# SEENG3320/6320: Software Verification and Validation

School of Electrical Engineering and Computing

Semester 1, 2020

# Software Quality Facts

- Only 32% of software projects are considered successful (full featured, on time, on budget)
- Software failures cost the US economy $59.5 billion dollars every year [NIST 2002 Report]
- On average, 1-5 bugs per KLOC (thousand lines of code)
  - In mature software (more than 10 bugs in prototypes)

Microsoft Windows 2000

- 35MLOC
- 63K known bugs at the time of release
- 2 bugs per KLOC

# Software fault examples

Intel Pentium Floating-Point Division Bug in1994:

Enter the following equation into your PC's calculator:
(4195835 / 3145727) * 3145727 – 4195835

- If the answer is zero, your computer is just fine.
- If you get anything else, you have an old Intel Pentium CPU with a floating-point division bug.
- Found by Dr. Thomas R. Nicely in 1994.

https://en.wikipedia.org/wiki/Pentium_FDIV_bug

# Software fault examples

- Mars Climate Orbiter (1998)
  - Sent to Mars to relay signal from Mars Lander
  - Smashed to the planet
- Cause: Failing to convert between different metric standards
  - Software that calculated the total impulse presented results in pound-seconds
  - The system using these results expected its inputs to be in newton-seconds

# Software fault examples

- USS Yorktown (1997)
  - Left dead in the water for 3 hours
- Cause: Divide by zero error

$$\frac{\text{Number}}{0} =$$

On 21 September 1997, a crew member entered a zero into a database field causing an attempted division by zero in the ship's Remote Data Base Manager, resulting in a buffer overflow which brought down all the machines on the network, causing the ship's propulsion system to fail.

# Software fault examples

- ATT (1990)
  - One switching system in New York City experienced an intermittent failure that caused a major service outage
  - The first major network problem in AT&T's 114-year history
- Cause: Wrong BREAK statement in C Code
  - Complete code coverage could have revealed this bug during testing

```
    case THING1:
        doit1();
        break;
case THING2:
         if (x == STUFF) {
            do_first_stuff();
             if (y == OTHER_STUFF)
                 break;
             do_later_stuff();}
/* coder meant to break to here... */
           initialize_modes_pointer();
           break;
   default:
          processing(); }
/* ...but actually broke to here! */
```

```
/* leaving the m
odes pointer
```

# Software fault examples

- Ariane 5 flight 501 (1996)
  - Destroyed 37 seconds after launch (cost: $370M)
- Cause: Arithmetic overflow
  - Data conversion from a 64-bit floating point to 16-bit signed integer value caused an exception
  - The software from Ariane 4 was re-used for Ariane 5 without re-testing

```
int tryIt () {
int i;
char string[5] = "hello";
int j;

for (i=0; i<n; i++)
    if (string[i] < j)
      j = string[i];

for (i=0; i<=5; i++)
   print(string[i]);
}
```

Any bugs
in this program?

```
for (i=0;i<10;i++) {
    if (i%3+i%2==0)
        break;
    else
        continue;
    x+=100;
    y--;
}
```

Any bugs
in this program?

```c
char *mail_auth(char *mechanism, authresponse_t resp, int argc, char *argv[])
{
    char tmp[MAILTMPLEN];
    AUTHENTICATOR *auth;
                                /* make upper case copy of mechanism name */
    ucase(strcpy(tmp, mechanism));
    for(auth = mailauthenticators; auth; auth = auth->next)
        if(auth->server && !strcmp(auth->name, tmp))
            return (*auth->server) (resp, argc, argv);
    return NIL;     /* no authenticator found */
}
```

Any bugs
in this program?

```
File file = new File("C:/robots.txt");
FileInputStream fis = null;
bool accessGranted = true;
try {
    fis = new FileInputStream(file);
    System.out.println("Total file size to read (in
bytes) : "   + fis.available());
    fis.close();
} catch (SecurityException x) {
   accessGranted = false;    // access denied
} catch (...)  {
    // something else happened
}
```

Any bugs
in this program?

# Software Verification and Validation

- Software verification and validation (V&V):
  - Verification: “Are we building the product right”.
    - The software should conform to its specification.
  - Validation: “Are we building the right product”.
    - The software should do what the user really requires.

- V&V Techniques:
  - “Formal” approaches
    - Formal verification
    - Model checking
    - …
  - “Informal” approaches
    - Code Review
    - Static Analysis
    - Software Testing
    - …

# Course Outline

# Contacts

- Course Coordinator: A/Prof. Hongyu Zhang
- Office: ES233
- Email: Hongyu.zhang@newcastle.edu.au
- URL: https://www.newcastle.edu.au/profile/hongyu-zhang

- Tutor: Fareed Fareed-Ud-Din
- Email: ffu388@newcastle.edu.au
- Office: ES223

# Study Timetable

- **Lectures:** Monday 1pm-3pm at EA101
- **Workshop:**
  - Monday 10am-12pm at CT314
  - Thursday 6pm-8pm at CT220
  - Starting from Week 2.
  - The workshop sessions will be organized by Tutor.

# Lecture Topics

- Software testing
  - Black box testing
    - Equivalence Partitioning/Boundary-Value Analysis
    - Random testing/fuzz testing
    - Combinatorial testing
  - White box testing
    - Test adequacy and coverage criteria
    - Symbolic execution
  - Automated testing tools and techniques
  - Testing lifecycle and test management
  - Non-functional testing
- Code review and inspection
- Formal methods for software verification

# Assignment

- Assignment 1 (25%)
- Assignment 2 (25%)
- Final examination (50%)

# Software Testing

“Software testing is the process of executing a program or a system with the intent of finding errors”

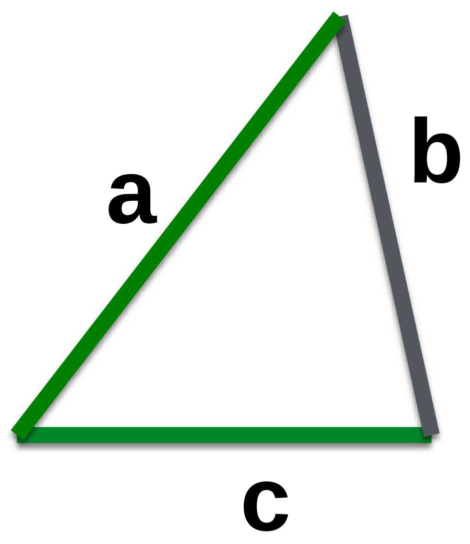
--- “The art of software testing” by G. J. Myers

# Terms

- **Fault**: incorrect portions of code (may involve missing code as well as incorrect code)
- **Failure**: observable incorrect behavior of a program.
- **Error**: cause of a fault: something bad a programmer did (conceptual, typo, etc)
- **Bug**: informal term for fault

# Test Cases

- Test Case: a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement




Test Case

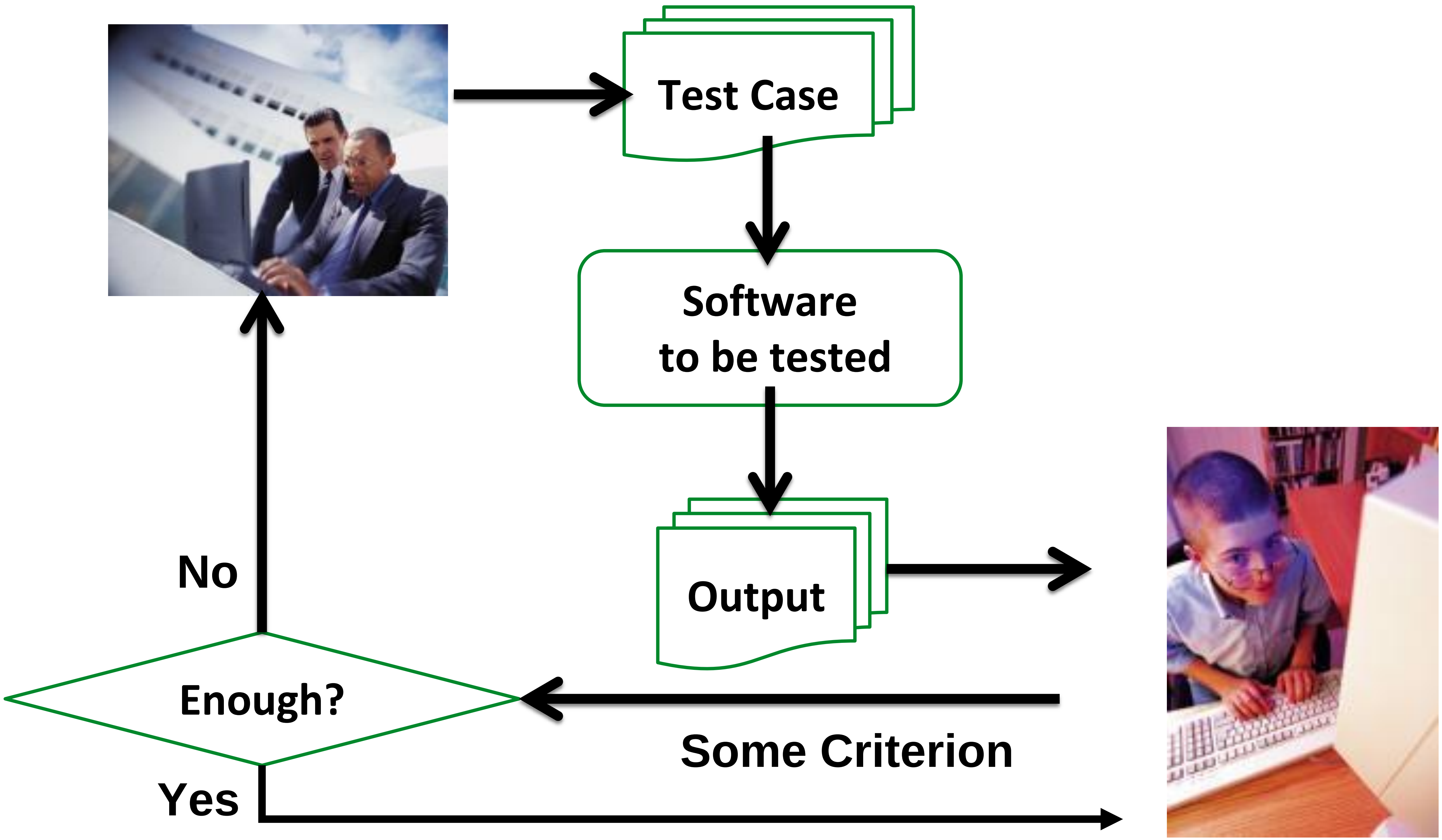
Test Input: a=3, b=4, c=5
Expected Result: right triangle

Test Case

Objective: Gmail-compose mail-able to edit the email
Test Input: 1. Click on compose 2. Type some text in the message box
Conditions: User should log in Gmail
Expected Result: User is able to type in the message box

# A Typical Software Testing Process

**Test case generation**



# Testing v.s. Debugging

- Testing: Finding inputs that cause the failure of a software
  - Failure is unknown.
  - Performed by “Testers”.

- Debugging: The process of finding and fixing a fault given a failure
  - Failure is known.
  - Performed by “Developers”.

# Exhaustive Testing is Hard /1

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return x;
}
```

- Number of possible test cases (assuming 32 bit integers)

  $$2^{32} \times 2^{32} = 2^{64}$$

- Test suite {(x=3,y=2),(x=4,y=3),(x=5,y=1)}

will not detect the error

- Test suite {(x=3,y=2), (x=2,y=3)}

will detect the error

- The power of the test suite is not determined by the number of test cases

# Exhaustive Testing is Hard /2

- Assume that the input for the `max` procedure was an integer array of size *n*
  - Number of test cases: $2^{32\times n}$
- Assume that the size of the input array is not bounded
  - Number of test cases: $\infty$
- The point is, naive exhaustive testing is pretty hopeless

# Testing Techniques

- Functional (Black box) vs. Structural (White box) testing
  - Functional testing: Generating test cases based on the functionality of the software
  - Structural testing: Generating test cases based on the structure of the program
  - Black box testing and white box testing are synonyms for functional and structural testing, respectively.
    - In black box testing the internal structure of the program is hidden from the testing process
    - In white box testing internal structure of the program is taken into account

# Black-Box Testing

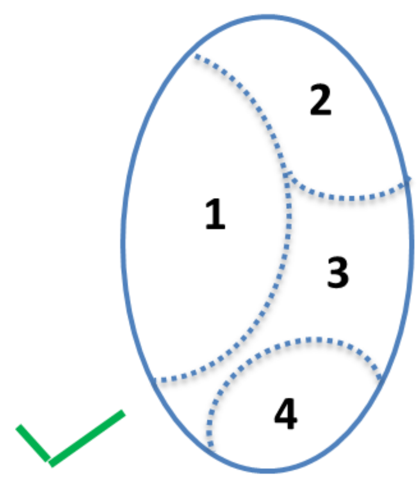
# Black-Box Testing

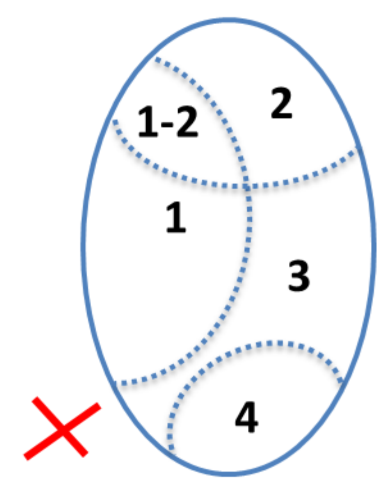
- Identify functions and design test cases that will check whether these functions are correctly performed by the software
  - Formal specifications of the functions
  - Informal specifications (*more popular in industry*)
- Techniques:
  - Equivalence Partitioning
  - Boundary-Value Analysis

# Equivalence Partitioning

- Divide the input domain into **equivalence partitions**
- Select one test case from each equivalence partition
- Characteristics of equivalence partitions
  - Disjointedness
    - No input belongs to more than one partition
  - Coverage
    - The input domain is covered by the entire collection of equivalence partitions

Every possible test input belongs to
<u>one and only one</u> partition



# Equivalence Partitioning (cont.)

- Let’s test a method “isEven(int n)”, which

  returns “true” for all even Inputs

  returns “false” for all odd Inputs

  , where $1000 \geq n \geq 1$

- We can create two equivalent partitions, i.e. even numbers and odd numbers between 1 and 1000

# Equivalence Partitioning Example

Suppose a Windows application requires a password, which has minimum 8 characters and maximum 12 characters.

Design test cases for the password length checking program.

# Equivalence Partitioning Example

| Invalid Partition | Valid Partition | Invalid Partition |
|---|---|---|
| Less than 8 | 8 - 12 | More than 12 |

# Boundary-Value Analysis (BVA)

Instead of selecting any element in an equivalence partition, inputs <u>close to</u> the boundaries of the equivalence classes are selected as test cases

1. Partition the input domain. This leads to as many partitions as there are input variables. We will generate several sub-domains in this step.
2. Identify the boundaries for each partition. Boundaries may also be identified using special relationships amongst the inputs.
3. Select test data such that each boundary value occurs in at least one test input.


t1: 7
t2: 8
t3: 9
t4: 12
t5: 13

# BVA: Example - Create equivalence classes

Assuming that an item code must be in the range 99..999 and quantity in the range 1..100,

Equivalence classes for code:

E1: Values less than 99.

E2: Values in the range.

E3: Values greater than 999.

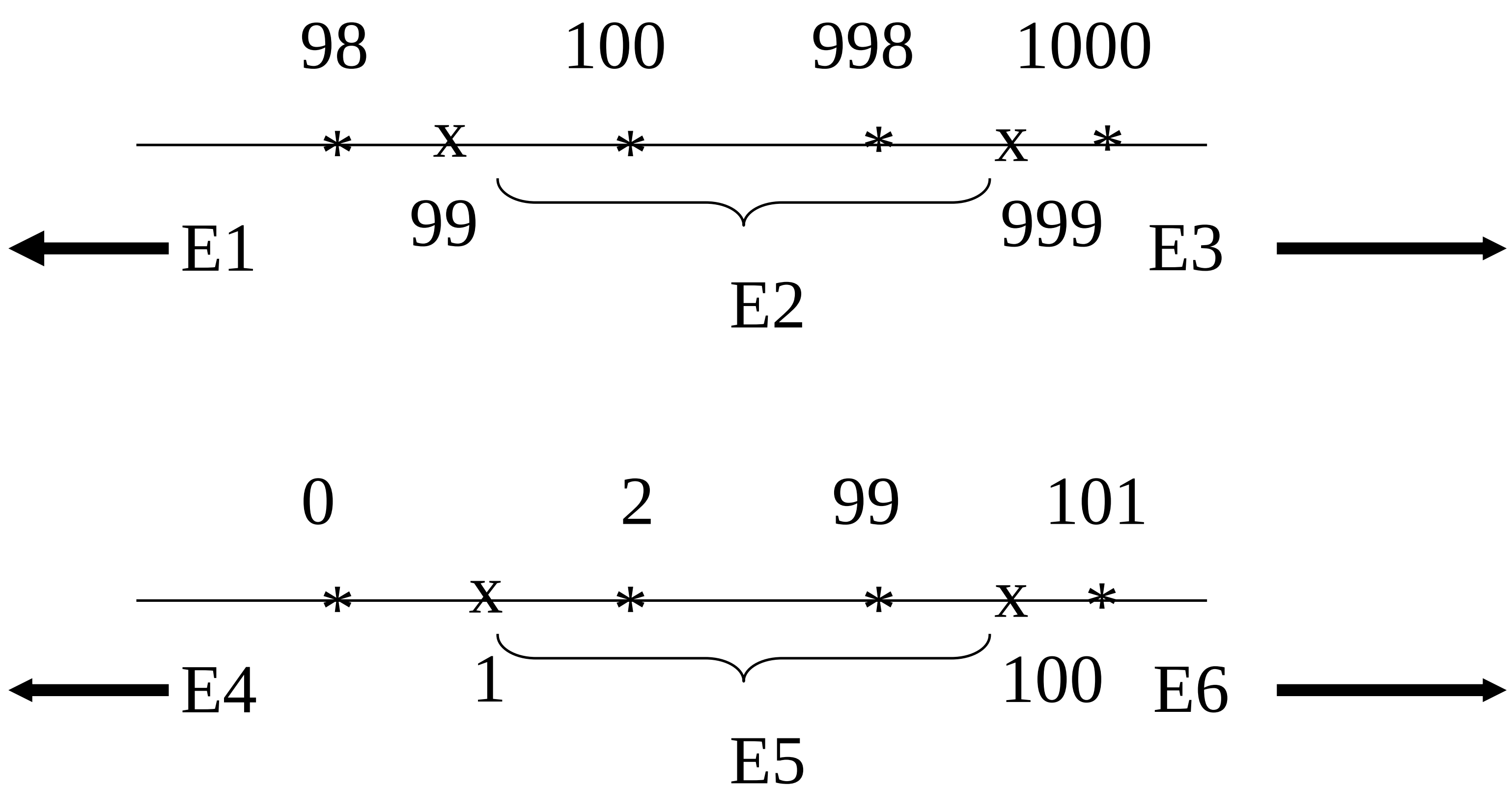
Equivalence classes for qty:

E4: Values less than 1.

E5: Values in the range.

E6: Values greater than 100.

# BVA: Example - Identify boundaries




Equivalence classes and boundaries

Boundaries are indicated with an x. Points near the boundary are marked *.

# BVA: Example - Construct test set

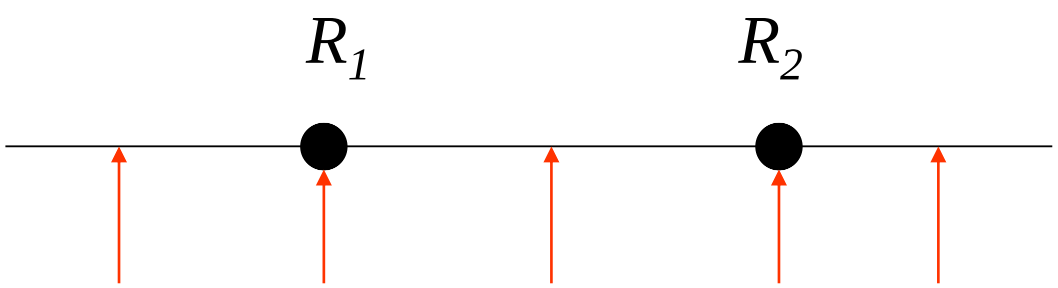
Test selection based on the boundary value analysis technique requires that tests must include, for each variable, values at and around the boundary.

For example:

t1: (code=98, qty=0),
t2: (code=99, qty=1),
t3: (code=100, qty=2),
t4: (code=998, qty=99),
t5: (code=999, qty=100),
t6: (code=1000, qty=101)

# Testing Boundary Conditions

- For each range [$R_1$, $R_2$] listed in either the input or output specifications, choose five cases:
  - Values less than $R_1$
  - Values equal to $R_1$
  - Values greater than $R_1$ but less than $R_2$
  - Values equal to $R_2$
  - Values greater than $R_2$
- For unordered sets select two values
  - 1) in, 2) not in
- For equality select 2 values
  - 1) equal, 2) not equal
- For sets, lists select two cases
  - 1) empty, 2) not empty



# Quiz

Suppose a Windows application requires users to enter their Date of Birth (in the form of DD/MM/YYYY)

Design test cases for checking the validity of a Date of Birth.


28/03/2016

1/1/1971

10/20/1989

...

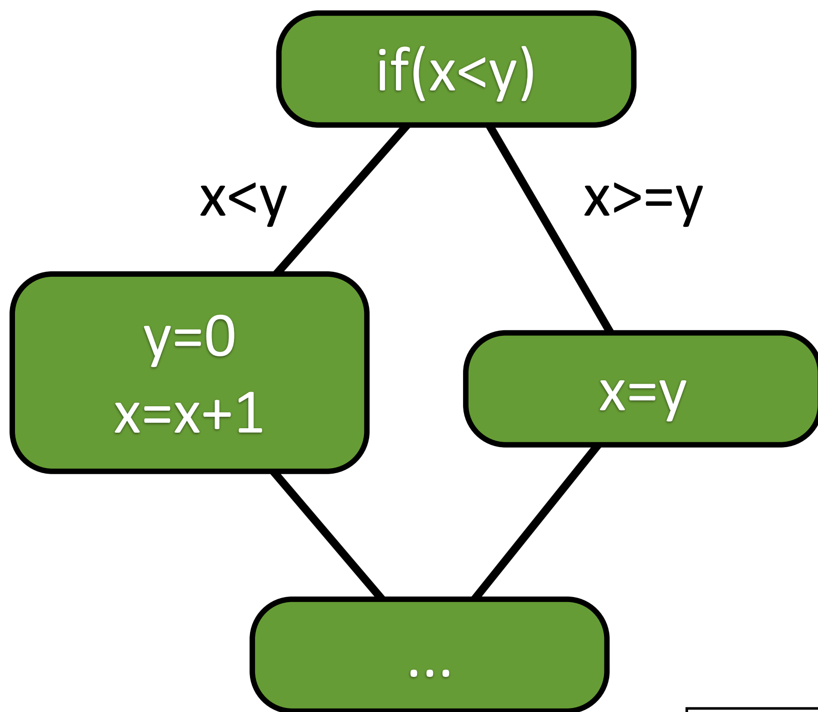
# White-Box Testing

# White Box testing

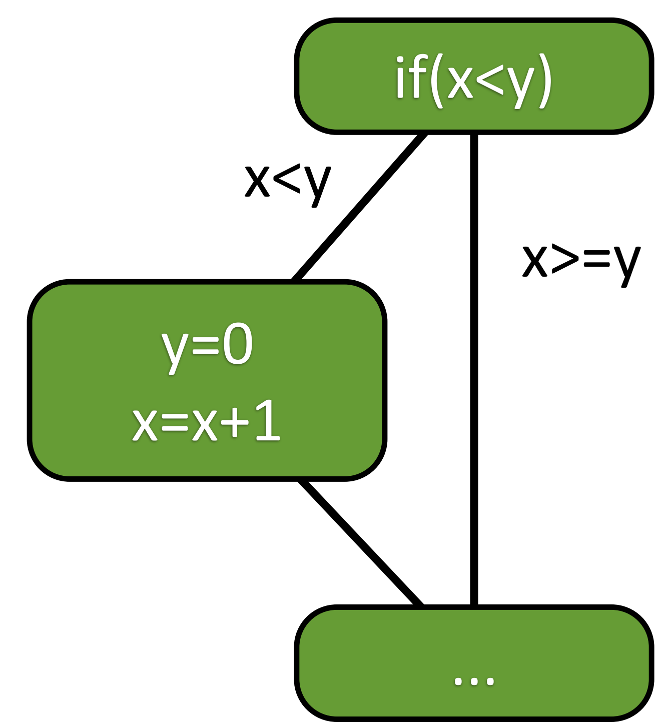
- White box testing (structural testing):
  - Generating test cases based on the structure of the program
  - A common way is to abstract program into control flow graph (CFG)
    - Node : Sequences of statements (basic block)
    - Edge : Transfers of control
- Coverage metrics:
  - *Statement coverage*: all statements in the programs should be executed at least once
  - *Branch coverage*: all branches in the program should be executed at least once
  - *Path coverage*: all execution paths in the program should be executed at lest once

# CFG : The if statement

```
if (x < y)
{
   y = 0;
   x = x + 1;
}
else
{
   x = y;
}
...
```

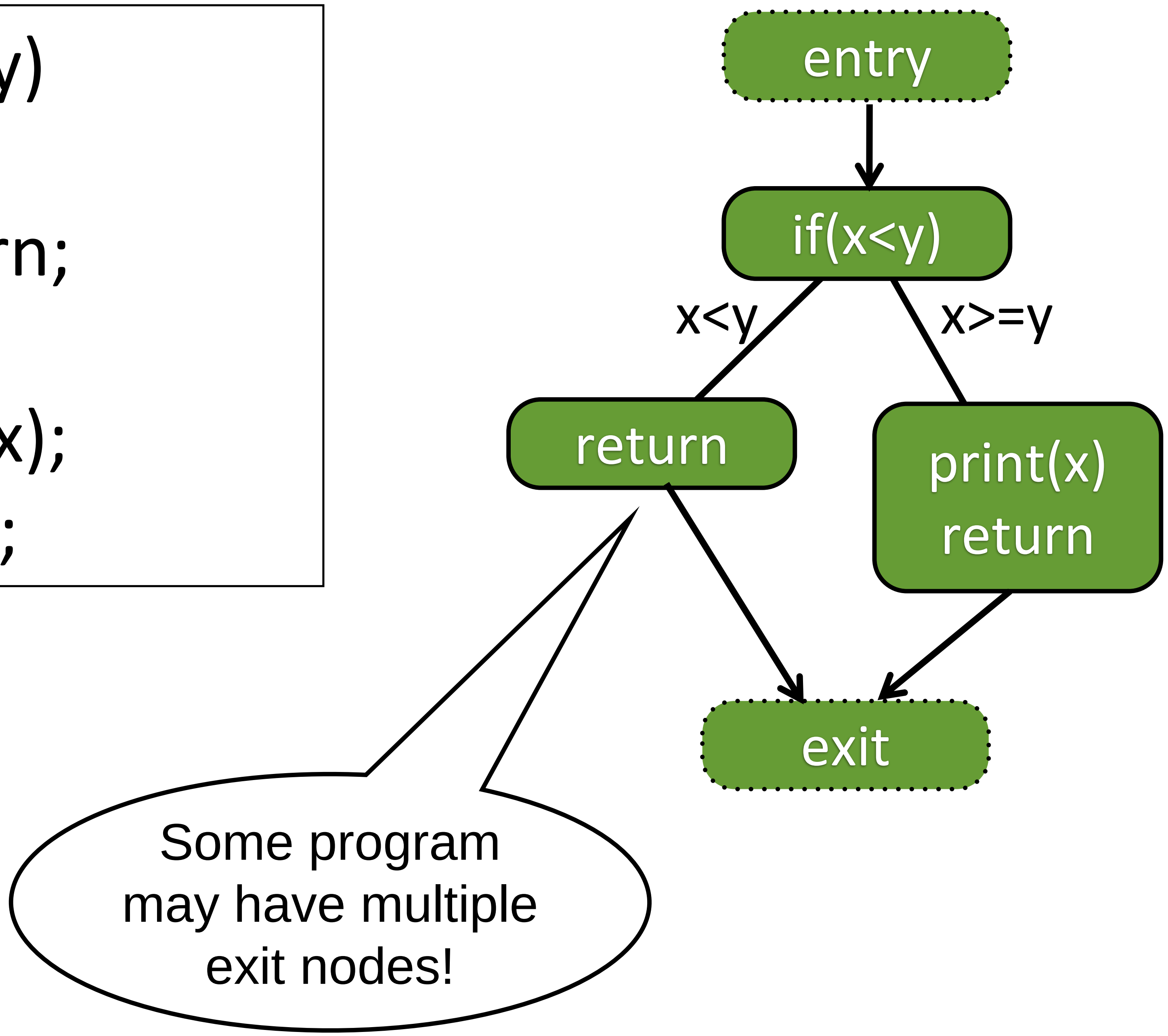



```
if (x < y)
{
   y = 0;
   x = x + 1;
}
...
```



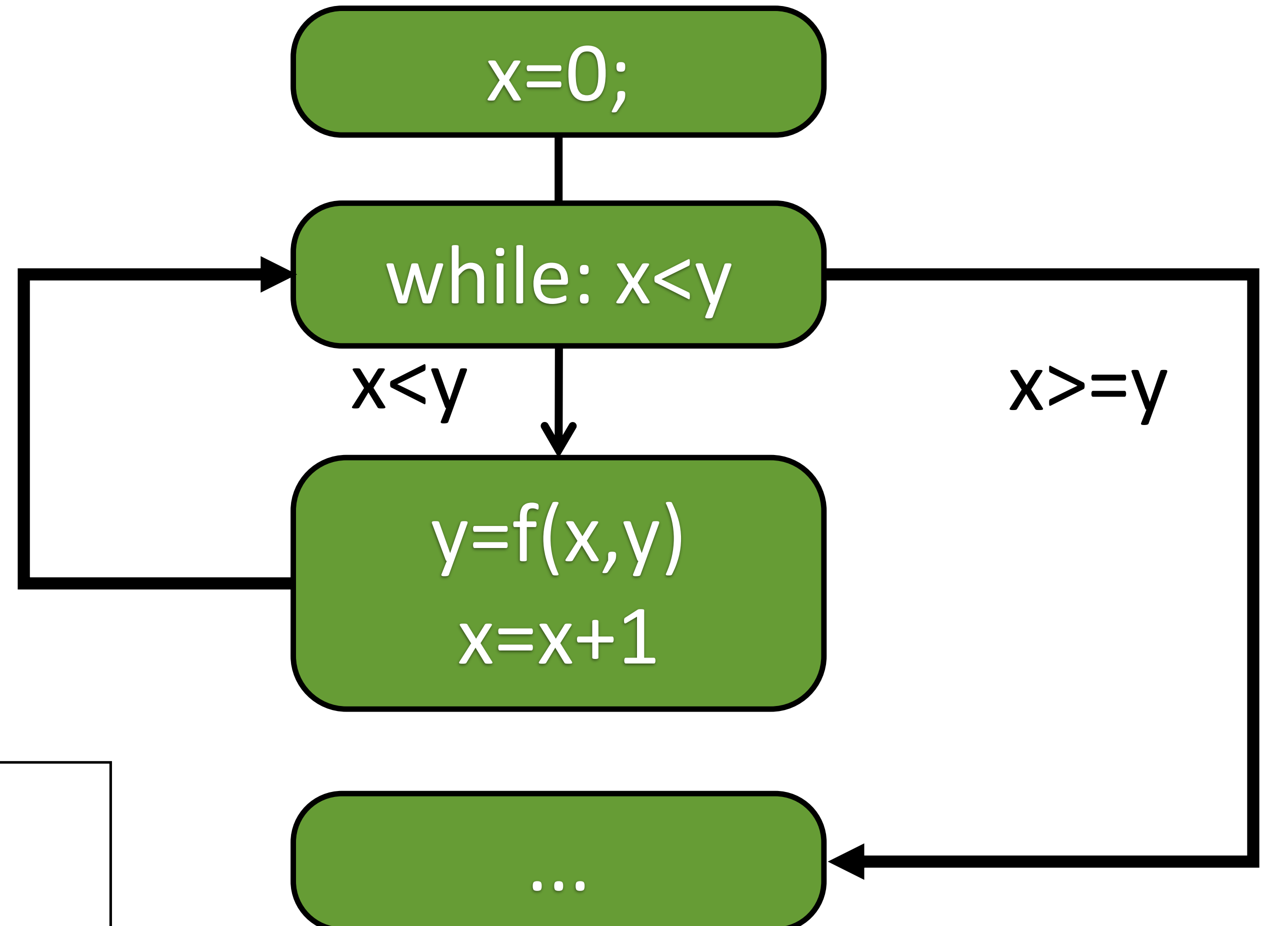
# CFG : The dummy nodes

```
if (x < y)
{
  return;
}
print (x);
return;
```



# CFG : while and for loops

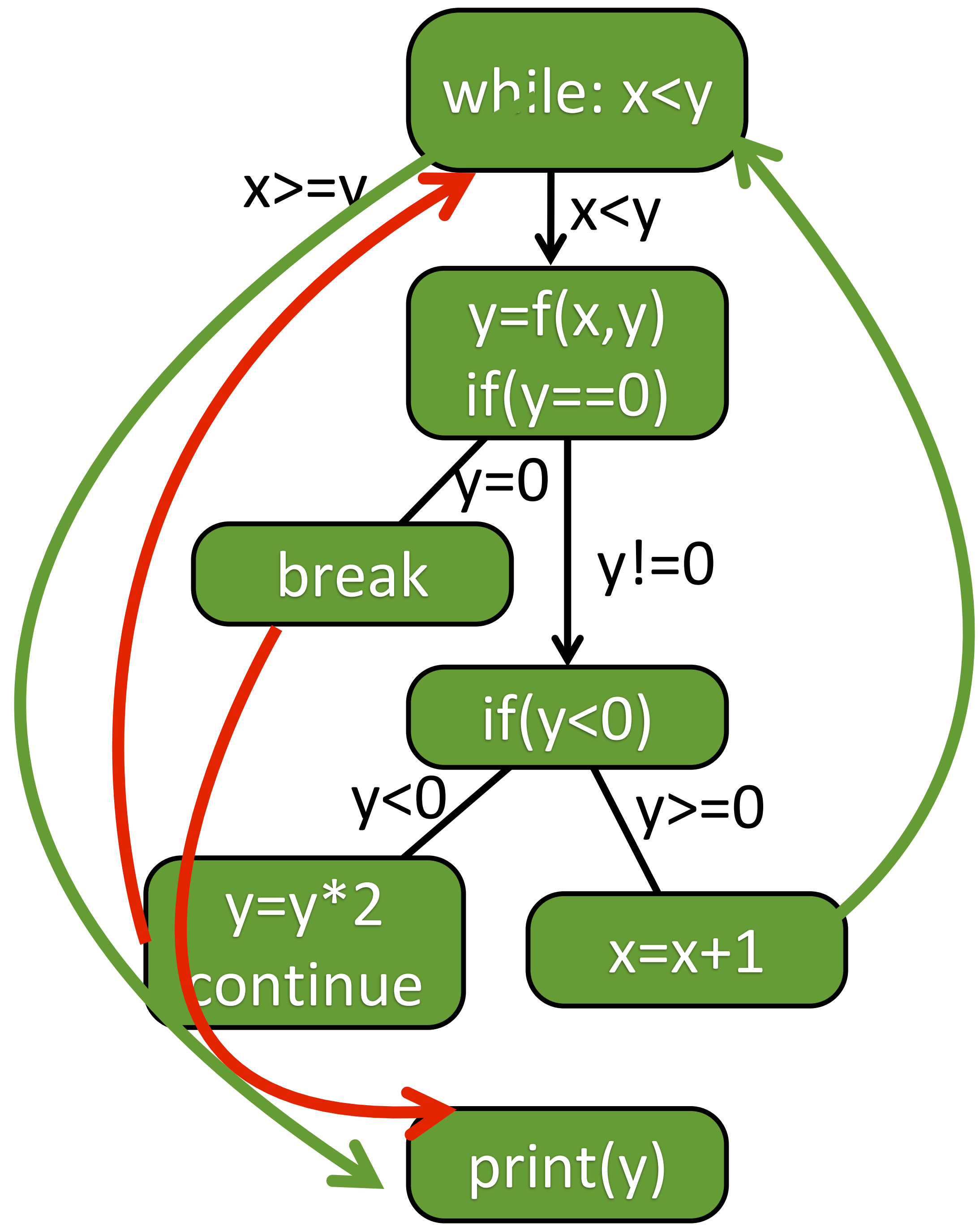
```
x=0;
while (x < y)
{
   y = f (x, y);
   x = x + 1;
}
...
```




```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```

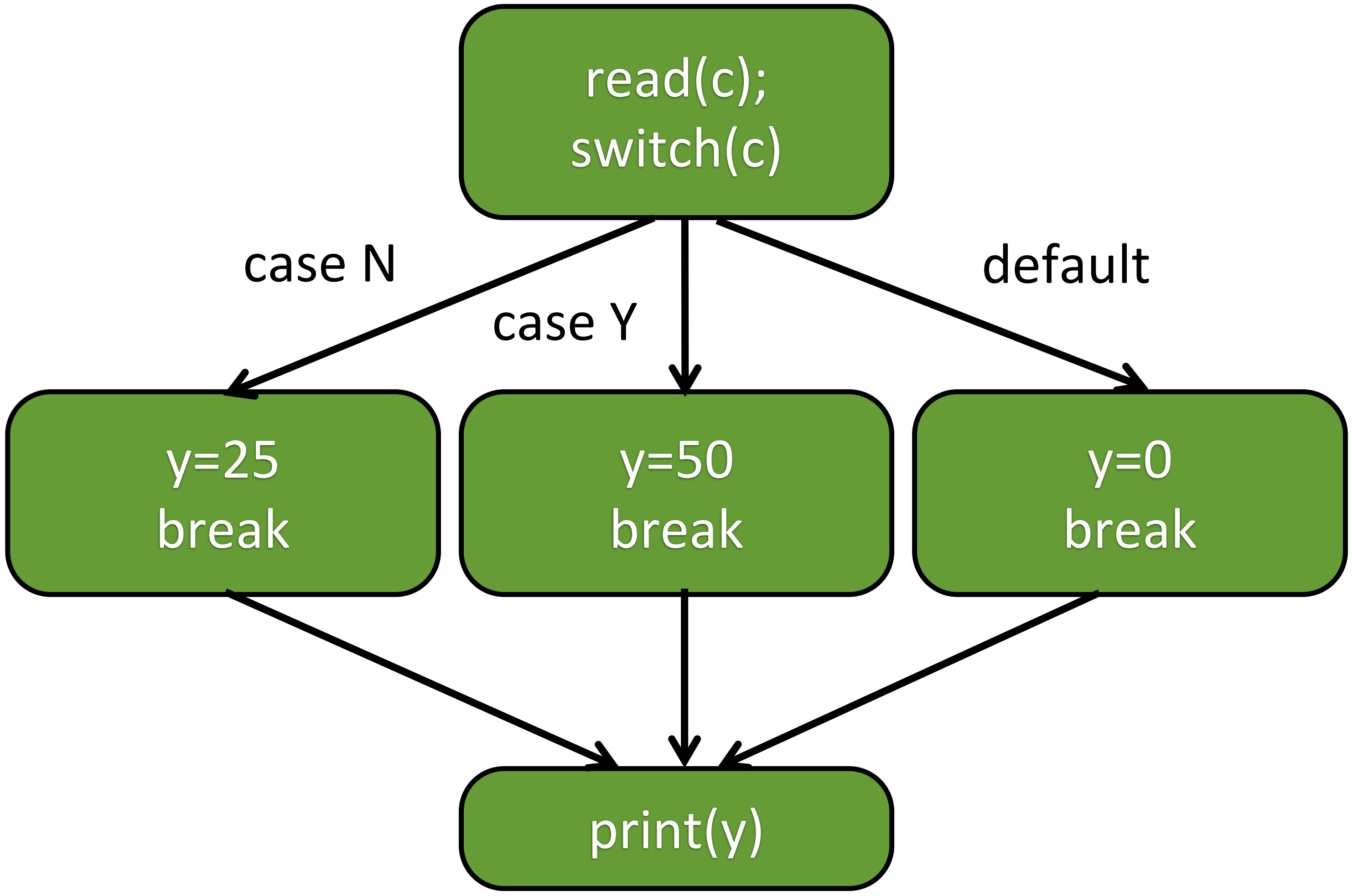
# CFG: break and continue

```
while (x < y)
{
  y = f (x, y);
  if (y == 0) {
    break;
  } else if (y<0) {
    y = y*2;
    continue;
  }
  x = x + 1;
}
print (y);
```



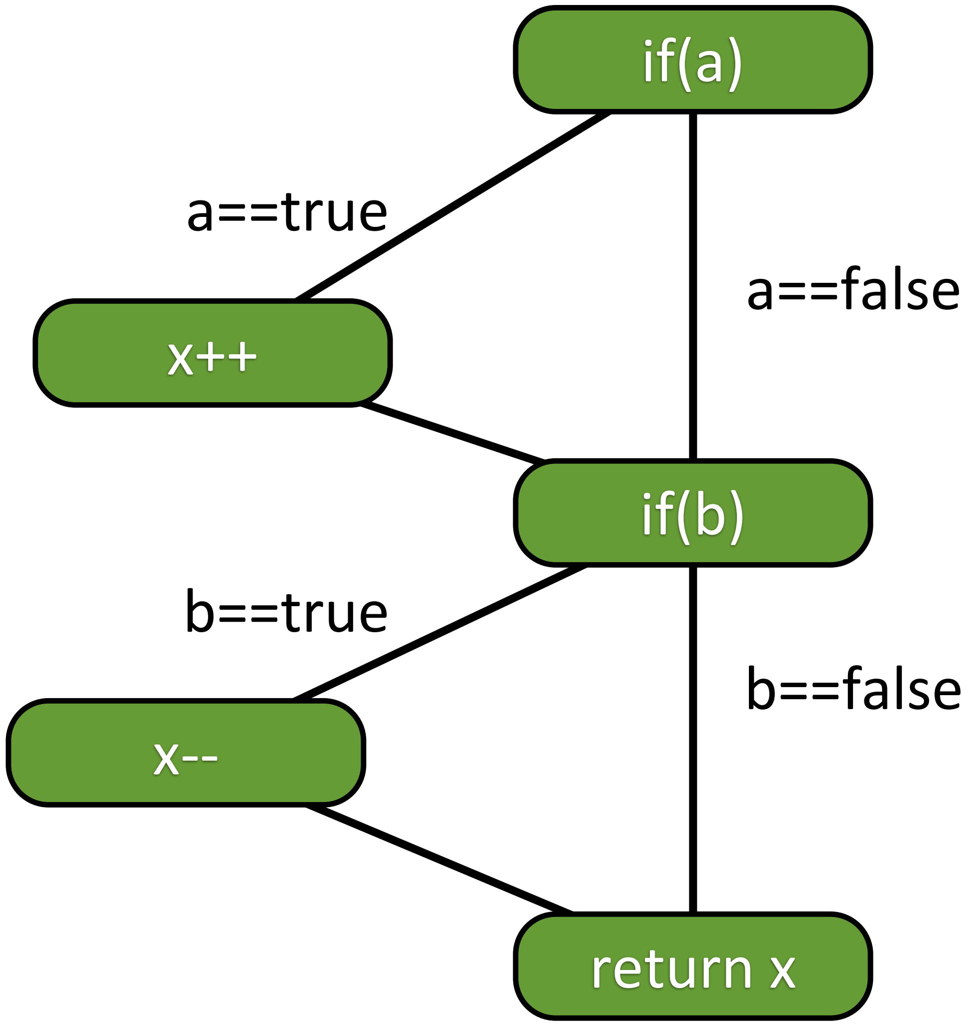
# CFG: switch

```
read ( c) ;
switch ( c )
{
  case ‘N’:
    y = 25;
    break;
  case ‘Y’:
    y = 50;
    break;
  default:
    y = 0;
    break;
}
print (y);
```



# CFG-based coverage: example

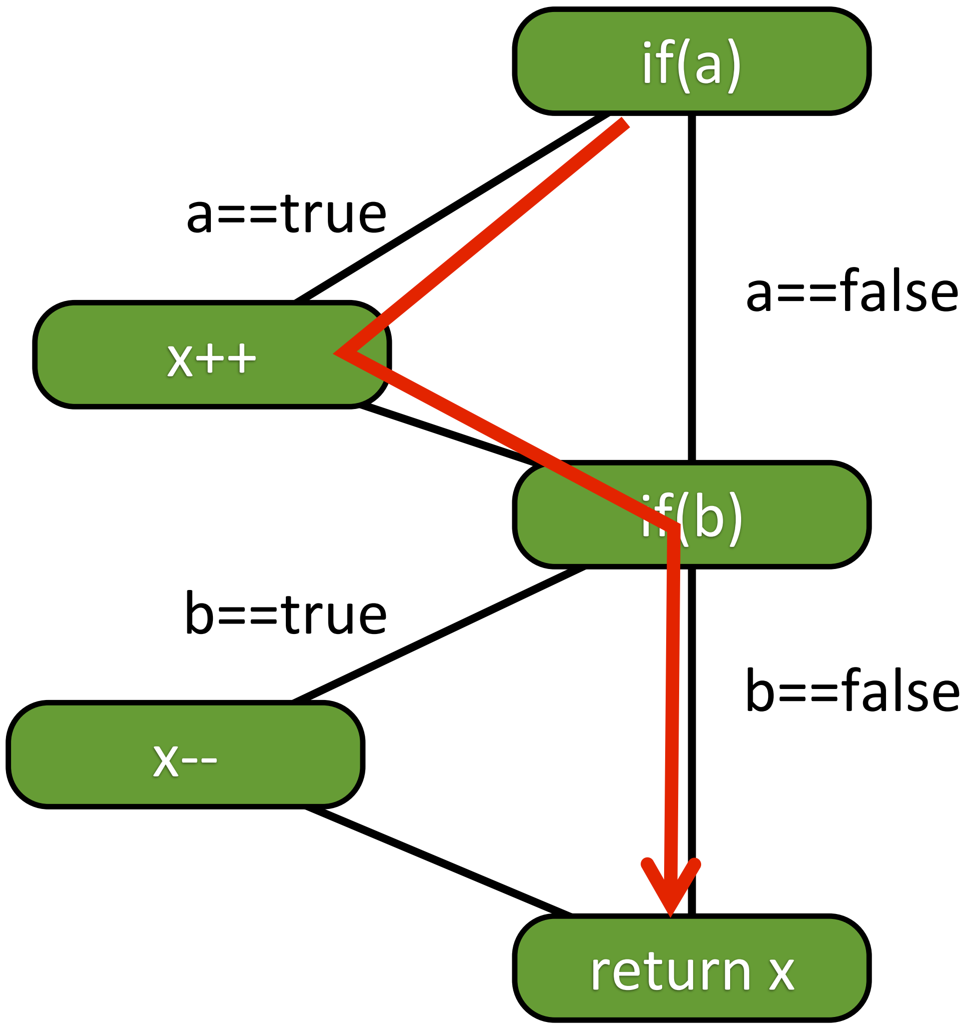
```
public class
CFGCoverageExample {
    public int testMe(int x,
boolean a, boolean b){
        if(a)
            x++;
        if(b)
            x--;
        return x;
    }
}
```



# CFG-based coverage: statement coverage

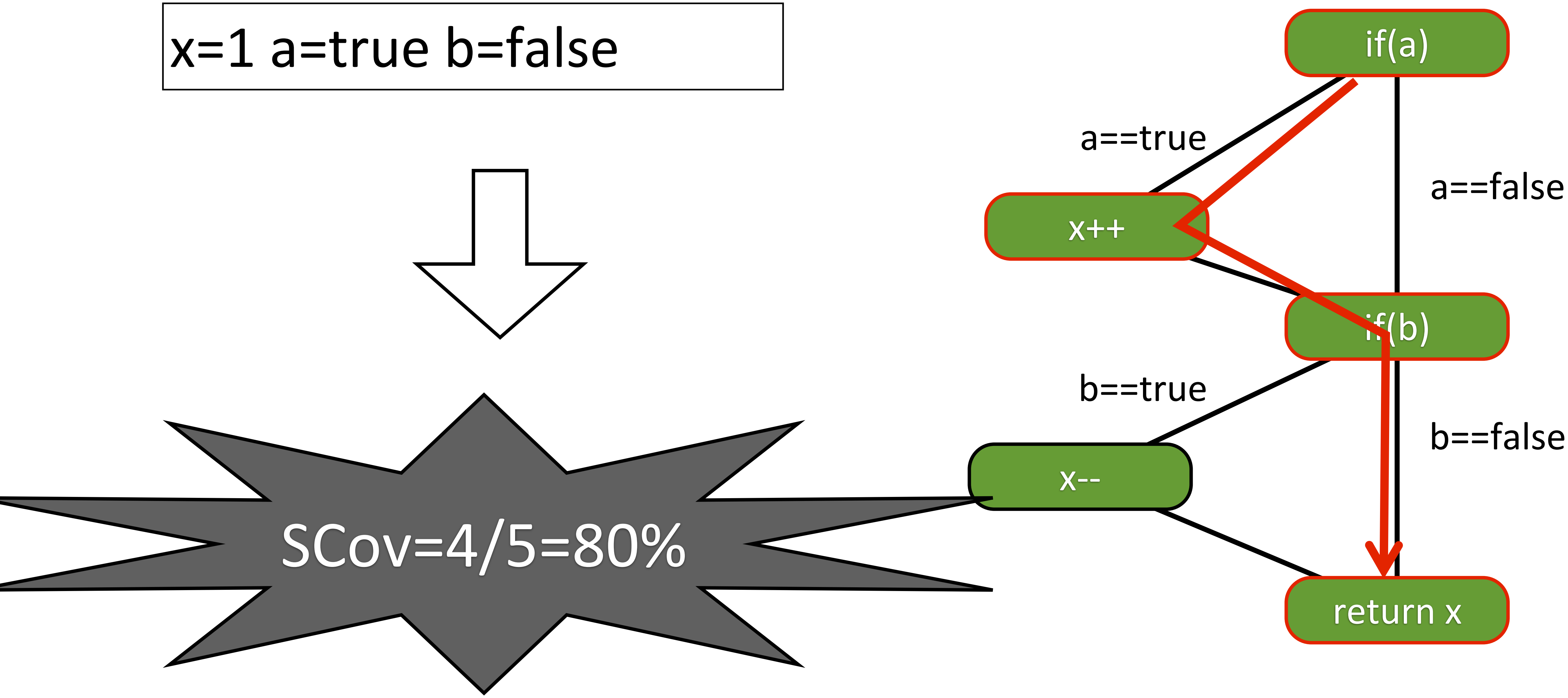
- The percentage of statements covered by the test

x=1 a=true b=false



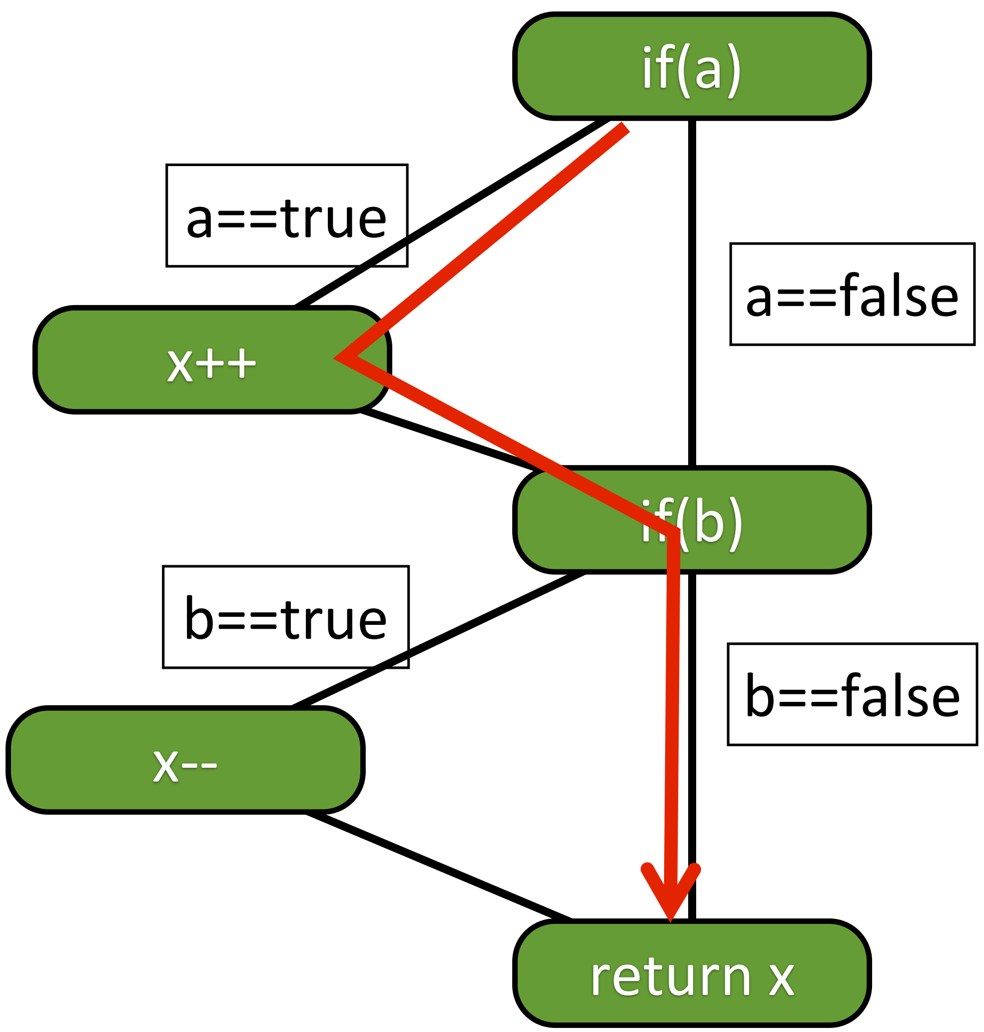
# CFG-based coverage: statement coverage

- The percentage of statements covered by the test



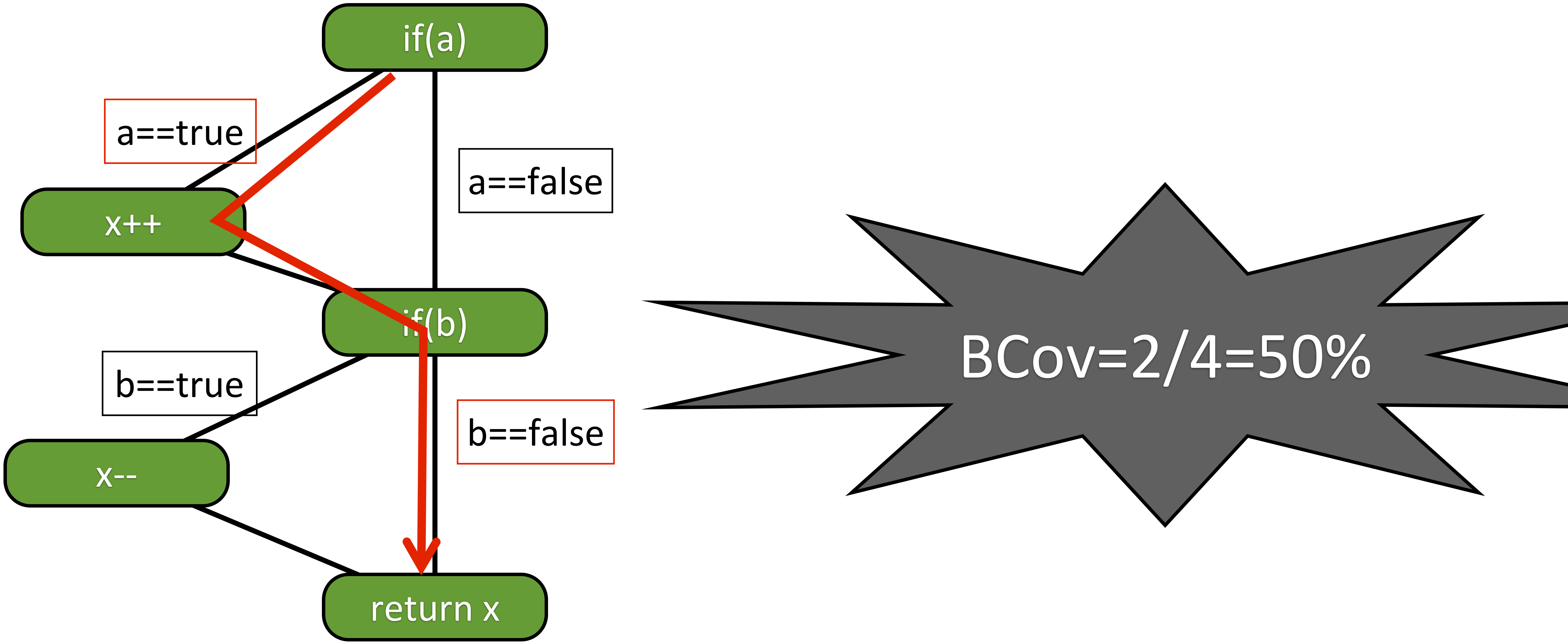
# CFG-based coverage: branch coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



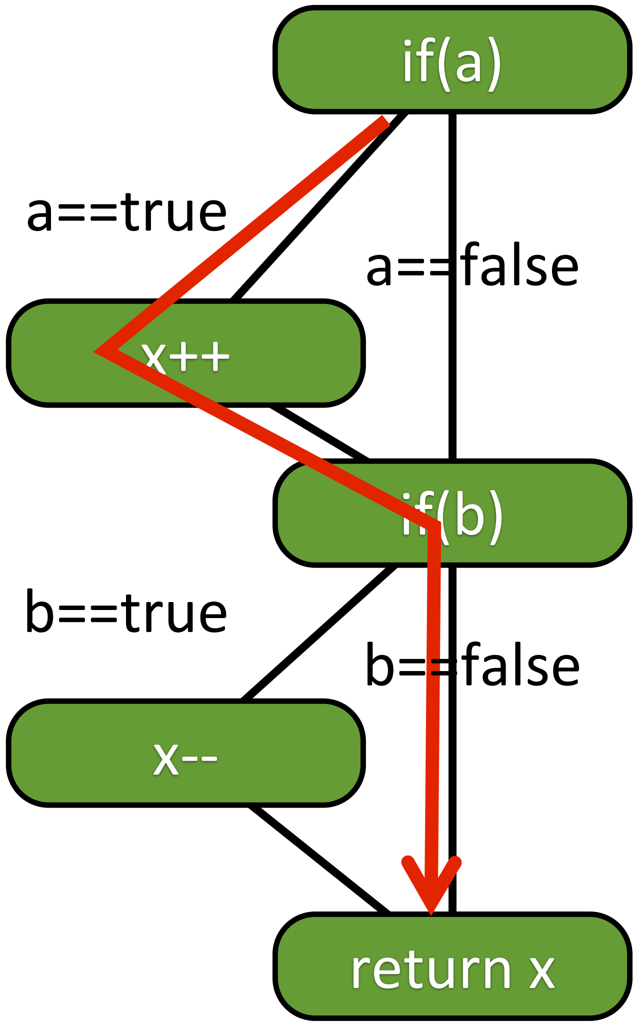
# CFG-based coverage: branch coverage

- The percentage of branches covered by the test
  - Consider both false and true branch for each conditional statement



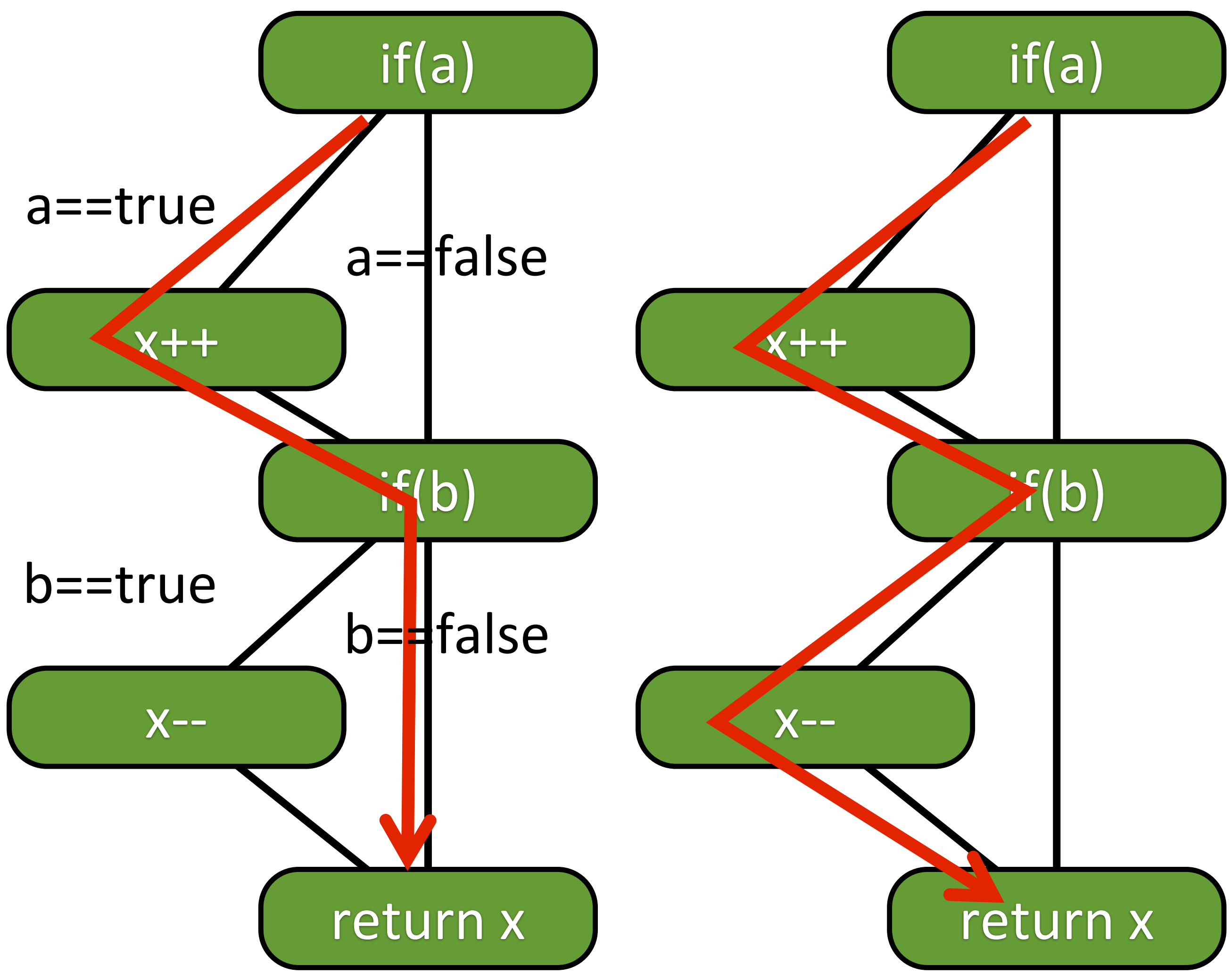
# CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



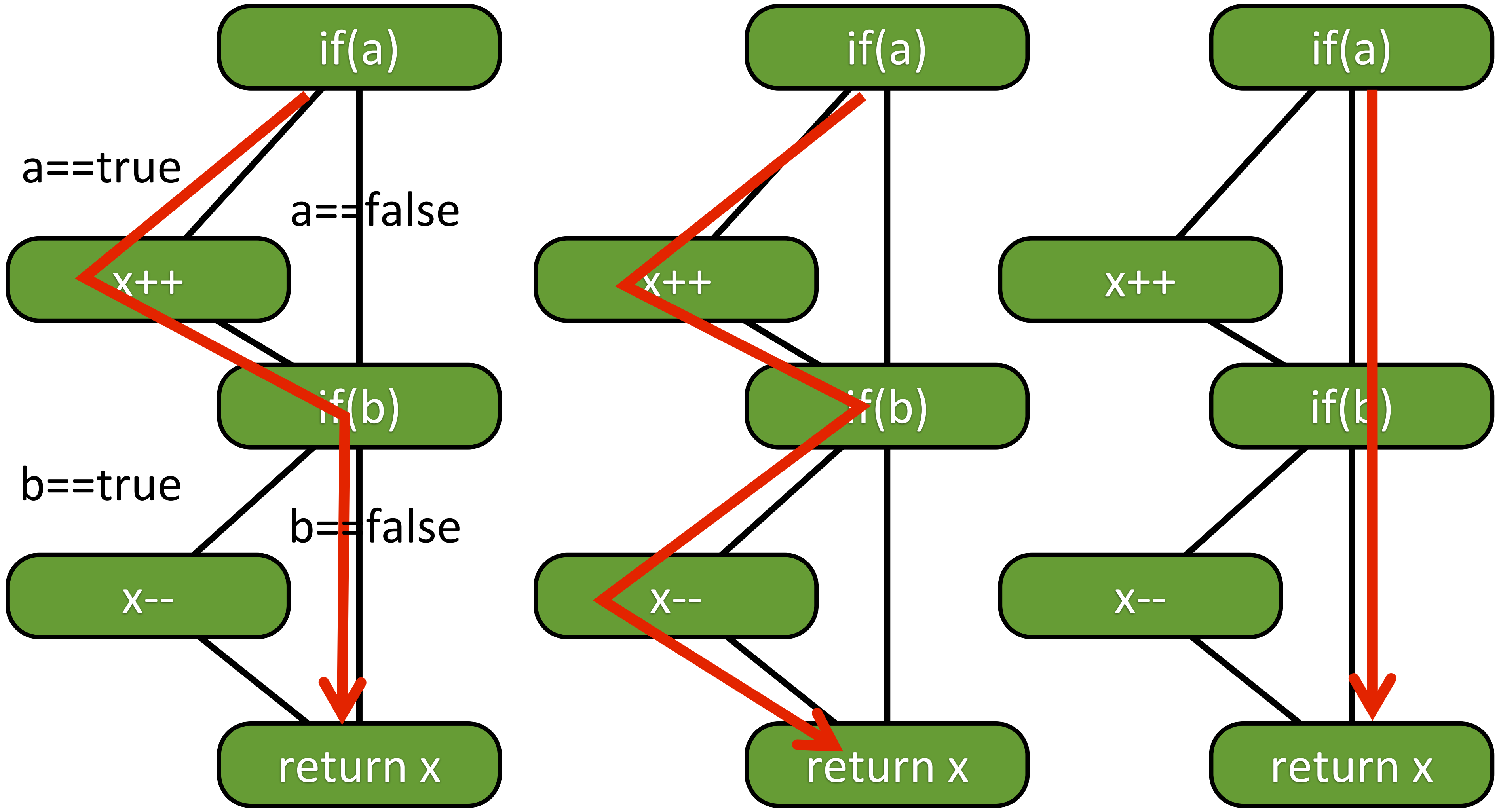
# CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



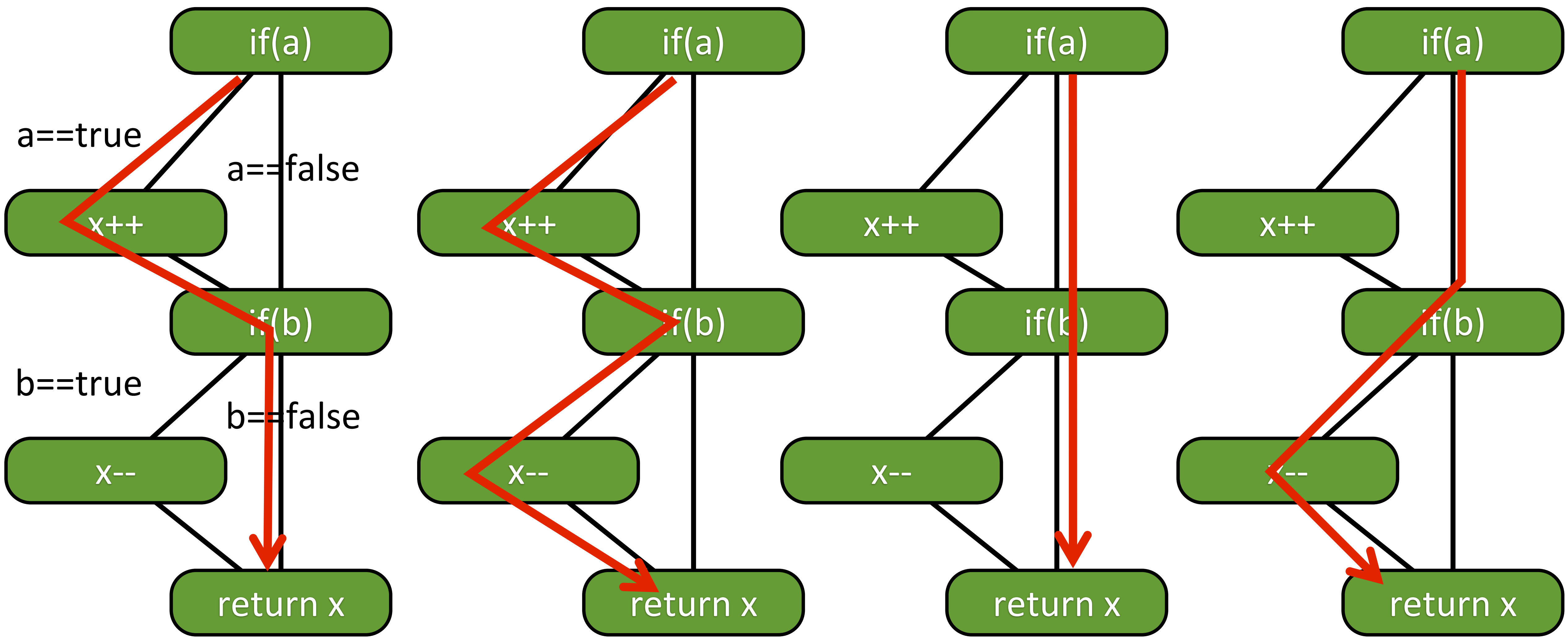
# CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths



# CFG-based coverage: path coverage

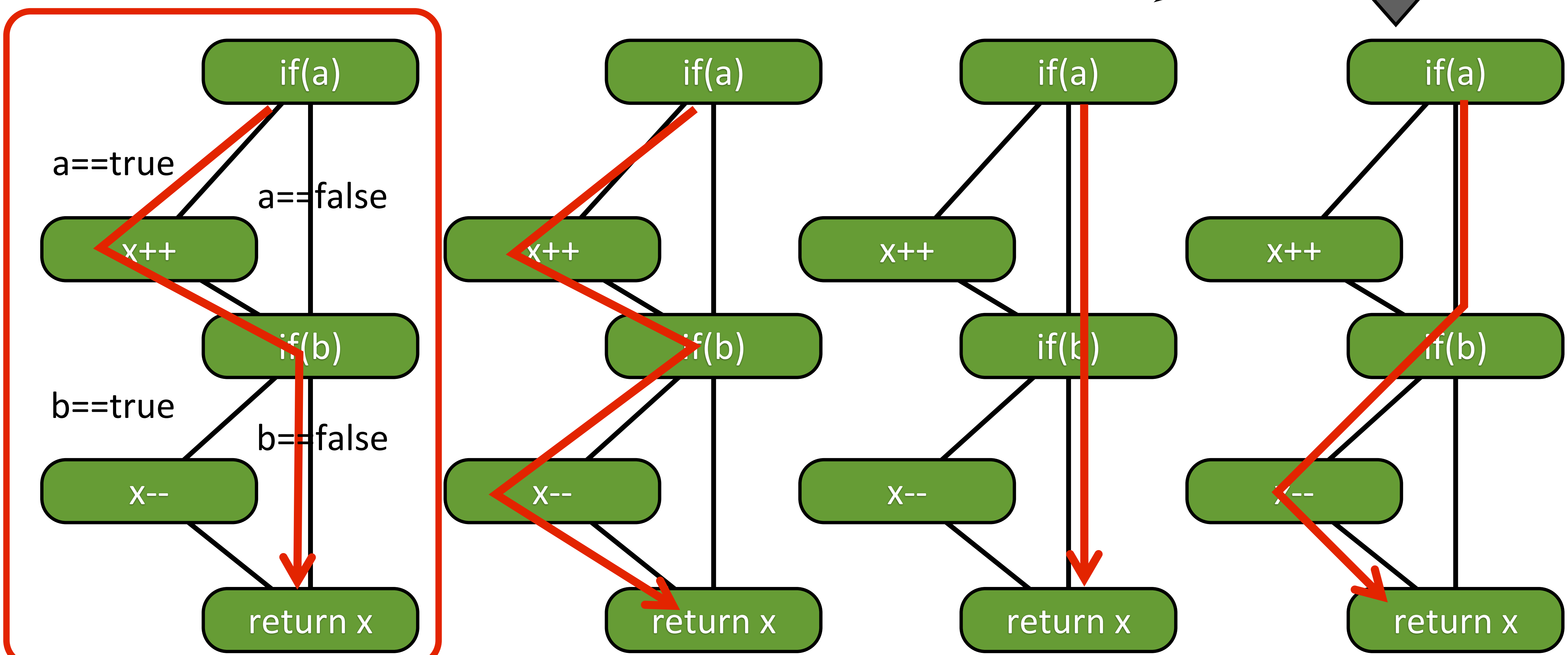
- The percentage of paths covered by the test
  - Consider all possible program execution paths



# CFG-based coverage: path coverage

- The percentage of paths covered by the test
  - Consider all possible program execution paths

PCov=1/4= 25%



# CFG-based coverage

```
public class
CFGCoverageExample {
    public int testMe(int x,
boolean a, boolean b){
        if(a)
            x++;
        if(b)
            x--;
        return x;
    }
}
```

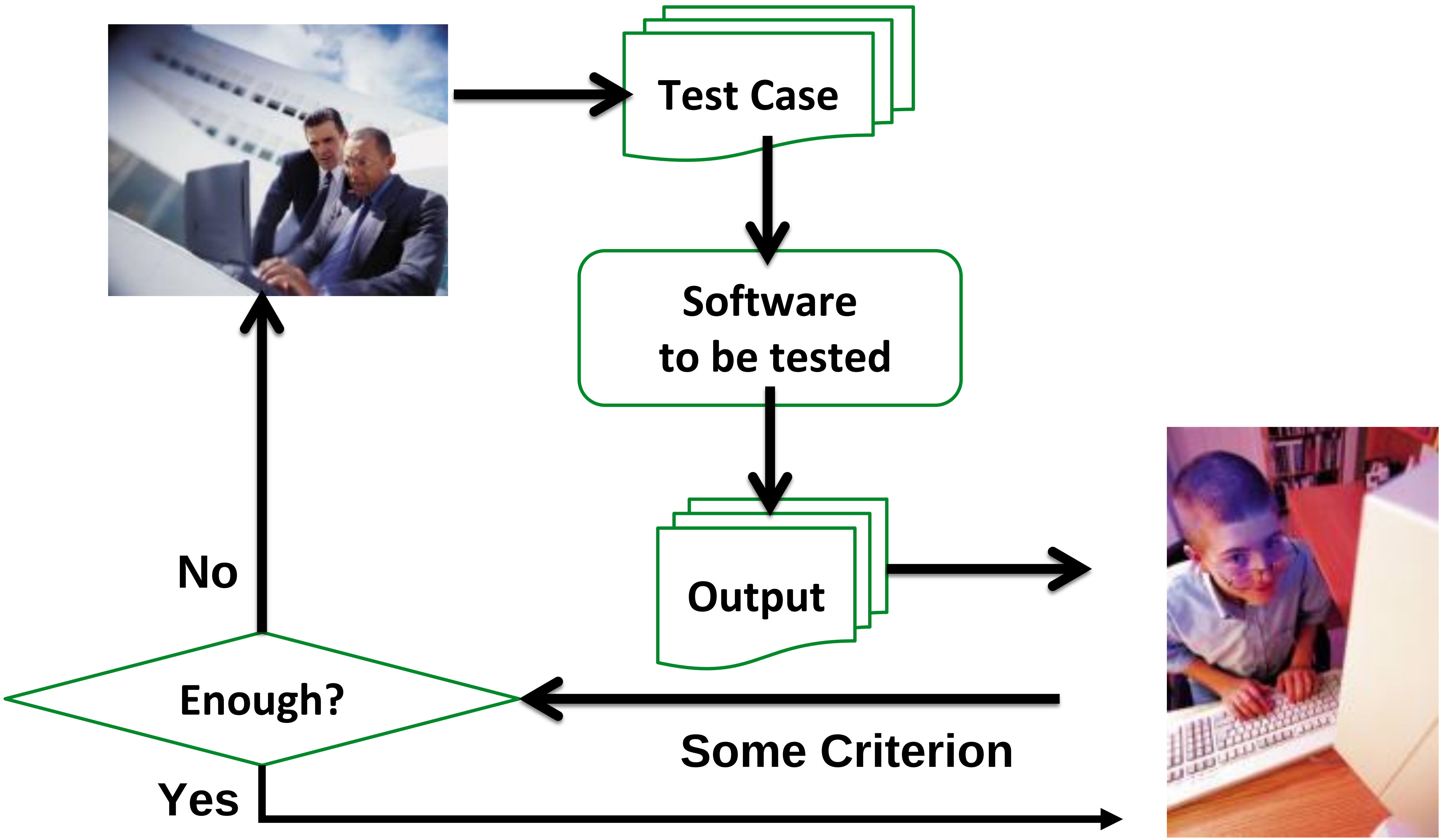
- Code coverage metrics can be used to measure test adequacy
- Usually, a test covering/executing more code may indicate better test quality

How good is it??

Test case: (1, true, false)

# A Typical Software Testing Process

**Test case generation**



# CFG-based coverage: comparison

```
public class CFGCoverageExample {
	public int testMe(int x, boolean a, boolean
b){
		if(a)
			x++;
		if(b)
			x--;
		return x;
	}
}
```

Test case: (0, true, false)

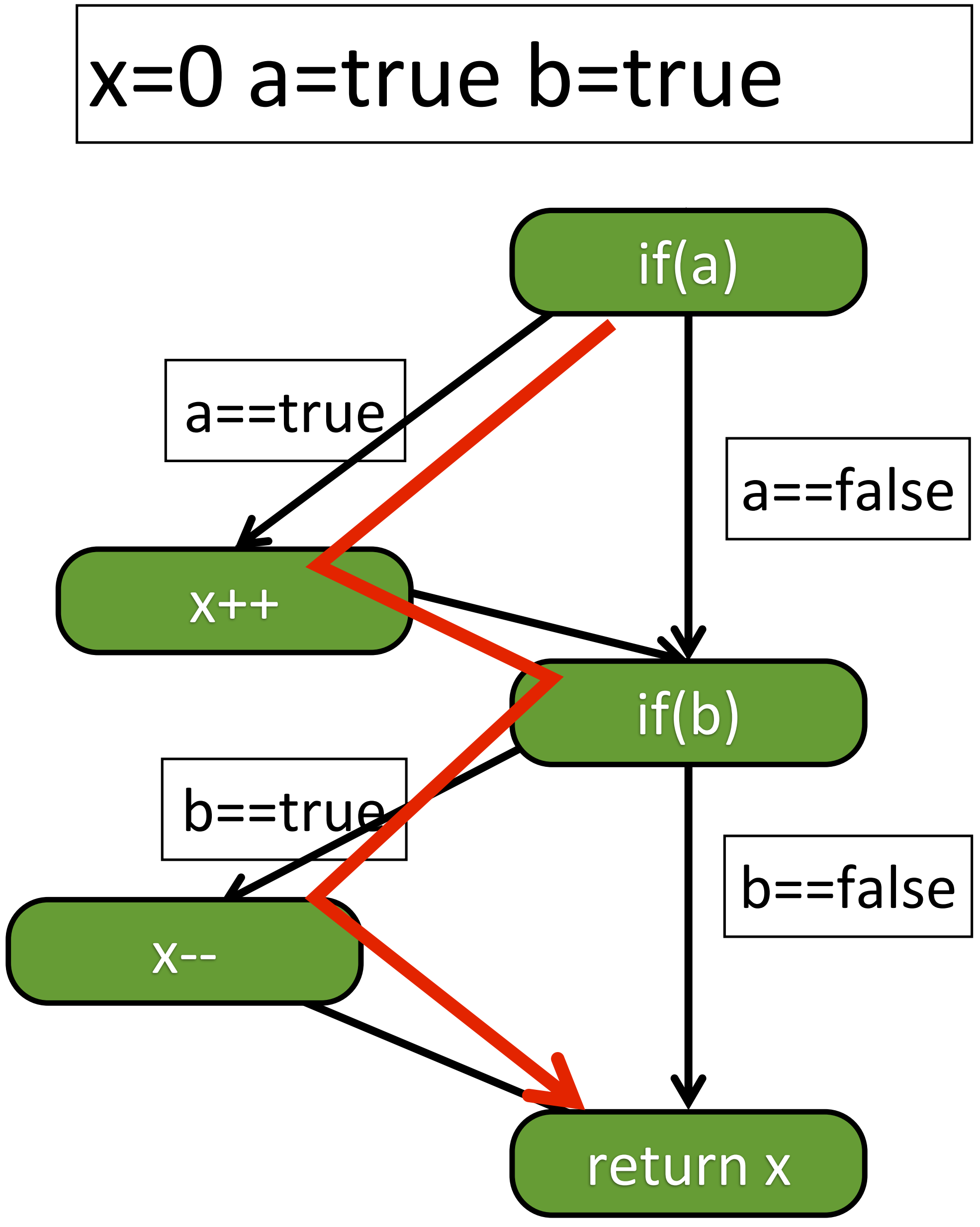
Statement coverage: 80%
Branch coverage: 50%
Path coverage: 25%

If we achieve 100% branch coverage, do we get 100% statement coverage for free?

If we achieve 100% path coverage, do we get 100% branch coverage for free?

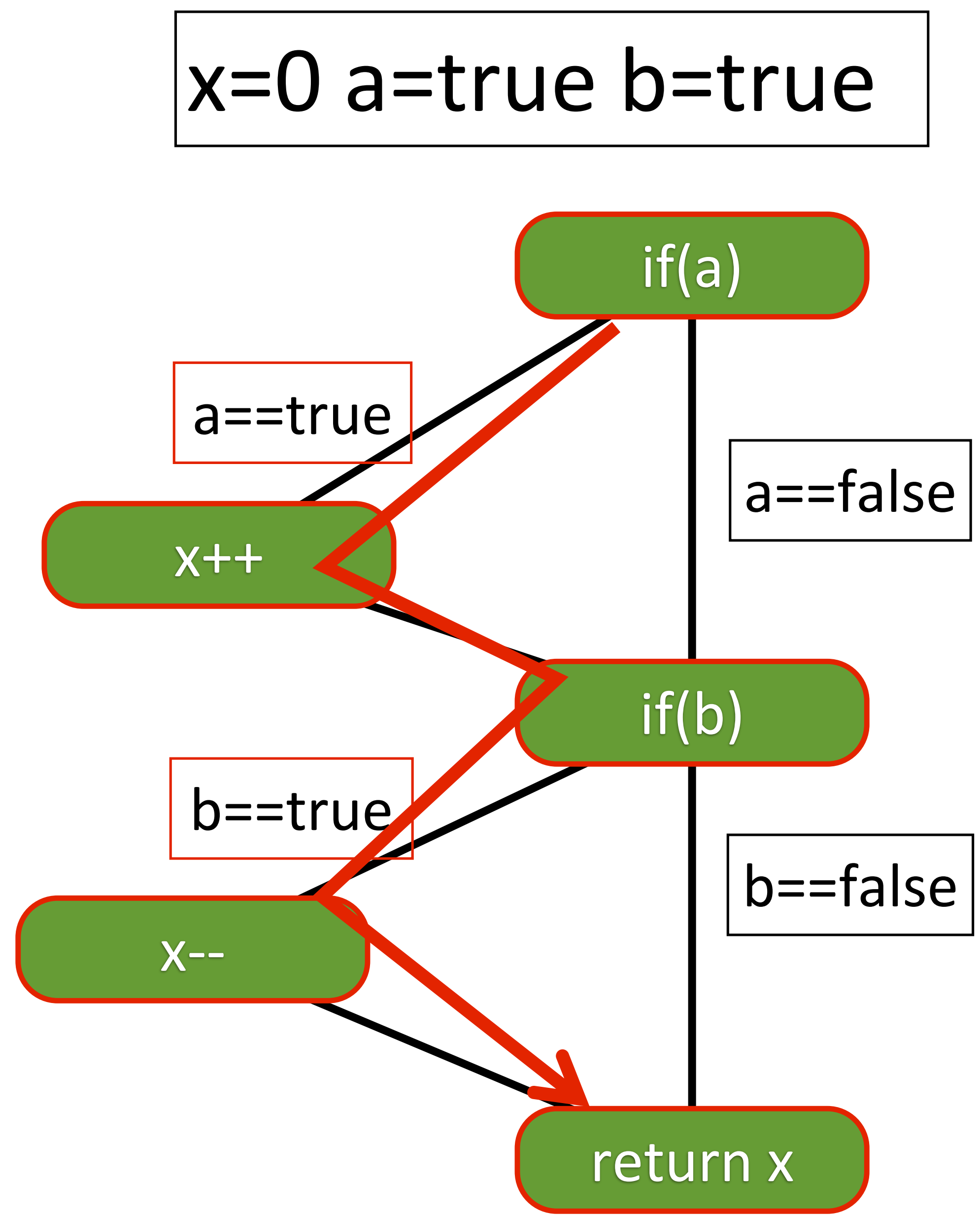
# Statement coverage VS. branch coverage

- If a test suite achieve 100% b-coverage, it must achieve 100% s-coverage
  - The statements not in branches will be covered by any test
  - All other statements are in certain branch
- If a test suite achieve 100% s-coverage, will it achieve 100% b-coverage?



# Statement coverage VS. branch coverage

- If a test suite achieve 100% b-coverage, it must achieve 100% s-coverage
  - The statements not in branches will be covered by any test
  - All other statements are in certain branch
- If a test suite achieve 100% s-coverage, will it achieve 100% b-coverage?



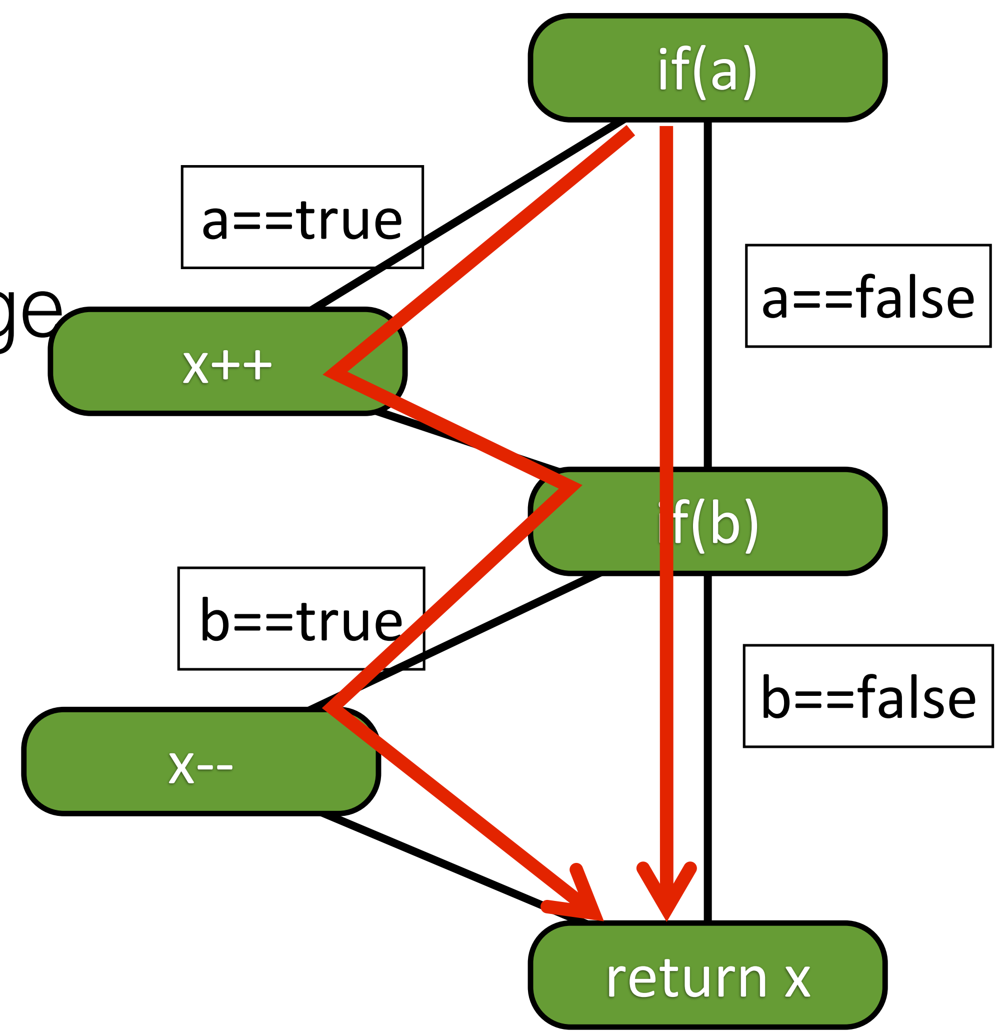

Branch coverage strictly subsumes statement coverage

# Branch coverage VS. path coverage

- If a test suite achieve 100% p-coverage, it must achieve 100% b-coverage
  - All the branch combinations have been covered indicate all branches are covered
- If a test suite achieve 100% b-coverage will it achieve 100% p-coverage?

x=0 a=true b=true

x=0 a=false b=false

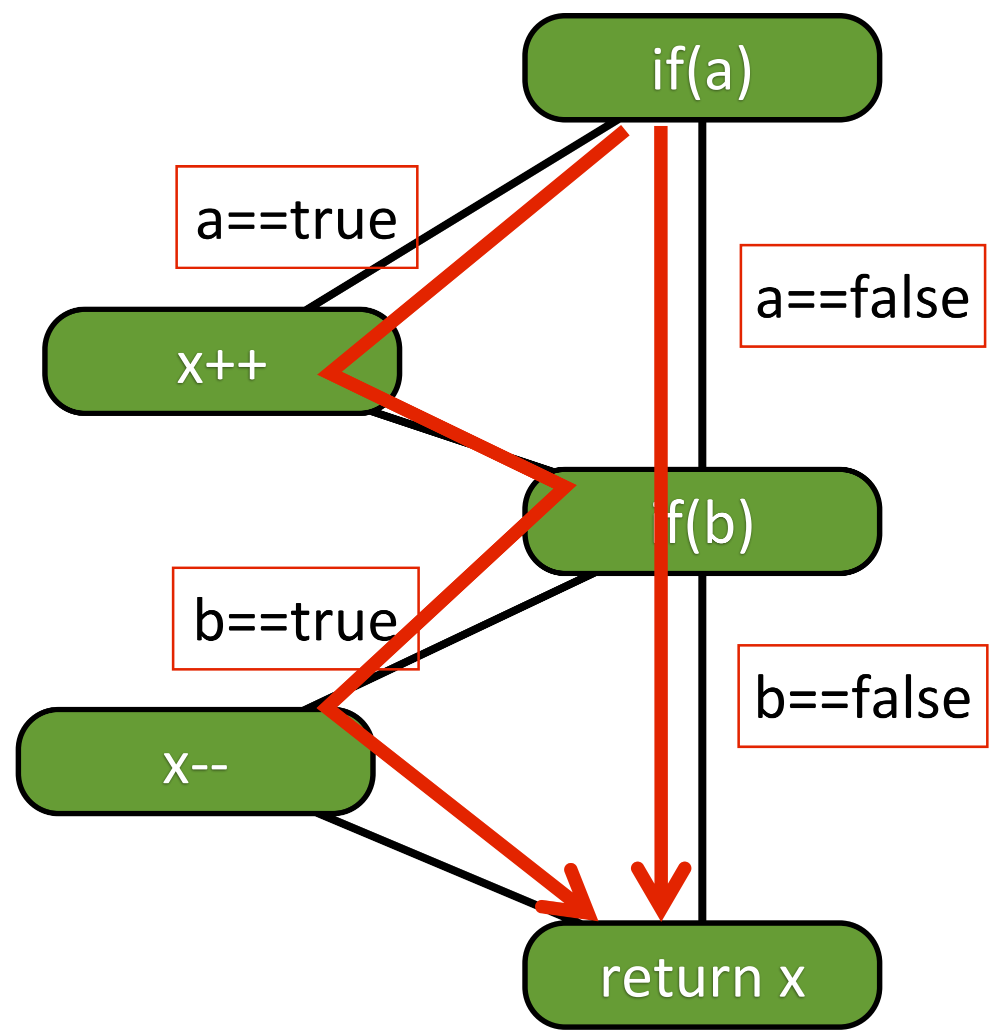


# Branch coverage VS. path coverage

- If a test suite achieve 100% p-coverage, it must achieve 100% b-coverage
  - All the branch combinations have been covered indicate all branches are covered
- If a test suite achieve 100% b-coverage, will it achieve 100% p-coverage?

x=0 a=true b=true

x=0 a=false b=false




Path coverage strictly subsumes branch coverage

# CFG-based coverage: comparison summary

Path coverage
strictly subsumes branch coverage
strictly subsumes statement coverage

Path Coverage

↓

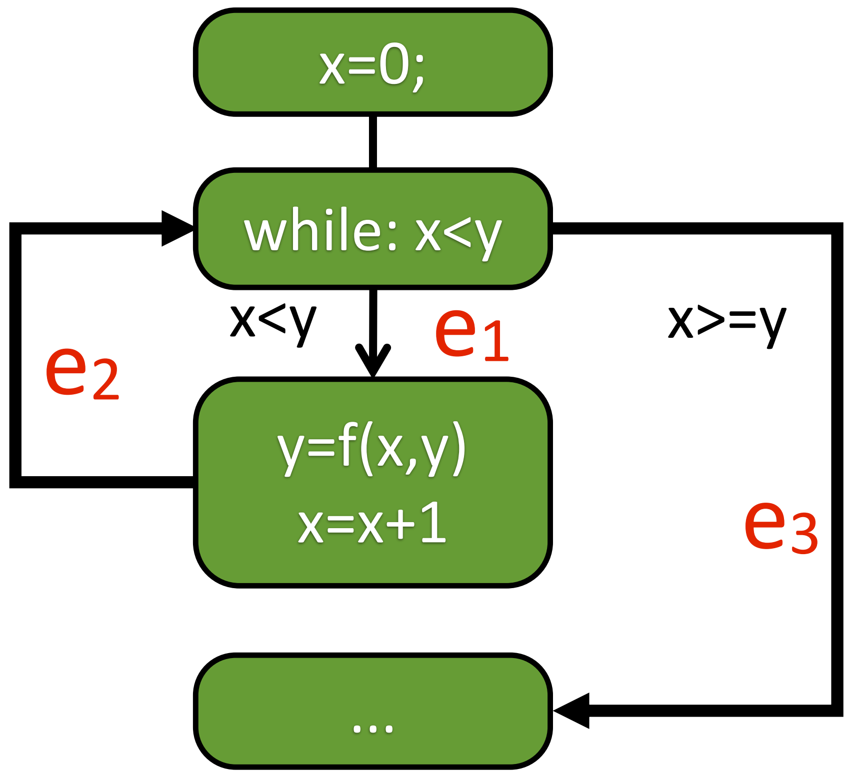
Branch Coverage

↓

Statement Coverage

# Should we just use path coverage?

```
while (x < y)
{
  y = f (x, y);
  x = x + 1;
}
...
```




**Possible Paths**

$e_3$

$e_1e_2e_3$

$e_1e_2e_1e_2e_3$

$e_1e_2e_1e_2e_1e_2e_3$

...

**Path coverage can be infeasible for real-world programs**

# CFG-based coverage: limitation

100% coverage may nc be possible due to infeasible paths/branches.

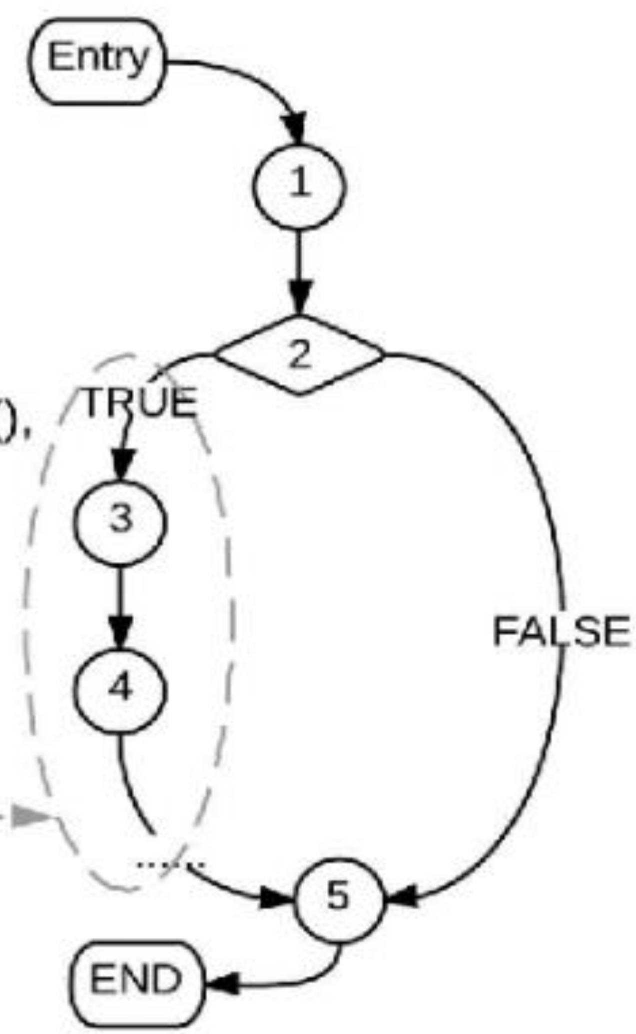
```
......
if ((cacheLen > 0) && (Y != NULL) {
  count += (cacheLen + counter.m_countNodesStartCount);
  if (cacheLen > 0)
    appendBtoFList(counter.m_countNodes,m_newFound);
  m_newFound.removeAllElements();
  return count;
}
......
```

```
void foo (int begin, int textLenth) {
1.  ttype = Token.SKIP;

2.  if (createToken && token
              ==null && ttype!=Token.SKIP ) {
3.      token = makeToken(ttype);
4.      token.setText(new String(text.getBuffer(),
              begin, textLenth));
        ........
    }

5.      returnToken = token;
}
```

Infeasible Branch



# CFG-based coverage: limitation

- 100% coverage of some aspect is never a guarantee of bug-free software

**Test case: (1, 0)**

```
public int sum(int x, int y){
        return x-y; //should be x+y
   }
```

Statement coverage: 100%

Branch coverage: 100%

Path coverage: 100%

Failed to detect the bug...

# Tool support - testing framework

- xUnit
  - Created by Kent Beck in 1989
    - This is the same guy who invented XP and TDD
    - The first one was sUnit (for smalltalk)
    - There are about 70 xUnit frameworks for corresponding languages
- JUnit
  - A simple, flexible, easy-to-use, open-source, and practical xUnit framework for Java.
  - Can deal with a large and extensive set of test cases.
  - Refer to [www.junit.org](http://www.junit.org).

# Tool support – A jUnit example

```
public class CFGCoverageExample {
	public int testMe(int x, boolean a, boolean b){
		if(a)
			x++;
		if(b)
			x--;
		return x;
	}
}
```

```
public class JUnitStatementCov {
	CFGCoverageExample tester;
	@Before
	public void initialize() {
		tester = new CFGCoverageExample();
	}
	@Test
	public void testCase() {
		assertEquals(1, tester.testMe(0, true, false));
	}
}
```

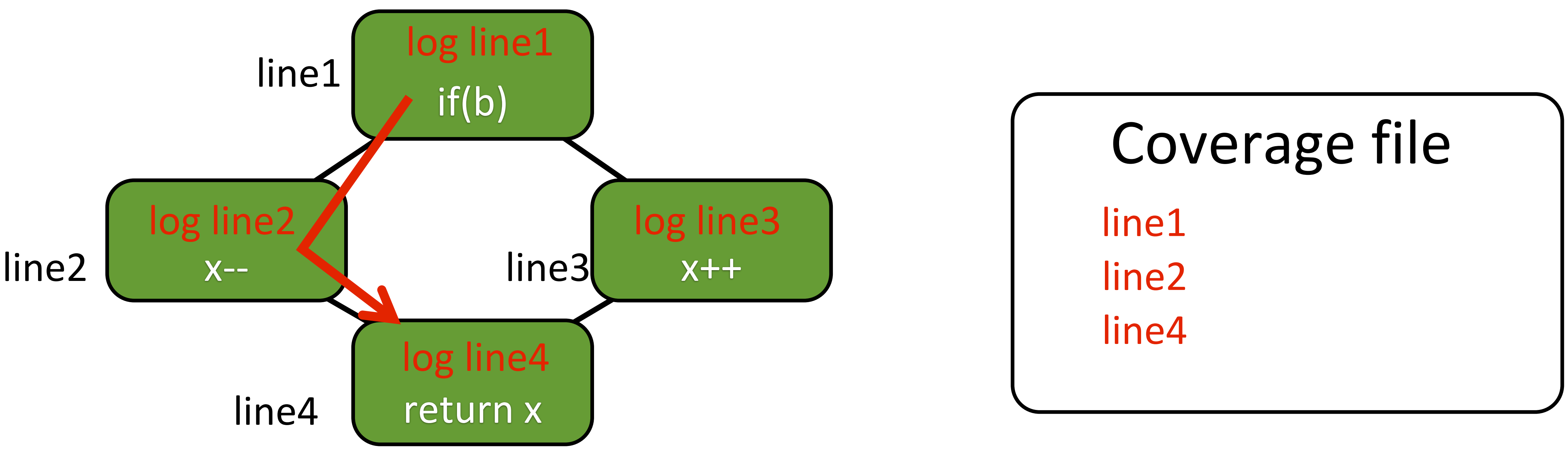
| | |
|---|---|
| assertEquals([msg], expected, actual) | Check that the two values are equal |

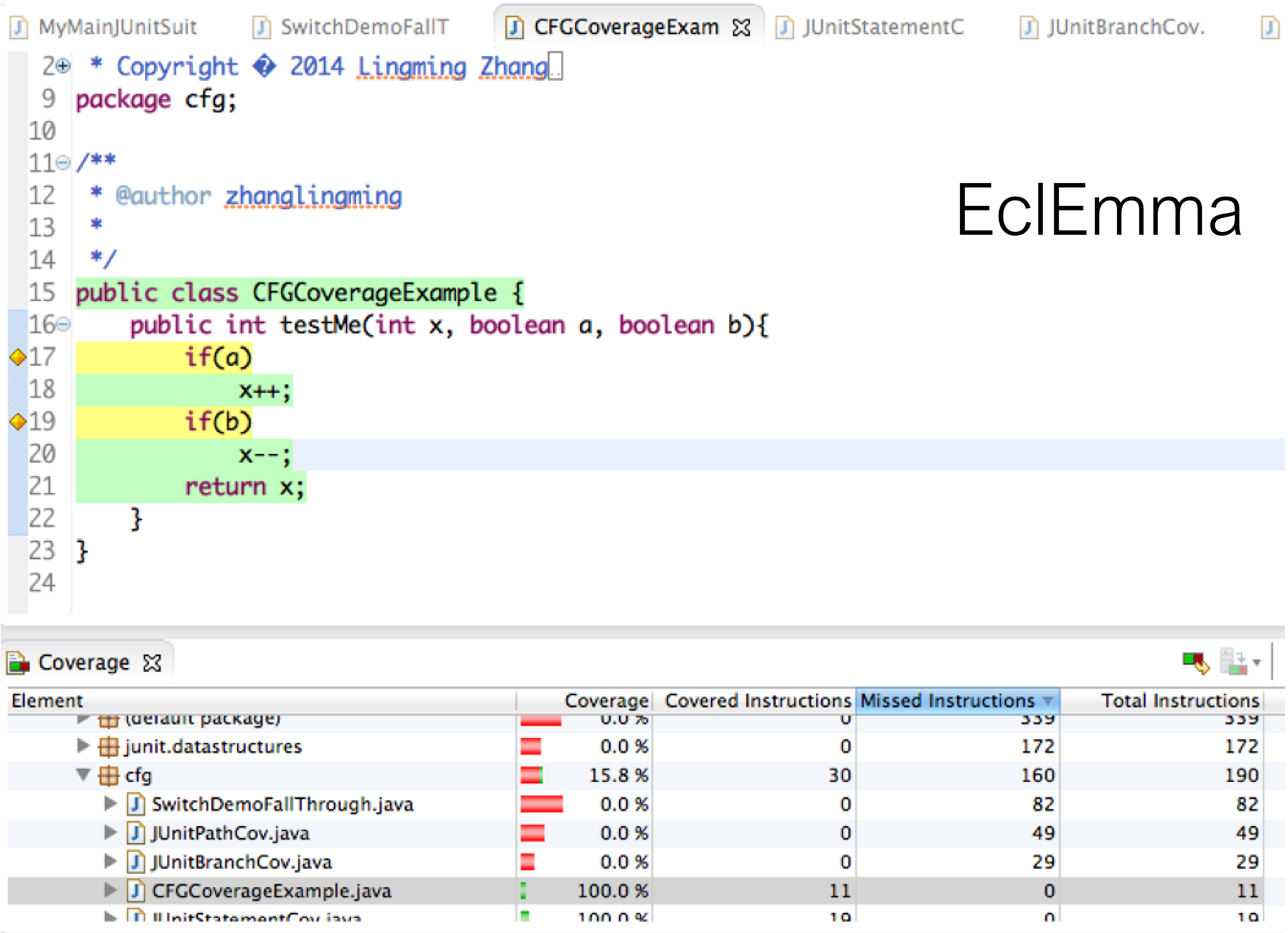
# Tool support - coverage collection

- Emma: http://emma.sourceforge.net/
- EclEmma: http://www.eclemma.org/installation.html/
- Cobertura: http://cobertura.github.io/cobertura/
- Clover: https://www.atlassian.com/software/clover/overview
- JCov: https://wiki.openjdk.java.net/display/CodeTools/jcov
- JaCoCo: http://www.eclemma.org/jacoco/

# Tool support - coverage collection

- The code under test is instrumented (source/binary)
  - Log code that writes to a trace file is inserted in every branch, statement etc.
- When the instrumented code is executed, the coverage info will be written to trace file

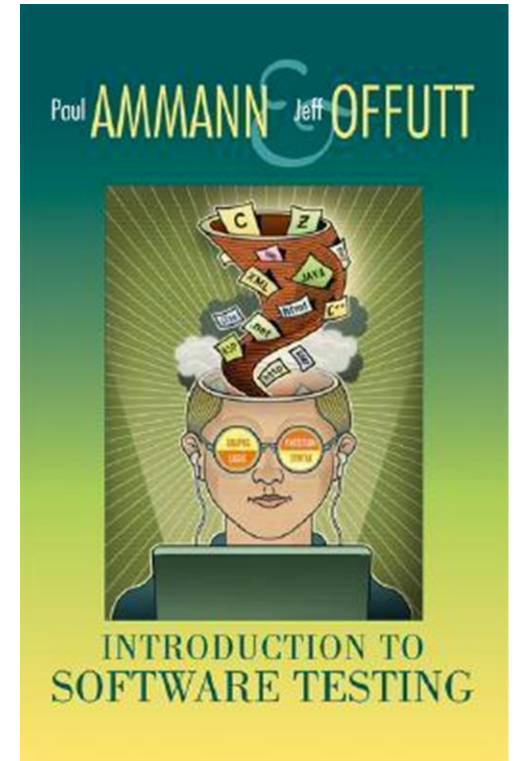


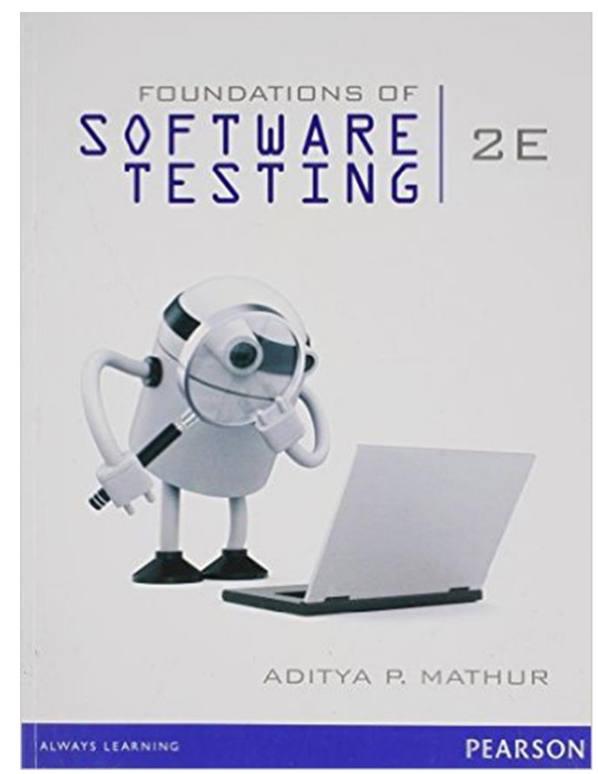

MyMainJUnitSuit
SwitchDemoFallT
CFGCoverageExam
JUnitStatementC
JUnitBranchCov.
* Copyright � 2014 Lingming Zhang
package cfg;
/**
* @author zhanglingming
*
*/
public class CFGCoverageExample {
public int testMe(int x, boolean a, boolean b){
if(a)
x++;
if(b)
x--;
return x;
}
}
Coverage
Element
Coverage
Covered Instructions
Missed Instructions
Total Instructions
junit.datastructures 0.0 % 0 172 172
cfg 15.8 % 30 160 190
SwitchDemoFallThrough.java 0.0 % 0 82 82
JUnitPathCov.java 0.0 % 0 49 49
JUnitBranchCov.java 0.0 % 0 29 29
CFGCoverageExample.java 100.0 % 11 0 11

# EclEmma

# References:


- Introduction to Software Testing (1st Edition), ISBN: 978-0521880381
- Foundations of Software Testing (2nd Edition), ISBN: 978-8131794760
- K Naik and P Tripathy, Software Testing and Quality Assurance: Theory and Practice, Wiley, ISBN: 978-0-471-78911-6, 2008.


# Acknowledgment:


. Dr Lingming Zhang, UT Dallas

. Dr Diana Kuo, Swinburne University of Technology

. A/Prof. Dan Hao, Peking University

Thanks!


Testing
Software
development
test
quality
articles
management
design
code
risk
good
usability
People
Project
information
system
check
Agile
web
use
article
Ruby