SOFTWARE ENGINEERING (CA725)

Dr. Ghanshyam S. Bopche
Assistant Professor
Dept. of Computer Applications

November 19, 2021

SOFTWARE ENGINEERING (CA725)

Syllabus

- **Unit-I**: Introductory concepts, The evolving role of software, Its characteristics, components and applications, A layered technology, The software process, Software process models, Software Development Life cycle, Software process and project metrics, Measures, Metrics and Indicators, Ethics for software engineers.
- **Unit-II**: Software Project Planning, Project planning objectives, Project estimation, Decomposition techniques, Empirical estimation models, System Engineering, Risk management, Software contract management, Procurement Management.
- **Unit-III**: Analysis and Design, Design concept and Principles, Methods for traditional, Real time of object oriented systems, Comparisons, Metrics, Quality assurance.

SOFTWARE ENGINEERING (CA725) (cont.)

- **Unit-IV**: Testing fundamentals, Test case design, White box testing, Basis path testing, Control structure testing, Black box testing, Strategies: Unit testing, integration testing, Validation Testing, System testing, Art of debugging, Metrics, Testing tools
- **Unit-V**: Formal Methods, Clean-room Software Engineering, Software reuse, Re-engineering, Reverse Engineering, Standards for industry.
- **References**:
  1. Rajib Mall, "Fundamentals of Software Engineering", 4th Edition, PHI, 2014.
  2. Roger S. Pressman, "Software Engineering-A practitioner's approach", 7 th Edition, McGraw Hill, 2010.
  3. Ian Sommerville, "Software engineering", 10th Edition, Pearson education Asia, 2016.

SOFTWARE ENGINEERING (CA725) (cont.)

4. Pankaj Jalote, “An Integrated Approach to Software Engineering”, Springer Verlag, 1997.
5. James F Peters, Witold Pedrycz, “Software Engineering – An Engineering Approach”, John Wiley and Sons, 2000.
6. Ali Behforooz, Frederick J Hudson, “Software Engineering Fundamentals”, Oxford University Press, 2009.
7. Bob Emery , “Fundamentals of Contract and Commercial Management”,Van Haren Publishing, Zaltbommel, 2013

Software Testing Fundamentals

- Goal of software testing: finding errors.
- A good software test: high probability of finding errors.
- Developer should design and implement a computer based system or product with "testability" in mind.
- **Testability**: "How easily a computer program can be tested?"

Software Testing Fundamentals (cont.)

- **Characteristics of Testable Software**
  1. **Operability**: "The better it works, the more efficiently it can be tested."
     - If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests, allowing testing to progress without fits and starts.
  2. **Observability**" "What you see is what you test."
     - Inputs provided as part of testing produce distinct outputs.
     - System states and variables are visible or queriable during execution.
     - Incorrect output is easily identified.
     - Internal errors are automatically detected and reported.
     - Source code is accessible.

Software Testing Fundamentals (cont.)

3. **Controllability**: "The better we can control the software, the more the testing can be automated and optimized."
    - All possible outputs can be generated through some combination of input, and I/O formats are consistent and structured.
    - All code is executable through some combination of input.
    - Software and hardware states and variables can be controlled directly by the test engineer.
    - Test can be conveniently specified, automated, and reproduced.

Software Testing Fundamentals (cont.)

4. **Decomposability**: "By controlling the scope of testing, we can more quickly isolate problems and perform smarter testing."
   - The software system is built from independent modules that can be tested independently.
5. **Simplicity**: "The less there is to test, the more quickly we can test it."
   - The program should exhibit functional stability (e.g., the feature set is the minimum necessary to meet requirements); structural simplicity (e.g., architecture is modularized to limit the propagation of faults); and code simplicity (e.g., a coding standard is adopted for ease of inspection and maintenance).

Software Testing Fundamentals (cont.)

6. **Stability**: "The fewer the changes, the fewer the disruptions to testing."
    - Changes to the software are infrequent, controlled when they do occur, and do not invalidate existing tests.
    - The software recover well from failure.
7. **Understandability**: "The more information we have, the smarter we will test."
    - The architectural design and the dependencies between internal, external, and shared components are well understood.
    - Technical documentation is instantly accessible, well organized, specific and detailed, and accurate.
    - Changes to the design are communicated to tester.

Software Testing Fundamentals (cont.)

Test Characteristics

- A good test has a high probability of finding an error.
  - Classes of failure are probed.
- A good test is not redundant.
  - Testing time and resources are limited.
  - Every test should have a different purpose.
- A good test should be "best of breed".
  - In a group of tests, the test that has the highest likelihood of uncovering a whole class of errors should be used.
- A good test should be neither too simple nor too complex.

Internal and External Views of Testing

Ways of testing engineered product:

1. 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.
    - Takes an external view of a system.
    - Black-box approach of software testing.
    - Input/Output driven approach.
2. Knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh", i.e., internal operations are performed according to specifications and all internal components have been adequately exercised.
    - Requires an internal view of a system.
    - White-box approach of software testing.
    - Logic driven approach.

White-box Testing

- Also called as grey-box testing or glass-box testing.
- Uses the control structure described as a part of component-level design to derive test cases.
- A tester can derive test cases that
  - 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.

White-box Testing (cont.)

- **Types of White-box Testing**
  1. Basis Path Testing
     1.1 Flow Graph Approach
     1.2 Graph Matrices
  2. Control Structure Testing
     2.1 Condition Testing
     2.2 Data Flow Testing
     2.3 Loop Testing

White-box Testing (cont.)

Basis Path Testing

- Enables the test-case designer to derive a logical complexity measure of a procedural design and use 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.
- Approaches of Basis Path Testing
  1. Flow Graph Approach
  2. Graph Matrices

White-box Testing (cont.)

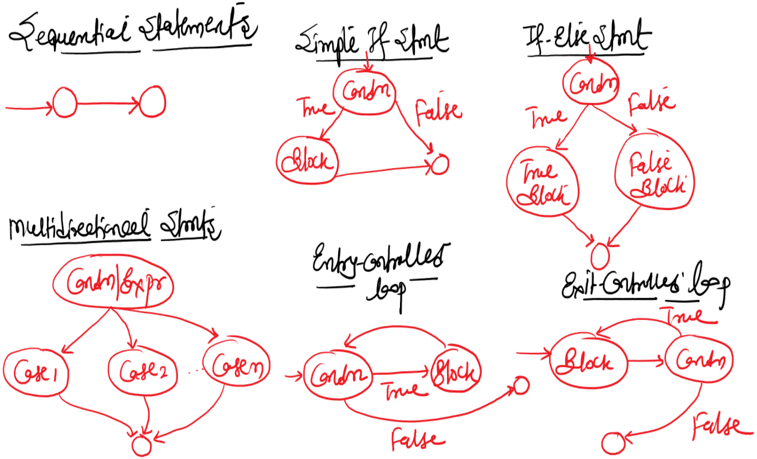
1) Flow Graph Approach

Steps to be followed

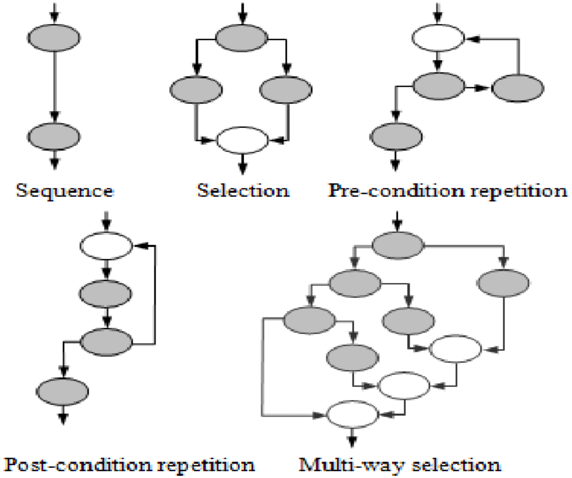
- Construct a flow graph for a given task.
- Determine cyclomatic complexity for the flow graph.
- Evaluate/identify independent paths for the constructed flow graph.
- Prepare/determine test cases equivalent to number of independent paths.

White-box Testing (cont.)

Flow Graph Notations



White-box Testing (cont.)



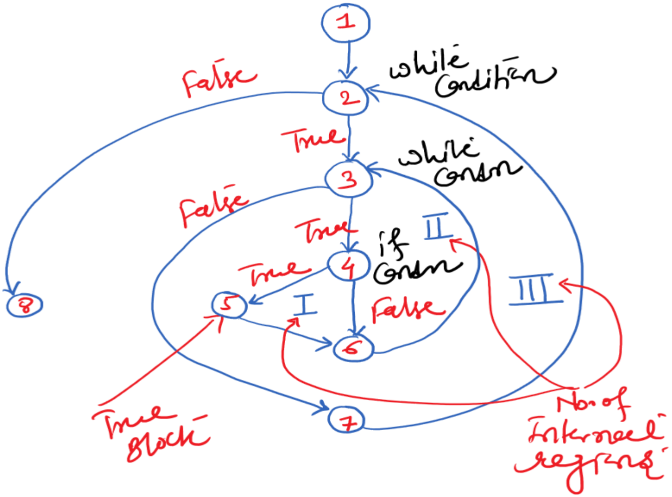
White-box Testing (cont.)

Case Study 1: Swapping of an array elements

```
i, j, N, A[100] : Integer
while(i $\leq$ N)
do
    while(j $\leq$ N)
    do
        if(A[i] $>$ A[j]) then
            swap(A[i], A[j])
        end if
    end do
end do
end
```

White-box Testing (cont.)

Step 1: Construction of Flow Graph



White-box Testing (cont.)

Step 2: Computing Cyclomatic Complexity
V(G) = E - N + 2
Here,

- E: number of edges in a flow graph.
- N: number of nodes in a flow graph.

V(G) = No. of internal regions + 1 External region
V(G) = P + 1
Here, P: number of predicates.

White-box Testing (cont.)

Step 3: Identifying independent paths

- 1 - 2 - 8
- 1 - 2 - 3 - 7 - 2 - 8
- 1 - 2 - 3 - 4 - 6 - 3 - 7 - 2 - 8
- 1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 2 - 8

Step 3: Prepare 4 test cases

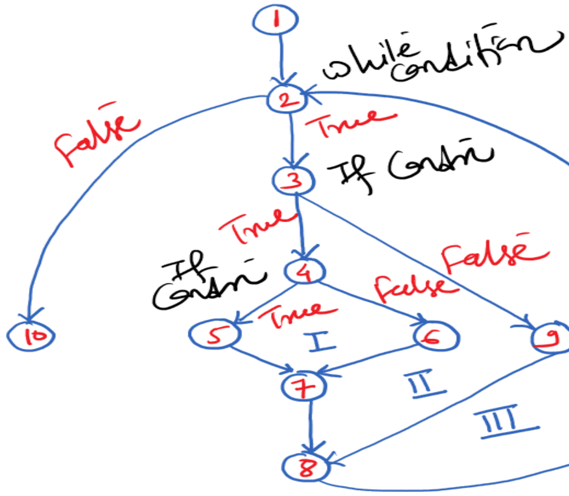
White-box Testing (cont.)

Case Study 2

```
Declaration
while(condition)
do
    if (condition) then
        if (condition) then
            True Block
        else
            False Block
    else
        False Block
    end if
end do
end
```

White-box Testing (cont.)

Step 1: Construction of Flow Graph



White-box Testing (cont.)

Step 2: Computing Cyclomatic Complexity
V(G) = E - N + 2
Here,

- E: number of edges in a flow graph.
- N: number of nodes in a flow graph.

V(G) = No. of internal regions + 1 External region
V(G) = P + 1
Here, P: number of predicates.

White-box Testing (cont.)

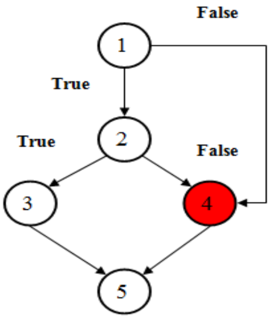
Step 3: Identifying independent paths

- 1 - 2 - 10
- 1 - 2 - 3 - 9 - 8 - 2 - 10
- 1 - 2 - 3 - 4 - 6 - 7 - 8 - 2 - 10
- 1 - 2 - 3 - 4 - 5 - 7 - 8 - 2 - 10

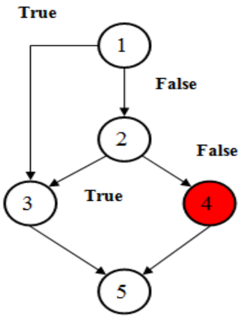
Step 3: Prepare 4 test cases

White-box Testing (cont.)

Compound Logic: && and ||



Logical AND (&&)



Logical OR (||)

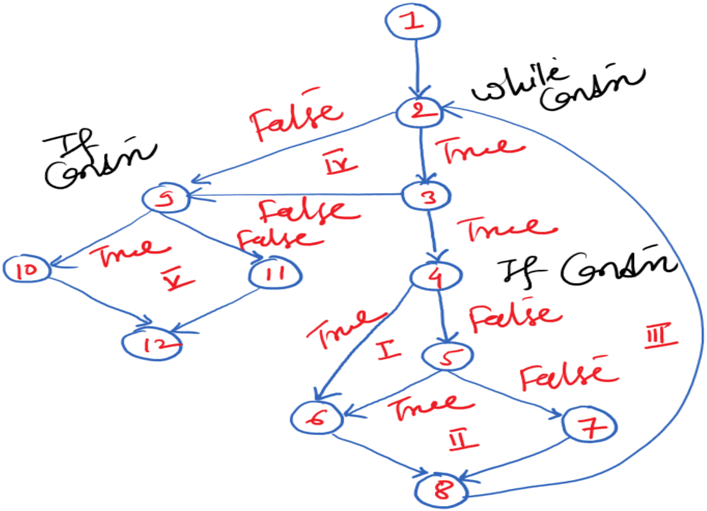
White-box Testing (cont.)

Case study 3

```
Declaration
while(condition1 && condition2)
do
    if (condition1 || condition2) then
        True Block
    else
        False Block
    end if
end do
if(condition) then
    True Block
else
    False Block
end if
end
```

White-box Testing (cont.)

Step 1: Construction of Flow Graph



White-box Testing (cont.)

Step 2: Computing Cyclomatic Complexity
V(G) = E - N + 2
Here,

- E: number of edges in a flow graph.
- N: number of nodes in a flow graph.

V(G) = No. of internal regions + 1 External region
V(G) = P + 1
Here, P: number of predicates.

White-box Testing (cont.)

Step 3: Identifying independent paths

- 1 - 2 - 9 - 10 - 12
- 1 - 2 - 9 - 11 - 12
- 1 - 2 - 3 - 9 - 10 - 12
- 1 - 2 - 3 - 9 - 11 - 12
- . . .

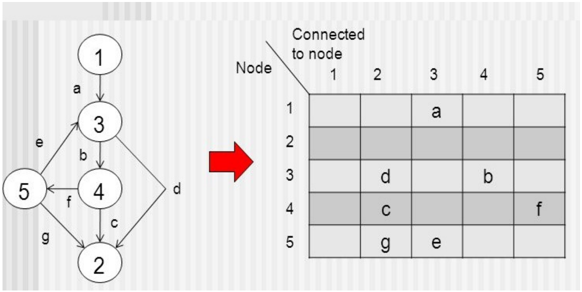
Step 3: Prepare 16 test cases

White-box Testing (cont.)

2) Graph Matrix Approach

- **Graph Matrix**
  - A square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on the flow graph.
  - Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
  - A tabular representation of a flow graph.

White-box Testing (cont.)

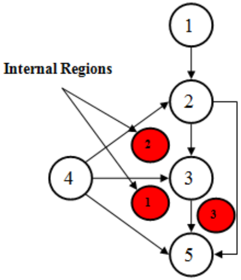


Node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Figure: Flow graph and graph matrix

White-box Testing (cont.)

Case Study 1



	1	2	3	4	5	
1		1				1 - 1 = 0
2			1		1	2 -1 = 1
3					1	1 - 1 = 0
4		1	1		1	3 - 1 = 2
5						-

3 + 1 = 4

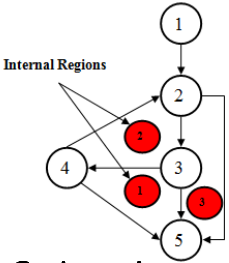
- **Cyclomatic complexity**: V(G) = E - N + 2 = 7 -5 + 2 = 4
- V(G) = P + 1 = 3 + 1 = 4
- **Independent Paths**

  1 - 2 - 5   1 - 2 - 3 - 5   4 - 2 - 5   4 - 2 - 3 - 5

  4 - 3 - 5   4 - 5

White-box Testing (cont.)

Case Studv 2



	1	2	3	4	5	
1		1				1 - 1 = 0
2			1		1	2 -1 = 1
3				1	1	2 - 1 = 1
4		1			1	2 - 1 = 1
5						-

3 + 1 = 4

- **Cyclomatic complexity**: V(G) = E - N + 2 = 7 -5 + 2 = 4
- V(G) = P + 1 = 3 + 1 = 4
- **Independent Paths**

  1 - 2 - 5

  1 - 2 - 3 - 5

  1 - 2 - 3 - 4 - 5

  1 - 2 - 3 - 4 - 2 - 5

  1 - 2 - 3 - 4 - 2 - 3 - 5

White-box Testing (cont.)

- **Reachability Measure**

  Reachability Measure $= \frac{\textit{Total number of paths for all nodes}}{\textit{Total number of nodes}}$

  - Paths for node 1: 1 (Self path)
  - Paths for node 2
    - 1 - 2
    - 1 - 2 - 3 - 4 - 2
  - Paths for node 3
    - 1 - 2 - 3
    - 1 - 2 - 3 - 4 - 2 - 3
  - Paths for node 4: 1 - 2 - 3 - 4

White-box Testing (cont.)

- Paths for node 5
  - 1 - 2 - 5
  - 1 - 2 - 3 - 5
  - 1 - 2 - 3 - 4 - 5
  - 1 - 2 - 3 - 4 - 2 - 5
  - 1 - 2 - 3 - 4 - 2 - 3 - 5
- Reachability Measure $= \frac{11}{5} = 2.2$

White-box Testing (cont.)

Control Structure Testing

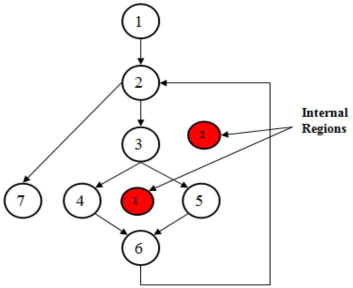
1. **Condition Testing**
   - A test case design method that exercises the logical conditions contained in a program module.
   - **Simple condition**
     - A boolean variable or a relational expression, possibly preceded with NOT ($\neg$) operator.
     - A relational expression takes the form
       $E_1 < relational - operator > E_2$
       where E_1 and E_2 are arithmetic expressions.
     - Number of test cases will be less.
   - **Composite condition** (a condition without relational expression)
     - Composed of two or more simple conditions, Boolean operators, and parentheses.
     - Boolean operators allowed in a compound condition include OR ($||$), AND ($\&\&$), and NOT ($\neg$).
     - Number of test cases will be more.

White-box Testing (cont.)

Case Study 1: **Simple condition**

```
Declaration
while(condition)
do
    if(condition) then
        True Block
    else
        False Block
    end if
end do
end
```

$\rightarrow$ **3 Test Cases**

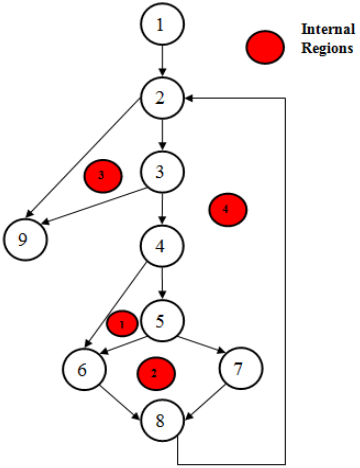


White-box Testing (cont.)

Case Study 2: Composite condition

```
Declaration
while(condition1 && condition2)
do
    if(condition1 || condition2)
then
        True Block
    else
        False Block
    end if
end do
end
```

→ **8 Test Cases**

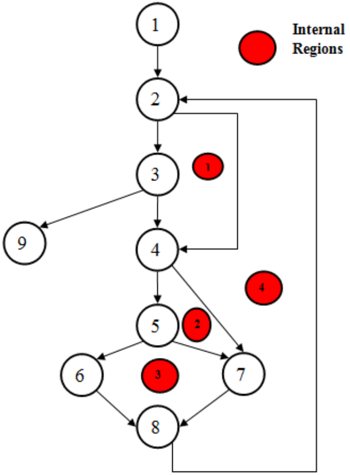


White-box Testing (cont.)

Case Study 3: **Composite condition**

```
Declaration
while(condition1 || condition2)
do
    if(condition1 && condition2)
then
        True Block
    else
        False Block
    end if
end do
end
```

→ **7 Test Cases**



White-box Testing (cont.)

2. **Data Flow Testing**
    - The data flow testing selects test paths of a program according to the locations of definitions and uses of variables in the program.
    - For a statement with **S** as its statement number,
      DEF(S) $= \{X \mid$ statement S contains a definition of X$\}$
      USE(S) $= \{X \mid$ statement S contains a use of X$\}$
    - If statement S is in loop or loop statement, its DEF set is empty and its USE set is based on the condition of statement S.
    - The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.

White-box Testing (cont.)

- A definition-use (DU) chain of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'.
- **Simple data flow testing strategy**: every DU chain be covered at least once.
- DU testing does not guarantee the coverage of all branches of a program.

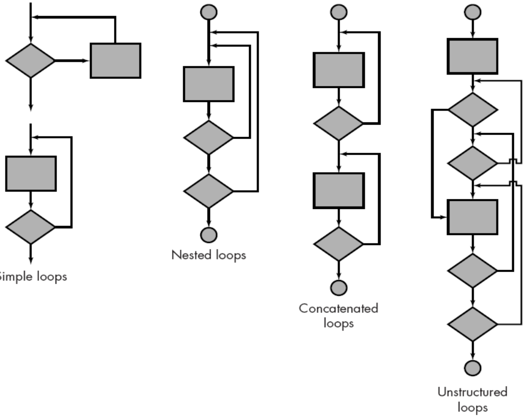
White-box Testing (cont.)

- Data flow testing make use of data coverage test cases.
- Consider a 32 bit Operating System (OS) and Four variables of the same data type, i.e., Data Type a, b, c, d
  - Minimum one test case for every variable.
  - For one variable maximum 2^{32} test cases $\rightarrow$ Impossible
  - For all variables maximum 2^{128} test cases $\rightarrow$ Impossible
  - **Case 1**: a, b, c $\rightarrow$ warning (2^{96} test cases, performance degrades at the rate of 2^{32}).
  - **Case 2**: a, b, c, d $\rightarrow$ Efficient (2^{128} test cases).
  - **Case 3**: a, b, c, d, e $\rightarrow$ Error (2^{160} test cases).

White-box Testing (cont.)

3. **Loop Testing**

   Loop testing focuses exclusively on the validity of loop constructs.



White-box Testing (cont.)

Classes of loops

3.1 Simple Loop

The following set of tests can be applied to simple loops, where 'n' is the maximum number of allowable passes through the loop.

- Skip the loop entirely.
- Only one pass through the loop.
- Two passes through the loop.
- 'm' passes through the loop where $m < n$.
- (n-1), n, (n+1) passes through the loop.

White-box Testing (cont.)

3.2 Nested Loops

- If the test approach of simple loop is extended to nested loop, the number of possible tests would grow geometrically as the level of nesting increases (impractical number of tests).
- **Approaches to reduce number of tests** (suggested by Beizer):

  → Start with the innermost loop. Set all other loops to minimum values.
  → Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add outer tests for out-of-range or excluded values.
  → Work outward, conducting tests for the next loop, but keeping all other outer loops at their minimum values and other nested loops to "typical" values.
  → Continue until all loops have been tested.

White-box Testing (cont.)

3.3 **Concatenated loops**

- Can be tested using the approach defined for simple loops, if each of the loops is independent of each other.
- If two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
- When the loops are not independent, the approach applied to nested loops is recommended.

3.4 **Unstructured loops**

- Whenever possible, unstructured loops should be redesigned to reflect the use of the structured programming constructs.

Black-box Testing

- I/O driven testing or functionality driven testing or behavioral testing.
- Enable test team to derive sets of input conditions that will fully exercise all functional requirements for a program.
- Complimentary approach (to white-box testing) that is likely to uncover a different class of errors than white-box testing.
- Evaluate functionality of the product based on inputs/outputs.
- Black-box testing attempts to find errors in the following categories:
    - Incorrect or missing functions,
    - Interface errors,
    - Errors in data structures or external database access,
    - Behavior or performance errors, and
    - Initialization and termination errors.

Black-box Testing (cont.)

- Unlike white-box testing, black-box testing tends to be applied during later stages of testing.
- Black-box testing focuses on information domain rather than control structures.
- Black-box tests are designed to answer following questions:
  - How is functional validity tested?
  - How are system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?

Black-box Testing (cont.)

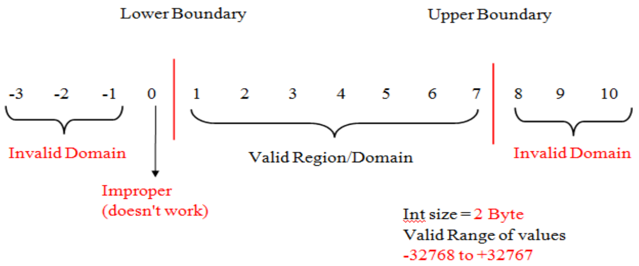
Black-box Testing Techniques

1. Equivalence partitioning testing
2. Boundary value analysis (BVA)
3. Logic coverage criteria
4. Error guessing
5. Comparison testing

Black-box Testing (cont.)

1) Equivalence partitioning testing

- The inputs and outputs are categorized into classes i.e., valid class and invalid class or domain.
- Test cases are prepared by taking few samples from each class or domain.
- **Case Study**: factorial of a number (n!).



Black-box Testing (cont.)

2) Boundary value analysis (BVA)

- Most of the bugs arises at boundary point such that boundary should be evaluated properly.
- It determines the limitation of the product i.e., how exactly the system behaves within the restricted region as well as beyond the restricted region such that test cases should be prepared with respect to lower boundary as well as upper boundary.

Black-box Testing (cont.)

3) Logic coverage criteria

- It is a criteria each program should fulfill for obtaining desired output for a given input.
- As per the criteria, every program should execute
  - every statement at least once (statement coverage),
  - every branch at least once (branch coverage), and
  - every path at least once (path coverage),

  during the course of execution.
- Criteria for flow graph (Exhaustive/complete)
  - Edges and nodes
  - Predicates
  - Regions

Black-box Testing (cont.)

4) Error Guessing

- An ad-hoc (temporary) approach used for preparing alerts prior to the release of the product.
- These alerts are help facility or aid for the customer to evaluate the mistakes.
- **Sample reasons for the bug**:
  - If mandatory fields are not filled in a given web-based application.
  - If a program attempts to write a file which doesn't have permission (or) if that file is deleted by someone (or) file is already full.
  - If a person attempts to withdraw an amount from ATM more than the amount available.

Black-box Testing (cont.)

5) Comparison Testing

- It is used for preparing test cases for real-time applications.
- In a real-time application during the development a software company assigns more than 1 team (say n teams).
- Each team develop a version of the product, which is evaluated using test cases prepared based on comparison testing.
- If all versions produce same desired output then any one version will be released to customer otherwise all versions are reevaluated.