TESTING-4

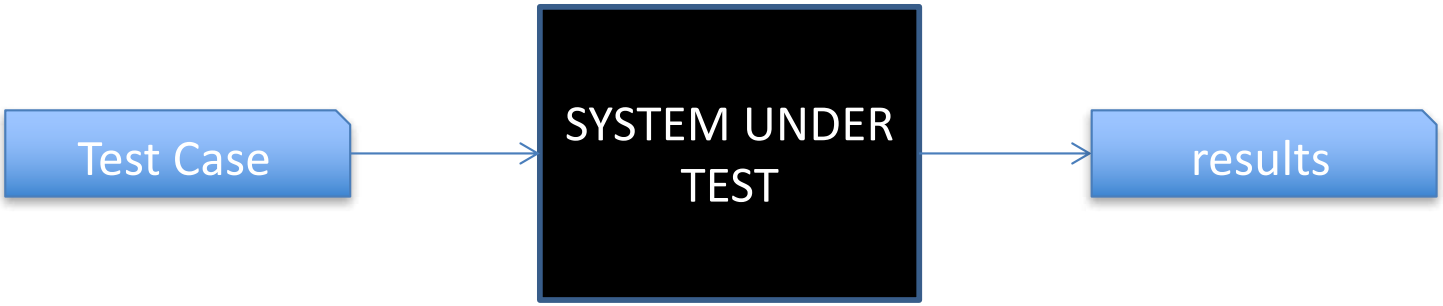
TEST CASE GENERATION

Black Box

Means you are testing from a functionality point of view.

Given a function how is it expected to behave?

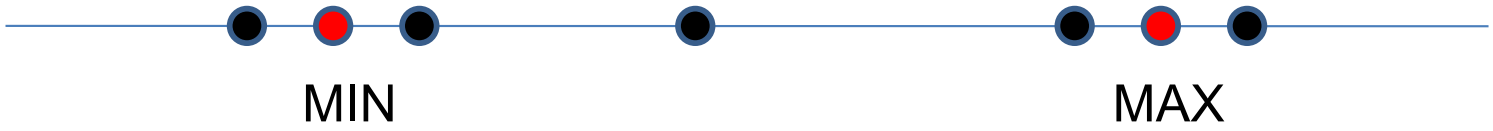
1) You do not have access to the source code OR you refrain from looking at the source code.



2) You know what the code is supposed to do.

You have to check whether it is working ok.

Boundary Value Testing



- Min, Min+1, Nominal, Max-1, Max
- Also, Min-1, Max+1   (For Robustness)

- Vary ONE variable.
- Hold value of rest of variables at a fixed value.

Examples

- triangle program
  - takes as input three sides, reports either not a triangle or that it is a scalene, equilateral, or isoceles triangle

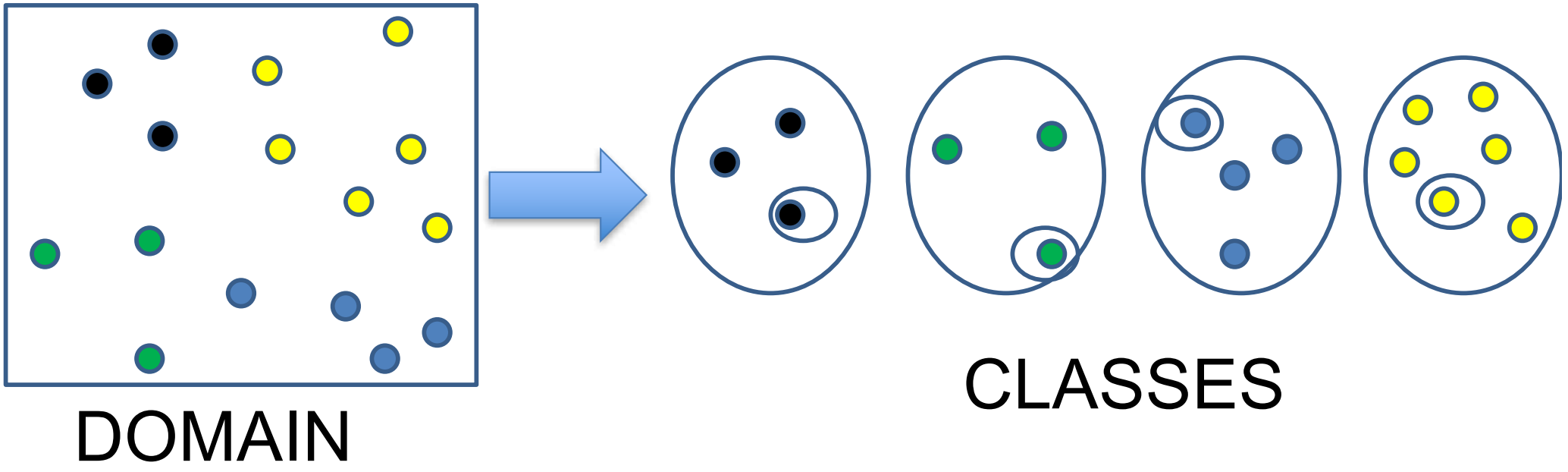
Assume: inputs are between 1 and 100

a	b	c	expected	actual?
0	50	50		
1	50	50		
2	50	50		
50	50	50		
99	50	50		
100	50	50		
101	50	50	NEXT – hold a and c at 50 and vary B (and then later vary C)	

- Easy to automatically generate?
- Good Quality Tests?

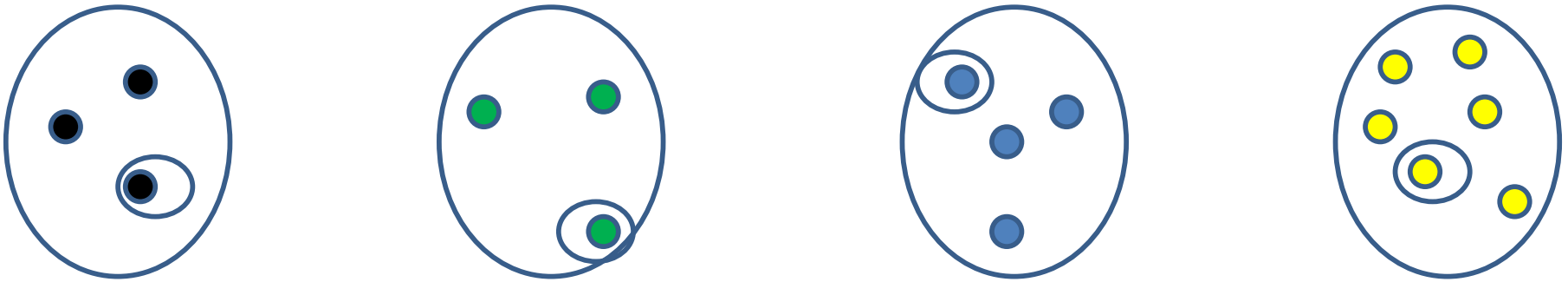
Equivalence Class Testing

- Equivalence class
  - Domain partitioned into **disjoint** classes (no overlap)
  - each element of domain must belong to one class.



Equivalence Class Testing

- all elements of a class are expected to behave similarly, i.e.
- testing one element would imply testing the whole class.



SELECT a representative from each class

example

Triangle program

- D1 - all sides are unequal
- D2 - a=b=c
- D3 - a=b a!=c
- D4 - a=c a!=b
- D5 - b=c a!=b
- D6 - a >= b+c <Not a triangle>
- D7 - b >= a+c <Not a triangle>
- D8 - c >= a+b <Not a triangle>

- You can choose an input from each of the above classes.
- IMPORTANT:
  - NEED TO SELECT FROM INVALID POSSIBILITIES ALSO.

Next Date program

- One example could be partitioning each input
  - Month: {30 days} (31 days} {feb}
  - Day: {1-17} {28} {29} {30} {31}
  - Year: {leap years} <non-leap years} {year 2000}

- Take one element from each class as a cartesian product.

- Easy to automatically generate?
- Good Quality Tests?

Random Testing

- Generate random tests (i.e. a, b, and c)

- Use **operational profile** (or frequency of use during actual operation) as a guide to generate tests.

  An Example Operational profile

  - Equilateral 30%
  - Isoceles 20%
  - Scalene 20%
  - Not a Triangle 30%

Example.

- Randomly generate three sides of triangle.
- However, keep a count of different types of triangles.
- Ensure %'s of different types of triangles match operational profile.
- i.e. if 1000 tests are generated, make sure that 300 are equilateral!

- Easy to automatically generate?
- Good Quality Tests?

WHITE BOX

means you also look at the source code and test the internal workings for defects.

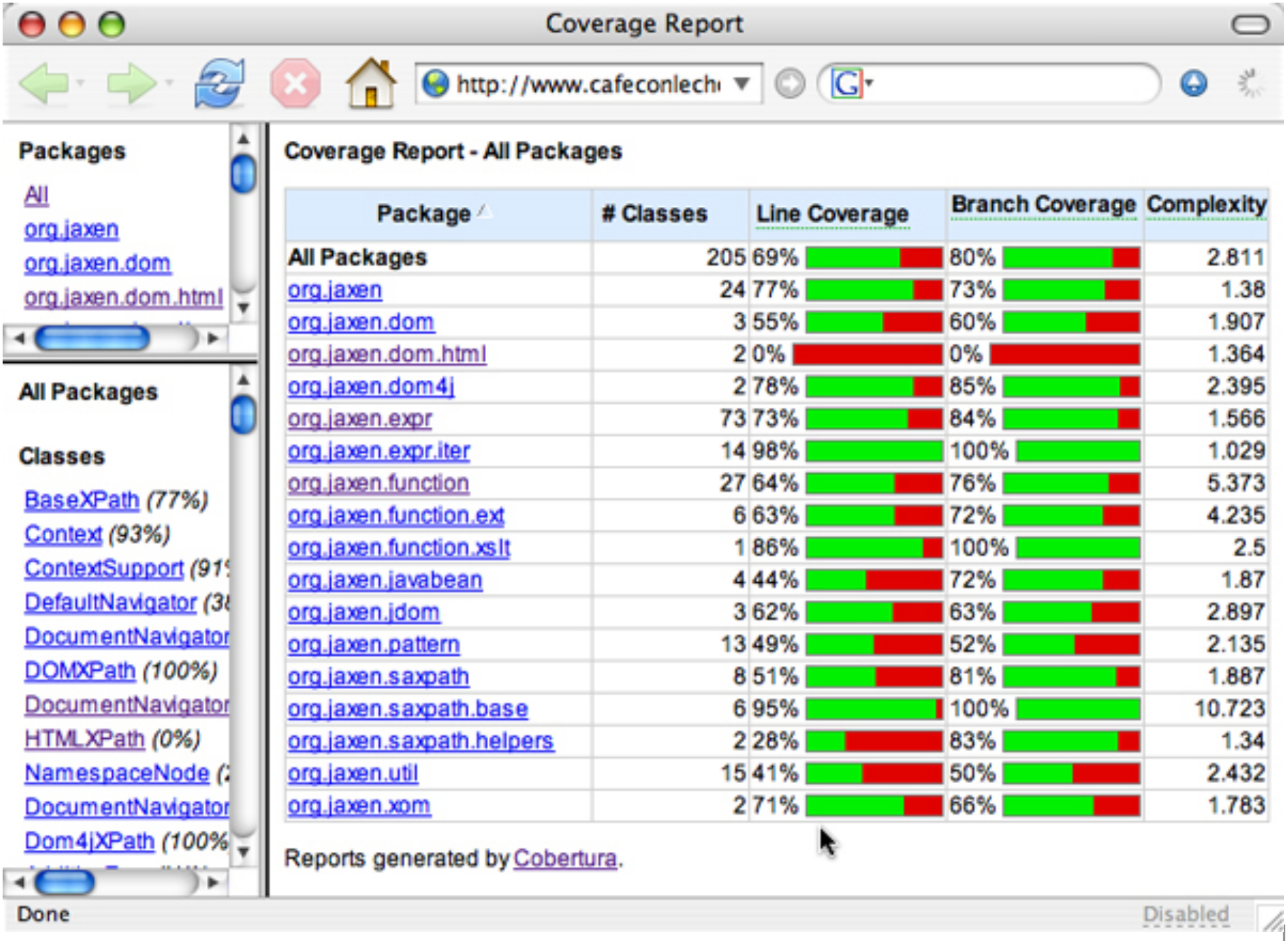
White box testing

- One way is to
  - first come up with black box tests.
  - Then check code coverage.
  - Now think of more black box tests that you may have missed.
  - check code coverage (and so on…)

- Another way is to "backtrack" and determine inputs that would force tests to execute chosen paths.

CODE COVERAGE

- There are several code coverage metrics which measure which parts of the code were executed by running all our tests.
- These metrics are gathered by software tools that instrument our code and gather information when we run our code.
- One such tool is djUnit.


Coverage Report
http://www.cafeconlech
Packages
All
org.jaxen
org.jaxen.dom
org.jaxen.dom.html
All Packages
Classes
BaseXPath (77%)
Context (93%)
ContextSupport (91
DefaultNavigator (3
DocumentNavigator
DOMXPath (100%)
DocumentNavigator
HTMLXPath (0%)
NamespaceNode (
DocumentNavigator
Dom4jXPath (100%
Coverage Report - All Packages
Package
Classes
Line Coverage
Branch Coverage
Complexity
All Packages 205 69% 80% 2.811
org.jaxen 24 77% 73% 1.38
org.jaxen.dom 3 55% 60% 1.907
org.jaxen.dom.html 2 0% 0% 1.364
org.jaxen.dom4j 2 78% 85% 2.395
org.jaxen.expr 73 73% 84% 1.566
org.jaxen.expr.iter 14 98% 100% 1.029
org.jaxen.function 27 64% 76% 5.373
org.jaxen.function.ext 6 63% 72% 4.235
org.jaxen.function.xslt 1 86% 100% 2.5
org.jaxen.javabean 4 44% 72% 1.87
org.jaxen.jdom 3 62% 63% 2.897
org.jaxen.pattern 13 49% 52% 2.135
org.jaxen.saxpath 8 51% 81% 1.887
org.jaxen.saxpath.base 6 95% 100% 10.723
org.jaxen.saxpath.helpers 2 28% 83% 1.34
org.jaxen.util 15 41% 50% 2.432
org.jaxen.xom 2 71% 66% 1.783
Reports generated by Cobertura.
Done
Disabled

Statement Coverage

- **Statement coverage** is a measure of the percentage of program statements that are run when your tests are executed.

- The objective should be to achieve 100% statement coverage through your testing. How about unreachable code??

```
1    float foo (int a, int b, int c, int d, float e)  {
2          float e;
3          if (a == 0)  {
4               return 0;
5          }
6          int x = 0;
7          if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9          }
10          e = 1/x;
11         return e;
12    }
```

- **Test Case 1:** call the method **foo(0, 0, 0, 0, 0.)**, expected return value of 0. Lines executed are lines 1-5 out of 12 lines of code - a 42% (5/12) statement coverage.

- **Test Case 2:** the method call **foo(1, 1, 1, 1, 1.)**, expected return value of 1. This executes the program statements on lines 6-12. Total is now a 100% statement coverage.

Decision/Branch Coverage

- **Decision or branch coverage** is a measure of how many decisions have been evaluated as both true and false in the testing.
- For decision/branch coverage, evaluate an entire Boolean expression as one true-or-false predicate

- The objective should be to achieve 100% branch coverage through your testing.

```
1    float foo (int a, int b, int c, int d, float e)  {
2          float e;
3          if (a == 0)  {
4               return 0;
5          }
6          int x = 0;
7          if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9          }
10          e = 1/x;
11         return e;
12    }
```

What's the coverage?

75%

Decision Coverage

Line #	Predicate	True	False
3	(a == 0)	Test Case 1	Test Case 2
		foo(0, 0, 0, 0, 0)	**foo(1, 1, 1, 1, 1)**
		return 0	**return 1**
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2	
		foo(1, 1, 1, 1, 1)	
		return 1	

```
1   float foo (int a, int b, int c, int d, float e)  {
2         float e;
3         if (a == 0)  {
4               return 0;
5         }
6         int x = 0;
7         if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9         }
10          e = 1/x;
11        return e;
12    }
```

Add new test case **Test Case 3** : **foo(1, 2, 1, 2, 1)** to bring us to 100% branch coverage(making the Boolean expression on line number 7 to False).

Decision Coverage

Line #	Predicate	True	False
3	(a == 0)	Test Case 1	Test Case 2
		foo(0, 0, 0, 0, 0)	**foo(1, 1, 1, 1, 1)**
		return 0	**return 1**
7	((a==b) OR ((c == d) AND bug(a)))	Test Case 2	
		foo(1, 1, 1, 1, 1)	
		return 1	

Condition Coverage

- **Condition coverage** reports the true or false outcome of ***each*** Boolean sub-expression of a compound predicate.
- ~~In line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a).~~
- Condition coverage measures the outcome of each of these sub-expressions independently of each other.
- With condition coverage, you ensure that each of these sub-expressions has independently been tested as both true and false.

```
1   float foo (int a, int b, int c, int d, float e)  {
2       float e;
3       if (a == 0)  {
4           return 0;
5       }
6       int x = 0;
7       if ((a==b) OR ((c == d) AND bug(a) )) {
8           x=1;
9       }
10        e = 1/x;
11      return e;
12  }
```

Condition coverage

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, x, x, 1) return value 0**	Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!**
(c==d)		Test Case 3 **foo(1, 2, 1, 2, 1) division by zero!**
bug(a)		

Condition Coverage(cont.)

- The true condition (c==d) has never been tested. Short-circuit Boolean has prevented the method bug(int) from ever being executed.
- Force c==d. **Test Case 4**: **foo(1, 2, 1, 1, 1)**, expected return value 1.
- Suppose bug(int) returns a value of true when passed a value of a=1 and returns a false value if input is any integer greater than 1.
- Force bug(a) to be false. **Test Case 5, foo(3, 2, 1, 1, 1)**, expected return value “division by zero error”.

```
1   float foo (int a, int b, int c, int d, float e)  {
2       float e;
3       if (a == 0)  {
4           return 0;
5       }
6       int x = 0;
7       if ((a==b) OR ((c == d) AND bug(a) )) {
8           x=1;
9       }
10        e = 1/x;
11      return e;
12  }
```

Condition Coverage Continued

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, x, x, 1)** **return value 0**	Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
(c==d)	Test Case 4 **foo(1, 2, 1, 1, 1)** **return value 1**	Test Case 3 **foo(1, 2, 1, 2, 1)** **division by zero!**
bug(a)	Test Case 4 **foo(1, 2, 1, 1, 1)** **return value 1**	Test Case 5 **foo(3, 2, 1, 1, 1)** **division by zero!**