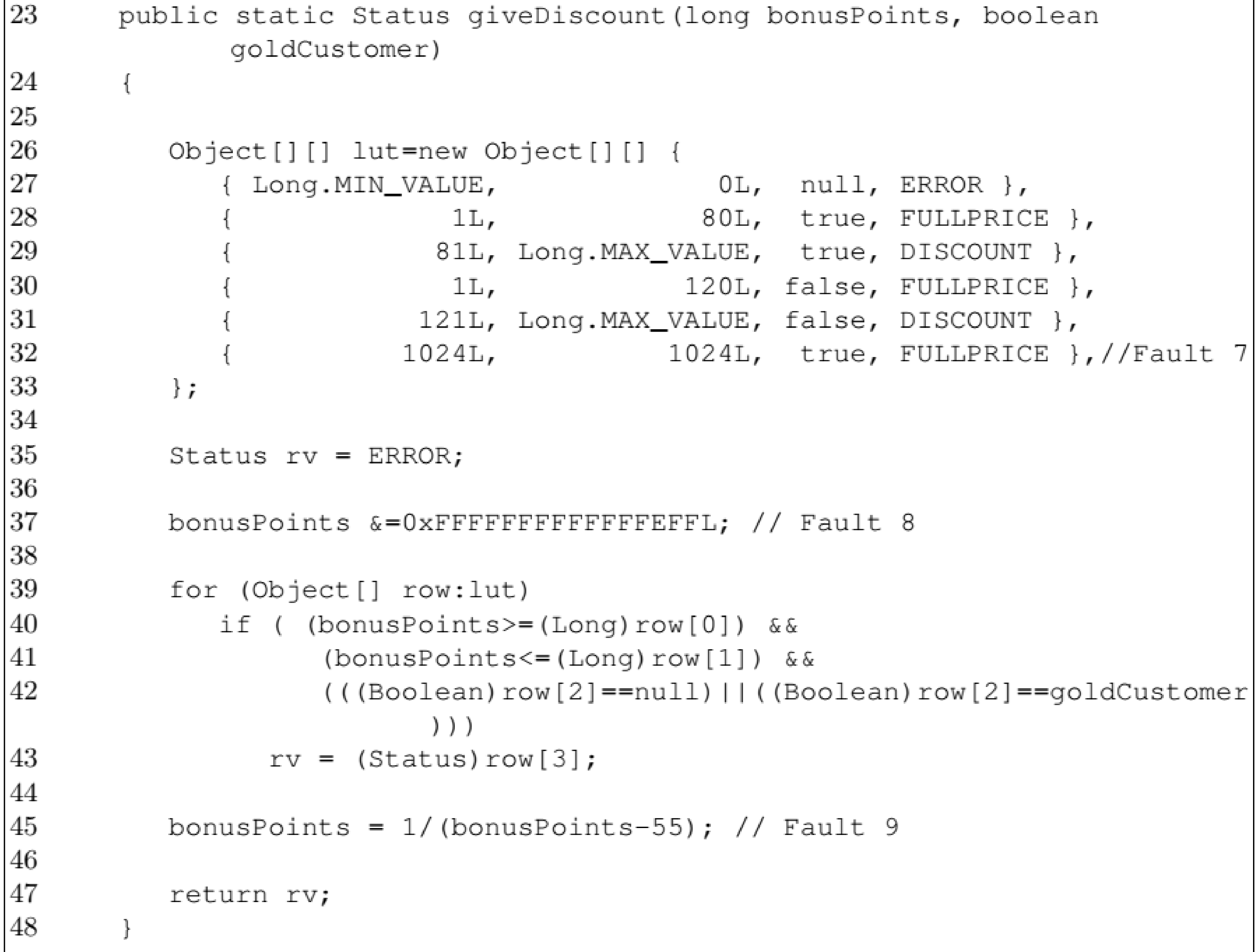


```
PASSED: test_giveDiscount("T1.1", 40, true, FULLPRICE)
PASSED: test_giveDiscount("T1.2", 100, false, FULLPRICE)
PASSED: test_giveDiscount("T1.3", 200, false, DISCOUNT)
PASSED: test_giveDiscount("T1.4", -100, false, ERROR)
PASSED: test_giveDiscount("T2.1", 1, true, FULLPRICE)
PASSED: test_giveDiscount("T2.2", 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

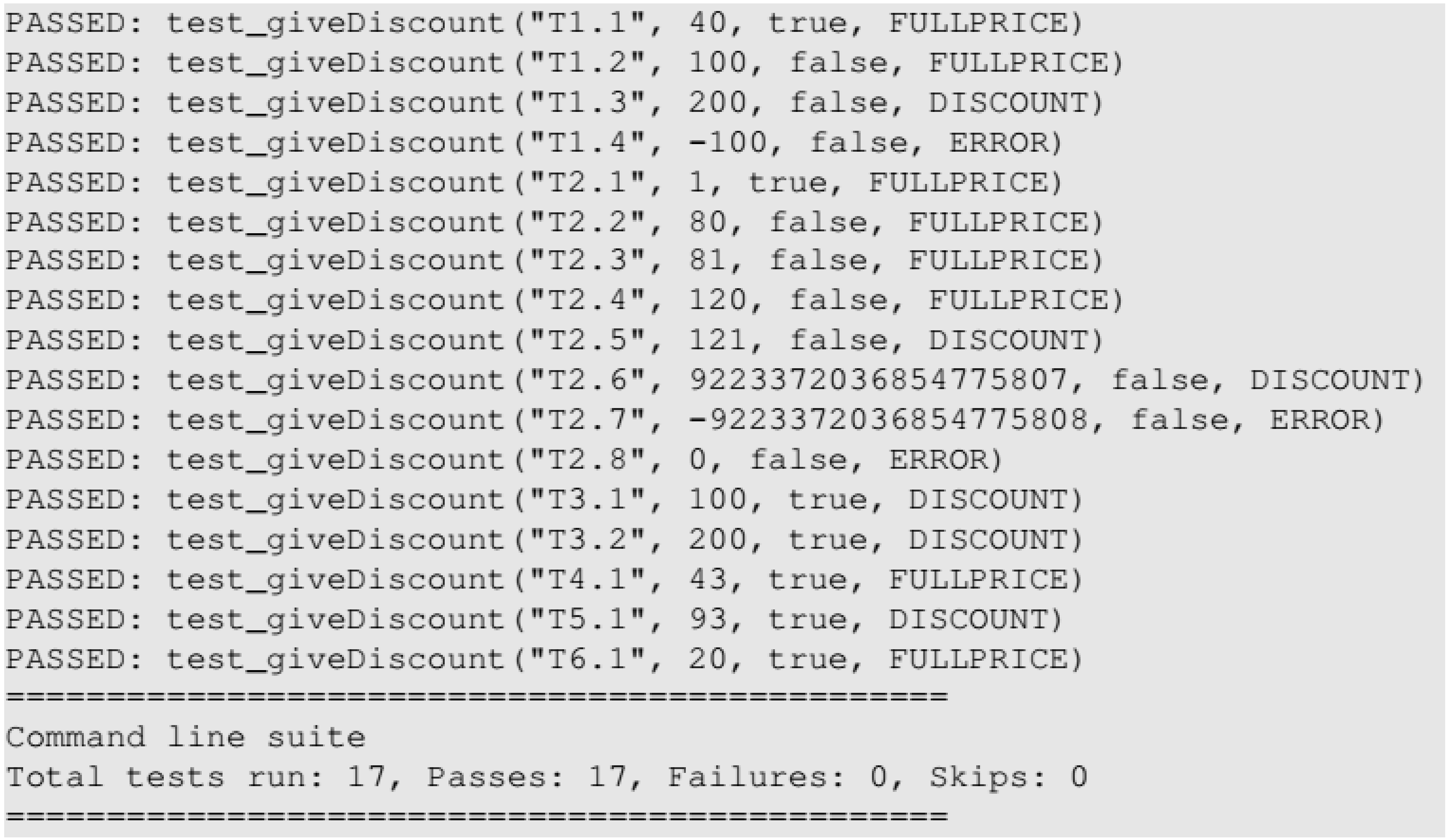
Limitations of AP

- 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’



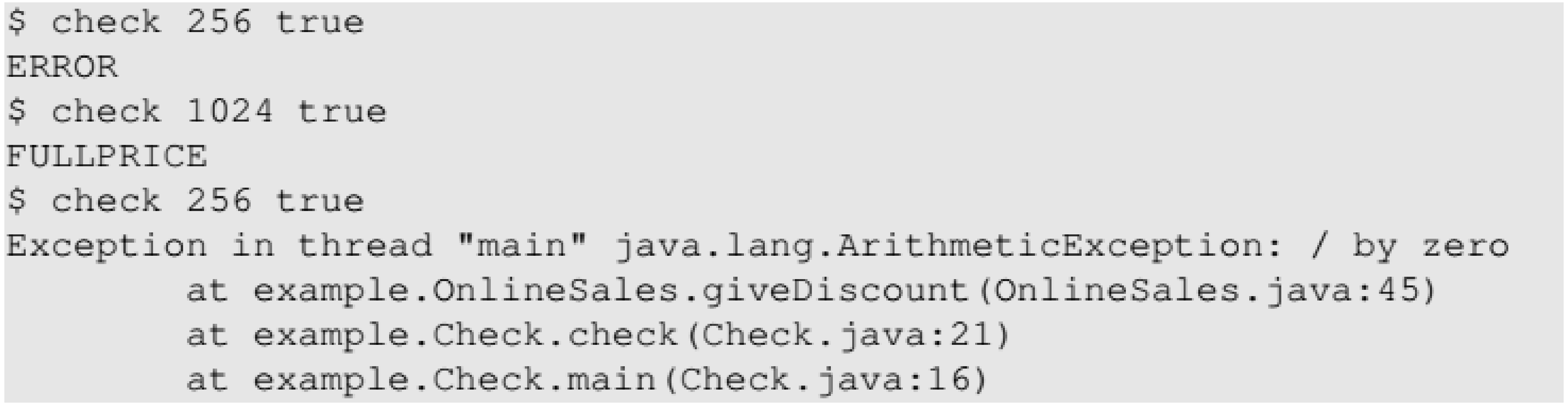
```
23      public static Status giveDiscount(long bonusPoints, boolean
                goldCustomer)
24      {
25
26         Object[][] lut=new Object[][] {
27            { Long.MIN_VALUE,               0L,  null, ERROR },
28            {             1L,              80L,  true, FULLPRICE },
29            {            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

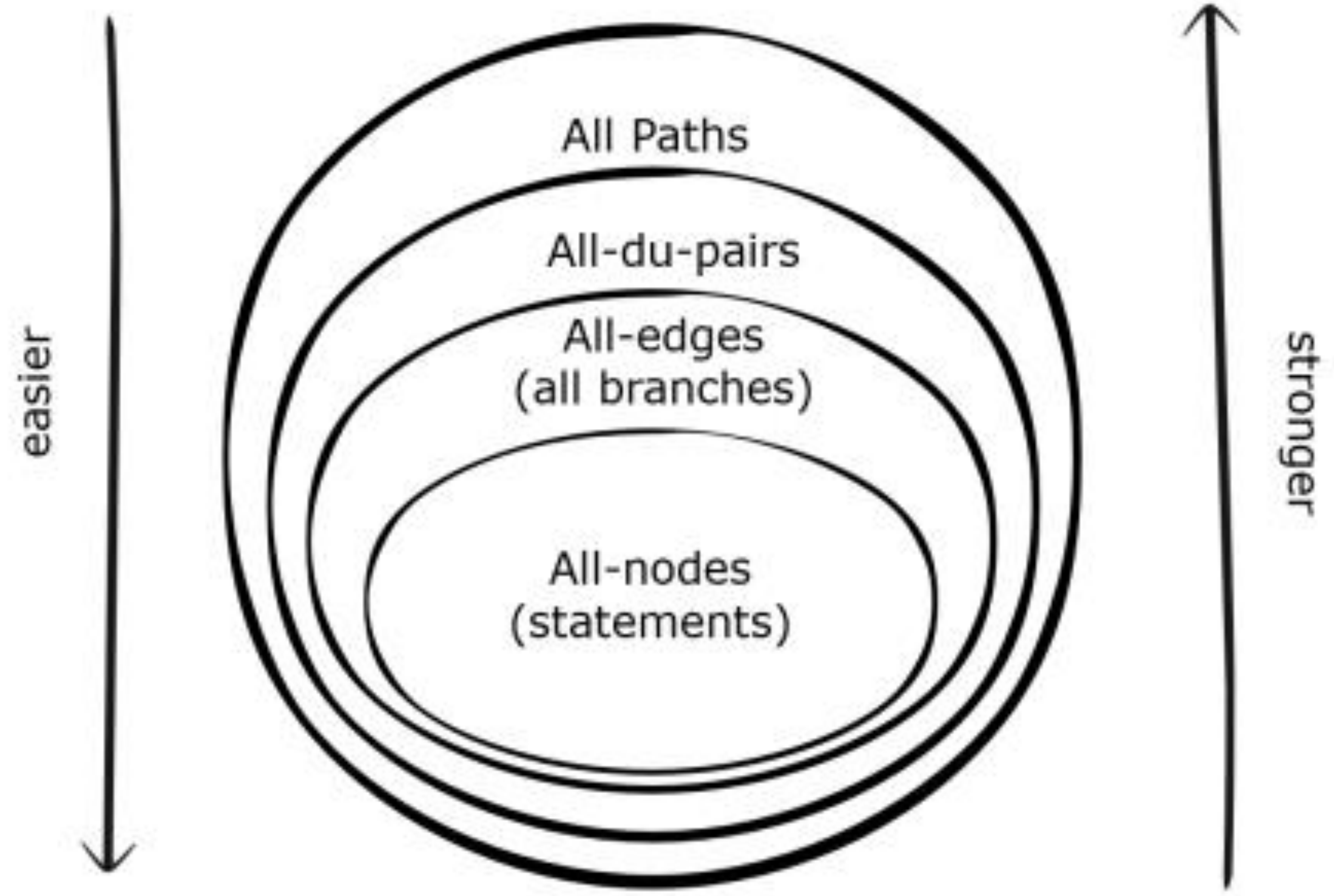
Strengths and Weaknesses

- **Strengths**
  - Covers all possible paths, which may have not been exercised using other methods
  - Guarantees statement coverage and branch coverage
- **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

White Box Test Ranking



Repair-Based Testing

- Additional tests to ensure a ‘repair’ works correctly
- **Specific Repair Test**: verifies that the software now operates correctly for the specific data inputs that caused the failure - WBT
- **Generic Repair Test**: verify repair works for other data values - BBT
- **Abstracted Repair Test**: In order to broaden the value of finding a fault, it is useful to abstract the fault as far as possible. BBT tests developed to find other places in the code where the same mistake may have been made by the programmer, leading to similar faults

Example

- A program searches a list of data. Due to a mistake by the programmer the code never finds penultimate entries in the list
- The fault is repaired by rewriting a line of code in the method that handles traversing the list
- **Specific**: to verify the repair, a white-box unit test is developed that exercises the new line of code
- **Generic**: A black-box unit test is then developed that ensures that, for different sets of data, the second last entry can be found in this list
- **Abstracted**: Finally, additional unit tests or system tests are developed that check that, for any other lists or collections of data in the program, the second-to-last entry is located correctly