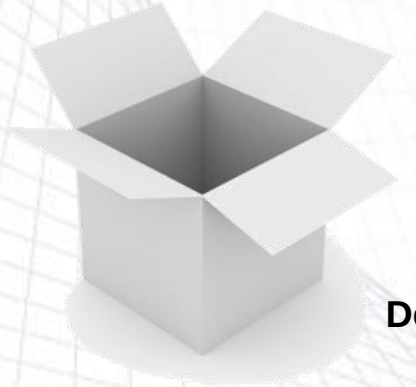
SCHULICH School of Engineering

UNIVERSITY OF CALGARY

SENG 637
Dependability and Reliability of Software Systems



Chapter 5A: White-box Testing
Control-flow Coverage

Department of Electrical & Software Engineering, University of Calgary

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

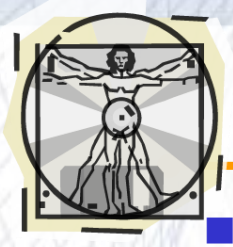
http://people.ucalgary.ca/~far

Contents

- Preliminary: Control Flow Graph (CFG)
- Control flow
  - Statement/Node/Line Coverage
  - Decision/Edge/Branch Coverage
  - Condition Coverage
  - Path Coverage
  - Cyclomatic complexity
- Data flow
- Coverage tools

Our focus in this week

Next week

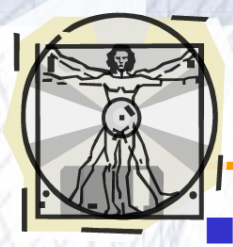


Review: Black- vs. White-box

- Black-box testing
  - Testing, either functional or non-functional, without reference to the internal structure of the component or system
    - Synonyms: specification-based testing
- White-box testing
  - Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system
    - Synonyms: structural, glass box, open box testing

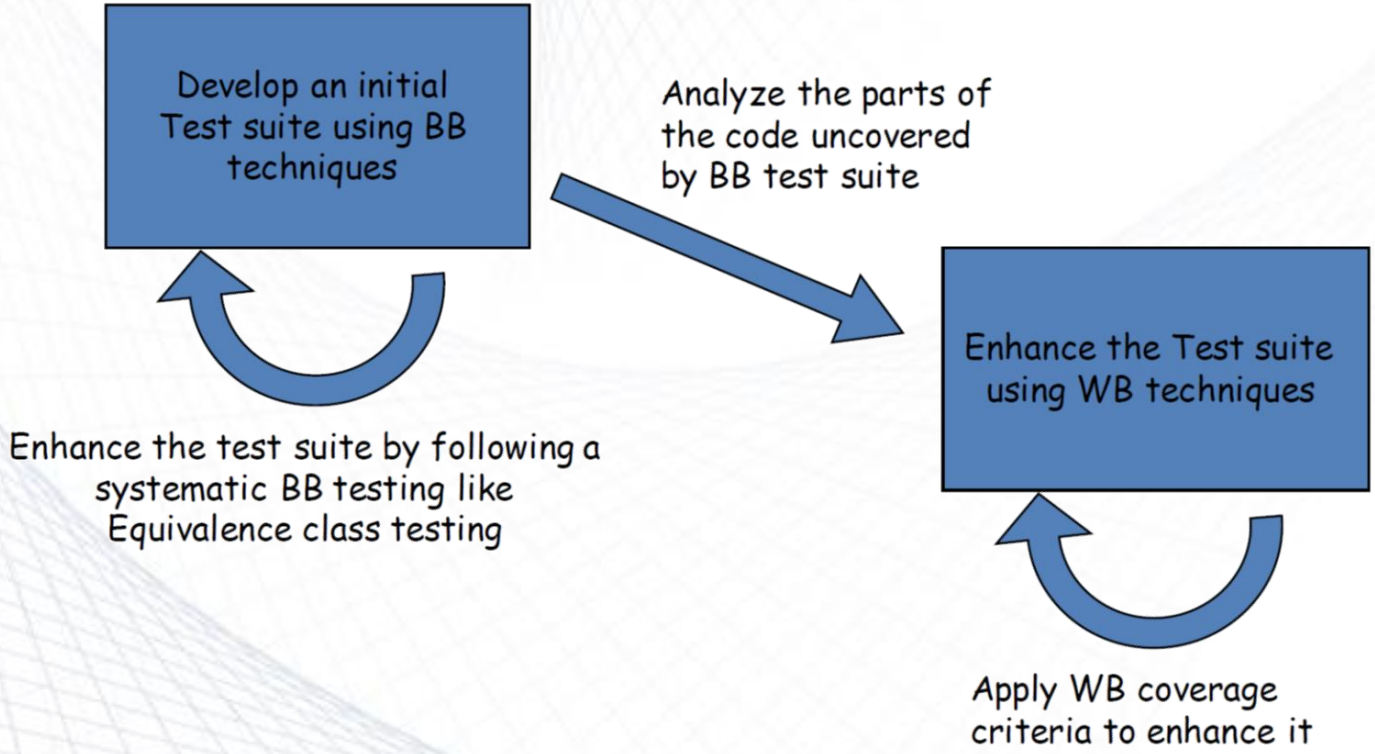
Review: White-box

- Also known as structural testing, glass box or open box testing
- A software testing technique in which explicit knowledge of the internal working of SUT is used to select tests, execute tests and collect test data
- Unlike black box testing that is using the program specification to examine outputs, white box testing is based on specific knowledge of the source code to define the test cases and to examine outputs



Review: Black- vs. White-box

- How do black- and white-box testing relate to one another in real-world projects?



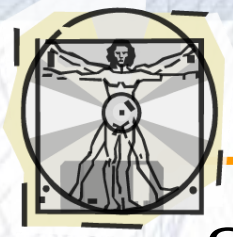
SCHULICH School of Engineering

UNIVERSITY OF CALGARY

Section 1

Preliminary:

Control Flow Graph (CFG)



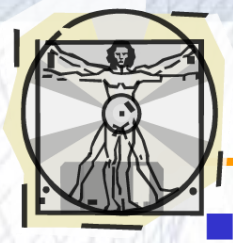
How to Represent Program Structure?

Software structure can have 3 attributes:

- **Control-flow structure:** Sequence of execution of instructions of the program (execution of code lines)
- **Data flow:** Keeping track of data as it is created or handled by the program
- **Data structure:** The organization of data itself independent of the program

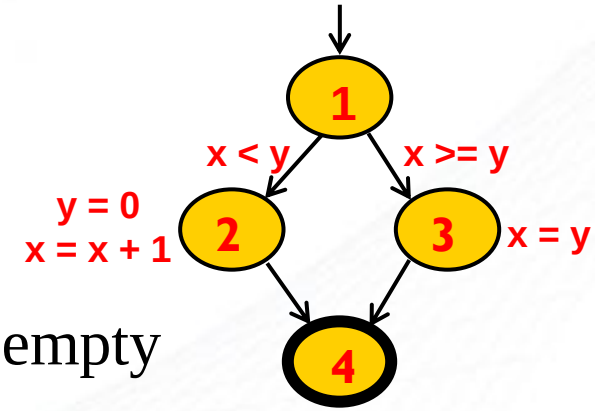
Dynamic

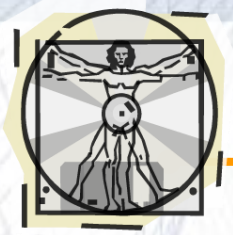
Static



Preliminary: Graph

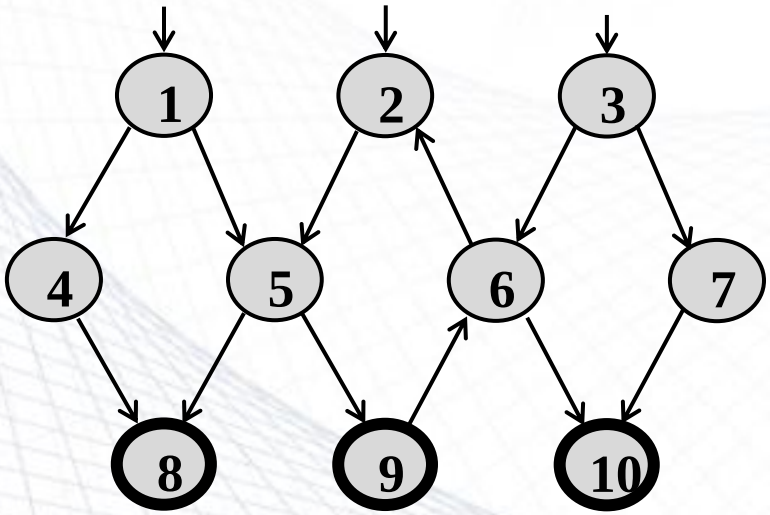
- Graphs are the most commonly used structure for white-box testing
- A graph is
  - A set N of nodes, N is not empty
  - A set N_0 of initial nodes, N_0 is not empty
  - A set N_f of final nodes, N_f is not empty
  - A set E of edges, each edge from one node to another
    - (n_i , n_j), i is predecessor, j is successor
- We usually intend to "cover" the graph equivalent to a program block through testing





Preliminary: Path in a Graph

- Path: A sequence of nodes – $[n_1, n_2, \ldots, n_M]$
  - Each pair of nodes is an edge
- Length: The number of edges
  - A single node is a path of length 0
- Subpath: A subsequence of nodes in *p* is a subpath of *p*

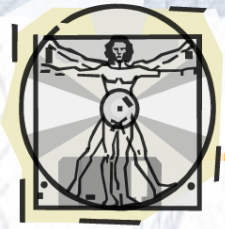


A Few Paths

[1, 4, 8]

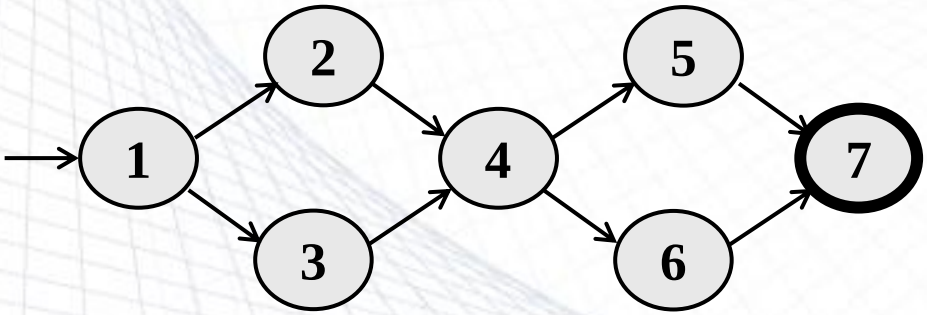
[2, 5, 9, 6, 2]

[3, 7, 10]



Preliminary: Test Path

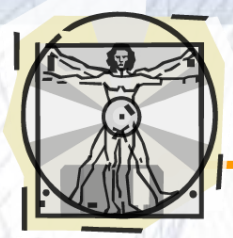
- Test Path: A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests



Double-diamond graph

Four test paths

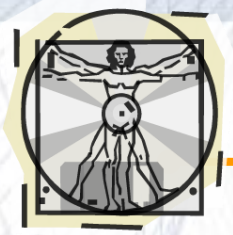
[1, 2, 4, 5, 7]
[1, 2, 4, 6, 7]
[1, 3, 4, 5, 7]
[1, 3, 4, 6, 7]



Testing & Covering Graphs

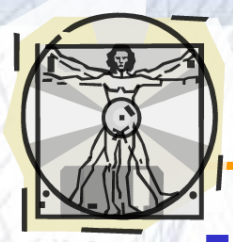
- We use graphs in testing so as to:
  - Develop a model of the software as a graph
  - Require tests to visit or tour specific sets of nodes and edges

- **Structural Coverage Criteria:** Defined on a graph just in terms of nodes and edges
- **Data Flow Coverage Criteria:** Requires a graph to be annotated with references to variables



Testing & Covering Graphs

- Graph: Usually the control flow graph (CFG)
- Node (line or statement) coverage: Execute every program statement
- Edge (decision or branch) coverage: Execute every branch
- Loops: Looping structures such as for-loops, while-loops, etc.
- Data flow coverage: Augment the CFG with extra information
  - branch predicates
  - **`defs`** are statements that assign values to variables
  - **`uses`** are statements that use variables



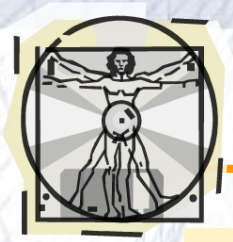
Control Flow Graph (CFG) /1

- Control flow structure is usually modeled by a directed graph (di-graph)

  CFG = {N, A}

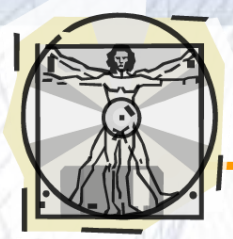
- Each node *n* in the set of nodes (N) corresponds to a program statement.
- Each directed arc (or directed edge) *a* in the set of arcs (A) indicates flow of control from one statement of program to another.
  - **Procedure nodes:** nodes with out-degree 1.
  - **Predicate nodes:** nodes with out-degree other than 1 and 0.
  - **Start node:** nodes with in-degree 0.
  - **Terminal (end) nodes:** nodes with out-degree 0.

This is where a branch Is created



CFG - Example

```
if a then
   if b then X
   Y
   while e do U
else
   if c then
      repeat V until d
   endif
endif
```

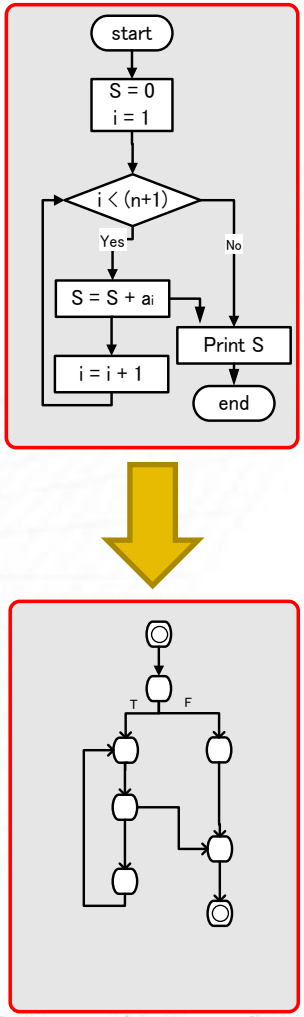


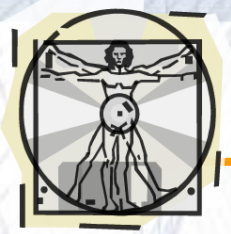
CFG vs. Flowchart

- Are CFG and flowchart the same?
- Can I use flowchart for testing a program?

Yes, if you want … but note that …

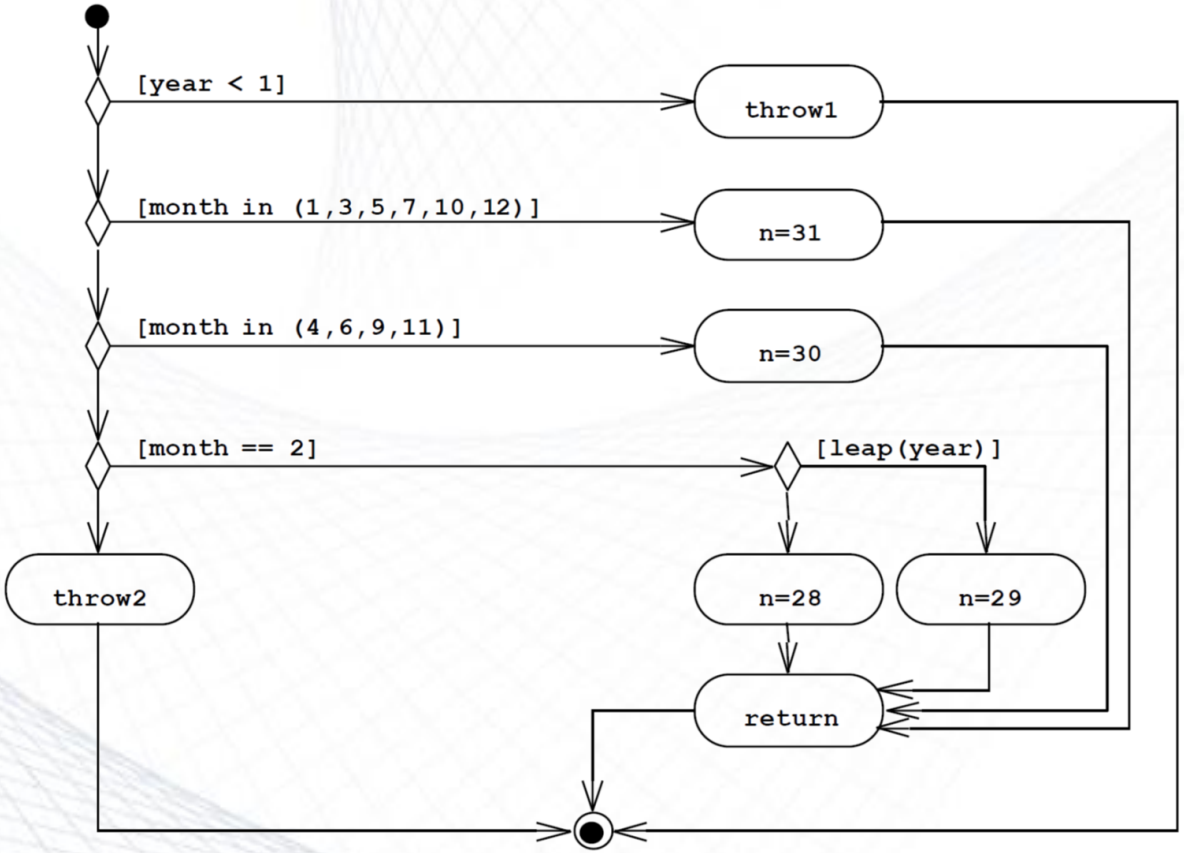
1. In flow graphs, we don't show the details of what is in a process block whereas in flow charts every part of the process block is defined and drawn
2. In flow charts, different types of nodes are represented by different symbols in flow charts, but we do not use different symbols in control flow graphs
3. The flow graph focuses on control flow of the program whereas the flowchart focuses on process steps
4. Flow charts can be used for test case design; they can be used for testing, e.g. path coverage; but we can simply use CFG instead

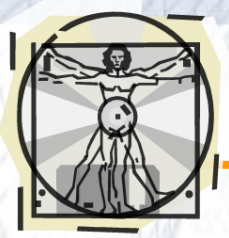




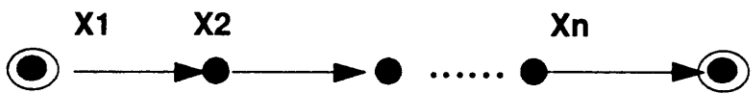
CFG vs. Activity Diagram

- UML Activity Diagram: can be used to model/draw CFGs



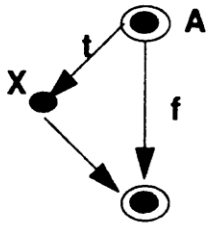


Common CFG Program Models



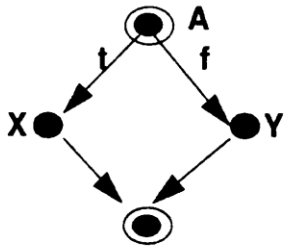
P_n or $P_n(X1,X2,...,Xn)$

X1;X2;...;Xn



D_0 or $D_0(A,X)$

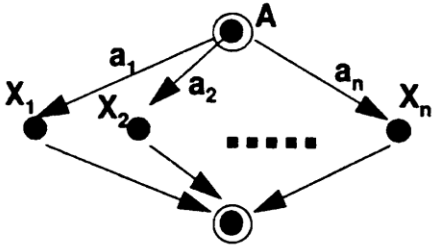
If A then X



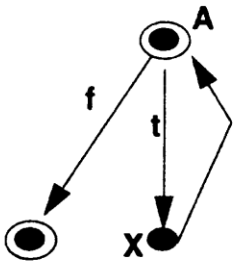
D_1 or $D_1(A,X,Y)$

If A then X
else Y

C_n or $C_n(A,X_1,...,X_n)$

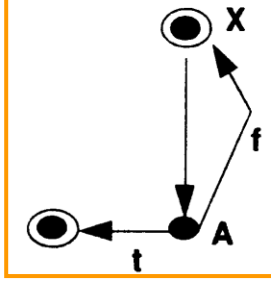


case A of
a1 : X1
a2 : X2
....
an : Xn



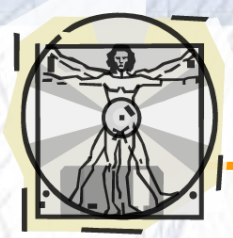
D_2 or $D_2(A,X)$

while A do X



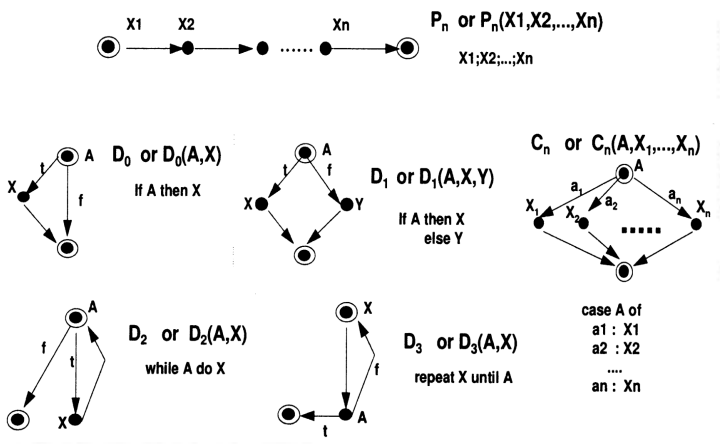
D_3 or $D_3(A,X)$

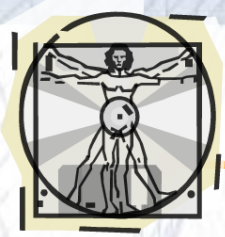
repeat X until A



Prime Flow Graphs

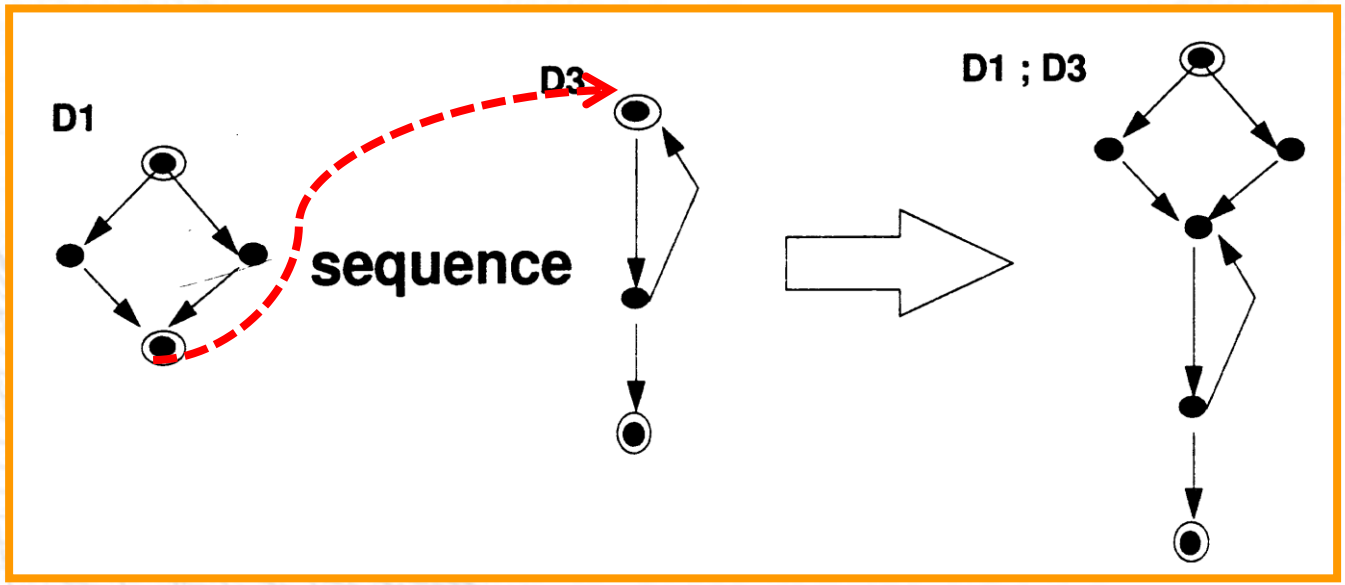
- Prime flow graphs are flow graphs that cannot be decomposed non-trivially by sequencing and nesting
- **Examples (according to [FeP97]):**
  - P_n (sequence of n statements)
  - D_0 (if-condition)
  - D_1 (if-then-else-branching)
  - D_2 (while-loop)
  - D_3 (repeat-loop)
  - C_n (case)

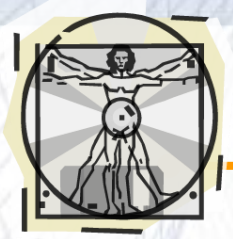




Sequencing & Nesting /1

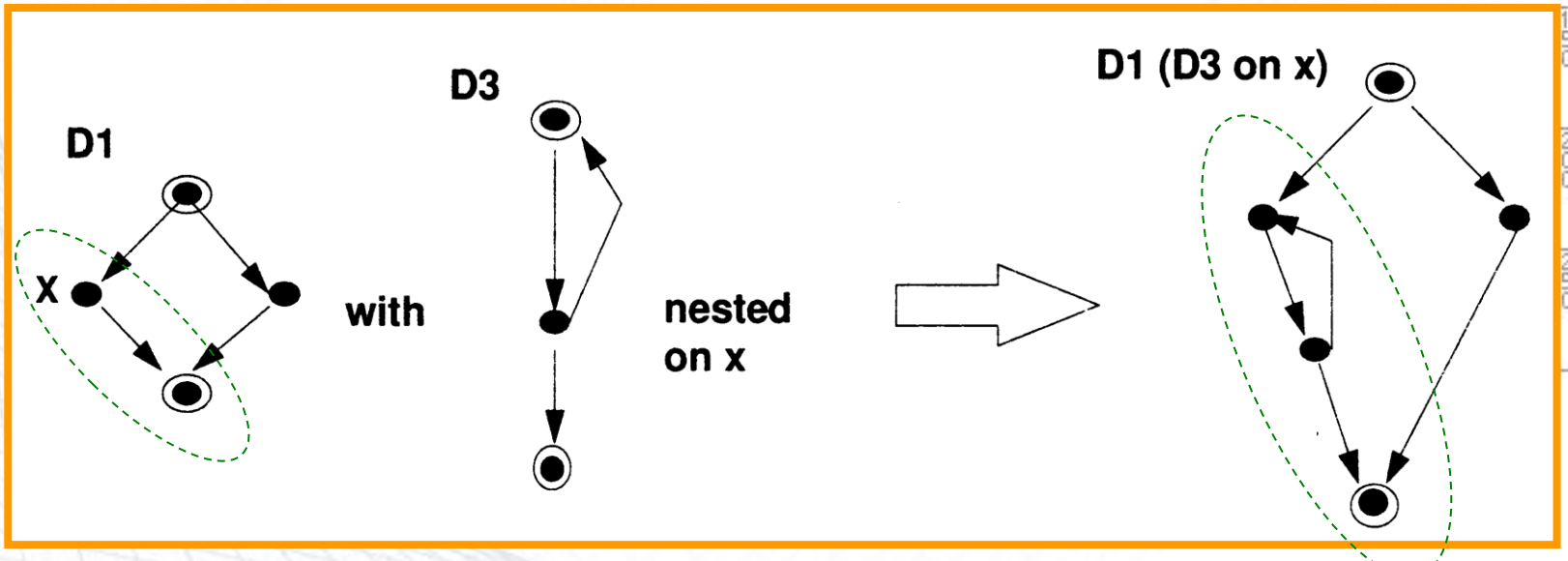
- Let **F1** and **F2** be two flowgraphs. Then, the sequence of **F1** and **F2**, (shown by **F1; F2**) is a flowgraph formed by merging the terminal node of **F1** with the start node of **F2**.

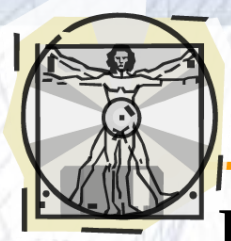




Sequencing & Nesting /2

- Let **F1** and **F2** be two flowgraphs. Then, the nesting of **F2** onto **F1** at *x*, shown by **F1(F2)** is a flowgraph formed from **F1** by replacing the arc from *x* with the whole of **F2**.





Example: Triangle Program

Problem Statement

The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The integers a, b, and c must satisfy the following conditions:

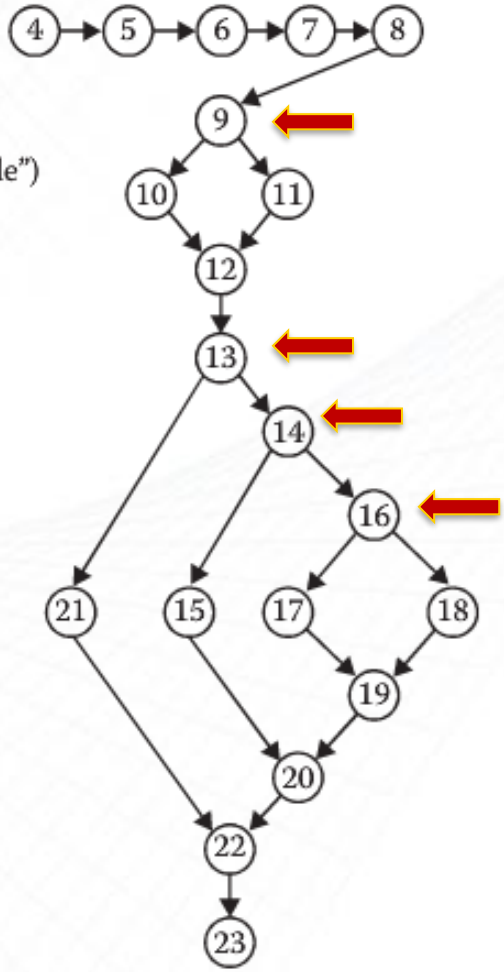
C1) $a < b + c$ C2) $b < a + c$ C3) $c < a + b$

The output of the program is the type of triangle determined by the three sides:

- If all three sides are equal, the program output is Equilateral
- If exactly one pair of sides is equal, the program output is Isosceles
- If no pair of sides is equal, the program output is Scalene
- If any of conditions is not met, the program output is NotATriangle

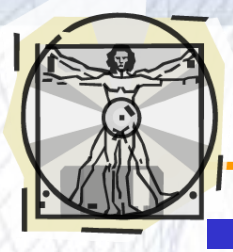
Example: CFG for Triangle

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10    Then IsATriangle = True
11    Else IsATriangle = False
12 EndIf
13 If IsATriangle
14    Then If (a = b) AND (b = c)
15            Then Output ("Equilateral")
16            Else If (a≠b) AND (a≠c) AND (b≠c)
17                    Then Output ("Scalene")
18                    Else Output ("Isosceles")
19                 EndIf
20            EndIf
21    Else  Output("Nota a Triangle")
22 EndIf
23 End triangle2
```



Constructing CFG

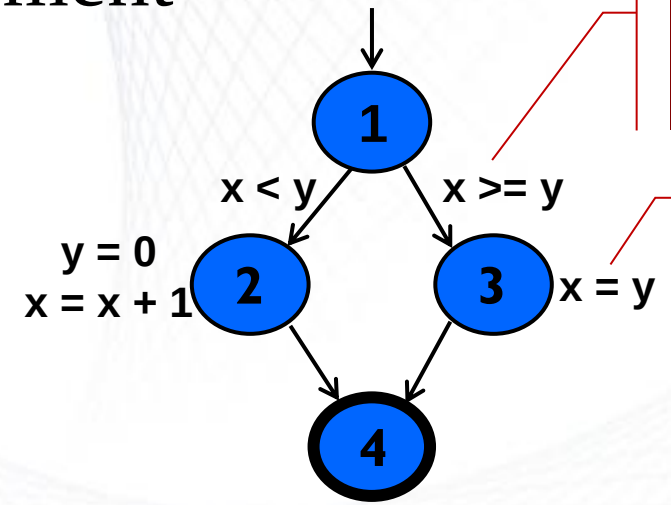
- Relatively easy to map a program to its equivalent CFG
- As mentioned earlier
  - Nodes (three kinds):
    - **Statement nodes:** represent single entry single exit sequence of statements
    - **Predicate nodes:** represent conditions for branching
    - **Auxiliary nodes:** for completing the graph
  - Edges: represent possible flow of control



Constructing CFG

- The **`if`** statement

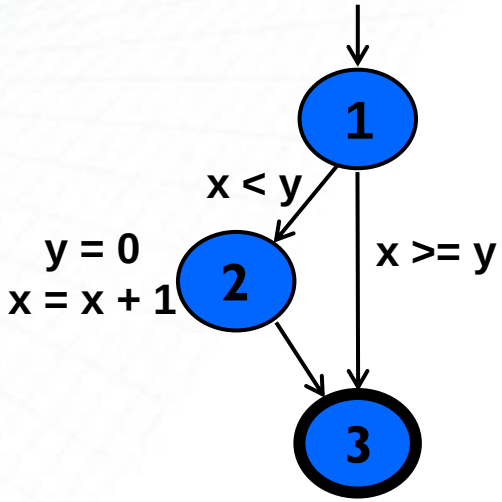
```
if (x < y)
{
    y = 0;
    x = x + 1;
}
else
{
    x = y;
}
```

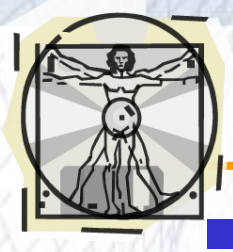


We need to identify the exact conditions for predicate nodes

We do not need to identify the job done by statement nodes

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

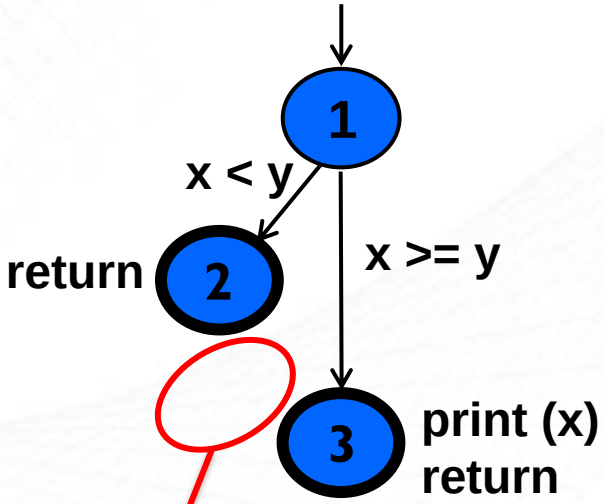




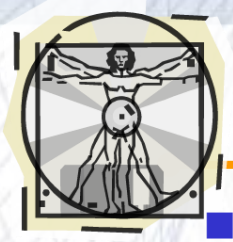
Constructing CFG

- The **`if`** return statement

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



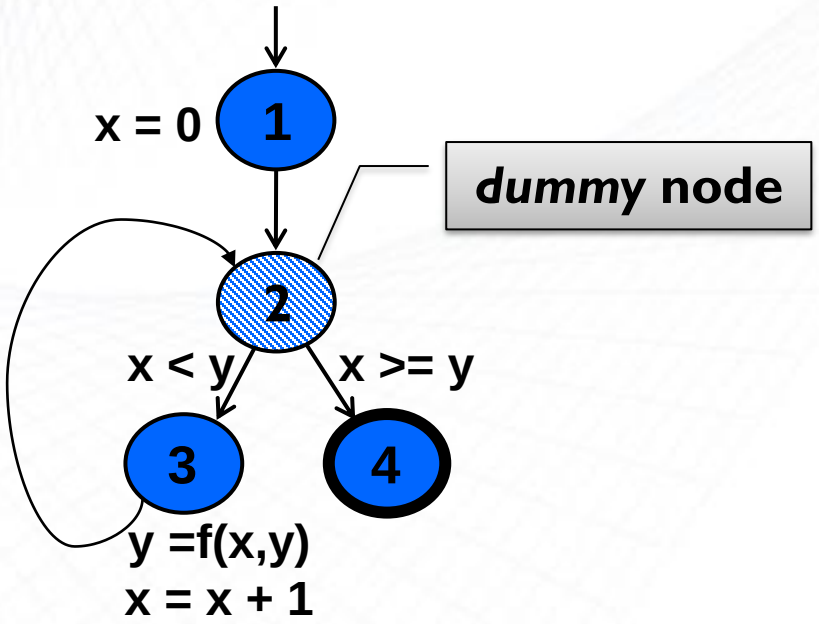
No edge from node 2 to 3.
The return nodes must be distinct.

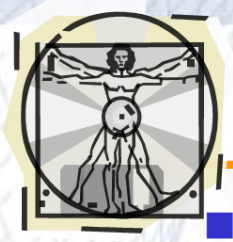


Constructing CFG

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

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

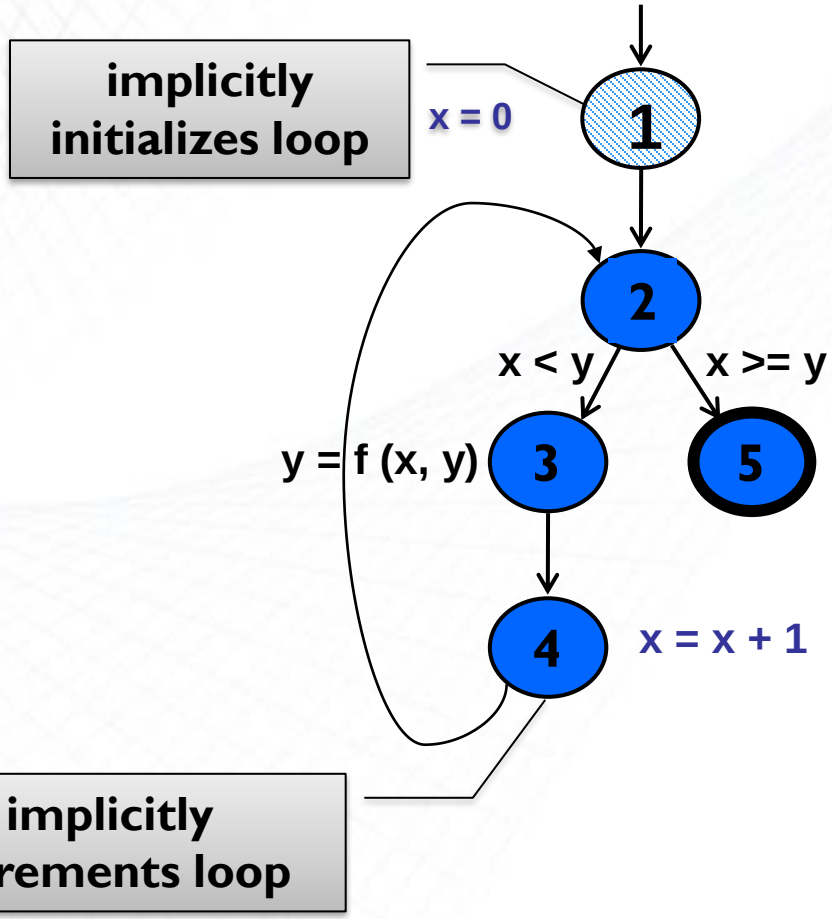


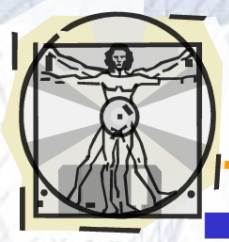


Constructing CFG

- The **`for`** Loop

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

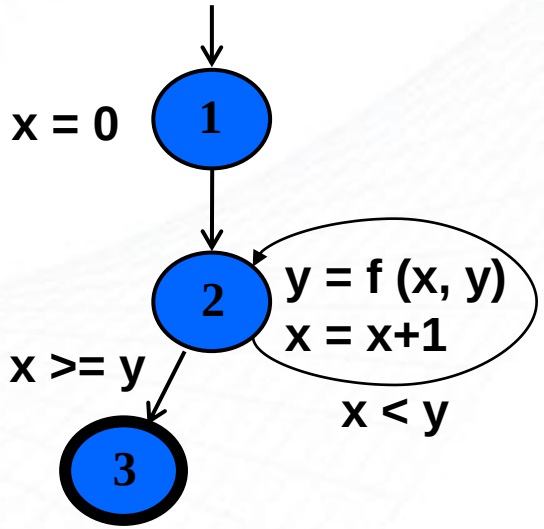


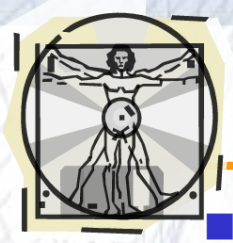


Constructing CFG

- The **do** Loop

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

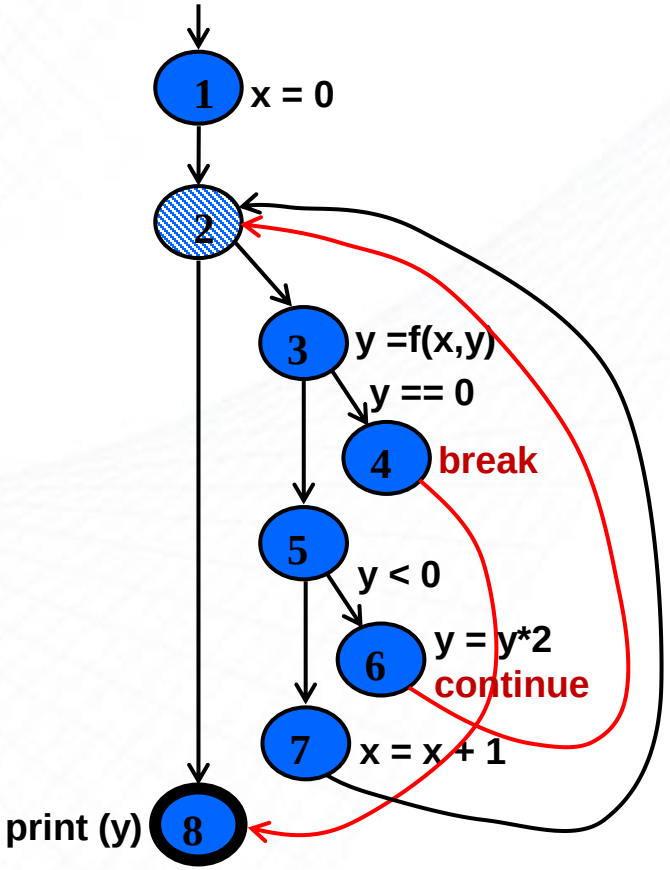


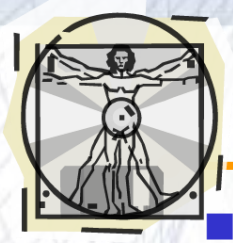


Constructing CFG

- The **do** Loop with **break** and **continue**

```
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);
```

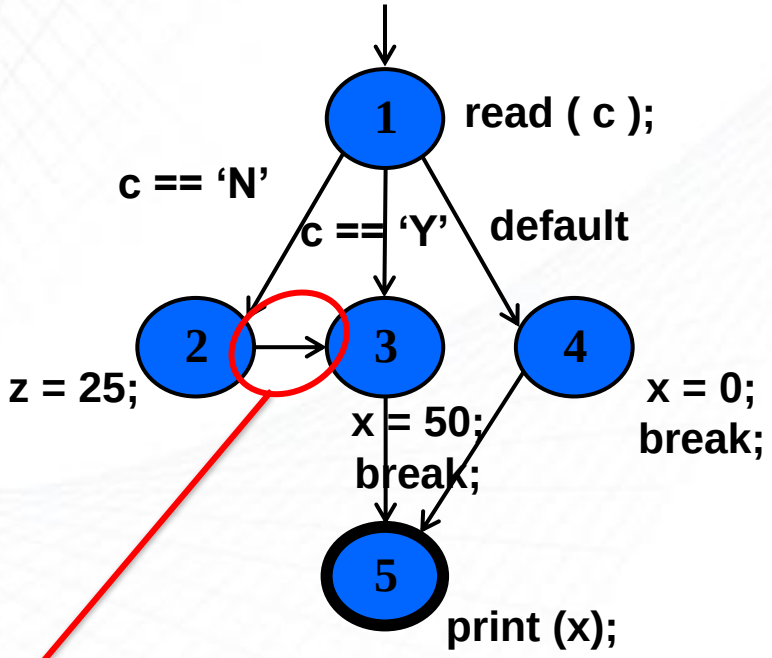




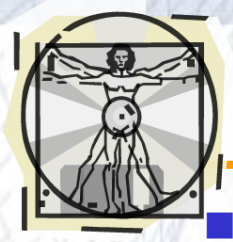
Constructing CFG

- The **switch** (**case**) structure

```
xread (c) ;
switch (c)
{
    case 'N':
        z = 25;
    case 'Y':
        x = 50;
        break;
    default:
        x = 0;
        break;
}
print (x);
```



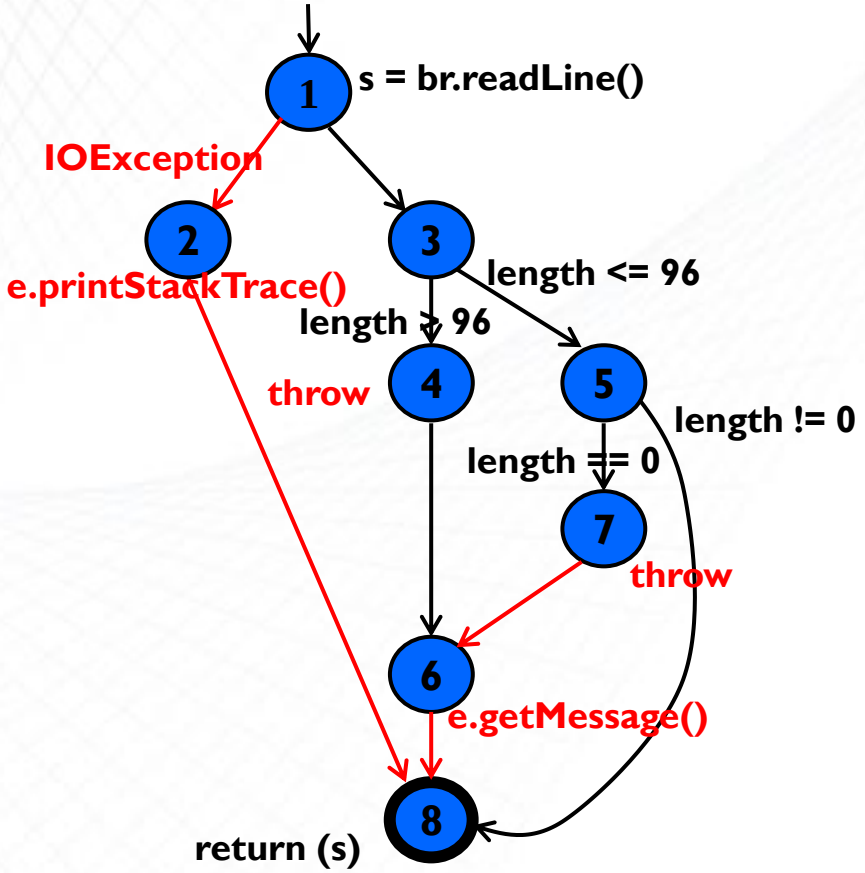
Cases without breaks fall through to the next case

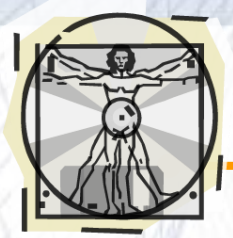


Constructing CFG

- The exception (**`try-catch`**) structure

```
try
{
    s = br.readLine();
    if (s.length() > 96)
        throw new Exception
            ("too long");
    if (s.length() == 0)
        throw new Exception
            ("too short");
} (catch IOException e) {
    e.printStackTrace();
} (catch Exception e) {
    e.getMessage();
}
return (s);
```



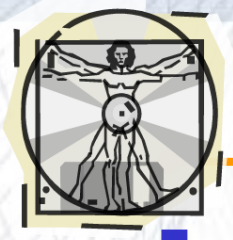


How to Draw CFG?

- Several online tools
- E.g.
  - https://code2flow.com/app

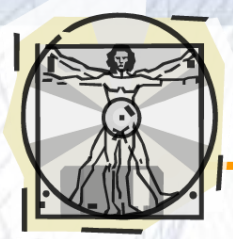
Section 2

Control Flow Coverage



Contrlo Flow Based Testing

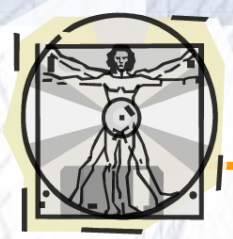
- **Step 1:** From the source code create a control flow graph (CFG)
  - Manually or automated or mentally
- **Step 2:** Design test cases to cover certain elements of CFG
  - Nodes, edges, conditions, paths, etc.
- **Step 3:** Decide upon appropriate coverage metrics to report test results
  - Statement, decision, condition, path coverage metrics
- **Step 4:** Execute tests, collect and report coverage data



Control Flow Criteria

- We can use different control-flow criteria:
  1. Statement/Block/Node/Line coverage
  2. Decision/Edge/Branch coverage
  3. Condition coverage
  4. Path coverage
- We can even analyze/measure complexity of the code and use it to create test cases
  - Cyclomatic complexity

Discussed in detail next…

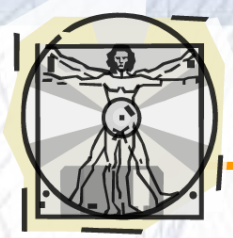


Coverage Metrics

- For each criterion we must define a metrics to measure it
- Test coverage measures the amount of testing performed by a set of test

$$\text{Coverage} = \frac{\text{Number of coverage items exercised}}{\text{Total number of coverage items}} \times 100\%$$

- Wherever we can count things and can tell whether or not each of those things has been tested by some test, then we can measure coverage and is known as test coverage

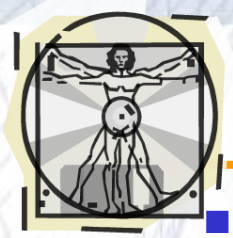


Coverage Metrics Examples

- We can calculate or verify coverage manually or use dedicated tools

$$\text{Statement coverage} = \frac{\text{Number of statements exercised}}{\text{Total number of statements}} \times 100\%$$

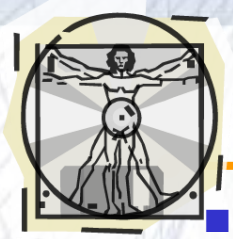
$$\text{Decision coverage} = \frac{\text{Number of decision outcomes exercised}}{\text{Total number of decision outcomes}} \times 100\%$$



1. Statement Coverage

- **Rationale** (why do we need it): Faults cannot be discovered if code lines containing them are not executed
- **Statement coverage criterion:** Equivalent to covering all **nodes** in CFG
- Executing a statement is a weak guarantee of correctness, but rather easy to achieve
- In general, several inputs execute the same statements

- An important question in practice is: How can we minimize (the number of) test cases so we can achieve a given statement coverage percentage?



Statement Coverage

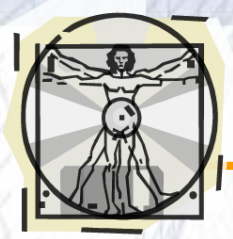
- Statement coverage criteria may lead to incompleteness (i.e. does not guarantee finding faults)
- **Example 1:**

```
public void myFunction(int a, int b){
    if (a < b){
        a = 1;
    }
    a = 1/a;

}
```

What if TC1: (a=0, b=2)?

- TC1: (a = 1, b = 2) will give 100% statement coverage
- Why 100% line coverage may not catch the fault here?

Conclusion: statement coverage does not guarantee bug-free software



Statement Coverage Exercise

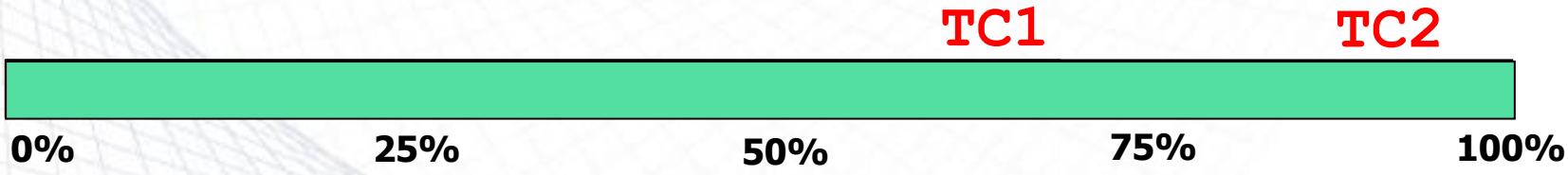
- Percentage of code covered by the tests

```
1: PrintSum (int a, int b){
2:    int result=a+b;
3:    if (result>0)
4:        printcol("red", result);
5:    else if (result<0)
6:  printcol("blue", result);
7: }
```

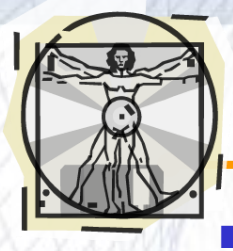
Test Suite

TC1:
a=3, b=9

TC2:
a=-5, b=-8



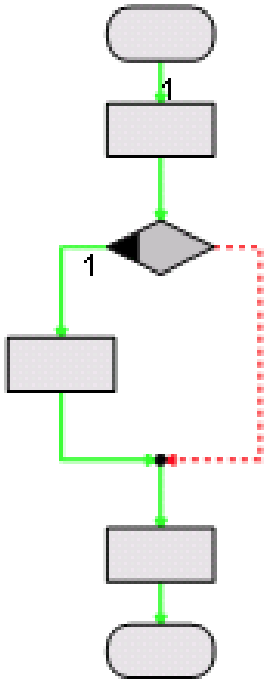
Note: this is usually done automatically by a test coverage tool



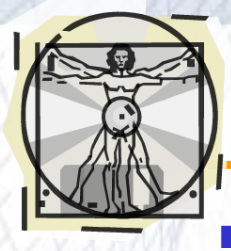
2. Decision (Branch) Coverage

- Branch coverage relates to decisions in a program
- e.g. an IF statement has always two branches
- **Example:** Branch coverage for the example given TC: (a = 1, b = 2)

```
1:  public void myFunction(int a, int b){
        if (a < b){
            a = 1;
        }
2:      a = 1/a;

    }
```

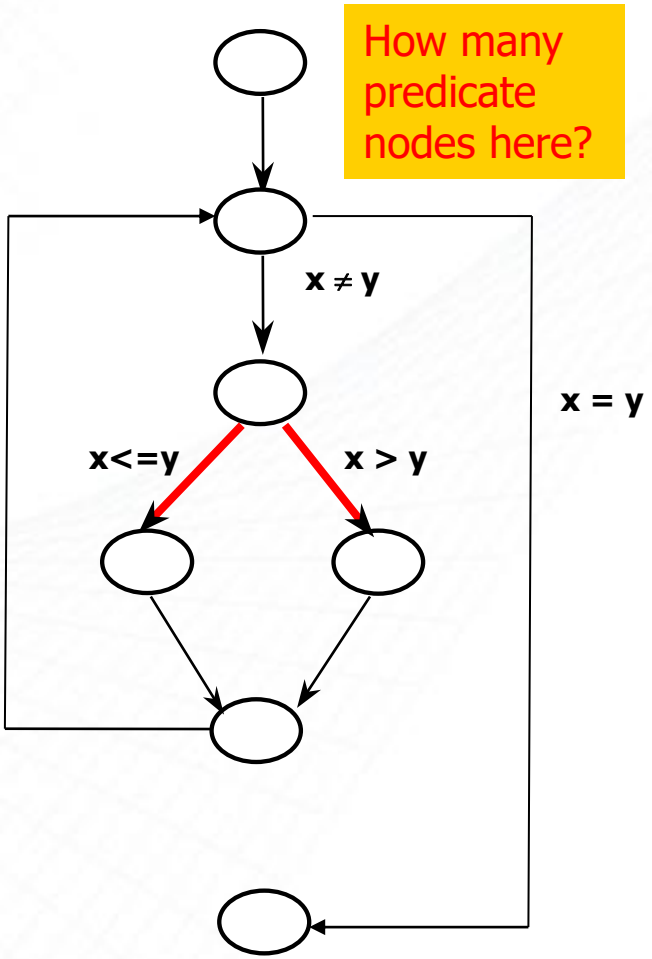


The else-branch with zero executions is in red. Executed branch is in green. Therefore branch coverage for the code excerpt is 50%.

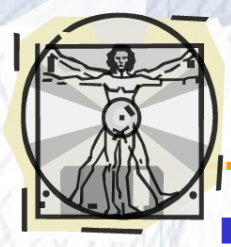


Decision (Branch) Coverage

- Given control flow graph (CFG) of a program
- Select a test set such that, by executing the program for each test case in the set, each edge of CFG's decision (predicate) node(s) is traversed at least once
- We need to exercise <u>all decision</u> that traverse the control flow of the program with True and False values



TC1: (x=5, y=7) enough?
TC2: (x=5, y=3) enough?
TC3: (x=5, y=5)



Edge (Branch) vs. Node (Statement)

- **Example**
- TS1:

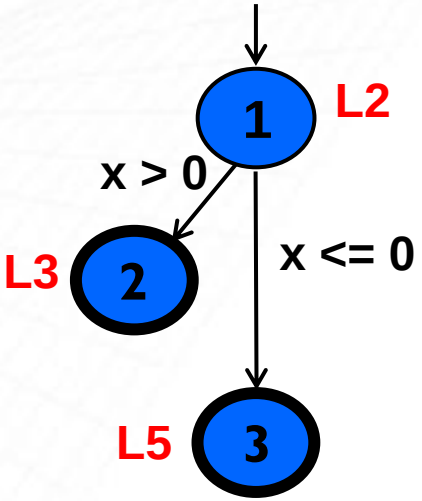
  TC1:<x=1, expected output=1>
  - Achieves 100% line coverage
  - Only 50% decision coverage

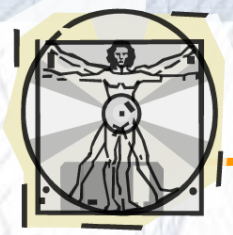
```
1: public static int abs(int x){
2:    if (x>0)
3:        return x;
4:    else
5:        return -x;
6: }
```

- TS2:

  TC1:<x=1, expected output=1>

  TC2:<x=-2, expected output=2>
  - Achieves 100% line coverage
  - Also 100% decision coverage
  - Subsumes statement coverage





Branch Coverage: Example 1

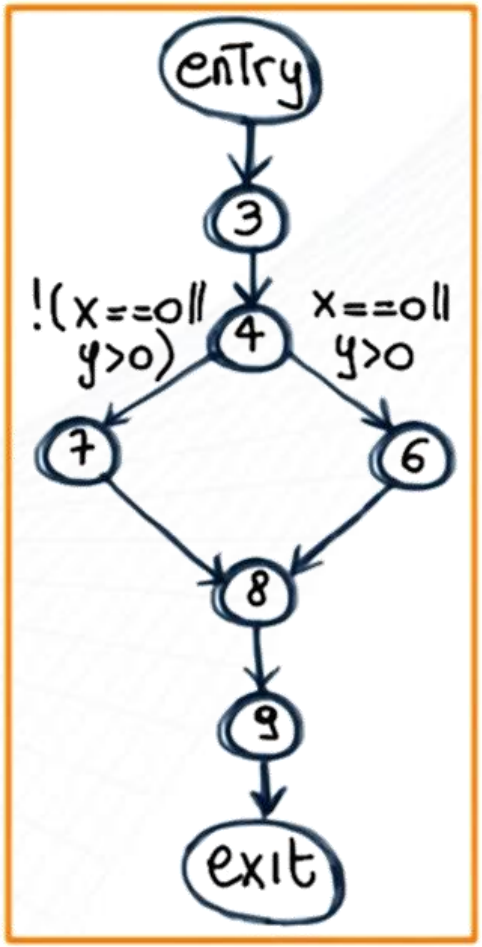
- In this example both statement and **branch coverage** will be 100% but still the test suite cannot catch the possible divide by zero fault (Line 6)

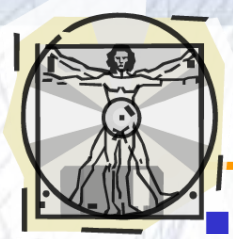
We need a more rigorous metrics to catch the bug

Condition Coverage: make each condition True or False

Tests: (x=5, y=6)
(x=5, y=-5)

```
1. void main(){
2.   float x,y;
3.   read(x);
4.   read(y);
5.   if((x==0)||(y>0))
6.     y=y/x;
7.   else x=y+2;
8.   write(x);
9.   write(y);
10. }
```





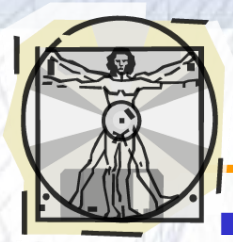
Example 2: Search Program

- Example of a common mistake in a search algorithm
- The range may go out of bound in certain cases

```
public class MyVector {
    Vector<Integer> contents;

    public MyVector(int size) {
        contents = new Vector<Integer>(size);
    }

    public void add(int i) {
        contents.add(new Integer(i));
    }

    public boolean search(int desired_element){

        boolean found = false;
        if (contents.size() != 0) {
            int counter =0;
            while (!found && counter<=contents.size()) {
                if (contents.elementAt(counter).
                        equals(new Integer(desired_element)))
                    found = true;
                counter = counter + 1;
            }
        }

        return found;
    }
}
```

Fault: Should have been <

Note: If the desired element = last element, will we exit the loop before finding it.



Example 2: Test Cases

- We design a test set with two test cases:
  - TC1: A vector with 0 items
  - TC2: A vector with 3 items

```
@Test
public void testEmptyVector() {
    MyVector vector;
    vector = new MyVector(0);
    assertEquals(false, vector.search(10));
}
```

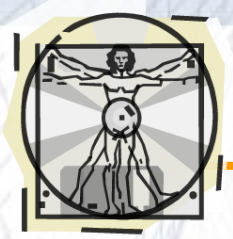
```
@Test
public void testVectorElementInMiddle() {
    MyVector vector;
    vector = new MyVector(3);
    vector.add(50);
    vector.add(100);
    vector.add(250);
    assertEquals(true, vector.search(100));
}
```

- Coverage of above test suite: 100% statement and 100% branch coverage

Name	Statement	Branch
search_for_element_in_arr	100.0 %	100.0 %
MyVector	100.0 %	100.0 %
MyVector	100.0 %	–
add	100.0 %	–
search	100.0 %	100.0 %

The edge coverage criterion is fulfilled but the out of bound fault is NOT discovered by the test set

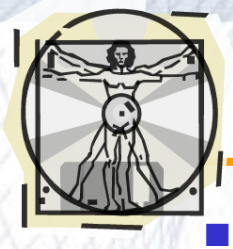
Conclusion: branch coverage does not guarantee bug-free software



3. Condition Coverage

- Condition coverage is a more powerful test coverage criteria
- **Condition Coverage (CC) Criterion**: Design a test set such that each individual condition in the program to be both True and False (regardless of the truth or falsity of predicates combining those conditions) → this may cause branch coverage suffer though (see next slide's example)
- We should usually consider both branch (decision) and condition coverage together
- How? by executing the program for each element in the set, all possible values of the constituents of *compound conditions* (defined below) are exercised at least once

- **Compound conditions**: C1 and C2 or C3 … where Ci's are relational expressions or Boolean variables (atomic conditions)



Condition Coverage - Example

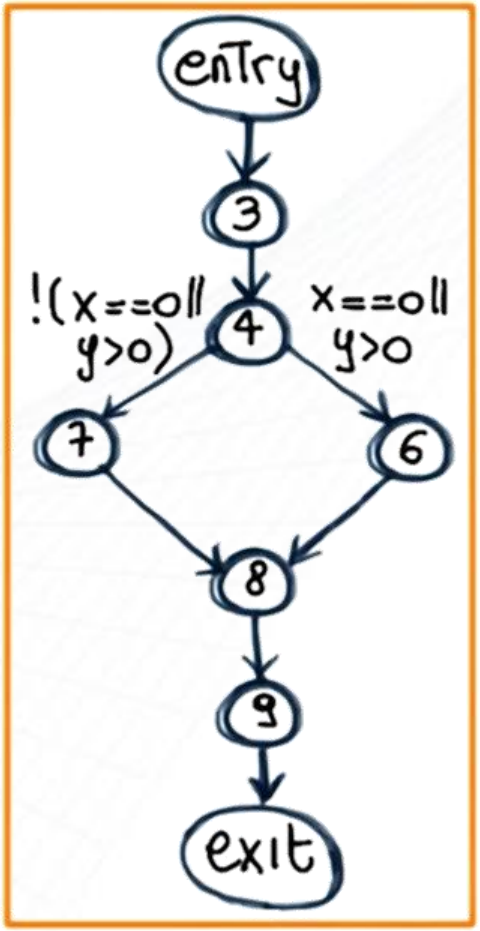
- Same example above, different test suite
- **Condition coverage** will be 100% however, branch (decision) coverage is only 50% (why?)

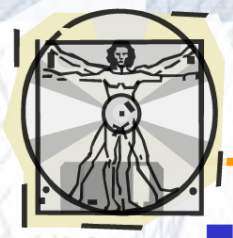
Both cases execute only the right side branch

Therefore:
We need to consider both branch and condition coverage!

Tests: (x=0, y=-5)
(x=5, y=5)

```
1.  void main(){
2.    float x,y;
3.    read(x);
4.    read(y);
5.    if((x==0)||(y>0))
6.      y=y/x;
7.    else x=y+2;
8.    write(x);
9.    write(y);
10. }
```

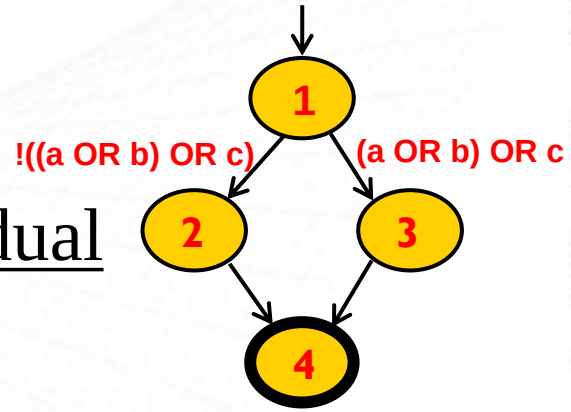




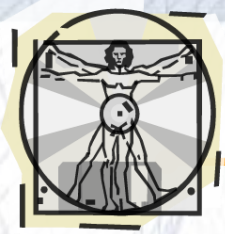
Condition vs. Decision Coverage

- For a compound predicate: **(a∨b)∨c**
- **Decision coverage** test suite T={TC1, TC2}, i.e. make the compound predicate True and False
  - TC1= <a=T, b=F, c=F>
  - TC2= <a=F, b=F, c=F>
- **Condition coverage** test suite T={TC3, TC4}, i.e. make individual conditions True and False
  - TC3= <a=T, b=F, c=T>
  - TC4= <a=F, b=T, c=F>

100% decision coverage



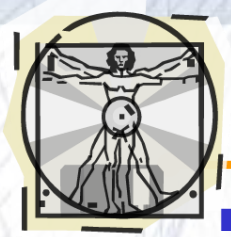
50% condition coverage



Condition-Decision Coverage

- Better choice: Both *conditions* and *decisions* should be covered for (T and F)
- **Multiple C/DC (all combinations)**
- e.g. $(a \vee b) \vee c$
  - {TTT, TTF, TFT, TFF, FTT, FTF, FFT, FFF}
- Not scalable
  - Exponential growth

n	Minimum tests	Time to execute all tests
1	2	2 ms
4	16	16 ms
8	256	256 ms
16	65536	65.5 seconds
32	4294967296	49.5 days



Condition Coverage

- An example Decision: C1 and C2 or C3 … where Ci's are relational expressions or Boolean variables (atomic conditions)
- In our example:

```
public boolean search(int desired_element){

    boolean found = false;
    if (contents.size() != 0) {
        int counter =0;
        while (!found && counter<=contents.size()) {
            if (contents.elementAt(counter).
                    equals(new Integer(desired_element)))
                found = true;
            counter = counter + 1;
        }
    }

    return found;
}
```

- We should exercise each atomic condition to T and F
- Decisions with one condition? 2 tests
- Decisions with more than one condition? Several test

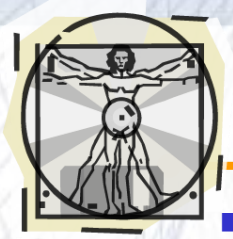
Condition Coverage

- Our test suite:
  - TC1: A vector with 0 items
  - TC2: A vector with 3 items

- How many conditions? Any unreachable one?
- How many condition cases are covered by the above two test cases?

```
public boolean search(int desired_element){

    boolean found = false;
    if (contents.size() != 0) {
        int counter =0;
        while (!found && counter<=contents.size()) {
            if (contents.elementAt(counter).
                    equals(new Integer(desired_element)))
                found = true;
            counter = counter + 1;
        }
    }

    return found;
}
```

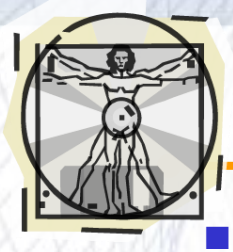
Condition Coverage: Let's make it 100%



- Our test suite:
  - TC1: A vector with 0 items
  - TC2: A vector with 3 items
- What TC(s) to add?

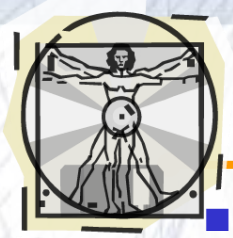
```
public boolean search(int desired_element){

    boolean found = false;
    if (contents.size() != 0) {
        int counter =0;
        while (!found && counter<=contents.size()) {
            if (contents.elementAt(counter).
                    equals(new Integer(desired_element)))
                found = true;
            counter = counter + 1;
        }
    }

    return found;
}
```

We should add a TC to make this False



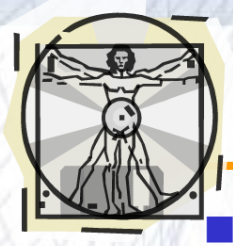
Multiple Condition

- Using condition coverage on some compound condition C implies that each simple condition within C has been evaluated to true and false
- However, it does not imply that all combinations of the values of the individual simple conditions in C have been exercised
- Number of test cases with condition coverage:
  - 2 *n n: number of conditions
- Number of test cases with multiple condition coverage:
  - 2 **n n: number of conditions



Modified Condition-Decision (MC/DC)

- **Modified Condition-Decision (MC/DC) Criterion:** Effectively test "important combinations" of conditions
- Only combinations of values such that every atomic condition C_i toggles (impacts) the overall condition's truth value (True and False), i.e., the outcome of a decision changes as a result of changing each single condition
  - Minimum set size = N+1 (Linear)
  - MC/DC subsumes decision coverage
  - Effective use in industry: The international standard DO-178B for Airborne Systems Certification (since 1992) requires testing the airborne software systems with MC/DC criterion



Exercise - MC/DC

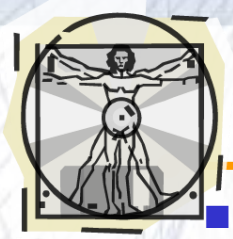
- **Example:** (a∧b∧c), in a *while* loop, let's look at its truth table and derive the test cases…
  - a affects t_1 and t_5
  - b affects t_1 and t_3
  - c affects t_1 and t_2
- Overall 4 test cases

a && b && c

Test case	a	b	c	outcome
1	True	True	True	True
2	True	True	False	False
3	True	False	True	False
4	True	False	False	False
5	False	True	True	False
6	False	True	False	False
7	False	False	True	False
8	False	False	False	False

1	True	True	True	True
5	False	True	True	False
3	True	False	True	False
2	True	True	False	False

Condition criterion: 8 cases 8
Modified criterion: 4 cases



Example - MC/DC

- **Example:** $A \wedge (B \vee C)$, e.g., in a *while* loop, let's look at its truth table and derive the test cases…
  - Condition criterion : 8 cases
  - Modified criterion:
    - Minimal set n+1=4 cases

Take a pair for each constituent:

- A: (1,5), or (2,6), or (3,7)
- B: (2,4)
- C: (3,4)

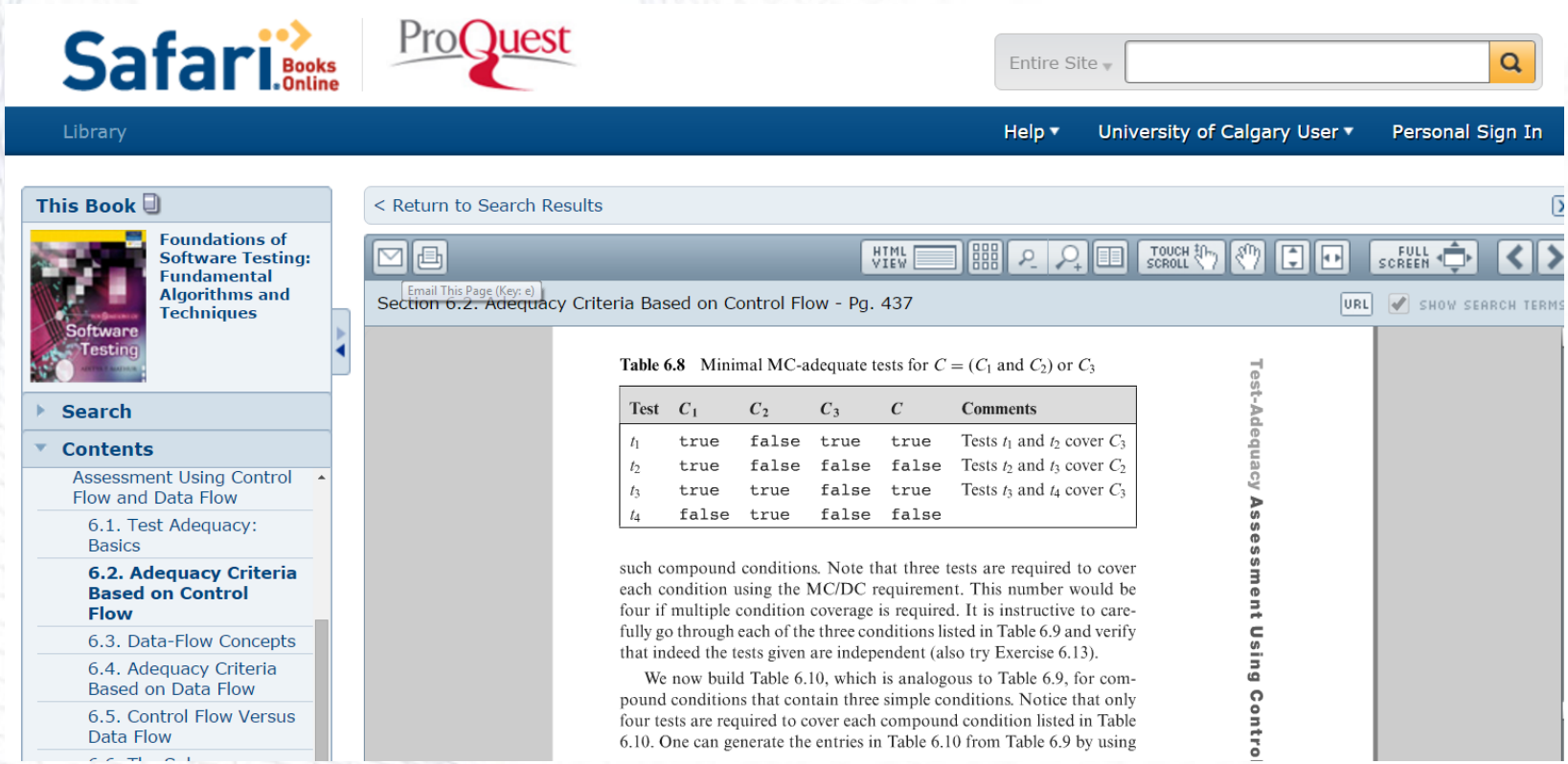
Two minimal sets to cover the MCC:

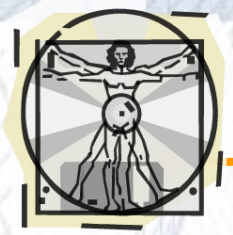
- (2,3,4,6) or (2,3,4,7)

	ABC	Results	Corresponding negate Case
1	TTT	T	A (5)
2	TTF	T	A (6), B (4)
3	TFT	T	A (7), C (4)
4	TFF	F	B (2), C (3)
5	FTT	F	A (1)
6	FTF	F	A (2)
7	FFT	F	A (3)
8	FFF	F	-

Examples with MC/MD

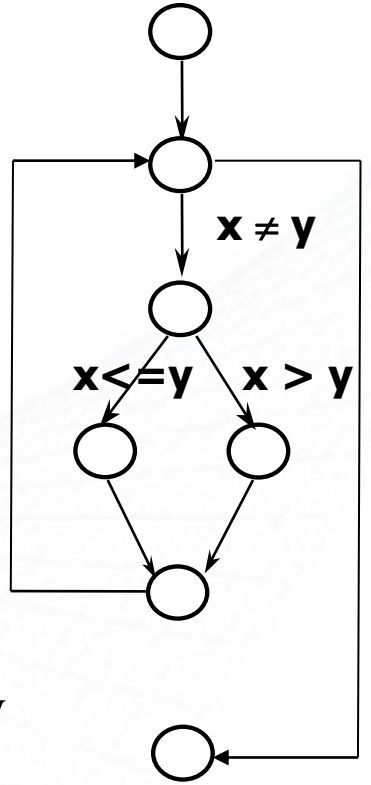
You can find several examples in the reference book …

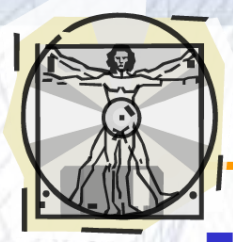




4. Path Coverage

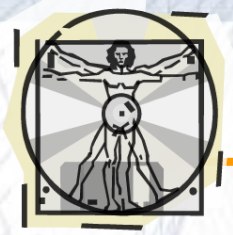
- **Path Coverage Criterion**: Select a test set such that by executing the program for each test case, all paths leading from the initial to the final node of program's control flow graph are traversed
  - In practice, however, the number of paths is too large, if not infinite (e.g., when we have loops)
  - Some paths are infeasible (e.g., not practical given the system's business logic)
  - It may be important to determine "critical paths", leading to more system load, security intrusions, etc.





Path Coverage Metrics

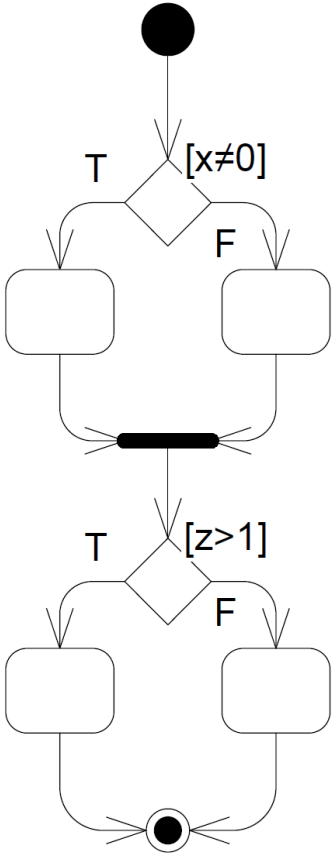
- Path coverage counts the number of full paths from input to output through a program that get executed
- Full path coverage will lead to full branch coverage



Path Coverage - Example

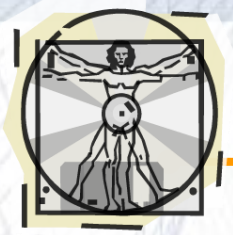
- Let's compare how the following two test sets cover this CFG:

```
if x ≠ 0 then
    y := 5;
else
    z := z - x;
end if;
if z > 1 then
    z := z / x;
else
    z := 0;
end if;
```



T1 (test set) =
{ TC11:<x=0, z =1>,
TC12:<x =1, z=3>
}

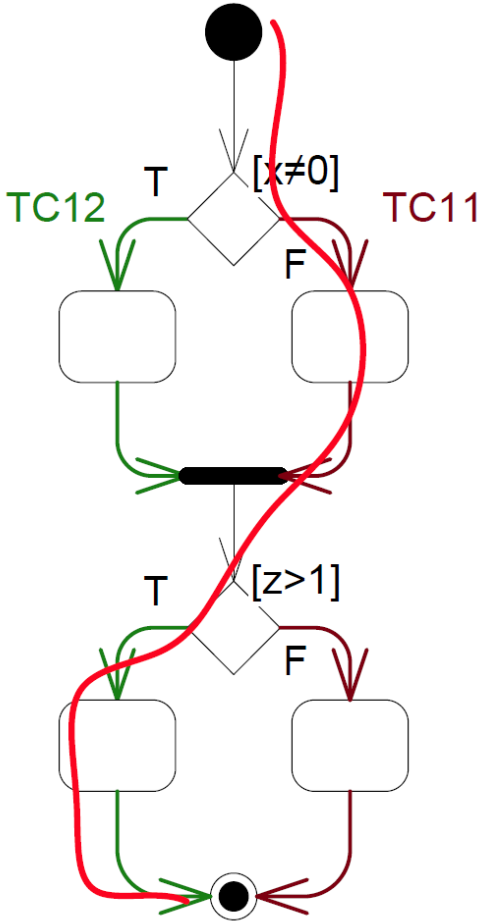
T2 (test set) =
{ TC21:<x=0, z =3>,
TC22:<x =1, z=1>
}



Path Coverage - Example

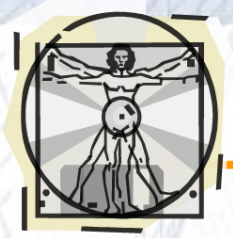
T1's coverage

```
if x ≠ 0 then
    y := 5;
else
    z := z - x;
end if;
if z > 1 then
    z := z / x;
else
    z := 0;
end if;
```



T1 = {TC11:<x=0, z =1>,
TC12:<x =1, z=3>}

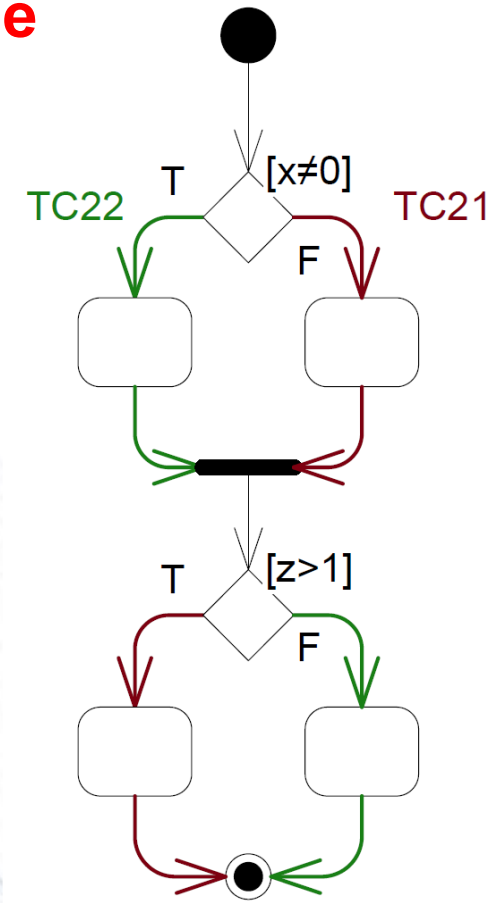
T1 executes all edges but…!
Do you see any testing issue (uncovered paths which can be sources of failure)?
T1 executes all edges and all conditions but does not test risk of division by 0.
(See the red "path")



Path Coverage - Example

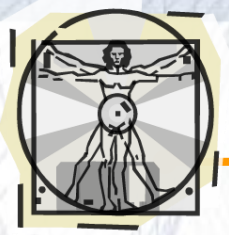
T2's coverage

```
if x ≠ 0 then
    y := 5;
else
    z := z - x;
end if;
if z > 1 then
    z := z / x;
else
    z := 0;
end if;
```

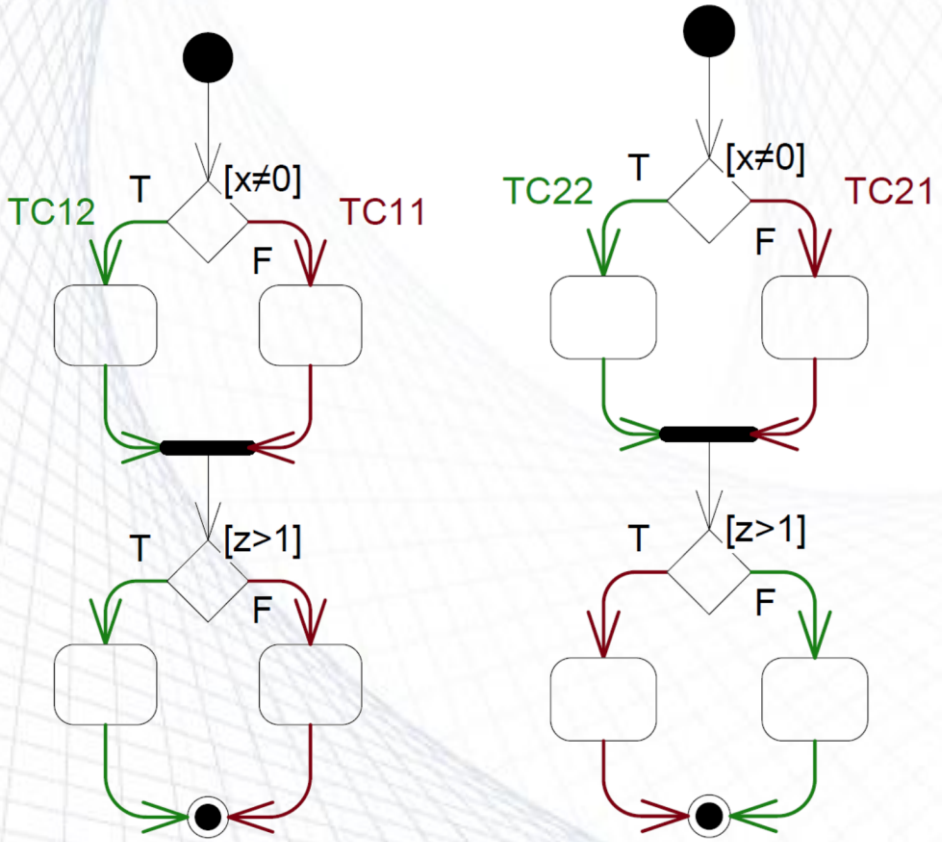


T2 = {TC21:<x=0, z =3>,
TC22:<x =1, z=1>}

T2 would find the problem (triggering division by 0) by exercising the remaining possible flows of control through the program fragment



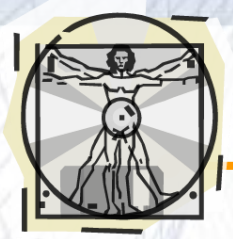
Path Coverage - Example



T1 (test set) = {TC11:<x=0, z =1>, TC12:<x =1, z=3>}
T1 executes all edges but do not show risk of division by 0

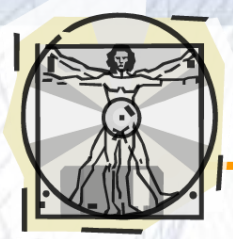
T2 (test set) = {TC21:<x=0, z =3>, TC22:<x =1, z=1>}
T2 would find the problem by exercising the remaining possible flows of control through the program fragment

Observation:
T1 and T2 → all paths covered



Path Coverage - Loops

- In practice, however, the number of paths can be too large, if not infinite (e.g., when we have loops) → Impractical
- A pragmatic heuristic: Look for conditions that execute loops
  - Zero times
  - A maximum number of times
  - An average number of times (statistical criterion)
- For example, in the array search algorithm
  - Skipping the loop (the table is empty)
  - Executing the loop once or twice and then finding the element
  - Searching the entire table without finding the desired element

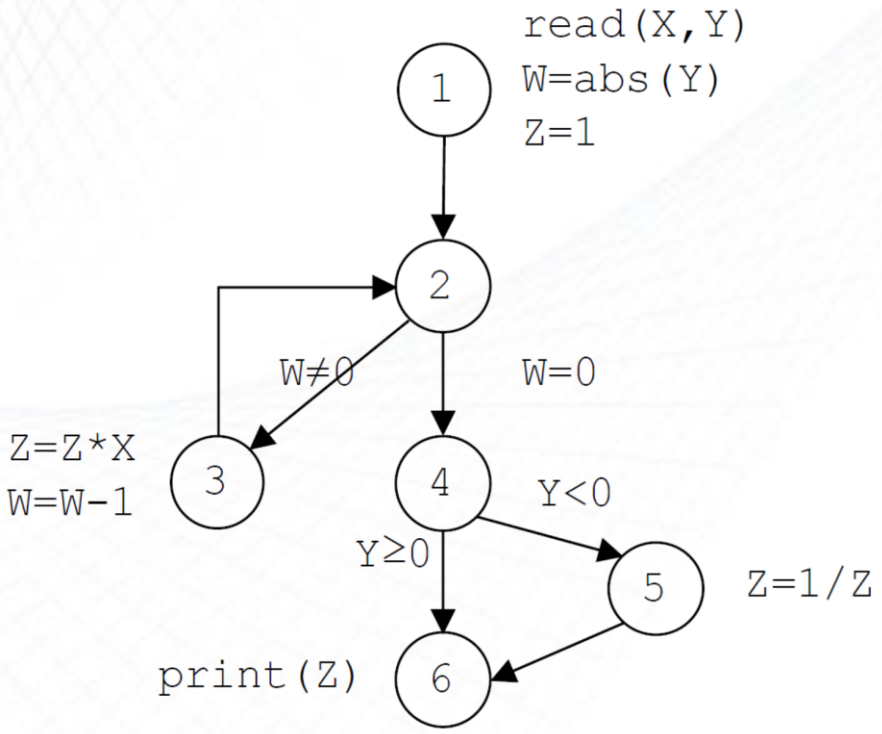


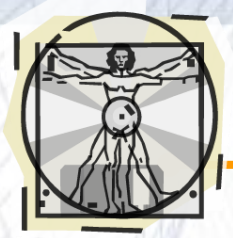
Path Coverage - Loops

- **Example:** Power function

Program computing Z=X^Y

```
BEGIN
  read (int X, int Y) ;
  W = abs(Y) ;
  Z = 1 ;
  WHILE (W <> 0) DO
    Z = Z * X ;
    W = W - 1 ;
  END
  IF (Y < 0) THEN
    Z = 1 / Z ;
  END
  print (Z) ;
END
```





Path Coverage - Loops

- **Example:** Power function - comparison with "all branches" and "all statements"

All statements

– One test case is enough

Y<0 : 1 → 2 → (3 → 2)+ → 4 → 5 → 6

All branches

– Two test cases are enough

Y<0 : 1 → 2 → (3 → 2)+ → 4 → 5 → 6

Y>0 : 1 → 2 → (3 → 2)* → 4 → 6

All paths

– Infeasible path

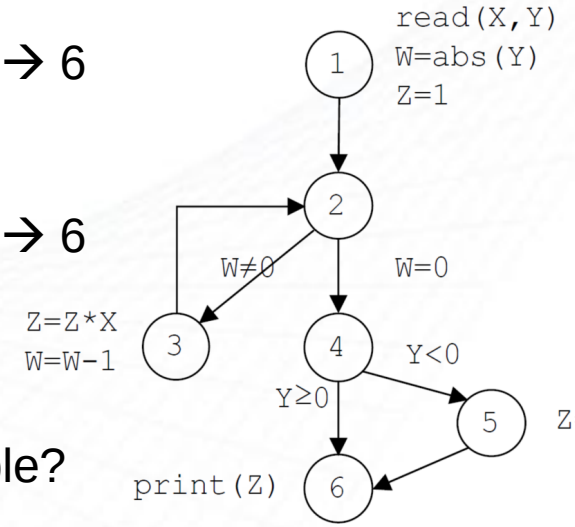
1 → 2 → 4 → 5 → 6, Why infeasible?

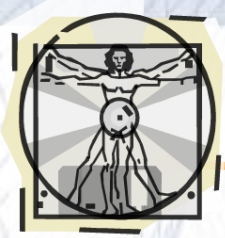
The way Y and W relate

– Potentially large number of paths (depends on Y)

As many ways to iterate

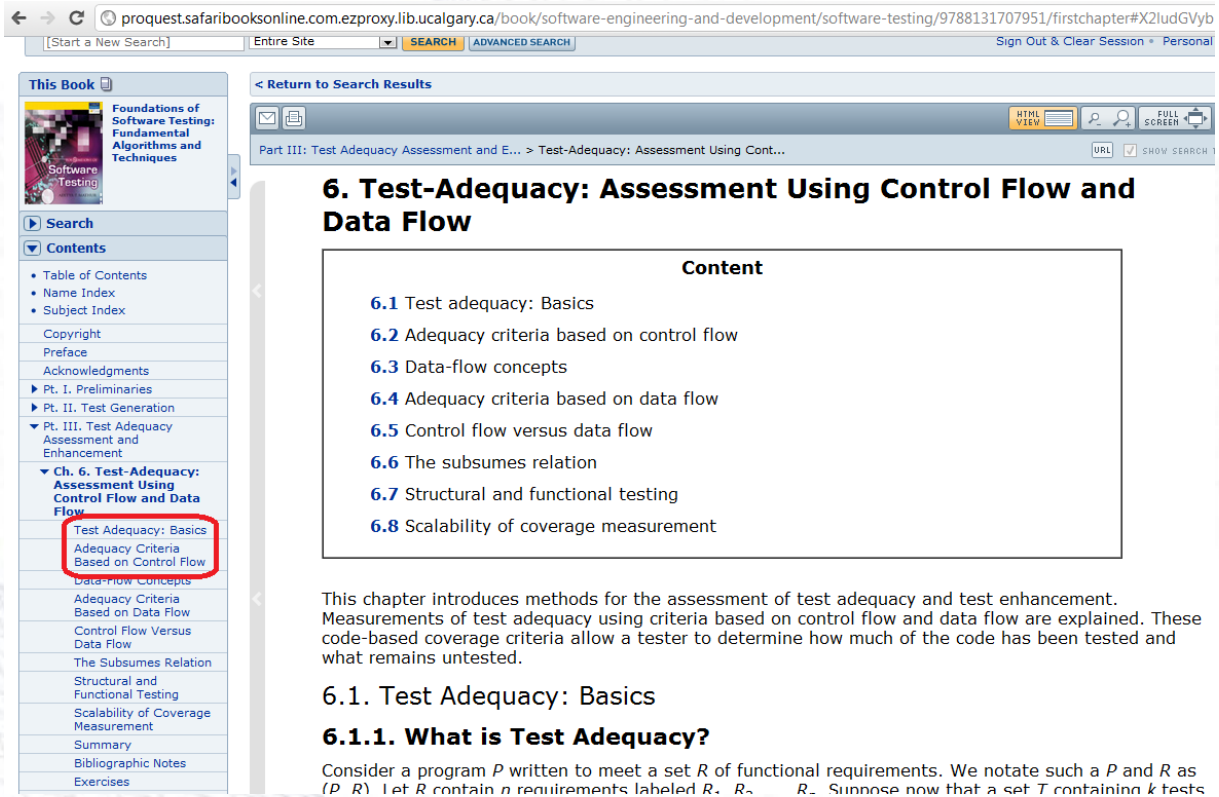
2 → (3 → 2)* as values of abs(Y)

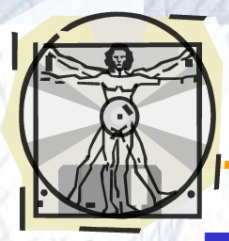




Summary - Control Flow

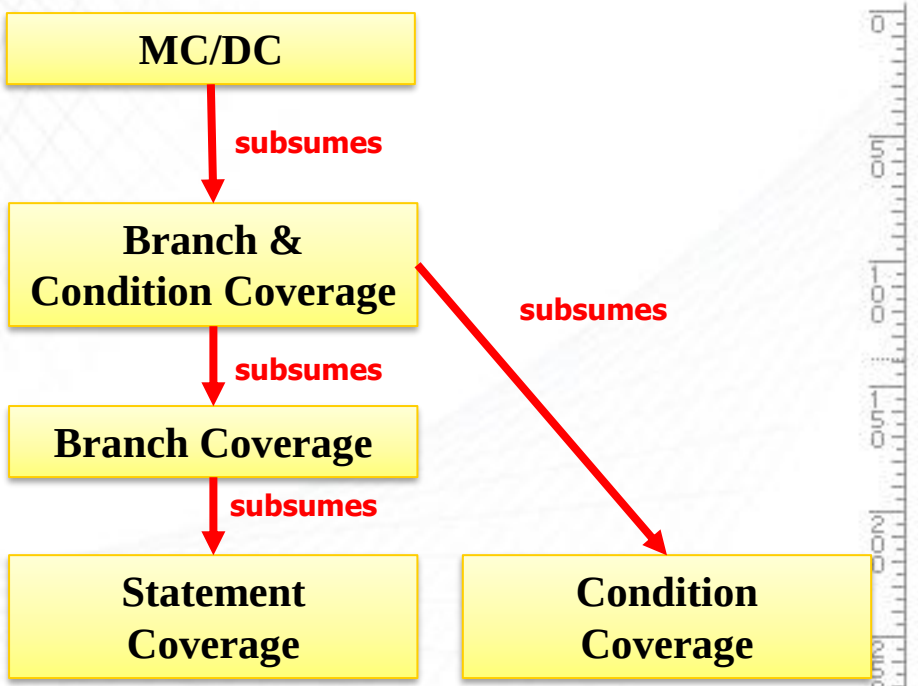
- Further Reading on Control-flow metrics:



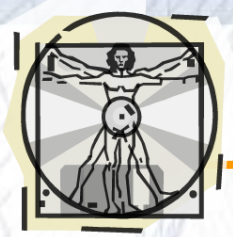


Subsumption

- Subsumption:
  - Coverage criterion C_1 subsumes C_2 if-and-only-if every test set that satisfies C_1 also satisfies C_2 e.g., branch coverage subsumes statement coverage

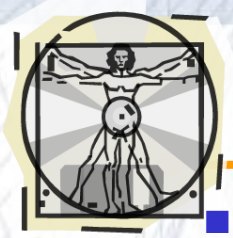


- If you have reached 100% branch coverage, you can deduct that you have also reached 100% statement coverage
- But, if you have reached a certain amount of branch coverage, you cannot deduct that you have the same amount of statement coverage
- MC/DC is the strongest coverage testing criterion



Code Coverage Tools

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



Control Flow Coverage - Reachability

- Not all statements are reachable in real-world programs
- It is **not always possible to decide automatically** if a statement is reachable and the percentage of reachable statements
- When one does not reach a 100% coverage, it is therefore difficult to determine the reason
- Tools are needed to support this activity but it **cannot be fully automated**

See coverage tools

- Research focuses on search algorithms to automate coverage
- Control flow testing is generally more applicable to testing in the small

Exercise - Coverage

1. Does *(x, true, true, true)* lead to 100% statement coverage?
2. Does *(x, true, true, true)* lead to 100% branch coverage?
3. Does *(x, true, true, true)* and *(x, false, false, false)* lead to 100% branch coverage?
4. How many separate paths in this program?
5. Calculate path coverage using

   (x, true, true, true) and

   (x, false, false, false)

```
public int returnInput(int
input, boolean condition1,
boolean condition2, boolean
condition3) {
  int x = input;
  int y = 0;
  if (condition1)
    x++;
  if (condition2)
    x--;
  if (condition3)
    y=x;
  return y;
}
```

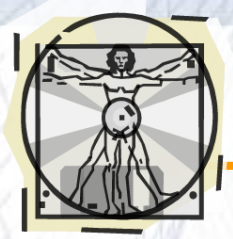
1. **Q1 answer is Yes**
2. **Q2 answer is No only 50%**
3. **Q3 answer is Yes**
4. **Q4 answer is 8**
5. **Q5 answer is 2/8**

Section 3

Cyclomatic Complexity

Cyclomatic Complexity

- A program's complexity can be measured by the cyclomatic number of the program flowgraph
- The cyclomatic number can be calculated in 3 different ways:
  - Flowgraph-based
  - Code-based
  - The numbers of regions of the flow graph

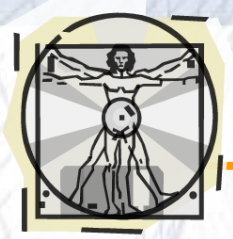


Cyclomatic Complexity /1

- For a program with the program flowgraph *G*, the cyclomatic complexity *v(G)* is measured as:

$$v(G) = e - n + 2p$$

  - *e* : number of edges
    - Representing branches and cycles
  - *n* : number of nodes
    - Representing block of sequential code
  - *p* : number of connected components
    - For a single component, *p=1*



Cyclomatic Complexity /2

- For a program with the program flowgraph *G*, the cyclomatic complexity *v(G)* is measured as:

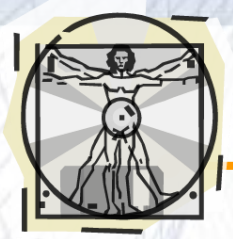
  $$v(G) = 1 + d$$

  - *d* : number of predicate nodes (i.e., nodes with out-degree other than 1)
    - d represents
      - number of loops in the graph

      or

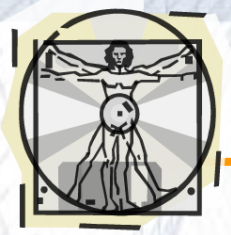
      - number of decision points in the program

i.e., Complexity of program depends only on predicate nodes (decision points).



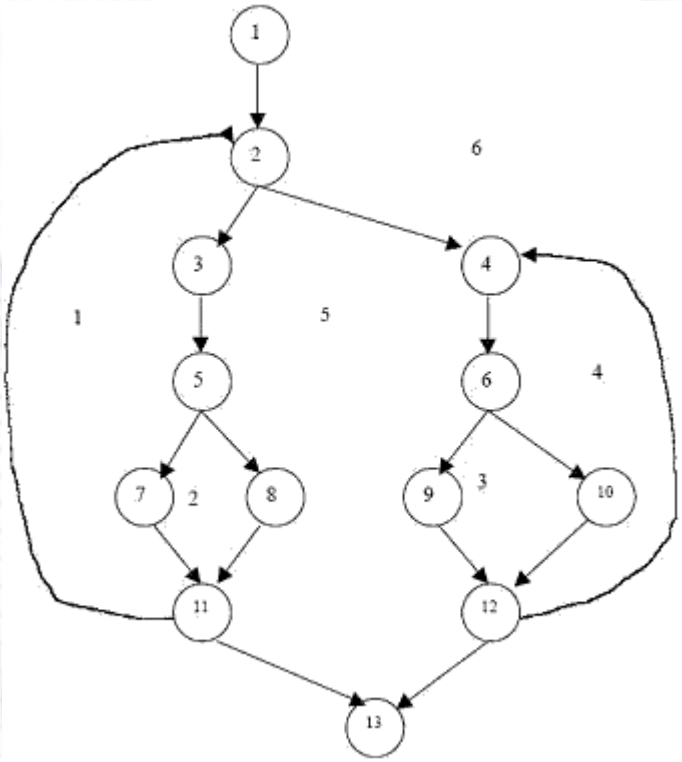
Cyclomatic Complexity /3

- **Cyclomatic Complexity** for a flow graph is the numbers of regions of the flow graph



Cyclomatic Complexity

- Example



Region, R= 6
Number of Nodes = 13
Number of edges = 17
Number of Predicate Nodes = 5

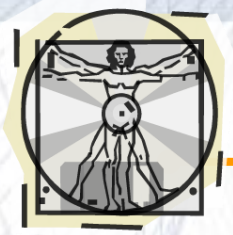
Cyclomatic Complexity, V(C) :
V(C) = R = 6;

Or

V(C) = Predicate Nodes + 1
= 5+1 = 6

Or

V(C)= E-N+2
= 17-13+2 = 6



Cyclomatic Complexity: Example

$$v(G) = e - n + 2p$$

$$v(G) = 7 - 6 + 2 \times 1$$

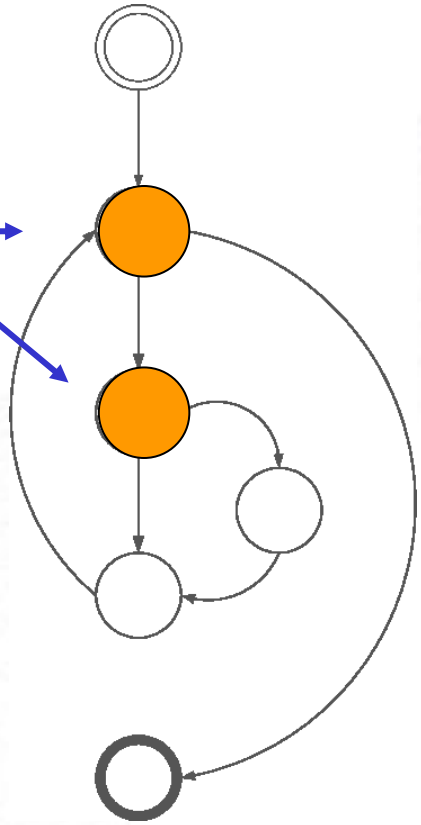
$$v(G) = 3$$

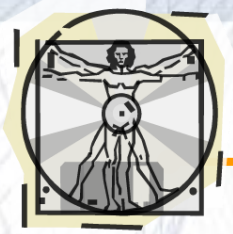
Or

$$v(G) = 1 + d$$

$$v(G) = 1 + 2 = 3$$

Predicate nodes (decision points)

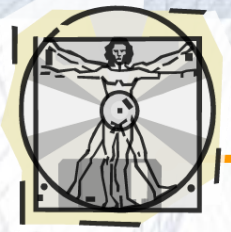




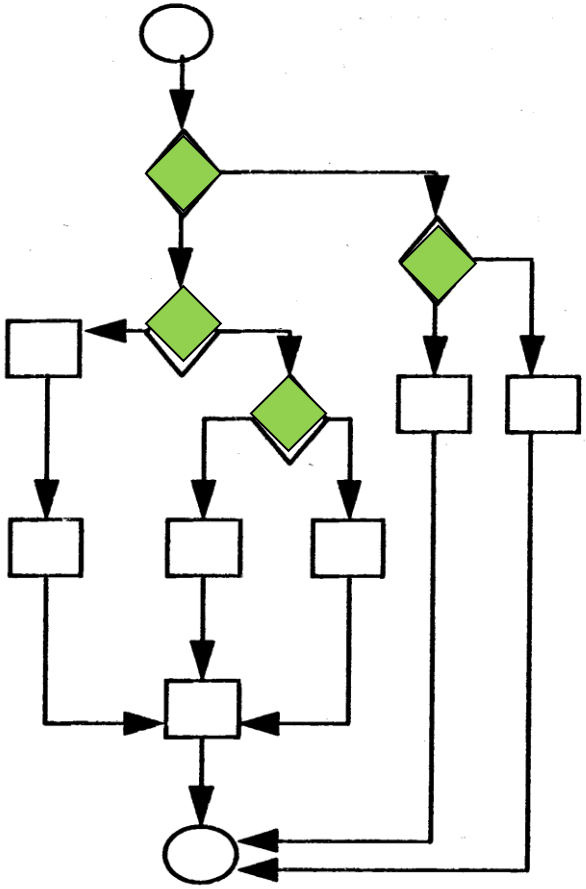
Example: Code Based

```
#include <stdio.h>
main()
{
int a ;
scanf (“%d”, &a);
if ( a >= 10 )
    if ( a < 20 )      printf ("10 < a< 20 %d\n" , a);
    else               printf ("a >= 20    %d\n" , a);
else                   printf ("a <= 10    %d\n" , a);
}
```

$$v(G) = 1+2 = 3$$



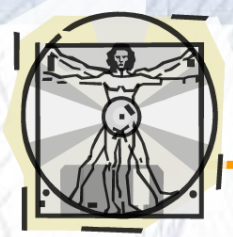
Example: Graph Based



$v(G) = 16 - 13 + 2 = 5$

or

$v(G) = 4 + 1 = 5$



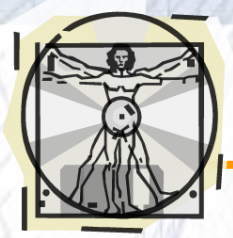
Example 1

- Determine cyclomatic complexity for the following program:

v = 1+d

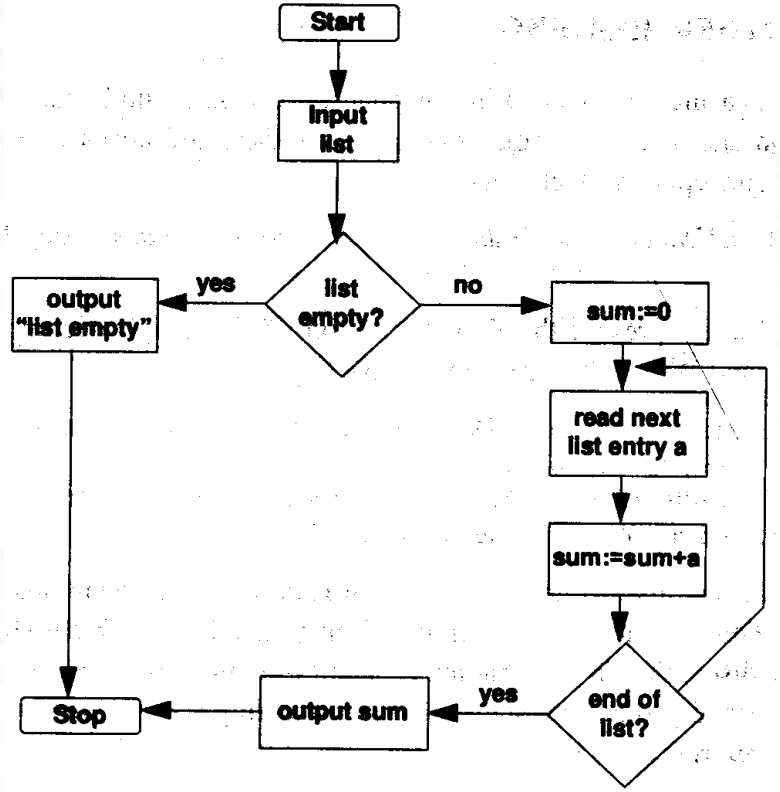
v = 1+6 = 7

```
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. }
```



Example 2

- Determine cyclomatic complexity for the following flow diagram:

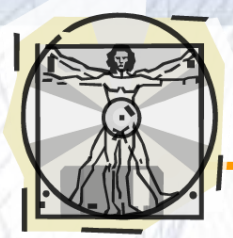


v = 1+d **v = 1+2 = 3**

or

v = e – n + 2

v = 11 – 10 + 2 = 3



Example 3

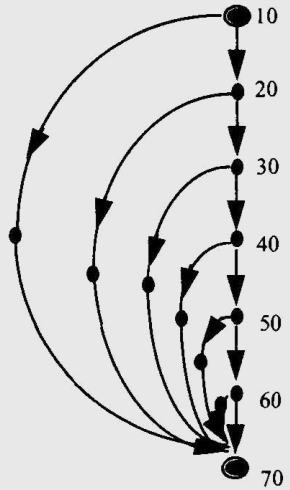
- Two functionally equivalent programs that are coded differently
- Calculate cyclomatic complexity for both

$$V_A = 7$$
$$V_B = 1$$

There is always a trade-off between control-flow and data structure. Programs with higher cyclomatic complexity usually have less complex data structure. Apparently program B requires more effort that program A.

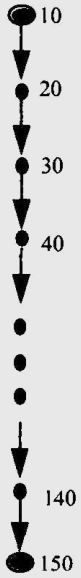
Program A

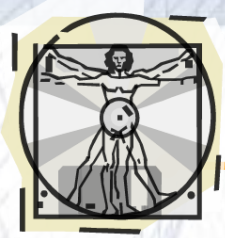
```
10 If score >= 80 Then Print " Grade A": GOTO 70
20 If score >= 70 Then Print " Grade B": GOTO 70
30 If score >= 60 Then Print " Grade C": GOTO 70
40 If score >= 50 Then Print " Grade D": GOTO 70
50 If score >= 40 Then Print " Grade E": GOTO 70
60 If score< 40     Then Print " Grade F": GOTO 70
70 END
```



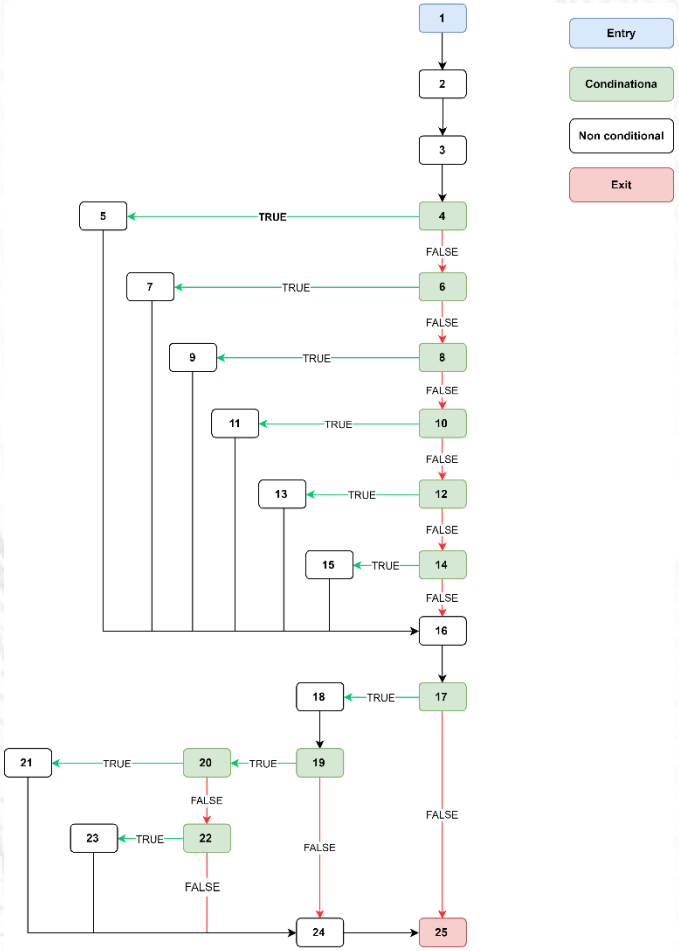
Program B

```
10 DIM A$(11)
20 A$(10) = " Grade A"
30 A$(9) = " Grade A"
40 A$(8) = " Grade A"
50 A$(7) = " Grade B"
60 A$(6) = " Grade C"
70 A$(5) = " Grade D"
80 A$(4) = " Grade E"
90 A$(3) = " Grade F"
100 A$(2) = " Grade F"
120 A$(1) = " Grade F"
130 A$(0) = " Grade F"
140 I = Int (score / 10)
140 Print A$(I)
150 END
```





Exercise: Draw CFG




```
int main() {
1.    int i, grade = 0;
2.    printf (" Enter points: \n");
3.    scanf ("%d", &i);
4.    if (i >= 50 && i <= 60)
5.       grade = 5;
6.    else if (i > 50 && i <= 60)
7.       grade = 6;
8.    else if (i > 60 && i <= 70)
9.       grade = 7;
10.   else if (i > 70 && i <= 80)
11.      grade = 8;
12.   else if (i > 80 && i <= 90)
13.      grade = 9;
14.   else if (i > 90 && i <= 100)
15.      grade = 10;
16.   char sign = ' ';
17.   if (grade) {
18.      int p = i % 10;
19.      if (grade != 5) {
20.         if (p >= 1 && p <= 3)
21.            sign = '-';
22.         else if (grade != 10 && (p >= 8 || p == 0))
23.            sign = '+';
       }
24.      printf (" The grade is %d%c. \n", grade, sign);
    }
25.   return 0;
 }
```