Introduction to Software Testing

Chapter 02
Graph Coverage

Paul Ammann & Jeff Offutt

Updated by Sunae Shin

Outline

- Graph coverage for source code
- Graph coverage for design elements
- Graph coverage for use cases

Graph Coverage for Source Code

Overview

- The most widely used graph coverage criteria are defined on **source code**
  - The basic pattern is the same for most common languages

- To apply one of the graph criteria the first step is to define the graph
  - The most common graph is called a **control flow graph (CFG)**

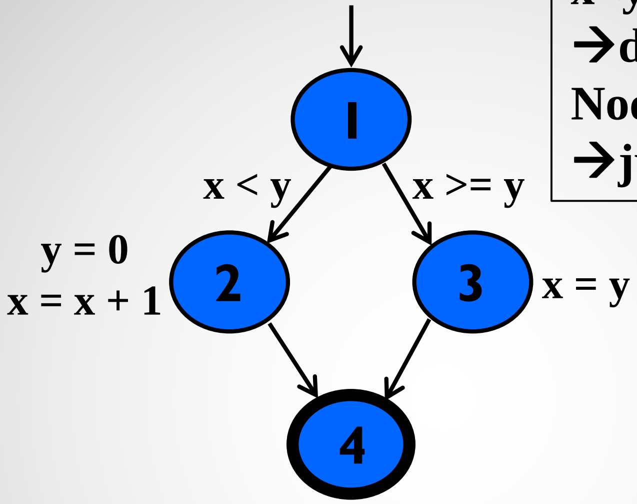
Control Flow Graphs

- A CFG models all executions of a method by describing control structures
  - Nodes : Statements or sequences of statements (basic blocks)
  - Edges : Transfers of control
  - **Basic Block** : A sequence of statements such that if the first statement is executed, all statements will be (no branches)

- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses

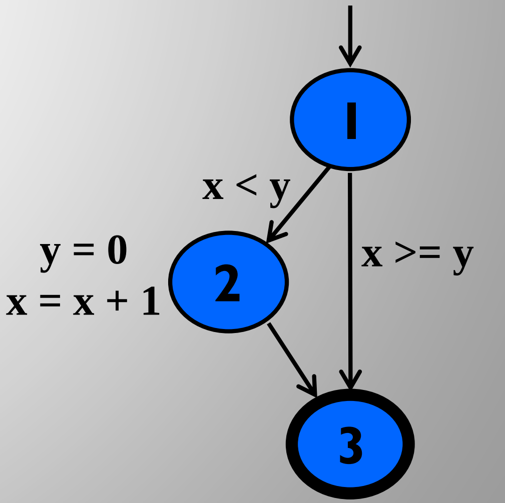
CFG : The if Statement

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



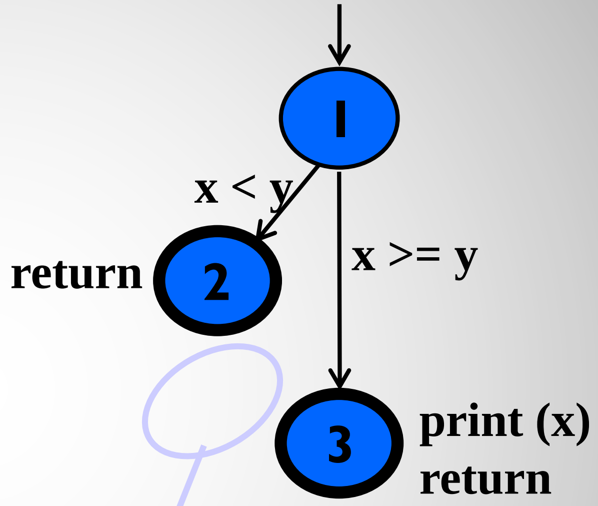
Node 1: represents the conditional test x<y has more than one out-edge
→decision node
Node 4: has more than one in-edge
→junction node

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



CFG : The if-Return Statement

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



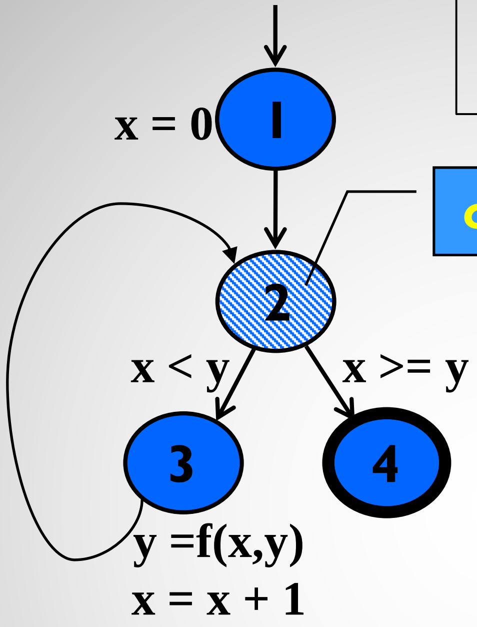
No edge from node 2 to 3.
The return nodes must be distinct.

Loops

- Loops require "*extra*" nodes to be added
- Nodes that do not represent statements or basic blocks

CFG : while and for Loops

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

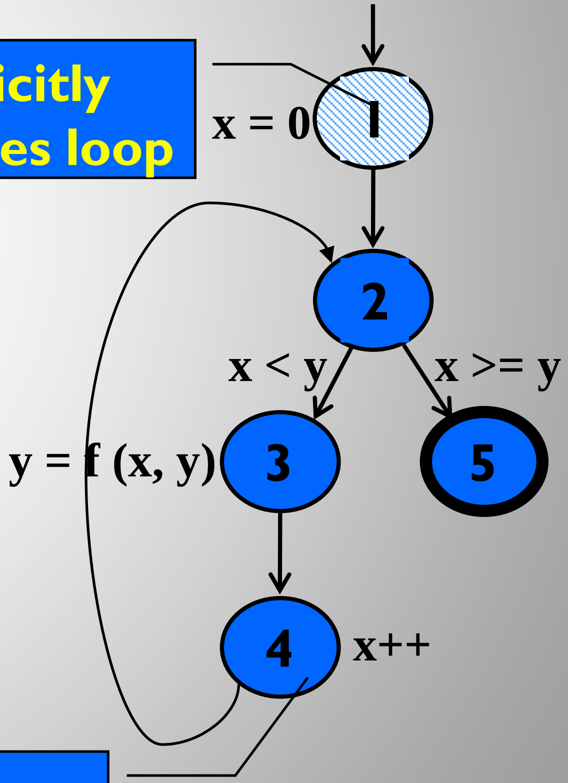


Common mistake – the edge goes to 1
→ Not correct

dummy node

implicitly initializes loop

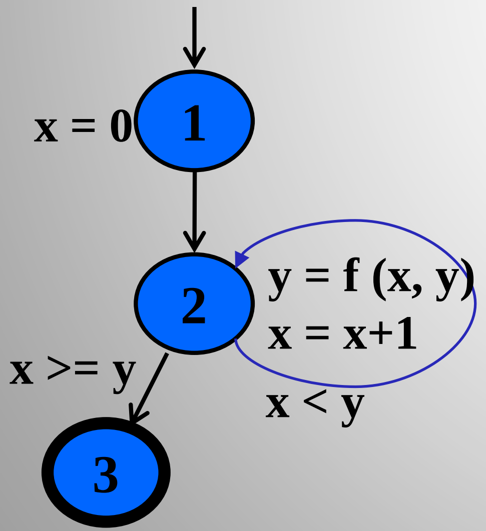
```
for (x = 0; x < y; x++)
{
   y = f (x, y);
}
```



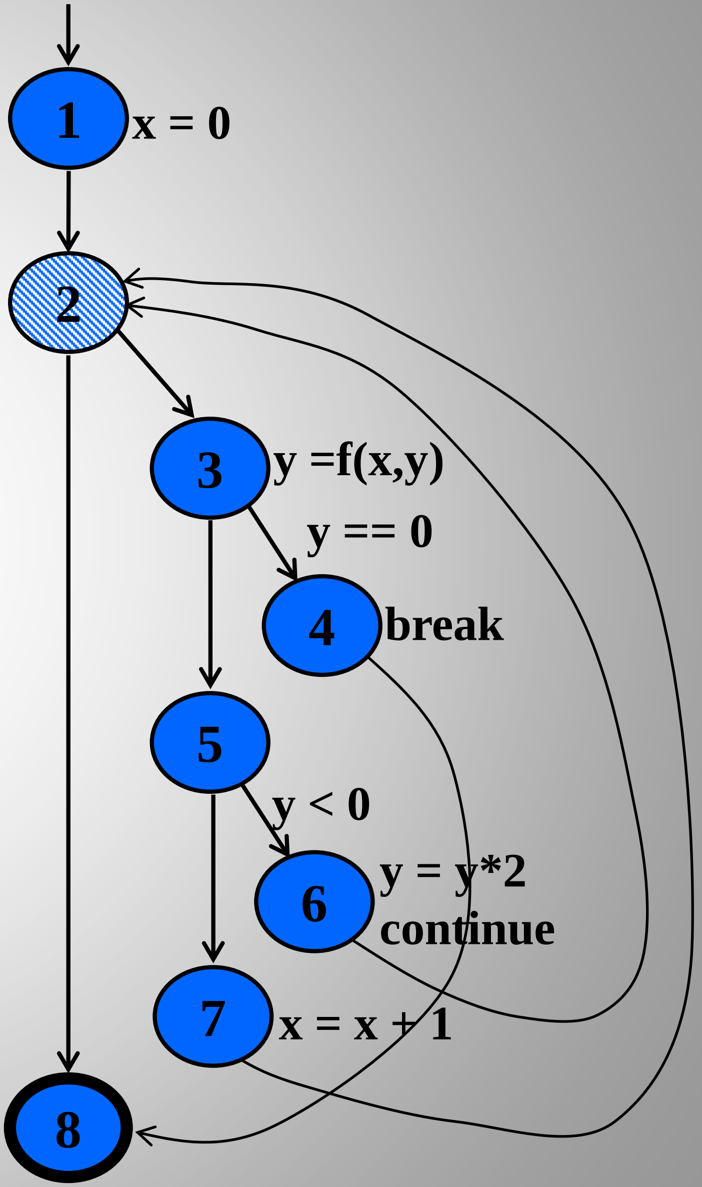
implicitly increments loop

CFG : do Loop, break and continue

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

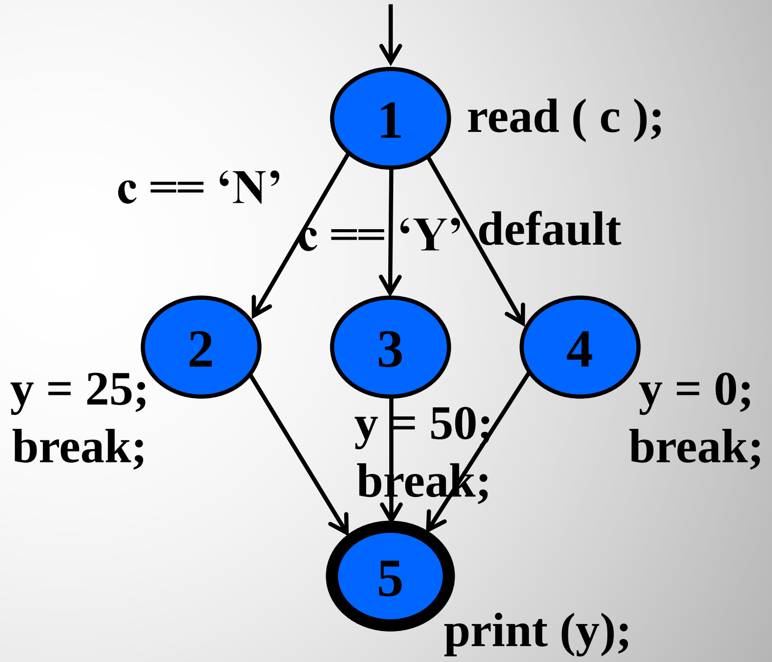


```
x = 0;
while (x < y)
{
   y = f (x, y);
   if (y == 0)
   {
      break;
   } else if (y < 0)
   {
      y = y*2;
      continue;
   }
   x = x + 1;
}
print (y);
```



CFG : The case (switch) Structure

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

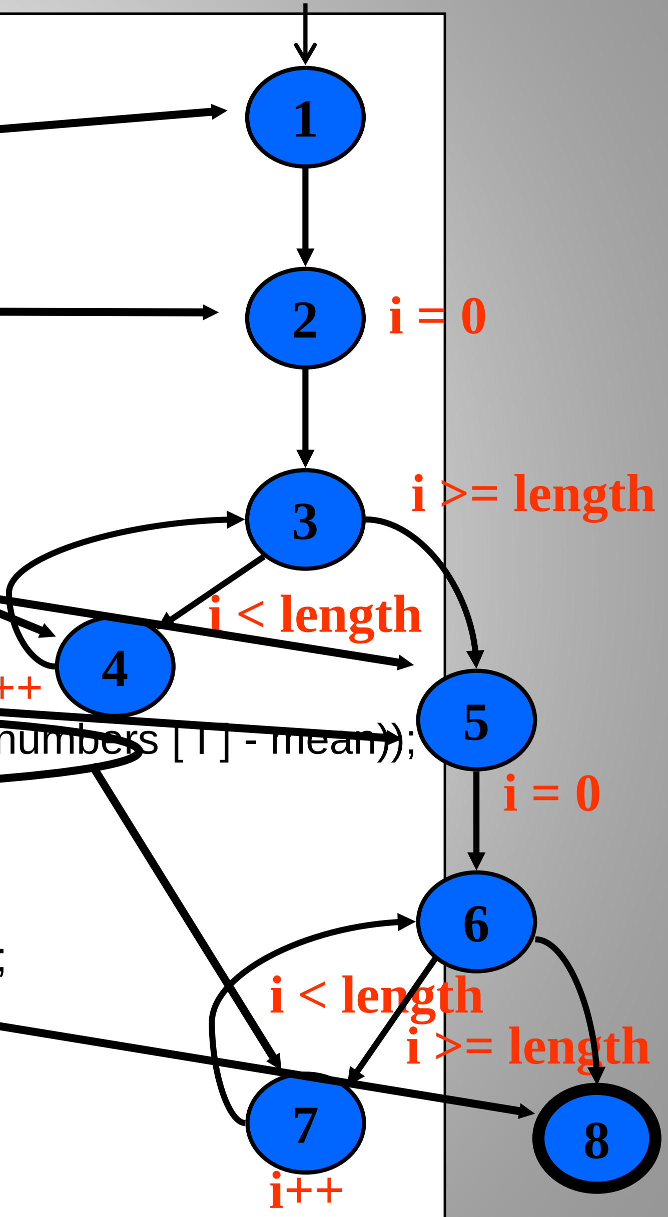


Example Control Flow – Stats

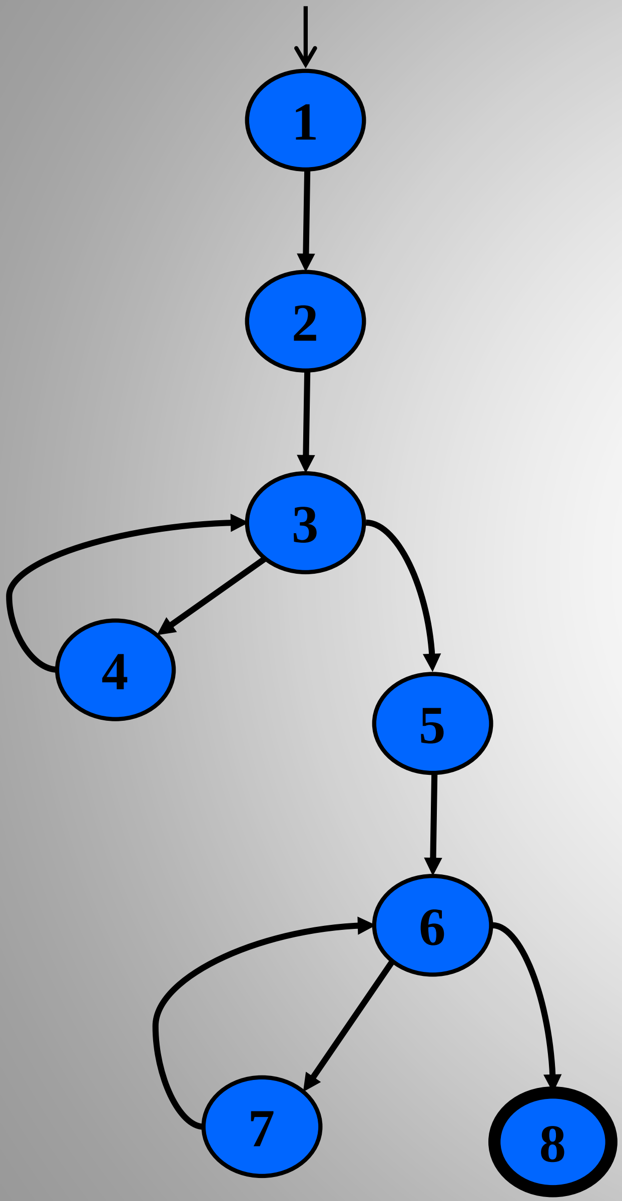
```
public static void computeStats (int [ ] numbers)
{
     int length = numbers.length;
     double med, var, sd, mean, sum, varsum;

     sum = 0;
     for (int i = 0; i < length; i++)
     {
          sum += numbers [ i ];
     }
     med   = numbers [ length / 2];
     mean = sum / (double) length;

     varsum = 0;
     for (int i = 0; i < length; i++)
     {
          varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
     }
     var = varsum / ( length - 1.0 );
     sd  = Math.sqrt ( var );

     System.out.println ("length:                    " + length);
     System.out.println ("mean:                      " + mean);
     System.out.println ("median:                    " + med);
     System.out.println ("variance:                  " + var);
     System.out.println ("standard deviation: " + sd);
}
```

Control Flow Graph for Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd   = Math.sqrt ( var );

    System.out.println ("length:                     " + length);
    System.out.println ("mean:                       " + mean);
    System.out.println ("median:                     " + med);
    System.out.println ("variance:                   " + var);
    System.out.println ("standard deviation: " + sd);
}
```



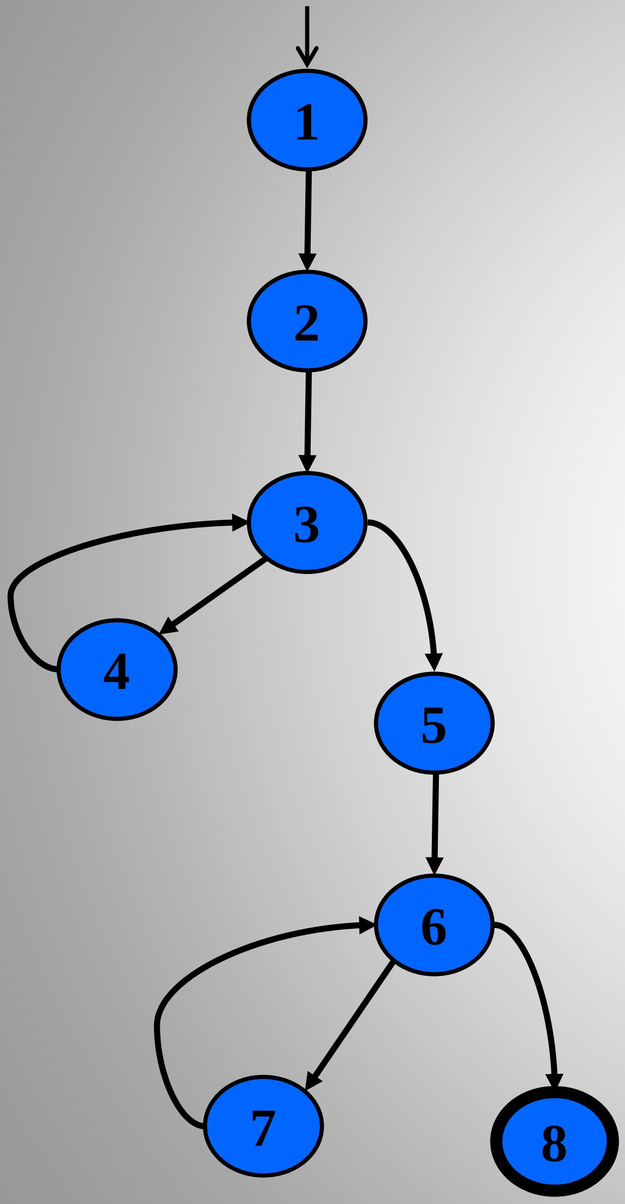
Control Flow TRs and Test Paths—EC



$$\begin{tikzcd}
 & 1 \arrow[d] & & \\
 & 2 \arrow[d] & & \\
 & 3 \arrow[dl] \arrow[dr] & & \\
4 \arrow[ur, bend left] & & 5 \arrow[d] & \\
 & & 6 \arrow[dl] \arrow[dr] & \\
 & 7 \arrow[ur, bend left] & & 8
\end{tikzcd}$$

Edge Coverage	
TR	**Test Path**
A. [1, 2]	[1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3]	
C. [3, 4]	
D. [3, 5]	
E. [4, 3]	
F. [5, 6]	
G. [6, 7]	
H. [6, 8]	
I. [7, 6]	

Control Flow TRs and Test Paths—EPC

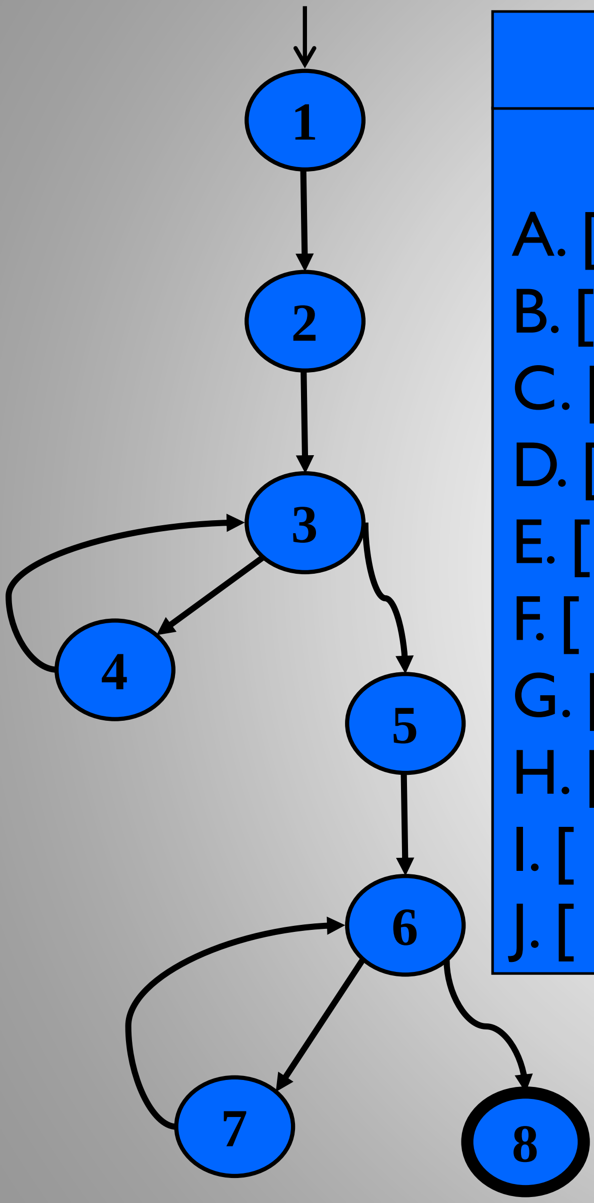


Edge-Pair Coverage

TR	Test Paths
A. [1, 2, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [2, 3, 4]	ii. [1, 2, 3, 5, 6, 8]
C. [2, 3, 5]	iii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
D. [3, 4, 3]	
E. [3, 5, 6]	
F. [4, 3, 5]	
G. [5, 6, 7]	
H. [5, 6, 8]	
I. [6, 7, 6]	
J. [7, 6, 8]	
K. [4, 3, 4]	
L. [7, 6, 7]	

15

Control Flow TRs and Test Paths—PPC



Prime Path Coverage	
TR	**Test Paths**
A. [3, 4, 3]	i. [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]
B. [4, 3, 4]	ii. [1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 8]
C. [7, 6, 7]	iii. [1, 2, 3, 4, 3, 5, 6, 8]
D. [7, 6, 8]	iv. [1, 2, 3, 5, 6, 7, 6, 8]
E. [6, 7, 6]	v. [1, 2, 3, 5, 6, 8]
F. [1, 2, 3, 4]	
G. [4, 3, 5, 6, 7]	
H. [4, 3, 5, 6, 8]	
I. [1, 2, 3, 5, 6, 7]	
J. [1, 2, 3, 5, 6, 8]	

16

Data Flow Coverage for Source

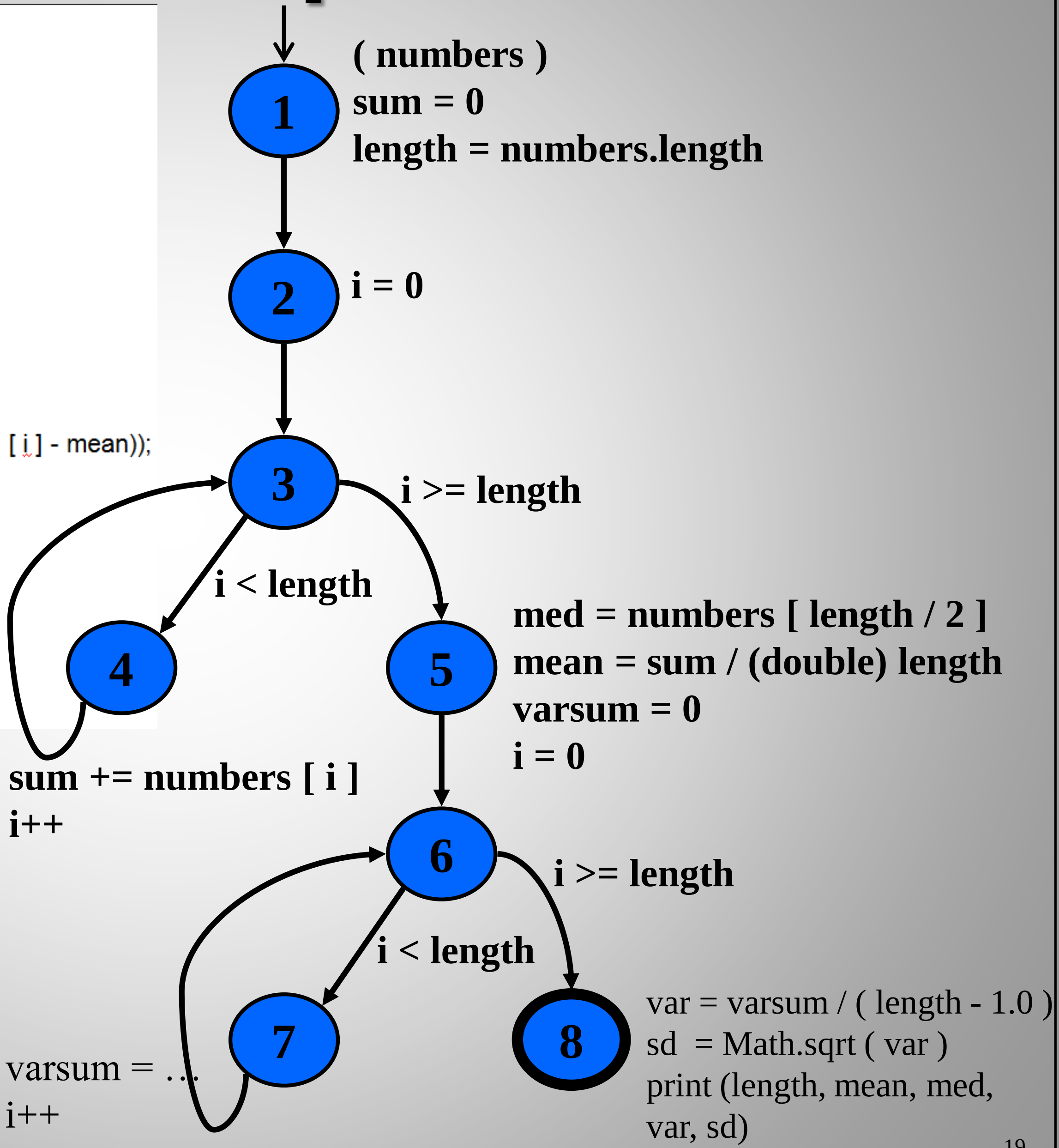
- **def : a location where a value is stored into memory**
  - x appears on the left side of an assignment (x = 44;)
  - x is an actual parameter in a call and the method changes its value
  - x is a formal parameter of a method (implicit def when method starts)
  - x is an input to a program
- **use : a location where variable's value is accessed**
  - x appears on the right side of an assignment
  - x appears in a conditional test
  - x is an actual parameter to a method
  - x is an output of the program
  - x is an output of a method in a return statement
- If a def and a use appear on the same node, then it is only a DU-pair if the def occurs after the use and the node is in a loop

Example Data Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                    " + length);
    System.out.println ("mean:                      " + mean);
    System.out.println ("median:                    " + med);
    System.out.println ("variance:                  " + var);
    System.out.println ("standard deviation: " + sd);
}
```

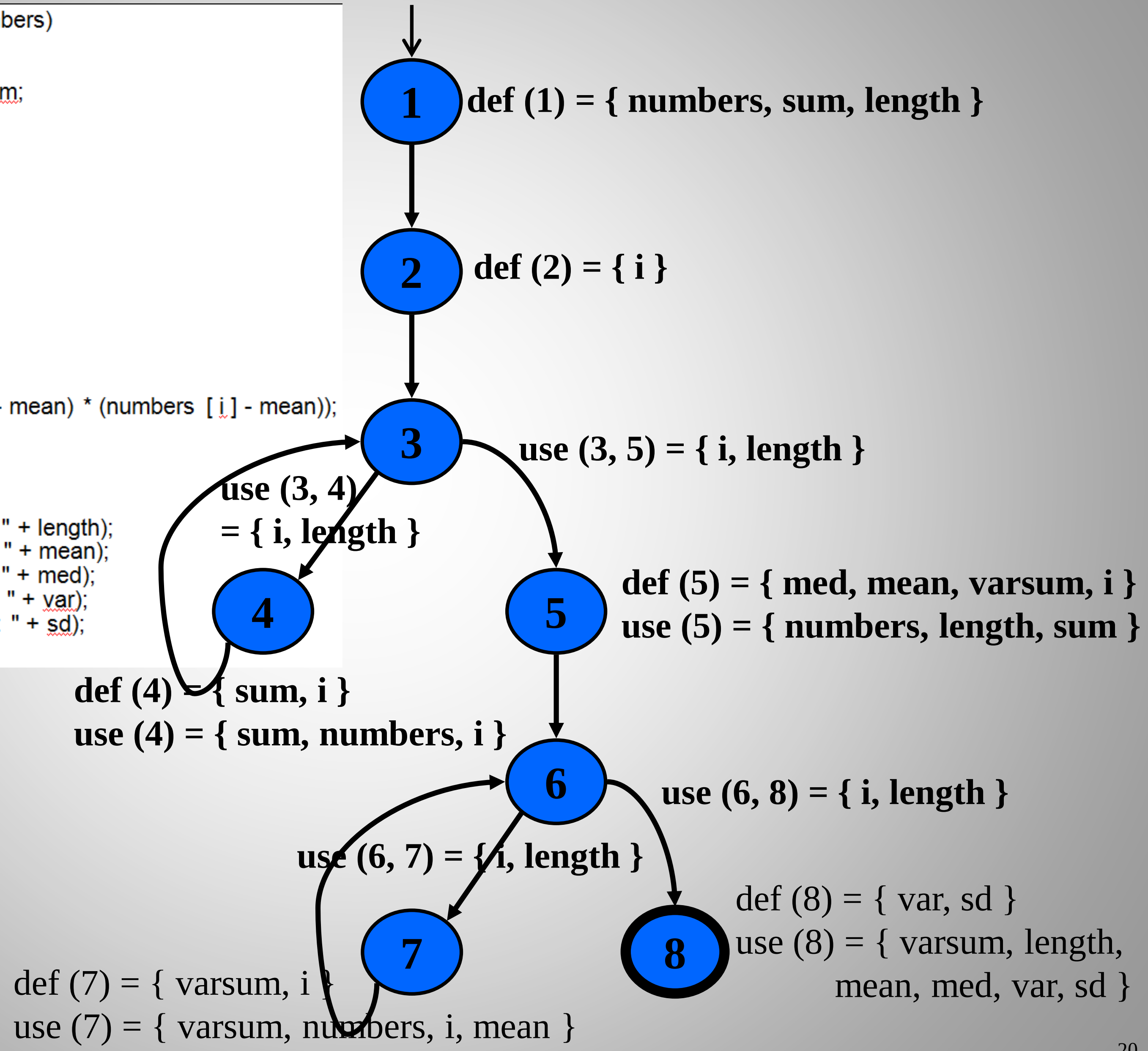
Control Flow Graph for Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:              " + length);
    System.out.println ("mean:                " + mean);
    System.out.println ("median:              " + med);
    System.out.println ("variance:            " + var);
    System.out.println ("standard deviation: " + sd);
}
```



CFG for Stats – With Defs & Uses

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0.0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0.o;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ i ] - mean) * (numbers [ i ] - mean));
    }
    var = varsum / ( length - 1 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:              " + var);
    System.out.println ("standard deviation: " + sd);
}
```



1
def (1) = { numbers, sum, length }
2
def (2) = { i }
3
use (3, 5) = { i, length }
use (3, 4) = { i, length }
4
5
def (5) = { med, mean, varsum, i }
use (5) = { numbers, length, sum }
def (4) = { sum, i }
use (4) = { sum, numbers, i }
6
use (6, 8) = { i, length }
use (6, 7) = { i, length }
7
8
def (8) = { var, sd }
use (8) = { varsum, length, mean, med, var, sd }
def (7) = { varsum, i }
use (7) = { varsum, numbers, i, mean }

Defs and Uses Tables for Stats

Node	Def	Use
1	{ numbers, sum, length }	{ numbers }
2	{ i }	
3		
4	{ sum, i }	{ numbers, i, sum }
5	{ med, mean, varsum, i }	{ numbers, length, sum }
6		
7	{ varsum, i }	{ varsum, numbers, i, mean }
8	{ var, sd }	{ varsum, length, var, mean, med, var, sd }

Edge	Use
(1, 2)	
(2, 3)	
(3, 4)	{ i, length }
(4, 3)	
(3, 5)	{ i, length }
(5, 6)	
(6, 7)	{ i, length }
(7, 6)	
(6, 8)	{ i, length }

DU Pairs for Stats

variable	DU Pairs
numbers	(1, 4) (1, 5) (1, 7)
length	(1, 5) (1, 8) (1, (3,4)) (1, (3,5)) (1, (6,7)) (1, (6,8))
med	(5, 8)
var	(8, 8)
sd	(8, 8)
mean	(5, 7) (5, 8)
sum	(1, 4) (1, 5) (4, 4) (4, 5)
varsum	(5, 7) (5, 8) (7, 7) (7, 8)
i	(2, 4) (2, (3,4)) (2, (3,5)) ~~(2, 7) (2, (6,7)) (2, (6,8))~~ (4, 4) (4, (3,4)) (4, (3,5)) ~~(4, 7) (4, (6,7)) (4, (6,8))~~ (5, 7) (5, (6,7)) (5, (6,8)) (7, 7) (7, (6,7)) (7, (6,8))

No def-clear path … different scope for i

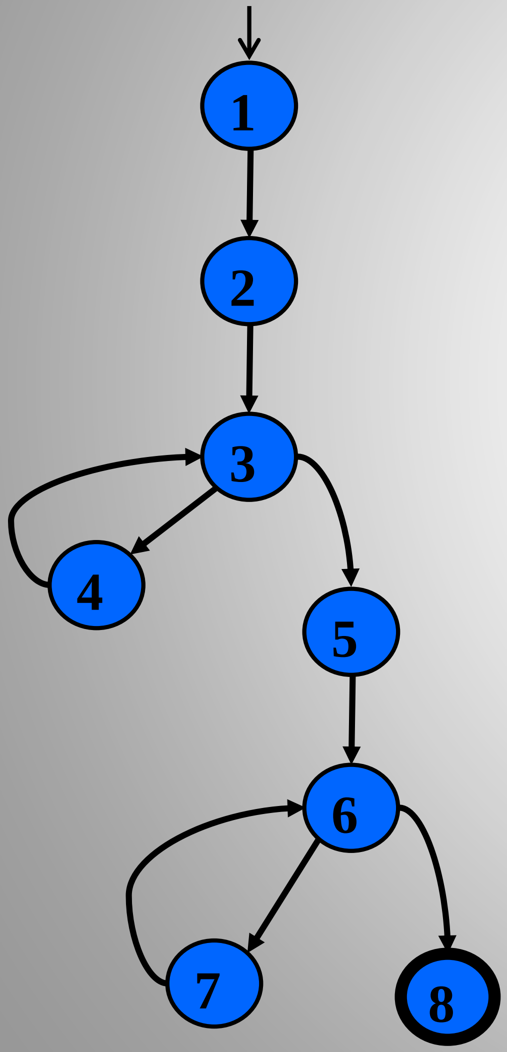
DU Paths for Stats

variable	DU Pairs	DU Paths
numbers	(1, 4)	[1, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(1, 7)	[1, 2, 3, 5, 6, 7]
length	(1, 5)	[1, 2, 3, 5]
	(1, 8)	[1, 2, 3, 5, 6, 8]
	(1, (3,4))	[1, 2, 3, 4]
	(1, (3,5))	[1, 2, 3, 5]
	(1, (6,7))	[1, 2, 3, 5, 6, 7]
	(1, (6,8))	[1, 2, 3, 5, 6, 8]
med	(5, 8)	[5, 6, 8]
sum	(1, 4)	[1, 2, 3, 4]
	(1, 5)	[1, 2, 3, 5]
	(4, 4)	[4, 3, 4]
	(4, 5)	[4, 3, 5]

variable	DU Pairs	DU Paths
mean	(5, 7)	[5, 6, 7]
	(5, 8)	[5, 6, 8]
varsum	(5, 7)	[5, 6, 7]
	(5, 8)	[5, 6, 8]
	(7, 7)	[7, 6, 7]
	(7, 8)	[7, 6, 8]
i	(2, 4)	[2, 3, 4]
	(2, (3,4))	[2, 3, 4]
	(2, (3,5))	[2, 3, 5]
	(4, 4)	[4, 3, 4]
	(4, (3,4))	[4, 3, 4]
	(4, (3,5))	[4, 3, 5]
	(5, 7)	[5, 6, 7]
	(5, (6,7))	[5, 6, 7]
	(5, (6,8))	[5, 6, 8]
	(7, 7)	[7, 6, 7]
	(7, (6,7))	[7, 6, 7]
	(7, (6,8))	[7, 6, 8]

DU Paths for Stats—No Duplicates

There are 38 DU paths for Stats, but only 12 unique



✦	[1, 2, 3, 4]	[4, 3, 4]	✹
★	[1, 2, 3, 5]	[4, 3, 5]	✦
✦	[1, 2, 3, 5, 6, 7]	[5, 6, 7]	✦
★	[1, 2, 3, 5, 6, 8]	[5, 6, 8]	★
✦	[2, 3, 4]	[7, 6, 7]	✹
★	[2, 3, 5]	[7, 6, 8]	✦

★ **4 expect a loop not to be "entered"**

✦ **6 require at least one iteration of a loop**

✹ **2 require at least <u>two</u> iterations of a loop**

Test Cases and Test Paths

Test Case : numbers = (44) ; length = 1

Test Path : [1, 2, 3, 4, 3, 5, 6, 7, 6, 8]

<u>Additional DU Paths covered (no sidetrips)</u>

[1, 2, 3, 4]   [2, 3, 4]   [4, 3, 5]   [5, 6, 7]   [7, 6, 8]

The five stars ✦ that require at least one iteration of a loop

Test Case : numbers = (2, 10, 15) ; length = 3

Test Path : [1, 2, 3, 4, 3, 4, 3, 4, 3, 5, 6, 7, 6, 7, 6, 7, 6, 8]

<u>DU Paths covered (no sidetrips)</u>

[4, 3, 4]   [7, 6, 7]

The two stars ✹ that require at least two iterations of a loop

Other DU paths ★ require arrays with length 0 to skip loops

But the method fails with index out of bounds exception…

```
med = numbers [length / 2];
```

A fault was found

Summary

- Applying the graph test criteria to control flow graphs is relatively straightforward
  - Most of the developmental research work was done with CFGs

- A few subtle decisions must be made to translate control structures into the graph

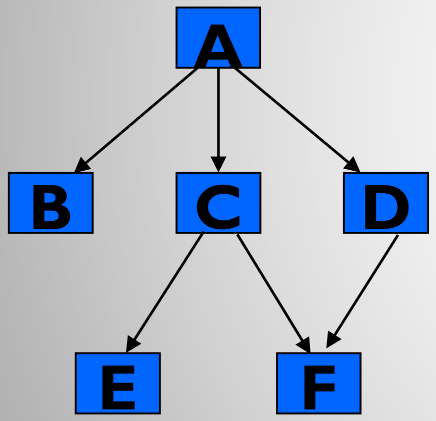
Graph Coverage for Design

OO Software and Designs

- Emphasis on modularity and reuse puts complexity in the design connections
- Testing design relationships is more important than before
- Graphs are based on the connections among the software components
  - Connections are dependency relations, also called couplings

Call Graph

- The most common graph for structural design testing
- Nodes : Units (in Java – methods)
- Edges : Calls to units



Example call graph

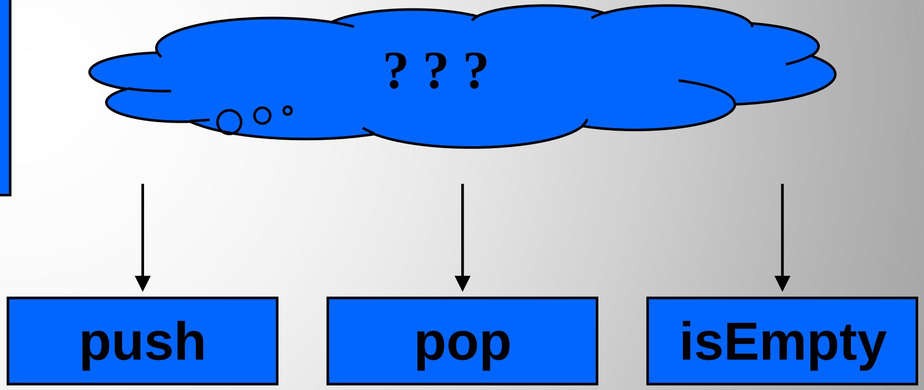
<u>Node coverage</u> : call every unit at least once (method coverage)

<u>Edge coverage</u> : execute every call at least once (call coverage)

Call Graphs on Classes

- Node and edge coverage of class call graphs often do not work very well
- Individual methods might not call each other at all!

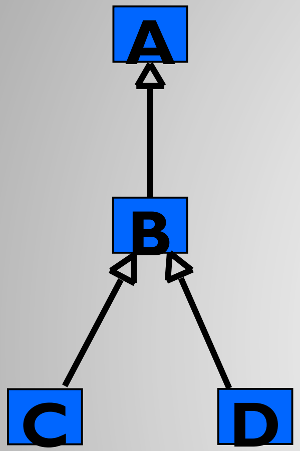
<u>Class stack</u>
public void push (Object o)
public Object pop ( )
public boolean isEmpty (Object o)



Other types of testing are needed – do <u>not</u> use graph criteria

Inheritance & Polymorphism

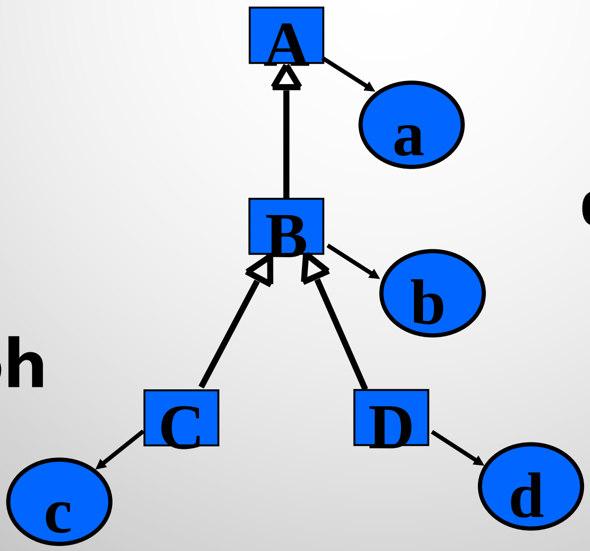
Caution : Ideas are preliminary and not widely used



Classes are not executable, so this graph is not directly testable

We need objects

Example inheritance hierarchy graph



objects

What is coverage on this graph ?

Coverage on Inheritance Graph

- Create an object for each class ?
  - This seems weak because there is no execution
- Create an object for each class and apply call coverage?

<u>OO Call Coverage</u> : TR contains each reachable node in the call graph of an object instantiated for each class in the class hierarchy.

<u>OO Object Call Coverage</u> : TR contains each reachable node in the call graph of <u>every</u> object instantiated for each class in the class hierarchy.

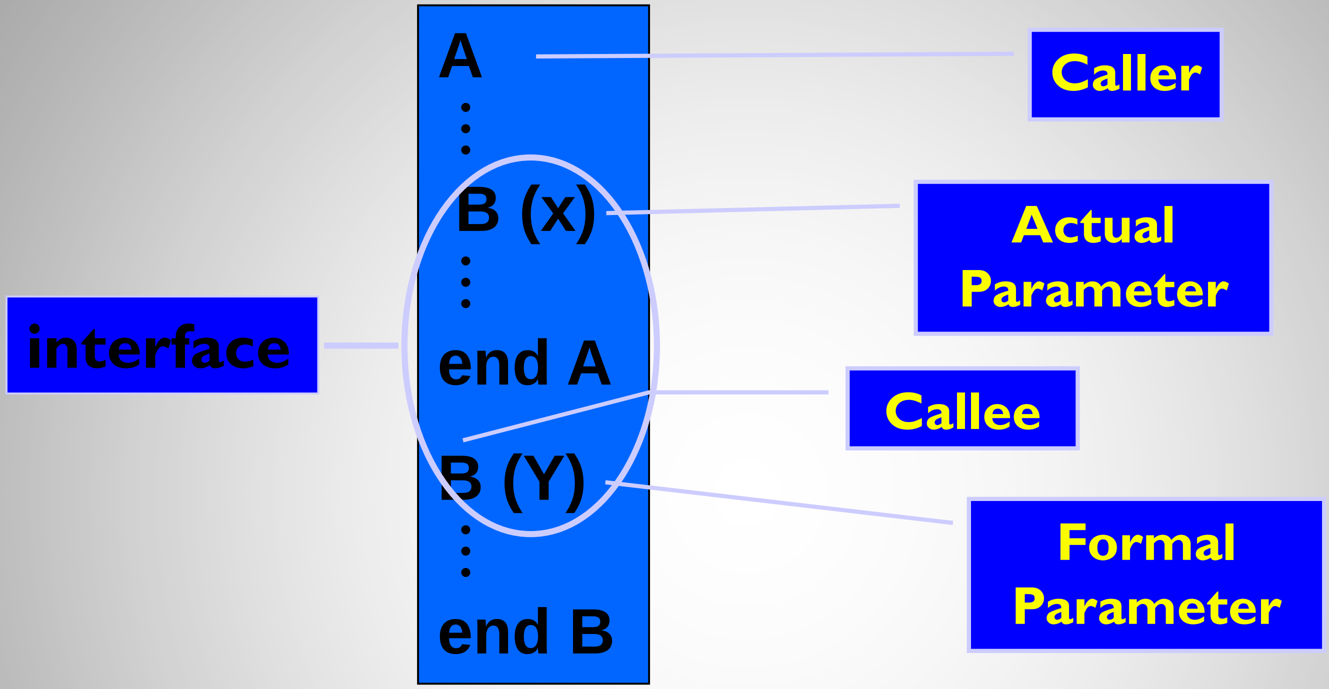
- Data flow is probably more appropriate …

Data Flow at the Design Level

- Data flow couplings among units and classes are complicated
  - When values are passed, they "change names"
  - Many different ways to share data
  - **Finding defs and uses can be difficult – finding which uses a def can reach is very difficult**
    - Primary issue – where the defs and uses occur
    - Testing program units – the defs and uses are in the same unit
    - During integration testing – the defs and uses are in different unit

- Compiler/program analysis terms
  - **Caller : A unit that invokes another unit**
  - **Callee : The unit that is called**
  - **Callsite : Statement or node where the call appears**
  - **Actual parameter : Variable in the caller**
  - **Formal parameter : Variable in the callee**

Example Call Site



- Applying data flow criteria to def-use pairs between units is too expensive
- Too many possibilities
- But this is integration testing, and we really only care about the **interface** …

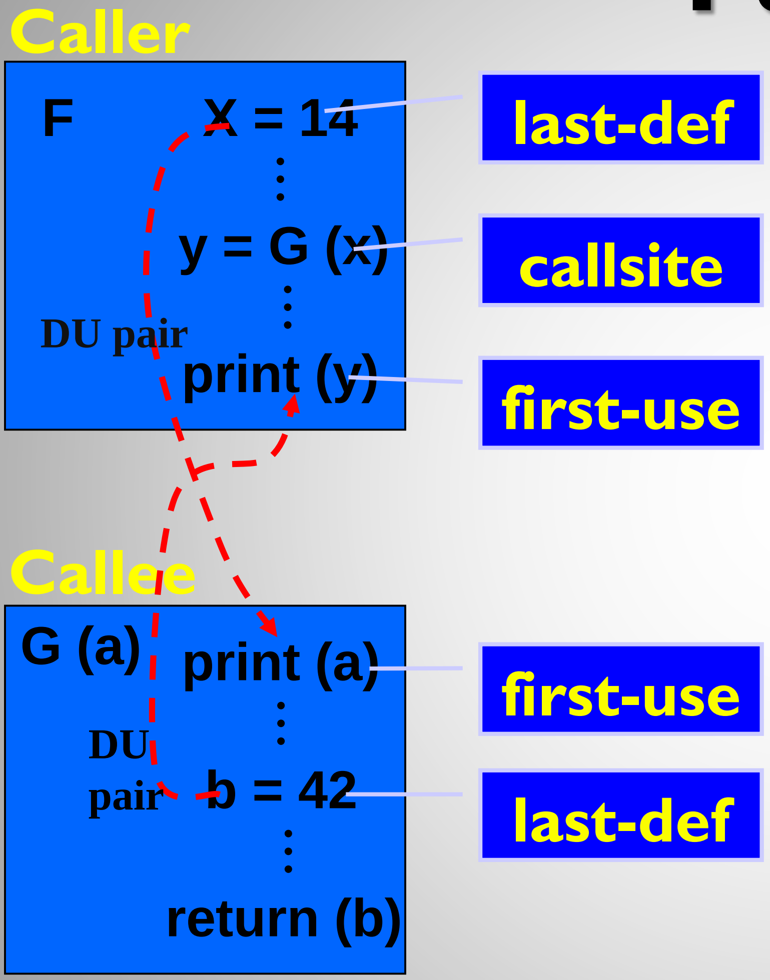
Inter-procedural DU Pairs

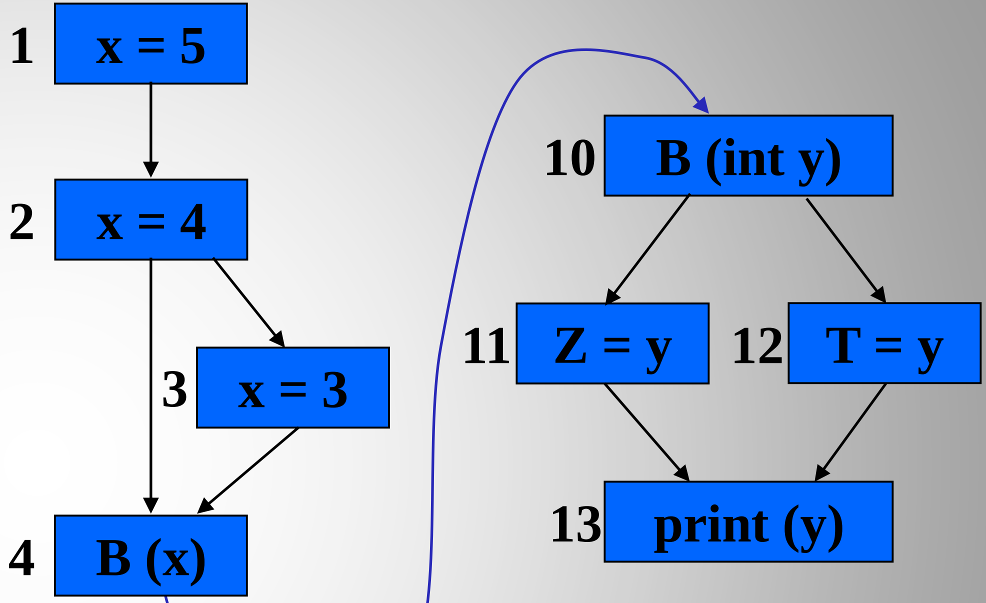
- **If we focus on the interface**,
  - then we just need to consider **the last definitions of variables before calls** and **returns and first uses inside units** and after calls

- **Last-def** : The set of nodes that define a variable *x* and has a def-clear path from the node through a callsite to a use in the other unit
  - Can be from caller to callee (parameter or shared variable) or from callee to caller as a return value

- **First-use**: The set of nodes that have uses of a variable *y* and for which there is a def-clear and use-clear path from the callsite to the nodes

Example Inter-procedural DU Pairs





Last Defs
2, 3

First Uses
11, 12

Example – Quadratic

```
// Program to compute the quadratic root for
two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
     int X, Y, Z;
    boolean ok;
    int controlFlag = Integer.parseInt (argv[0]);
    if (controlFlag == 1)
    {
        X = Integer.parseInt (argv[1]);
        Y = Integer.parseInt (argv[2]);
        Z = Integer.parseInt (argv[3]);
     }
    else
    {
         X = 10;
         Y = 9;
         Z = 12;
     }
        ok = Root (X, Y, Z);
        if (ok)
          System.out.println
               (“Quadratic: ” + Root1 + Root2);
        else
            System.out.println (“No Solution.”);
 }

// Three positive integers, finds quadratic root
 private static boolean Root (int A, int B, int C)
 {
    float D;
    boolean Result;
    D = (float) Math.pow ((double)B,
          (double2-4.0)*A*C );
    if (D < 0.0)
    {
       Result = false;
       return (Result);
    }
    Root1 = (float) ((-B + Math.sqrt(D))/(2.0*A));
    Root2 = (float) ((-B – Math.sqrt(D))/(2.0*A));
    Result = true;
    return (Result);
 } / /End method Root

 } // End class Quadratic
```

```
// Program to compute the quadratic root for two numbers
import java.lang.Math;

class Quadratic
{
 private static float Root1, Root2;

 public static void main (String[] argv)
 {
      int X, Y, Z;
     boolean ok;
     int controlFlag = Integer.parseInt (argv[0]);
     if (controlFlag == 1)
     {
         X = Integer.parseInt (argv[1]);
         Y = Integer.parseInt (argv[2]);
         Z = Integer.parseInt (argv[3]);
      }
     else
     {
          X = 10;
          Y = 9;
          Z = 12;
      }
```

shared variables

last-defs

first-use

first-use

last-def

last-defs

```
            ok = Root (X, Y, Z);
            if (ok)
              System.out.println
                  (“Quadratic: ” + Root1 + Root2);
            else
               System.out.println (“No Solution.”);
    }

    // Three positive integers, finds the quadratic root
    private static boolean Root (int A, int B, int C)
    {
       float D;
       boolean Result;
       D = (float) Math.pow ((double)B, (double2-4.0)*A*C);
       if (D < 0.0)
       {
          Result = false;

          return (Result);
       }
       Root1 = (float) ((-B + Math.sqrt(D)) / (2.0*A));
       Root2 = (float) ((-B – Math.sqrt(D)) / (2.0*A));
       Result = true;
       return (Result);
    } / /End method Root

} // End class Quadratic
```

Quadratic – Coupling DU-pairs

Pairs of locations: method name, variable name, statement

(main (), X, 15) – (Root (), A, 38)

(main (), Y, 16) – (Root (), B, 38)

(main (), Z, 17) – (Root (), C, 38)

(main (), X, 21) – (Root (), A, 38)

(main (), Y, 22) – (Root (), B, 38)

(main (), Z, 23) – (Root (), C, 38)

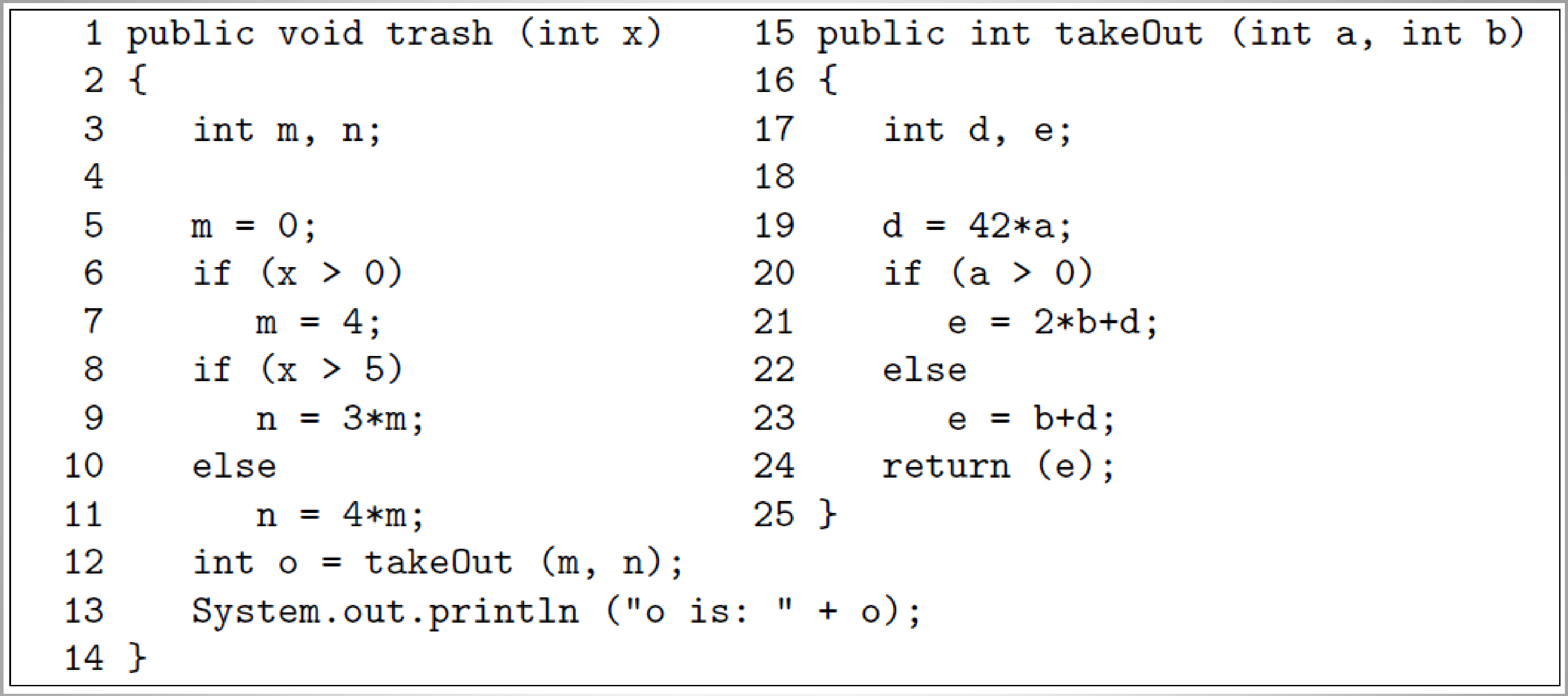
(Root (), Root1, 44) – (main (), Root1, 28)

(Root (), Root2, 45) – (main (), Root2, 28)

(Root (), Result, 41) – (main (), ok, 26)

(Root (), Result, 46) – (main (), ok, 26)

Exercise



```
 1 public void trash (int x)
 2 {
 3    int m, n;
 4
 5    m = 0;
 6    if (x > 0)
 7       m = 4;
 8    if (x > 5)
 9       n = 3*m;
10    else
11       n = 4*m;
12    int o = takeOut (m, n);
13    System.out.println ("o is: " + o);
14 }
```

```
15 public int takeOut (int a, int b)
16 {
17    int d, e;
18
19    d = 42*a;
20    if (a > 0)
21       e = 2*b+d;
22    else
23       e = b+d;
24    return (e);
25 }
```

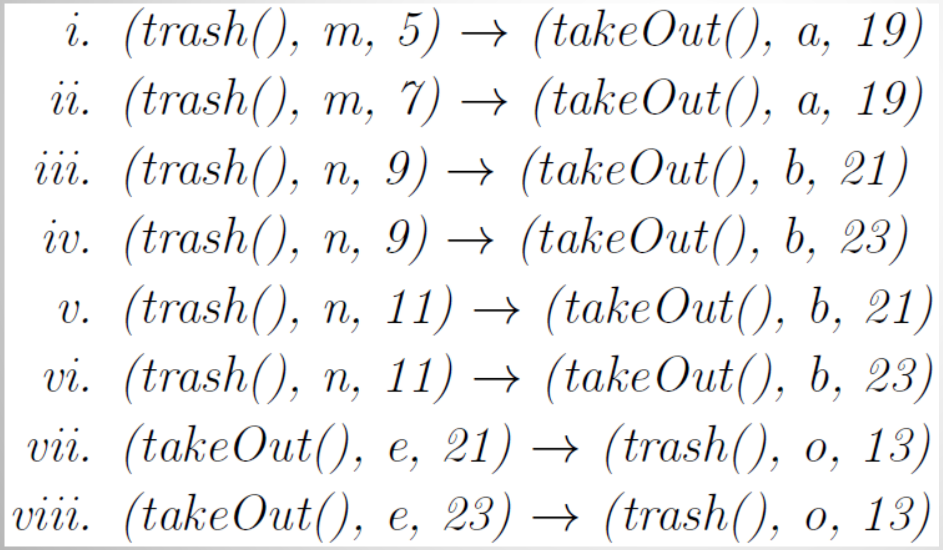
a) Give all callsites using the line numbers given

b) Give all pairs of last-defs and first-uses

Exercise Solution

- (a) Only one callsite
  - trash(), Line 12 → takeout()

- (b)



i. $(trash(),\ m,\ 5) \rightarrow (takeOut(),\ a,\ 19)$

ii. $(trash(),\ m,\ 7) \rightarrow (takeOut(),\ a,\ 19)$

iii. $(trash(),\ n,\ 9) \rightarrow (takeOut(),\ b,\ 21)$

iv. $(trash(),\ n,\ 9) \rightarrow (takeOut(),\ b,\ 23)$

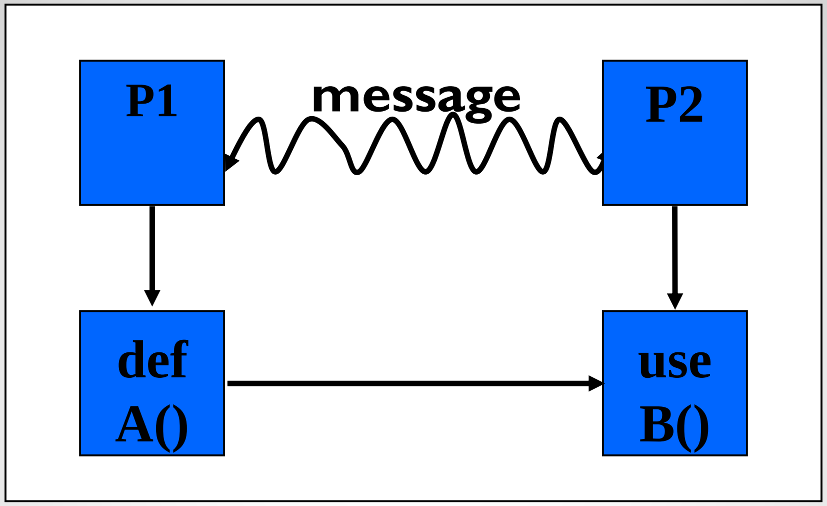
v. $(trash(),\ n,\ 11) \rightarrow (takeOut(),\ b,\ 21)$

vi. $(trash(),\ n,\ 11) \rightarrow (takeOut(),\ b,\ 23)$

vii. $(takeOut(),\ e,\ 21) \rightarrow (trash(),\ o,\ 13)$

viii. $(takeOut(),\ e,\ 23) \rightarrow (trash(),\ o,\ 13)$

Web Applications and Other Distributed Software



distributed software data flow

- "message" could be HTTP, RMI, or other mechanism
- A() and B() could be in the same class or accessing a persistent variable such as in a web session
- Beyond current technologies

Summary

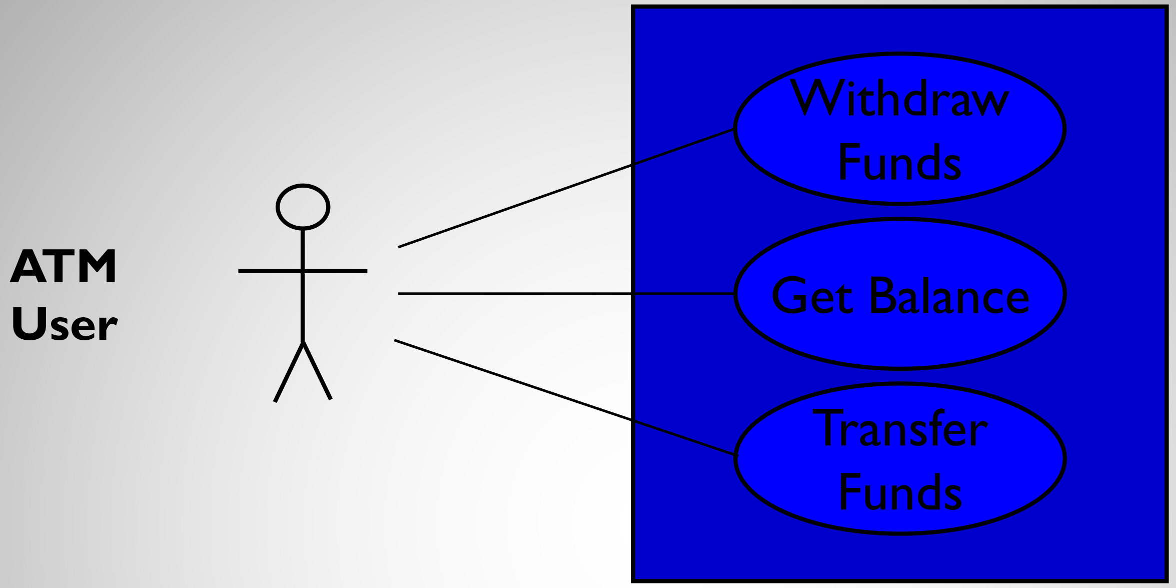
- Call graphs are common and very useful ways to design integration tests
- Inter-procedural data flow is relatively easy to compute and results in effective integration tests
- The ideas for OO software and web applications are preliminary and have not been used much in practice

Graph Coverage for Use cases

UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- Can be help the tester start testing activities early
  - Use cases are developed early in software development
- We won't teach use cases, but show examples

Simple Use Case Example



- Actors : Humans or software components that use the software being modeled
- Use cases : Shown as circles or ovals
- Node Coverage : Try each use case once …

Use case graphs, by themselves, are not useful for testing

Elaboration

- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

Elaboration of ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)

Elaboration of ATM Use Case–(2/3)

- Description (continued) :
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Wait A Minute …

- What does this have to do with testing ?
- Specifically, what does this have to do with graphs ???
- Remember our admonition : Find a graph, then cover it!
- UML has something very similar :

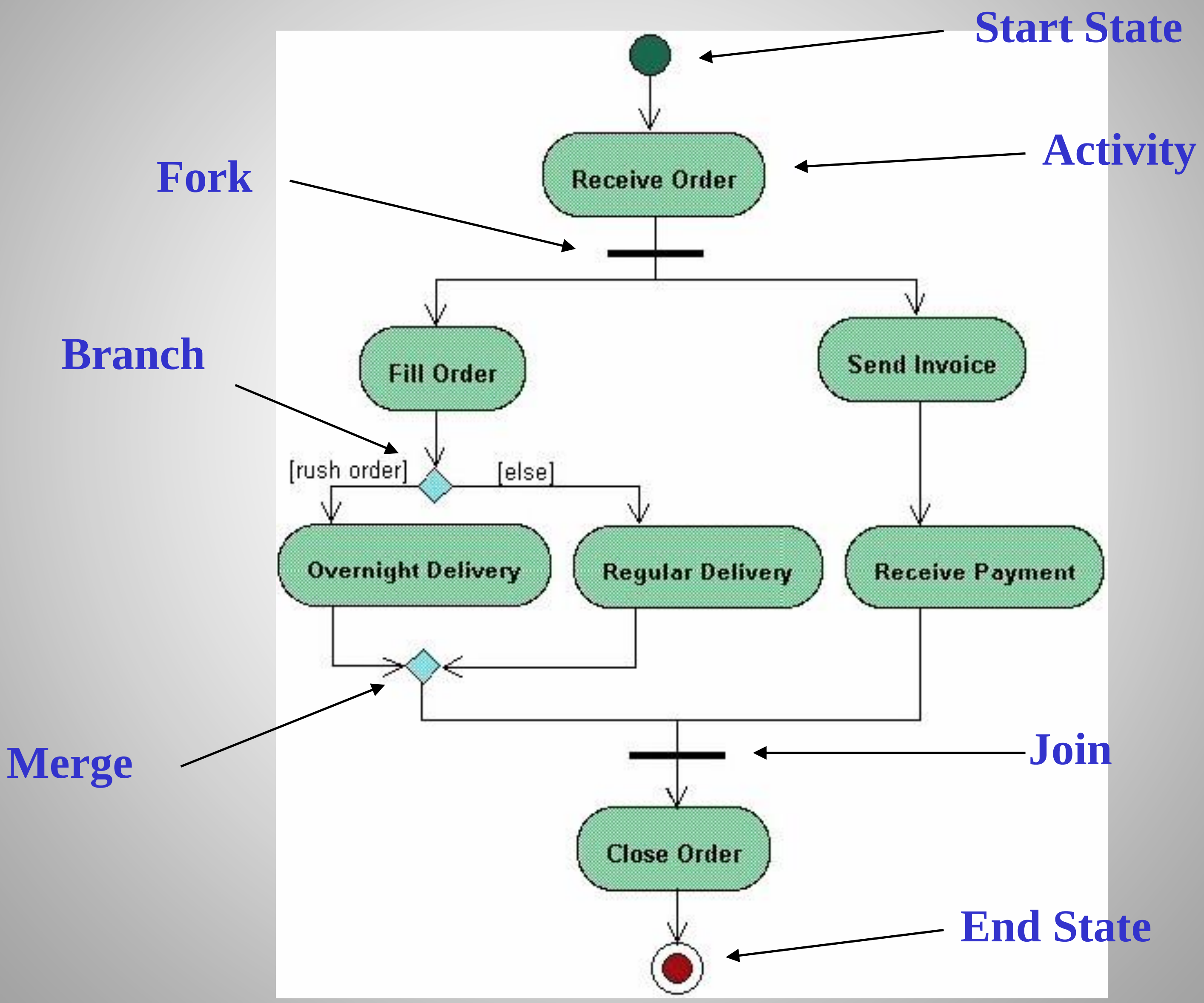
Activity Diagrams

Use Cases to Activity Diagrams

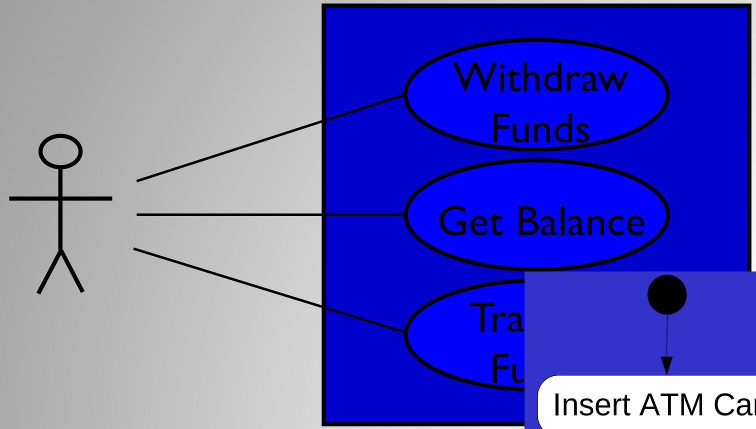
- Activity diagrams indicate flow among activities
  - Activities should model user level steps
  - Two kinds of nodes:
    - Action states
    - Sequential branches

  - Use case descriptions become action state nodes in the activity diagram
  - Flow among steps are edges
  - Activity diagrams usually have some helpful characteristics:
    - Usually do not have many loops
    - Simple predicates
    - No obvious DU pairs

Activity Diagram Example

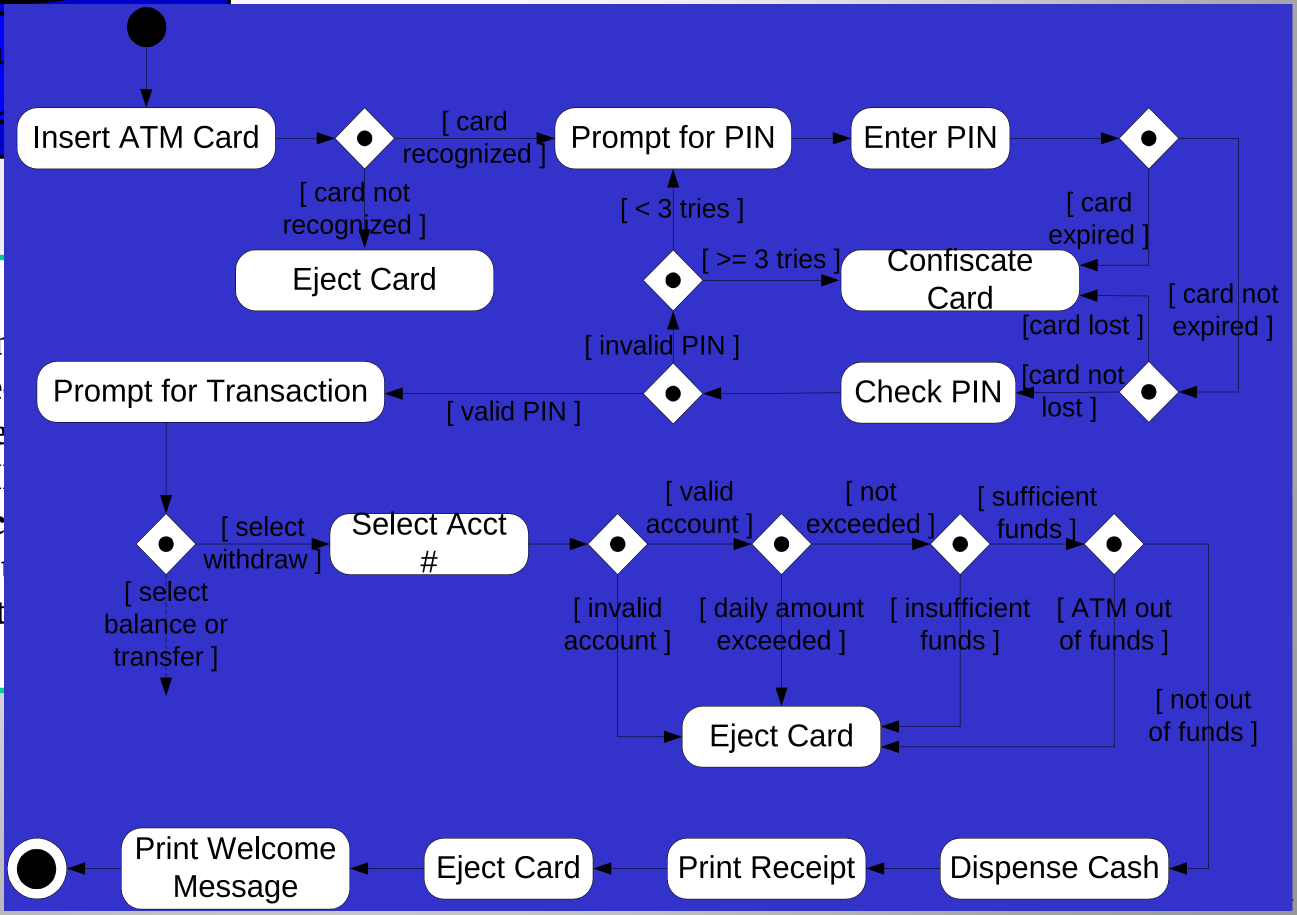


ATM Withdraw Activity Graph

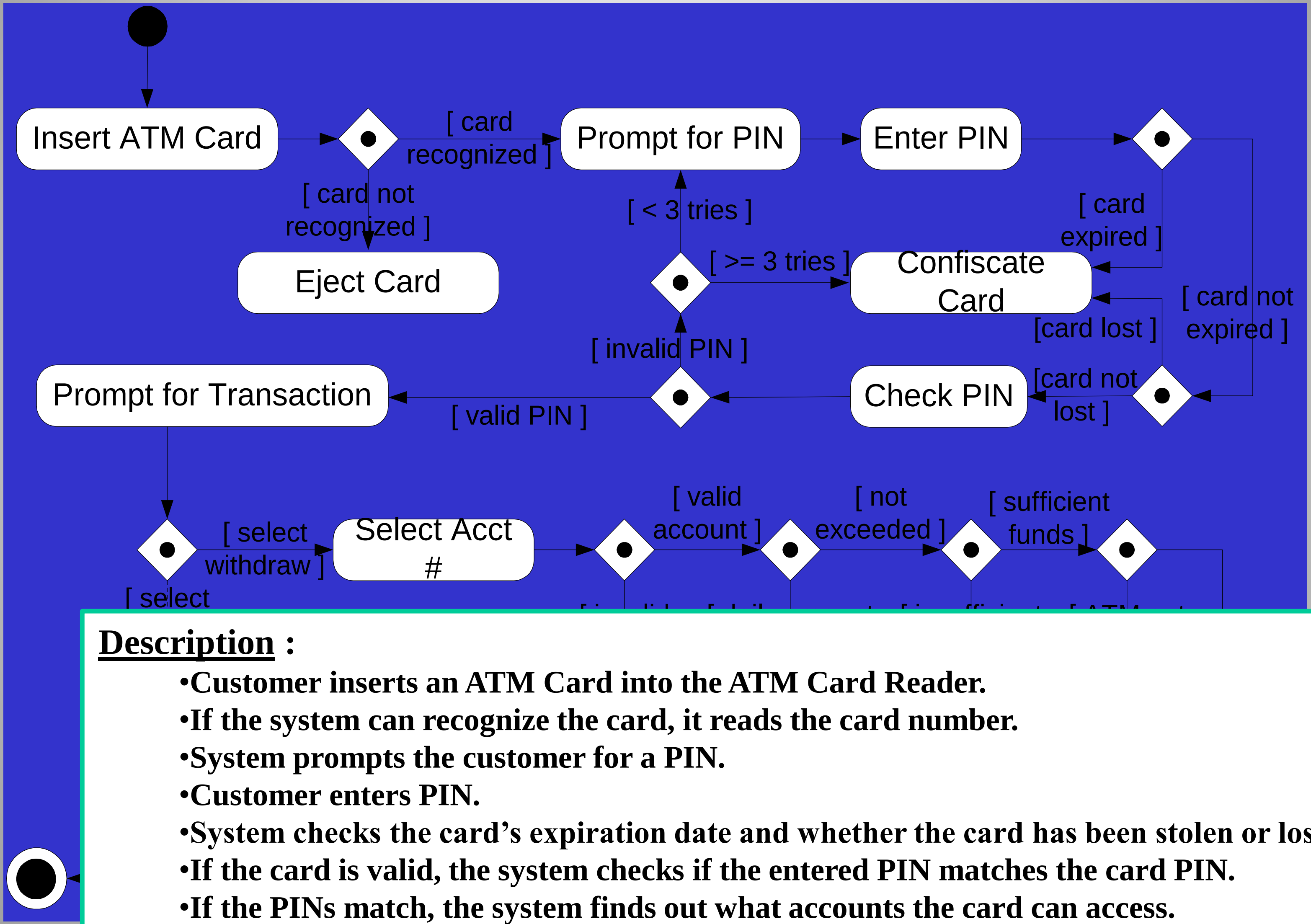


Description :

- •Customer inserts ar
- •If the system can re
- •System prompts the
- •Customer enters PI
- •System checks the c
- •If the card is valid, t
- •If the PINs match, t
- •…



ATM Withdraw Activity Graph



Description :

- Customer inserts an ATM Card into the ATM Card Reader.
- If the system can recognize the card, it reads the card number.
- System prompts the customer for a PIN.
- Customer enters PIN.
- System checks the card's expiration date and whether the card has been stolen or lost.
- If the card is valid, the system checks if the entered PIN matches the card PIN.
- If the PINs match, the system finds out what accounts the card can access.
- …

Covering Activity Graphs

- Node Coverage
  - Inputs to the software are derived from labels on nodes and predicates
  - Used to form test case values
- Edge Coverage
- Data flow techniques do not apply
- Scenario Testing
  - Scenario : A complete path through a use case activity graph
  - Should make semantic sense to the users
  - Number of paths often finite
  - If not, scenarios defined based on domain knowledge
  - Use "specified path coverage," where the set S of paths is the set of scenarios

Summary of Use Case Testing

- Use cases are defined at the requirements level
- Can be very high level
- UML Activity Diagrams encode use cases in graphs
  – Graphs usually have a fairly simple structure
- Requirements-based testing can use graph coverage
  – Straightforward to do by hand
  – Specified path coverage makes sense for these graphs