Software Engineering(SE)

CSC 601

Subject Incharge

Varsha Nagpurkar

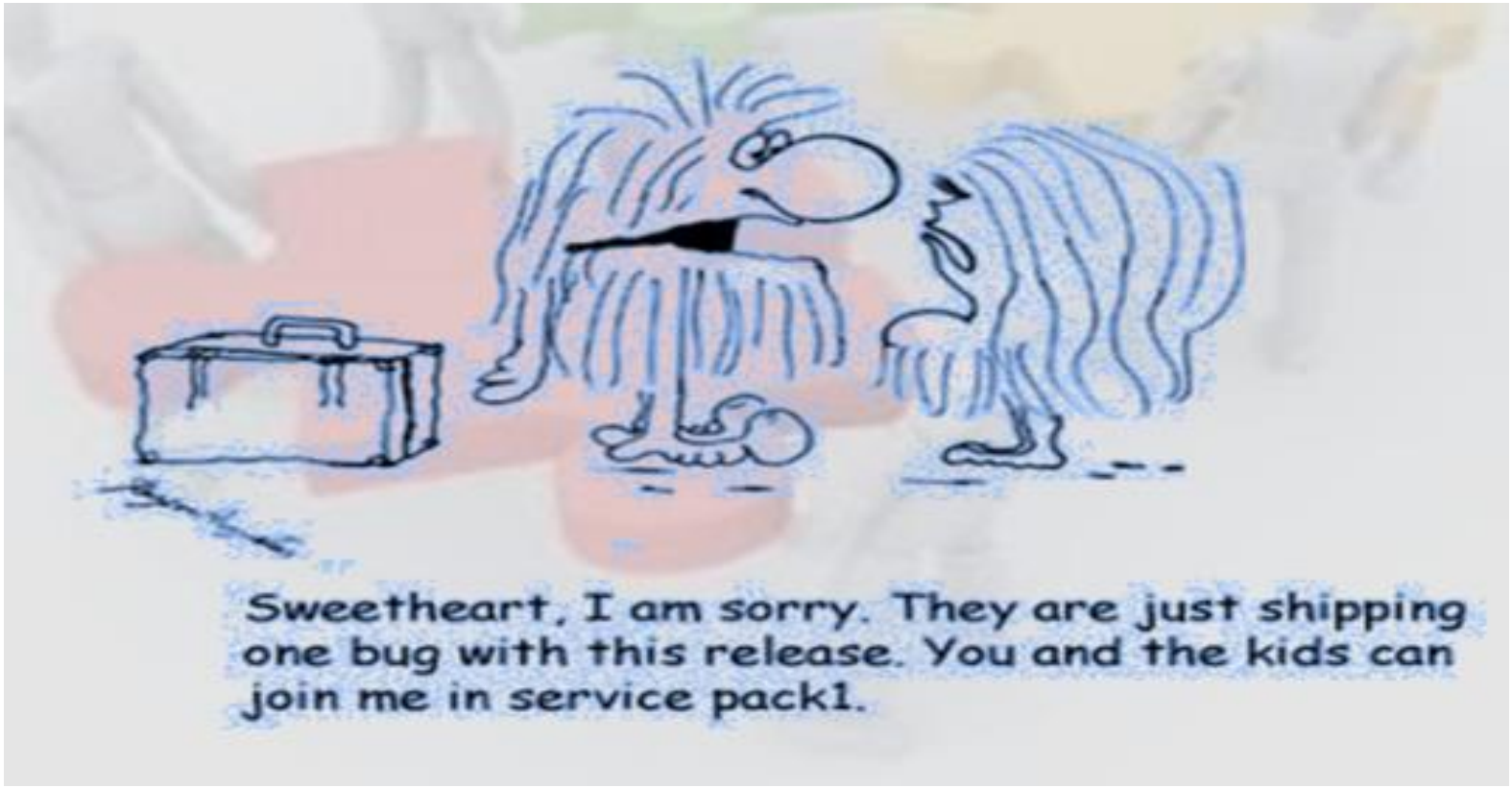
Assistant Professor

Room No. 407

email: varshanagpurkar@sfit.ac.in

CHAPTER 6

Software Testing



Module-6

6.0		Software Testing and Maintenance	10
	6.1	Strategic Approach to Software Testing, Unit testing, Integration testing Verification, Validation Testing, System Testing	
	6.2	Software Testing Fundamentals, White-Box Testing , Basis Path Testing, Control Structure Testing, Black-Box Testing,	
	6.3	Software maintenance and its types, Software Re-engineering, Reverse Engineering	

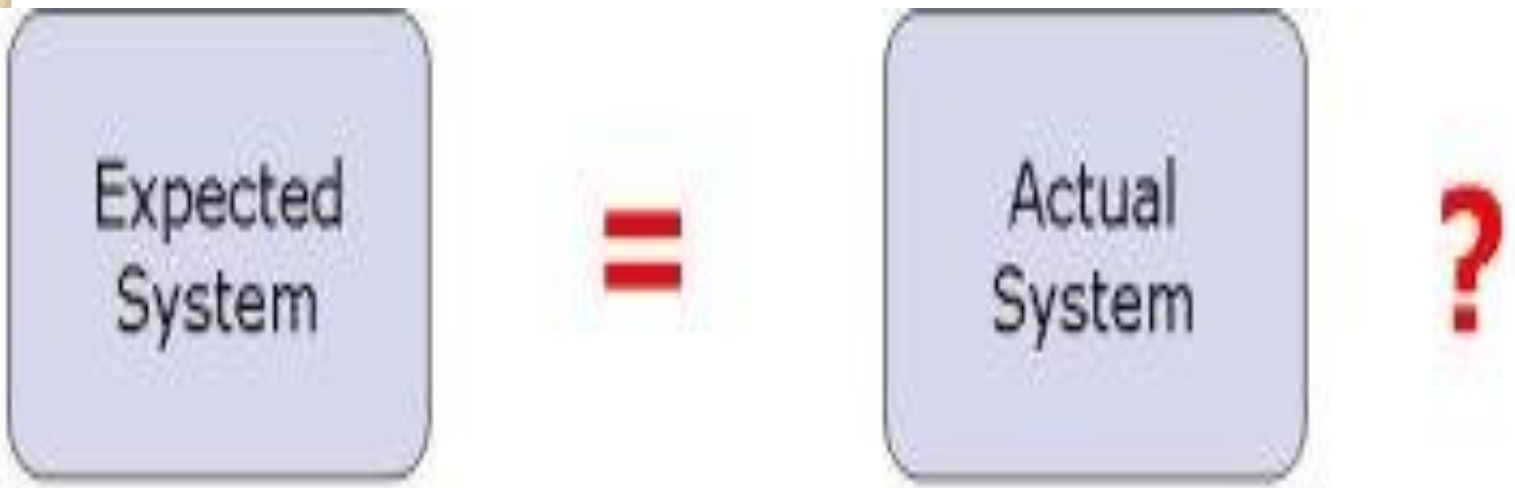
A Strategic approach to Software Testing

- Testing is a set of activities that can be planned in advance and conducted systematically
- For implementing testing ,there are various testing strategies defined in software engineering literature
- All these strategies provide a testing template to the developers
- The testing templates should possess following characteristics that is applicable to any software development process

Characteristics of Testing Template

- For effective testing,formal technical reviews must be conducted by the development team.Frequent communication between developer and customer resolves most of the complexities and confusion in the beginning itself.Thus before start of the testing process,number of errors may be uncovered and then fixed
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system
- Different testing techniques are appropriate at different points in time
- Testing is conducted by the developer of the software and (for large projects)an independent test group
- Testing and debugging are different activities,but debugging must be accomodated in any testing strategy

What's our target for Testing?



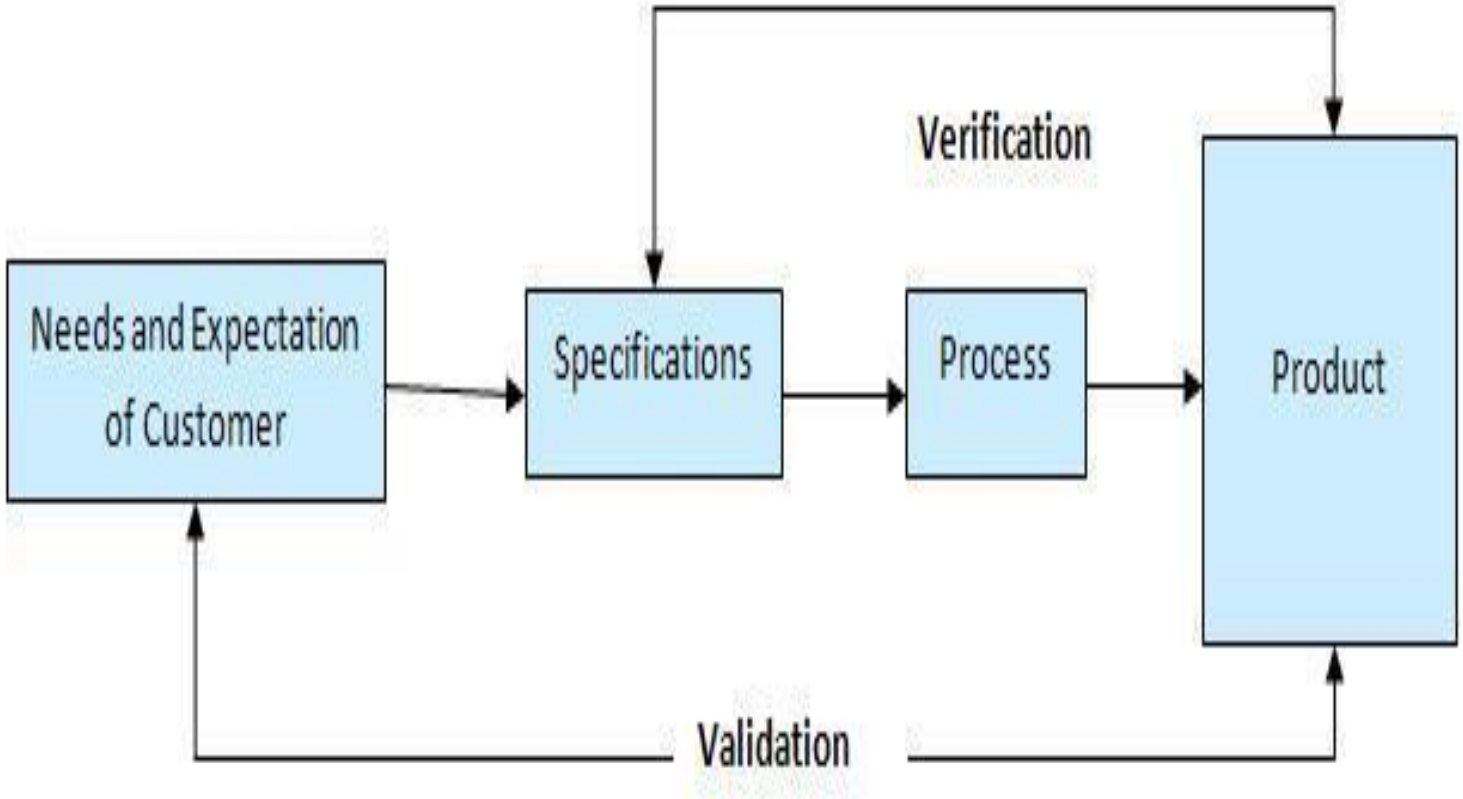
Are the expected behaviour (specified in requirement specification and system models) the same as the observed behaviour of the actual system?

V & V

Verification refers to the set of activities/tasks that ensure that software correctly implements a specific function.

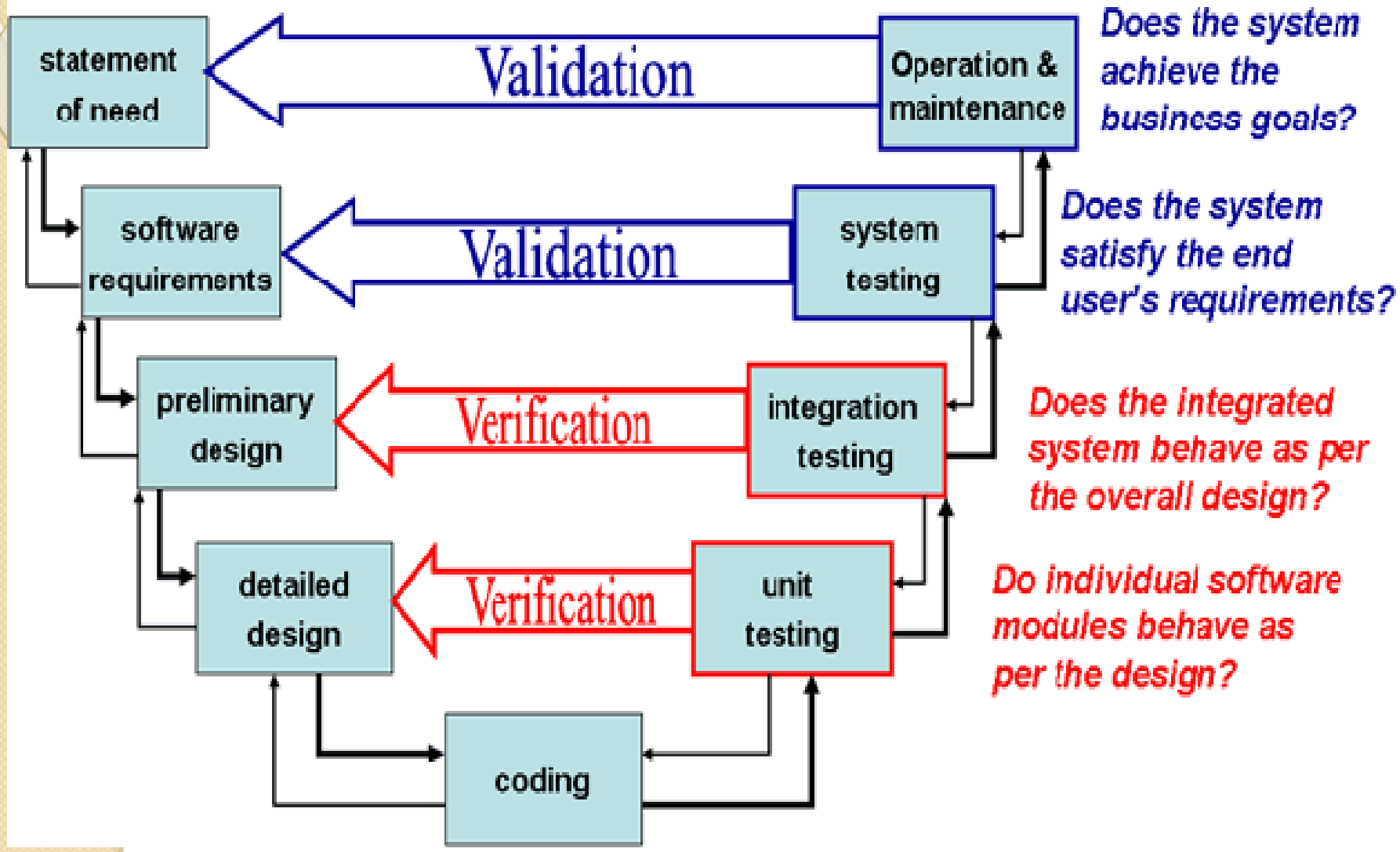
Validation refers to a different set of activities/tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

Verification: **"Are we building the product right?"**
Validation: **"Are we building the right product?"**


Verification
Needs and Expectation of Customer
Specifications
Process
Product
Validation

VERIFICATION	VALIDATION
1. Verification is the process of determining whether or not a product of given phase of software development fulfil the support specification.	2. Validation is the process of evaluating software at the end of software development process to determine whether it is in accordance with the software requirement.
2. It does not involve executing the code.	2. It always involves executing the code.
3. Verification evaluates each phase end product in order to ensure concurrency.	3. Validation test software specification in order to ensure its user requirements
4. Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	4. Validation uses methods like black box (functional) testing, glass box testing, and white box (structural) testing etc.
5. Verification is to check whether the software conforms to specifications.	5. Validation is to check whether software meets the customer expectations and requirements.
6. Verification ensure that each step in the process off software development yield the right product.	6. Validation ensures that the software being developed will satisfy functional and other requirements.
7. Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	7. Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
8. Verification process is done at early stage.	8. Validation process is done at Late (End) stage.
9. It generally comes first-done before validation.	9. It generally follows after verification.
10. Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	10. Validation is carried out with the involvement of testing team.
11. Verification is a static practice of verifying documents, design, code and program.	11. Validation is a dynamic mechanism of validating and testing the actual product.
12. It is human based checking of documents and files.	12. It is computer based execution of program.
13. It can catch errors that validation cannot catch. It is low level exercise.	13. It can catch errors that verification cannot catch. It is High Level Exercise.

Dynamic Testing



Validation

- Validation concentrates on the big picture of whether the software can do what the user wants.
- The focus is on seeing if the software is suitable for a ***"specific intended use or application"***.
- The Validation diagram shows that at whatever stage of development you are you need to check back to the System Specification, User Requirements and Business Case to see if it meets the purpose.
- The V-model diagram shows this also includes when tests are developed, because the user is interested in what the software does, and these are the most important tests for them.

So what qualities do we measure?

- Correctness (Functional Requirements)
- Reliability (Non-functional Requirements)
- Robustness
- Security
- Performance
- Usability

Unit Testing

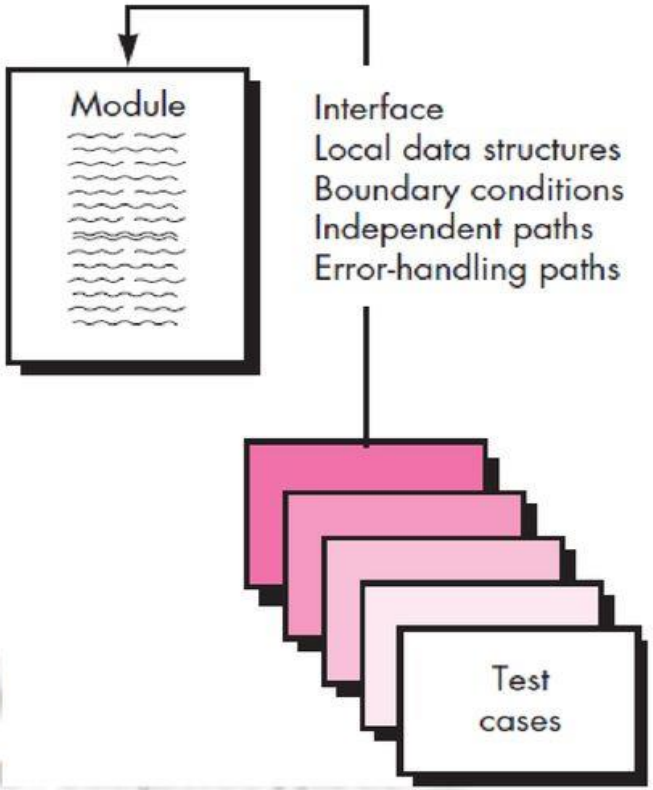
- It focuses verification effort on the smallest unit of software design-the software component or module
- Using the component-level design description as a guide,important control paths are tested to uncover errors within the boundary of the module
- It focuses on the internal processing logic and data structures within the boundaries of a component
- This type of testing can be conducted in parallel for multiple components

Unit test considerations

Test Strategies for Conventional Software

Unit Test Considerations

1. Module interface - - tested
2. Local data structures r examined
3. All independent paths thru control structure r exercised
4. Boundary conditions r tested
5. All error-handling paths r tested.



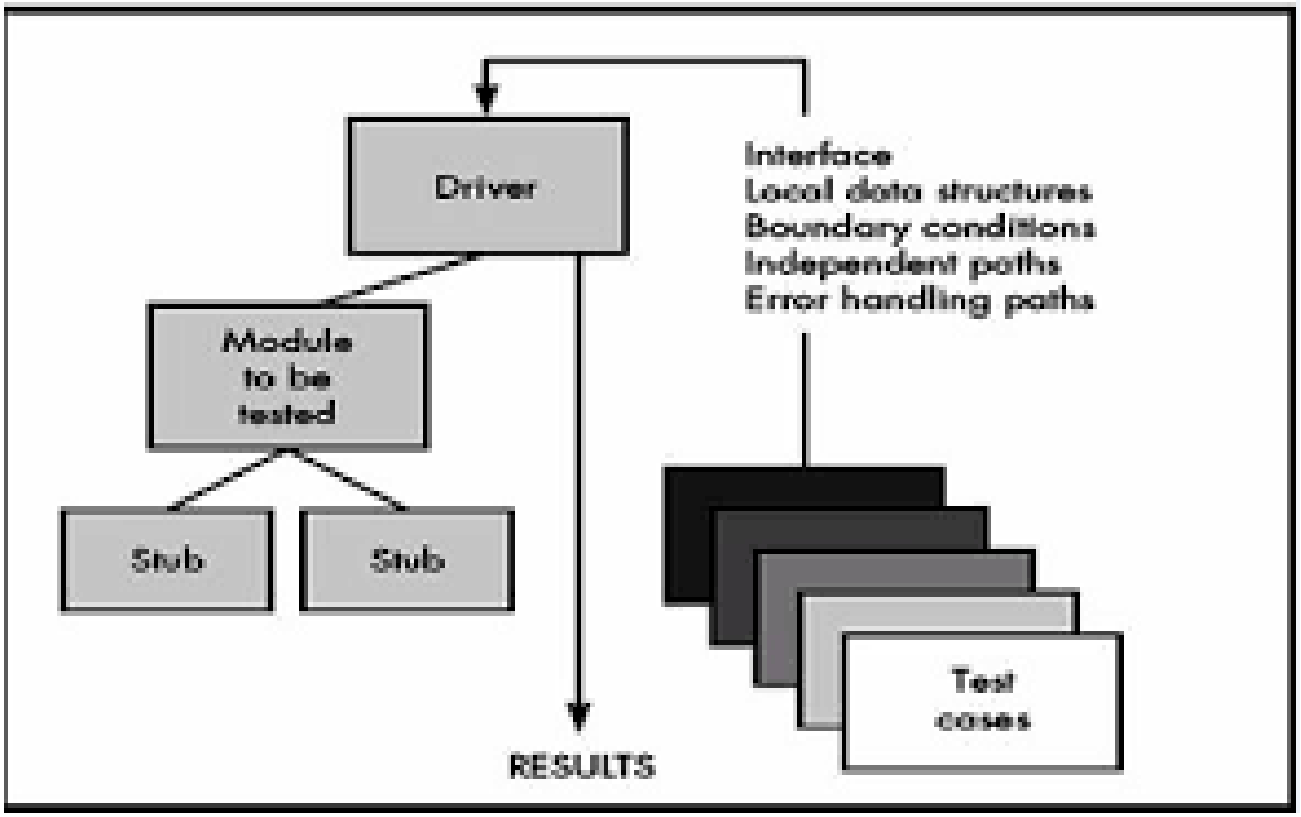
Unit test considerations

- The Module interface is tested to ensure that information properly flows into and out of the program unit under test
- Local data structures are examined to ensure its integrity in the execution of the algorithm
- All independent paths(basis paths) through the control structure are evaluated to ensure that all statements in a module have been executed at least once
- Boundary conditions are tested to ensure that the module operates properly within boundaries or within limit s
- All error handling paths are tested

Unit test procedures

- Unit testing is normally considered as an adjunct to the coding step
- A review of design information provides guidance for establishing test cases that are likely to uncover errors

Unit test environment



Unit test environment

- In most applications a driver is nothing but a “main program” that accepts test case data,passes such data to the component(to be tested),and prints relevant results
- Stubs serve to replace modules that are subordinate to(called by) the component to be tested

INTEGRATION TESTING

- It is a level of software testing where individual units / components are combined and tested as a group.
- The purpose of this level of testing is to expose faults in the interaction between integrated units.
- Test drivers and test stubs are used to assist in Integration Testing.

Integration Testing

- It is a systematic technique for constructing software architecture while at the same time conducting tests to uncover errors associated with interfacing
- The objective is to take unit tested components and build a program structure that has been dictated by design
- It is often observed that there is a tendency to attempt always non-incremental approaches.All the components are integrated in the beginning and the program is tested as a whole

Integration Testing

- In this approach a set of errors are encountered and correction seems to be very difficult due to isolation of causes and the complications by program since program is expanded over several modules
- After correction of these errors,some new may appear and the process continues,like an infinite loop
- Small increments of the program are tested in an incremental approach
- In incremental integration approach,the error detection and correction is quite simple

Different Incremental Integration Strategies

- Top-down integration
- Bottom-up integration

Top down Integration Testing Approach

- It is an incremental approach to construction of the software architecture
- Modules are integrated by moving downward through the control hierarchy,beginning with the main control module(main program)
- Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner

Depth-first Integration

- Depth-first integration integrates all components on a major control path of the program structure
- Selection of a major path is somewhat arbitrary and depends on application-specific characteristics
- For example,selecting the left hand path ,components M1,M2,M5 would be integrated first
- Next,M8 or(if necessary for proper functioning of M2)M6 would be integrated
- Then, the central and right-hand control paths are built

Breadth-first Integration

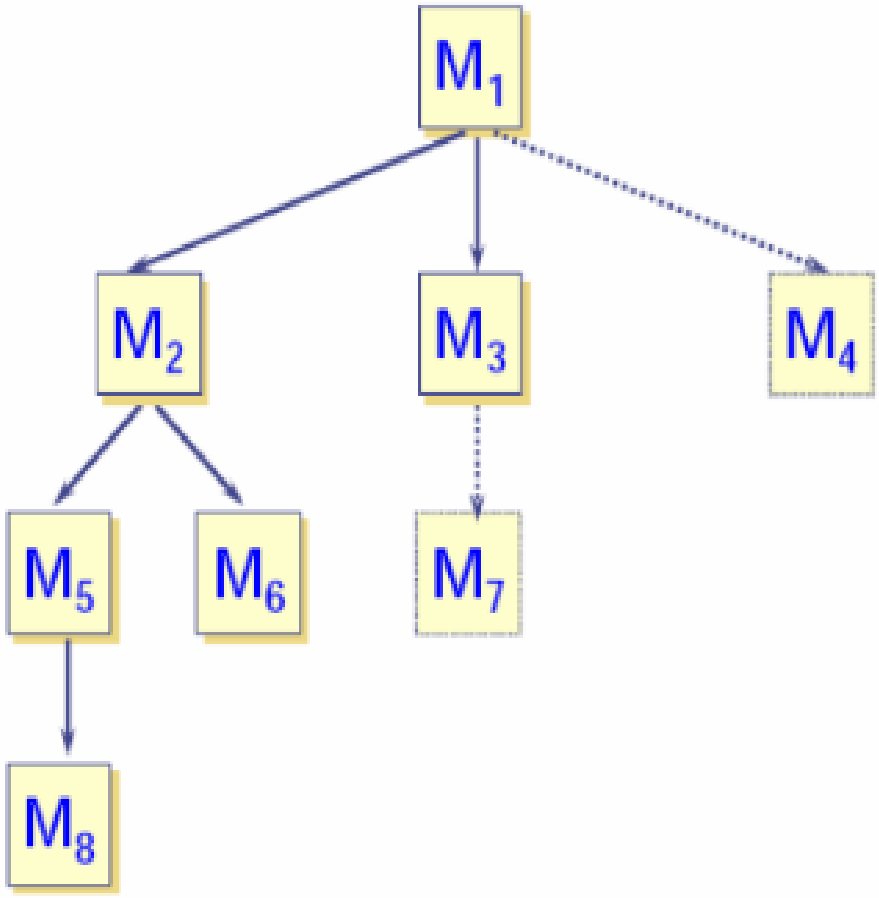
- It incorporates all components directly subordinate at each level,moving across the structure horizontally
- From the figure,components M2,M3,and M4 would be integrated first
- The next control level,M5,M6 and so on,follows

Top-Down Testing with Depth-First

- M1, M2, M5, M8
- M6
- M3, M7
- M4

Top-Down Testing with Breath-First

- M1
- M2, M3, M4
- M5, M6, M7
- M8



Integration Testing: Top-Down

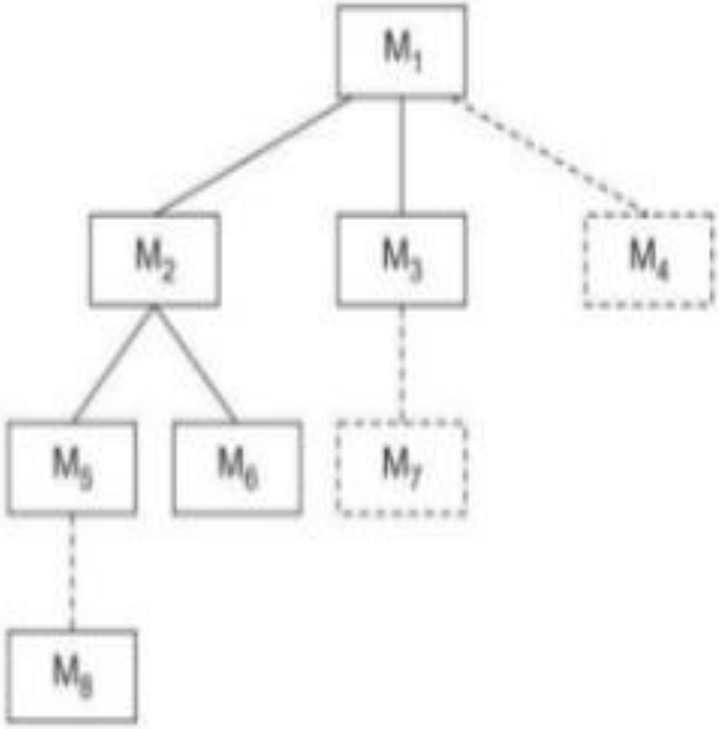
- Modules are integrated by moving downward through the control hierarchy beginning with the main control module.

- **Depth-first integration:**

M1, M2, and M5 would be integrated first. Next, M8 or M6 would be integrated. Then, the central and right-hand control paths are built.

- **Breadth-first integration:**

M2, M3, and M4 would be integrated first. The next control level M5, M6, and so on follows.



Top-Down Integration Testing

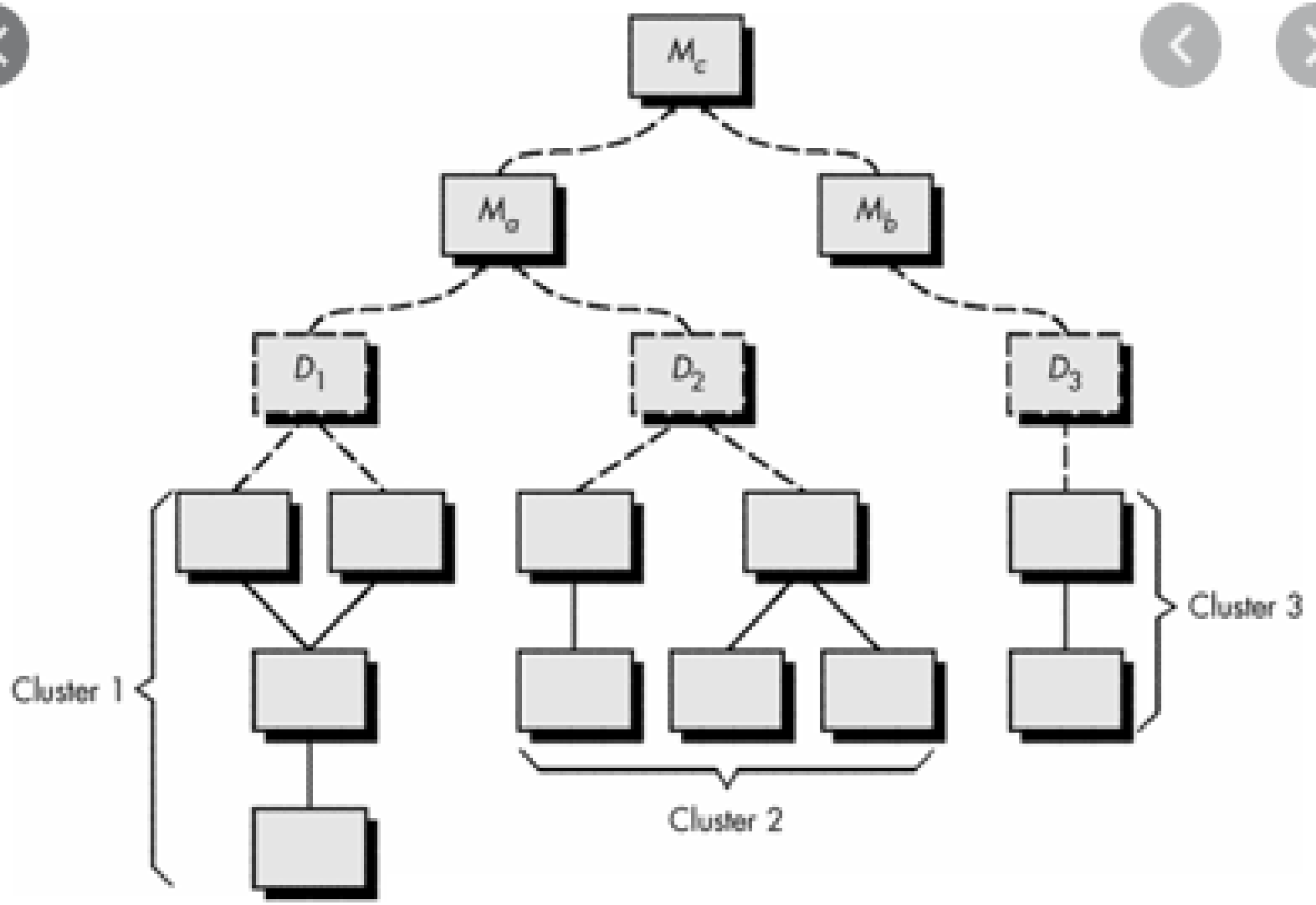
Bottom-up integration

- As the name implies,begins construction and testing with atomic modules(i.e.,components at the lowest levels in the program structure
- Because components are integrated from the bottom up,processing required for components subordinate to a given level is always available and the need for stubs is eliminated
- A bottom-up integration strategy may be implemented with the following steps

Bottom-up integration

- Low-level components are combined into clusters(sometimes called builds) that perform a specific software subfunction
- A driver(a control program for testing) is written to coordinate test case input and output
- The cluster is tested
- Drivers are removed and clusters are combined moving upward in the program structure

Bottom-up integration



Bottom-up integration

- Components are combined to form clusters 1,2 and 3.
- Each of the clusters is tested using a driver(shown as a dashed block)
- Components in clusters 1 and 2 are subordinate to Ma
- Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma
- Similarly,driver D3 for cluster 3 is removed prior to integration with module Mb
- Both Ma and Mb will ultimately be integrated with component Mc

Top-down *approach*	**Bottom-up** *approach*
Undifferentiated and "one-way" approach.	Specific and analytic approach.
Targets given by executives.	Executives set the direction and define the mission.
Employers follow instructions (push).	Employers involvement in identifying and delivering optimizations (share).
Typically, long execution time.	Typically, short execution time.
Typically, short-term impact on company's *status quo*.	Strong impact on company's *status quo* in the long-term. Tangible and lasting results.

System Testing

- System testing is actually a series of different tests whose purpose is to fully exercise the computer-based system
- Although each test has a different purpose,all work to verify that system elements have been properly integrated and perform allocated functions

Recovery Testing

- Computer-based systems must recover from faults and resume processing within a prescribed time
- In some cases, a system must be fault tolerant;that is processing faults must not cause overall system function to cease
- In other cases, a system failure must be corrected within a specified period of time or severe economic damage will occur

Recovery Testing

- Force the software to fail in a variety of ways
- Verify that recovery is properly performed
- If recovery is automatic, re-initialisation, checkpointing mechanisms, data recovery, and restart are evaluated for correctness
- If recovery requires human intervention, mean time to repair is evaluated

Security Testing

- All the computer-based systems that handles sensitive information or causes the actions that can harm or benefit any individuals is tested for illegal attacks to the system
- The examples of illegal activities are:
  - Hackers who try to attack systems for sport
  - Detained employees who try to attack the system for revenge
  - Some mischievous individuals who try to attack system for illicit personal gain
- Security testing verifies whether a protection mechanism is built or not.This mechanism should protect the system from illegal attacks and unauthorized access

Security Testing

- In security testing,the tester may work as an intruder that tries to attack the system to check its security
- The tester can do anything to check the system's security.
- For example
  - the tester may try to obtain passwords and may attack the system to break down the defences that were constructed

Security Testing

- Overload the system by series of requests to cause denial of service
- Purposely cause system errors
- Attack the system during recovery
- Browse through insecure data
- Find the key to enter into the system

Security Testing

- Verify that protection mechanisms built into the system will protect it from improper penetration
- Tester plays the role(s) of individual who desires to penetrate the system
- System designer to make penetration cost greater than the value of information obtained

Stress Testing

- It executes a system in a manner that demands resources in abnormal quantity,frequency,or volume
- For example,(1)special tests may be designed that generate ten interrupts per second,when one or two is the average rate,(2)input data rates may be increased by an order of magnitude to determine how input functions will respond,(3)test cases that require maximum memory or other resources are executed,(4)test cases that may cause memory management problems are designed,(5)test cases that may cause excessive hunting for disk-resident data are created

Performance Testing

- It is designed to test the run-time performance of software within the context of an integrated system
- It occurs throughout all steps in the testing process
- Even at the unit level,the performance of an individual module may be assessed as tests are conducted
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation
- That is often necessary to measure resource utilization(e.g.,processor cycles)
- External instrumentation can monitor execution intervals,log events(e.g.,interrupts)as they occur,and sample machine states on a regular basis

Performance Testing

- It is generally conducted for real-time and embedded systems
- If the software is not functioning as per the requirements,then it is unacceptable

Testing Techniques

- White-Box Testing
- Black-Box Testing

White-Box Testing

- Knowing the internal workings of a product,tests can be conducted to ensure that internal operations are performed according to specifications,and all internal components have been adequately exercised
- Sometimes called as glass-box testing,that uses the control structure described as part of component-level design to derive test cases
- In this technique,the software engineer can derive test cases that (1) guarantee that all independent paths within a module have been exercised at least once,(2)exercise all logical decisions on their true and false side,

White-Box Testing

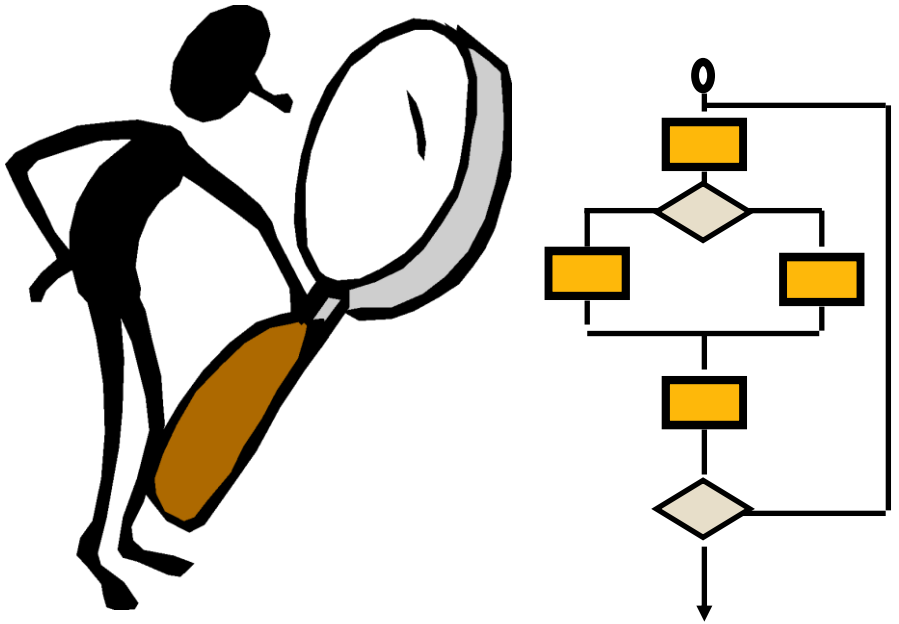
- (3)execute all loops at their boundaries and within their operational bounds, and(4)exercise internal data structures to ensure their validity

Black-Box Testing

- Knowing the specified function that a product has been designed to perform,tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function

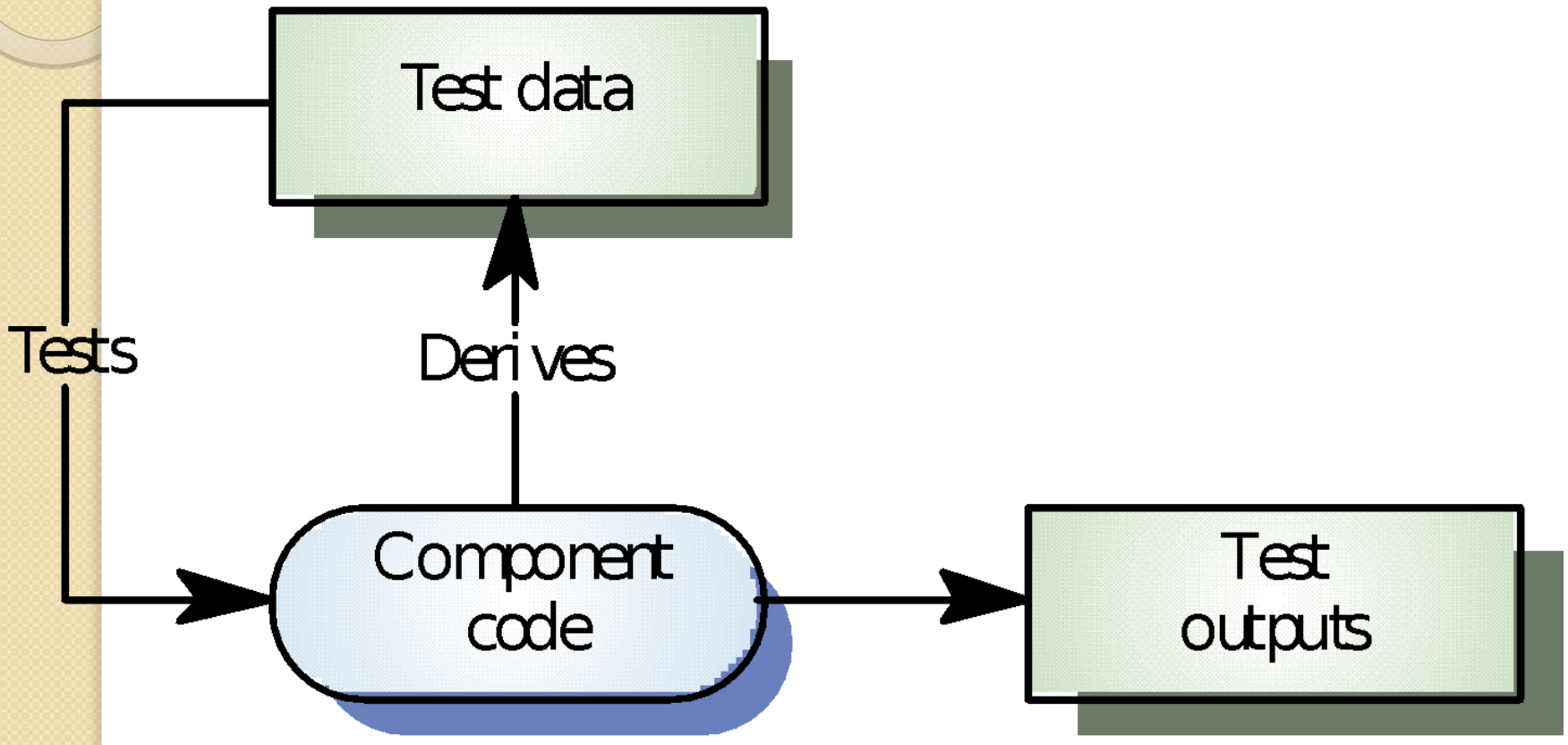
White-box Testing

White-Box Testing



... Our goal is to ensure that all statements and conditions have been executed at least once.

White-box testing



White-box Testing

- Uses the control structure part of **component-level** design to derive the test cases
- These test cases
  - Guarantee that **all independent paths within a module have been exercised at least once**
  - Exercise all logical decisions on their true and false sides
  - Execute all loops at their boundaries and within their operational bounds,and
  - Exercise internal data structures to ensure their validity

Basis Path Testing

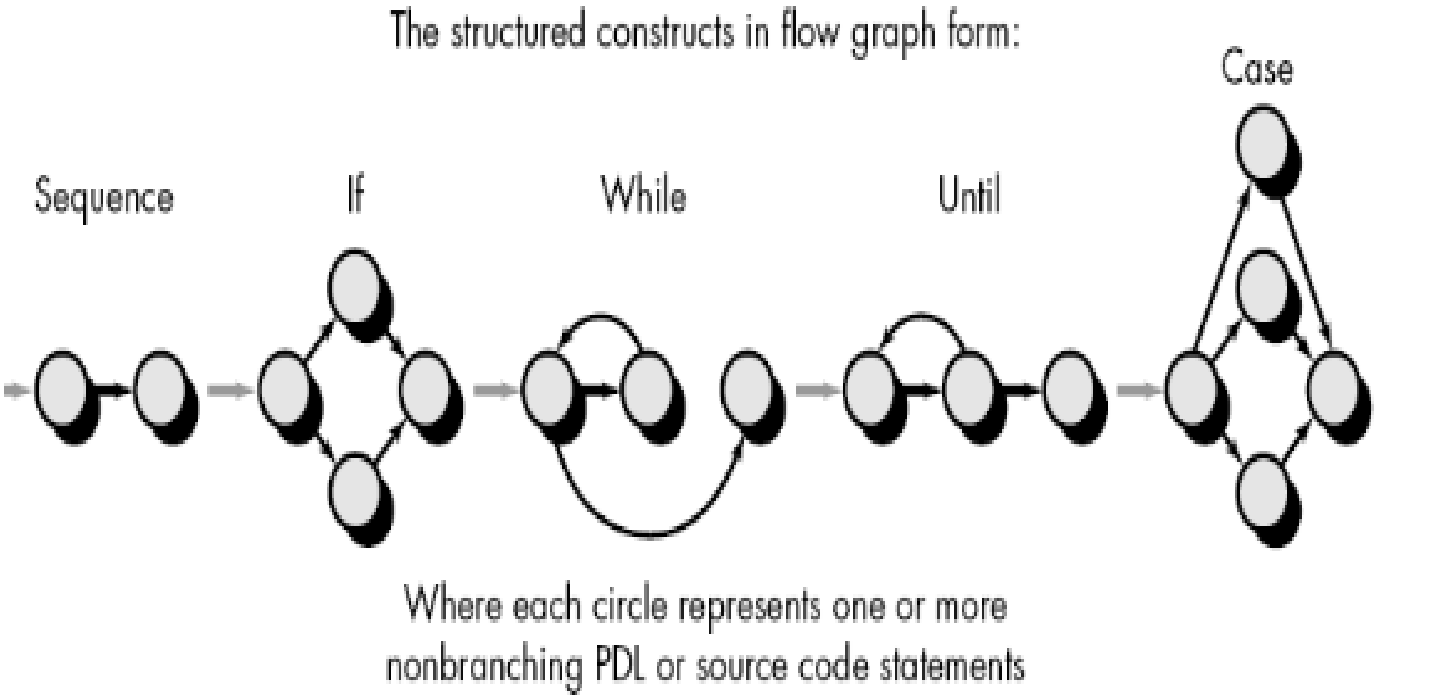
White-box testing technique proposed by Tom McCabe

Enables the test case designer to derive a logical complexity measure of a procedural design

Uses this measure as a guide for defining a basis set of execution paths

Test cases derived to exercise the basis set are guaranteed to execute **every statement** in the **program at least one time** during testing

Flow graph notation

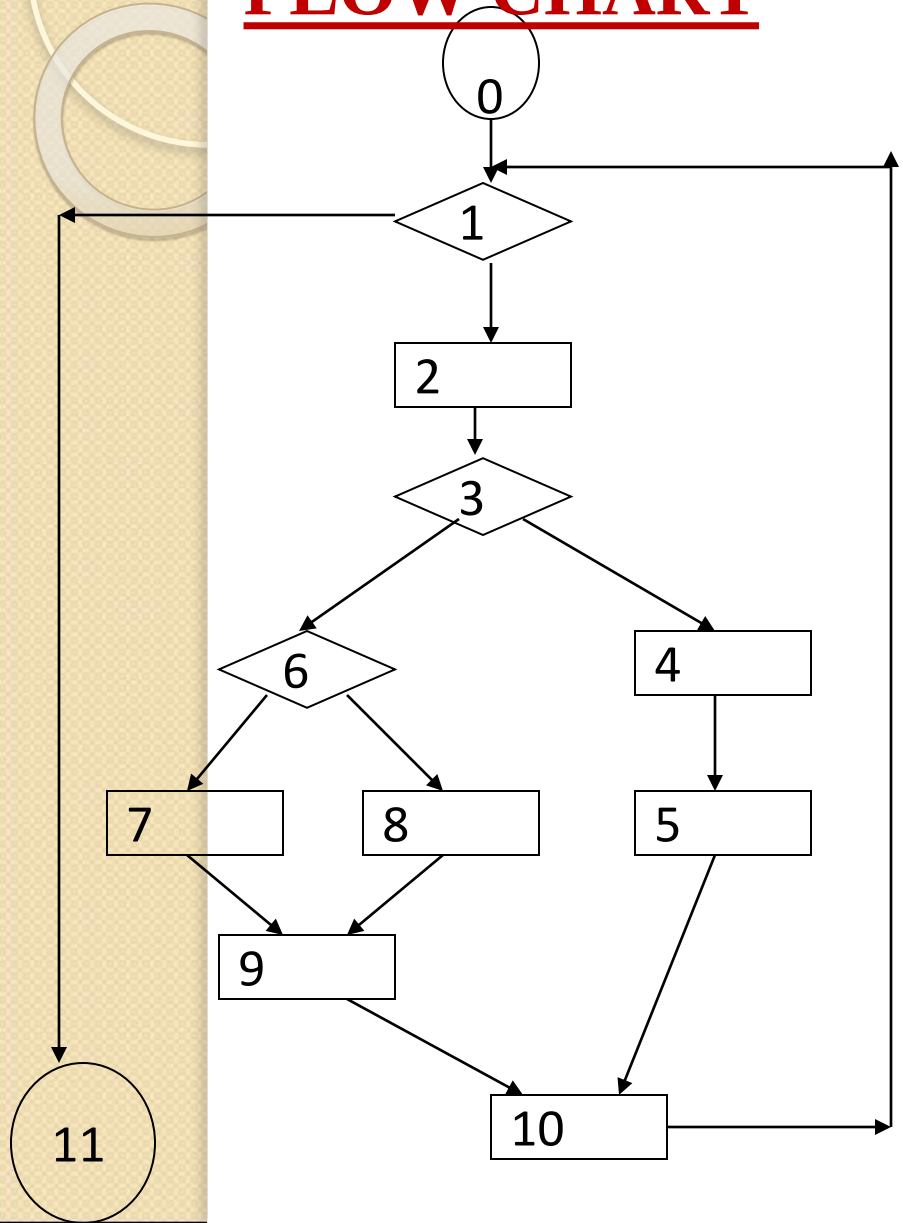


Flow Graph Notation

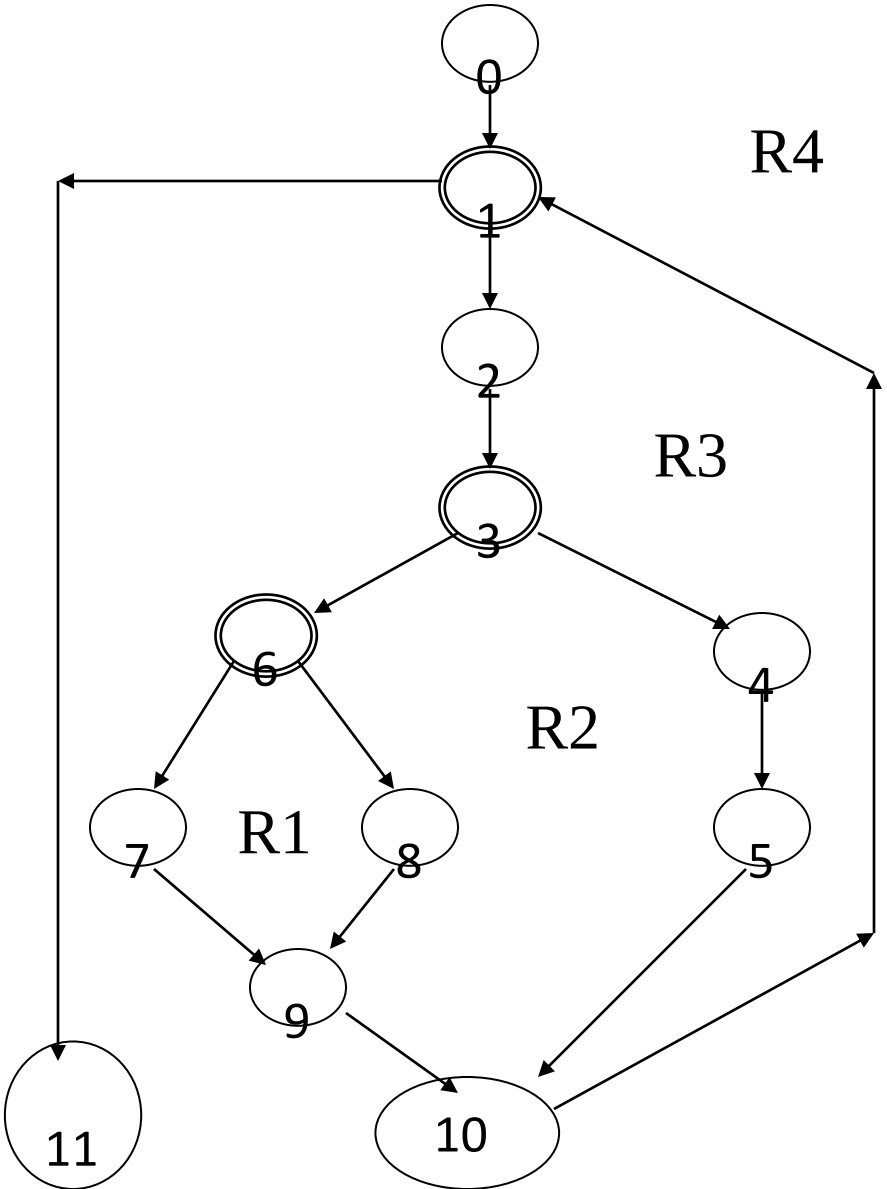
- A circle in a graph represents a **node**, which stands for a **sequence** of one or more procedural statements
- A node containing a simple conditional expression is referred to as a **predicate node**
  - A predicate node **has two edges** leading out from it (True and False)
- An **edge,** or a link, is an arrow representing flow of control in a specific direction
  - **An edge must start and terminate at a node**
  - **An edge does not intersect or cross over another edge**
- Areas bounded by a set of edges and nodes are called **regions**
- When counting regions, include the **area outside the graph as a region, too**

Flow Graph Example

FLOW CHART



FLOW GRAPH



Independent Program Paths

Must move along **at least one** edge that has not been traversed before by a previous path

Basis set for flow graph on previous slide

- Path 1: 0-1-11
- Path 2: 0-1-2-3-4-5-10-1-11
- Path 3: 0-1-2-3-6-8-9-10-1-11
- Path 4: 0-1-2-3-6-7-9-10-1-11

The **number of paths** in the basis set is determined by the **cyclomatic complexity**

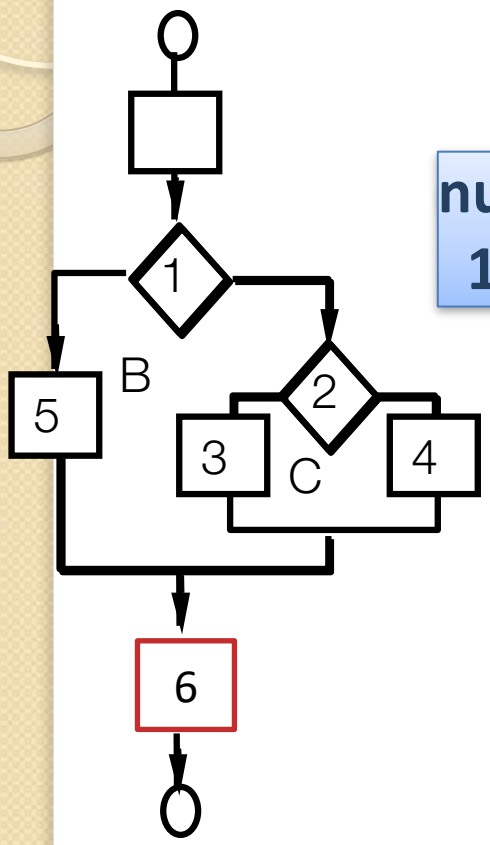
Cyclomatic Complexity

- It is a software metric that provides a quantitative measure of the **logical complexity** of a program
- Defines the **number of independent paths** in the basis set
- Provides an upper bound for the number of tests that must be conducted to ensure **all statements** have been executed **at least once**
- Can be computed three ways

  V(G)=Number of bounded region + 1
  V(G) = E – N + 2, where E is the number of edges and N is the number of nodes in graph G
  V(G) = P + 1, where P is the number of predicate nodes in the flow graph G

- Results in the following equations for the example flow graph
  - Number of regions = 4
  - V(G) = 14 edges – 12 nodes + 2 = 4
  - V(G) = 3 predicate nodes + 1 = 4

Basis Path Testing --



First, we compute the cyclomatic Complexity::

number of simple decisions (predicate node) + 1
1,2,= 2 Nodes +1

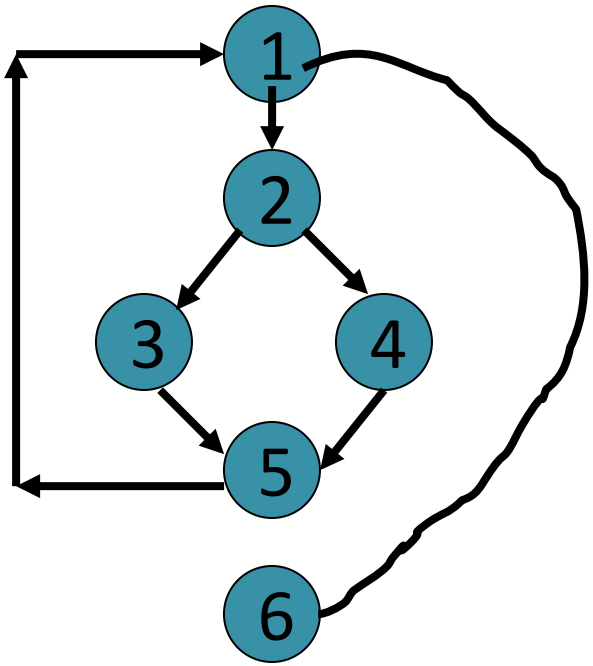
or

number of enclosed areas + 1
B,C= 2areas +1

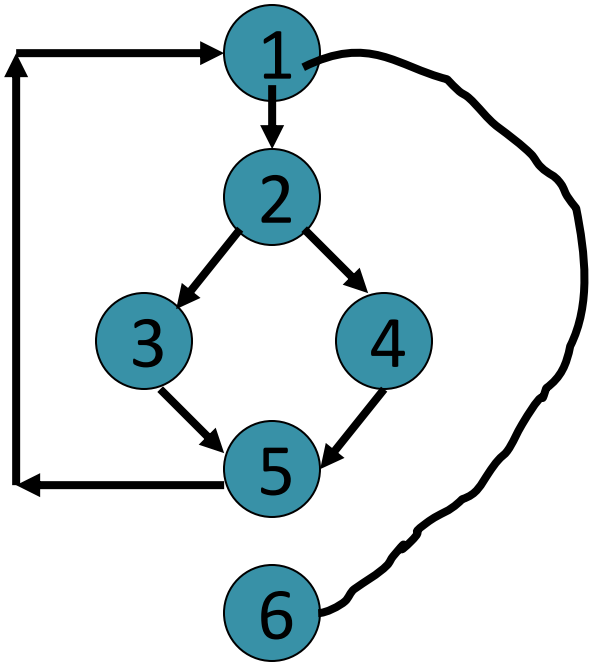
Path to be tested are :
1-5-6 ,1-2-3-6 , 1-2-4-6

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



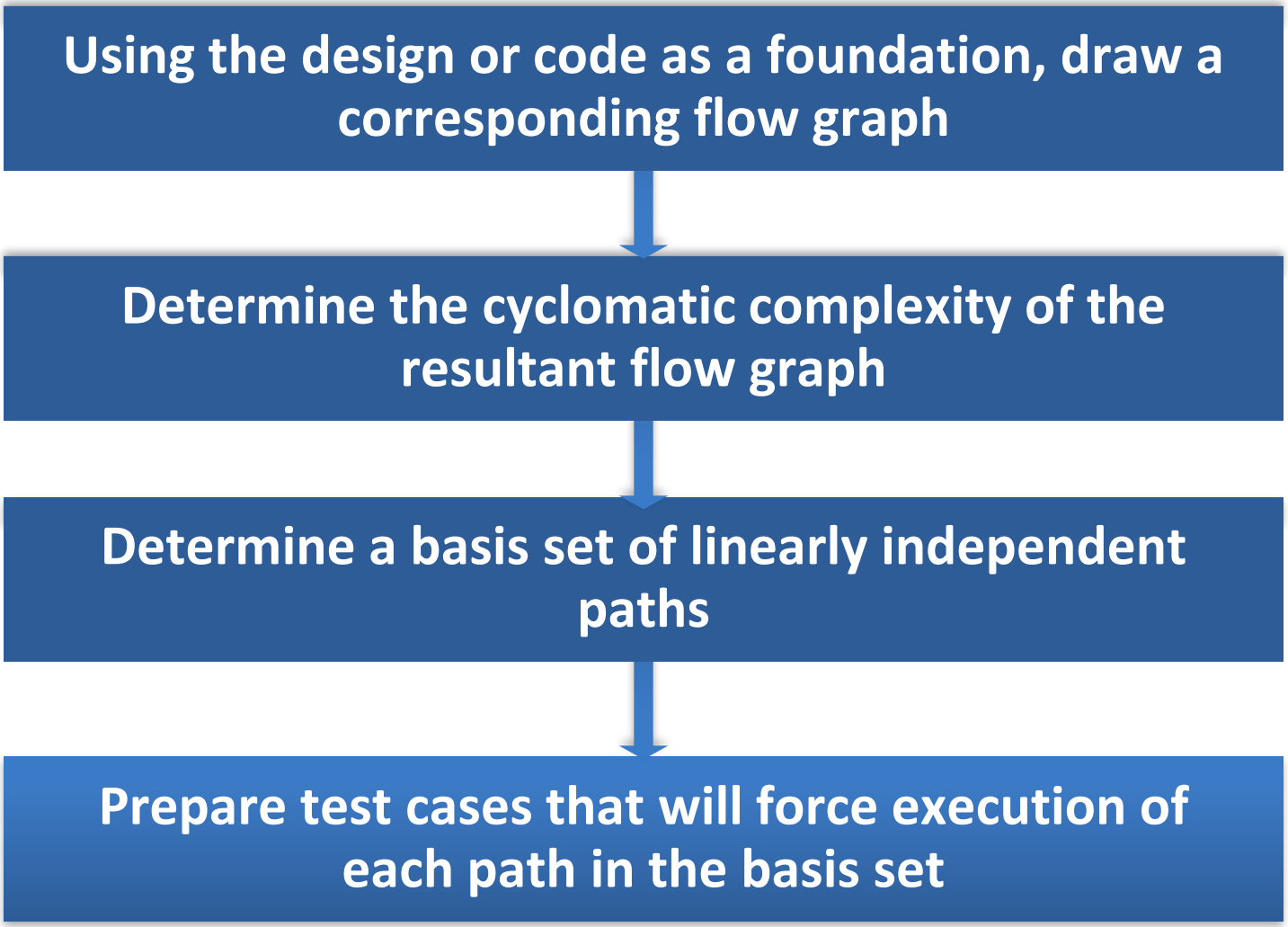
Cyclometic **complexcity =**
V(G)= E-N+2

Cyclomatic complexity =
7 – 6 + 2 = 3.

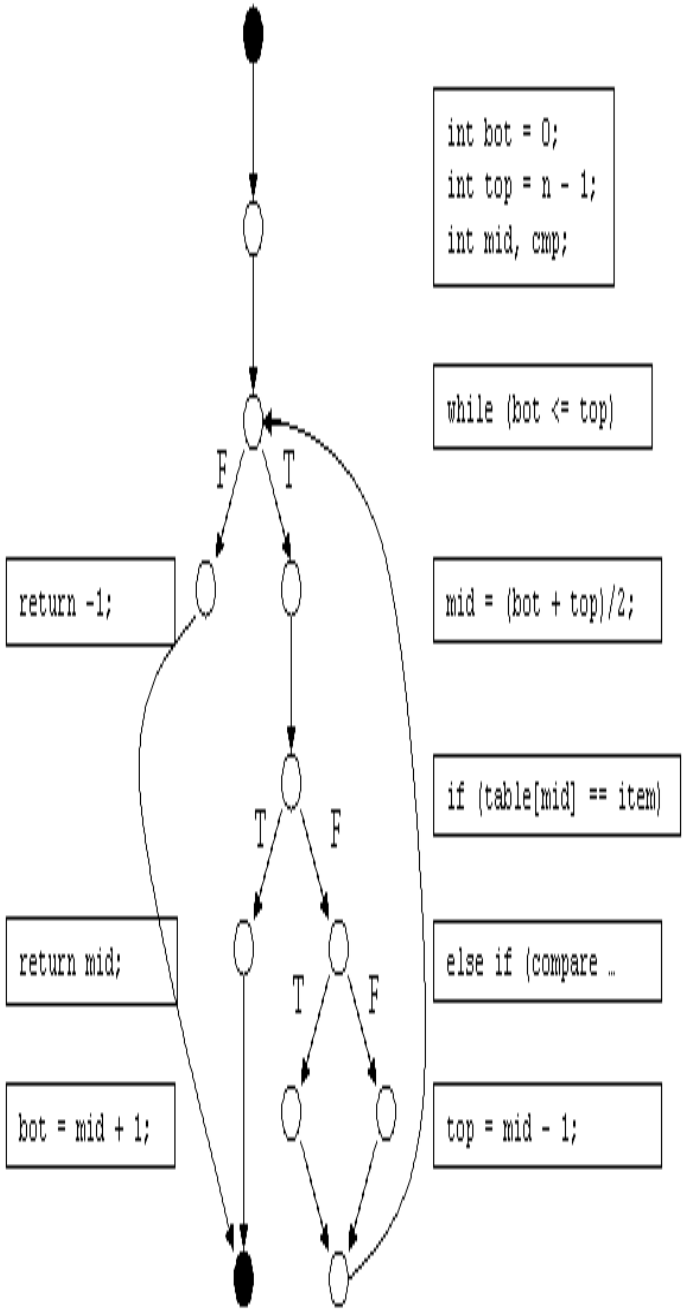
V(G) = Total number of bounded areas + 1

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

Deriving the Basis Set and Test Cases



```
nt BinSearch (char *item, char *table[], int n)
{
   int bot = 0;
    int top = n - 1;
    int mid, cmp;
    while (bot <= top) {
       mid = (bot + top) / 2;
       if (table[mid] == item)
          return mid;
       else if (compare(table[mid], item) < 0)
         top = mid - 1;
       else     bot = mid + 1;
   }
    return -1; // not found
 }
```



CC = number of edges – number of nodes + 2

CC = 14 – 12 + 2

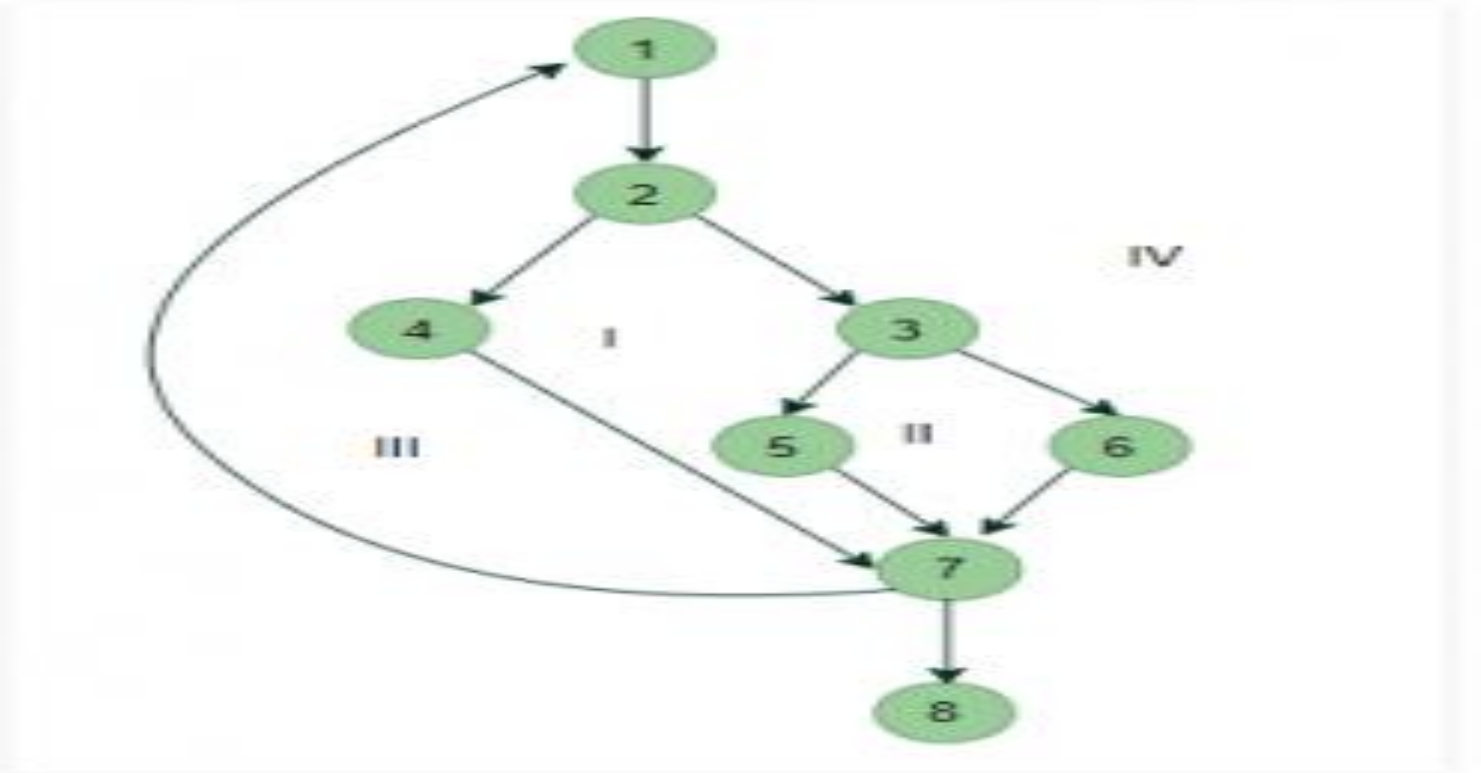
= 4

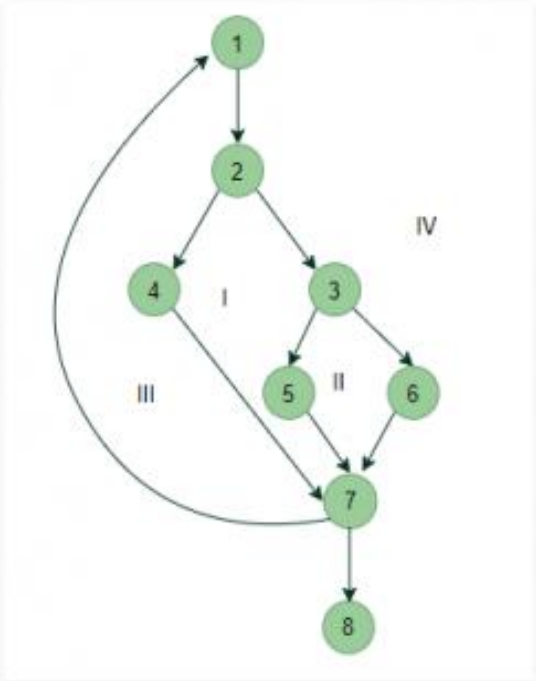
CC = number of decision point + 1

= 3 + 1

= 4

i) 1,2,4,7,8 ii) 1,2,3,5,7,8 iii) 1,2,3,6,7,8 iv) 1,2,4,7,1,2,3,6,7,8

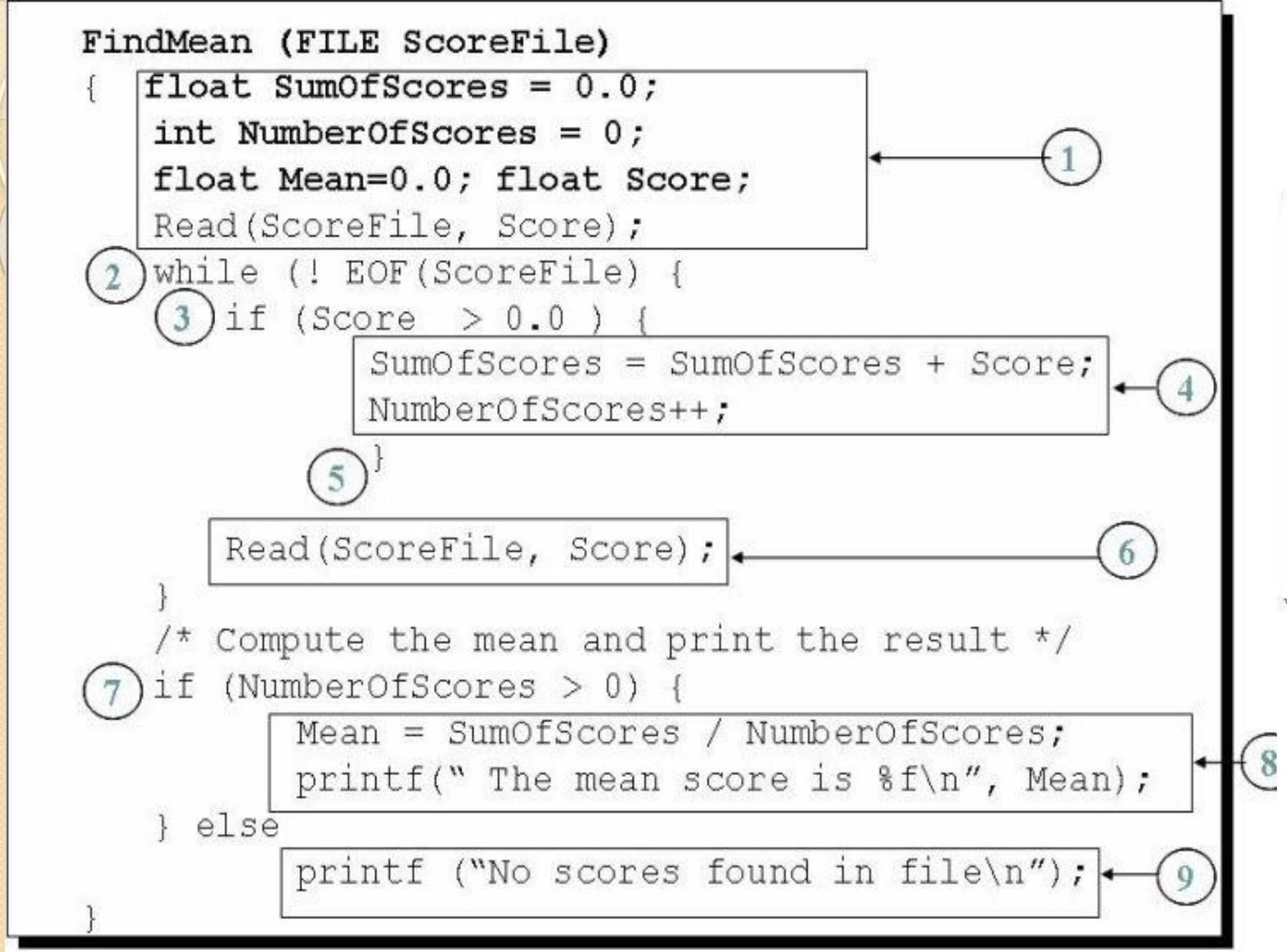




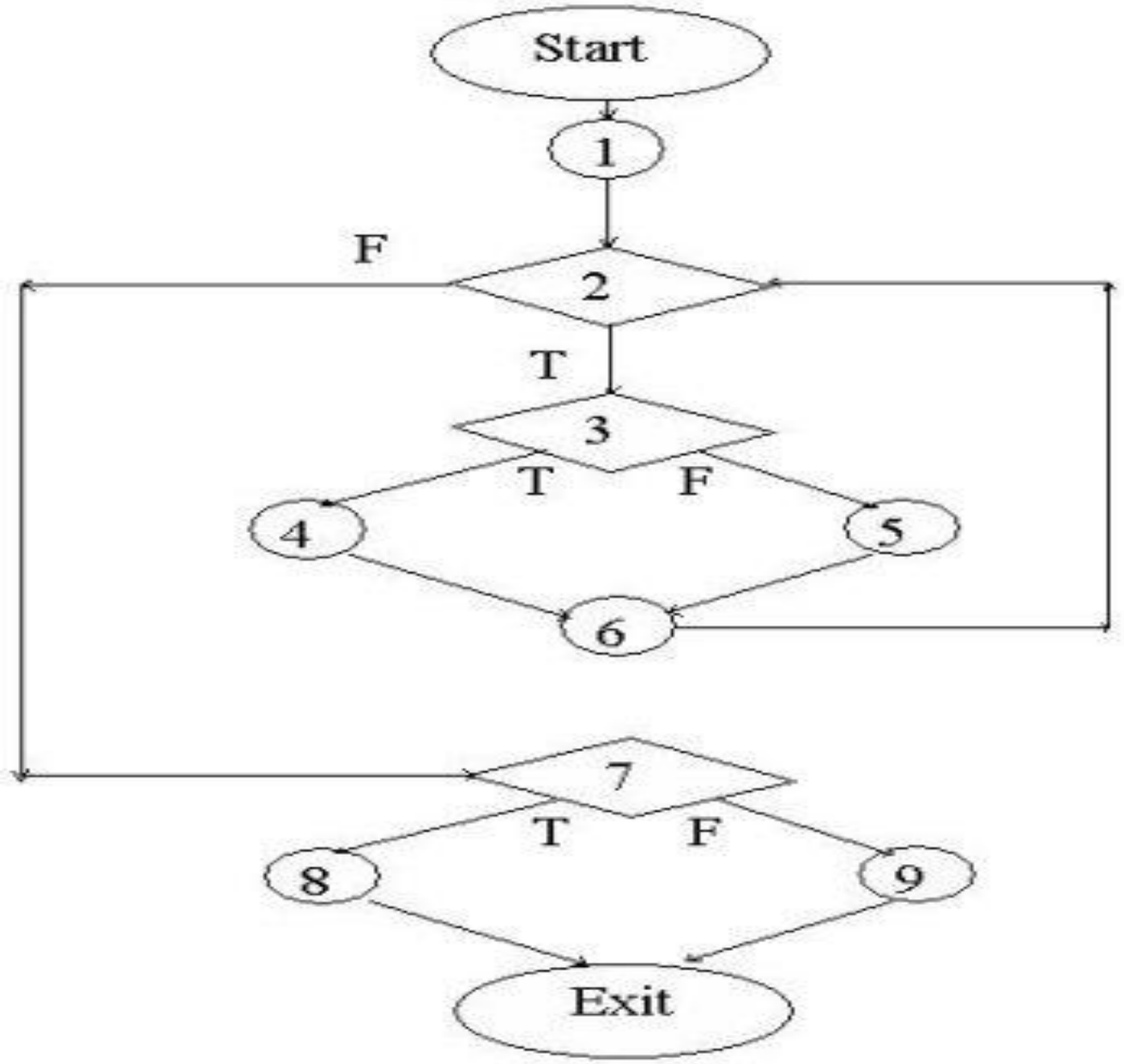
of the above formulae)

No of independent paths = 4

- #P1: 1 – 2 – 4 – 7 – 8
- #P2: 1 – 2 – 3 – 5 – 7 – 8
- #P3: 1 – 2 – 3 – 6 – 7 – 8
- #P4: 1 – 2 – 4 – 7 – 1 – . . . – 7 – 8


FindMean (FILE ScoreFile)
{ float SumOfScores = 0.0;
int NumberOfScores = 0;
float Mean=0.0; float Score;
Read(ScoreFile, Score);
1
2 while (! EOF(ScoreFile) {
3 if (Score > 0.0) {
SumOfScores = SumOfScores + Score;
NumberOfScores++;
4
}
5
Read(ScoreFile, Score);
6
}
/* Compute the mean and print the result */
7 if (NumberOfScores > 0) {
Mean = SumOfScores / NumberOfScores;
printf(" The mean score is %f\n", Mean);
8
} else
printf ("No scores found in file\n");
9
}

Constructing the Logic Flow Diagram



A Second Flow Graph Example

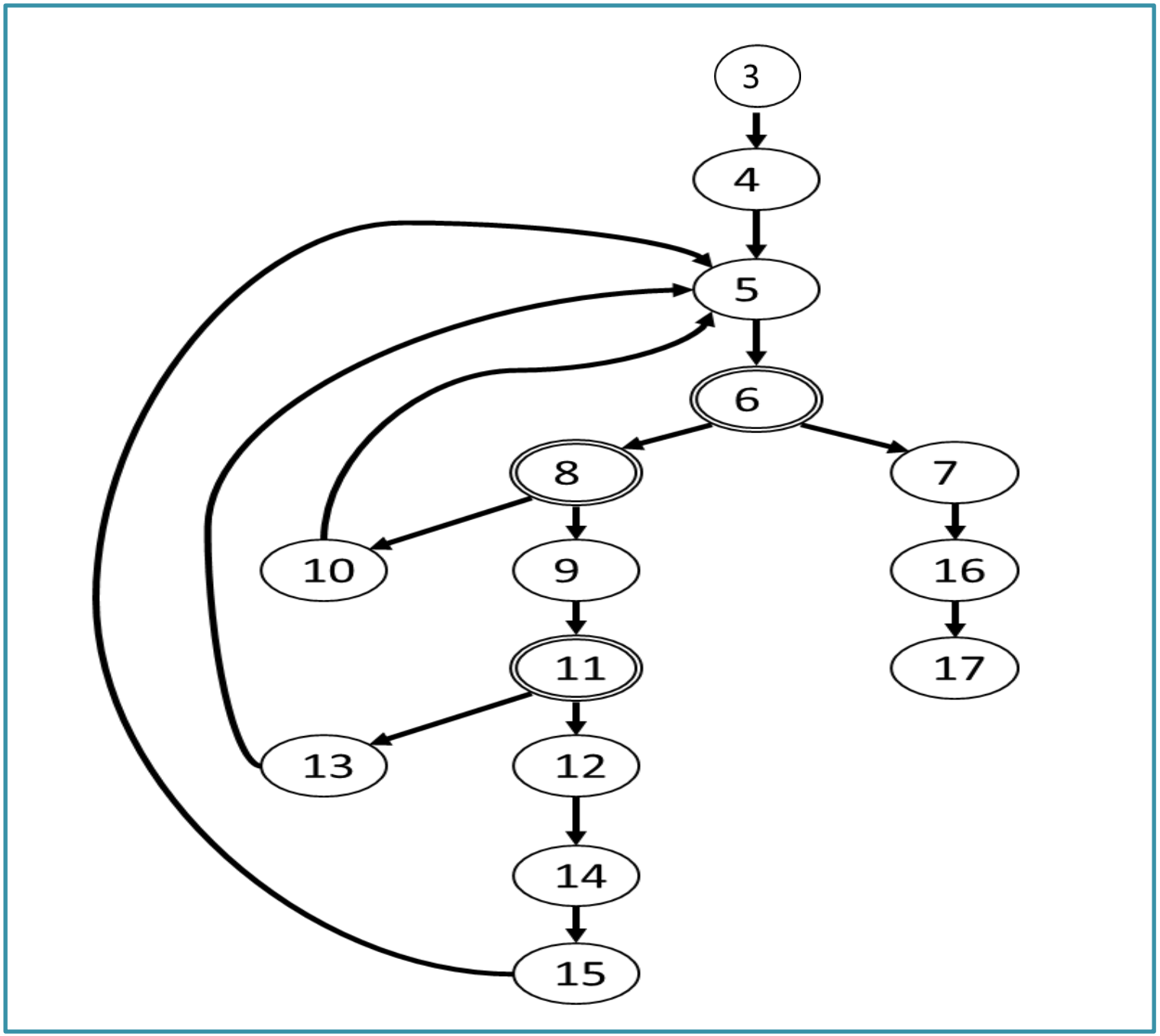
```
 1  int functionY(void)
 2  {
 3     int x = 0;
 4     int y = 19;

 5  A: x++;
 6     if (x > 999)
 7        goto D;
 8     if (x % 11 == 0)
 9        goto B;
10     else goto A;

11  B: if (x % y == 0)
12        goto C;
13     else goto A;

14  C: printf("%d\n", x);
15     goto A;

16  D: printf("End of list\n");
17     return 0;
18  }
```


3
4
5
6
8
7
10
9
16
11
17
13
12
14
15

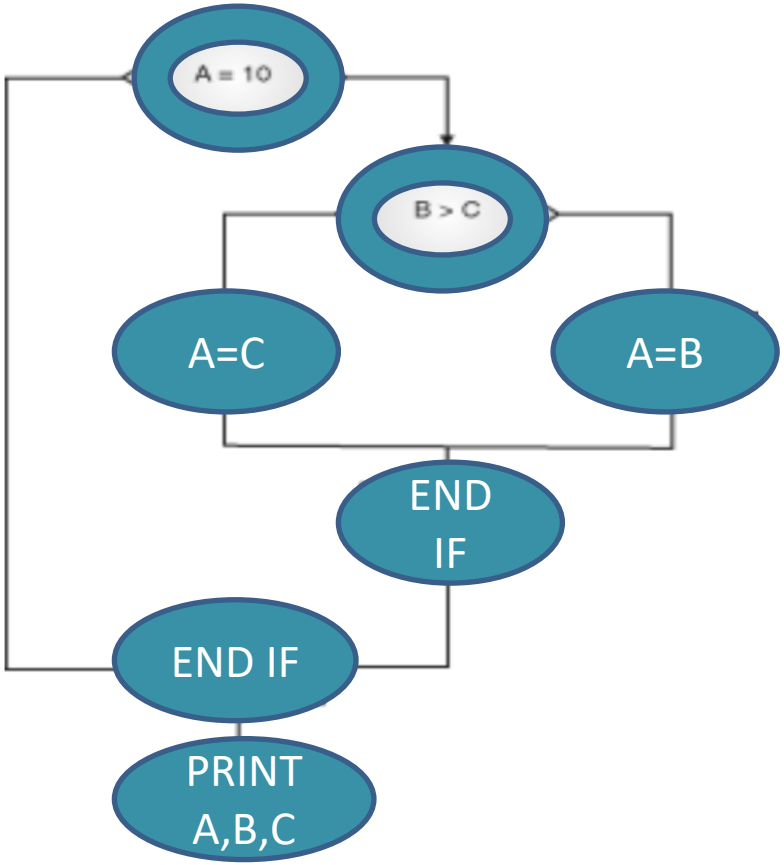
FIND CYCLOMATIC COMPLEXITY

```
IF A = 10 THEN
IF B > C THEN
        A=B
        ELSE A= C
END IF
END IF

        PRINT A
        PRINT B
        PRINT C
```

SOLUTION

Flow graph



The Cyclomatic complexity is calculated using the ab
flow diagram that shows seven nodes (shapes) and eight ed
hence the cyclomatic complexity is 8 - 7 + 2 = 3

```
IF A = 10 THE
IF B > C THE
    A=
    EL
END IF
END IF
    PR
    PR
    PR
```

Test Cases

- Derived during all phases of the development cycle
- Determine what are your expected results before you run a test-case
  - Then, running a test case provides the actual results
- Test cases are presented as a table.

Test Case Table

Test id. (path/no.)	Description	Input	Expected Results	Actual Results	Pass/ Fail

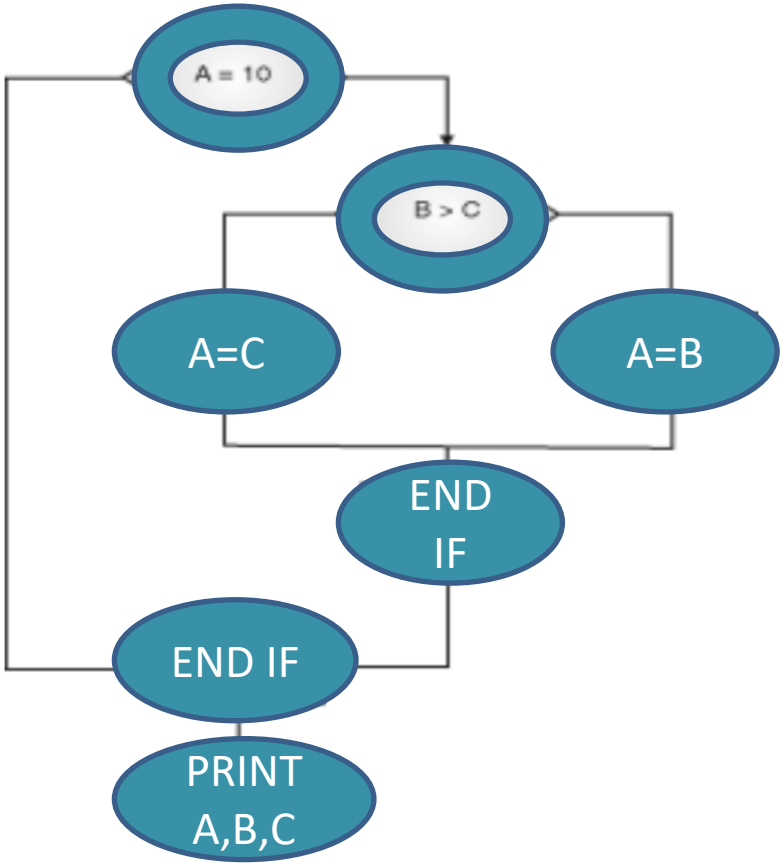
FIND CYCLOMATIC COMPLEXITY

```
IF A = 10 THEN
IF B > C THEN
        A=B
        ELSE A= C
END IF
END IF

        PRINT A
        PRINT B
        PRINT C
```

SOLUTION

Flow graph



The Cyclomatic complexity is calculated using the above control flow diagram that shows seven nodes (shapes) and eight edges (lines), hence the cyclomatic complexity is 8 - 7 + 2 = 3

Test Case Table

Test ID	Description	Input	Expected Result	Actual Result	PASS/FAIL
1	If A=10 ,Compare B & C	10	Comparing B & C	Comparing B & C	PASS
2	If A=10 ,Compare B & C	9	Print A	Print A	Pass
	If A=10 ,Compare B & C	8	Print A	Comparing B & C	Fail
3	If B>C,ASSIGN C to A ,print A,B,C	B=7,C=5	A=5,B=7,C =5	A=5,B=7,C =5	Pass
4	If B>C,ASSIGN C to A ,print A,B,C				

Test ID	Description	Input	Expected Result	Actual Result	Pass/Fail
1	Navigate to the Login Page	User= abc@gmail.com	User should be able to login and navigate to view product menu	User is able to navigate to view product menu	Pass
2	Provide valid user name and password	Username =abc@gmail.com Pwd=asdf	Validate password	User is validated	pass
3	Provide valid user name and password	Username =abc@gmail.com Pwd=qwerty	Validate password	Invalid password	fail

Black-box Testing

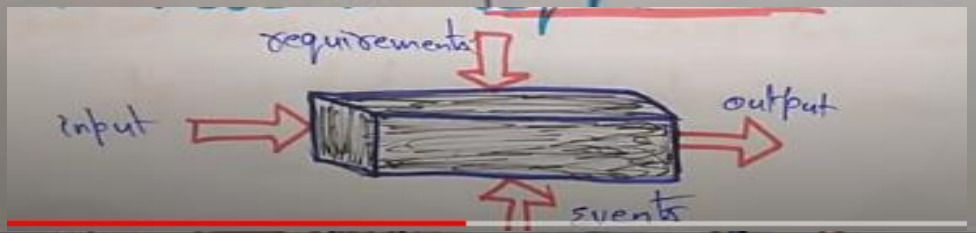
Black-box testing, also called behavioral testing focuses on the functional requirements of the software

Black-Box Testing

⇒ It is a method of s/w testing that examines the functionality of an application without looking into its internal structures or workings

⇒ This method is applied at every level of s/w testing: unit, integration, system and acceptance.

⇒ Tests are based on requirements and functionality.



Black-Box Testing

⇒ It is carried out to test functionality of the program also called as 'Behavioral testing'.

⇒ The tester in this case,
has a set of i/p values and desired results.
- on providing i/p, if the o/p matches with the desired results, the program is tested 'ok'.

* In this ~~testcase~~ testing method, the design & structure of code are not known to tester and testing engineers and end users conduct this test on s/

Acceptance Testing

- A testing technique performed to determine whether or not the software system has met the requirement specifications
- Referring to the functional testing of a user story by the software development team during the implementation phase in agile development

Black-box Testing Steps

- Initially requirements and specifications of the system are examined.
- Tester chooses **valid inputs** (positive test scenario) to check. Also some **invalid inputs** (negative test scenario) are chosen.
- Tester determines expected outputs for all those inputs.
- Software tester constructs test cases with the selected inputs.
- The test cases are executed.
- Software tester compares the actual outputs with the expected outputs.
- Defects if any are fixed and re-tested.

Black box testing strategy:

- **Equivalence Class Testing:** It is used to minimize the number of possible test cases to an optimum level while maintains reasonable test coverage.

- **Boundary Value Testing:** Boundary value testing is focused on the values at boundaries. This technique determines whether a certain range of values are acceptable by the system or not. It is very useful in reducing the number of test cases. It is mostly suitable for the systems where input is within certain ranges.

BVA & EP

i. **BVA (Boundary Value Analysis)** : Boundary Value Analysis focuses on testing boundary conditions e.g.: If we have to derive test conditions to test a text box that accepts age between 18 and 100. Then test conditions would be 17, 18, 19, 50, 99,100 and 101. As you can see, test conditions focus on testing just below boundary, at boundary and over boundary.

ii. **Equivalence Partitioning (EP)** : Equivalence partitioning focuses deriving test conditions based on classes of data at a minimal of 2 classes i.e. Positive and error classes of data. e.g. If we have to test password text box, Positive class would be valid password and invalid class would be password in different case, incomplete password or different password.

Equivalence Partitioning

A black-box testing method that **divides the input domain of a program into classes** of data from which test cases are derived

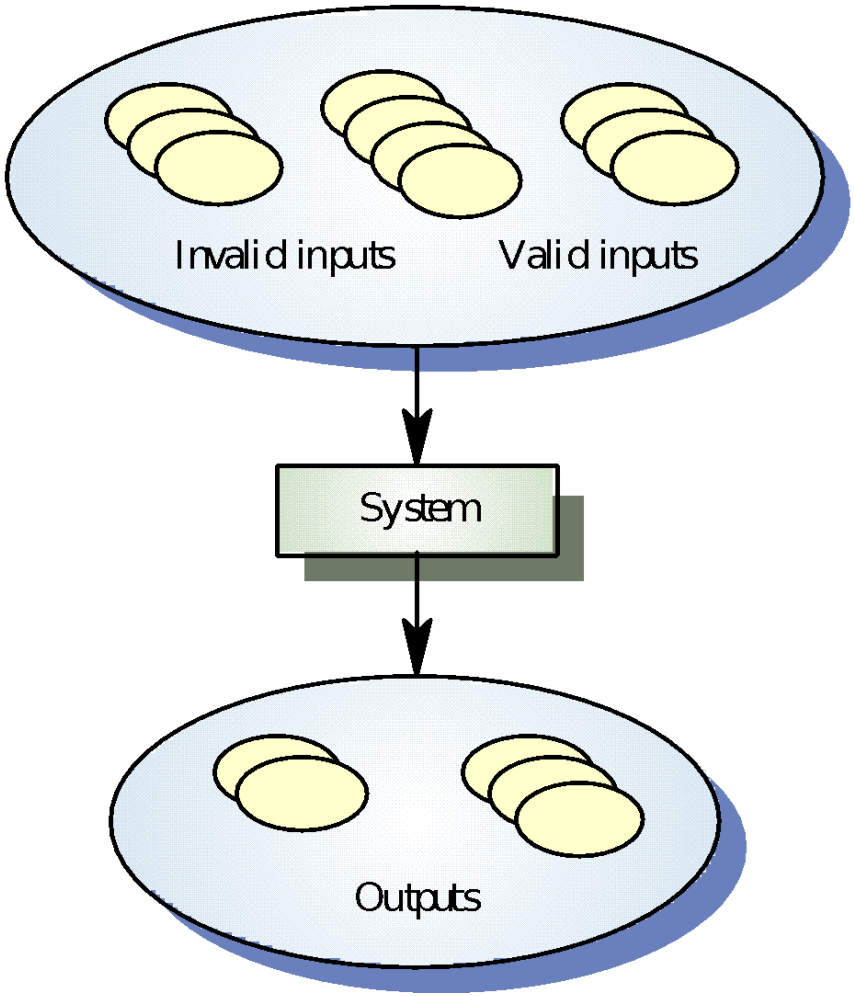
An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed

Test case design is based on an evaluation of equivalence classes for an input condition

An equivalence class represents a **set of valid or invalid states for input conditions**

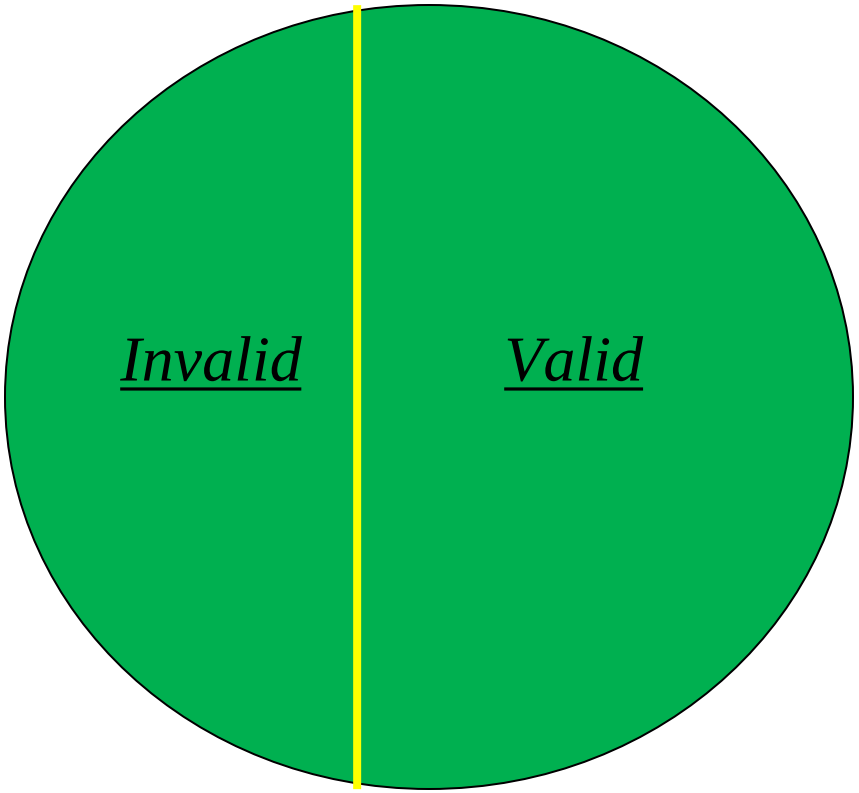
From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercise at once

Equivalence partitioning



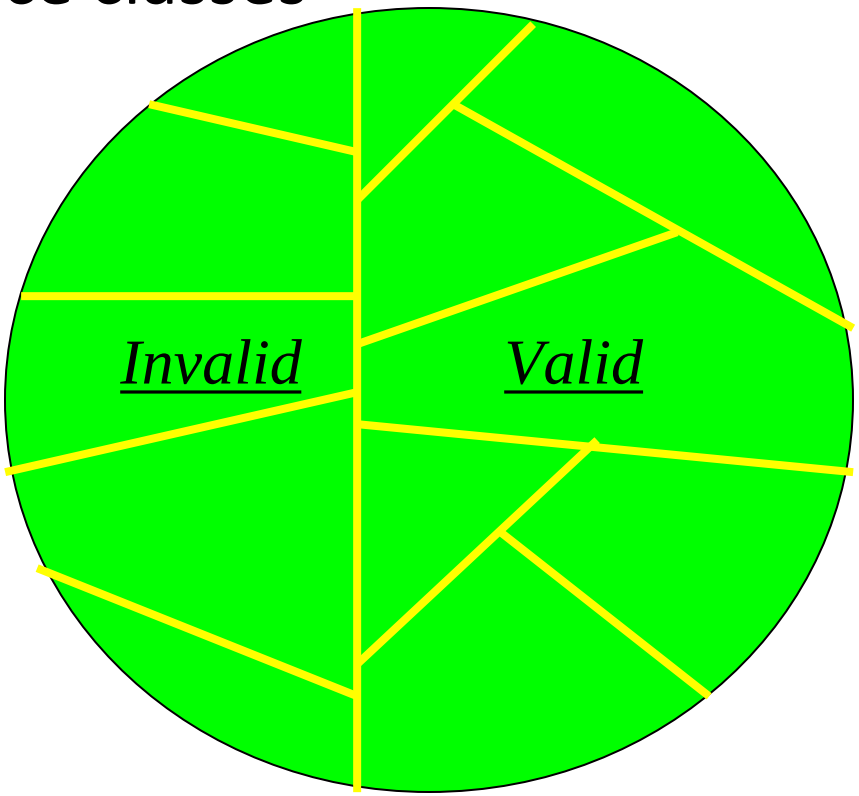
Equivalence Partitioning

First-level partitioning: Valid vs. Invalid values



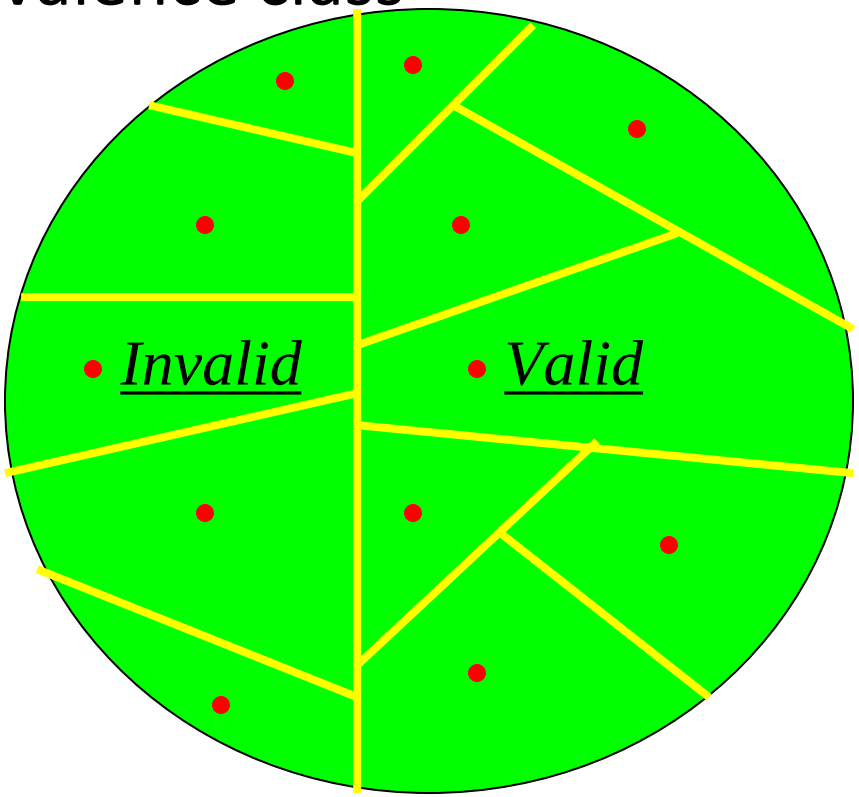
Equivalence Partitioning

Partition valid and invalid values into equivalence classes



Equivalence Partitioning

Create a test case for at least one value from each equivalence class



Guidelines for Defining Equivalence Classes

If an input condition specifies a range, one valid and two invalid equivalence classes are defined

- **Input range: 1 – 10** **Eq classes: {1..10}, {x < 1}, {x > 10}**

If an input condition requires a specific value, one valid and two invalid equivalence classes are defined

- **Input value: 250** **Eq classes: {250}, {x < 250}, {x > 250}**

If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined

- **Input set: {-2.5, 7.3, 8.4}** **Eq classes: {-2.5, 7.3, 8.4}, {any other x}**

If an input condition is a Boolean value, one valid and one invalid class are define

- **Input: {true condition}** **Eq classes: {true condition}, {false condition}**

EQUIVALENCE PARTITIONING

- Test design technique
- Divide input test data into partitions
- Test each partition *ONCE* (the assumption is that any input within a partition is equivalent i.e. produces the same output)

EP Example - Date (1 to 31)

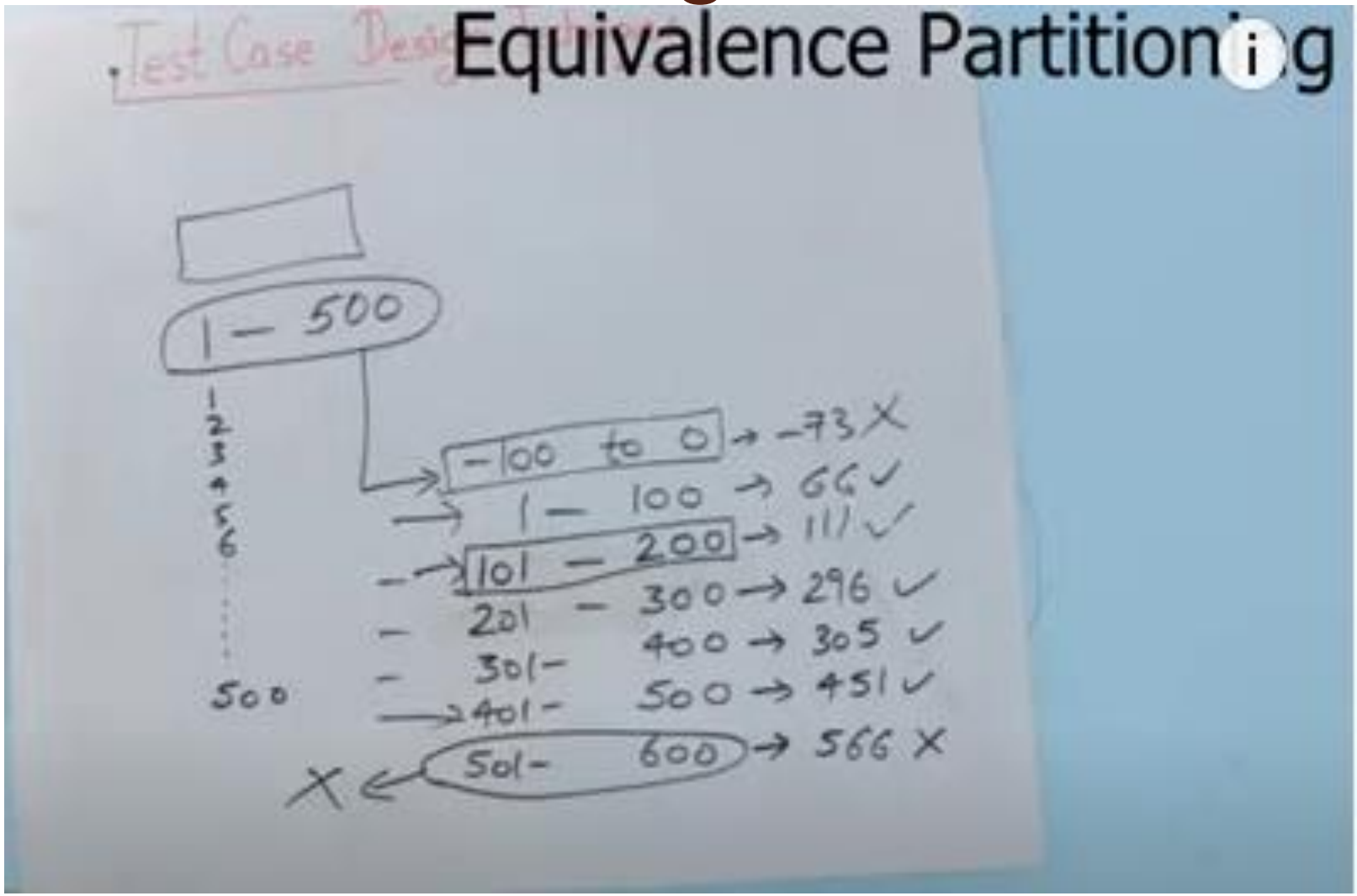
EP Example - Username (6 to 10 characters)

Invalid Partition	Valid Partition	Invalid Partition
• 0 character • 1 character • 2 characters • 3 characters • 4 characters • 5 characters	• 6 characters • 7 characters • 8 characters • 9 characters • 10 characters	• 11 characters • 12 characters • ...

EP Example - Age (18 to 80 years except 60 to 65 years)

*

Black-Box Testing



Boundary Value Analysis

A greater number of errors occur at the boundaries of the input domain rather than in the "center“

Boundary value analysis is a test case design method that complements equivalence partitioning

- It selects test cases at the edges of a class

Guidelines for Boundary Value Analysis

1. If an input condition specifies **a range bounded by values *a* and *b*,** test cases should be designed with values *a* and *b* as well as values **just above and just below *a* and *b***

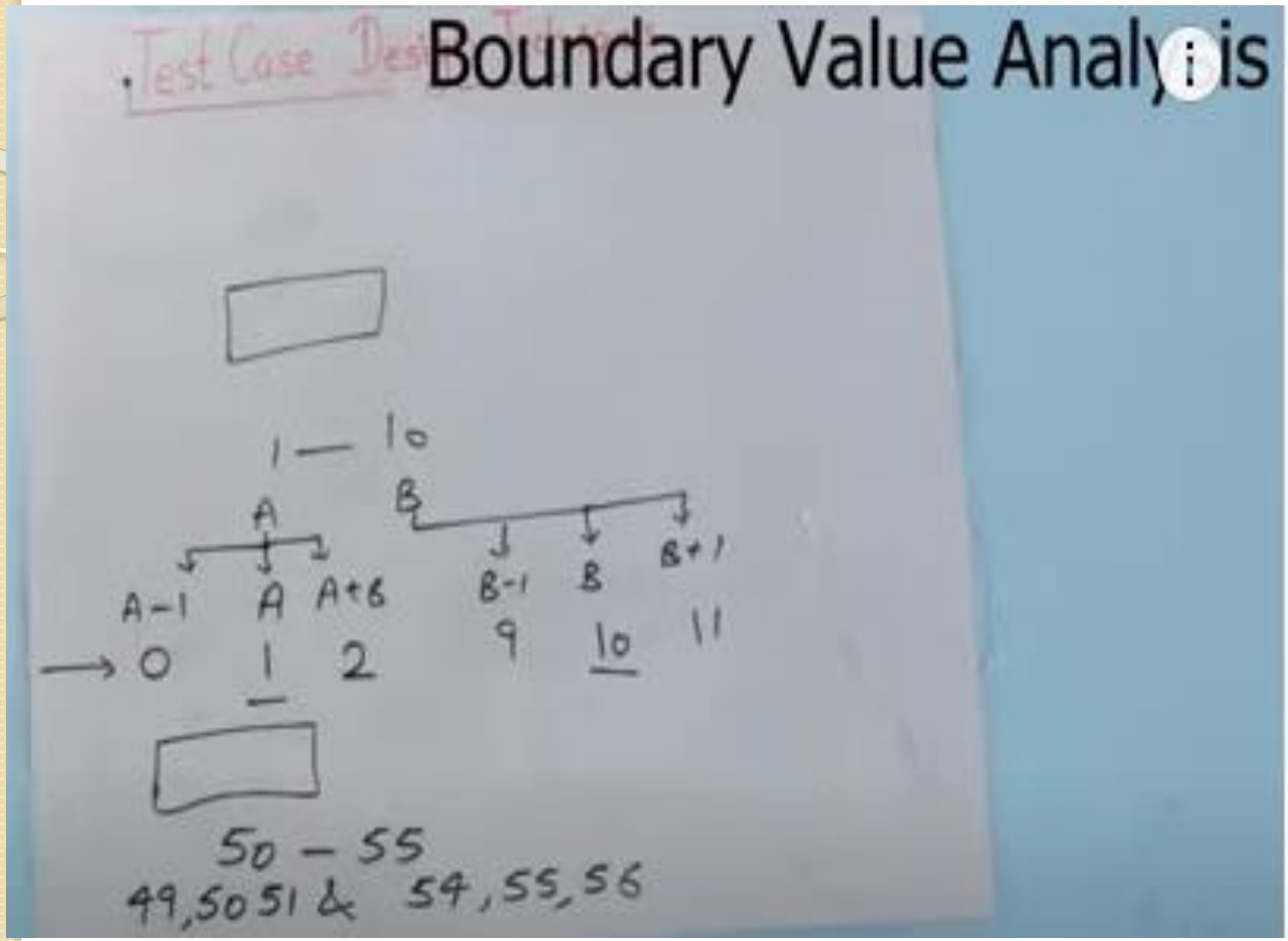
2. If an input condition specifies a **number of values,** test case should be developed that **exercise the minimum and maximum numbers. Values just above and just below the minimum and maximum are also tested**

Apply guidelines 1 and 2 to output conditions; produce output that reflects the minimum and the maximum values expected; also test the values just below and just above

BOUNDARY VALUE ANALYSIS

- Test design technique
- Related to Equivalence Partitioning
- Test *BOTH SIDES* of each boundary (the assumption is that the system behaves differently on either side of a boundary)

Boundary Value Analysis



BVA Example - Date (1 to 31)

Invalid Partition – Valid Partition Lower Boundary		Invalid Partition – Valid Partition Upper Boundary	
Boundary value just below the boundary	Boundary value just above the boundary	Boundary value just below the boundary	Boundary value just above the boundary
0	1	31	32

A=1 (LB),B=31(UB)

A-1	A	A+1		B-1	B	B+1
0	1	2		30	31	32

*

#	Black Box Testing	White Box Testing
1	Black box testing is the Software testing method which is used to test the software without knowing the internal structure of code or program.	White box testing is the software testing method in which internal structure is being known to tester who is going to test the software.
2	This type of testing is carried out by testers.	Generally, this type of testing is carried out by software developers.
3	Implementation Knowledge is not required to carry out Black Box Testing.	Implementation Knowledge is required to carry out White Box Testing.
4	Programming Knowledge is not required to carry out Black Box Testing.	Programming Knowledge is required to carry out White Box Testing.
5	Testing is applicable on higher levels of testing like System Testing, Acceptance testing.	Testing is applicable on lower level of testing like Unit Testing, Integration testing.
6	Black box testing means functional test or external testing.	White box testing means structural test or interior testing.
7	In Black Box testing is primarily concentrate on the functionality of the system under test.	In White Box testing is primarily concentrate on the testing of program code of the system under test like code structure, branches, conditions, loops etc.

8	The main aim of this testing to check on what functionality is performing by the system under test.	The main aim of White Box testing to check on how System is performing.
9	Black Box testing can be started based on Requirement Specifications documents.	White Box testing can be started based on Detail Design documents.
10	The Functional testing, Behavior testing, Close box testing is carried out under Black Box testing, so there is no required of the programming knowledge.	The Structural testing, Logic testing, Path testing, Loop testing, Code coverage testing, Open box testing is carried out under White Box testing, so there is compulsory to know about programming knowledge.

Software Maintenance

- The software maintenance is an activity that actually begins after the software product delivered at the client's end
- In software maintenance,the modifications are carried out or the updates in the software are taken place
- In software maintenance phase no major changes are implemented
- In this,the changes are done in the existing program or the some small new functionality is added

Software Maintenance

- Three decades ago, software maintenance was given the name as iceberg,where the potential problems reside inside and it is not visible to the clients.if maintenance is not done properly,it will sink the entire ship,as icebergs do

Modifiability

- The major cost of the software during its life cycle is the maintenance cost
- When software is designed,the stakeholders want the software should be designed in such a way that future changes can easily be accomodated and implemented with less maintenance cost.
- The modifiability means the ability of the software to be modified
- Modifications include:improvements,corrections,adaptibility in changing environments and in changing requirements and the functional specifications

*

Modifiability

- Modifiability is the ease with a software system can be modified to accommodate changes in requirements,environments and functional specifications
- The maintainability involves modifications in requirements as well as correction in the bugs whereas modifiability does not involve in correcting the bugs

Types of maintenance

- Corrective maintenance
- Adaptive maintenance
- Perfective maintenance
- Preventive mintenance

Corrective maintenance

- It is the type of maintenance in which errors are fixed when it is observed during the use of the software
- In this the errors may be caused due to faulty software design,incorrect logic and improper coding
- All these errors are called as residual errors and they prevent the final specification

Adaptive maintenance

- Adaptive maintenance means the implementation of the modification in the system.
- The changes may be of hardware or the operating system environments

Perfective maintenance

- It deals with the modified and changed user requirements
- The functional enhancements are taken into consideration
- The function and efficiency of the code is continuously improved

Preventive maintenance

- It is used to prevent the possible errors to occur
- Complexity is minimized and the quality of the program is enhanced

Youtube link for software maintenance

- https://www.youtube.com/watch?v=8swQr0kckZI

University Questions

- Differentiate between white box and black box testing(5/10 marks)
- Explain Cyclomatic Complexity.How it is computed?(10 marks)
- What is maintenance?Explain the different types of maintenance.(10 marks)

Software Engineering(SE)

CSC 601

Subject Incharge

Varsha Nagpurkar

Assistant Professor

Room No. 407

email: varshanagpurkar@sfit.ac.in

Software Engineering-Syllabus

- Software Re-engineering
- Reverse Engineering

Software Re-engineering

Clip slide

What is it?

- Consider any technology, you use it regularly.
- It is getting old, breaks too often, takes longer to repair and no longer represents the newest technology.
- If the product is hardware ,throw it away and buy newer model.
- But, if it is custom-built software,need to rebuil it.

"Create a product with added functionality, better performance and reliability and improved maintainability"

Software Re-engineering

Who does it?

- At business level, reengineering performed by business specialist.
- At software level,reengineering performed by software engineers.

Software Re-engineering-Business Process Reengineering(BPR)

Clip slide

Business Process Reengineering (BPR)(I)

- A *business process;* is a set of logically related tasks performed to achieve a defined business outcome.
- Within the business process, people, equipment, material resources and business procedures are combined.
- Examples; Designing a new product, Hiring a new employee...Each demands a set of tasks and needs diverse resources.

Software Re-engineering-Business Process Reengineering(BPR)

Business Process Reengineering (BPR)(II)

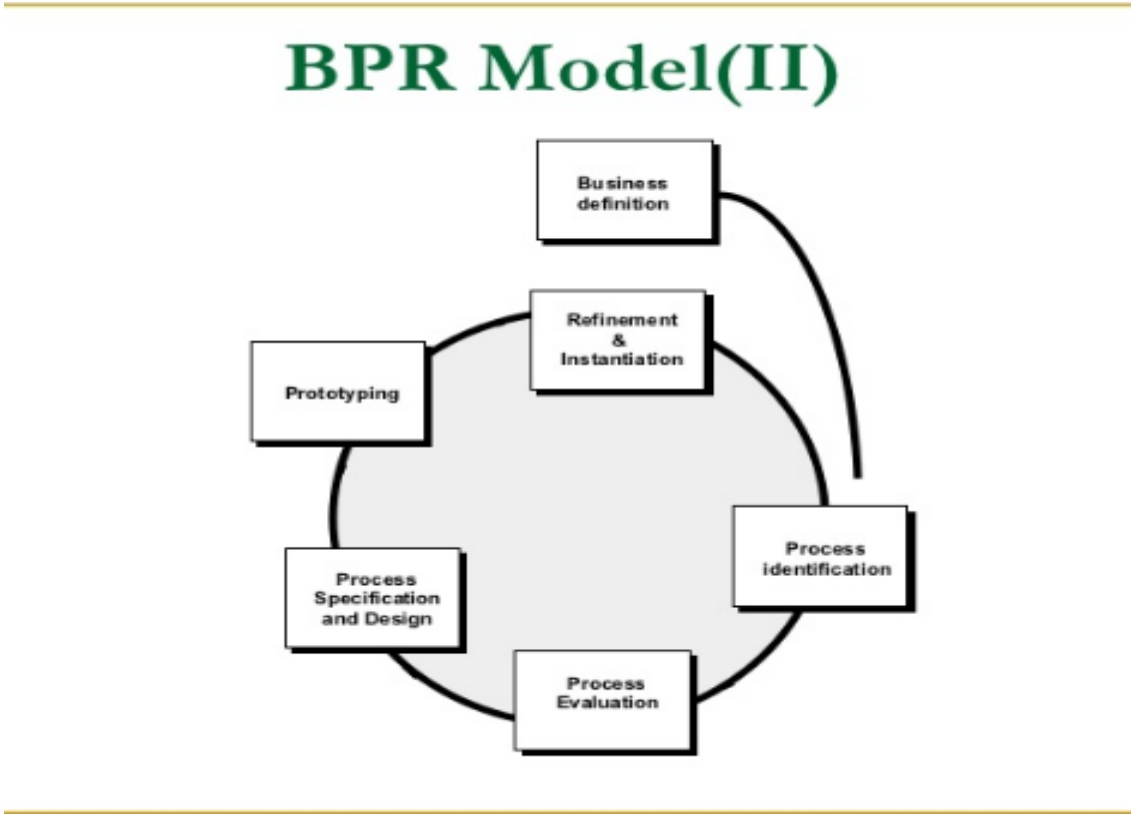
- Overall bussiness;
  The business
    business systems
      business process
        business sub-processes
- BPR can be applied at any level of hierarchy, but towards to upward, the risks grow dramatically.
- Most BPR efforts focus on indivudial processs or sub-processes.

Software Re-engineering-Business Process Reengineering(BPR)

BPR Model(I)

- Like most engineering activities, business process reengineering is iterative.
- There is no start and end to BPR.
- It is an evolutionary process.
- The model defines the six activities.

Software Re-engineering-Business Process Reengineering(BPR)



Software Re-engineering-Business Process Reengineering(BPR)

BPR Model(III)

- **Business redefinition**
  - business goals identified in the context of key drivers
    - cost reduction
    - time reduction
    - quality improvement
    - empowerment
- **Process identification**
  - processes that are critical to achieving business goals are identified and prioritized
- **Process evaluation**
  - existing processes are analyzed and measured
  - process costs and time are noted
  - quality/performance problems are isolated

Software Re-engineering-Business Process Reengineering(BPR)

BPR Model(IV)

- **Process specification and design**
  - based on information obtained during three BPR activities, use-cases are prepared for each process to be redesigned
  - new tasks are designed for each process
- **Prototyping**
  - used to test redisgned processes before integrating them into the business
- **Refinement and instantiation**
  - based on feedback from the prototype, business processes are refined
  - refined processes then instantiated within a business system

Software Re-engineering-Business Process Reengineering(BPR)

Software Reengineering

- Much of the software we depend on today is average 10 to 15 years old.
- Even when these programs were created using the best design and coding techniques known at the time(and most were not), they were created when program size and storage space were principle concerns.
- They were then migrated to new platforms.
- Ajusted for changes in machine and operating system technology.
- Enhanced to meet new users.

"The result is poorly designed structures, poor coding, poor logic and poor documentation"

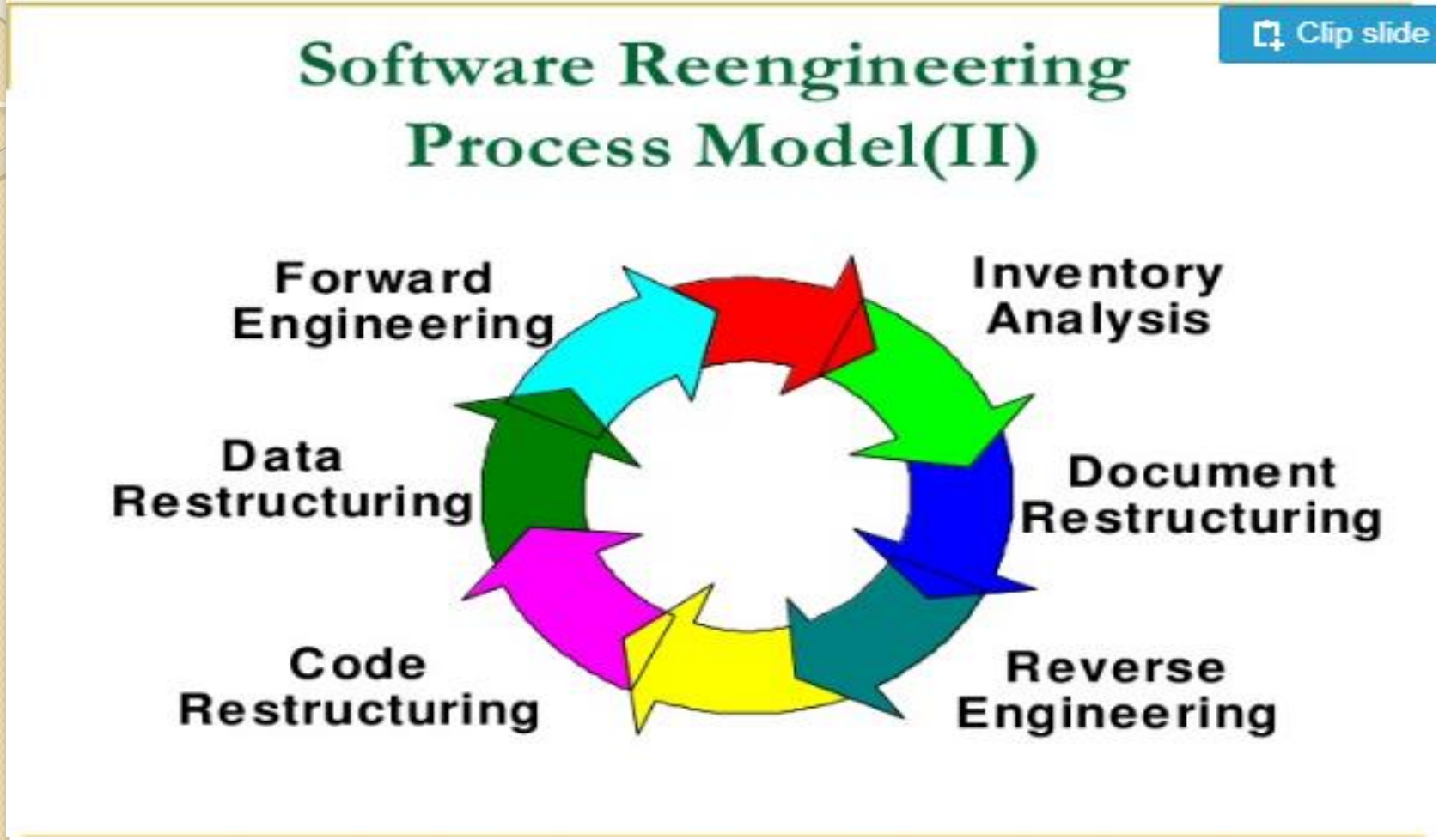
*

Software Re-engineering-Process Model

Software Reengineering Process Model(I)

- Reengineering takes time.
- It costs significant amounts of money, it absorbs resources.
- So,it is accomplished in a few months or even a few years.

Software Re-engineering-Process Model



Software Re-engineering-Process Model

Inventory Analysis

- **Build a table that contains all applications**
- **Establish a list of criteria, e.g.,**
  - name of the application
  - year it was originally created
  - number of substantive changes made to it
  - total effort applied to make these changes
  - date of last substantive change
  - effort applied to make the last change
  - system(s) in which it resides
  - applications to which it interfaces, ...
- **Analyze and prioritize to select candidates for reengineering**

Software Re-engineering-Process Model

Clip slide

Document Restructuring

- Options range from *doing nothing* to *regeneration of all documentation* for critical system.
  - "Creating is far too time consuming". In some cases correct approach. It is not possible to recreate documentation for hundreds of computer programs.
  - "Documentation must be updated, but have limited resources". It may not be necessary to fully redocument an application. Rather then portions of the system that are currently undergoing change are fully documented.
  - "The system is business ciritical and must be fully redocumented. "

Reverse Engineering(I)

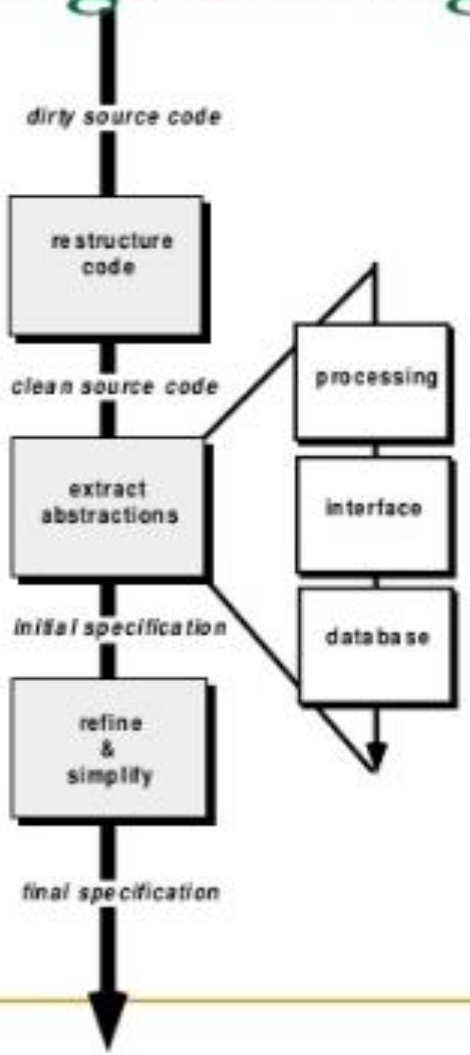
- The term "Reverse Engineering" has its origins in the hardware world.
- A company disassembles a competitive hardware product, to understand its competitor's design and manufacturing secrets.
- For software quite similar.But not a competitor's software. It is company's own program.

"Anayzing a program in an effort to create a representation of the program at a higher level of abstraction than source code"

Reverse Engineering(II)

- Reverse engineering conjures a *"magic slot"*. We feed undocumented source listing into the slot and out the other end comes full documentation for the computer program.
- Reverse engineering can extract design information from source code, but some properties exist;
  - ***Abstraction Level*** : Sophistication of design information that can be extracted from source code. Should be as high as possible.
  - ***Completeness*** : Amount of detail gathered at the abstraction level. Decreases as abstraction level increases.
  - ***Directionality***

Reverse Engineering(III)



Reengineering

17

Reverse Engineering(IV)

- **R.E. To Understand Processing**
  - The overall functionality of the entire application must be understood.
  - A block diagram, representing the interaction between functional abstractions is created.
  - Things become more complex when the code inside a component is considered.
  - In almost every component has a section of code prepares data for processing, a different section of code does processing and another section which prepares the result.
  - For large system CASE tools are used to parse the semantics of existing code.

Reverse Engineering(V)

- **R.E. To Understand Data**
  - Internal data structures
    - Definition of classes
    - Identify local data structures that record information about global data structures
    - Identify relationships between these local data structures & the global data structures
    - Group variables that have logical connections
  - Database structure
    - Build initial object model
    - Determine candidate keys
    - Refine tentative classes – possibly combine to single class
    - Define generalizations
    - Discover associations

Reverse Engineering(VI)

- **R.E. User Interfaces**
  - To fully understand an existing user interface, the structure and behavior of the interface must be specified.
    - What are the basic actions?(keystrokes, mouse clicks...)
    - What is the response of the system to these actions.
  - It is important to note that a replacement GUI may not mirror old interface. In fact, it may be radically different.

Restructuring(I)

- **Does not modify the overall architecture**
  - Code Restructuring
  - Data Restructuring
- **Advantages**
  - Adhere to modern software engineering standards.
  - Frustration among software engineers reduced.
  - Maintenance efforts reduced.
  - Easier to test & debug.

Restructuring(II)

- Code Restructuring
  - The most common type of restructuring.
  - If modules were coded in a way that makes them difficult to understand, test and maintain.
  - To accomplish this activity
    - The source code is analyzed using a restructuring tool
    - Violations of structured programming constructs are noted.
    - Code then restructured manual or automatically.
    - Th resultant restructured code is reviewed and tested.

Restructuring(III)

- Code Restructuring

```
Start:   Get (Time-on, Time-off, Time, Setting, Temp, Switch)
         if Switch = off goto off
         if Switch = on goto on
         goto Cntrld
off:  if Heating-status = on goto Sw-off
         goto loop
on:   if Heating-status = off goto Sw-on
         goto loop
Cntrld:  if Time = Time-on goto on
         if Time = Time-off goto off
         if Time < Time-on goto Start
         if Time > Time-off goto Start
         if Temp > Setting then goto off
         if Temp < Setting then goto on
Sw-off:  Heating-status := off
         goto Switch
Sw-on:   Heating-status := on
Switch:  Switch-heating
loop:    goto Start
```

Spaghetti Control Logic

```
loop
    -- The Get statement finds values for the given variables from the system's
-- environment.
    Get (Time-on, Time-off, Time, Setting, Temp, Switch) ;
    case Switch of
        when On => if Heating-status = off then
                        Switch-heating ; Heating-status := on ;
                   end if ;
        when Off => if Heating-status = on then
                        Switch-heating ; Heating-status := off ;
                   end if;
        when Controlled =>
            if Time >= Time-on and Time < = Time-off then
                if Temp > Setting and Heating-status = on then
                    Switch-heating; Heating-status = off;
                elsif Temp < Setting and Heating-status = off then
                    Switch-heating; Heating-status := on ;
                end if;
            end if ;
    end case ;
end loop ;
```

Structured Control Logic

*

Restructuring(IV)

- Data Restructuring
  - Firstly "Data Analysis" must be done
    - Current data architecture is examined in detail and data models are defined.
    - Data objects and attributes identified, and existing data structures are reviewed for quality.
  - Then "Data Redesign" commences.
    - Clarifies data definitions to achieve consistency among data item names and physical record formats.
  - "Data Name Rationalization"
    - Ensures all data naming conventions conform to local standard.

Forward Engineering (I)

- Also called “renovation” or “reclamation”
- Uses design information from existing software to alter the existing system in an effort to improve its overall quality.

Clip slide

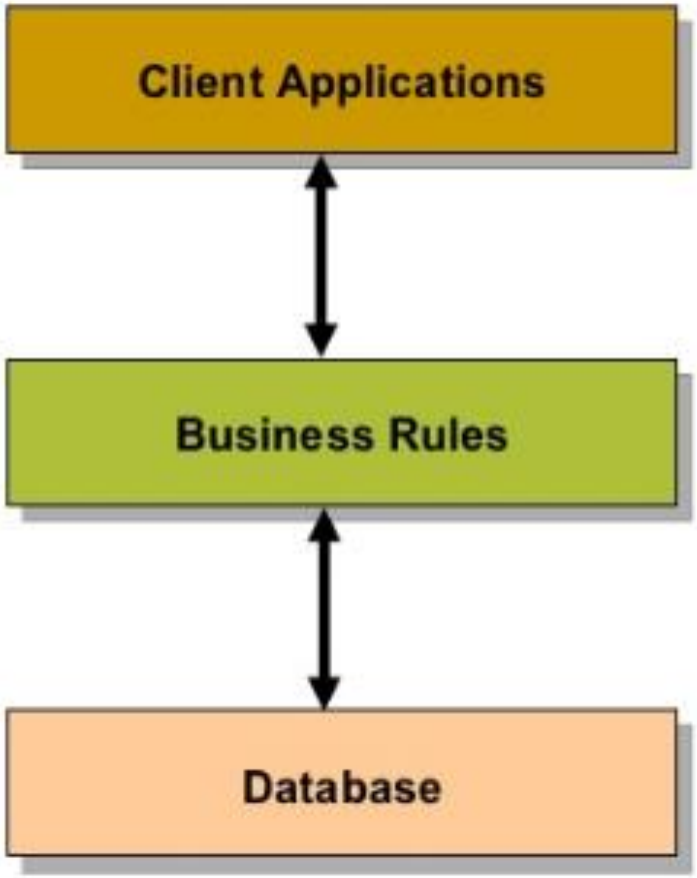
Forward Engineering (II)

- **Options for poorly designed source code**
  - Change necessary lines of source code
  - Understand bigger picture & use this to guide modifications
  - Redesign, recode, & test portions that need modification
  - Completely redesign, recode, and test system

Forward Engineering (III)

- **F.E. For Client/server architectures**
  - Distribution of resources.
  - Migrate application functionality to client.
  - New GUI interfaces are implemented at client.
  - Database functions are allocated at server.
  - Specialized functionality may remain at the server site.
  - New communications, security and control. requirement must be established at both the client & server.
- **Three layers of abstractions can be identified**

Forward Engineering (IV)



-Implement business functions that required by specific group of end-users

-Represents software at both the client and server.
-Software performs control and coordination tasks to ensure the transactions between client and server.

-Manages transactions and queries from server applications

*

Forward Engineering (V)

- **F.E. For Object-Oriented Architectures**
  - Existing software is reverse engineered so that appropriate data, functional and behavioral models can be created.
  - Data models are used to establish the basis for the definition og classes.
  - Then class hierarchies, object relationship models are defined.

Forward Engineering (VI)

- **Forward Engineering User interfaces**
  - Understand original interface & data
  - Remodel behavior of old interface into abstractions in context of GUI
  - Add improvements that make the mode of interaction more efficient.
  - Build and integrate new GUI.