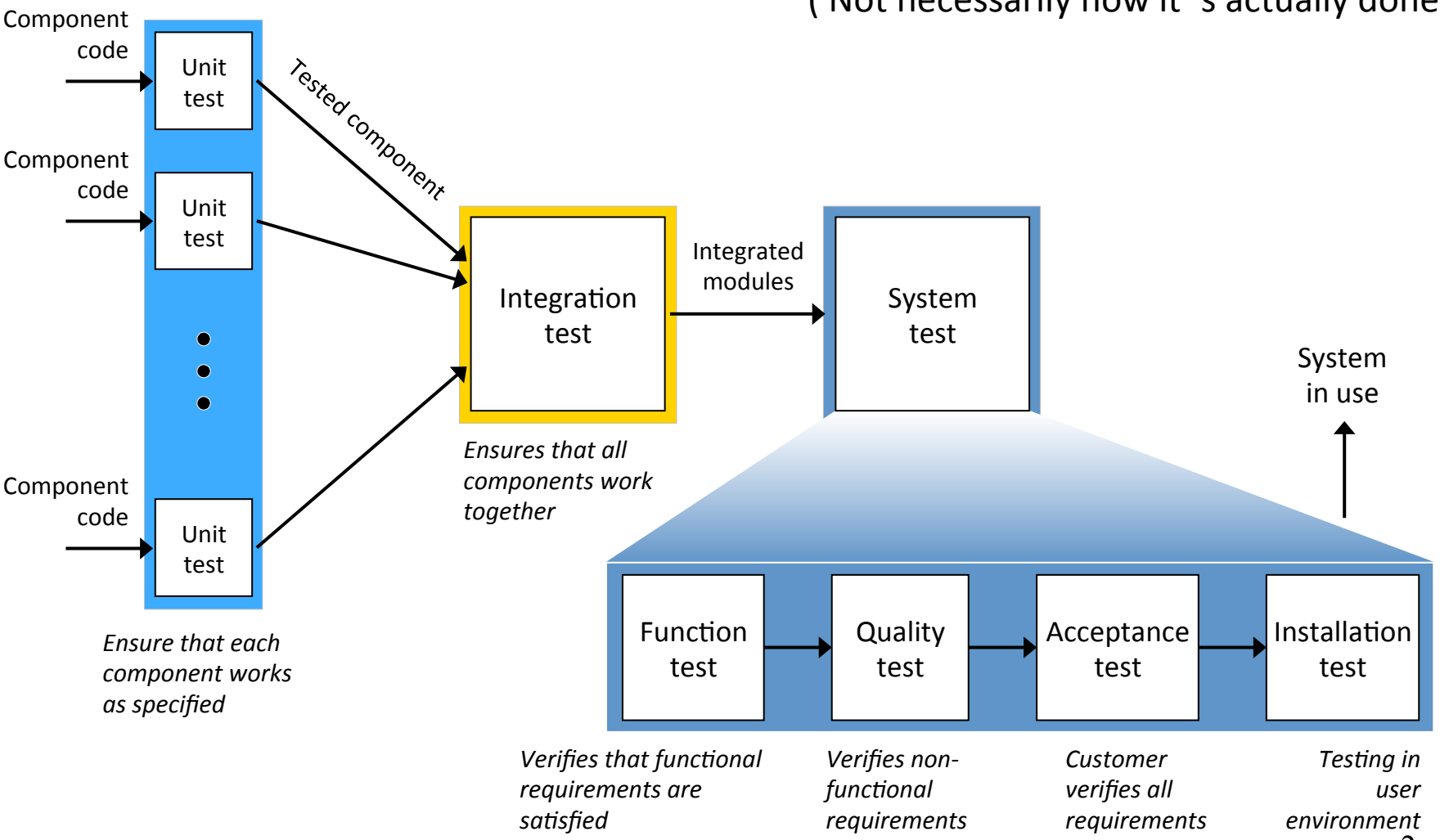
CMPS411
Spring 2018

Lecture 12

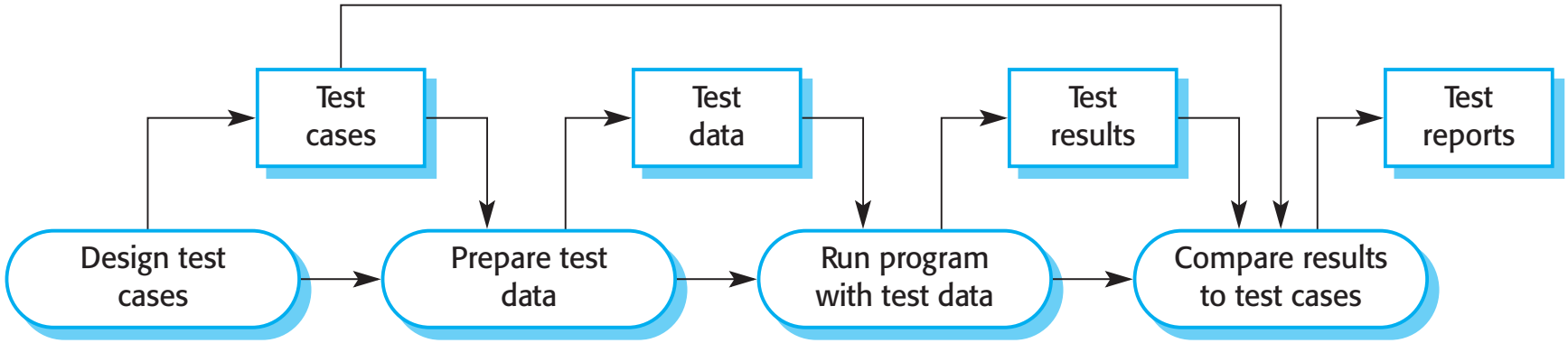
White-Box Testing

Logical Organization of Testing

(Not necessarily how it's actually done!)

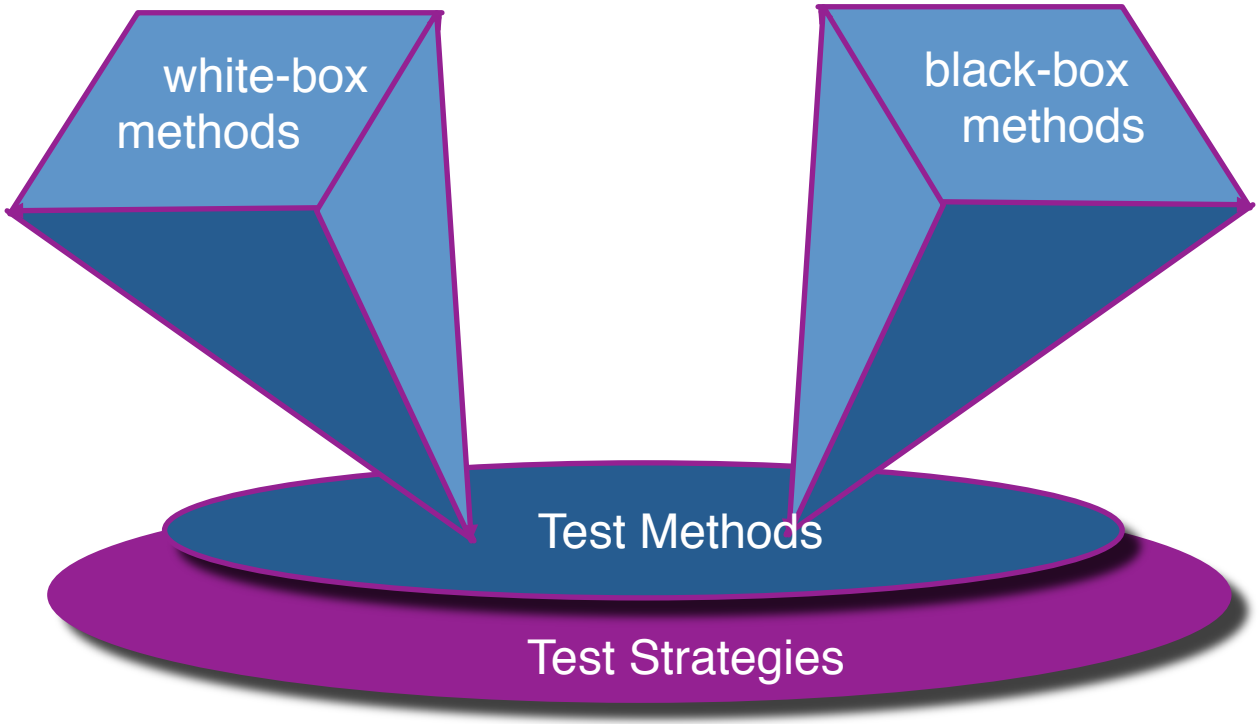


A model of the software testing process

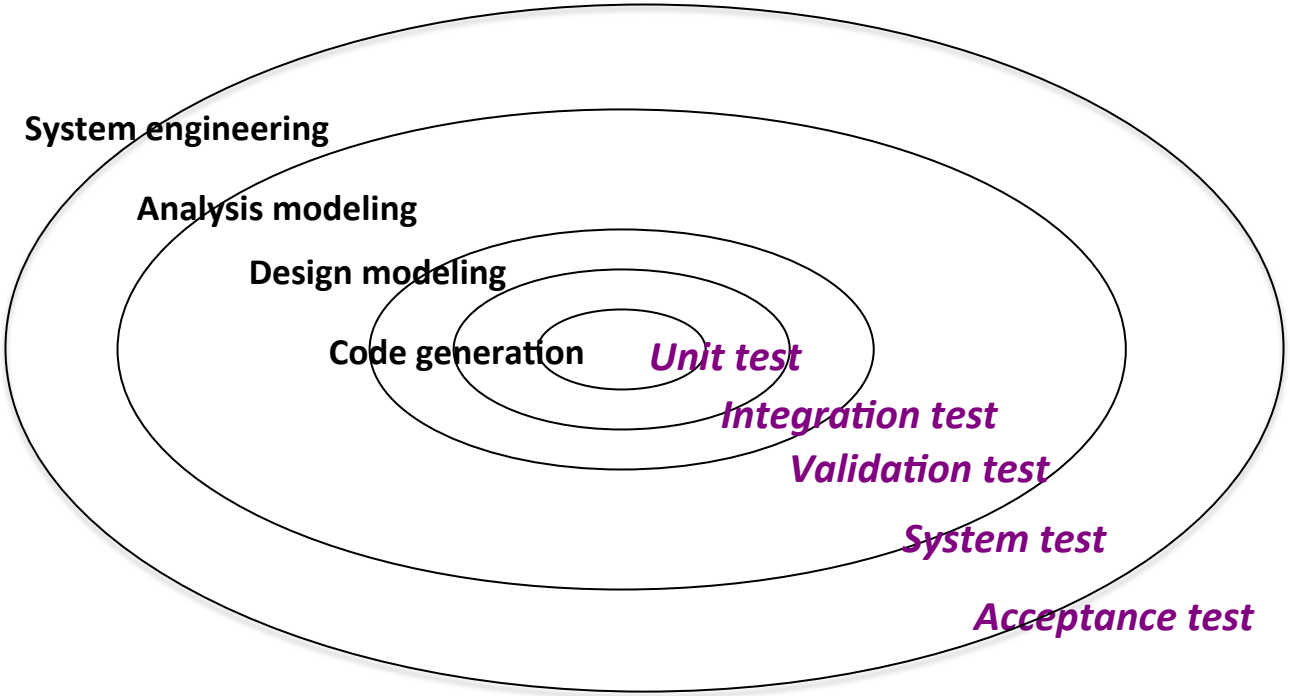


- A **test case** : execution of the program that may expose a bug
- **Test suite**: a *set* of executions of a program, grouped together
- The mechanism for determining whether a software program or system has passed or failed such a test is known as a ***test oracle***.
- In some settings, an oracle could be a **requirement** or **use case**
- It may take many test cases to determine that a software program or system is considered to be released.

Software Testing



Testing Strategy

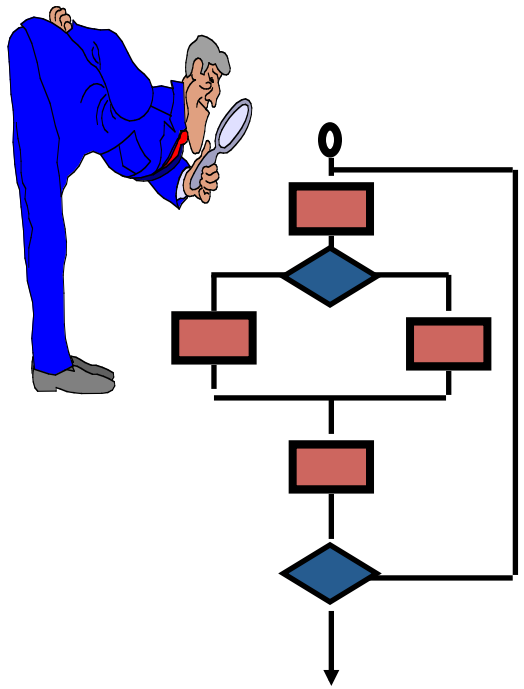


Test Method: White-Box Testing

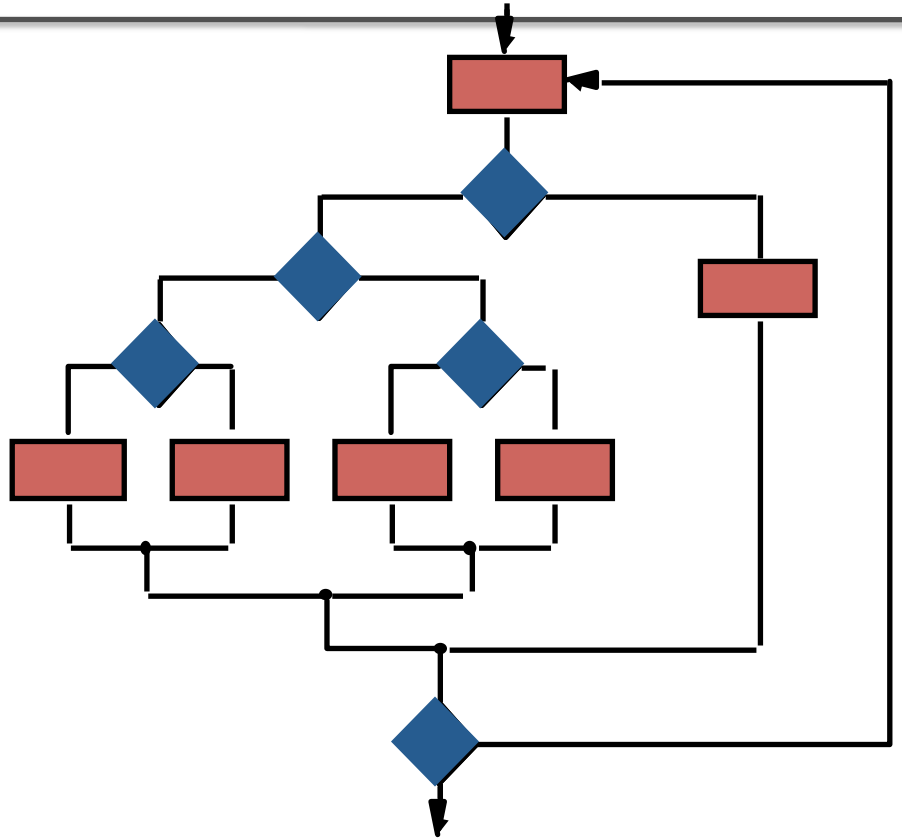
... our goal is to ensure that all statements and conditions have been covered (executed) at least once ...

Unit testing

1. Coverage testing
2. Cyclomatic complexity
3. Morphology
4. Coupling
5. Information flow testing
6. System Complexity
7. Structural complexity
8. Intramodular Data complexity
9. Intermodular Complexity

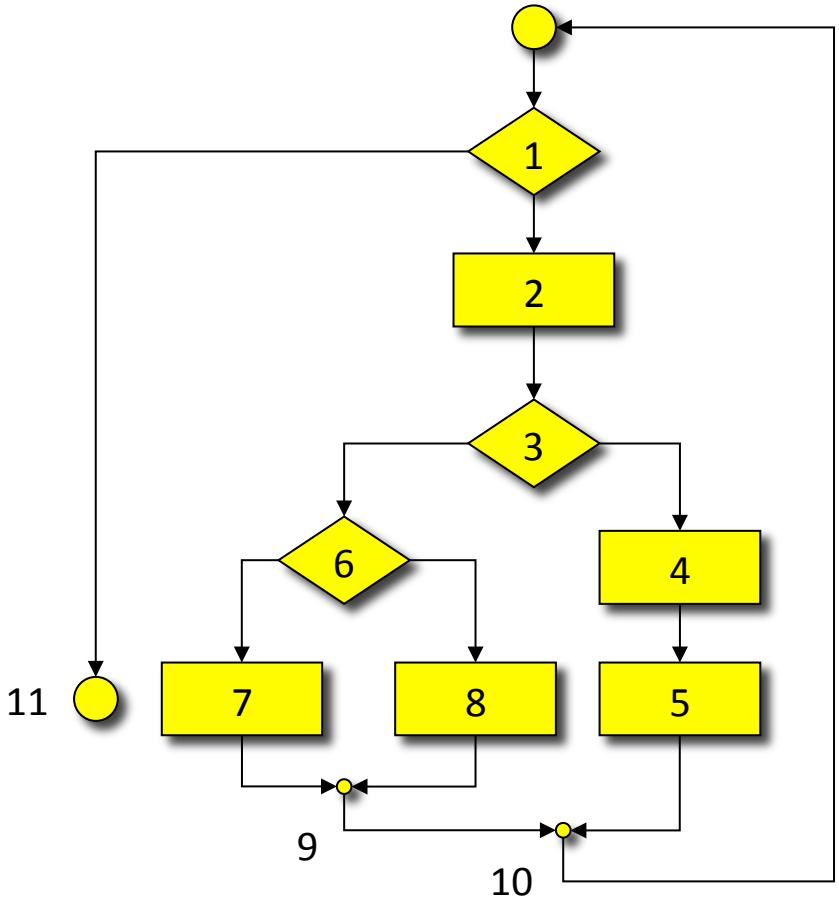


Exhaustive Testing

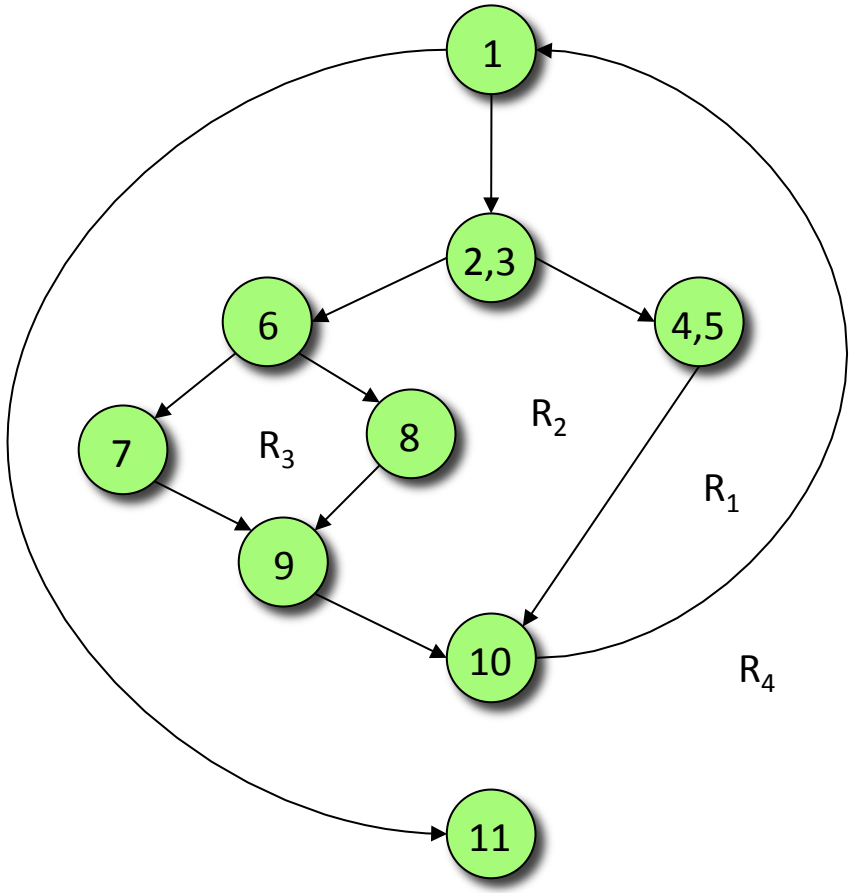


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Basics of Flow Graph and Flowchart



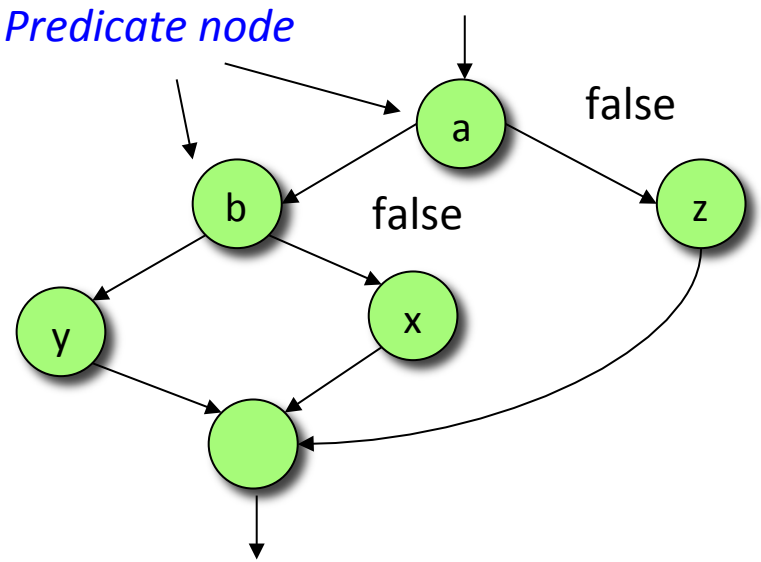
Flowchart

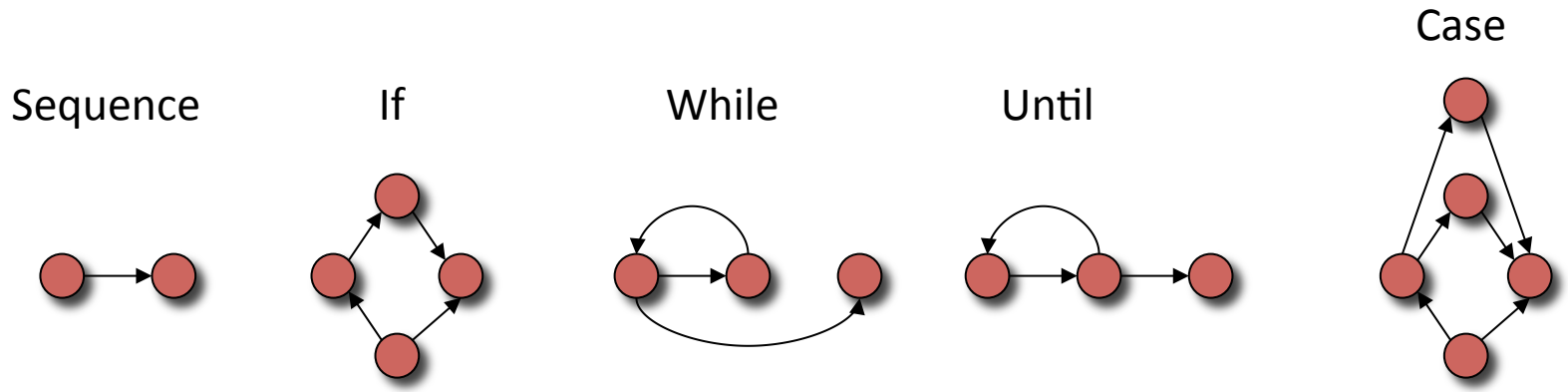


Flow Graph

More on Flow Graph and Variations

IF **a** and **b**
then procedure **y**
else procedure **z** or **x**
ENDIF





Test Coverage

- **Test coverage** measures the degree to which the specification or code of a software program has been exercised by tests
- **Code coverage** measures the degree to which the source code of a program has been tested. It includes:
  - equivalence testing
  - boundary testing
  - control-flow testing
  - state-based testing

Coverage Testing

- Coverage testing
  - ***Statement coverage***: all statements in the programs should be executed at least once
  - ***Branch coverage***: all branches in the program should be executed at least once
  - ***Path coverage***: all execution paths in the program should be executed at lest once
- The best case would be to execute all paths through the code, but there are some problems with this:
  - the number of paths increases fast with the number of branches in the program
  - the number of executions of a loop may depend on the input variables and hence may not be possible to determine
  - most of the paths can be infeasible

Statement Coverage

- Choose a test set *T* such that by executing program *P* for each test case in *T*, each basic statement of *P* is executed at least once
- Executing a statement once and observing that it behaves correctly is not a guarantee for correctness, but it is an heuristic
  - this goes for all testing efforts since in general checking correctness is undecidable

```
bool isEqual(int x, int y)
{
  if (x = y)
    z := false;
  else
    z := false;
  return z;
}
```

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

Statement Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
    print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
    print("y is negative");
}
```

Following test set will give us statement coverage:

T_1 = {(x=12,y=5), (x= -1,y=35),
(x=115, y=-13),(x=-91, y= -2)}

There are smaller test cases which will give us statement coverage too:

T_2 = {(x=12, y= - 5), (x= -1, y=35)}

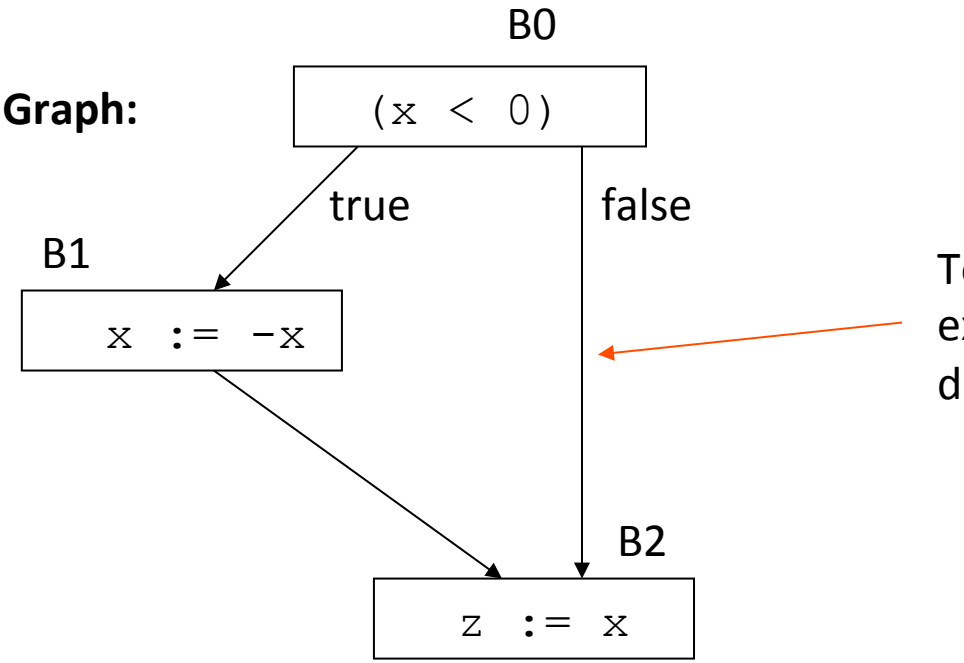
There is a difference between these two test sets though

Statement vs. Branch Coverage

```
assignAbsolute(int x)
{
  if (x < 0)          B0
    x := -x;          B1
  z := x;             B2
}
```

Consider this program segment, the test set T = {x= -1} will give statement coverage, however not branch coverage

Control Flow Graph:

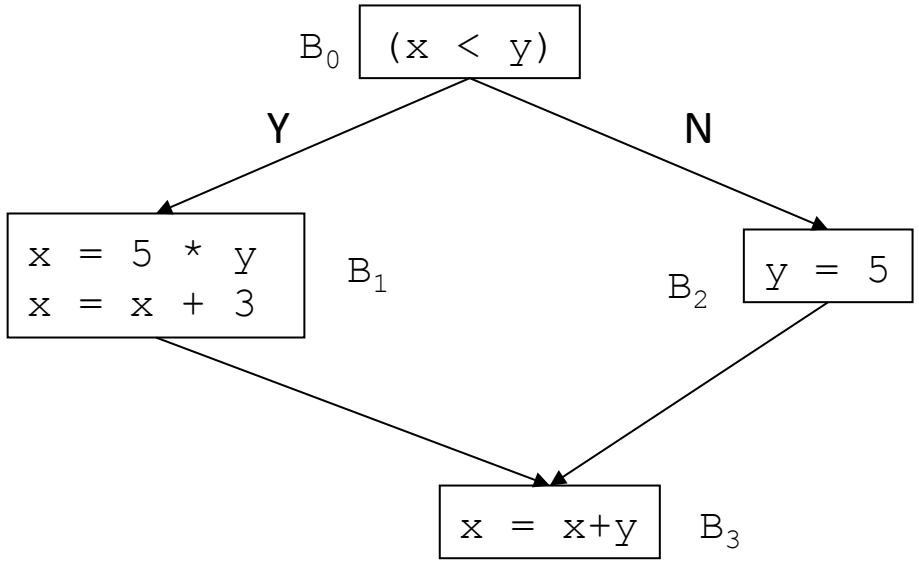


Test set {x=-1} does not execute this edge, hence, it does not give branch coverage

Control Flow Graphs (CFGs)

- Nodes in the control flow graph are basic blocks
  - A ***basic block*** is a sequence of statements always entered at the beginning of the block and exited at the end
- Edges in the control flow graph represent the control flow

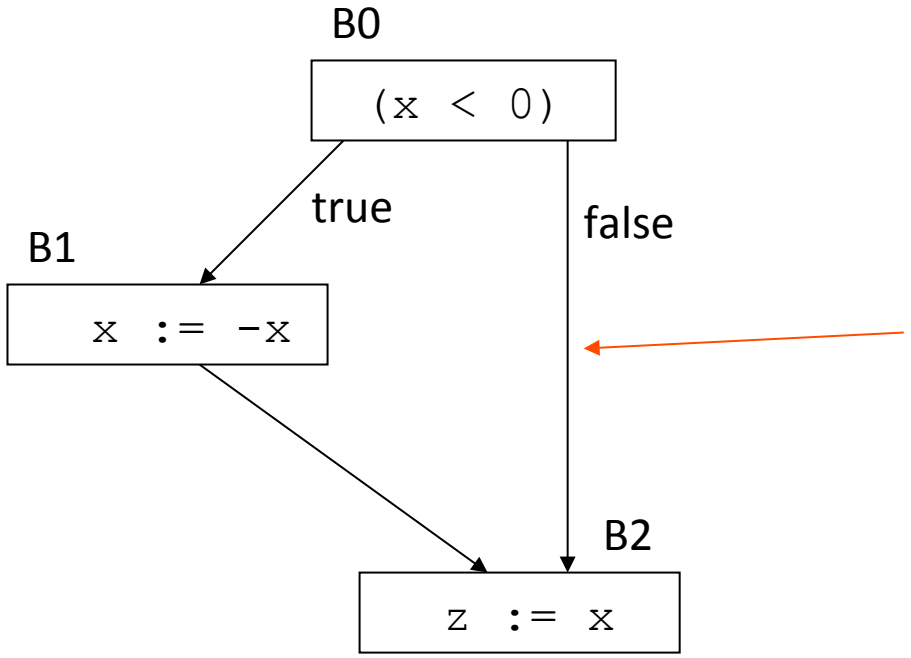
```
if (x < y) {
  x = 5 * y;
  x = x + 3;
}
else  {
  y = 5; }
x = x+y;
```



- Each block has a sequence of statements
- No jump from or to the middle of the block
- Once a block starts executing, it will execute till the end

Branch Coverage

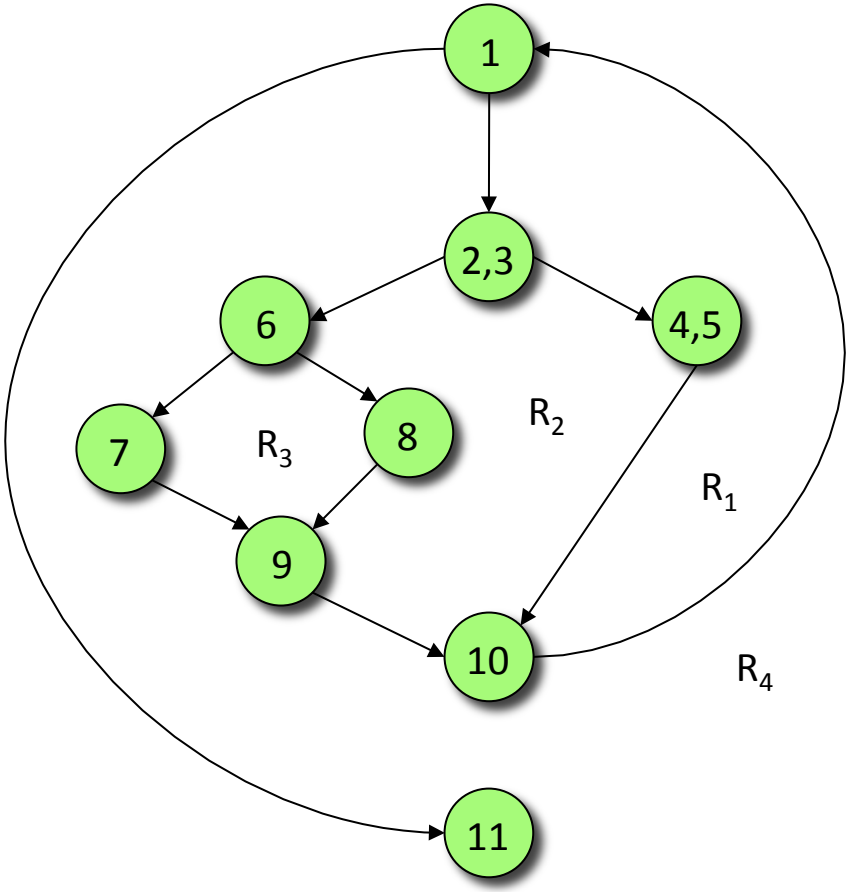
- Construct the control flow graph
- Select a test set *T* such that by executing program *P* for each test case *d* in *T*, each edge of *P*' s control flow graph is traversed at least once



Test set {x=-1} does not execute this edge, hence, it does not give branch coverage

Test set {x= -1, x=2}gives both statement and branch coverage

Basis Path Testing



Basis path set

path 1: 1-11

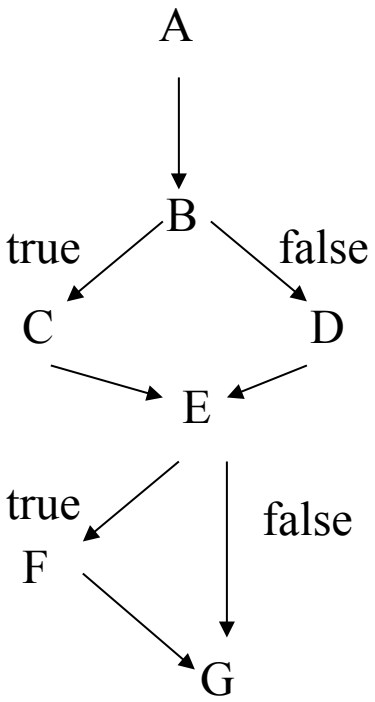
path 2: 1-2-3-4-5-10-1-11

path 3: 1-2-3-6-8-9-10-1-11

path 4: 1-2-3-6-7-9-10-1-11

Path Coverage

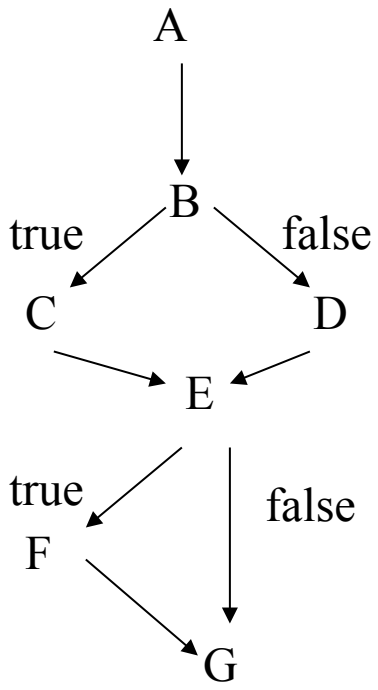
- The most exhaustive white-box testing strategy is to select test cases such that every possible program path is executed at least once. This is called **path coverage**
- Computing every single path through the flow graph
- Sometimes it is not possible to test every path
- An **infeasible path** is a program path that cannot be executed for any input
- The path in the flow graph shown right side, <ABCEFG> is infeasible. Why?
- What would be the output if an input "110" is entered? What path it will execute? What does it signify?
- If we run the program cited below with input "60" and "90", how many statements of the program are covered? How many edges are covered? And how many paths are covered?




```
A     Input(Score)
B     IF score <45
C           then print ( 'Fail' )
D           else print ( 'pass' )
E      If score > 80
F     Then print( 'with distinction' )
G     End.
```

Path Coverage Testing: An Example

```
A    Input(Score)
B    IF score <45
C         then print ( 'Fail' )
D         else print ( 'pass' )
E     If score > 80
F    Then print( 'with distinction' )
G    End.
```



A complete statement coverage

The program executes the path <ABDEFG> for an input "90"

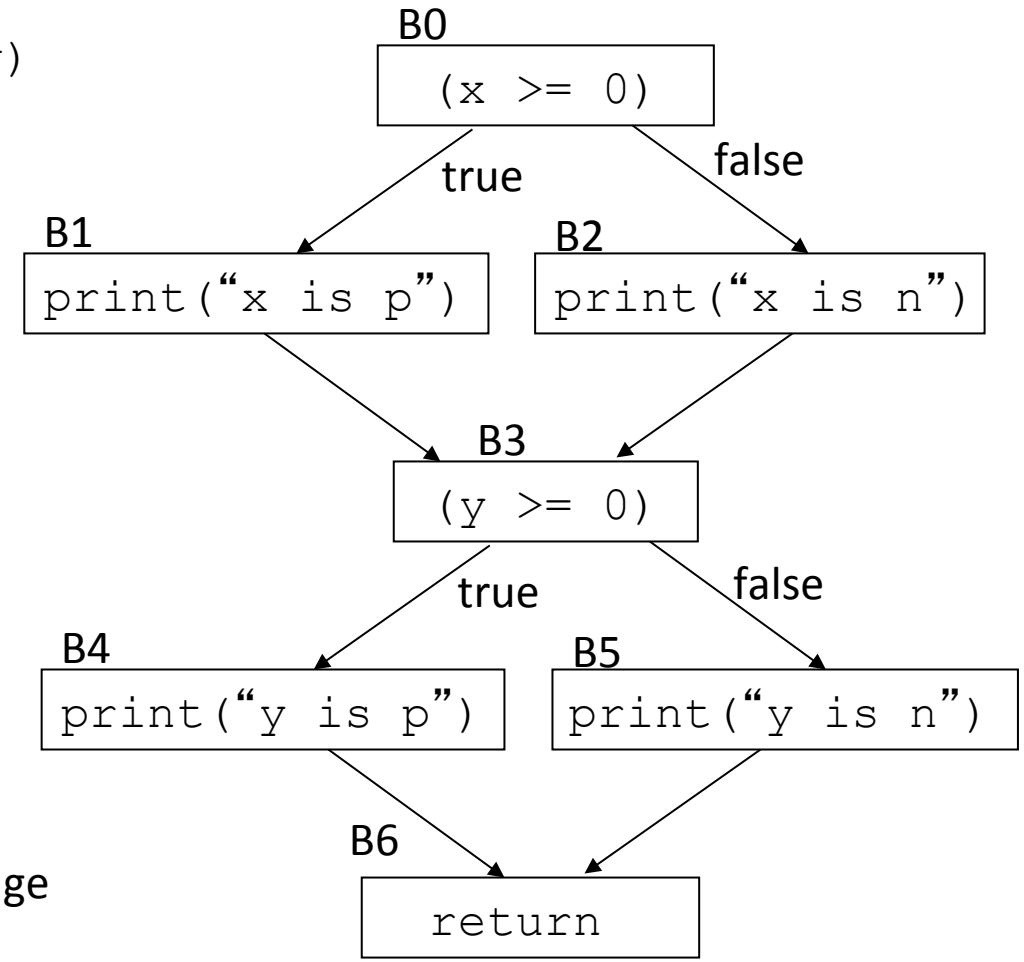
The program executes the path <ABCEG> for an input "40"

•A branch coverage finds a set of path such that every edge lies on at least one path

•The above set of test cases "40" and "90" satisfies branch and statement coverage. However, the same program the two paths ABCEFG and ABDEFG satisfy 100% statement coverage but fail to cover the edge EG.

Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```



Test set:
T_2 = {(x=12, y= - 5), (x= -1, y=35)}
gives both branch and statement coverage but it does not give path coverage

Set of all execution paths: {(B0,B1,B3,B4,B6), (B0,B1,B3,B5,B6), (B0,B2,B3,B4,B6), (B0,B2,B3,B5,B6)}

Test set T_2 executes only paths: (B0,B1,B3,B5,B6) and (B0,B2,B3,B4,B6)

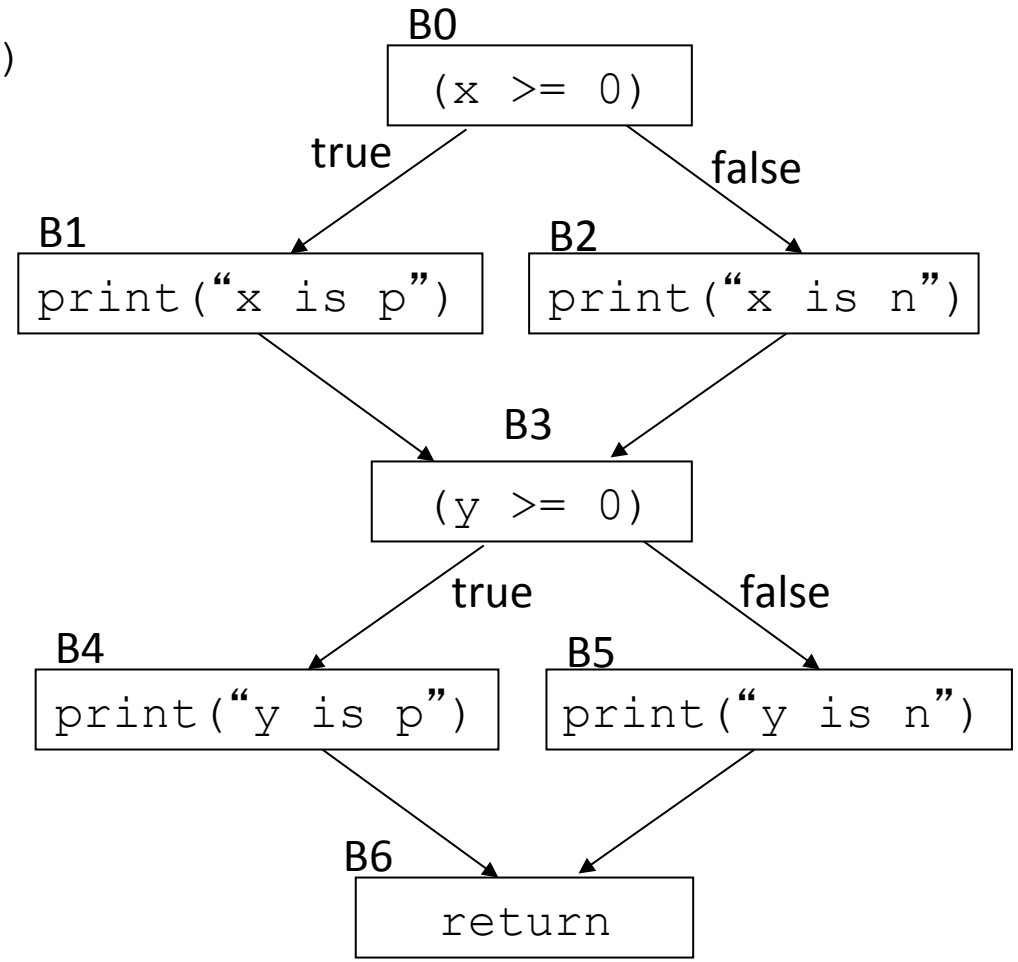
Path Coverage

```
areTheyPositive(int x, int y)
{
  if (x >= 0)
     print("x is positive");
  else
    print("x is negative");
  if (y >= 0)
    print("y is positive");
  else
     print("y is negative");
}
```

Test set:

T_1 = {(x=12,y=5), (x= -1,y=35),
(x=115,y=-13),(x=-91,y= -2)}

gives both branch, statement and path coverage



Path Coverage

- Number of paths is exponential in the number of conditional branches
  - testing cost may be expensive

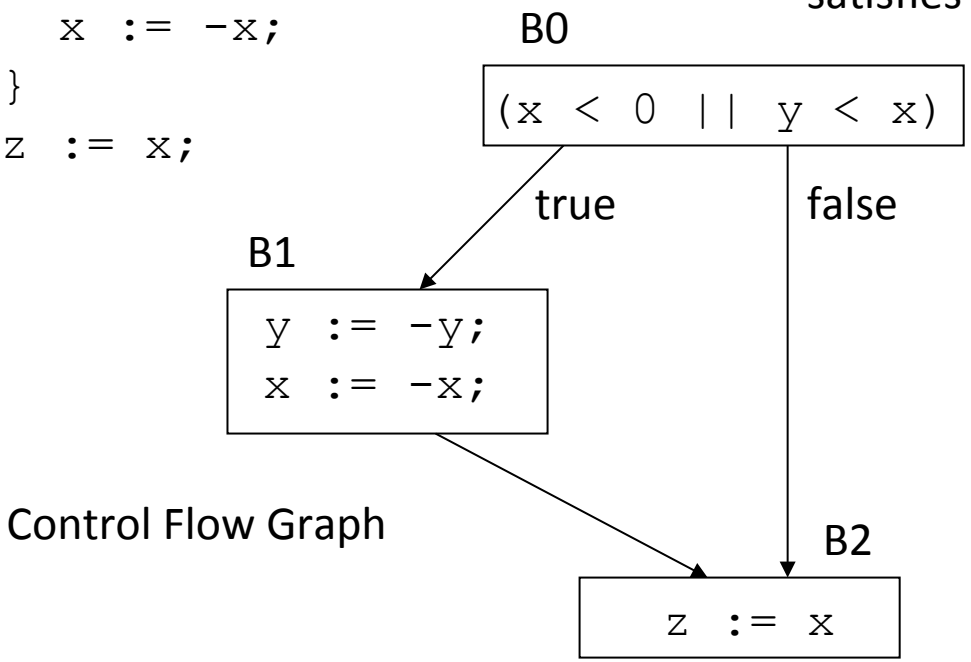
- Note that every path in the control flow graphs may not be executable (Infeasible path)
  - It is possible that there are paths which will never be executed due to dependencies between branch conditions
- In the presence of cycles in the control flow graph (for example loops) we need to clarify what we mean by path coverage
  - Given a cycle in the control flow graph we can go over the cycle arbitrary number of times, which will create an infinite set of paths
  - Redefine path coverage as: each cycle must be executed 0, 1, ..., k times where k is a constant (k could be 1 or 2)

Condition Coverage

- In the branch coverage we make sure that we execute every branch at least once
  - For conditional branches, this means that, we execute the TRUE branch at least once and the FALSE branch at least once
- Conditions for conditional branches can be compound boolean expressions
  - A compound boolean expression consists of a combination of boolean terms combined with logical connectives AND, OR, and NOT
- Condition coverage:
  - Select a test set *T* such that by executing program *P* for each test case *d* in *T*,
    - **(1)** each edge of *P*' s control flow graph is traversed at least once, **and**
    - **(2)** each boolean term that appears in a branch condition takes the value TRUE at least once and the value FALSE at least once
- Condition coverage is a refinement of branch coverage.

Condition Coverage

```
something(int x)
{
  if (x < 0 ||  y < x)
  {
    y := -y;
    x := -x;
  }
  z := x;
}
```



Control Flow Graph

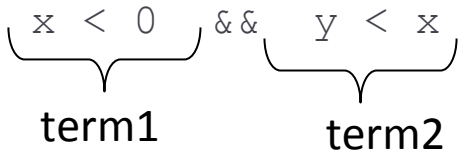
T = {(x=-1, y=1), (x=1, y=1)} will achieve statement, branch and path coverage, however T will not achieve condition coverage because the boolean term `(y < x)` never evaluates to true. This test set satisfies part (1) but does not satisfy part (2).

T = {(x=-1, y=1), (x=1, y=0)} will not achieve condition coverage either. This test set satisfies part (2) but does not satisfy part (1). It does not achieve branch coverage since both test cases take the true branch, and, hence, it does not achieve condition coverage by definition.

T = {(x=-1, y=-2), {(x=1, y=1)} achieves condition coverage.

Multiple Condition Coverage

- Multiple Condition Coverage requires that all possible combination of truth assignments for the boolean terms in each branch condition should happen at least once. For example for the previous example we had:

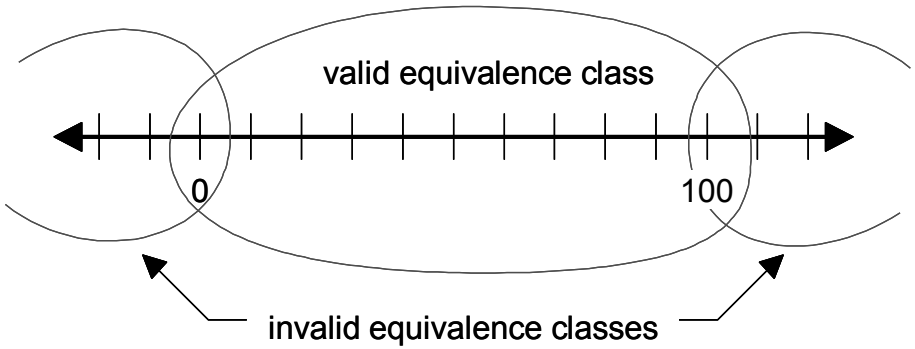


- Test set {(x=-1, y=-2), (x=1, y=1)}, achieves condition coverage:
  - test case (x=-1, y=-2) makes term1=true, term2=true, and the whole expression evaluates to true (i.e., we take the true branch)
  - test case (x=1, y=1) makes term1=false, term2=false, and the whole expression evaluates to false (i.e., we take the false branch)
- However, test set {(x=-1, y= -2), (x=1, y=1)} does not achieve multiple condition coverage since we did not observe the following truth assignments
  - term1=true, term2=false
  - term1=false, term2=true

Code Coverage: Equivalence Testing

- **Equivalence testing** is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program "behaves the same" on each group
- Two steps:
  1. partitioning the values of input parameters into equivalence groups
  2. choosing the test input values

Equivalence classes:



Heuristics for Finding Equivalence Classes

- For an input parameter specified over a **range of values**, partition the value space into one valid and two invalid equivalence classes
- For an input parameter specified with a **single value**, partition the value space into one valid and two invalid equivalence classes
- For an input parameter specified with a **set of values**, partition the value space into one valid and one invalid equivalence class
- For an input parameter specified as a **Boolean value**, partition the value space into one valid and one invalid equivalence class

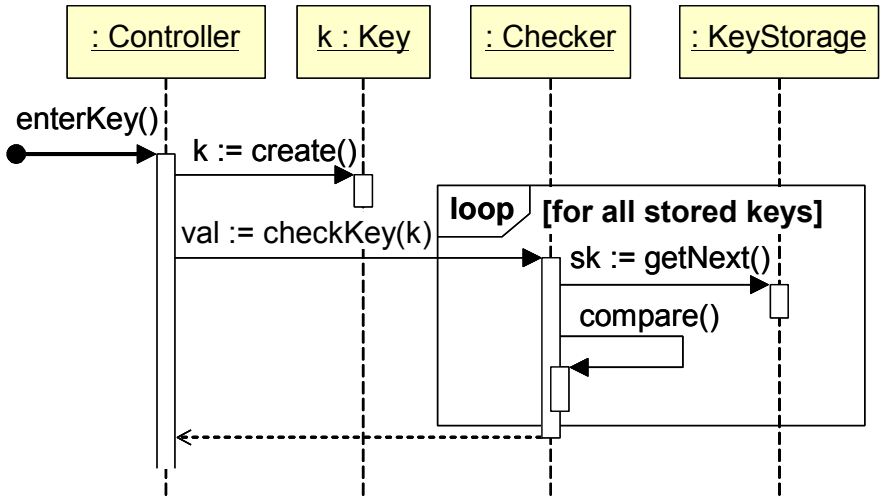
Code Coverage: Boundary Testing

- **Boundary testing** is a special case of equivalence testing that focuses on the boundary values of input parameters
  - Based on the assumption that developers often overlook special cases at the boundary of equivalence classes
- Selects elements from the "edges" of the equivalence class, or "outliers" such as
  - zero, min/max values, empty set, empty string, and null
  - confusion between > and >=
  - etc.

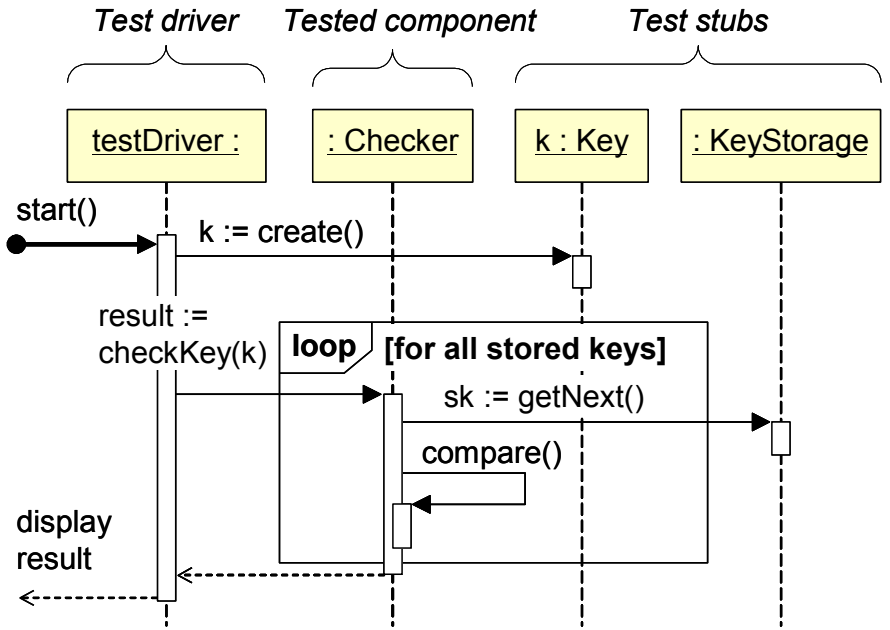
Practical Aspects of Unit Testing

- Mock objects:
  - A test **driver** simulates the part of the system that invokes operations on the tested component
  - A test **stub** simulates the components that are called by the tested component
- The unit to be tested is also known as the **fixture**
- Unit testing follows this cycle:
  1. Create the thing to be tested (fixture), the driver, and the stub(s)
  2. Have the test driver invoke an operation on the fixture
  3. Evaluate that the actual state equals expected state

Testing the KeyChecker (Unlock Use Case)



(a)



(b)

xUnit / JUnit

- Verification is usually done using the `assert_*_()` methods that define the expected state and raise errors if the actual state differs
- http://www.junit.org/
- Examples:
  - `assertTrue(4 == (2 * 2));`
  - `assertEquals(expected, actual);`
  - `assertNull(Object object);`
  - etc.

Example Test Case

Listing 2-1: Example test case for the Key Checker class.

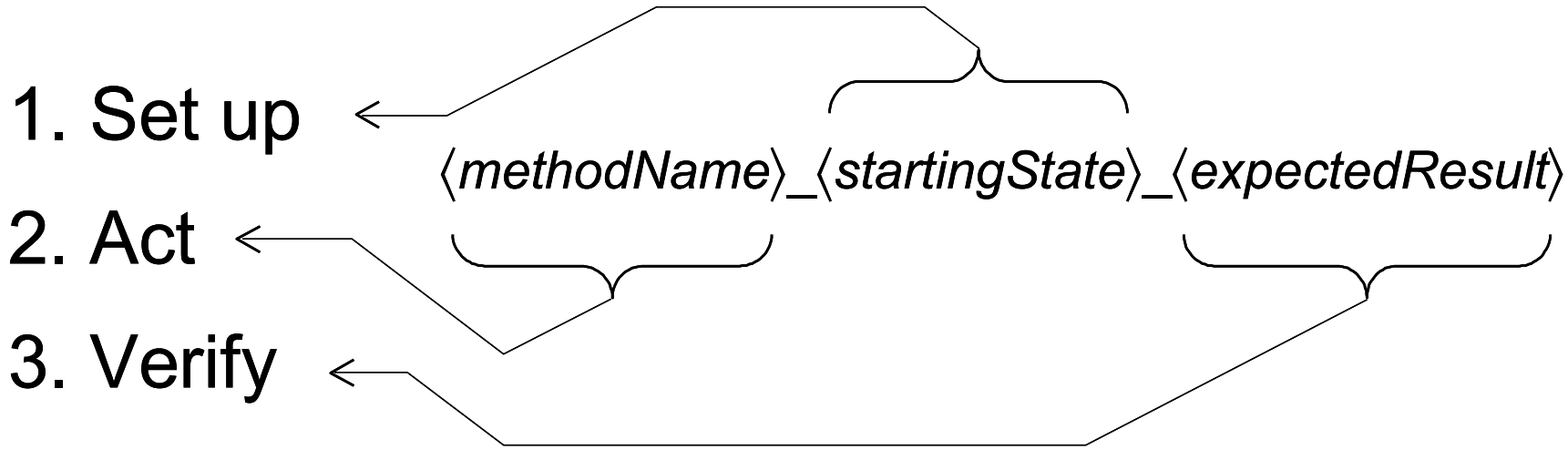
```
public class CheckerTest {
  // test case to check that invalid key is rejected
  @Test public void
    checkKey_anyState_invalidKeyRejected() {

    // 1. set up
    Checker checker = new Checker( /* constructor params */ );

    // 2. act
    Key invalidTestKey = new Key( /* setup with invalid code
*/ );
    boolean result = checker.checkKey(invalidTestKey);

    // 3. verify
    assertEqual(result, false);
  }
}
```

Test Case Method Naming



Example test case method name:

```
checkKey_anyState_invalidKeyRejected()
```

Another Test Case Example

Listing 2-2: Example test case for the Controller class.

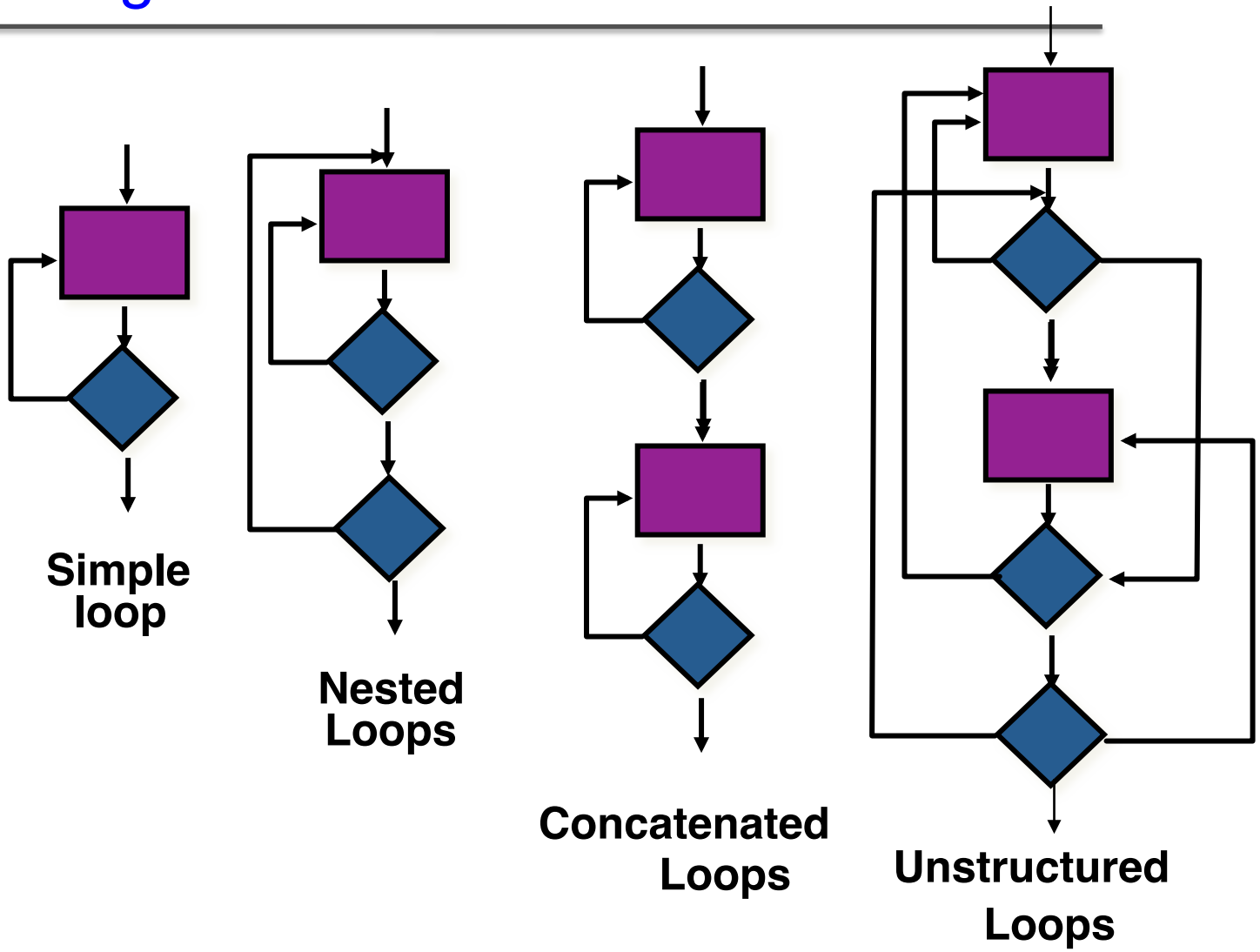
```
public class ControllerTest {
  // test case to check that the state Blocked is visited
  @Test public void
    enterKey_accepting_toBlocked() {

    // 1. set up: bring the object to the starting state
    Controller cntrl = new Controller( /* constructor params */ );
    // bring Controller to the Accepting state, just before it blocks
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    for (i=0; i < cntrl.getMaxNumOfAttempts(); i++) {
        cntrl.enterKey(invalidTestKey);
    }
    assertEqual(   // check that the starting state is set up
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts() - 1
    );

    // 2. act
    cntrl.enterKey(invalidTestKey);

    // 3. verify
    assertEqual(   // the resulting state must be "Blocked"
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts()
    );
    assertEqual(cntrl.isBlocked(), true);
  }
}
```

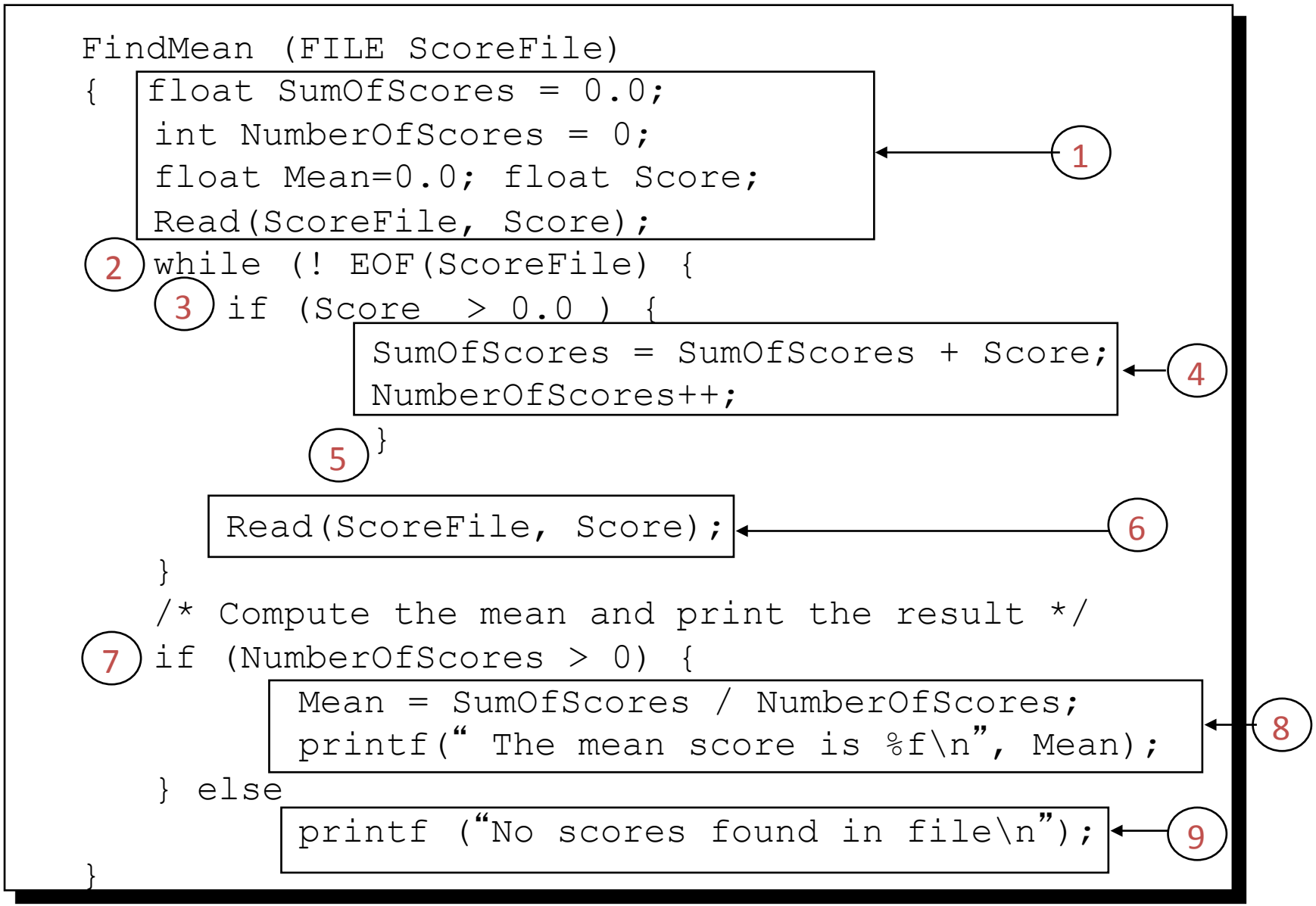
Loop Testing



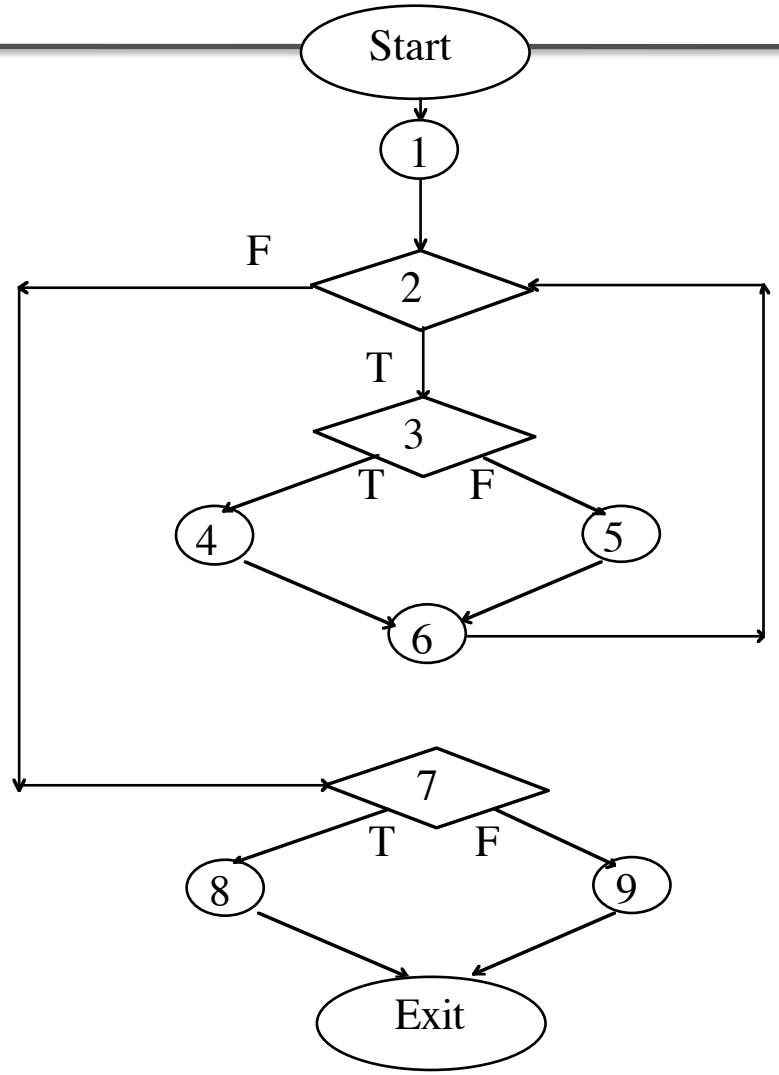
White-box Testing Example

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
              NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
             Mean = SumOfScores/NumberOfScores;
             printf("The mean score is %f \n",  Mean);
 } else
             printf("No scores found in file\n");
}
```

White-box Testing Example: Determining the Paths



Constructing the Logic Flow Diagram



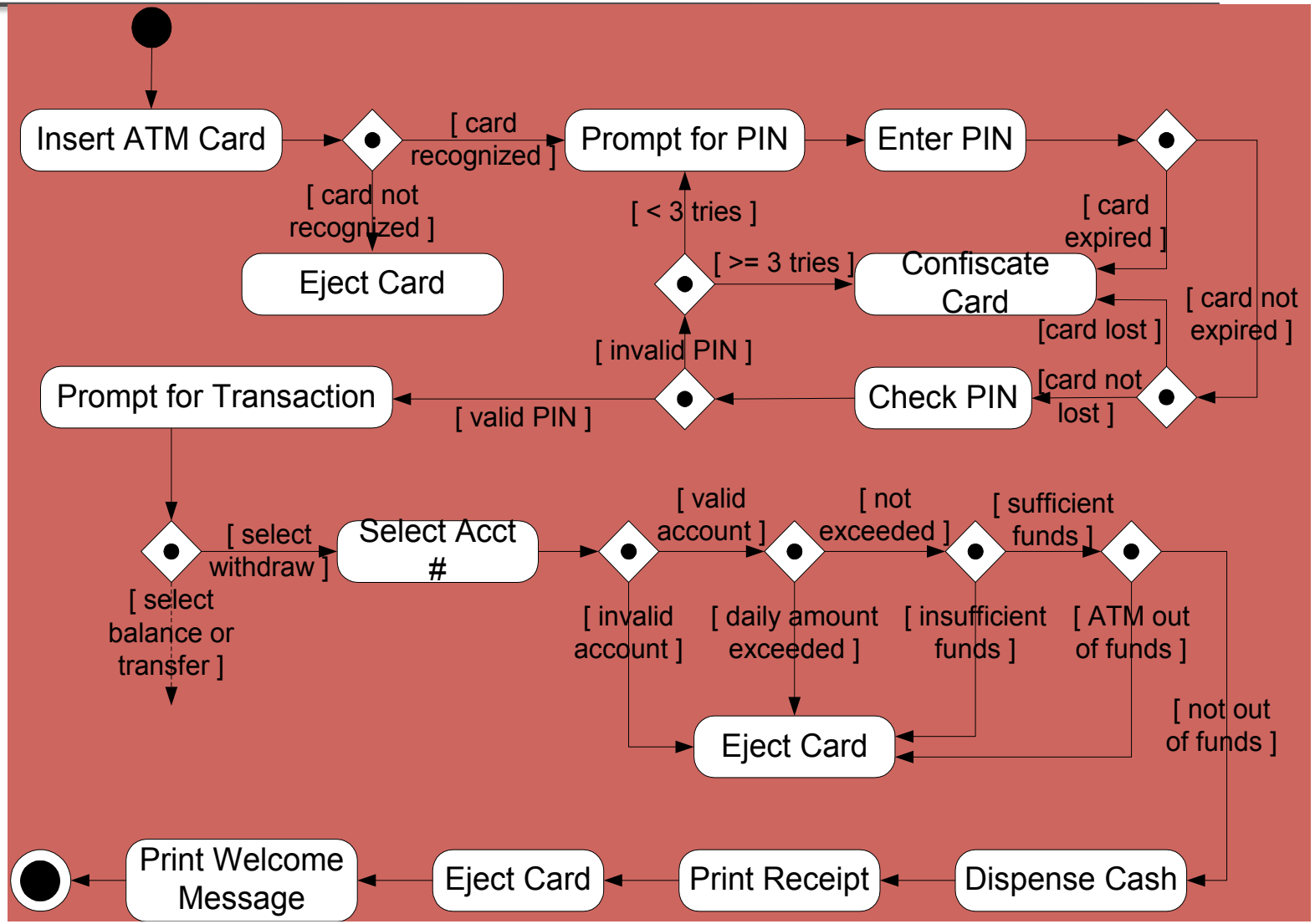
How to Test Use Case?

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

Use-case Testing

- The use-cases developed to identify system interactions can be used as a basis for system testing.
- Each use case usually involves several system components so testing the use case forces these interactions to occur.
- The sequence diagrams associated with the use case documents the components and interactions that are being tested.

A Graph of an ATM Machine System



Covering a Graph of Use Case

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

Cyclomatic Complexity

V(F) = e - n +2p

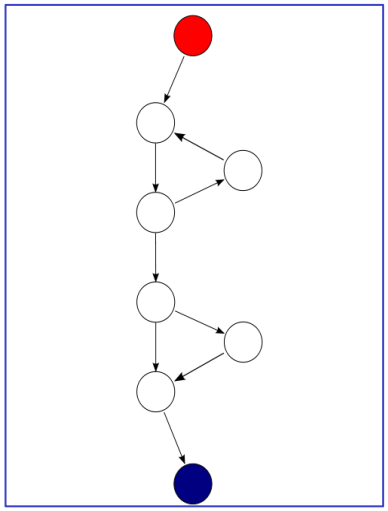
Where, F has e arcs and n nodes

V(F) = cyclomatic number of F

e = number of edges, n = number of nodes,

p = number of exit nodes.

It is based on control flow graph of a module.



- A control flow graph of a simple program.
- The program begins executing at the red node, then enters a loop (group of three nodes immediately below the red node).
- On exiting the loop, there is a conditional statement (group below the loop), and
- Finally the program exits at the blue node.
- This graph has 9 edges, 8 nodes, and 1 exit node.
- So the cyclomatic complexity (V(F)) is: 9 – 8 + (2 * 1).
- No program module should exceed a cyclomatic complexity of 10.

An Example: Cyclomatic Complexity

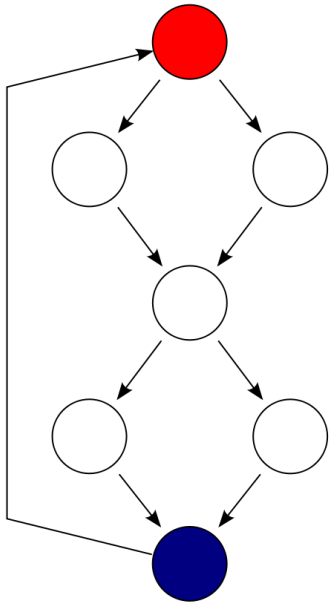
```
if( c1() )
   f1();
else
   f2();

if( c2() )
   f3();
else
   f4();
```

Consider the program above.

- What would be the control flow graph of this program?
- What is the V(F)?

Solution to the Example



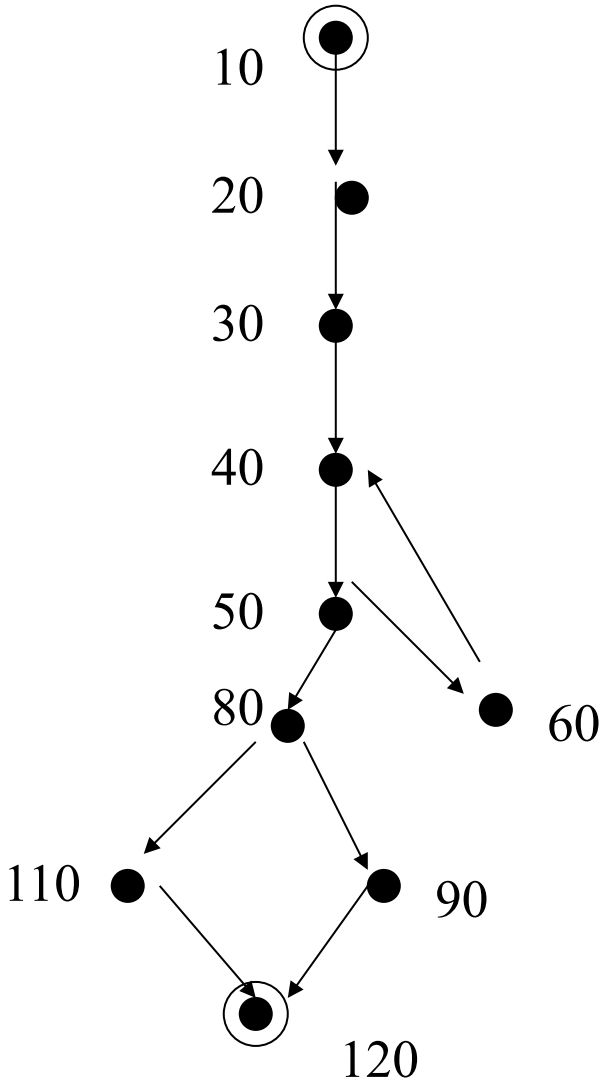
- The cyclomatic complexity of the program is 4
- The program contains 9 edges, 7 nodes and 1 exit node) (9-7+2)

Control Flow graph: Model of Structure

- A graph consists of a set of nodes and edges.
- In a directed graph, each edge is assigned a direction, indicated by an arrowhead on the edge
- This directed edge is called an arc.
- We write an arc as an ordered pair <x, y>, where x and y are the nodes forming the endpoints of the arc, and the arrow indicates that the direction is from x to y.
- The **indegree** of a node is the number of arcs arriving at the node
- The **outdegree** is the number of arcs that leave the node
- A path is a sequence of consecutive (directed) edges
- A simple path is one in which there are no repeated edges.
- We distinguish the start and stop nodes in control flow graphs.

A Simple Control Flow Graph

```
10  INPUT  P
20 Div= 2
30 Lim = INT(SQR(P))
40 Flag = P/Div-INT(P/Div)
50 IF Flag = 0 or Div = lim          THEN 80
60 Div -= Div + 1
70 GO TO 40
80 IF Flag <> 0 or P>4 THEN 110
90 PRINT Div;
100 GO TO 120
110 PRINT P;
120 END
```

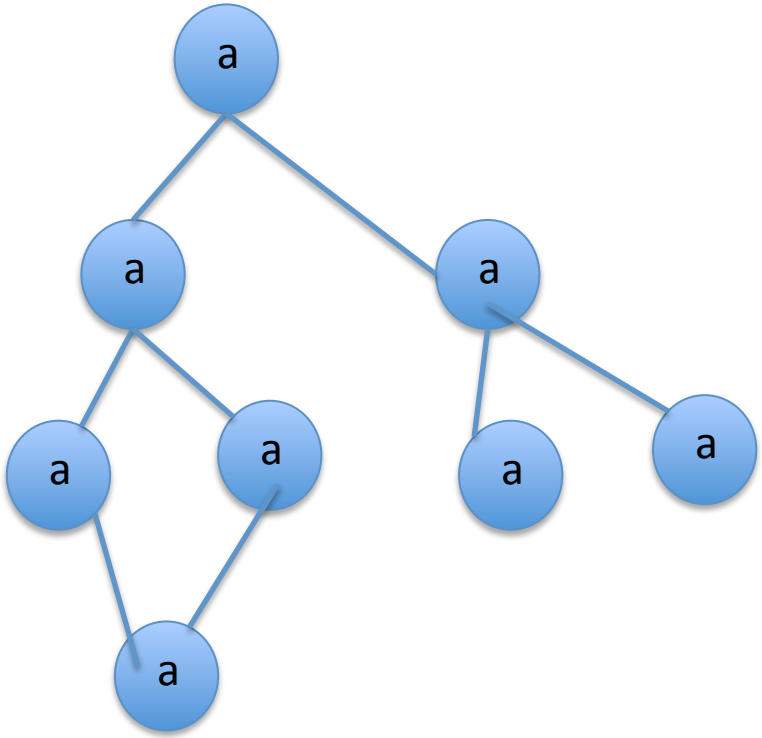


- In this flow graph, the node labeled “50” has indegree 1 and outdegree 2.
- The following sequence S of edge is a path:
- <30,40>, <40,50>, <50,60>, <60,40>, <40,50>,<50,80>
- S is not a simple path. Why?

Computing Morphology

- Morphology is used to refer to the the shape of the overall system structure
- Measure of morphology
  - Size: n + a, where n is the number of nodes and a is the number of arcs
  - Depth : length of the longest path from the root node to a leaf node (number of nodes)
  - Width: maximum number of nodes at any one level of the architecture
  - Arc-to-node ratio: a / n, a kind of connectivity density measure, since it increases as we add more connections between nodes

Example: Computing Morphology



- Size: 8 nodes, 8 edges
- Depth: 3
- Width: 4
- Edge-to-node ratio: 1.00

Coupling

- Coupling is a measure of the degree of interdependence between modules
- Coupling is an attribute of pairs of modules
- Binary relations defined on pairs of modules x, y:

No coupling: R_0: (x, y) E R_0 if x and y no communication, i.e. are totally independent Most desirable.

COUPLING TYPE

1. **Data coupling**: R_1: (x, y) E R_1 if x and y communicate by parameters, each one being either a single data element or a homogeneous set of data items which do not incorporate any control element. (e.g., passing an integer to y that computes a square root).
2. **Stamp coupling**: R_2: (x, y) E R_2 if x and y accept the same record type as a parameter. (e.g., passing a whole record to y that only needs one field of it).
3. **Control coupling**: R_3: (x, y) E R_3 if x passes a parameter to y with the intention of controlling its behaviour (i.e. by passing information to y on what to do)
4. **Common coupling**: R_4: (x, y) E R_4 if x and y refer to the same global data. This type of coupling is undesirable
5. **Content coupling**: R_5: (x, y) E R_5 if x refers to the inside of y, i.e. It branches into, changes data, or alters a statement in y. Most undesirable coupling
   - x modifies or relies on the internal workings of y (e.g., accessing local data of another module).

Methods of Counting Coupling

- Each arc from a node x to a node y represents some coupling between modules x and y
- Each arc is labelled by a pair of numbers where
  - **first number (between 1 and 5) represents the type of coupling, and**
  - **the second number represents the number of times the given type of coupling occurs between x and y**
- A measure **c** of coupling between modules x, y which is on an ordinal scale is:

$$c(x,y) = i + \frac{n}{n+1}$$

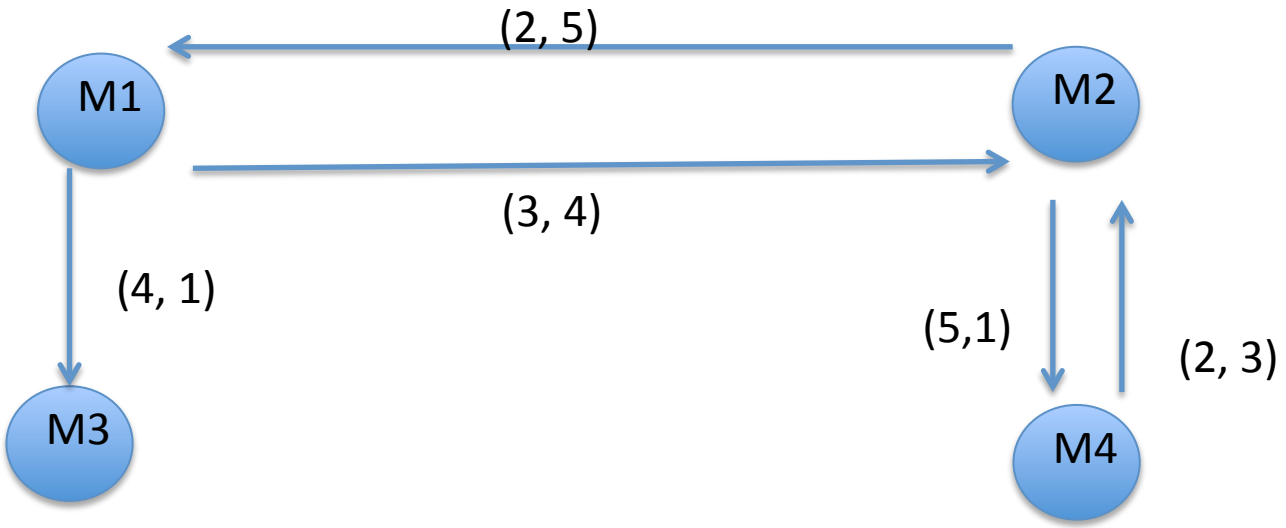
where

i is the number corresponding to the worst coupling type between x and y (e.g., i = 3 if x, y have control coupling)

n is the number of interconnections between x and y

- **Global coupling is the median value of the set {c(Di, Dj): 1 <= i <+ j <= n},**
- **This can be derived from coupling between two modules in a system.**

Example: Counting Coupling



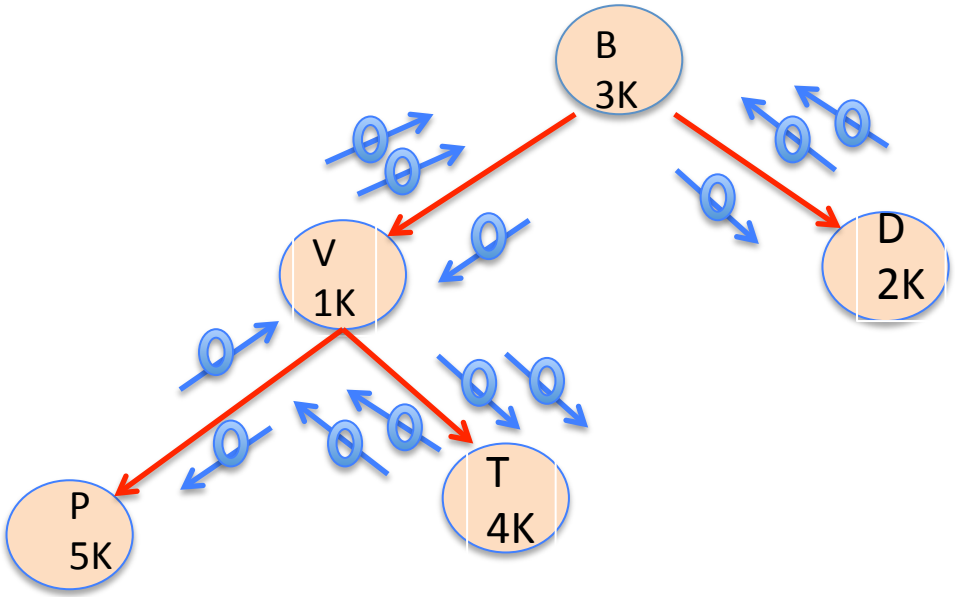
- c(M1, M2) = 3 + (4/(4+1)) = 3.8
- c(M2, M1) = 2 + (5 /(5 + 1)) = 2.83
- c(M1, M3) = 4 + (1/(1+1)) = 4.5
- c(M2, M4) = 5 + (1/(1+1)) = 5.5
- c(M4, M2) = 2 + (3/(3+1)) = 2.75

Global coupling is: 3.8 (the median value)

Information Flow Complexity

- **Fan-in** is a count of the number of modules that call a given module. Fan-in of a module M is the number of local flows that terminate at M, plus the number of data structures from which information is retrieved by M.
- **Fan-out** is a count of the number of modules that are called by a given module. Fan-out of a module M is the number of local flows that emanate from m, plus the number of data structures that are updated by M
- A **global flow** exists if there is a flow of information from one module to another via a global data structures
- Modules with a large fan-in are relatively small and simple and are usually located at the lower layers in the design structure
- Modules that are large and complex are likely to have a small fan-in
- Modules with a large fan-in are expected to have negative or insignificant correlation with defect levels
- Modules with a large fan-out are expected to have a positive correlation with defect rates

Example: Computing Information Flow Complexity



This symbol means data flow. This is a classical symbol used in structured system.

- Information flow complexity (M) =
  - Module ID - Fan-in - Fan-out - (fan-in(M) x fan-out(M))^2 - Length – Complexity

B	-	4	-	2	-	((4 (B) x 2(B))^2 = 64	-	3	-	192
V	-	4	-	5	-	((4 (V) x 5(V))^2 = 400	-	1	-	400
P	-	1	-	1	-	((1 (P) x 1(P))^2 = 1	-	5	-	5
T	-	2	-	2	-	((2 (T) x 2(T))2 = 16	-	4	-	64
D	-	1	-	2	-	((1 (D) x 2(D))2 = 4	-	2	-	8

System Complexity

- It is a sum of structural (intermodule) complexity and overall data (intramodule) complexity
- Structural complexity is defined as the mean (per module) of squared values of fan-out
- Believe that complexity increases as the square of connections between programs (fan-out) as fan-in is not an important complexity indicator
- Data complexity is defined as a function that is directly dependent on the number of I/O variables and inversely dependent on the number of fans-out in the module
- The meanings of the metrics at the module level are as follows:
  - D_i : data complexity of module i
  - S_i : structural complexity of module i,a measure of the module's interaction with other modules
  - $C_i = S_i + D_i$: the module's contributions to overall system complexity

System Complexity Model

- Developed by Card and Glass (1990) based on various approaches to structure complexity and metric complexity

$C_t = S_t + D_t$

where C_t = system complexity

S_t = structural(intermodule) complexity

D_t = Intramodule complexity of the system

= To get D_t, we need to first calculate intramodule complexity of each module in the system. See slides 41 and 42.

Relative system complexity :

$C_r = C_t/n$

where n is the number of modules in the system

Structural (Intermodule) Complexity

S_t (structural complexity) is further defined as

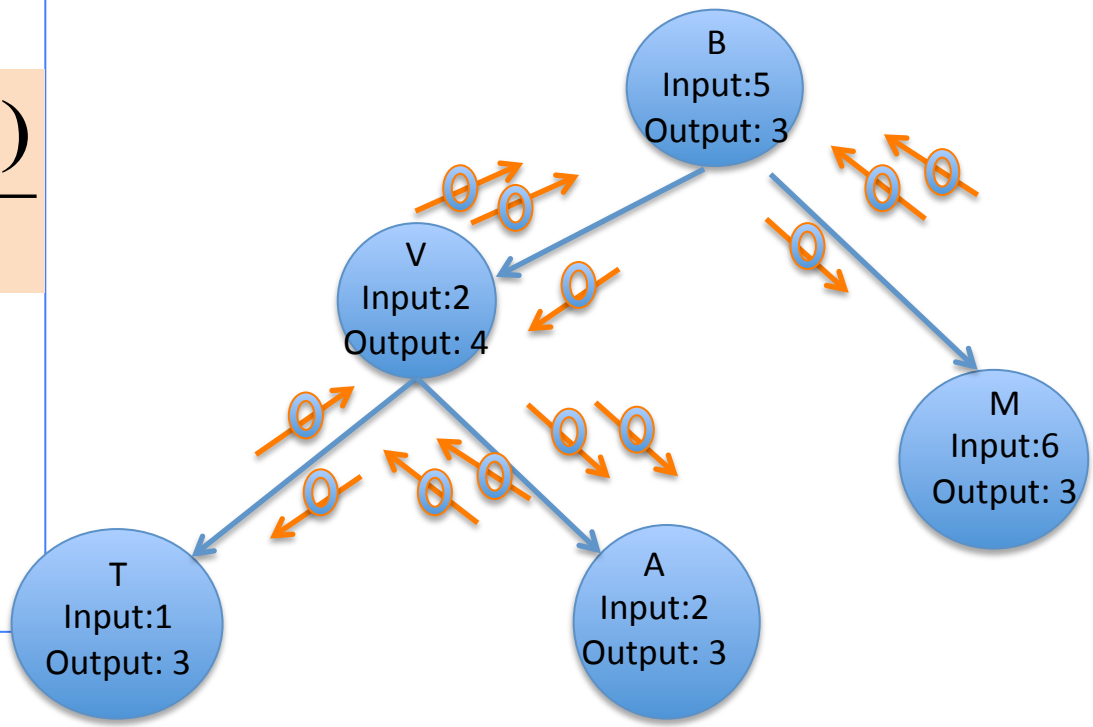
$$S_t = \frac{\sum f \cdot (i)}{n}$$

where

S_t = structural complexity

$f(i)$ = fan-out of module i

n = number of modules in system



Example:
S_t = (2+ 5 + 1 + 2 + 2) \5 = 12/5 = 2.4
Structural complexity of this system is: 2.4

57

Intramodule Data Complexity

Intramodule Data complexity is defined as :

$$D_i = \frac{V(i)}{f(i)+1}$$

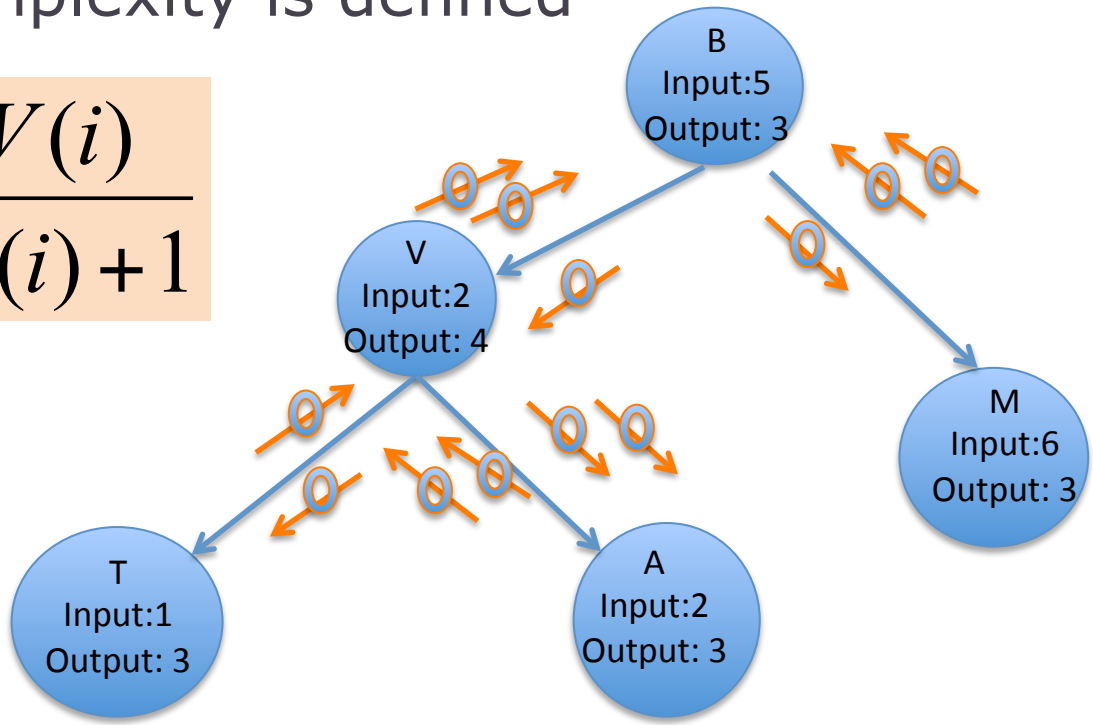
Where

D_i = data complexity of module *i*

V(i) = I/O variables in module *i*

ƒ (i) = fan-out of module *i*

Example:
$D_B = 8/(2+1) = 2.66$
$D_T = 4/(1+1) = 2.00$
$D_A = 5/(2+1) = 1.66$
$D_V = 6/(5+1) = 1.00$
$D_M = 9/(2+1) = 3.00$



Intramodule Data Complexity

Intermodule data complexity is defined as :

$$D_t = \frac{\sum D(i)}{n}$$

Where

D_t = intermodule data complexity

D(i) = data complexity of module i

n = number of module in system

Example:
D_t = (2.66 + 2.00 + 1.66 + 1.00 + 3.00)/5 = 10.32/5 = 2.64

System Complexity Metrics

- System Complexity is:
  - $C_t = S_t + D_t = 2.4 + 2.64 = 5.04$
  - Relative complexity $C_r = 5.04/5 = 1.00$
- More I/O variables in a module, the more functionality needs to be accomplished by the module, therefore, the higher internal complexity
- More fan-out means that functionality is deferred to modules of lower levels, therefore the internal complexity of a module is reduced
- Overall data complexity is defined as the average of data complexity of all new modules

Reference

- Your text book by Pressman (Chapter 17)
- Everett and Raymond: Software Testing. Wilkey and IEEE CS Press
- Black: Pragmatic Software Testing. Wiley.
- Perry: Effectice Methods of Software Testing, Wiley.
- Lee, and Yannakakis, "Principles and Methods of Testing Finite State Machines: A Survey", Proceedings of The IEEE, Vol. 84, No. 8, August 1996,
- Chow, "Testing Software Design Modeled by Finite-State Machines" IEEE Transactions on Software Engineering, vol.4, no. 3, pp. 178-187, May 1978