



UNIVERSITY OF  
CALGARY

# SENG 637

## Dependability and Reliability of Software Systems

### Chapter 5B: White-box Testing

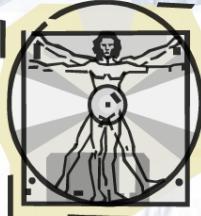
### Data-flow Coverage & Coverage Tools



Department of Electrical & Software Engineering, University of Calgary

B.H. Far (far@ucalgary.ca)

<http://people.ucalgary.ca/~far>



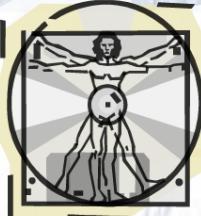
# Contents

- Preliminary
- Control flow
  - Statement/Node/Line Coverage
  - Decision/Edge/Branch Coverage
  - Condition Coverage
  - Path Coverage
- Data flow
- Coverage tools



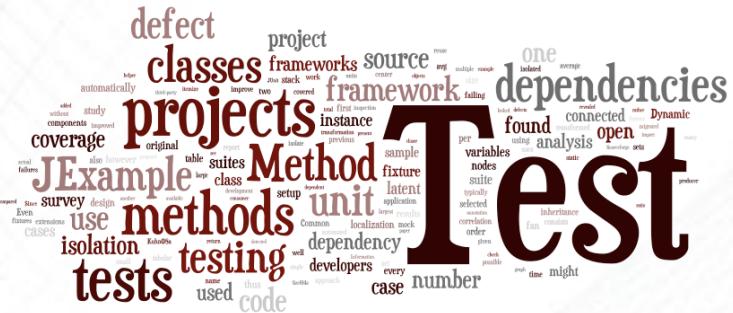
**Our focus in this Chapter**





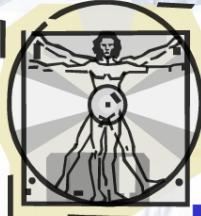
# What We've Learnt So Far?

- Statement/Decision/Condition/Path coverage → using Control Flow Graph (CFG)
- Acceptance criteria:
  - Statement, decision, basic and compound condition adequacy, MC/DC
- **Deficiencies:**
  - Node and edge coverage don't test interactions
  - Path-based criteria require impractical number of test cases and only a few paths uncover additional faults
- Need to distinguish “*important*” execution paths



# Data Flow Coverage

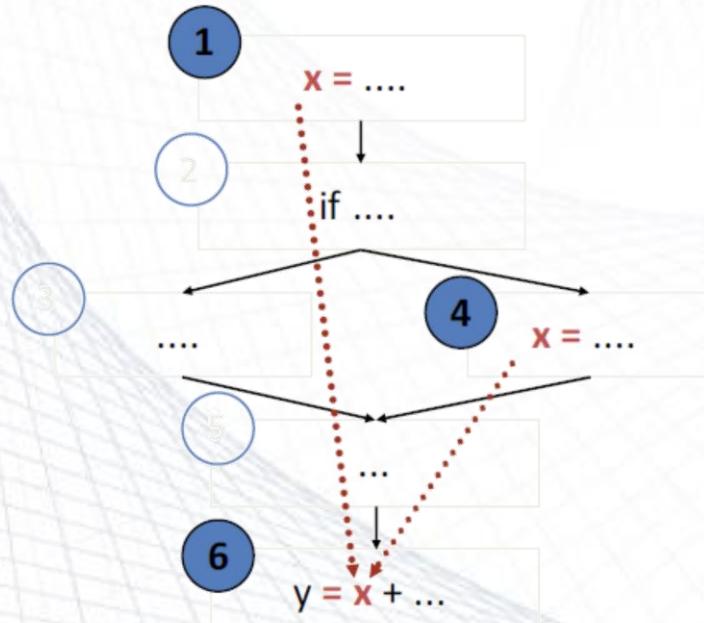
# How to enhance a test suite by finding “meaningful” execution paths to test?



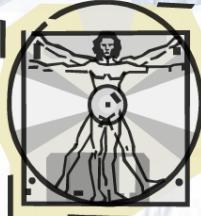
# Motivation

## ■ Intuition: Statements interact through *data flow*

- Value computed in one statement, used in another
- Bad value computation revealed only when it is used



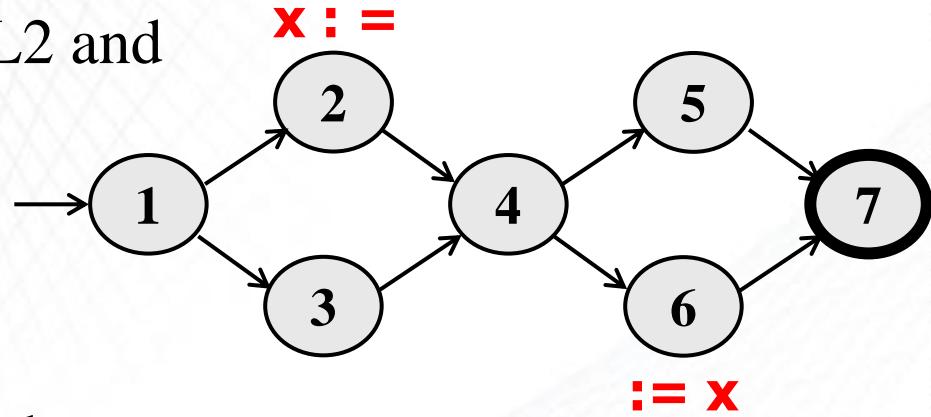
- Value of  $x$  at node 6 could be computed using its definition at node 1 or node 4
- We must test any path between (1,6) or (4,6) to be sure that value of  $x$  at 6 is calculated correctly
- Each of (1,6) or (4,6) are called “def-use” pairs (or simply “DU” pair)
- Def (write) at 1 and 4; Use (read) at 6



# Data Flow: What is?

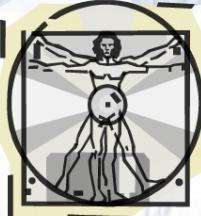
- Suppose we define  $x$  in L2 and have ref to it in L6

**Branches tested:**  
[1, 2, 4, 5, 7],  
[1, 3, 4, 6, 7]



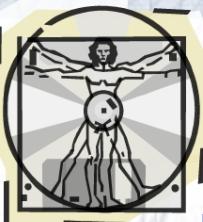
- The two branches tested do not exercise the relationship between the definition of  $x$  in L2 and the use of  $x$  in L6
- Does this cause a problem?

**We need to know where the value for a given variable is defined and where it is used (correctly)**



# Data-flow-based Testing

- **Basic idea:** test the connections between variable **definitions** (“write”) and variable **uses** (“read”)
- Starting point: **variation of** the control flow graph (CFG) annotated with location of defined and used variables
- This will be called data flow graph (DFG)
  - Set **def (n)** : contains variables that are defined at node **n** (i.e. written)
  - Set **use (n)** : contains variables that are used at node **n** (i.e. read)



# Example 1

- L<sub>7</sub>, L<sub>8</sub>, L<sub>9</sub>: define objects and variables
- At L<sub>16</sub>, **i** is both defined and used
- At L<sub>21</sub> **day** defined
- At L<sub>22</sub> and L<sub>25</sub> **day** used
- Etc.

**There is a possibility in some programs that a “used” value is modified somewhere on the way between “def” and “use” and that may cause a program Failure!**

```
01. import java.util.*;  
02. public class CalendarTest  
03. {  
04. public static void main(String[] args)  
05. {  
06. // construct d as current date  
07. GregorianCalendar d = new GregorianCalendar();  
08. int today = d.get(Calendar.DAY_OF_MONTH);  
09. int month = d.get(Calendar.MONTH);  
10. // set d to start date of the month  
11. d.set(Calendar.DAY_OF_MONTH, 1);  
12. int weekday = d.get(Calendar.DAY_OF_WEEK);  
13. // print heading  
14. System.out.println("Sun Mon Tue Wed Thu Fri Sat");  
15. // indent first line of calendar  
16. for (int i = Calendar.SUNDAY; i < weekday; i++)  
17. System.out.print(" ");  
18. do  
19. {  
20. // print day  
21. int day = d.get(Calendar.DAY_OF_MONTH);  
22. if (day < 10) System.out.print(" ");  
23. System.out.print(day);  
24. // mark current day with *  
25. if (day == today)  
26. System.out.print("* ");  
27. else  
28. System.out.print(" ");  
29. // start a new line after every Saturday  
30. if (weekday == Calendar.SATURDAY)  
31. System.out.println();  
32. // advance d to the next day  
33. d.add(Calendar.DAY_OF_MONTH, 1);  
34. weekday = d.get(Calendar.DAY_OF_WEEK);  
35. }  
36. while (d.get(Calendar.MONTH) == month);  
37. // the loop exits when d is day 1 of the next month  
38. // print final end of line if necessary  
39. if (weekday != Calendar.SUNDAY)  
40. System.out.println();  
41. }  
42. }
```



# Need for Data Flow Testing

- Let's see the need for data-flow-based testing

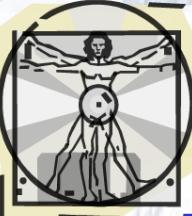
```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)          // x neq 0
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)          // y neq 0
9:         z=z/x;
10:    else
11:        z=z*x;
12:    return z;
13: }
```

Q. What happens if at execution time  
we get (x=0 and y=1)?

Test	x	y	z
$t_1$	0	0	0.0
$t_2$	1	1	1.0

- Here is a MC/DC-adequate test set
  - i.e. coverage=100%
- Exercise:** Verify that it is MC/DC-adequate

Note where the value for x is defined and then used may belong to different paths not traversed by the test suite

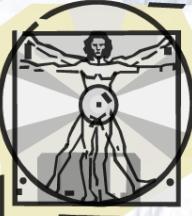


# Need for Data Flow Testing

- The given test suite does not reveal the fault (i.e. division by zero)
- Neither of the two test cases forces the use of x at L<sub>9</sub>, defined on L<sub>1</sub>, and possibly with the value of x=0
- We need a test for a path between L<sub>1</sub>-L<sub>9</sub> and L<sub>1</sub>-L<sub>11</sub>
- In larger programs this will be more problematic

```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)      // x neq 0 ← Use x
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)      // y neq 0
9:         z=z/x;          ← Use x
10:    else
11:        z=z*x;          ← Use x
12:    return z;
13: }
```

Test	x	y	z
$t_1$	0	0	0.0
$t_2$	1	1	1.0



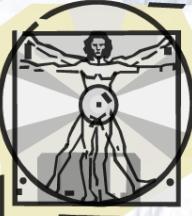
# Need for Data Flow Testing

- To do so, we need a test case that causes condition at L<sub>8</sub> to be true (e.g. y=1), and x=0
- An MC/DC-adequate test does not force the execution of this path with x=0 and hence the “divide by zero” will not be detected
- Again, consider large systems with thousands LOC

Therefore, %100 MC/DC coverage cannot guarantee that the program dose not fail

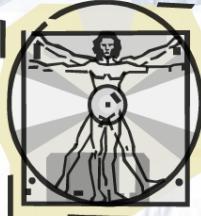
```
1: int calculate(int x, int y)
2: {
3:     double z=0;
4:     if (x<>0)      // x neq 0
5:         z=z+y;
6:     else
7:         z=z-y;
8:     if (y<>0)      // y neq 0
9:         z=z/x;           ← Use x
10:    else
11:        z=z*x;           ← Use x
12:    return z;
13: }
```

Test	x	y	z
$t_1$	0	0	0.0
$t_2$	1	1	1.0



# Data Flow Analysis - Definition

- In data flow analysis, we focus on paths that are significant for the data flow in the program
- Focus of testing is on the assignment of values to objects/variables and their uses
- Analysis of occurrences of variables:
  - **Definition occurrence:** a value is written (bound) to a variable
  - **Use occurrence:** value of a variable is **read** (referred)
    - **Predicate use (p-use):** a variable is used to decide whether a predicate **evaluates** to true or false
    - **Computational use (c-use):** **compute** a value for defining other variables or output values



# Data Flow Analysis - Example

- Let's see an example

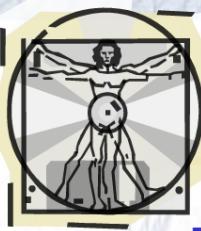
```
1. public int factorial(int x) {  
2.     int i, result = 1;  
3.     for (i=2; i<=x; i++) {  
4.         result = result * i;  
5.     }  
6.     return result;  
7. }
```

Q. How to ensure all def coverage  
and all use coverage?

Def-Use Table

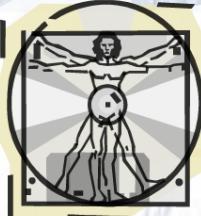


Variable	Def-line	Use-line



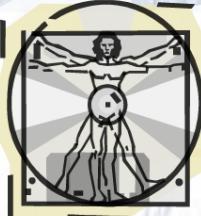
# Def and Use

- **Goal:** Try to ensure that values of the declared variables are assigned and used correctly
  - We annotate CFG for this purpose
- **def:** a location where a value for a variable is stored  
**use:** a location where a variable's value is accessed (read – evaluate – compute)
- **def(n):** The set of variable defined by node n
- **use(n):** the set of variable used by node n
  - We usually differentiate between computational (c-use) and predicate (p-use) type

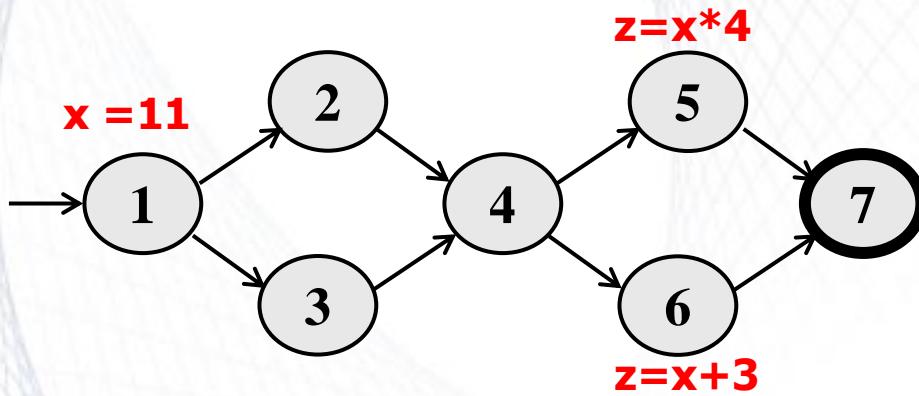


# Def and Use Path

- A **du-pair** is a pair of *definition* and *use* for some variable  $v$
- A **du-path** is a simple path where the initial node of the path is the only defining node of  $v$  in the path
  - $du(n_i, v)$ : the set of du-paths that start at  $n_i$  for variable  $v$
  - $du(n_i, n_j, v)$ : the set of du-paths from  $n_i$  to  $n_j$  for variable  $v$
- A **definition-clear path (def-clear)  $p$**  with respect to  $v$  is a sub-path where  $v$  is not defined at any of the nodes in  $p$
- **Reach**: if there is a def-clear path from the nodes  $m$  to  $p$  with respect to  $v$ , then the def of  $v$  at  $m$  reaches the use at  $p$



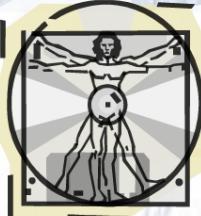
# Def and Use Path Example



defs:	$\text{def}(1) = \{ x \}$ $\text{def}(5) = \{ z \}$ $\text{def}(6) = \{ z \}$
uses:	$\text{use}(5) = \{ x \}$ $\text{use}(6) = \{ x \}$
du-pairs:	for $x$ (1,5) (1,6)

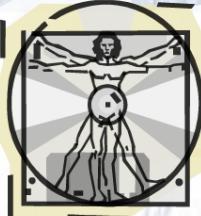
- du-path: for variable  $x$

$$\text{du}(1,x) = \{ [1,2,4,5], [1,3,4,5], [1,2,4,6], [1,3,4,6] \}$$

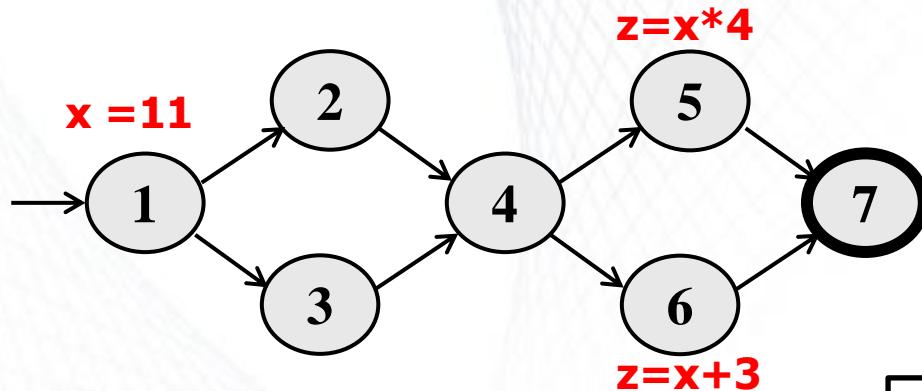


# Categorizing du-paths

- The test criteria for data flow will be defined as sets of du-paths, thus, we first categorize the du-paths according to:
- **def-path set**
  - $\text{du}(n_i, v)$ : All simple paths w.r.t. a given variable  $v$  defined in a given node
- **def-pair set**
  - $\text{du}(n_i, n_j, v)$ : All simple paths w.r.t. a given variable  $v$  from a given definition ( $n_i$ ) to a given use ( $n_j$ )



# du-path, du-pair Example



**du-path sets**

$\text{du}(1, x) =$

$\{[1,2,4,5],$   
 $[1,3,4,5],$   
 $[1,2,4,6],$   
 $[1,3,4,6]\}$

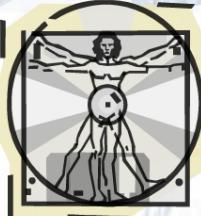
**du-pair sets**

$\text{du}(1, 5, x) =$

$\{[1,2,4,5],$   
 $[1,3,4,5]\}$

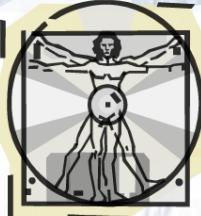
$\text{du}(1, 6, x) =$

$\{[1,2,4,6],$   
 $[1,3,4,6]\}$



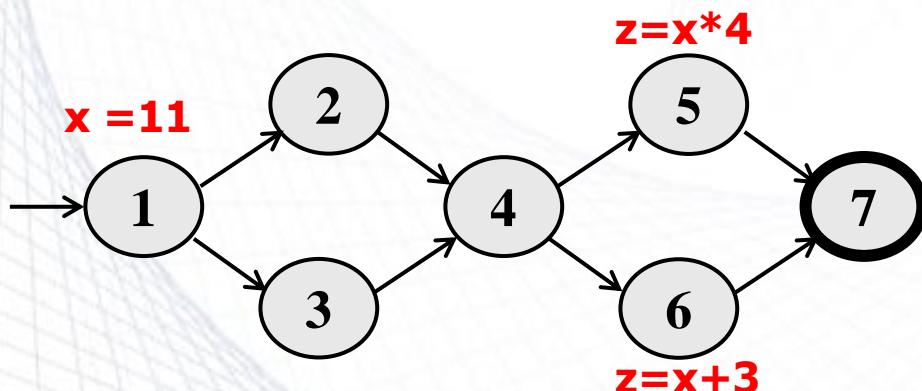
# Coverage Criteria

- DFG metrics:
  - All-Defs coverage (ADC)
    - Use every def
  - All-Uses Coverage (AUC)
    - Get to every use
  - All-DU-Paths Coverage (ADUPC)
    - Follow all du-paths



# All-Defs Coverage (ADC)

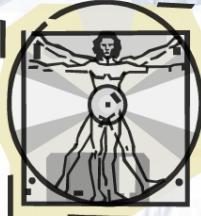
- For each set of du-paths  $S = du(n, v)$ , the test requirement (TR) contains at least one path  $d$  in  $S$ 
  - For each def, at least one use must be reached



**du-path sets**  
 $du(1, x) =$   
[[1,2,4,5],  
 [1,3,4,5],  
 [1,2,4,6],  
 [1,3,4,6]]

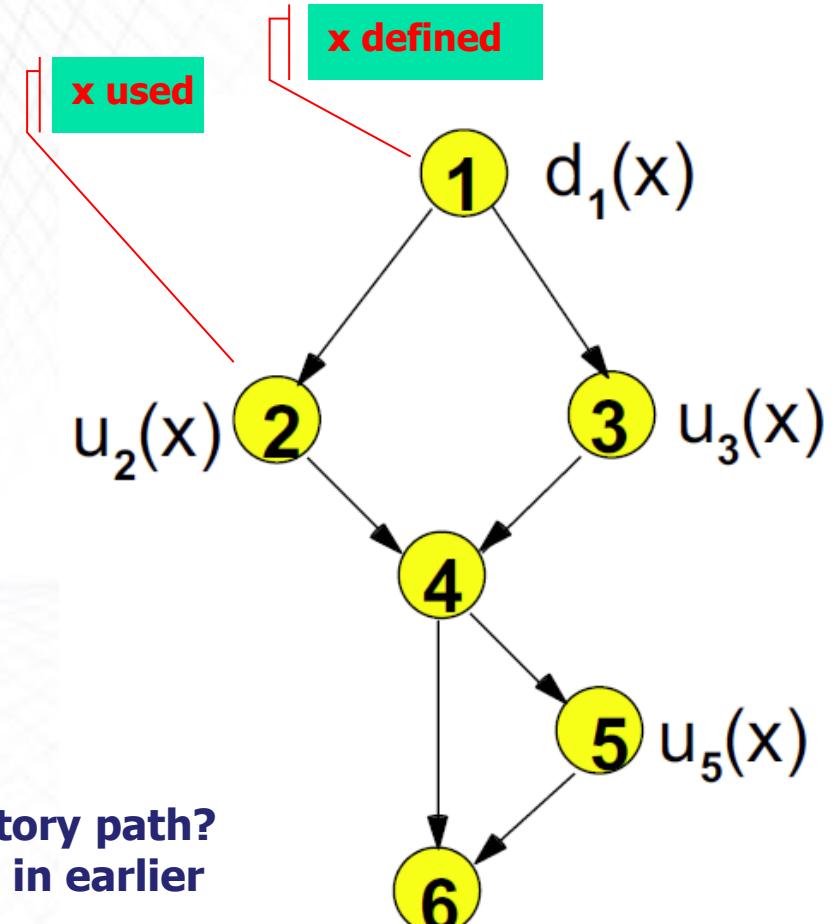
TR for  $x = \{1,2,4,5\}$   
Test path = {[1,2,4,5,7]}

To assure that anything defined is used later at least once!



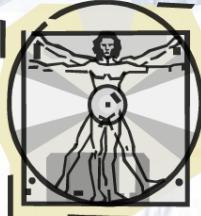
# All-Defs Coverage: Example

- Requires:  
 $d_1(x)$  to a use
- Satisfactory Paths:  
[1, 2, 4, 6] --or--  
[1, 3, 4, 6]



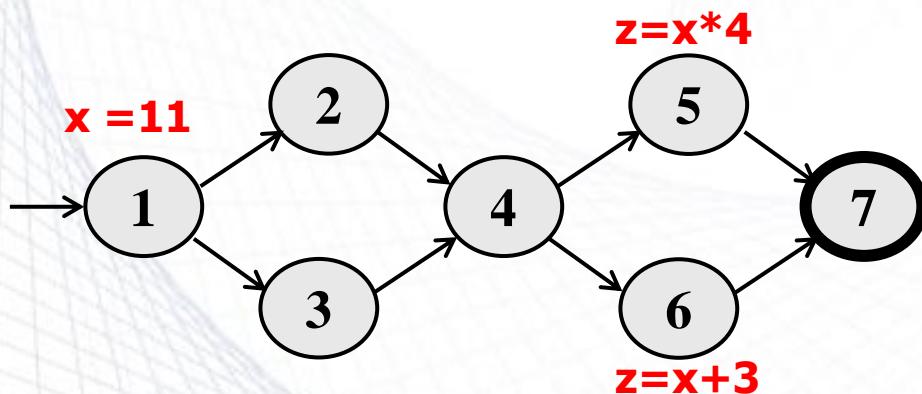
**Q. Why 5 is not part of ADC satisfactory path?**  
**A. Because value of x is overwritten in earlier nodes 2 or 3**

**Note:  $d_n(x)$  and  $u_n(x)$  mean x defined and used on node n, respectively**



# All-Uses Coverage (AUC)

- For each set of du-paths  $S = du(n_i, n_j, v)$ , TR contains at least one path  $d$  in  $S$ 
  - For each def, all uses must be reached



TR for  $x = \{[1,2,4,5], [1,2,4,6]\}$

Test paths =  $\{[1,2,4,5,7], [1,2,4,6,7]\}$

**du-pair sets**

$du(1, 5, x) =$

$\{[1,2,4,5],$   
 $[1,3,4,5]\}$

$du(1, 6, x) =$

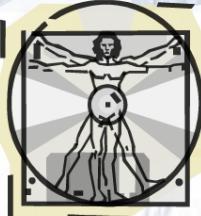
$\{[1,2,4,6],$   
 $[1,3,4,6]\}$

To assure that every use  
is covered!



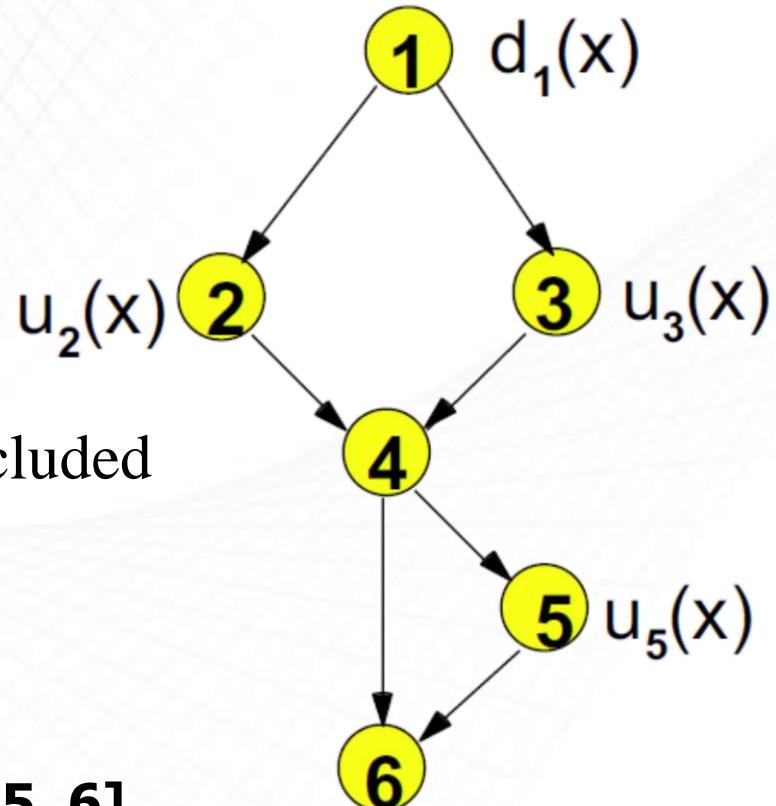
UNIVERSITY OF  
CALGARY

Sometimes we distinguish between predicate (dpu) and computation (dcu) path

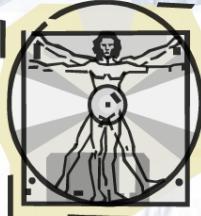


# All-Uses Coverage Example

- Requires:
  - $d_1(x)$  to  $u_2(x)$
  - $d_1(x)$  to  $u_3(x)$
  - $d_1(x)$  to  $u_5(x)$ 
    - Note that  $U_5(x)$  is included
- Satisfactory Path:
  - $[1, 2, 4, 6]$ ,
  - $[1, 3, 4, 6]$ ,
  - $[1, 2, 4, 5, 6]$  or  $[1, 3, 4, 5, 6]$

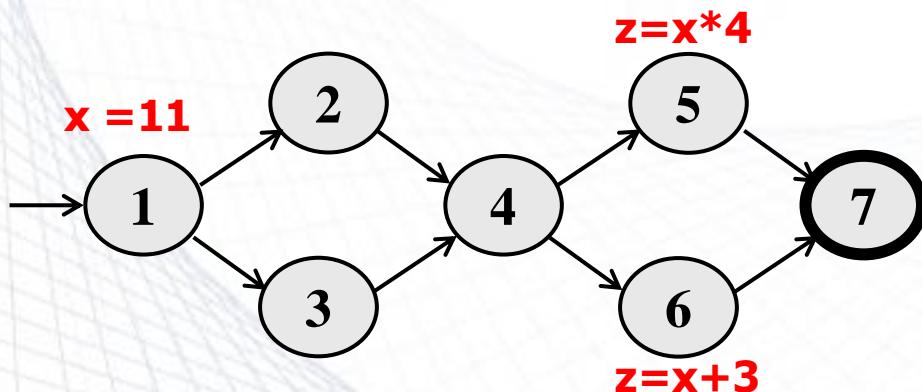


Note:  $d_n(x)$  and  $u_n(x)$  mean  $x$  defined and used on node  $n$ , respectively



# All-Du-Path Coverage (ADUPC)

- For each set of du-paths  $S = \text{du}(n_i, n_j, v)$ , TR contains every path  $d$  in  $S$ 
  - For each def-use pair, all paths between defs and uses must be covered
- Combination of ADC and AUC



**du-pair sets**

$\text{du}(1, 5, x) =$

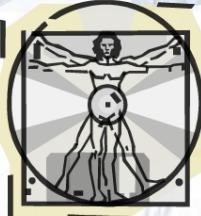
$\{[1,2,4,5],$   
 $[1,3,4,5]\}$

$\text{du}(1, 6, x) =$

$\{[1,2,4,6],$   
 $[1,3,4,6]\}$

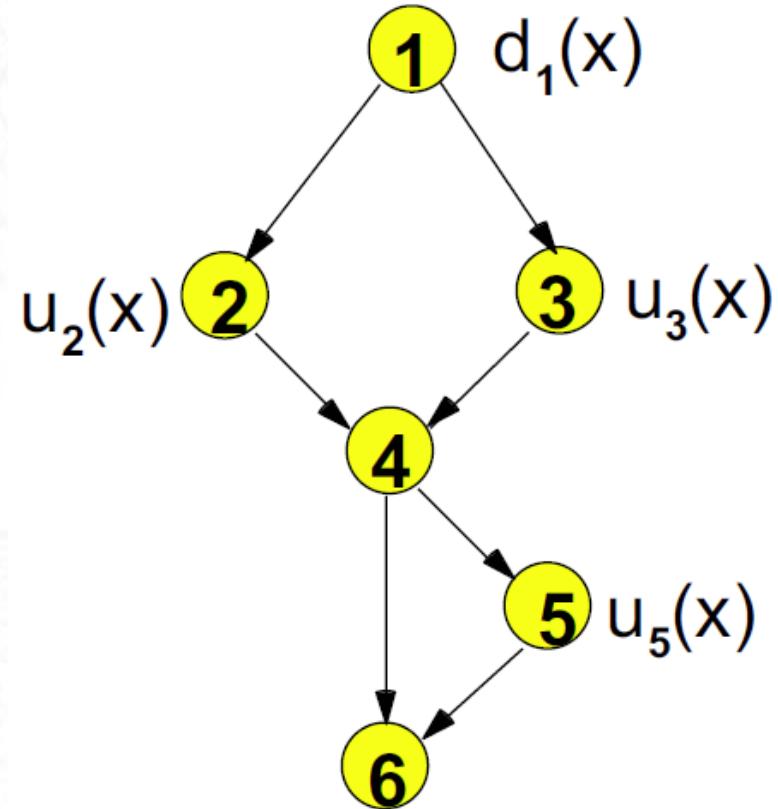
TR for  $x = \{[1,2,4,5], [1,3,4,5], [1,2,4,6], [1,3,4,6]\}$

Test paths =  $\{[1,2,4,5,7], [1,3,4,5,7], [1,2,4,6,7], [1,3,4,6,7]\}$

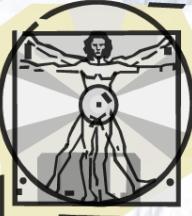


# All-Du-Path Coverage Example

- Requires:
  - All  $d_1(x)$  to  $u_2(x)$ : [1,2]
  - All  $d_1(x)$  to  $u_3(x)$  : [1,3]
  - All  $d_1(x)$  to  $u_5(x)$  :  
[1,2,4,5], [1,3,4,5]
  
- Satisfactory path:
  - [1, 2, 4, 5, 6]
  - [1, 3, 4, 5, 6]



Note:  $d_n(x)$  and  $u_n(x)$  mean  $x$  defined and used on node  $n$ , respectively

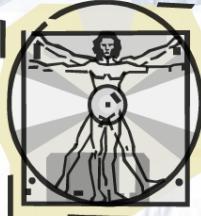


# Data Flow Based Testing

## Adequacy Criteria

- **All du-pairs:** Each du-pair is exercised by at least one test case
- **All du-paths:** Each *simple* (non looping) du-path is exercised by at least one test case
  - Remember that for each du-pair there can be several du-paths
- **All definitions:** For each definition, there is at least one test case which exercises a du-pair containing it
  - (Every defined value is used somewhere)
- Corresponding coverage fractions can also be defined

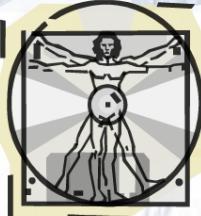
**All du-paths > All du-pairs > All definitions**



# Data Flow Based Testing

- How to enhance test suite using Data Flow based testing?
  - If a variable is defined at node-m and used at node-n, then the path between node m and n should be tested
  - If a variable is modified (c-use) within the path between m and n, say in node-q, then we should have separate tests for
    - m to q (called def-coverage)
    - q to n and m to n (called use-coverage)
  - The def+use coverage is combining both

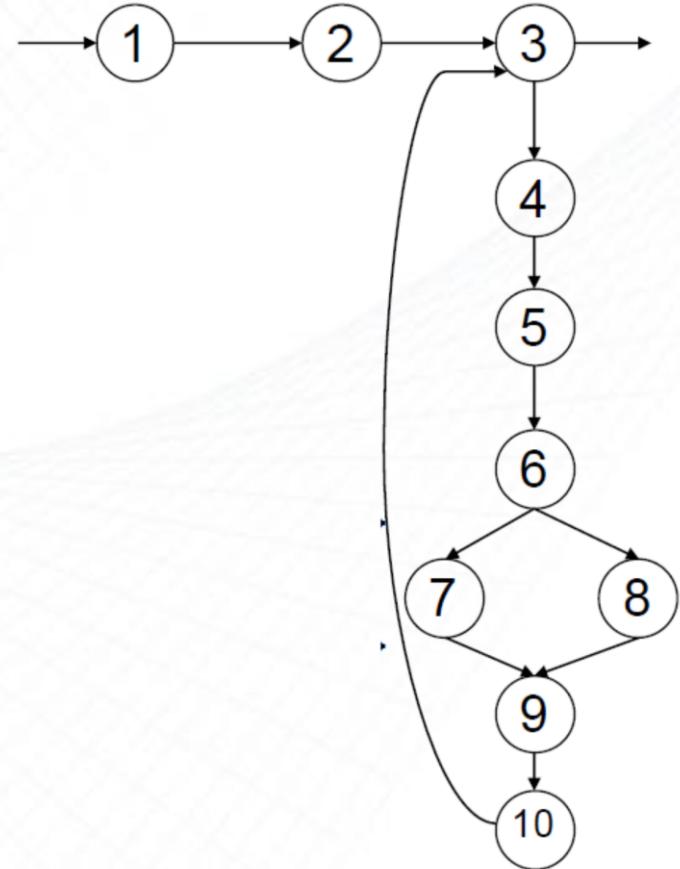
You can use the code itself, DFD and Def-Use table for this purpose

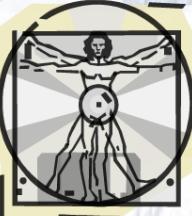


# Exercise: Def and Use

- Assume **y** is already initialized

```
1:      s:= 0;  
2:      x:= 0;  
3:      while (x<y) {  
4:          x:=x+3;  
5:          y:=y+2;  
6:          if (x+y<10)  
7:              s:=s+x+y;  
8:          else  
9:              s:=s+x-y;  
10:         endif  
11:     }
```





# Exercise: Reach

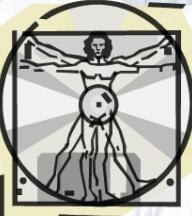
- A definition of variable **s** at node **n<sub>1</sub>** reaches node **n<sub>2</sub>** if and only if there is a path between **n<sub>1</sub>** and **n<sub>2</sub>** that does not contain a definition of **s**

```
DEF(1) := {s}, USE(1) := ∅
DEF(2) := {x}, USE(2) := ∅
DEF(3) := ∅, USE(3) := {x,y}
DEF(4) := {x}, USE(4) := {x}
DEF(5) := {y}, USE(5) := {y}
DEF(6) := ∅, USE(6) := {x,y}
DEF(7) := {s}, USE(7) := {s,x,y}

DEF(8) := {s}, USE(8) := {s,x,y}
DEF(9) := ∅, USE(9) := ∅
DEF(10) := ∅, USE(10) := ∅
```

```
1:   s := 0;
2:   x := 0;
3:   while (x < y) {
4:     x := x + 3;
5:     y := y + 2;
6:     if (x + y < 10)
7:       s := s + x + y;
8:     else
9:       s := s + x - y;
10:    endif
11: }
```

This is the Def-Use table



# Exercise: Def and Use Pairs

Reaches nodes 2, 3, 4, 5, 6, 7, 8, but not 9,10

For this definition, two DU pairs:  
1-7, 1-8

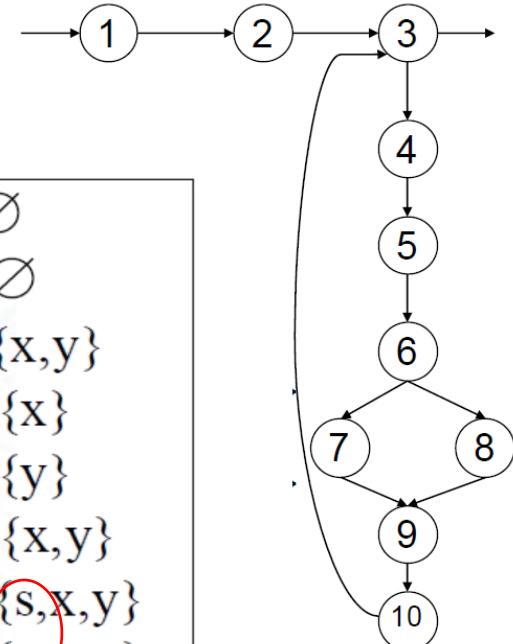
DU pair tests:

TC1:  $x=0, y=3$  (for L7)  
TC2:  $x=0, y=6$  (for L8)

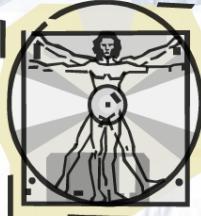
Note: there is no path between L7 and L8 so no need to test it

DEF(1) :=  $\{s\}$ , USE(1) :=  $\emptyset$   
DEF(2) :=  $\{x\}$ , USE(2) :=  $\emptyset$   
DEF(3) :=  $\emptyset$ , USE(3) :=  $\{x,y\}$   
DEF(4) :=  $\{x\}$ , USE(4) :=  $\{x\}$   
DEF(5) :=  $\{y\}$ , USE(5) :=  $\{y\}$   
DEF(6) :=  $\emptyset$ , USE(6) :=  $\{x,y\}$   
DEF(7) :=  $\{s\}$ , USE(7) :=  $\{s,x,y\}$   
DEF(8) :=  $\{s\}$ , USE(8) :=  $\{s,x,y\}$

This is the CFG

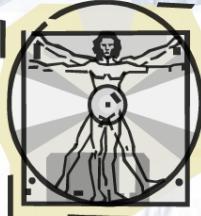


This is the Def-Use table



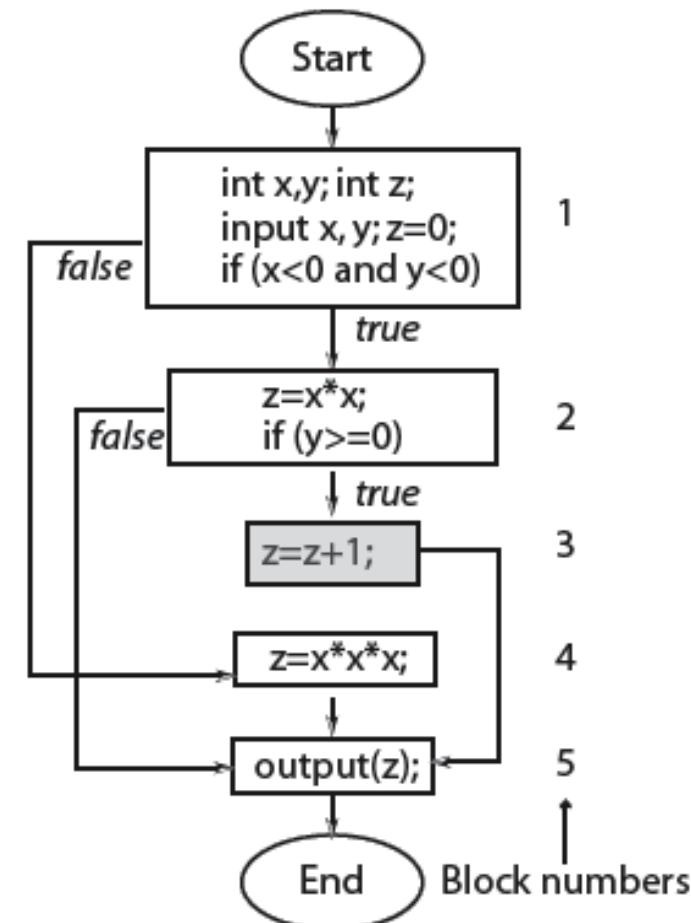
# Data Flow Graph (DFG)

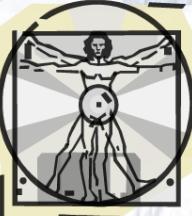
- DFG is a variation of CFG in which:
  - Nodes are annotated by data “def” and data “use”
    - “use” can be further broken down to predicate “p-use” and computation “c-use”
  - Arcs are as before annotated by “predicate conditions” as True or False
- Def-use table is usually good enough to identify paths to be tested



# Data Flow Graph (DFG): Example

- Let's derive the DFG for this program given its code/flowchart
- First the def-use table  
→Your exercise
- Then, the data flow graph (DFG)  
→Next slides

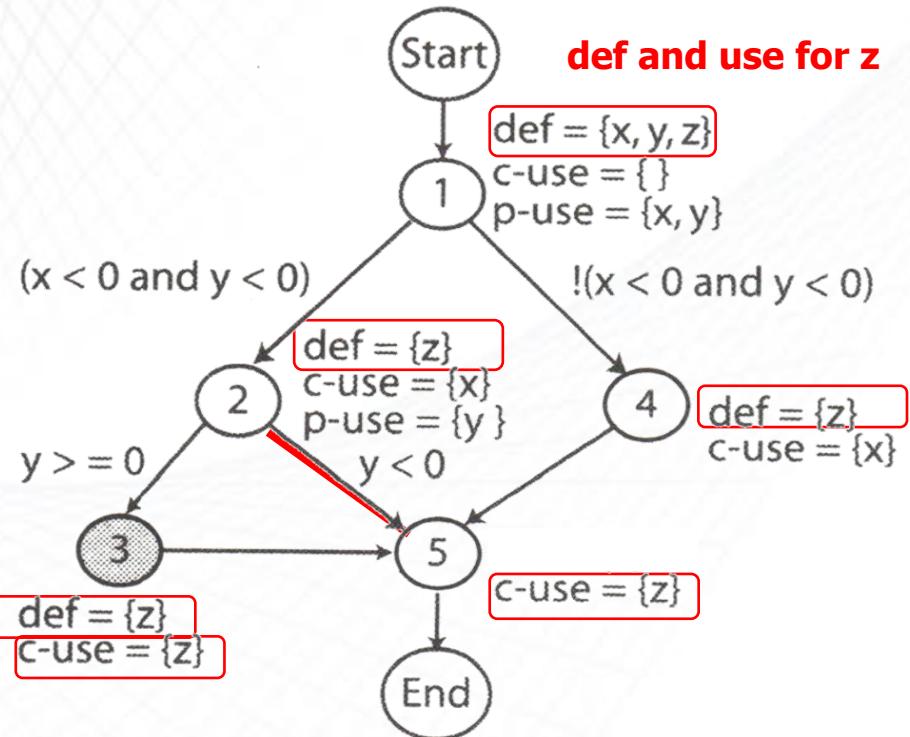




# Definition-clear (def-clear) Path

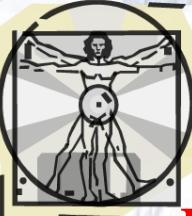
- Any path starting from a node at which a variable is defined and ending at a node at which that variable is used, without redefining the variable anywhere else along the path, is a **def-clear path** for that variable

This is the DFG



Therefore, any values set for z at L1 is redefined at L2, L3, L4 and L5, but anything defined at L2 or L4, is still live at L5

c-use: computational use  
p-use: predicate use

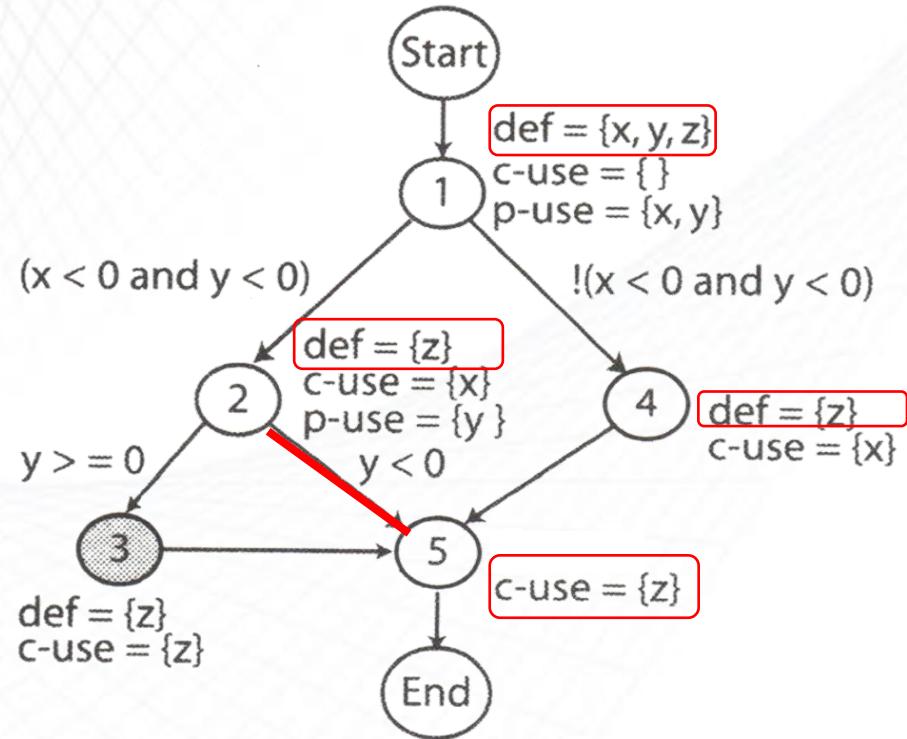


# Definition-clear (def-clear) Path

## Example:

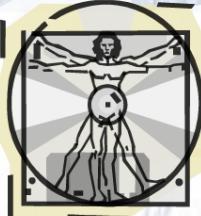
- Path 2-5 is def-clear for variable z defined at node 2 and used at node 5
- Path 1-2-5 is NOT def-clear for variable z defined at node 1 and used at node 5 (since it is redefined at node 2)
- The definition of z at node 2 is live at node 5,
- ... while the definition of z at node 1 is not live at node 5

→ **Exercise: derive ADC and AUC tests for z**



c-use: computational use

p-use: predicate use



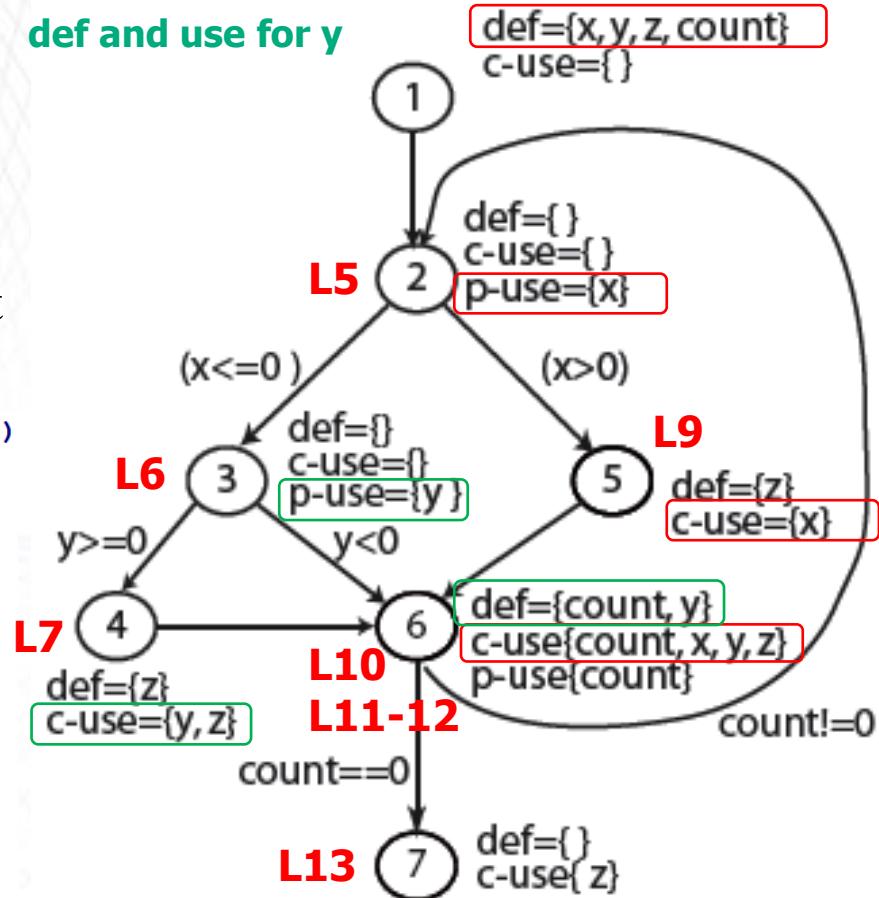
# Example 2: Def-clear Path

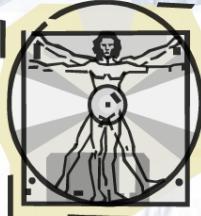
- Draw DFG for the program given below
- Find the def-clear paths for x and y and z
- Which definitions are live at node 4?

```

1  double calculate(int x, int y, int count)
2  {
3      double z=0.0;
4      do {
5          if (x<=0)
6              if (y>=0)
7                  z=y*z+1;
8          else
9              z=1/x;
10         y=(int) (x*y+z);
11         count--;
12     } while (count >0);
13     return z;
14 }
```

**def and use for x**  
**def and use for y**



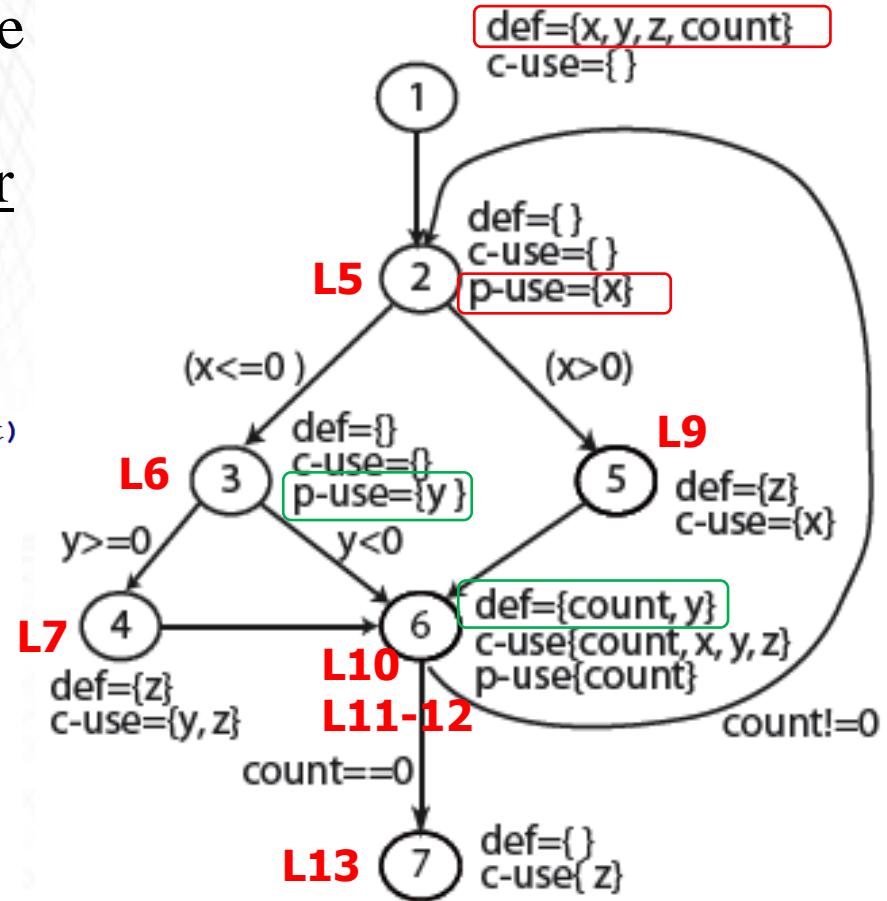


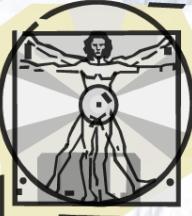
# Example 2: Def-use Pairs

- Definitions of a variable  $L_{11}$  and its use at  $L_{12}$  constitute a def-use pair
- $(L_{11} \text{ and } L_{12})$  can be the same, e.g.,  $y=++x;$

```

1  double calculate(int x, int y, int count)
2  {
3      double z=0.0;
4      do {
5          if (x<=0)
6              if (y>=0)
7                  z=y*x+1;
8          else
9              z=1/x;
10         y=(int) (x*y+z);
11         count--;
12     } while (count >0);
13     return z;
14 }
```

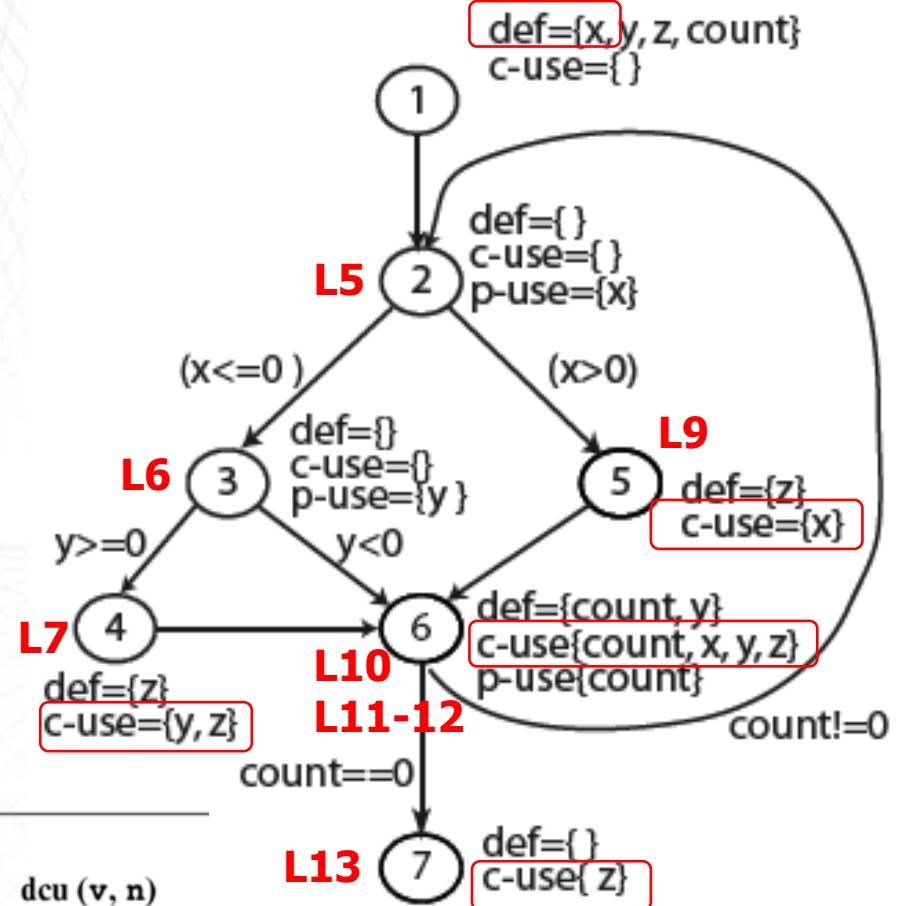




# Example 2: Def-use Measure

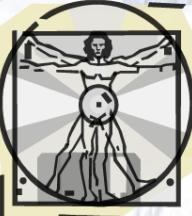
- **Definition c-use pairs  $dcu(x)$ :**

denotes the set of all nodes using variable  $x$  from the live definitions of the variable at a given previous node  $i$



Therefore, tests for (1-5), (1-6) and (5-6) should be added

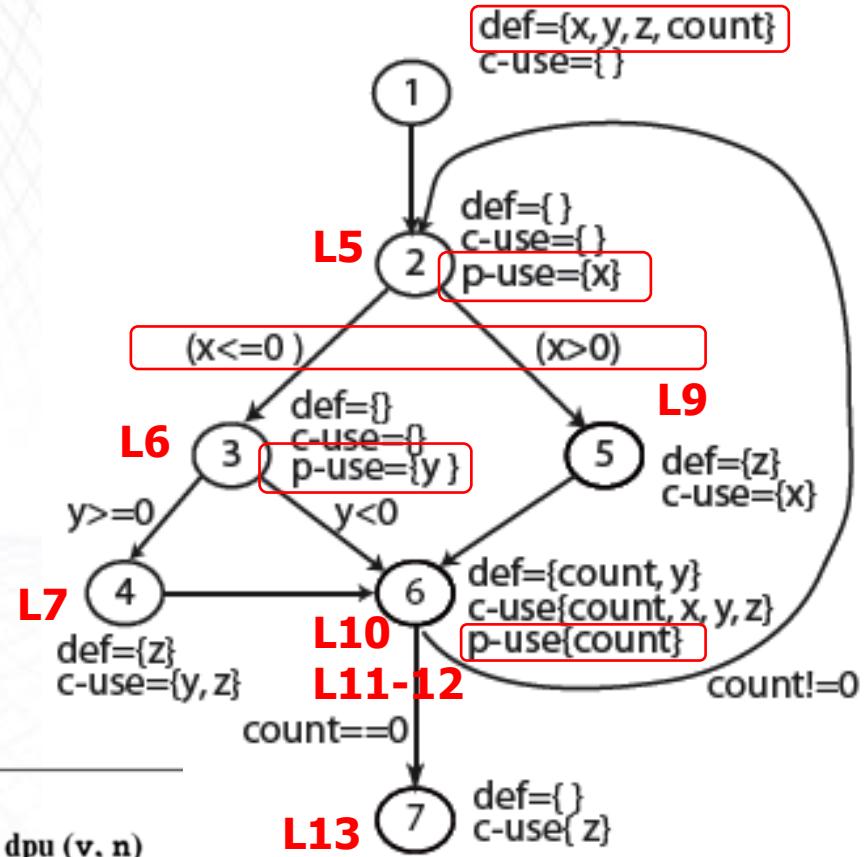
Variable ( $v$ )	Defined at node ( $n$ )	$dcu(v, n)$
x	1	{5, 6}



# Example 2: Def-use Measure

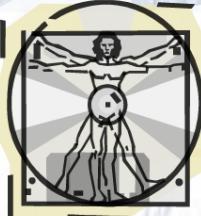
- **Definition p-use pairs  $dpu(x)$ :**

denotes the set of all edges  $(k, l)$  such that there is a def-clear path from node  $i$  to edge  $(k, l)$  and  $x$  is used at node  $k$



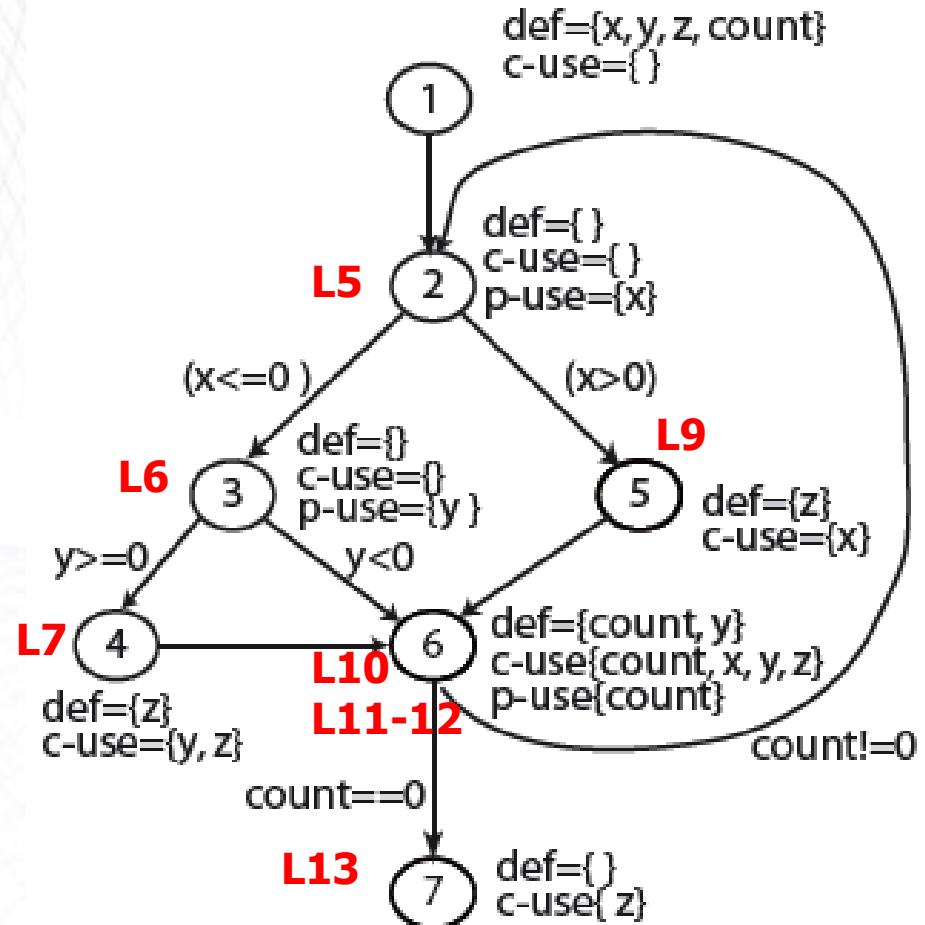
Therefore, tests for (2-3) and (2-5) should be added

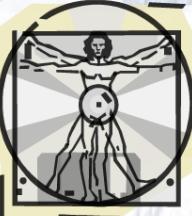
Variable ( $v$ )	Defined at node ( $n$ )	$dcu(v, n)$	$dpu(v, n)$
$x$	1	{5, 6}	$\{(2, 3), (2, 5)\}$



# Example 2: Def-use Measure

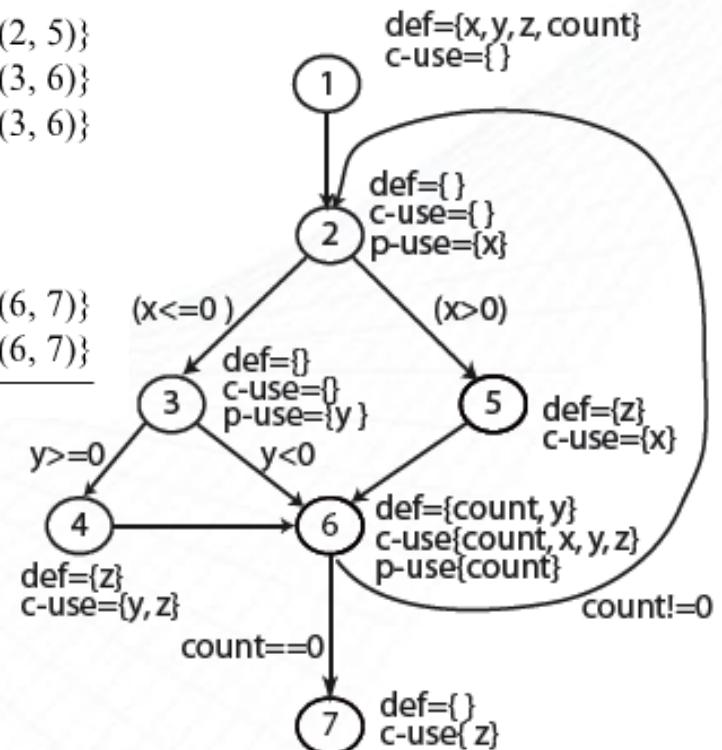
- Let's derive:
  - $dn(v)$
  - $dcu(v, n)$
  - $dpu(v, n)$

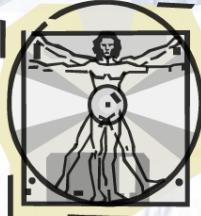




# Example 2: Answer

Variable (v)	Defined at node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7 }	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	{(6, 2), (6, 7)}
count	6	{6 }	{(6, 2), (6, 7)}





# Data-flow Coverage Measure

- To calculate data-flow test coverage, we should measure how many c-uses and p-uses we have in total

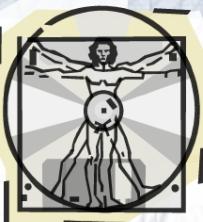
$$CU = \sum_{i=1}^n \sum_{\forall n} |dcu(v_i, n)|$$

$$PU = \sum_{i=1}^n \sum_{\forall n} |dpu(v_i, n)|$$

**Loop for variables**

**Loop for all nodes defining a variable**

- CU: total number of c-uses in a program
- PU: total number of p-uses
- Given a total of **n** variables  $v_1, v_2 \dots v_n$



# Data-flow Coverage: Example

- Calculate CU and PU

$$CU = \sum_{i=1}^n \sum_{\forall n} |dcu(v_i, n)|$$

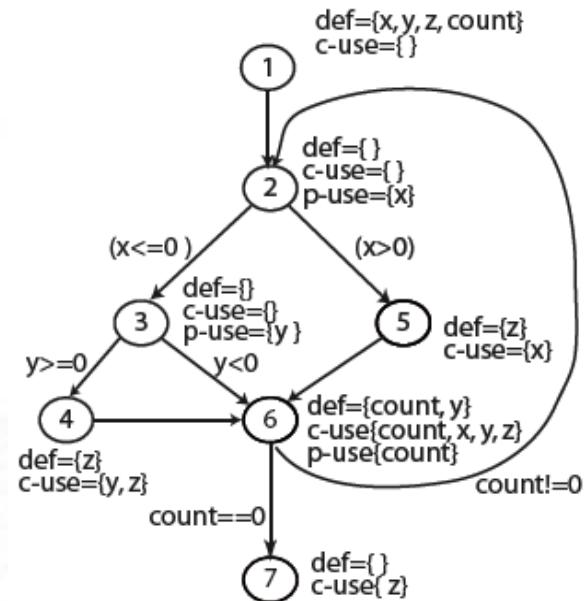
$$PU = \sum_{i=1}^n \sum_{\forall n} |dpu(v_i, n)|$$

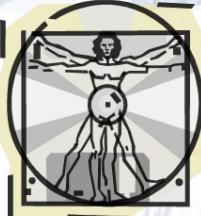
Variable (v)	Defined in node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{ }
z	4	{4, 6, 7}	{ }
z	5	{4, 6, 7}	{ }
count	1	{6}	
count	6	{6}	{(6, 2), (6, 7)}

Total:

17

8



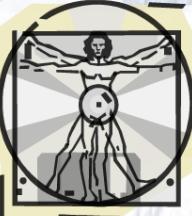


# All-uses Coverage Measure

- All-uses coverage is computed as

$$\frac{(CU_c + PU_c)}{((CU + PU) - (CU_f + PU_f))}$$

- Where CU is the total c-uses,  $CU_c$  is the number of c-uses covered,  $PU_c$  is the number of p-uses covered,  $CU_f$  the number of infeasible c-uses and  $PU_f$  the number of infeasible p-uses
- It is considered adequate with respect to the all-uses coverage criterion if the c-use coverage is % 100



# All-uses Coverage – Example

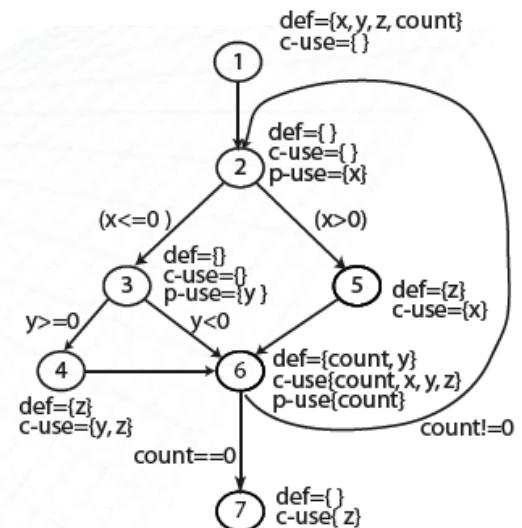
**Exercise:** For the following DFG, show by analysis whether  $T=\{TC1, TC2\}$  is adequate w.r.t. to all-uses coverage. Calculate all-uses coverage ratio. If coverage ratio is not 100%, what def-use pairs need to be covered, and what test cases should be added to cover them?

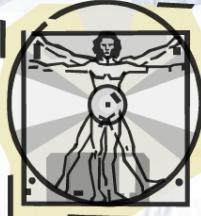
- TC1:< $x=5, y=-1, \text{count}=1$ >
- TC2:< $x=-2, y=-1, \text{count}=3$ >

Variable (v)	Defined in node (n)	dcu (v, n)	dpu (v, n)
x	1	{5, 6}	{(2, 3), (2, 5)}
y	1	{4, 6}	{(3, 4), (3, 6)}
y	6	{4, 6}	{(3, 4), (3, 6)}
z	1	{4, 6, 7}	{}
z	4	{4, 6, 7}	{}
z	5	{4, 6, 7}	{}
count	1	{6}	
count	6	{6}	{(6, 2), (6, 7)}

```

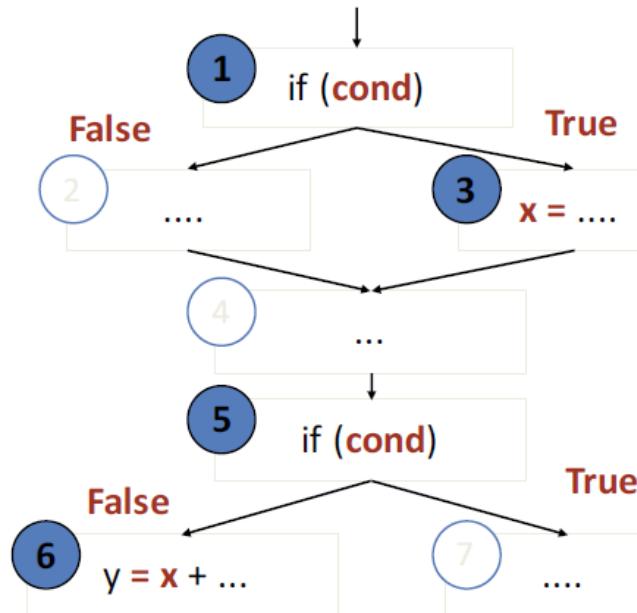
1  double calculate(int x, int y, int count)
2  {
3      double z=0.0;
4      do {
5          if (x<=0)
6              if (y>=0)
7                  z=y*z+1;
8          else
9              z=1/x;
10         y=(int) (x*y+z);
11         count--;
12     } while (count >0);
13     return z;
14 }
```

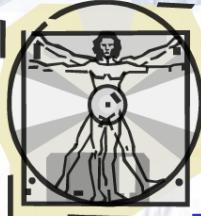




# Infeasibility

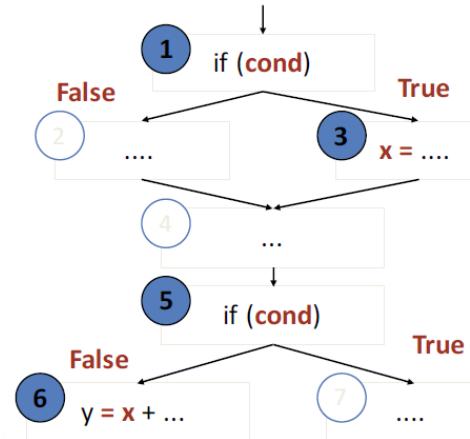
- Suppose *cond* has not changed between 1 and 5
  - Or the conditions could be different, but the first implies the second
- Then (3,6) is not a (feasible) DU pair
  - But it is difficult or impossible to determine which pairs are infeasible
- Infeasible test obligations are a problem
  - No test case can cover them

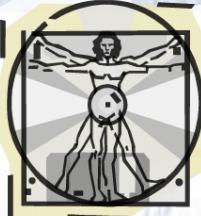




# Infeasibility

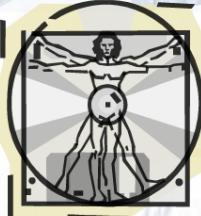
- The path-oriented nature of data flow analysis makes the infeasibility problem especially relevant
  - Combinations of elements matter!
  - Impossible to (precisely) distinguish feasible from infeasible paths
  - More paths = more work to check manually.
- In practice, reasonable coverage is often, not always achievable
  - Number of paths is linear (often) or exponential (worst case)
- All DU paths is more often impractical





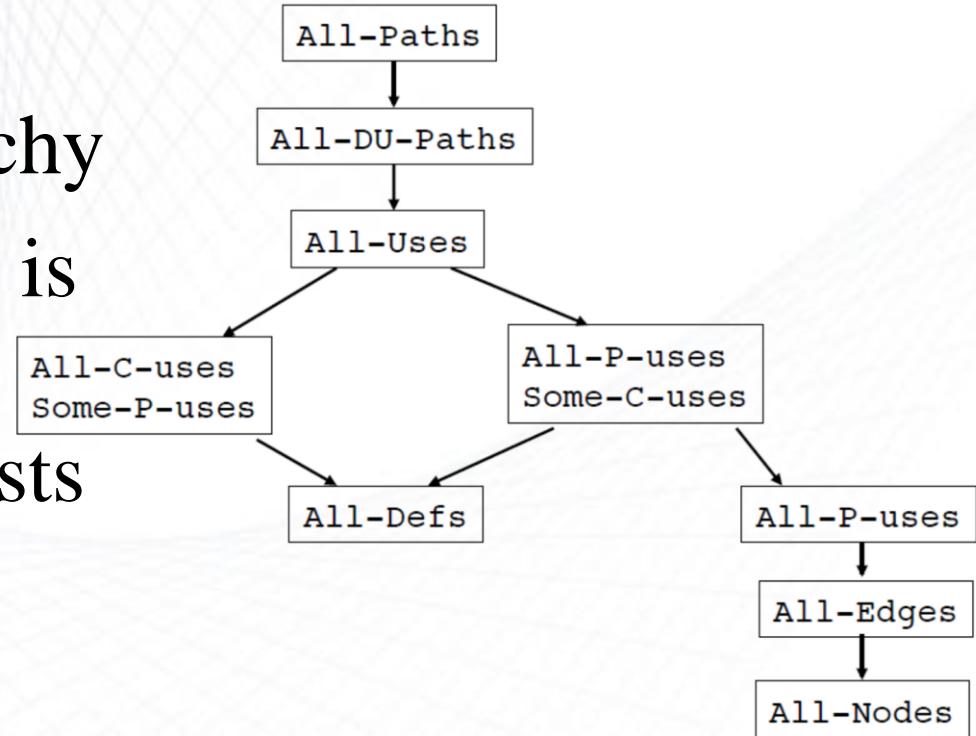
# Slice Based Testing

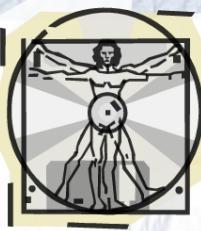
- Kind of data flow testing
- What is a slice:
  - Given a program  $P$ , program graph  $G(P)$  and set of variables (in  $P$ )  $V$
  - Slice on  $V$  at statement (fragment)  $n - S(V, n)$
  - $S(V, n)$  is set of node numbers of all statements in  $P$  prior to  $n$  that contribute to values of variables in  $V$  at  $n$
  - Exclude all non-executable statements
- Testing focus will be on the slice rather than the whole program



# Subsumption

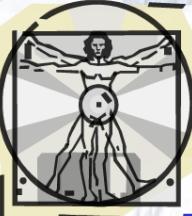
- Rapps-Weyuker data flow hierarchy
- The higher level is stronger but requires more tests





# Advantages of Data Flow Testing

- Data Flow testing helps us to pinpoint any of the following issues:
  - A variable that is declared but never used within the program
  - A variable that is used but never declared
  - A variable that is defined multiple times before it is used
  - Deallocating a variable before it is used



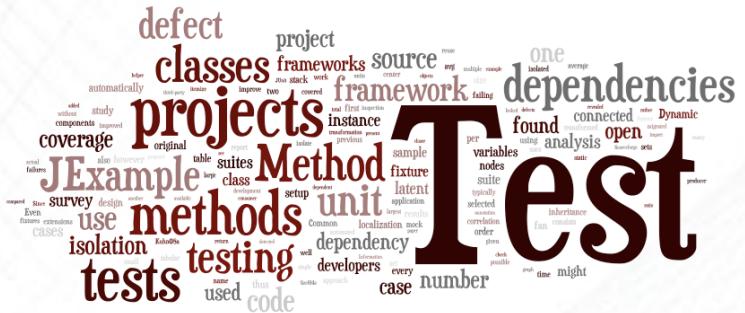
# Summary - Data Flow Testing

- Data flow testing attempts to distinguish “important” paths:  
Interactions between statements
  - Intermediate between simple statement and branch coverage and more expensive path-based structural testing
- Cover Def-Use (DU) pairs: From computation of value to its use
  - Intuition: Bad computed value is revealed only when it is used
  - Levels: All DU pairs, all DU paths, all defs (some use)
- Limits: Aliases, infeasible paths
  - Worst case is bad (undecidable properties, exponential blow up of paths), so pragmatic compromises are required

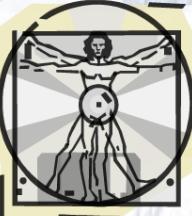


UNIVERSITY OF  
CALGARY

# Section 5



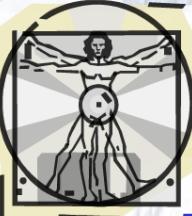
# White Box Coverage Tools



# Measuring Test Coverage

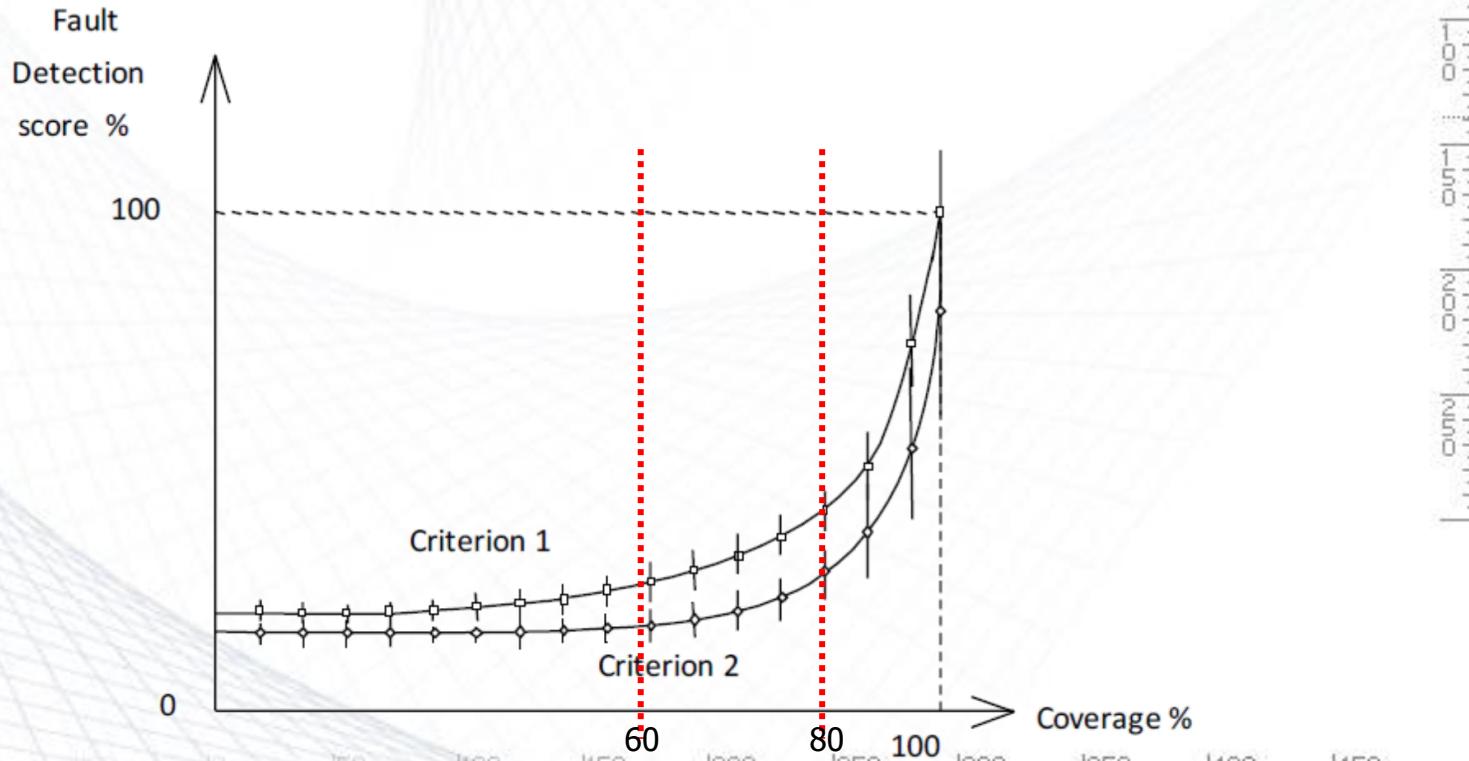
---

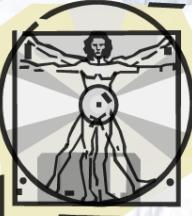
- One advantage of coverage criteria is that it can be measured *automatically* (usually)
  - To control testing progress
  - To assess testing completeness in terms of remaining faults and reliability
- Remember
  - High coverage is not a guarantee of fault-free software



# Analysis of Coverage Data

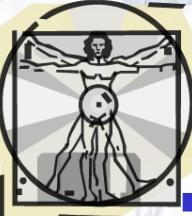
- Regardless of criterion used for WB testing, only higher coverage rates contribute to finding more faults ← e.g. try at least 60-80% coverage to find reasonable number of faults





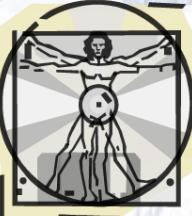
# Test Coverage: Benefits

- If you haven't exercised part of the code, it could contain a bug
- Any coverage criteria performs better than random test selection – especially DU-coverage
- Helps us in measuring how much unit testing has been done and how much is left
- Significant improvements occurred as coverage increased from 90% to 100%
- Helps us in test case design (deriving test cases)
- We can prioritize the most critical modules (units) of the system and target higher coverage for those critical modules, would mean more rigorous testing for them



# Test Coverage: Caveats

- Coverage tests that the code has been exercised (mostly in unit testing level), and that each unit has been verified
- But it does not tell us that we have built what the customer wanted
- If the logical structure of the code contains errors such as a missed case in a switch statement, the absence of code may not be detected
- 100% coverage alone is not a reliable indicator of the effectiveness of a test set – especially statement coverage
- Yet, 100% coverage is often unrealistic, especially for fine-grained coverage measures (e.g., condition coverage)
- Effort to achieve high coverage may be unjustified in large scale projects, in terms of time spent and producing unreliable test cases
- 60%-80% coverage is a “rule of thumb” (according to several surveys)



# Test Coverage Measurement

- Two usages:
- Will help us derive test cases
- And to also know the progress of test activities

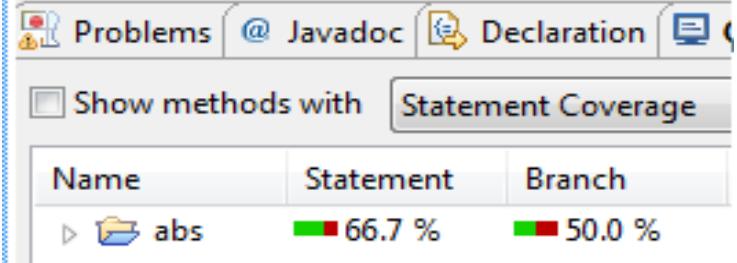
```
public class absTestSuite {  
  
    private absClass a;  
  
    @Before  
    public void setUp() throws Exception  
    {  
        a = new absClass();  
    }  
  
    @Test  
    public void testAbsPositive() {  
        assertTrue(a.abs(5)==5);  
    }  
}
```

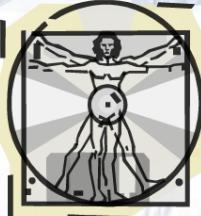
Tests



Manual Tests

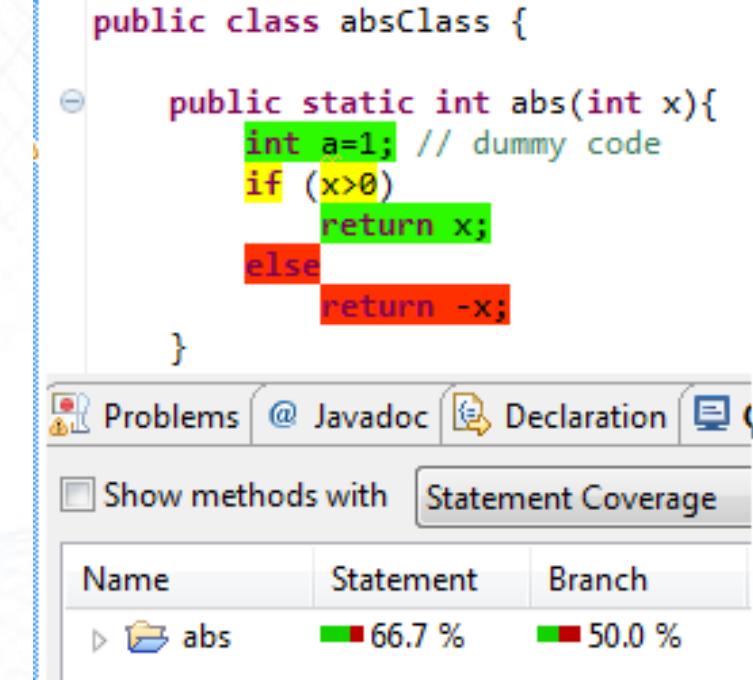
```
public class absClass {  
  
    public static int abs(int x){  
        int a=1; // dummy code  
        if (x>0)  
            return x;  
        else  
            return -x;  
    }  
}
```





# Test Coverage Tools

- Most established languages have solid test coverage tools available for them, but the depth of functionality differs significantly from one to another
- Python has sys.settrace to tell you directly which lines are executing
- Emma (for Java) has a ClassLoader which re-writes byte-code on the fly

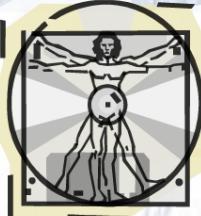


The screenshot shows a Java code editor with the following code:

```
public class absClass {  
    public static int abs(int x){  
        int a=1; // dummy code  
        if (x>0)  
            return x;  
        else  
            return -x;  
    }  
}
```

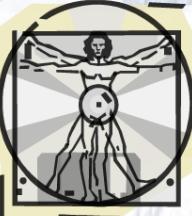
Below the code, the IDE interface includes tabs for Problems, Javadoc, Declaration, and a dropdown menu. A 'Statement Coverage' button is highlighted. A coverage summary table is displayed:

Name	Statement	Branch
abs	66.7 %	50.0 %

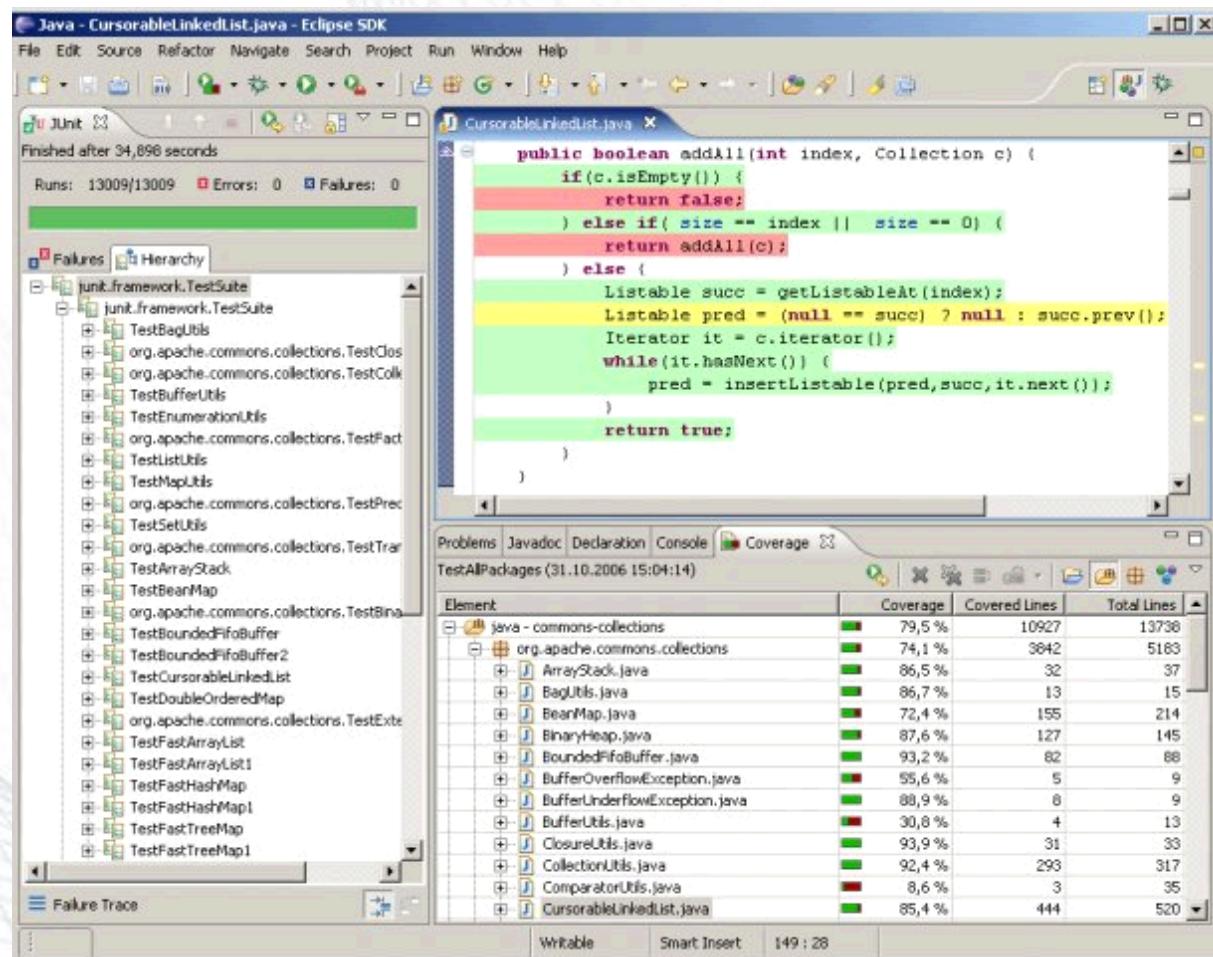


# Code Coverage Tools

- CodeCover (<http://codecover.org/>)
- EclEmma (<http://www.eclemma.org/>)
- Clover  
(<https://www.atlassian.com/software/clover>)
- Coverlipse (<http://coverlipse.sourceforge.net/>)
- JaCoCo (<http://www.eclemma.org/jacoco/>)
- Cobertura (<http://cobertura.github.io/cobertura/>)



# Code Coverage Tools

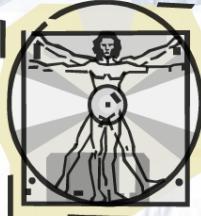


The screenshot shows the Eclipse IDE interface with several windows open:

- Java - CursorableLinkedList.java - Eclipse SDK**: The main editor window displaying Java code for `CursorableLinkedList.java`. The code is color-coded to show coverage status: green for covered lines and yellow for uncovered lines.
- JUnit**: A view showing test results: "Finished after 34,896 seconds", "Runs: 13009/13009", "Errors: 0", and "Failures: 0".
- Coverage**: A view showing detailed coverage statistics for various packages and files. The table includes columns for Element, Coverage, Covered Lines, and Total Lines.

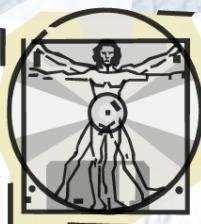
Element	Coverage	Covered Lines	Total Lines
java - commons-collections	79,5 %	10927	13738
org.apache.commons.collections	74,1 %	3842	5183
ArrayList.java	86,5 %	32	37
BagUtils.java	86,7 %	13	15
BeanMap.java	72,4 %	155	214
BinaryHeap.java	87,6 %	127	145
BoundedFifoBuffer.java	93,2 %	82	88
BufferOverflowException.java	55,6 %	5	9
BufferUnderflowException.java	88,9 %	8	9
BufferUtils.java	30,8 %	4	13
ClosureUtils.java	93,9 %	31	33
CollectionUtils.java	92,4 %	293	317
ComparatorUtils.java	8,6 %	3	35
CursorableLinkedList.java	85,4 %	444	520

Emma (command line) and EclEmma



# Emma

- Open-source tool
- Supports class, method, block, and line coverage
- Eclipse plug-in EclEmma also available
  - <http://www.eclemma.org>
- Uses byte-code instrumentation
  - Classes can be instrumented in advance, or during class loading
- Tool keeps a metadata file to associate byte-code with source code



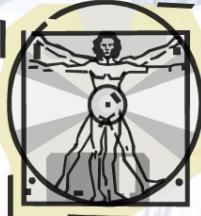
# CodeCover

- CodeCover is an open-source code coverage tool for Java under Eclipse
- It can be used inside Eclipse
- There are many other code coverage tool out there, but this is one of the lightest and powerful ones
- See the list @ <http://java-source.net/open-source/codecoverage>

<http://www.codecover.org>

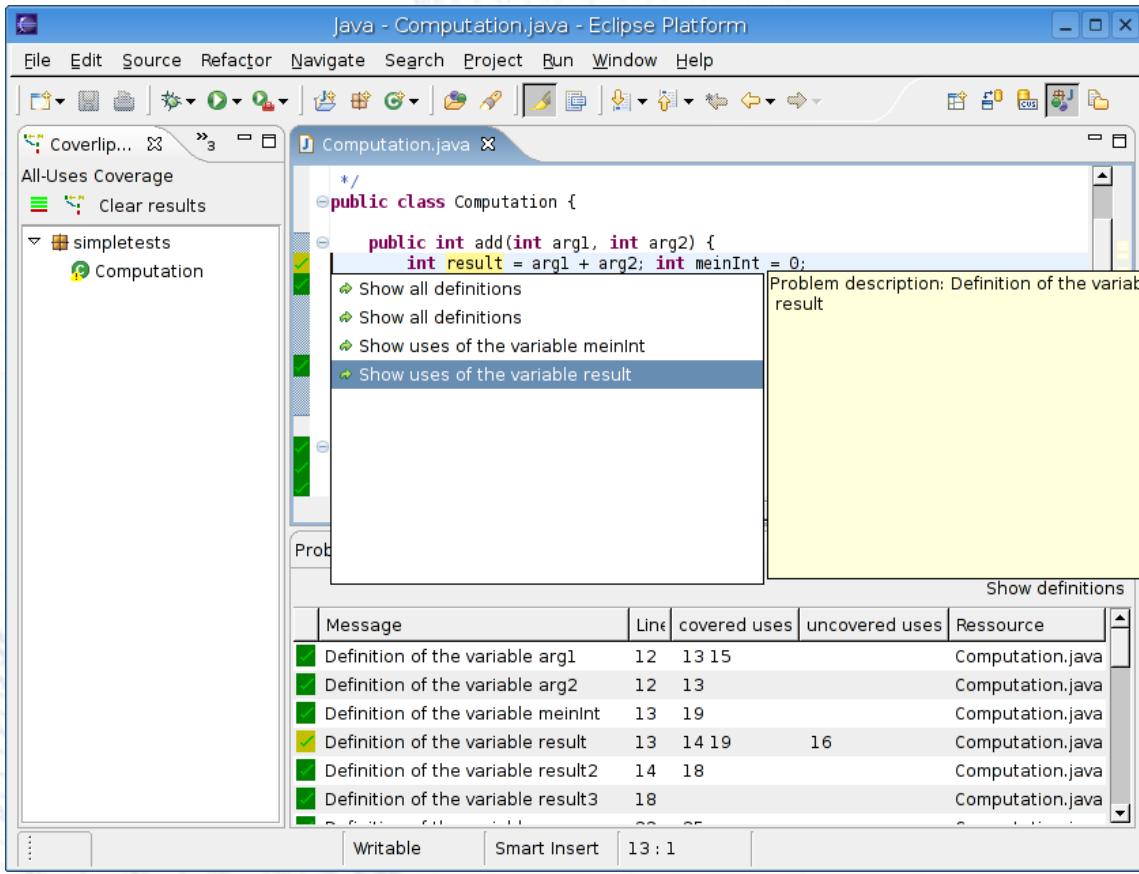


CodeCover



# Code Coverage Tools - CoverLipse

CoverLipse (supports data-flow based testing)



# Code Coverage Tools - Clover

**Clover Coverage Report**

[Dashboard](#) [Coverage Reports](#) [Coverage \(Aggregate\)](#) [Test Code \(Aggregate\)](#) [Test Results](#)

**Application Packages**

com.jcatalog.wiki	(0%)
com.jcatalog.wiki.block	(0%)
org.weeem.blog	(70.6%)
org.weeem.content	(87.6%)
org.weeem.controllers	(11.4%)
org.weeem.css	(71.4%)
org.weeem.event	(-)
org.weeem.export	(68.5%)
org.weeem.files	(29.5%)
org.weeem.html	(87%)
org.weeem.jobs	(8.3%)
org.weeem.js	(69.2%)
org.weeem.security	(88.3%)

**Classes Tests Results**

Class	Tests	Coverage
AccessDeniedException		(0%)
AdminTagLib		(20%)
Blog		(66.7%)
BlogEntry		(73.7%)
CacheService		(18%)
CacheTagLib		(11.6%)
Comment		(80%)
ConfluenceSpaceImporter		(79.3%)
Content		(94%)
ContentController		(7.1%)
ContentDirectory		(32.9%)
ContentFile		(24.8%)
ContentRepositoryService		(48.3%)
ContentVersion		(100%)
ContentVersionService		(0%)
DefaultSpaceExporter		(0%)
DefaultSpaceImporter		(69.3%)
DiffUtils		(0%)
DomainConverter		(0%)
EditorController		(9.7%)
EditorFieldTagLib		(27.4%)
EditorService		(89.8%)
EditorsBuilder		(100%)
EventsService		(100%)
ExternalLink		(54.5%)
ExternalLinksJob		(8.3%)
Folder		(100%)
HTMLContent		(82%)
ImportException		(0%)
ImportExportConverter		(52.9%)
ImportExportService		(61.9%)
JavaScript		(69.2%)
ParagraphBlock		(0%)
PortalController		(54.5%)
RelatedContent		(25%)
RepositoryController		(13.6%)
SAXConfluenceParser		(97.8%)
SecurityPermissionsBuilder		(92.3%)
SecurityPolicyBuilder		(100%)
SimpleSpaceExporter		(0%)

**Clover Coverage Report - Coverage timestamp: Mon Feb 15 2010 15:29:05 EST**

[Overview](#) [Package](#) [File](#)  
[FRAMES](#) [NO FRAMES](#) [SHOW HELP](#)

**Coverage** 61 classes, 1,919 / 4,554 elements  
42.1%

**Test Results** 58 / 58 tests 8.86 secs  
100%

**Class Coverage Distribution**

**Class Complexity**

**Top 20 Project Risks**

- DiffUtils
- WidgetTagLib
- ParagraphBlock
- ContentController
- SynchronizationController
- RepositoryController
- DefaultSpaceExporter
- VersionController
- CacheTagLib
- ContentRepositoryService
- ContentDirectory
- WikiItemRenderController
- WeeemTagLib
- EditorController
- DefaultSpaceImporter
- SpaceController
- ContentFile
- DomainConverter
- CacheService
- WeeemDialect

**Coverage Tree Map**

**Least Tested Methods**

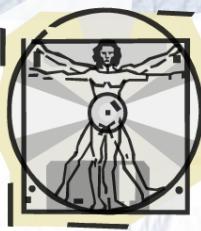
1. 0% ContentController.field show() (42)
2. 0% WeeemTagLib.field menu() (49)
3. 0% ContentRepositoryService.createContentFile(def) : def (31)
4. 0% RepositoryController.field searchRequest() (31)
5. 0% WidgetTagLib.field widget() (29)
6. 0% DefaultSpaceExporter.execute(Space) : File (7)
7. 0% ContentRepositoryService.updateNode(Content,def) : def (25)
8. 0% RepositoryController.field beforeInterceptor() (19)
9. 0% DiffUtils.diffString(def,def) : def (15)
10. 0% SynchronizationController.field linkContentFile() (13)
11. 0% ContentDirectory.deleteContent() : Boolean (17)
12. 0% SynchronizationController.field deleteContent() (13)
13. 0% DiffUtils.diff(def,def) : def (11)
14. 0% ContentRepositoryService.synchronizeSpace(def) : def (13)
15. 0% WeeemTagLib.field eachSibling() (17)
16. 0% RepositoryController.field uploadFile() (15)
17. 0% CacheTagLib.field cache() (13)
18. 0% CacheService.getOrPut(def,boolean,def,def) : def (13)
19. 0% WeeemTagLib.field humanDate() (11)
20. 0% WeeemTagLib.field breadcrumb() (11)

**Most Complex Packages**

1. 52.1% org.weeem.services (560)
2. 11.4% org.weeem.controllers (424)
3. 25.7% org.weeem.tags (342)
4. 68.5% org.weeem.export (195)
5. 87.6% org.weeem.content (101)

**Most Complex Classes**

1. 48.3% ContentRepositoryService (444)
2. 29.5% WeeemTagLib (251)
3. 13.6% RepositoryController (226)
4. 94% Content (58)
5. 79.7% SimpleSpaceImporter (54)



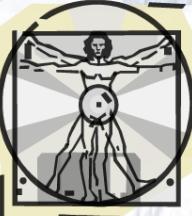
# Comparison of Coverage Tools

*2011 International Conference on Telecommunication Technology and Applications  
Proc .of CSIT vol.5 (2011) © (2011) IACSIT Press, Singapore*

## An Evaluation of Test Coverage Tools in Software Testing

Muhammad Shahid<sup>1</sup>, Suhaimi Ibrahim

Advanced Informatics School (AIS), Universiti Teknologi Malaysia  
International Campus, Jalan Semarak, Kuala Lumpur, Malaysia  
[smuhammad4@live.utm.my](mailto:smuhammad4@live.utm.my), [suhaimiibrahim@utm.my](mailto:suhaimiibrahim@utm.my)



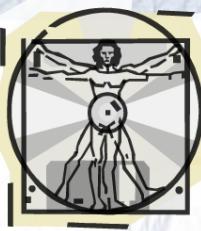
# Comparison of Coverage Tools

Table 1 Tools With Supported Language

Tools	Java	C/C++	Other
<i>JavaCodeCoverage</i>	×		
<i>JFeature</i>	×		
<i>JCover</i>	×		
<i>Cobertura</i>	×		
<i>Emma</i>	×		
<i>Clover</i>	×		.Net
<i>Quilt</i>	×		
<i>Code Cover</i>	×		COBOL
<i>Jester</i>	×		
<i>GroboCodeCoverage</i>	×		
<i>Hansel</i>	×		
<i>Gretel</i>	×		
<i>BullseyeCoverage</i>		×	
<i>NCover</i>			.Net
<i>Testwell CTC++</i>		×	
<i>TestCocoon</i>		×	C#
<i>eXVantage</i>	×	×	
<i>OCCF</i>	×	×	×
<i>JAZZ</i>	×		

Table 3 Coverage Measurement Level

Tools	Statement/ Block	Branch / Decision	Method/ Function	Class	Requirement
<i>JavaCodeCoverage</i>	×	×	×		
<i>JFeature</i>			×		×
<i>JCover</i>	×	×	×	×	
<i>Cobertura</i>	×	×			
<i>Emma</i>	×		×	×	
<i>Clover</i>	×	×	×	×	
<i>Quilt</i>	×	×			
<i>Code Cover</i>	×	×			
<i>Jester</i>					
<i>GroboCodeCoverage</i>				×	
<i>Hansel</i>			×		
<i>Gretel</i>	×				
<i>BullseyeCoverage</i>		×	×		
<i>NCover</i>			×	×	
<i>Testwell CTC++</i>		×			
<i>TestCocoon</i>					
<i>eXVantage</i>	×	×	×		
<i>OCCF</i>	×	×			
<i>JAZZ</i>		×			



# Resources

- Code Coverage and Test Tools

[www.opensourcetesting.org](http://www.opensourcetesting.org)

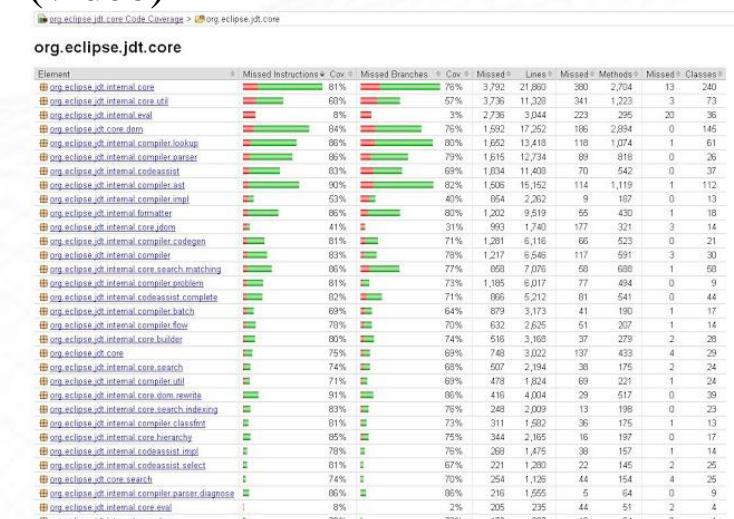
- Code Coverage of Eclipse Code-base

<http://relengofthererds.blogspot.com/2011/03/sdk-code-coverage-with-jacoco.html>

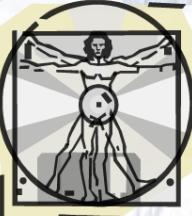
- Demo of test coverage for Python code

[www.youtube.com/watch?v=jGJa\\_2UyHrY](http://www.youtube.com/watch?v=jGJa_2UyHrY) (video)

etc.



Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed Lines	Cov.	Missed Methods	Cov.	Missed Classes	
org.eclipse.jdt.internal.core	81%	19%	76%	76%	3,792	21,860	380	2,704	13	240
org.eclipse.jdt.internal.core.util	69%	30%	57%	57%	3,736	11,029	341	1,223	3	73
org.eclipse.jdt.internal.eval	8%	92%	3%	3%	2,736	3,044	223	295	20	36
org.eclipse.jdt.core.admin	94%	6%	76%	76%	1,592	17,262	166	2,954	0	145
org.eclipse.jdt.internal.compiler.lookup	86%	14%	80%	80%	1,652	13,418	118	1,074	1	61
org.eclipse.jdt.internal.compiler.parser	86%	14%	79%	79%	1,615	12,734	89	818	0	26
org.eclipse.jdt.internal.compiler.classassist	0%	100%	69%	69%	1,034	11,808	70	542	0	37
org.eclipse.jdt.internal.compiler.ast	90%	10%	82%	82%	1,508	15,152	114	1,119	1	119
org.eclipse.jdt.internal.compiler.impl	59%	41%	40%	40%	1,202	9,919	55	430	1	18
org.eclipse.jdt.internal.compiler.formatter	86%	14%	82%	82%	993	7,170	9	107	0	13
org.eclipse.jdt.internal.core.jclm	41%	59%	31%	31%	993	7,170	177	321	3	14
org.eclipse.jdt.internal.compiler.codgen	0%	100%	71%	71%	5,116	16,523	0	21	0	21
org.eclipse.jdt.internal.compiler.codgen\$1	93%	7%	78%	78%	2,127	5,546	117	507	3	30
org.eclipse.jdt.internal.core.search.matching	0%	100%	77%	77%	669	7,076	99	688	1	69
org.eclipse.jdt.internal.compiler.problem	81%	19%	73%	73%	1,185	5,017	77	484	0	9
org.eclipse.jdt.internal.compiler.complete	0%	100%	71%	71%	669	5,212	61	541	0	44
org.eclipse.jdt.internal.compiler.batch	69%	31%	64%	64%	879	3,173	41	180	1	17
org.eclipse.jdt.internal.compiler.flow	70%	30%	70%	70%	708	632	51	207	1	14
org.eclipse.jdt.internal.core.builder	0%	100%	74%	74%	516	3,168	37	279	2	20
org.eclipse.jdt.internal.core.codetransform	76%	24%	69%	69%	749	3,023	137	403	4	39
org.eclipse.jdt.internal.core.astebh	74%	26%	68%	68%	607	2,194	20	176	2	24
org.eclipse.jdt.internal.compiler.util	71%	29%	69%	69%	479	1,804	69	221	1	24
org.eclipse.jdt.internal.core.com.rewrite	91%	9%	66%	66%	416	4,004	29	517	0	39
org.eclipse.jdt.internal.core.search.indexing	83%	17%	76%	76%	249	2,009	13	198	0	23
org.eclipse.jdt.internal.compiler.classfield	91%	9%	73%	73%	311	1,692	36	175	1	13
org.eclipse.jdt.internal.core.hierarchy	85%	15%	75%	75%	344	2,165	16	197	0	17
org.eclipse.jdt.internal.codestransform.impl	79%	21%	76%	76%	269	1,475	38	157	1	14
org.eclipse.jdt.internal.codestransform.select	0%	100%	67%	67%	221	1,200	22	146	2	26
org.eclipse.jdt.core.search	74%	26%	70%	70%	254	1,106	44	154	4	26
org.eclipse.jdt.internal.compiler.parser.diagnose	0%	100%	66%	66%	216	1,665	5	64	0	9
org.eclipse.jdt.internal.core.eval	1%	99%	2%	2%	205	295	44	51	2	4
org.eclipse.jdt.internal.core.com	70%	30%	79%	79%	173	807	16	34	0	1



# Conclusion

- Coverage is a measure of white-box testing effort to detect potential faults
- Typical coverage targets: statements, branches, conditions, decisions, data defined and used
  - 100% statement coverage means testing for every fault that can be revealed by execution of a line of code
  - 100% branch coverage means testing for every fault that can be revealed by testing each branch in the program
  - 100% coverage means that you tested for every possible fault, which is obviously impossible for larger projects