

# Software Testing




# Organization of this Lecture



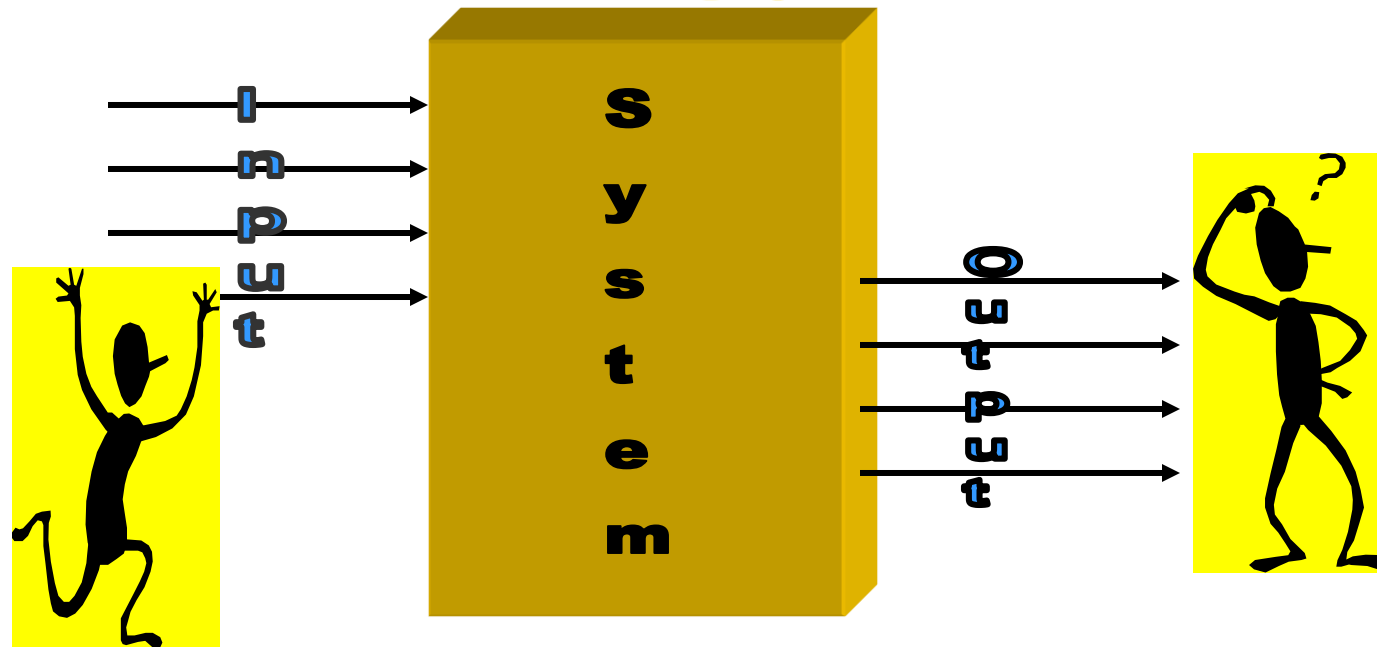
- Introduction to Testing.
- White-box testing:
  - statement coverage
  - path coverage
  - branch testing
  - condition coverage
  - Cyclomatic complexity
- Summary

# How do you test a system?



- Input test data to the system.
- Observe the output:
  - Check if the system behaved as expected.

# How do you test a system?



# How do you test a system?

- If the program does not behave as expected:
  - note the conditions under which it failed.
  - later debug and correct.

# Errors and Failures

- A failure is a manifestation of an error (aka defect or bug).
- mere presence of an error may not lead to a failure.

# Test cases and Test suite

□ Test a software using a set of carefully designed test cases:

□ the set of all test cases is called the test suite

# Test cases and Test suite

- A test case is a triplet  $[I, S, O]$ :
  - I is the data to be input to the system,
  - S is the state of the system at which the data is input (like edit mode, view mode, create new or display data etc)
  - O is the expected output from the system.



# Verification versus Validation



□ Verification is the process of determining:

□ whether output of one phase of development conforms to its previous phase.

□ Validation is the process of determining

□ whether a fully developed system conforms to its SRS document.

# Verification versus Validation



- Aim of Verification:

- phase containment of errors

- Aim of validation:

- final product is error free.

# Verification versus Validation



□ Verification:

□ are we doing right?

□ Validation:

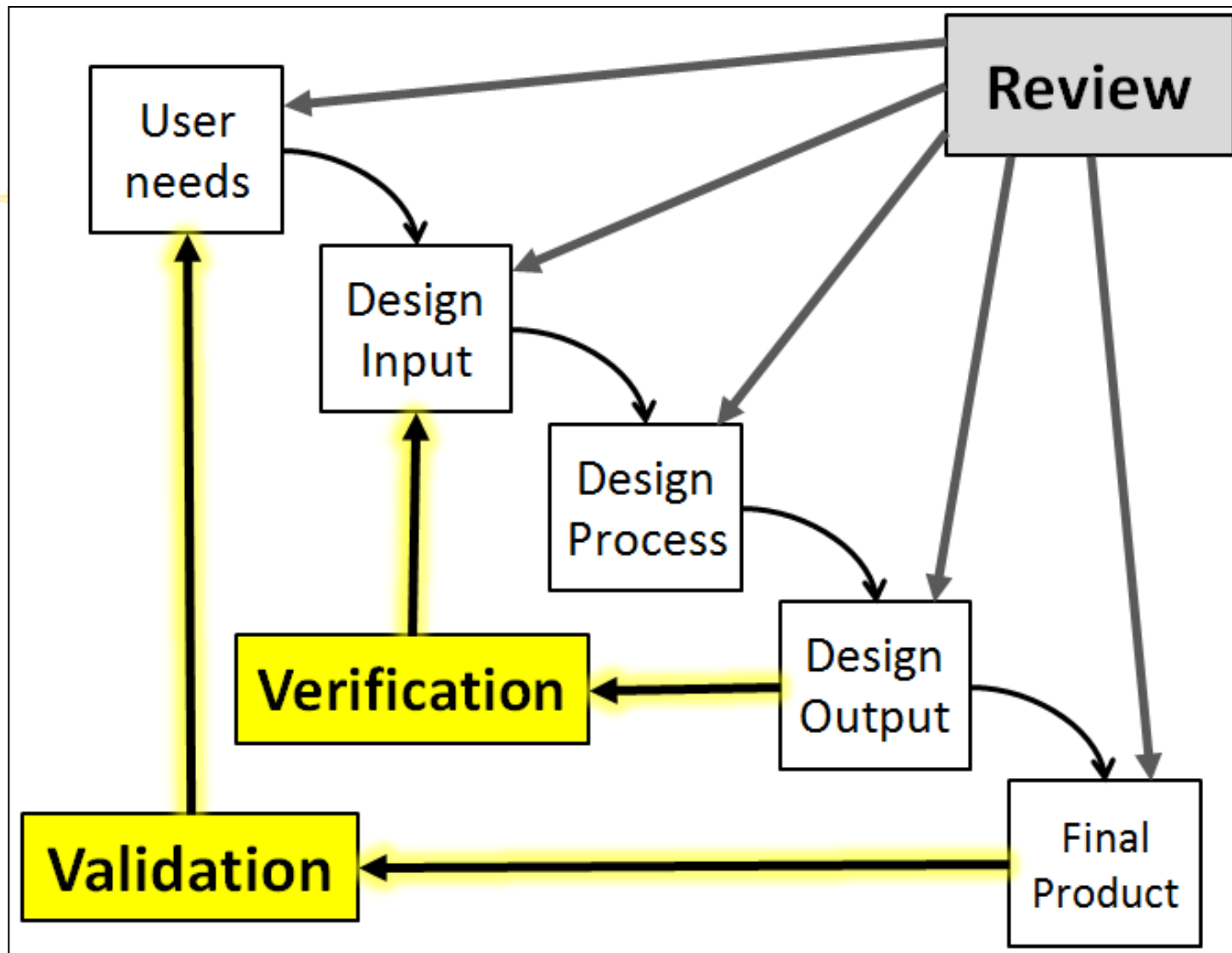
□ have we done right?

Verification	Validation
Are we building the system right?	Are we building the right system?
<b>Verification</b> is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	<b>Validation</b> is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in <b>Verification</b> : Reviews, Meetings and Inspections.	Following activities are involved in <b>Validation</b> : Testing like black box testing, white box testing, gray box testing etc.
<b>Verification</b> is carried out by QA team to check whether implementation software is as per specification document or not.	<b>Validation</b> is carried out by testing team.

# Verification

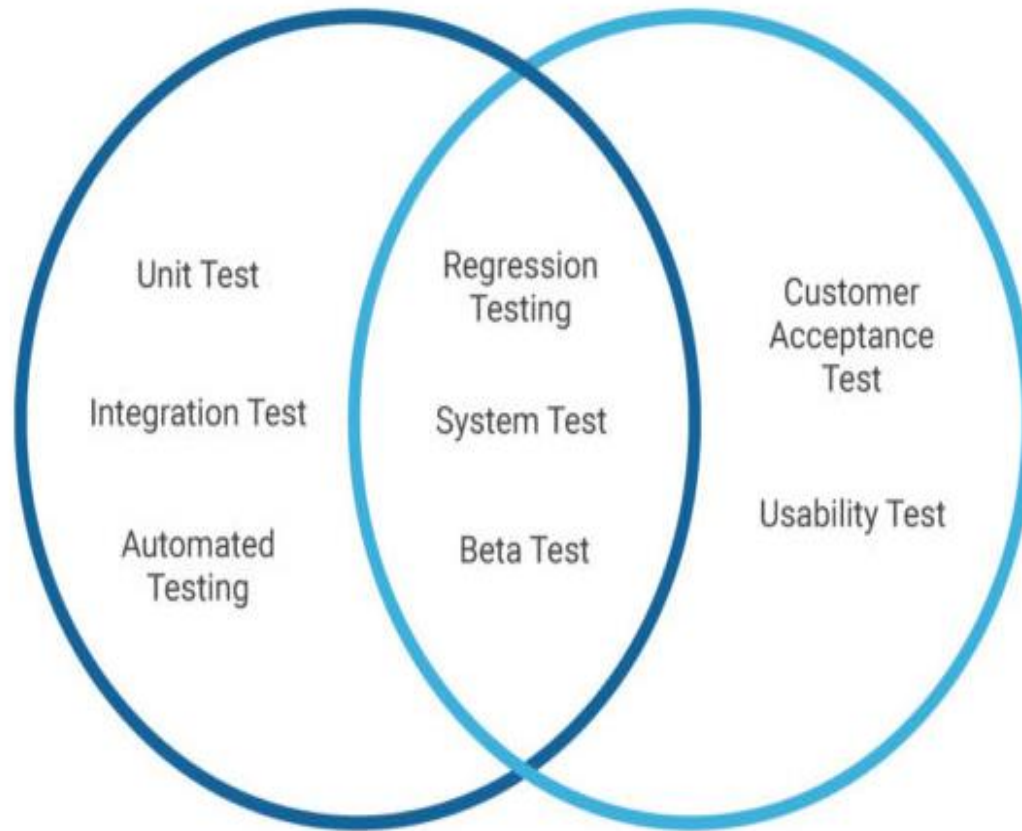
# Validation

Execution of code is not comes under <b>Verification</b> .	Execution of code is comes under <b>Validation</b> .
<b>Verification</b> process explains whether the outputs are according to inputs or not.	<b>Validation</b> process describes whether the software is accepted by the user or not.
<b>Verification</b> is carried out before the Validation.	<b>Validation</b> activity is carried out just after the Verification.
Following items are evaluated during <b>Verification</b> : Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during <b>Validation</b> : Actual product or Software under test.
Cost of errors caught in <b>Verification</b> is less than errors found in Validation.	Cost of errors caught in <b>Validation</b> is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.



## VERIFICATION

Am I building  
the product right?



## VALIDATION

Am I building the  
right product?

# Design of Test Cases

- Exhaustive testing of any non-trivial system is impractical:
  - input data domain is extremely large.
- Design an **optimal test suite**:
  - of reasonable size
  - to uncover as many errors as possible.



# Design of Test Cases

- If test cases are selected randomly:
  - many test cases do not contribute to the significance of the test suite,
  - do not detect errors not already detected by other test cases in the suite.
- The number of test cases in a randomly selected test suite:
  - not an indication of the effectiveness of the testing.

# Design of Test Cases



- Testing a system using a large number of randomly selected test cases:
  - does not mean that many errors in the system will be uncovered.
- Consider an example:
  - finding the maximum of two integers  $x$  and  $y$ .

# Design of Test Cases

- ```
If (x>y) max = x;  
    else max = x;
```
- The code has a simple error:
- test suite  $\{(x=3, y=2); (x=2, y=3)\}$   
can detect the error,
- a larger test suite  
 $\{(x=3, y=2); (x=4, y=3); (x=5, y=1)\}$   
does not detect the error.

# Design of Test Cases

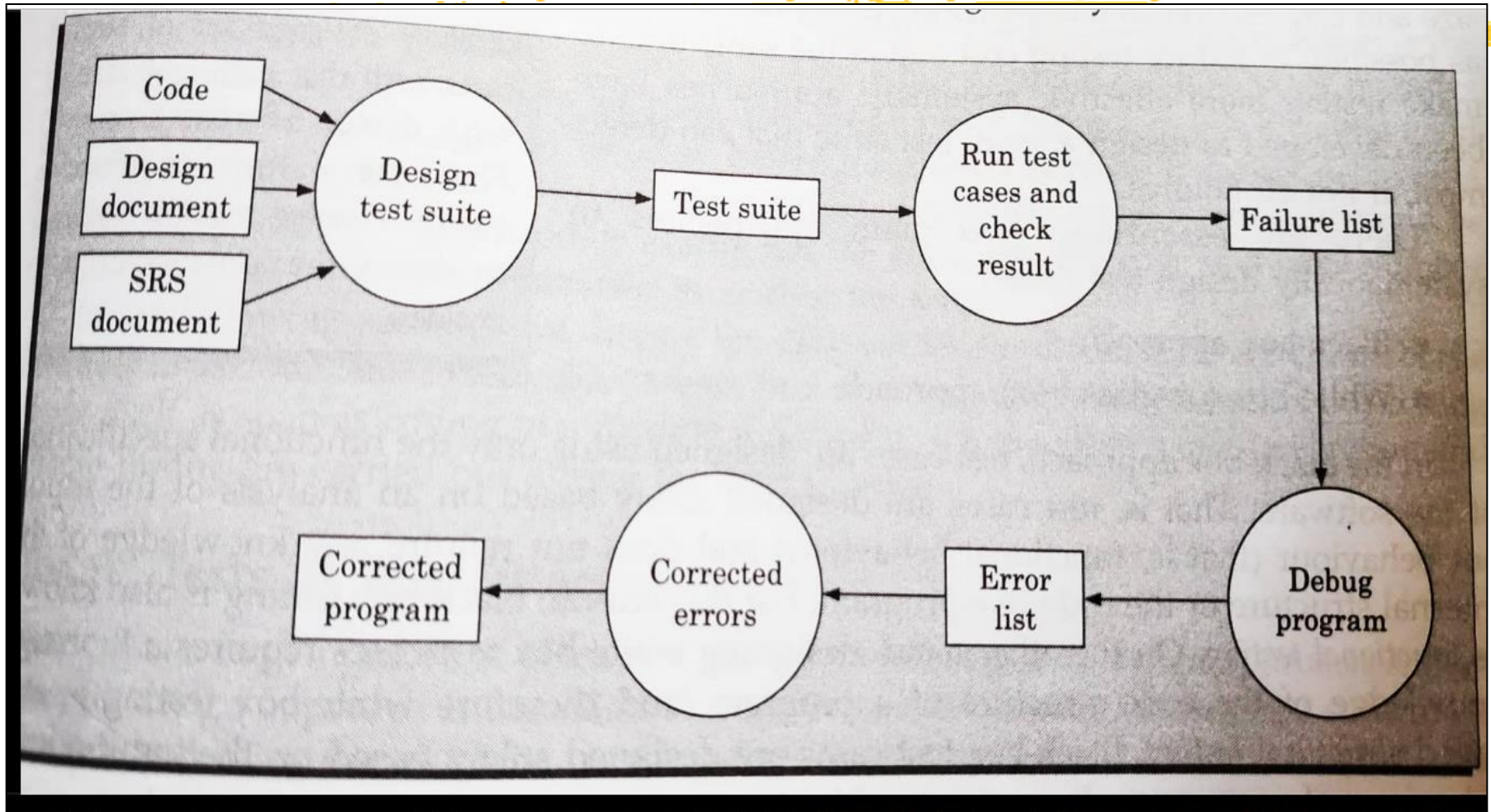


- Systematic approaches are required to design an optimal test suite:
  - each test case in the suite should detect different errors.

# Design of Test Cases

- Two main approaches to design test cases:
  - Black-box approach
  - White-box (or glass-box) approach

# Testing process



# Black-box Testing

- Test cases are designed using only functional specification of the software:
  - without any knowledge of the internal structure of the software.
- For this reason, black-box testing is also known as functional testing.

# White-box Testing

- Designing white-box test cases:
  - requires knowledge about the internal structure of software.
  - white-box testing is also called structural testing.



# Black-box Testing



- Two main approaches to design black box test cases:
  - Equivalence class partitioning
  - Boundary value analysis

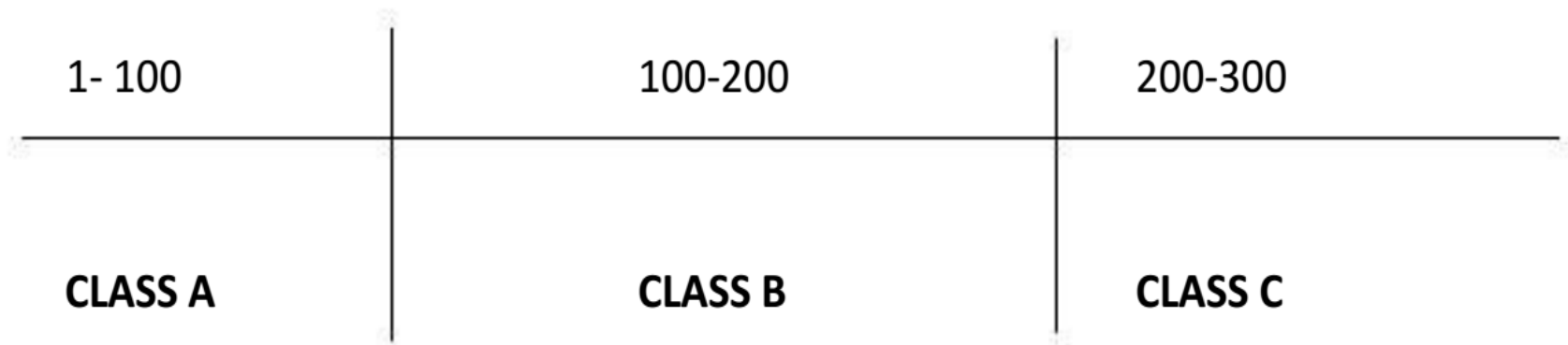
# Equivalence class partitioning



- Equivalence partitioning or equivalence class partitioning is a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once.
- The main idea is that testing the code with any one value belonging to an equivalence class is as good as testing the code with any other value in that class.

- In this method the input domain data is divided into different equivalence data classes. This method is typically used **to reduce the total number of test cases** to a finite set of testable test cases, still covering maximum requirements.
- In short it is the process of taking all possible test cases and placing them into classes. One test value is picked from each class while testing.

- Suppose the input domain for a function is **1 to 300**.
- It is impossible to create 300 test cases .
- So we divide the input variables into classes.
- Then we select one value from each class.
- This one value will represent whole of the class



# ECP - Guidelines



Rule 1 : For a range, 1 valid and 2 invalid inputs sets.

Example : Equivalence class is a set of integers between 1 and 10.

The classes

1. Infinity to 0 (invalid)
2. 1 to 10 (valid)
3. 11 to Infinity (invalid)

Rule 2 : For discrete values.

Example : Valid equivalence classes are {A, B, C}

Invalid class would be All possible inputs - {A, B, C}

**PROBLEM 10.8** Consider a program unit that takes an input integer that can assume values in the range of 0 and 5000 and computes its square root. Determine the equivalence classes and the black box test suite for the program unit.

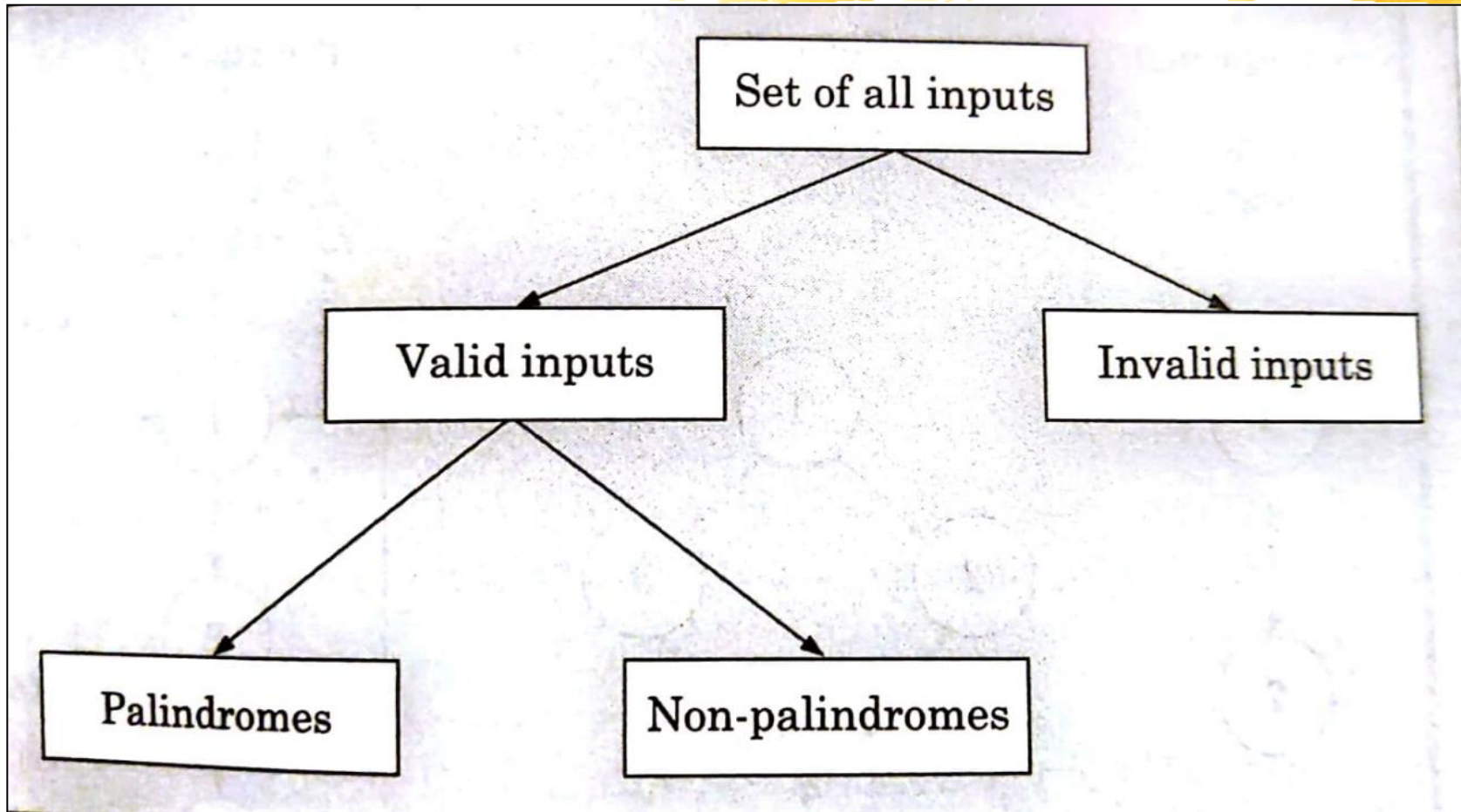
**Solution:** There are three equivalence classes—The set of negative integers, the set of integers in the range of 0 and 5000, and the set of integers larger than 5000. Therefore, the test cases must include representatives for each of the three equivalence classes. A possible test suite can be:  $\{-5, 500, 6000\}$ .

**PROBLEM 10.10** Design equivalence class partitioning test suite for a function that reads a character string of size less than five characters and displays whether it is a palindrome.

**Solution:** The domain of all input values can be partitioned into two broad equivalence classes: valid values and invalid values. The valid values can be partitioned into palindromes and non-palindromes. The equivalence classes are the leaf level classes shown in Figure 10.4. The equivalence classes are palindromes, non-palindromes, and invalid inputs. Now, selecting one representative value from each equivalence class, we have the required test suite:  $\{abc, aba, abcdef\}$ .



# ECP - Palindrome



# Boundary value analysis




- Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values in a range.

The idea comes from the boundary.

- The test cases use the values at the boundaries of different equivalence class.



- 
- Boundary value analysis focuses on the boundary of the input space to identify test cases.
  - The rationale behind boundary value analysis is that errors tend to occur near the extreme values of an input variable.

- This is because programmer may have used
- $\leq$  instead of  $<$  or conversely
- $<=$  for  $<$  etc.

# BVA : Guidelines

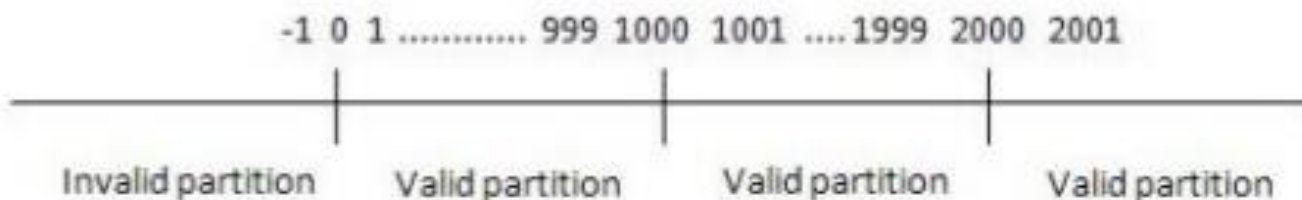
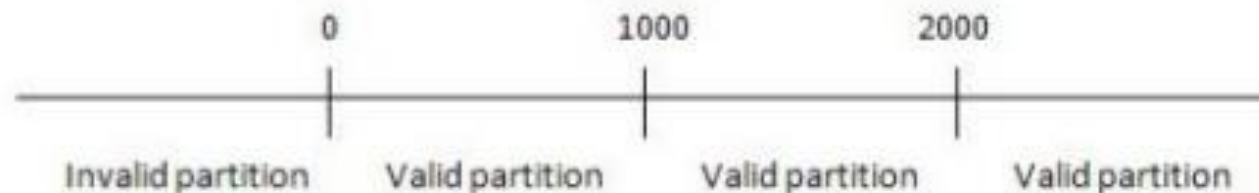


- Determine equivalence class.
- Take boundaries.
- For a set of integers 1 to 10, test set for BVA is {0, 1, 10, 11}).
- Values before and after both the boundaries.

# Boundary value analyze

**Example:** bank has different charges depending on the transaction done.

- 5% of the amount for transaction less than or equal to 1000
- 6% of the amount for transaction more than 1000 and less than or equal to 2000
- 7% of the amount for transaction more than 2000



# White-Box Testing



- There exist several popular white-box testing methodologies:
  - Statement coverage
  - branch coverage
  - path coverage
  - condition coverage
  - mutation testing
  - data flow-based testing

# Statement Coverage

- Statement coverage methodology:
  - design test cases so that
    - every statement in a program(source code) is executed at least once.

# Statement Coverage



- The principal idea:
  - unless a statement is executed,
  - we have no way of knowing if an error exists in that statement.

# Statement coverage criterion



- Based on the observation:
  - an error in a program can not be discovered:
    - unless the part of the program containing the error is executed.

# Statement coverage criterion



- Observing that a statement behaves properly for one input value:
  - no guarantee that it will behave correctly for all input values.



# Example



```
□ int f1(int x, int y){  
□ 1 while (x != y){  
□ 2   if (x>y) then Euclid's GCD Algorithm  
□ 3       x=x-y;  
□ 4   else y=y-x;  
□ 5 }  
□ 6 return x;      }
```

# Euclid's GCD computation algorithm



- By choosing the test set  $\{(x=3, y=3), (x=4, y=3), (x=3, y=4)\}$
- all statements are executed at least once.

# Switch case on students grade

## PROGRAM CODE

```
#include<stdio.h>

int main()
{
    int total_mark,tm;
    printf("\nEnter total mark secured by a student: ");
    scanf("%d",&total_mark);
    tm=total_mark/10;
    switch(tm)
    {
        case 9: printf("\nSecured grade is O");
                break;
        case 8: printf("\nSecured grade is E");
                break;
        case 7: printf("\nSecured grade is A");
                break;
        case 6: printf("\nSecured grade is B");
                break;
        case 5: printf("\nSecured grade is C");
                break;
        case 4: printf("\nSecured grade is D");
                break;
        default: printf("FAIL");
    }
    return 0;
}
```

# Branch Coverage



- Test cases are designed such that:
  - different branch conditions
  - given true and false values in turn.

# Branch Coverage



- Branch testing guarantees statement coverage:
  - a stronger testing compared to the statement coverage-based testing.

# Stronger testing

- Test cases are a superset of a weaker testing:
  - discovers at least as many errors as a weaker testing
  - contains at least as many significant test cases as a weaker test.

# Example



```
□ int f1(int x,int y){  
□ 1 while (x != y){  
□ 2     if (x>y) then  
□ 3         x=x-y;  
□ 4     else y=y-x;  
□ 5 }  
□ 6 return x;      }
```

# Example



- Test cases for branch coverage can be:
- $\{(x=3, y=3), (x=3, y=2), (x=4, y=3), (x=3, y=4)\}$



# Branch coverage is stronger than statement coverage

**Theorem 10.1** Branch coverage-based testing is stronger than statement coverage-based testing.

*Proof:* We need to show that (a) branch coverage ensures statement coverage, and (b) statement coverage does not ensure branch coverage.

- (a) Branch testing would guarantee statement coverage since every statement must belong to some branch (assuming that there is no unreachable code).
- (b) To show that statement coverage does not ensure branch coverage, it is sufficient to give an example of a test suite that achieves statement coverage, but does not cover at least one branch. Consider the following code, and the test suite {5}.

`if (x > 2) x += 1;`

The test suite would achieve statement coverage. However, it does not achieve branch coverage, since the condition  $(x > 2)$  is not made false by any test case in the suite.

# Condition Coverage



- Test cases are designed such that:
  - each component of a composite conditional expression
    - given both true and false values.

# Example



□ Consider the conditional expression

□ **`((c1.and.c2).or.c3):`**

**BCC = Basic Condition Coverage**

Each of `c1`, `c2`, and `c3` are exercised at least once,

i.e. given true and false values.

**MCC = Multiple Condition Coverage**

Each of `c1`, `c2`, and `c3` are exercised with all possible combination of true and false values.

# Example

```
if ((A || B) && C)
{
    << Few Statements >>
}
else
{
    << Few Statements >>
}
```

## Result

In order to ensure complete Condition coverage criteria for the above example, A, B and C should be evaluated at least once against "true" and "false".

So, in our example, the 3 following tests would be sufficient for 100% Condition coverage

A = true | B = not eval | C = false

A = false | B = true | C = true

A = false | B = false | C = not eval

# Branch testing

- Branch testing is the simplest condition testing strategy:
  - compound conditions appearing in different branch statements
    - are given true and false values.

# Comparision



- *Condition testing (MCC)*
  - *stronger testing than branch testing:*
- *Branch testing*
  - *stronger than statement coverage testing.*

# Condition coverage



- Consider a boolean expression having  $n$  components:
  - for condition coverage **we require  $2^n$  test cases.**

# Condition coverage



- Condition coverage-based testing technique:
  - practical only if  $n$  (the number of component conditions) is small.



# Path Coverage



- Design test cases such that:
  - all **linearly independent paths** in the program are executed at least once.

# Linearly independent paths



- Defined in terms of
  - control flow graph (CFG) of a program.

# Path coverage-based testing



- To understand the path coverage-based testing:
  - we need to learn how to draw control flow graph of a program.

# Control flow graph (CFG)



- A control flow graph (CFG) describes:
  - the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

# How to draw Control flow graph?



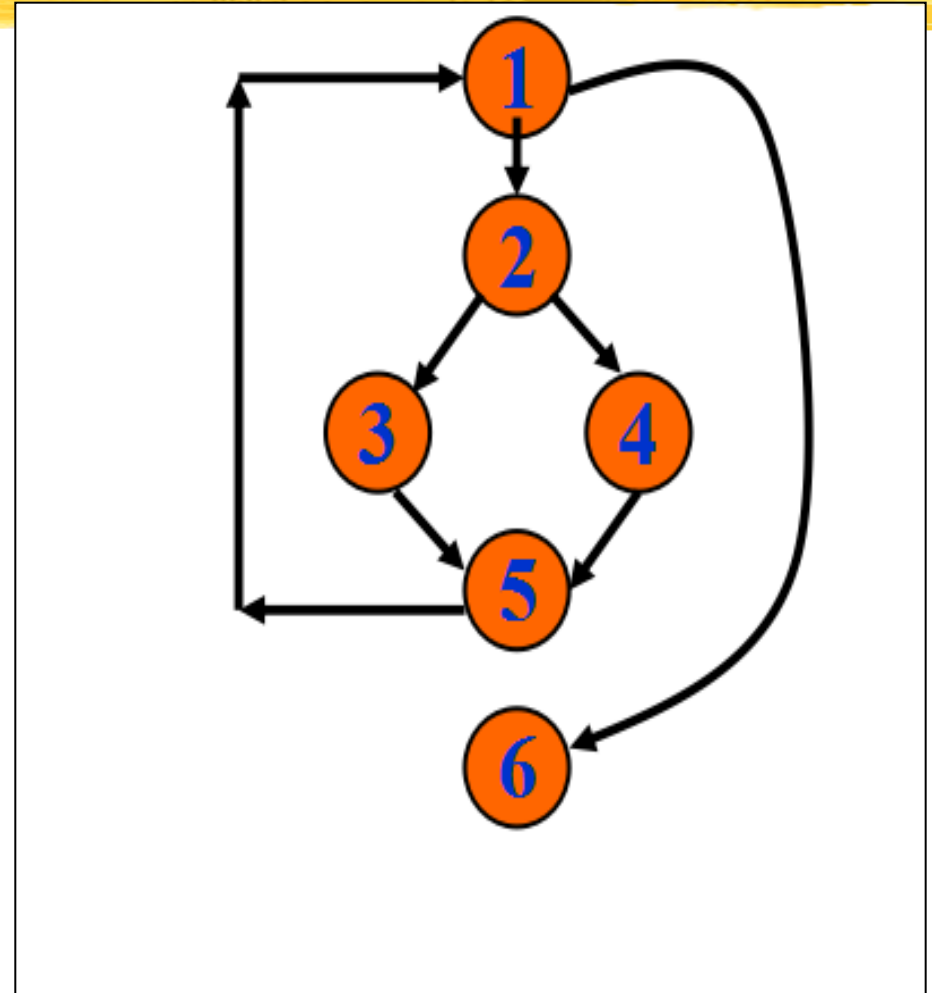
- Number all the statements of a program.
- Numbered statements:
  - represent nodes of the control flow graph.

# How to draw Control flow graph?

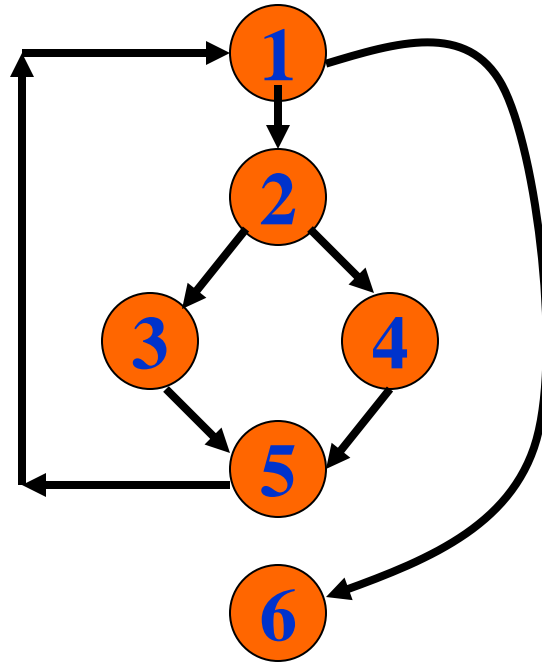
- An edge from one node to another node exists:
  - if execution of the statement representing the first node
    - can result in transfer of control to the other node.

# Example

```
□ int f1(int x,int y){  
□ 1 while (x != y){  
□ 2   if (x>y) then  
□ 3     x=x-y;  
□ 4   else y=y-x;  
□ 5 }  
□ 6 return x;    }
```



# Example Control Flow Graph



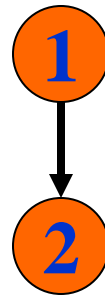


# How to draw Control flow graph?

## 1. Sequence:

□1 a=5;

□2 b=a\*b-1;



# How to draw Control flow graph?

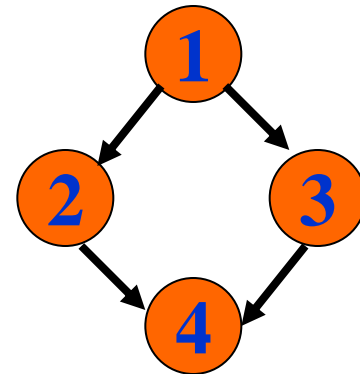
## 2. Selection:

□ 1 if( $a > b$ ) then

□ 2             $c = 3;$

□ 3 else     $c = 5;$

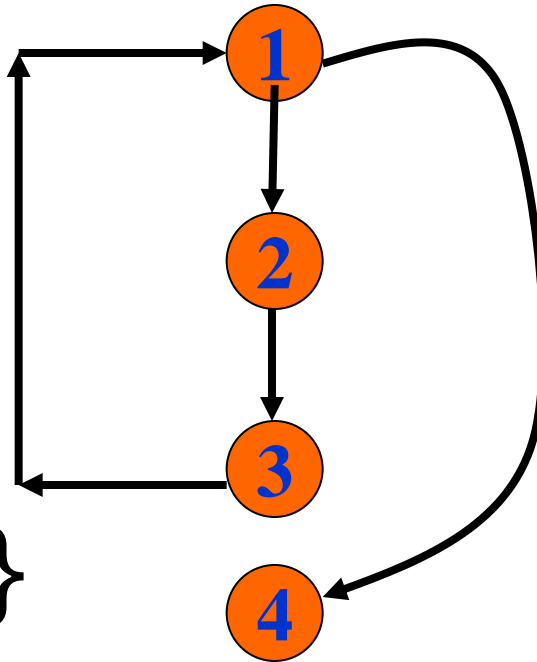
□ 4  $c = c * c;$



# How to draw Control flow graph?

## 3. Iteration:

```
□1 while(a>b){  
□2     b=b*a;  
□3     b=b-1;}  
□4 c=b+d;
```



# Path



- A path through a program:
  - a node and edge sequence from the starting node to a terminal node of the control flow graph.
  - There may be several terminal nodes for program.

# Independent path

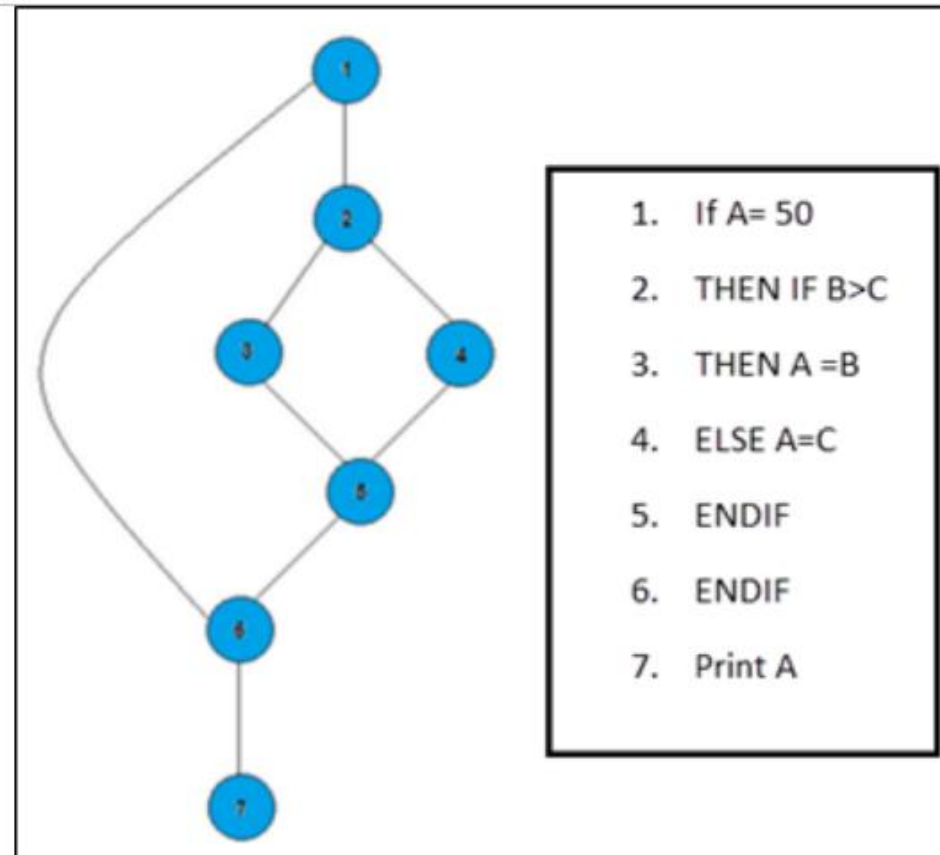


- Any path through the program:
  - introducing at least one new node:
    - that is not included in any other independent paths.

# Independent path



- It is straight forward:
  - to identify linearly independent paths of simple programs.
- For complicated programs:
  - it is not so easy to determine the number of independent paths.



In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 path or condition that need to be tested to get the output,

- **Path 1:** 1,2,3,5,6, 7
- **Path 2:** 1,2,4,5,6, 7
- **Path 3:** 1, 6, 7

# McCabe's cyclomatic metric



- An upper bound:
  - for the number of linearly independent paths of a program
- Provides a practical way of determining:
  - the maximum number of linearly independent paths in a program.

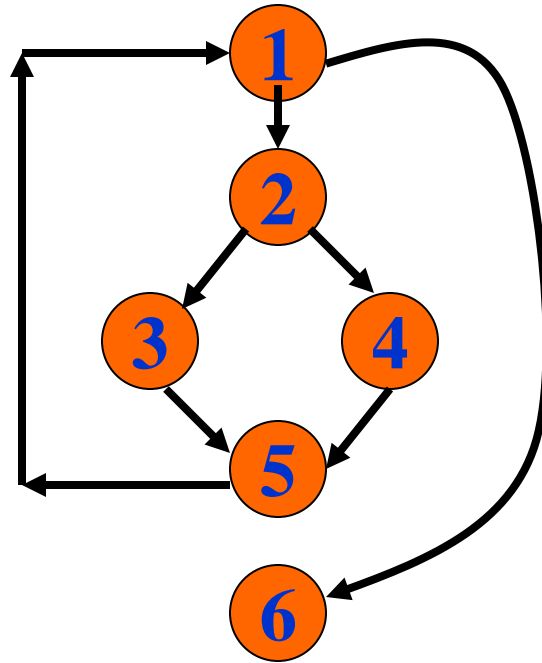


# McCabe's cyclomatic metric



- Given a control flow graph  $G$ , cyclomatic complexity  $V(G)$ :
  - **$V(G) = E - N + 2$** 
    - $N$  is the number of nodes in  $G$  (shapes)
    - $E$  is the number of edges in  $G$  (lines)

# Example Control Flow Graph



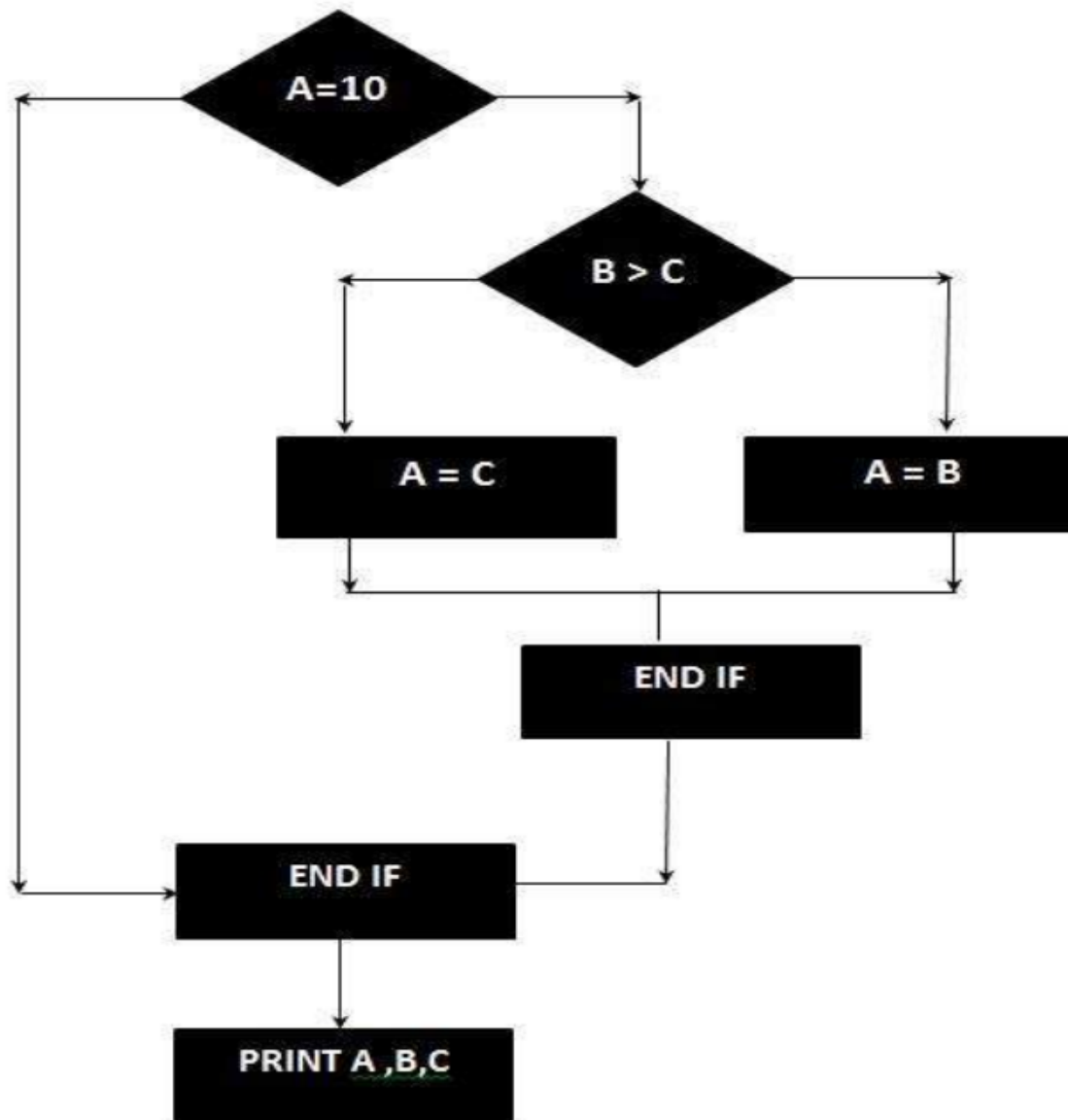
Cyclomatic complexity =  
**7 (lines) - 6 (circles) + 2 = 3.**

# Example 2



```
IF A = 10 THEN
    IF B > C THEN
        A = B
    ELSE
        A = C
    ENDIF
ENDIF
Print A
Print B
Print C
```

## FlowGraph:



**IF A = 10**  
**THEN**  
    **IF B > C**  
        **THEN**  
            **A = B**  
        **ELSE**  
            **A = C**  
        **ENDIF**  
    **ENDIF**  
    **Print A**  
    **Print B**  
    **Print C**  
**8 - 7 + 2 =**  
**3**

# Cyclomatic complexity

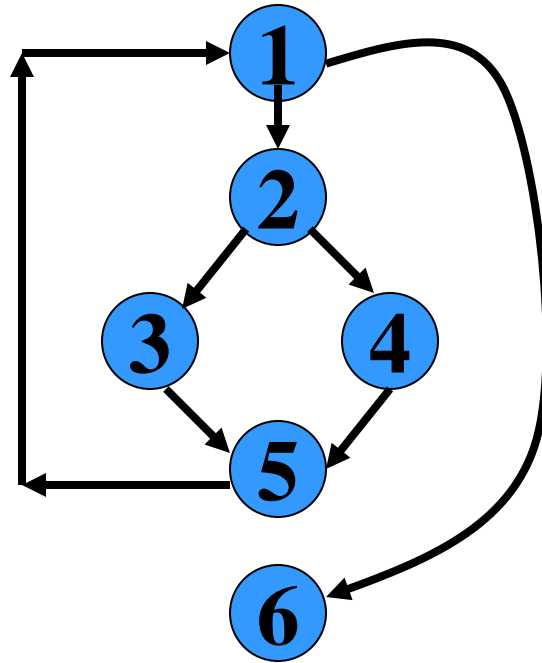
- Another way of computing cyclomatic complexity:
  - inspect control flow graph
  - determine number of bounded areas in the graph
- $V(G) = \text{Total number of bounded areas} + 1$

# Bounded area



- Any region enclosed by a nodes and edge sequence.

# Example Control Flow Graph



# Example



- From a visual examination of the CFG:
  - the number of bounded areas is 2.
  - cyclomatic complexity =  $2+1=3$ .



# Cyclomatic complexity



- McCabe's metric provides:
  - a quantitative measure of testing difficulty and the ultimate reliability
- Intuitively,
  - number of bounded areas increases with the number of decision nodes and loops.

# Cyclomatic complexity



- The first method of computing  $V(G)$  is amenable to automation:
  - you can write a program which determines the number of nodes and edges of a graph
  - applies the formula to find  $V(G)$ .

# Cyclomatic complexity



- The cyclomatic complexity of a program provides:
  - a lower bound on the number of test cases to be designed
  - to guarantee coverage of all linearly independent paths.

# Cyclomatic complexity



- Defines the number of independent paths in a program.
- Provides a lower bound:
  - for the number of test cases for path coverage.

# Cyclomatic complexity



- Knowing the number of test cases required:
  - does not make it any easier to derive the test cases,
  - only gives an indication of the minimum number of test cases required.

# Path testing



- The tester proposes:
  - an initial set of test data using his experience and judgement.

# Path testing



- A dynamic program analyzer is used:
  - to indicate which parts of the program have been tested
  - the output of the dynamic analysis
    - used to guide the tester in selecting additional test cases.

# Derivation of Test Cases



- Let us discuss the steps:
  - to derive path coverage-based test cases of a program.



# Derivation of Test Cases



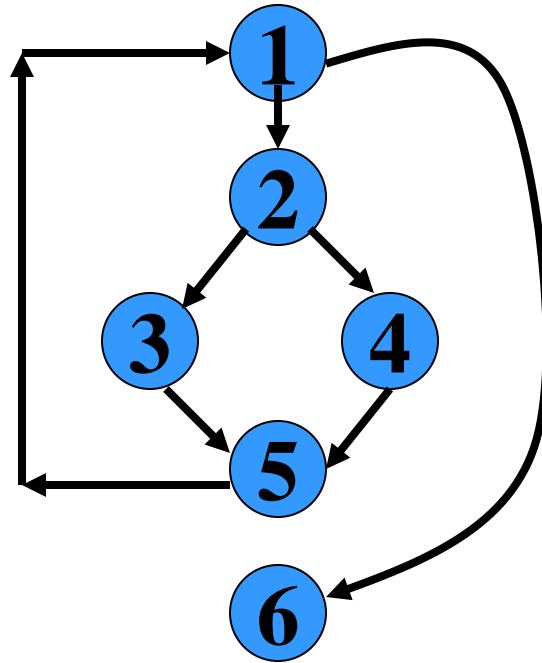
- Draw control flow graph.
- Determine  $V(G)$ .
- Determine the set of linearly independent paths.
- Prepare test cases:
  - to force execution along each path.

# Example



```
□ int f1(int x,int y){  
□ 1 while (x != y){  
□ 2     if (x>y) then  
□ 3         x=x-y;  
□ 4     else y=y-x;  
□ 5 }  
□ 6 return x;      }
```

# Example Control Flow Diagram



# Derivation of Test Cases

□ Number of independent paths:  
3

□ 1,6 test case (x=1, y=1)

□ 1,2,3,5,1,6 test case(x=10,  
y=2)

□ 1,2,4,5,1,6 test case(x=2,  
y=10)

# **An interesting application of cyclomatic complexity**



- Relationship exists between:
  - McCabe's metric
  - the number of errors existing in the code,
  - the time required to find and correct the errors.

# Cyclomatic complexity



- Cyclomatic complexity of a program:
  - also indicates the psychological complexity of a program.
  - difficulty level of understanding the program.

# Cyclomatic complexity



- From maintenance perspective,
  - limit cyclomatic complexity
    - of modules to some reasonable value.
- Good software development organizations:
  - restrict cyclomatic complexity of functions to a maximum of ten or so.

# Testing Process - Validation Order after coding

- Unit testing (UT) - Individual modules
- Integration Testing (IT) - Integrate every module in the desired hierarchy.
- - Top down , bottom up
- System testing (ST) - Try to map the application features developed in accordance with SRS.
- User Acceptance Testing (UAT)



# Summary



- Exhaustive testing of non-trivial systems is impractical:
  - we need to design an optimal set of test cases
    - should expose as many errors as possible.

# Summary



- If we select test cases randomly:
  - many of the selected test cases do not add to the significance of the test set.

# Summary



- There are two approaches to testing:
  - black-box testing and
  - white-box testing.

# Summary



- Designing test cases for black box testing:
  - does not require any knowledge of how the functions have been designed and implemented.
  - Test cases can be designed by examining only SRS document.

# Summary



- White box testing:
  - requires knowledge about internals of the software.
  - Design and code is required.

# Summary



- We have discussed a few white-box test strategies.
  - Statement coverage
  - branch coverage
  - condition coverage
  - path coverage

# Summary



- A stronger testing strategy:
  - provides more number of significant test cases than a weaker one.
  - Condition coverage is strongest among strategies we discussed.

# Summary



- We discussed McCabe's Cyclomatic complexity metric:
  - provides an upper bound for linearly independent paths
  - correlates with understanding, testing, and debugging difficulty of a program.