CHƯƠNG 5
TEST TECHIQUES

BỘ MÔN CÔNG NGHỆ PHẦN MỀM

KHOA CNTT-ĐH NGOẠI NGỮ TIN HỌC TPHCM

OBJECTIVES

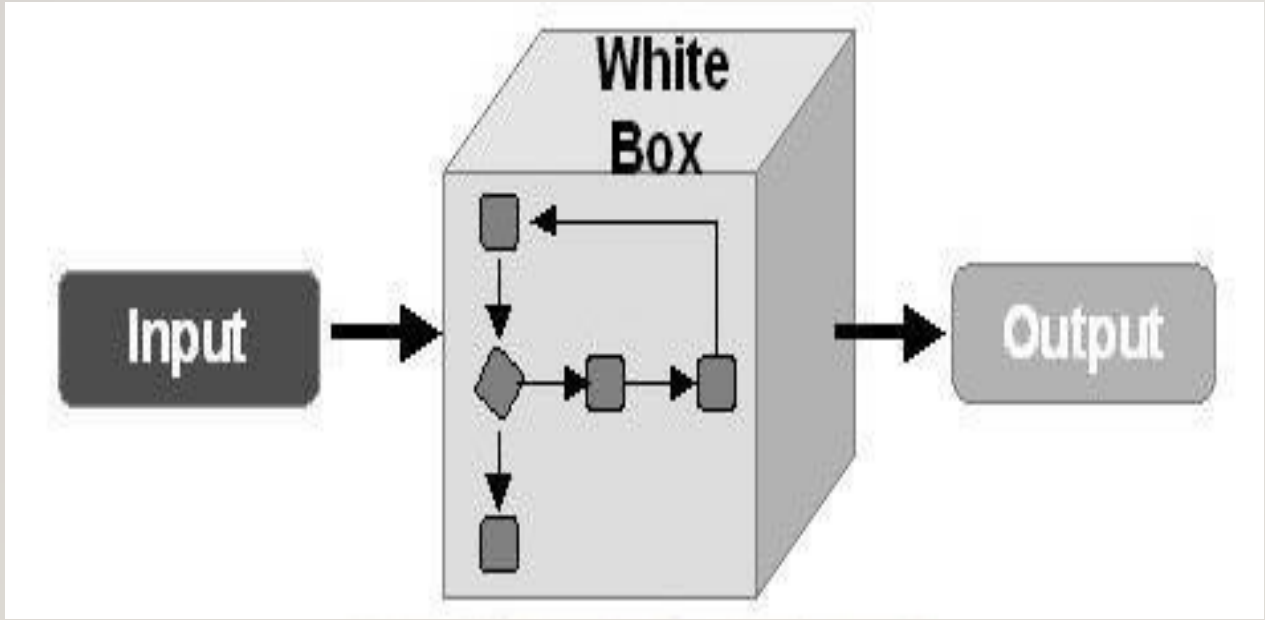
Help students understand and apply:

- Black box and white box testing techniques
- The importance of each type
- When, where, what application, by whom for each technique
- Apply the methods of each technique in software testing

CONTENTS

1. Test coverage
2. White box testing technique
   - Statement Coverage
   - Decision Coverage
   - Branch Coverage
   - Toggle Coverage
   - FSM Coverage
   - CONTROL FLOW TESTING
3. Black box testing technique
   - Equivalence Partitioning
   - Boundary Value Analysis
   - Cause-Effect Graphing:
   - State Transition Testing

WHITEBOX TESTING



WHITEBOX TESTING

White box testing (also known as Clear Box Testing, Open Box Testing, Glass Box Testing, Transparent Box Testing, Code-Based Testing or Structural Testing) is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester. The tester chooses inputs to exercise paths through the code and determines the appropriate outputs. Programming know-how and the implementation knowledge is essential. White box testing is testing beyond the user interface and into the nitty-gritty of a system.

Definition by ISTQB

- **white-box testing:** Testing based on an analysis of the internal structure of the component or system.
- **white-box test design technique:** Procedure to derive and/or select test cases based on an analysis of the internal structure of a component or system.

WHITEBOX TESTING

White Box Testing level is applicable to the following levels of software testing:

- Unit Testing: For testing paths within a unit.
- Integration Testing: For testing paths between units.
- System Testing: For testing paths between subsystems.

However, it is mainly applied to Unit Testing.

Advantages

- Testing can be commenced at an earlier stage. One need not wait for the GUI to be available.
- Testing is more thorough, with the possibility of covering most paths.

Disadvantages

- Since tests can be very complex, highly skilled resources are required, with a thorough knowledge of programming and implementation.
- Test script maintenance can be a burden if the implementation changes too frequently.
- Since this method of testing is closely tied to the application being tested, tools to cater to every kind of implementation/platform may not be readily available.

WHITE BOX TESTING

Code coverage

- Code coverage is a measure which describes the degree of which the source code of the program has been tested. It is one form of white box testing which finds the areas of the program not exercised by a set of test cases. It also creates some test cases to increase coverage and determining a quantitative measure of code coverage.
- In most cases, code coverage system gathers information about the running program. It also combines that with source code information to generate a report about the test suite's code coverage.

WHITE BOX TESTING

Basic coverage criteria

- **Function coverage** – has each function (or subroutine) in the program been called?
- **Statement coverage** – has each statement in the program been executed?
- **Edge coverage** – has every edge in the Control flow graph been executed?
- **Branch coverage** – has each branch (also called DD-path) of each control structure (such as in *if* and *case* statements) been executed? For example, given an *if* statement, have both the true and false branches been executed? This is a subset of **edge coverage.**
- **Condition coverage** (or predicate coverage) – has each Boolean sub-expression evaluated both to true and false?

WHITE BOX TESTING

Code coverage Methods

1. Statement Coverage
2. Decision Coverage
3. Branch Coverage
4. Toggle Coverage
5. FSM Coverage

WHITE BOX TESTING

1. Statement Coverage

Statement coverage is a white box test design technique which involves execution of all the executable statements in the source code at least once. It is used to calculate and measure the number of statements in the source code which can be executed given the requirements.

- Statement coverage is used to derive scenario based upon the structure of the code under test.

$$\text{Statement Coverage} = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

WHITE BOX TESTING

1. Statement Coverage

Ex

```
Prints (int a, int b) {                          ------------  Printsum is a functi
on
    int result = a+ b;
    If (result> 0)
        Print ("Positive", result)
    Else
        Print ("Negative", result)
    }                                           -----------   End of the source cod
e
```

Scenerio 1:

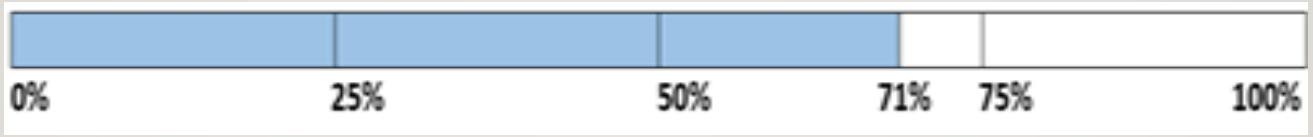
If a= 3, b = 9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
8
```

WHITE BOX TESTING

1. Statement Coverage

- The statements marked in yellow color are those which are executed as per the scenario
- Number of executed statements = 5, Total number of statements = 7
- Statement Coverage: 5/7 = 71%



Scenario 2:

- If a = -3, b = -9

```
1  Prints (int a, int b) {
2  int result = a+ b;
3  If (result> 0)
4      Print ("Positive", result)
5  Else
6      Print ("Negative", result)
7  }
```

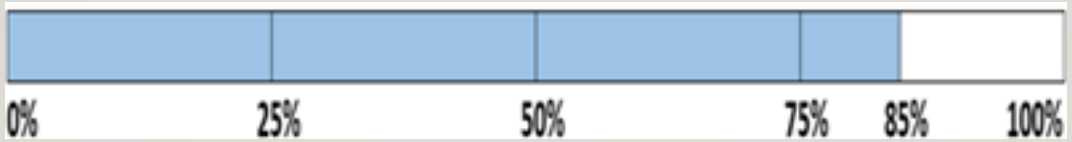
WHITE BOX TESTING

1. Statement Coverage

- The statements marked in yellow color are those which are executed as per the scenario.
- Number of executed statements = 6
- Total number of statements = 7

$$\text{Statement Coverage} = \frac{Number\ of\ executed\ statments}{Total\ number\ of\ statments}$$

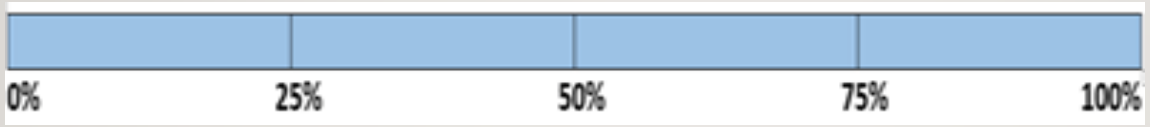
- Statement Coverage: 6/7 = 85%



WHITE BOX TESTING

1. Statement Coverage

~~But overall if you see, all the statements are being covered by 2nd scenario's considered. So~~ we can conclude that overall statement coverage is 100%.



- **What is covered by Statement Coverage?**
  - Unused Statements
  - Dead Code
  - Unused Branches
  - Missing Statements

WHITE BOX TESTING

2. Decision Coverage

Decision coverage reports the true or false outcomes of each Boolean expression. In this coverage, expressions can sometimes get complicated. Therefore, it is very hard to achieve 100% coverage.

That's why there are many different methods of reporting this metric. All these methods focus on covering the most important combinations. It is very much similar to decision coverage, but it offers better sensitivity to control flow.

$$Decision\ Coverage = \frac{Number\ of\ Decision\ Outcomes\ Excercised}{Total\ Number\ of\ Decision\ Outcomes}$$

WHITE BOX TESTING

2. Decision Coverage

Ex

Scenario 1:

- Value of a is 2
- The code highlighted in yellow will be executed. Here the "No" outcome of the decision If (a>5) is checked.
- Decision Coverage = 50%

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```

```
1  Demo(int a) {
2        If (a> 5)
3           a=a*3
4        Print (a)
5  }
```

WHITE BOX TESTING

2. Decision Coverage

Scenario 2:

- Value of a is 6
- The code highlighted in yellow will be executed. Here the "Yes" outcome of the decision If (a>5) is checked.
- Decision Coverage = 50%

```
1  Demo(int a) {
2        If (a> 5)
3           a=a*3
4        Print (a)
5  }
```

Test Case	Value of A	Output	Decision Coverage
1	2	2	50%
2	6	18	50%

WHITE BOX TESTING

3. Branch Coverage

- In the branch coverage, every outcome from a code module is tested. For example, if the outcomes are binary, you need to test both True and False outcomes.
- It helps you to ensure that every possible branch from each decision condition is executed at least a single time.
- By using Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out which is sections of code don't have any branches.
- The formula to calculate Branch Coverage:

$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

WHITE BOX TESTING

3. Branch Coverage

- In the branch coverage, every outcome from a code module is tested. For example, if the outcomes are binary, you need to test both True and False outcomes.
- It helps you to ensure that every possible branch from each decision condition is executed at least a single time.
- By using Branch coverage method, you can also measure the fraction of independent code segments. It also helps you to find out which is sections of code don't have any branches.
- The formula to calculate Branch Coverage:

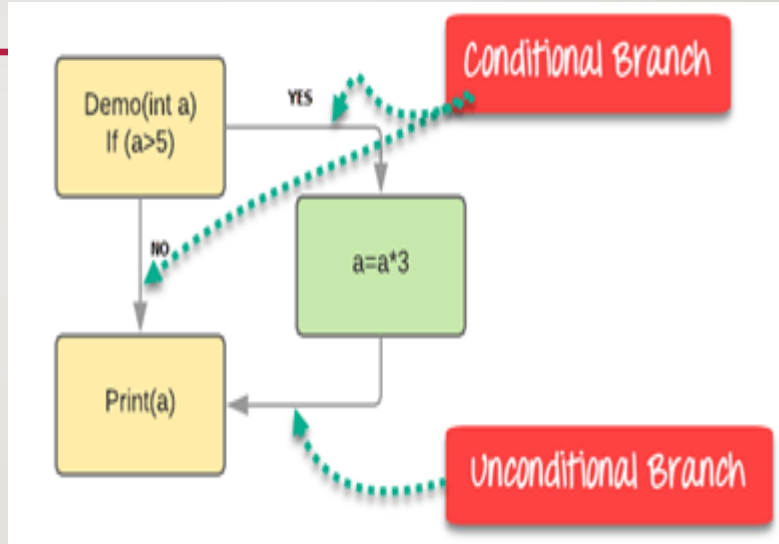
$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

WHITE BOX TESTING

3. Branch Coverage

Ex

```
Demo(int a) {
     If (a> 5)
        a=a*3
     Print (a)
    }
```



Branch Coverage will consider unconditional branch as well

Test Case	Value of A	Output	Decision Coverage	Branch Coverage
1	2	2	50%	**33%**
2	6	18	50%	**67%**

WHITE BOX TESTING

3. Branch Coverage

Advantages of Branch coverage

- Allows you to validate-all the branches in the code
- Helps you to ensure that no branched lead to any abnormality of the program's operation
- Branch coverage method removes issues which happen because of statement coverage testing
- Allows you to find those areas which are not tested by other testing methods
- It allows you to find a quantitative measure of code coverage
- Branch coverage ignores branches inside the Boolean expressions

WHITE BOX TESTING

3. Branch Coverage

- Conditional coverage or expression coverage will reveal how the variables or subexpressions in the conditional statement are evaluated. In this coverage expressions with logical operands are only considered.
- For example, if an expression has Boolean operations like AND, OR, XOR, which indicated total possibilities.
- Conditional coverage offers better sensitivity to the control flow than decision coverage. Condition coverage does not give a guarantee about full decision coverage
- The formula to calculate Condition Coverage:

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

WHITE BOX TESTING

3. Branch Coverage

Ex:

```
1   IF (x < y) AND (a>b) THEN
```

- For the above expression, we have 4 possible combinations
- TT
- FF
- TF
- FT

Consider the following input			
X=3 Y=4	(x<y)	TRUE	Condition Coverage is ¼ = 25%
A=3 B=4	(a>b)	FALSE	

WHITE BOX TESTING

4. Toggle Coverage

Toggle Coverage is basically a check which tells the percentage of the I/Os of your module toggling with your test suite. If your test suite is exercising all the boundary signals of your module and none of the I/Os are tied or left dangling, then your toggle coverage will come up to be 100%.
Toggle coverage is generally needed in case of system level verification of your module. When your module is hooked up in the chip and you need to ensure that all the connections of the module within the chip are ok. This is basically to put focus on interface checking of your module in the chip.

Toggle coverage is important for module internal verification.
the cases satistified line coverage might not for toggle coverage, it implied the uncoverage is not triggered by pattern then you lost 50% coverage on the signal functionally.
ideally all the signals should be toggled by test-cases.

WHITE BOX TESTING

5. Finite State Machine Coverage

Finite state machine coverage is certainly the most complex type of code coverage method. This is because it works on the behavior of the design. In this coverage method, you need to look for how many time-specific states are visited, transited. It also checks how many sequences are included in a finite state machine.

In order to select a coverage method, the tester needs to check that the

- code under test has single or multiple undiscovered defects
- cost of the potential penalty
- cost of lost reputation
- cost of lost sale, etc.

WHITE BOX TESTING

CONTROL FLOW TESTING

- Control-flow testing is most applicable to new software for unit testing.
- Control-flow testing assumptions:
  - specifications are correct
  - data is defined and accessed properly
  - there are no bugs other than those that affect control flow
- Structured and OO languages reduce the number of control-flow bugs.

WHITE BOX TESTING

CONTROL FLOW TESTING

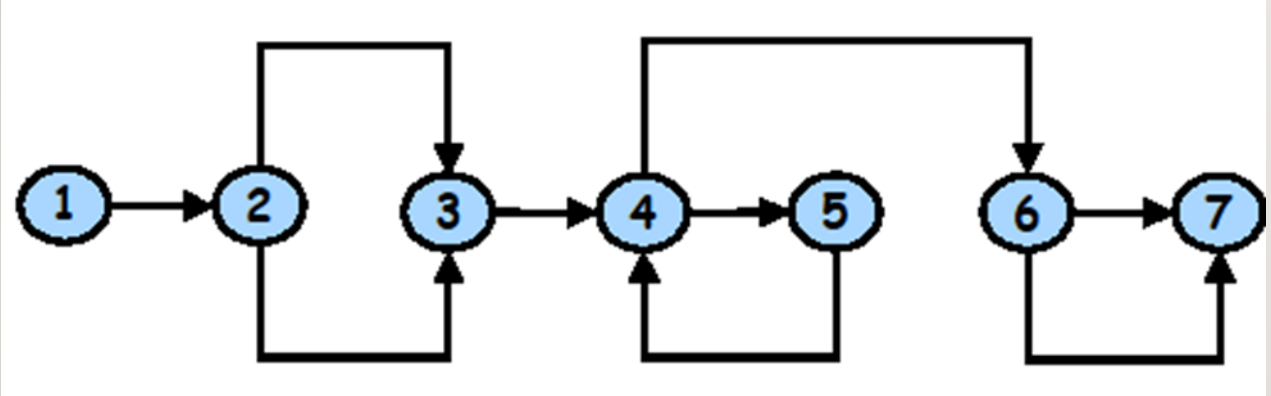
Control Flowgraphs

- The control flowgraph is a graphical representation of a program's control structure.
- Flowgraphs Consist of Three Primitives
  - A **decision** is a program point at which the control can diverge. (*e.g.*, if and case statements).
  - A **junction** is a program point where the control flow can merge. (*e.g.*, end if, end loop, goto label)
  - A **process block** is a sequence of program statements uninterrupted by either decisions or junctions. (*i.e.*, straight-line code).
    - A process has one entry and one exit.
    - A program does not jump into or out of a process.

WHITE BOX TESTING

CONTROL FLOW TESTING

Exponentiation Algorithm

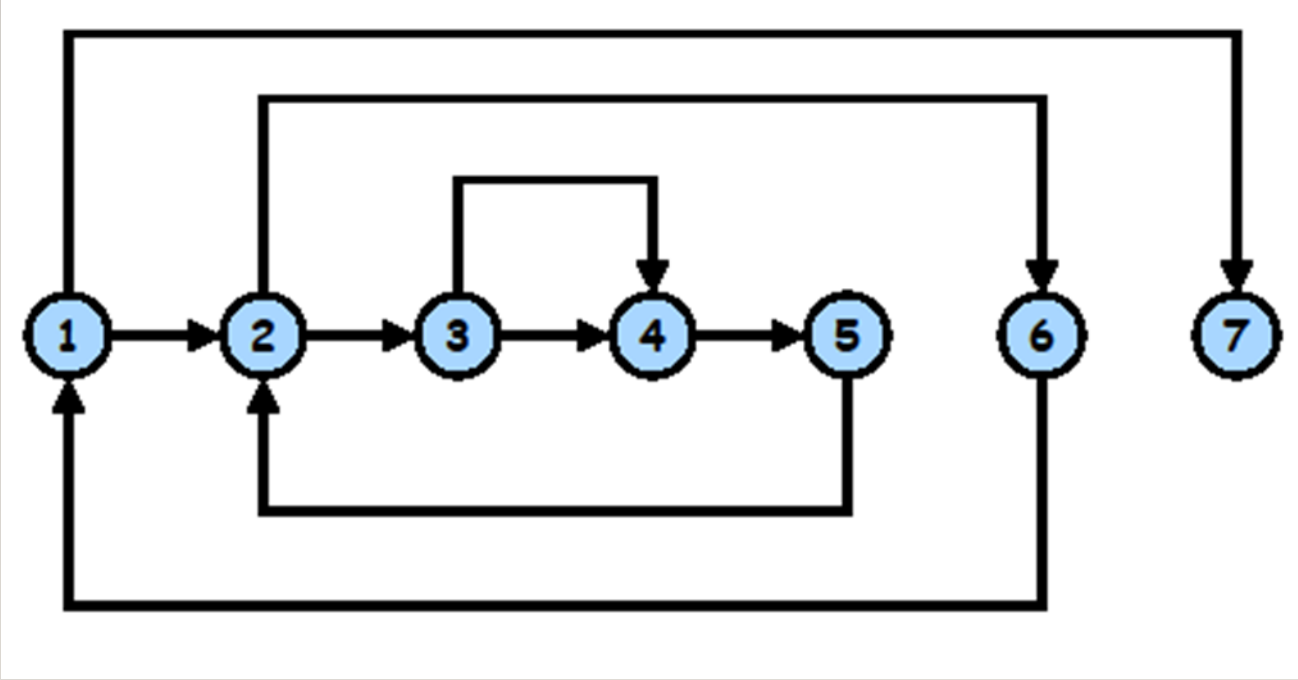



```
1  scanf("%d %d",&x, &y);
2  if (y < 0)
            pow = -y;
   else
            pow = y;
3  z = 1.0;
4  while (pow != 0) {
            z = z * x;
            pow = pow - 1;
5  }
6  if (y < 0)
            z = 1.0 / z;
7  printf ("%f",z);
```

WHITE BOX TESTING

CONTROL FLOW TESTING

Bubble Sort Algorithm




```
1     for (j=1; j<N; j++) {
            last = N - j + 1;
2           for (k=1; k<last; k++) {
3                 if (list[k] > list[k+1]) {
                    temp = list[k];
                    list[k] = list[k+1];
                    list[k+1] = temp;
4                   }
5           }
6     }
7     print("Done\n");
```

WHITE BOX TESTING

CONTROL FLOW TESTING

Control-flow Testing Criteria

- 3 testing criteria:
  - **Path Testing:**
    - 100% path coverage
    - Execute all possible control flow paths through the program.
  - **Statement Testing:**
    - 100% statement coverage.
    - Execute all statements in a program at least once under some test.
  - **Branch Testing:**
    - 100% branch coverage.
    - Execute enough tests to assure that every branch alternative has been exercised at least once under some test.

WHITEBOX TESTING

CONTROL FLOW TESTING

Two Detailed Examples of Control-flow Testing

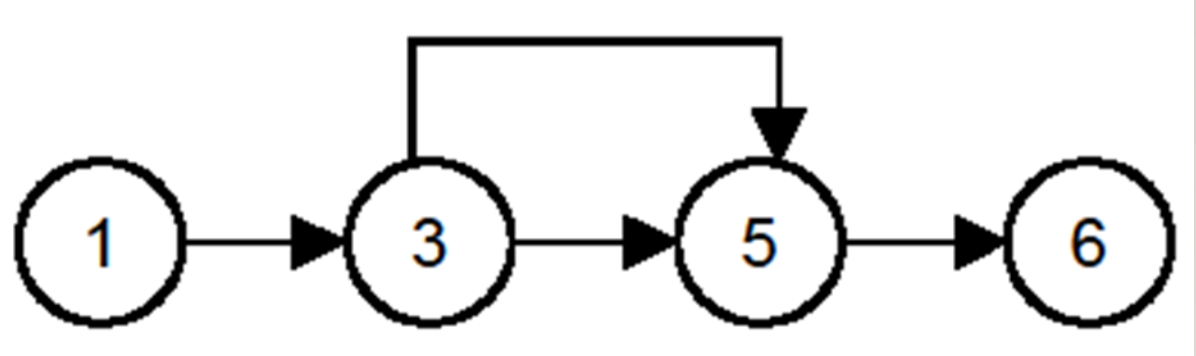
Using Control-flow Testing to Test Function ABS

- Consider the following function:

```
/* ABS
   This program function returns the absolute value of the integer
    passed to the function as a parameter.
   INPUT: An integer.
   OUTPUT: The absolute value if the input integer.
*/

1       int ABS(int x)
2       {
3               if (x < 0)
4                       x = -x;
5               return x;
6       }
```

The Flowgraph for ABS

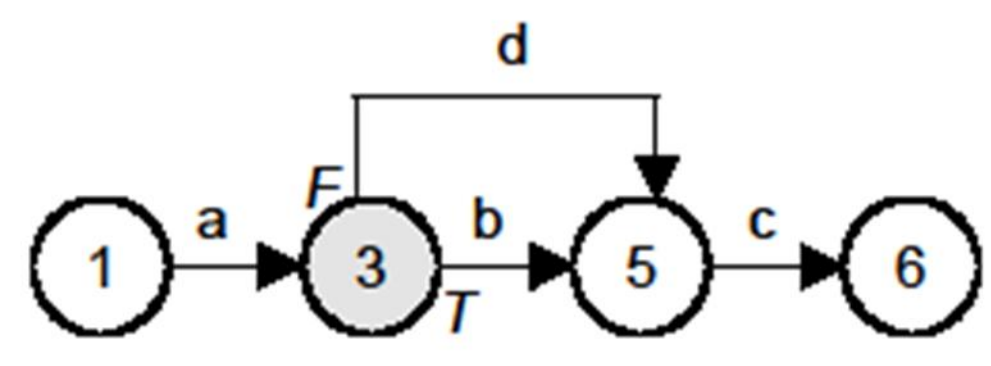


WHITE BOX TESTING

CONTROL FLOW TESTING

Test Cases to Satisfy Path Coverage for ABS

ABS takes as its input any integer. There are many integers (depending on the maximum size of an integer for the language) that could be input to *ABS* making it impractical to test all possible inputs to ABS.



PATHS	PROCESS LINKS				TEST CASES	
	a	b	c	d	INPUT	OUTPUT
abc	√	√	√		A Negative Integer, x	-x
adc	√		√	√	A Positive Integer, x	x

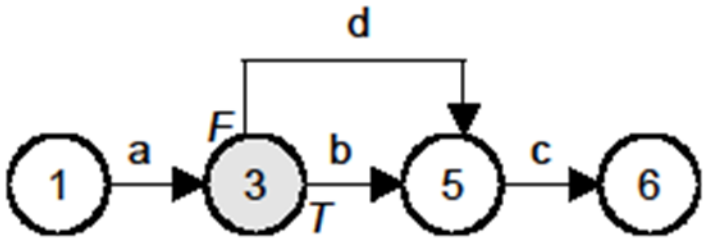
Test Cases to Satisfy Statement Testing Coverage for ABS

WHITEBOX TESTING

CONTROL FLOW TESTING

Test Cases to Satisfy Branch Testing Coverage for ABS

PATHS	DECISIONS	TEST CASES	
		INPUT	OUTPUT
abc	T	A Negative Integer, x	-x
adc	F	A Positive Integer, x	x



WHITE BOX TESTING

CONTROL FLOW TESTING

Example: Using Control-flow Testing to Test Program COUNT

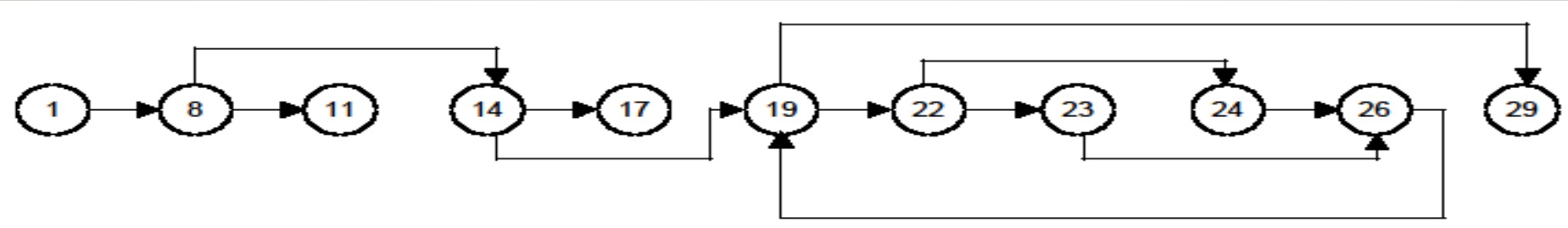
```
/* COUNT
   This program counts the number of characters and lines in a text file.
   INPUT: Text File
   OUTPUT: Number of characters and number of lines.
*/
1 main(int argc, char *argv[])
2 {
3   int numChars = 0;
4   int numLines = 0;
5   char chr;
6   FILE *fp = NULL;
8   if (argc < 2)
9   {
10     printf("\nUsage: %s <filename>", argv[0]);
11     return (-1);
12  }
13  fp = fopen(argv[1], "r");
14  if (fp == NULL)
15  {
16     perror(argv[1]); /* display error message */
17     return (-2);
18  }
19  while (!feof(fp))
20  {
21     chr = getc(fp); /* read character */
22     if (chr == '\n') /* if carriage return */
23        ++numLines;
24     else
25        ++numChars;
26  }
27  printf("\nNumber of characters = %d", numChars);
28  printf("\nNumber of lines = %d", numLines);
29 }
```

WHITEBOX TESTING

CONTROL FLOW TESTING

The Flowgraph for COUNT

The junction at line 12 and line 18 are not needed because if you are at these lines then you must also be at line 14 and 19 respectively.



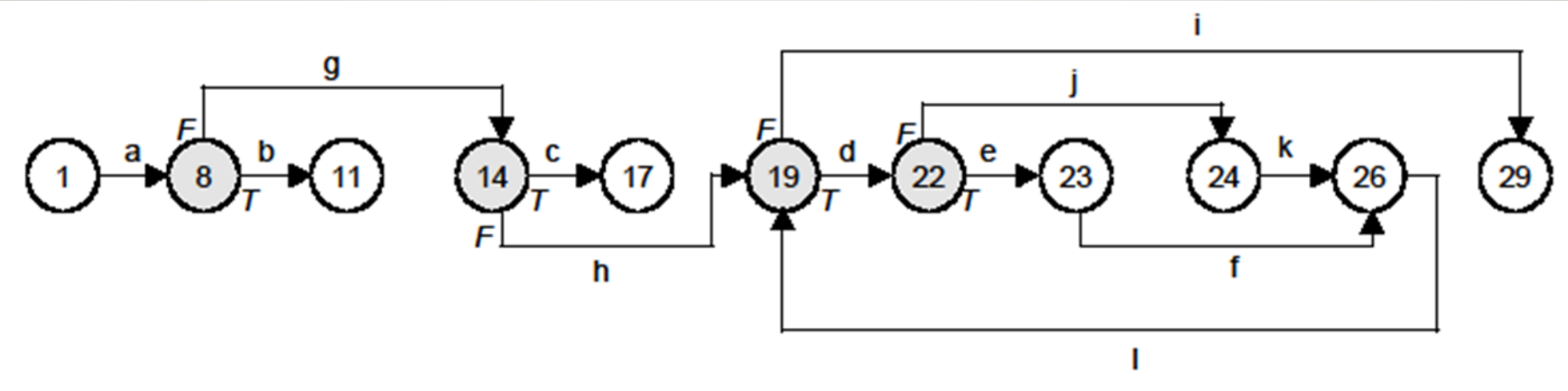
Test Cases to Satisfy Path Coverage for COUNT

Complete path testing of *COUNT* is impossible because there are an infinite number of distinct text files that may be used as inputs to *COUNT*

WHITEBOX TESTING

CONTROL FLOW TESTING

Test Cases to Satisfy **Statement Testing** Coverage for COUNT



WHITE BOX TESTING

CONTROL FLOW TESTING

Test Cases to Satisfy **Statement Testing** Coverage for COUNT

PATHS	PROCESS LINKS												TEST CASES	
	a	b	c	d	e	f	g	h	i	j	k	l	INPUT	OUTPUT
ab	√	√											None	"Usage: COUNT <filename>"
agc	√		√				√						Invalid Input Filename	Error Message
aghdj kli	√			√			√	√	√	√	√	√	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghd efli	√			√	√	√	√	√	√			√	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

WHITEBOX TESTING

CONTROL FLOW TESTING

Test Cases to Satisfy **Branch Testing** Coverage for COUNT

PATHS	DECISIONS				TEST CASES	
	8	14	19	22	INPUT	OUTPUT
ab	T				None	"Usage: COUNT <filename>"
agc	F	T			Invalid Input Filename	Error Message
aghdjkli	F	F	T, F	F	Input File with one character and no Carriage Return at the end of the line	Number of characters = 1 Number of lines = 0
aghdefli	F	F	T, F	T	Input file with no characters and one carriage return	Number of characters = 0 Number of lines = 1

WHITE BOX TESTING

CONTROL FLOW TESTING

Examples: Code segments in Canonical loop structures

For Loop

```
buffer= new char[nchar + 1];                                   A
for (n=0; n< nchars; ++n) {                                    B, D
        buffer[n] = newChar;                                   C
}
```

While Loop

```
buffer= new char[nchar + 1];                                   A
int n =0;
while ( n< nchars) {                                           B
        buffer[n] = newChar;                                   C
        ++n;
}
```

Until Loop

```
buffer= new char[nchar + 1];                                   A
int n =0;
do  {                                                          B
        buffer[n] = newChar;
        ++n;
}
While (n < nchars);                                            C
```

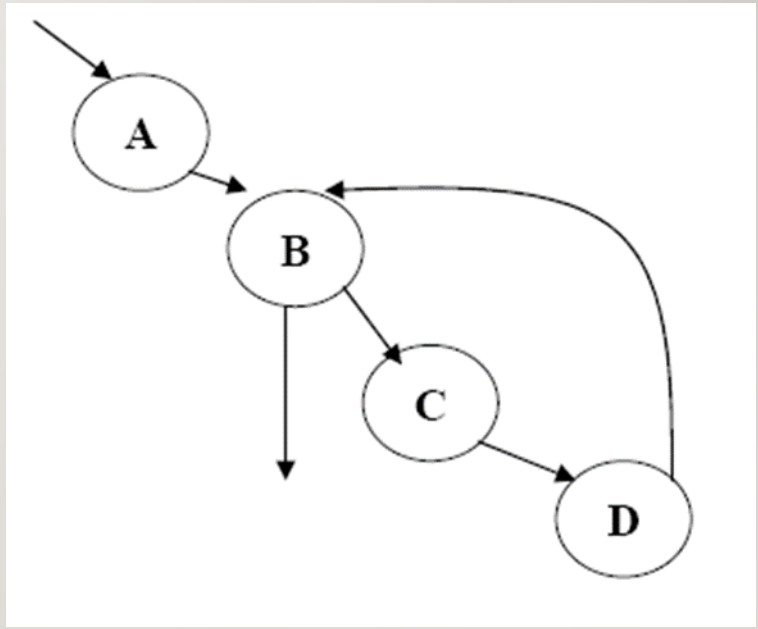
WHITEBOX TESTING

CONTROL FLOW TESTING

Flow graphs for canonical loop structures

For Loop

```
buffer= new char[nchar + 1];                          A
for (n=0; n< nchars; ++n) {                           B, D
        buffer[n] = newChar;                          C
}
```



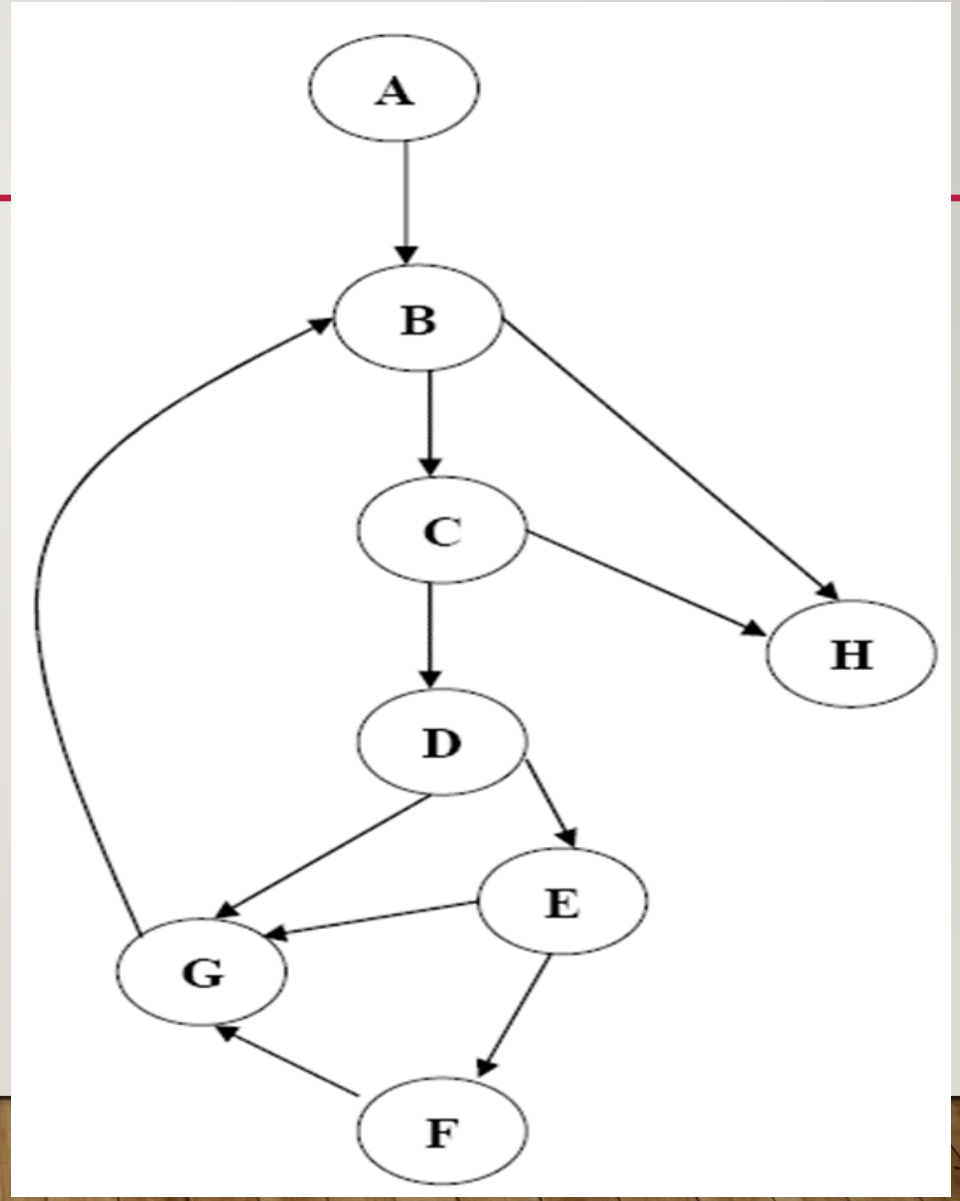
WHITE BOX TESTING

Example: CFG for a method

```
public class Authenticator {
   final int MAX = 10000;
   String [] userids = new String [MAX];
   String [] passwords = new String [MAX];
                …
```

public boolean verify (String uid, String pwd) { boolean result =false; int i = 0;		A
while ((result ==false)	&& (i < MAX)) {	B, C
if ((userids[i]==uid)	&& (passwords[i] == pwd))	D, E
result=true;		F
++i; }		G
return result; }		H

```
            //…Other methods
} //Class end
```

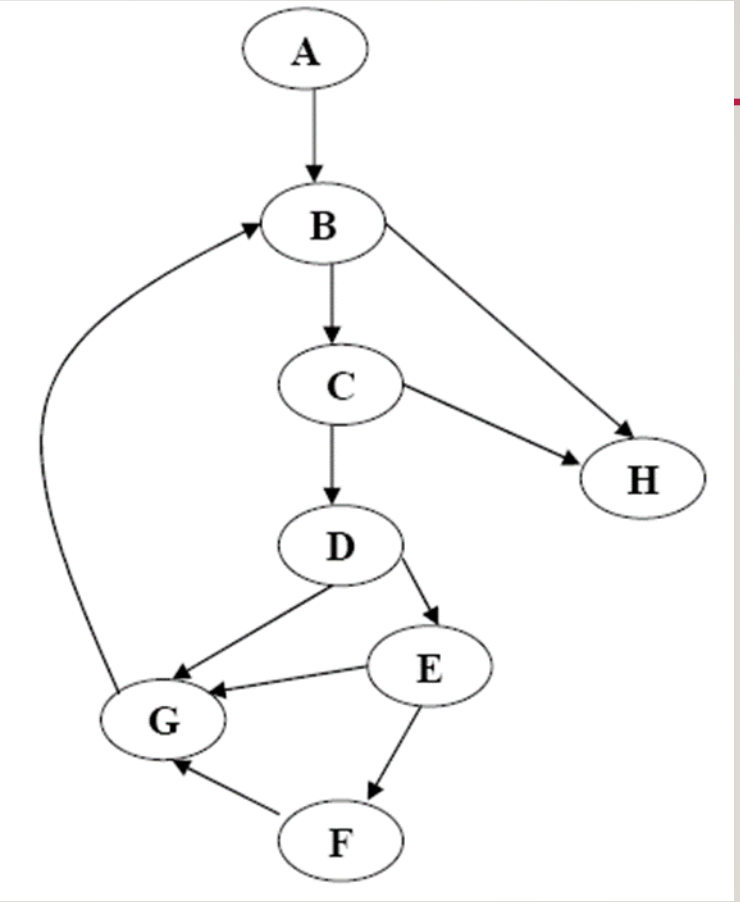


WHITEBOX TESTING

CONTROL FLOW TESTING

Examples of Entry/Exit Paths:

-ABH
-ABCH
-A(BCDEFG)*BH
-A(BCDEG)*BH
-A(BCDG)*BH
-A(BCDG)*BCH

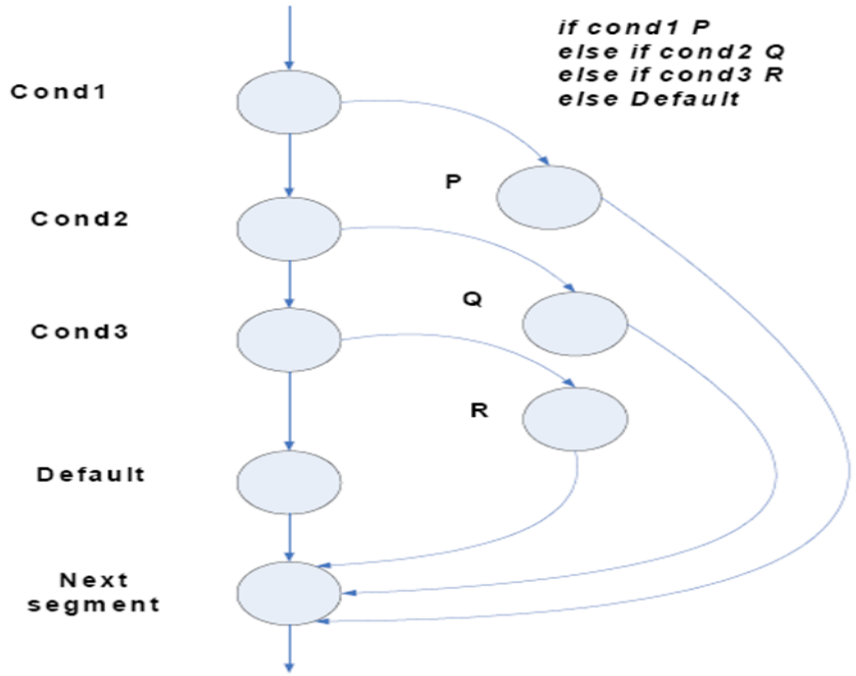


WHITE BOX TESTING

CONTROL FLOW TESTING

Case and Multiple-if Statements

-Case and multiple-if statements are modeled by specifying a separate node for each predicate, each conditional action, and the default action.

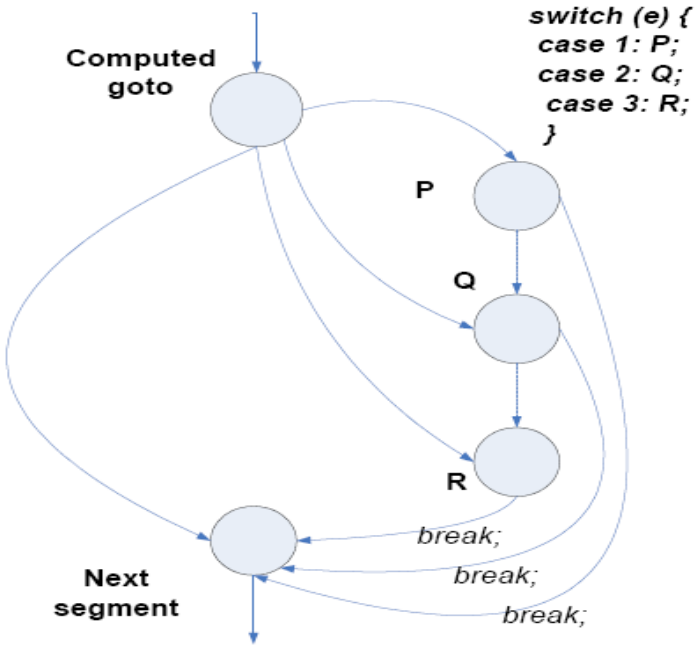


WHITE BOX TESTING

CONTROL FLOW TESTING

Switch Statements

-Switch statements are modeled by specifying a node for the switch expression, and a separate node for each action.



WHITEBOX TESTING

CONTROL FLOW TESTING

Example: Binary Search Routine

Code

```
Class BinSearch {
        public static void search(int key, int[] elemArray, Result r) {
                int bottom = 0;
                int top = elemArray.length – 1;
                int mid;
                r.found = false; r.index=-1;
                while (bottom <= top) {
                        mid = (top + bottom)/2;
                        if (elemArray [mid] == key) {
                                r.index = mid;
                                r.found = true;
                                return;
                        }
                        else {
                                if (elemArray[mid] < key) bottom = mid + 1;
                                else top = mid – 1;
                        }
                }
        }
}
```

WHITE BOX TESTING

CONTROL FLOW TESTING

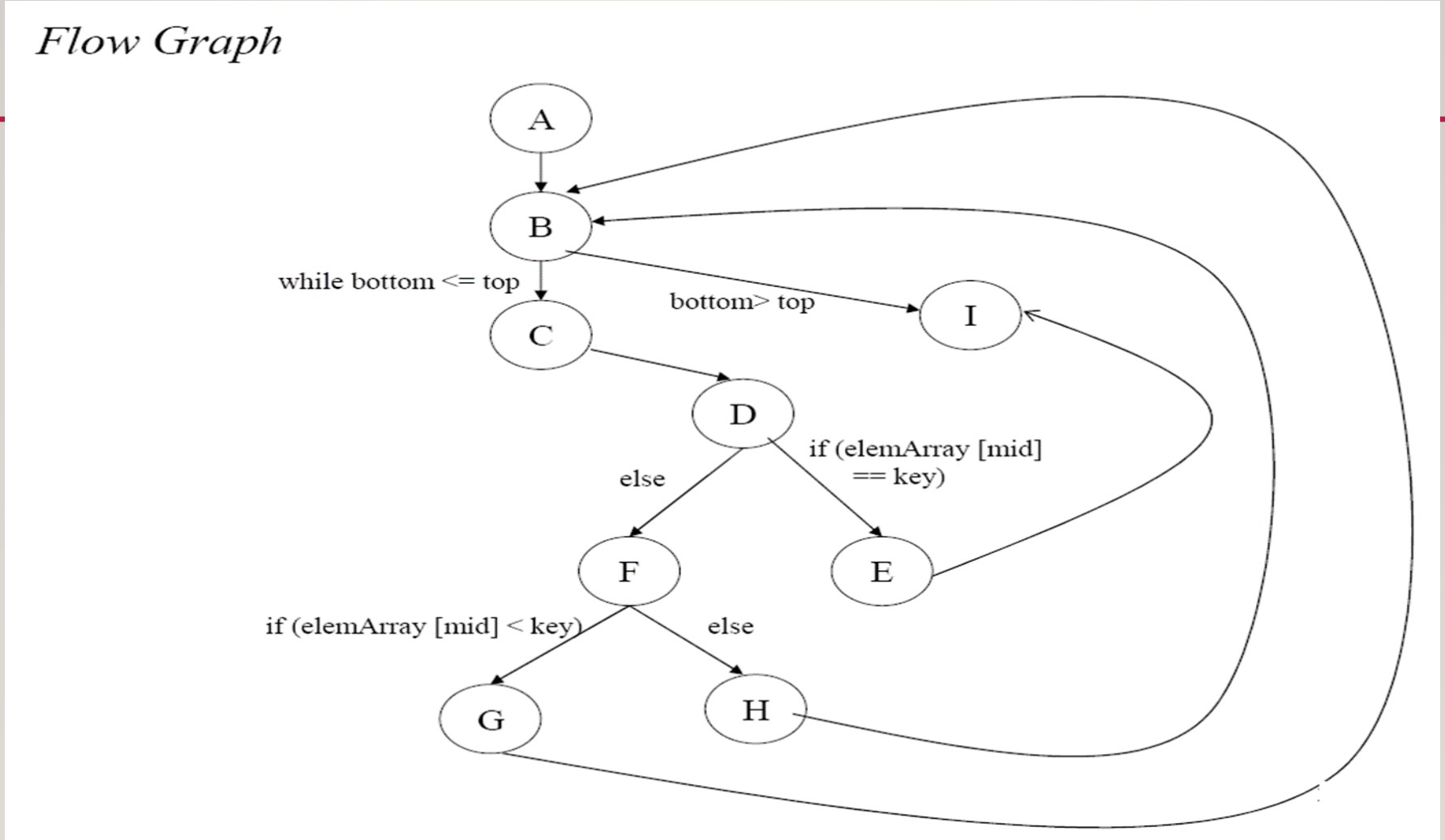
```
Class BinSearch {

        public static void search(int key, int[] elemArray, Result r) {
                int bottom = 0;
                int top = elemArray.length – 1;
                int mid;
                r.found = false; r.index=-1;
                while (bottom <= top)    {
                        mid = (top + bottom)/2;
                        if (elemArray [mid] == key) {
                                r.index = mid;
                                r.found = true;
                                return;
                        }
                        else {
                                if (elemArray[mid] < key)
                                        bottom = mid + 1;
                                else top = mid – 1;
                        }
                }
        }
}
```

A B C D E F G H I

WHITEBOX TESTING

CONTROL FLOW TESTING



WHITEBOX TESTING

CONTROL FLOW TESTING

Basis-Path Model

Cyclomatic Complexity Metric

-The cyclomatic complexity metric C is defined as the number of edges minus the number of nodes plus 2:

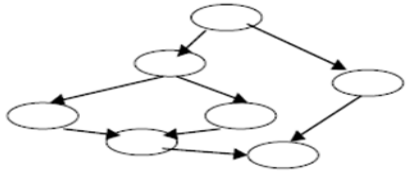
$$C = e - n + 2$$

- Where *e* and *n* stand for the number of edges and the number of nodes in corresponding CFG, respectively.

-Also called **McCabe complexity** metric

- Evaluate the ***complexity of algorithms*** involved in a method.
- Give a count of the minimum number of test cases needed to test a method comprehensively.
- Low C value means reduced testing, and better understandability.

-Example:



CC= e- n + 2 = 8 – 7 + 2 = 3

WHITEBOX TESTING

CONTROL FLOW TESTING

Independent Paths and Number of Test cases

ABI
ABCDEI
A(BCDFG)*BI
A(BCDFH)*BI

-The minimum number of test cases required to test all program paths is equal to the cyclomatic complexity (CC).

CC = Number (edges) – Number (nodes) + 2 = 11 – 9 + 2 = 4

WHITEBOX TESTING

CONTROL FLOW TESTING

Summary

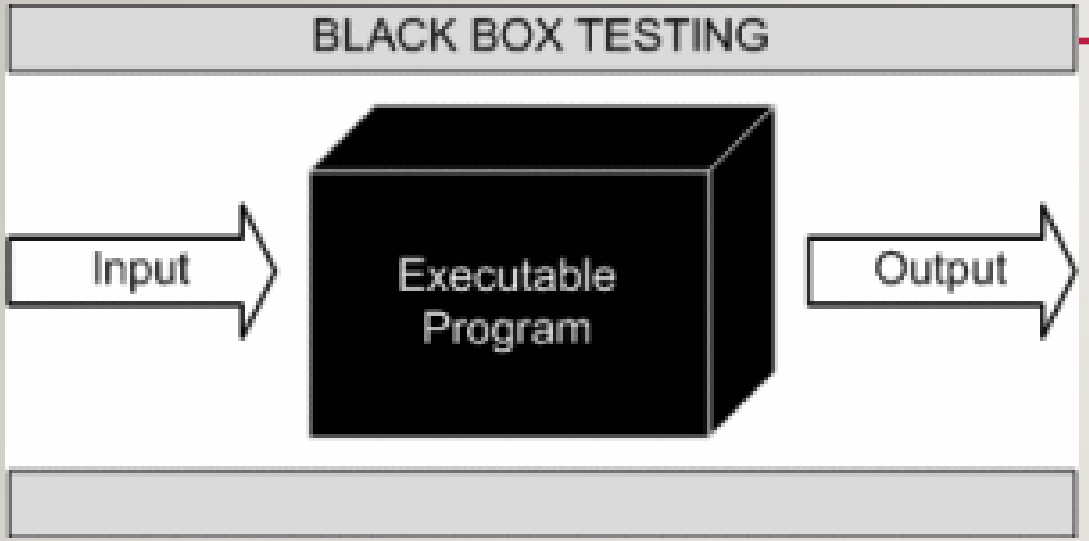
- The object of control-flow testing is to execute enough tests to assure that statement and branch coverage has been achieved.
- Select paths as deviation from the normal paths. Add paths as needed to achieve coverage.
- Use instrumentation (manual or using tools) to verify paths.
- Document all tests and expected test results.
- A test that reveals a bug has succeeded, not failed.

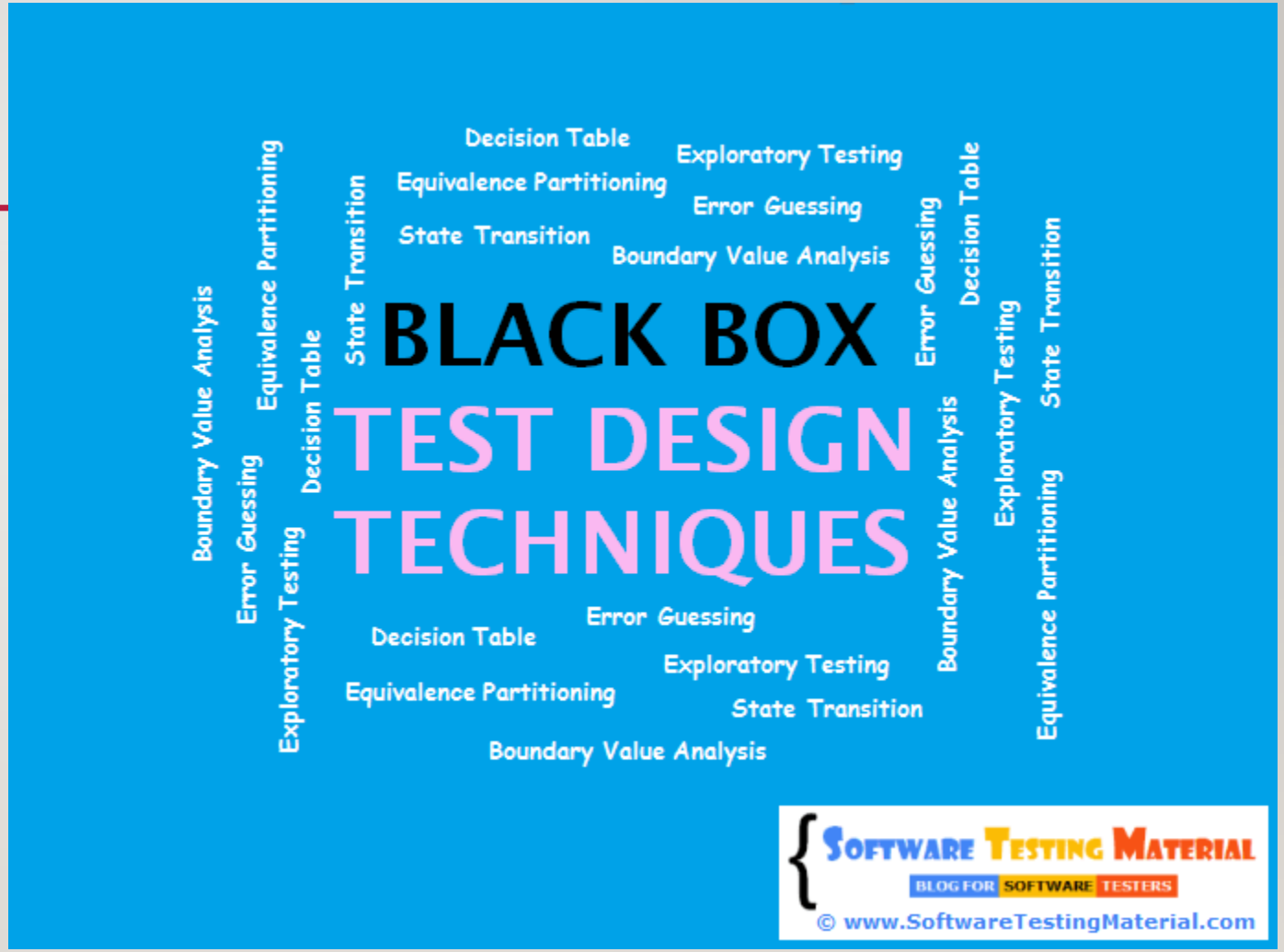
Learn more

https://viblo.asia/p/ky-thuat-kiem-thu-hop-trang-white-box-testing-maGK7MpOlj2

https://viblo.asia/p/data-flow-testing-3NVRkbKpv9xn

BLACKBOX TESTING





BLACK BOX TESTING

- **Black box testing**, also known as Behavioral Testing, is a software testing method in which the internal structure/design/implementation of the item being tested is **not** known to the tester. These tests can be functional or non-functional, though usually functional.
- **Black box testing:** Testing, either functional or non-functional, without reference to the internal structure of the component or system.
- **Black box test design technique:** Procedure to derive and/or select test cases based on an analysis of the specification, either functional or non-functional, of a component or system without reference to its internal structure.

BLACKBOX TESTING

Black Box Testing method is applicable to the following levels of software testing:

- Integration Testing
- System Testing
- Acceptance Testing

The higher the level, and hence the bigger and more complex the box, the more black-box testing method comes into use.

BLACKBOX TESTING

Techniques

- *Equivalence Partitioning:* It is a software test design technique that involves dividing input values into valid and invalid partitions and selecting representative values from each partition as test data.
- *Boundary Value Analysis:* It is a software test design technique that involves the determination of boundaries for input values and selecting values that are at the boundaries and just inside/ outside of the boundaries as test data.
- *Cause-Effect Graphing: (decision table)* It is a software test design technique that involves identifying the cases (input conditions) and effects (output conditions), producing a Cause-Effect Graph, and generating test cases accordingly.
- *State Transition Testing*

BLACK BOX TESTING

Advantages

- Tests are done from a user's point of view and will help in exposing discrepancies in the specifications.
- Tester need not know programming languages or how the software has been implemented.
- Tests can be conducted by a body independent from the developers, allowing for an objective perspective and the avoidance of developer-bias.
- Test cases can be designed as soon as the specifications are complete.

BLACKBOX TESTING

Disadvantages

- Only a small number of possible inputs can be tested and many program paths will be left untested.
- Without clear specifications, which is the situation in many projects, test cases will be difficult to design.
- Tests can be redundant if the software designer/developer has already run a test case.
- Ever wondered why a soothsayer closes the eyes when foretelling events? So is almost the case in Black Box Testing.

BLACK BOX TESTING

EQUIVALENCE PARTITIONING:

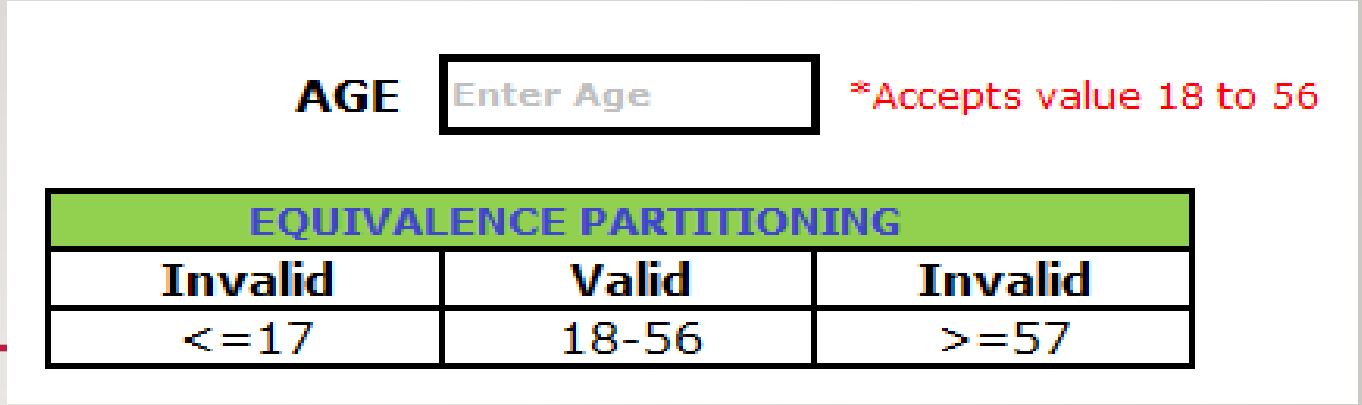
- It is also known as Equivalence Class Partitioning (ECP).
- Using equivalence partitioning test design technique, we divide the test conditions into class (group). From each group we do test only one condition. Assumption is that all the conditions in one group works in the same manner. If a condition from a group works then all of the conditions from that group work and vice versa. It reduces lots of rework and also gives the good test coverage. We could save lots of time by reducing total number of test cases that must be developed.
- **For example:** A field should accept numeric value. In this case, we split the test conditions as Enter numeric value, Enter alpha numeric value, Enter Alphabets, and so on. Instead of testing numeric values such as 0, 1, 2, 3, and so on.

BLACK BOX TESTING

- Equivalence Partitioning is also known as Equivalence Class Partitioning. In equivalence partitioning, inputs to the software or system are divided into groups that are expected to exhibit similar behavior, so they are likely to be proposed in the same way. Hence selecting one input from each group to design the test cases.
- Each and every condition of particular partition (group) works as same as other. If a condition in a partition is valid, other conditions are valid too. If a condition in a partition is invalid, other conditions are invalid too.
- It helps to reduce the total number of test cases from infinite to finite. The selected test cases from these groups ensure coverage of all possible scenarios.
- Equivalence partitioning is applicable at all levels of testing

BLACK BOX TESTING

Example 1:



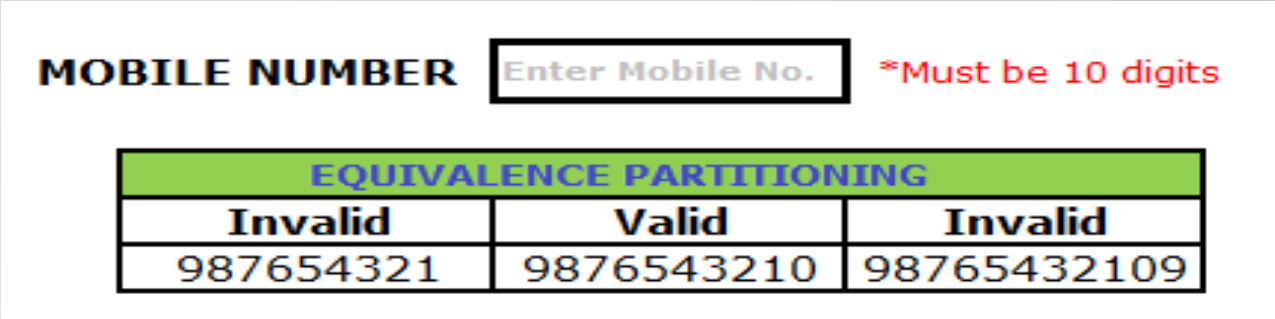
EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
<=17	18-56	>=57

Assume, we have to test a field which accepts Age 18 – 56

- Valid Input: 18 – 56
- Invalid Input: less than or equal to 17 (<=17), greater than or equal to 57 (>=57)
- Valid Class: 18 – 56 = Pick any one input test data from 18 – 56
- Invalid Class 1: <=17 = Pick any one input test data less than or equal to 17
- Invalid Class 2: >=57 = Pick any one input test data greater than or equal to 57
- We have one valid and two invalid conditions here.

BLACK BOX TESTING

Example 2:



EQUIVALENCE PARTITIONING		
Invalid	**Valid**	**Invalid**
987654321	9876543210	98765432109

- Assume, we have to test a filed which accepts a Mobile Number of ten digits.
- Valid input: 10 digits
- Invalid Input: 9 digits, 11 digits
- Valid Class: Enter 10 digit mobile number = 9876543210
- Invalid Class Enter mobile number which has less than 10 digits = 987654321
- Invalid Class Enter mobile number which has more than 11 digits = 98765432109

BLACK BOX TESTING

BOUNDARY VALUE ANALYSIS:

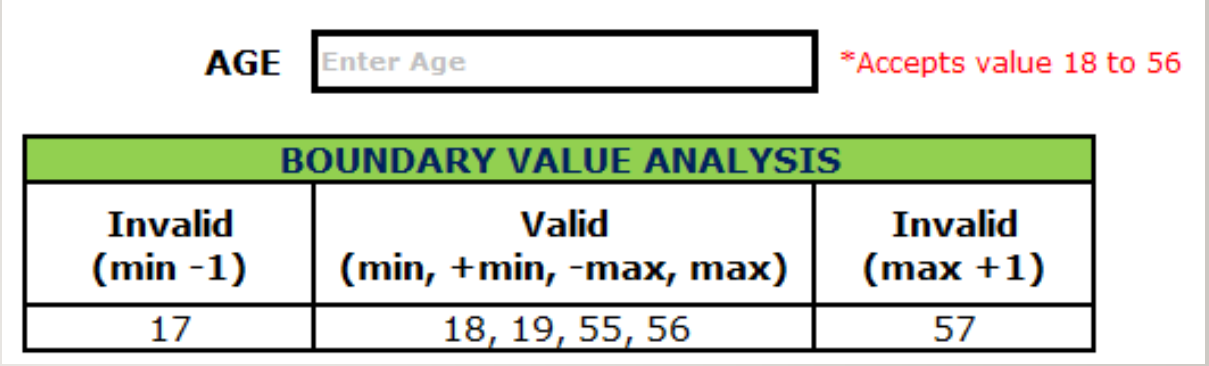
- Using Boundary value analysis (BVA), we take the test conditions as partitions and design the test cases by getting the boundary values of the partition. The boundary between two partitions is the place where the behavior of the application varies. The test conditions on either side of the boundary are called boundary values. In this we have to get both valid boundaries (from the valid partitions) and invalid boundaries (from the invalid partitions).
- For example: If we want to test a field which should accept only amount more than 10 and less than 20 then we take the boundaries as 10-1, 10, 10+1, 20-1, 20, 20+1. Instead of using lots of test data, we just use 9, 10, 11, 19, 20 and 21

BLACK BOX TESTING

- Boundary value analysis (BVA) is based on testing the boundary values of valid and invalid partitions. The Behavior at the edge of each equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.
- Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition.
- A boundary value for a valid partition is a valid boundary value. Similarly a boundary value for an invalid partition is an invalid boundary value.
- Tests can be designed to cover both valid and invalid boundary values. When designing test cases, a test for each boundary value is chosen.
- For each boundary, we test +/-1 in the least significant digit of either side of the boundary.
- Boundary value analysis can be applied at all test levels.

BLACK BOX TESTING

Example 1



BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	**Valid (min, +min, -max, max)**	**Invalid (max +1)**
17	18, 19, 55, 56	57

- Assume, we have to test a field which accepts Age 18 – 56
- Minimum boundary value is 18
- Maximum boundary value is 56
- Valid Inputs: 18,19,55,56
- Invalid Inputs: 17 and 57

Test case 1: Enter the value 17 (18-1) = Invalid
Test case 2: Enter the value 18 = Valid
Test case 3: Enter the value 19 (18+1) = Valid
Test case 4: Enter the value 55 (56-1) = Valid
Test case 5: Enter the value 56 = Valid
Test case 6: Enter the value 57 (56+1) =Invalid

BLACK BOX TESTING

Name Enter Name *Accepts characters length (6 - 12)

BOUNDARY VALUE ANALYSIS		
Invalid (min -1)	Valid (min, +min, -max, max)	Invalid (max +1)
5 characters	6, 7, 11, 12 characters	13 characters

Example 2:

Assume we have to test a text field (Name) which accepts the length between 6-12 characters.

- Minimum boundary value is 6
- Maximum boundary value is 12
- Valid text length is 6, 7, 11, 12
- Invalid text length is 5, 13
- Test case 1: Text length of 5 (min-1) = Invalid
- Test case 2: Text length of exactly 6 (min) = Valid
- Test case 3: Text length of 7 (min+1) = Valid
- Test case 4: Text length of 11 (max-1) = Valid
- Test case 5: Text length of exactly 12 (max) = Valid
- Test case 6: Text length of 13 (max+1) = Invalid

BLACK BOX TESTING

Decision Table:

Decision Table is aka Cause-Effect Table. This test technique is appropriate for functionalities which has logical relationships between inputs (if-else logic). In Decision table technique, we deal with combinations of inputs. To identify the test cases with decision table, we consider conditions and actions. We take conditions as inputs and actions as outputs.

Examples

Here the conditions to transfer money are ACCOUNT ALREADY APPROVED, OTP (One Time Password) MATCHED, SUFFICIENT MONEY IN THE ACCOUNT

And the actions performed are TRANSFER MONEY, SHOW A MESSAGE AS INSUFFICIENT AMOUNT, BLOCK THE TRANSACTION INCASE OF SUSPICIOUS TRANSACTION.

BLACK BOX TESTING

- In the first column I took all the conditions and actions related to the requirement. All the other columns represent Test Cases.
- T = True, F = False, X = Not possible

DECISION TABLE

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3	TEST CASE 4	TEST CASE 5
Condition 1	Account Already Approved	T	T	T	T	F
Condition 2	OTP (One Time Password) Matched	T	T	F	F	X
Condition 3	Sufficient Money in the Account	T	F	T	F	X
Action 1	Transfer Money	Execute				
Action 2	Show a Message as 'Insufficient Amount'		Execute			
Action 3	Block The Transaction Incase of Suspicious Transaction			Execute	Execute	X

© www.SoftwareTestingMaterial.com

BLACK BOX TESTING

- From the case 3 and case 4, we could identify that if condition 2 failed then system will execute Action 3. So we could take either of case 3 or case 4
- So finally concluding with the below tabular column.
- We write 4 test cases for this requirement.

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3	TEST CASE 4
Condition 1	Account Already Approved	T	T	T	F
Condition 2	OTP (One Time Password) Matched	T	T	F	X
Condition 3	Sufficient Money in the Account	T	F	T	X
Action 1	Transfer Money	Execute			
Action 2	Show a Message as 'Insufficient Amount'		Execute		
Action 3	Block The Transaction Incase of Suspicious Transaction			Execute	X

BLACK BOX TESTING

Example 2:

Take another example – login page validation. Allow user to login only when both the 'User ID' and 'Password' are entered correct.

- Here the Conditions to allow user to login are Enter Valid User Name and Enter Valid Password.
- The Actions performed are Displaying home page and Displaying an error message that User ID or Password is wrong.

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3	TEST CASE 4
Condition 1	Valid User ID	T	T	F	F
Condition 2	Valid Password	T	F	T	F
Action 1	Home Page	Execute			
Action 2	Show a Message as 'Invalid User Credentials'		Execute	Execute	Execute

BLACK BOX TESTING

- From the case 2 and case 3, we could identify that if one of the condition failed then the system will display an error message as Invalid User Credentials.
- Eliminating one of the test case from case 2 and case 3 and concluding with the below tabular column

ID	CONDITIONS/ACTIONS	TEST CASE 1	TEST CASE 2	TEST CASE 3
Condition 1	Valid User ID	T	T	F
Condition 2	Valid Password	T	F	F
Action 1	Home Page	Execute		
Action 2	Show a Message as 'Invalid User Credentials'		Execute	Execute

BLACK BOX TESTING

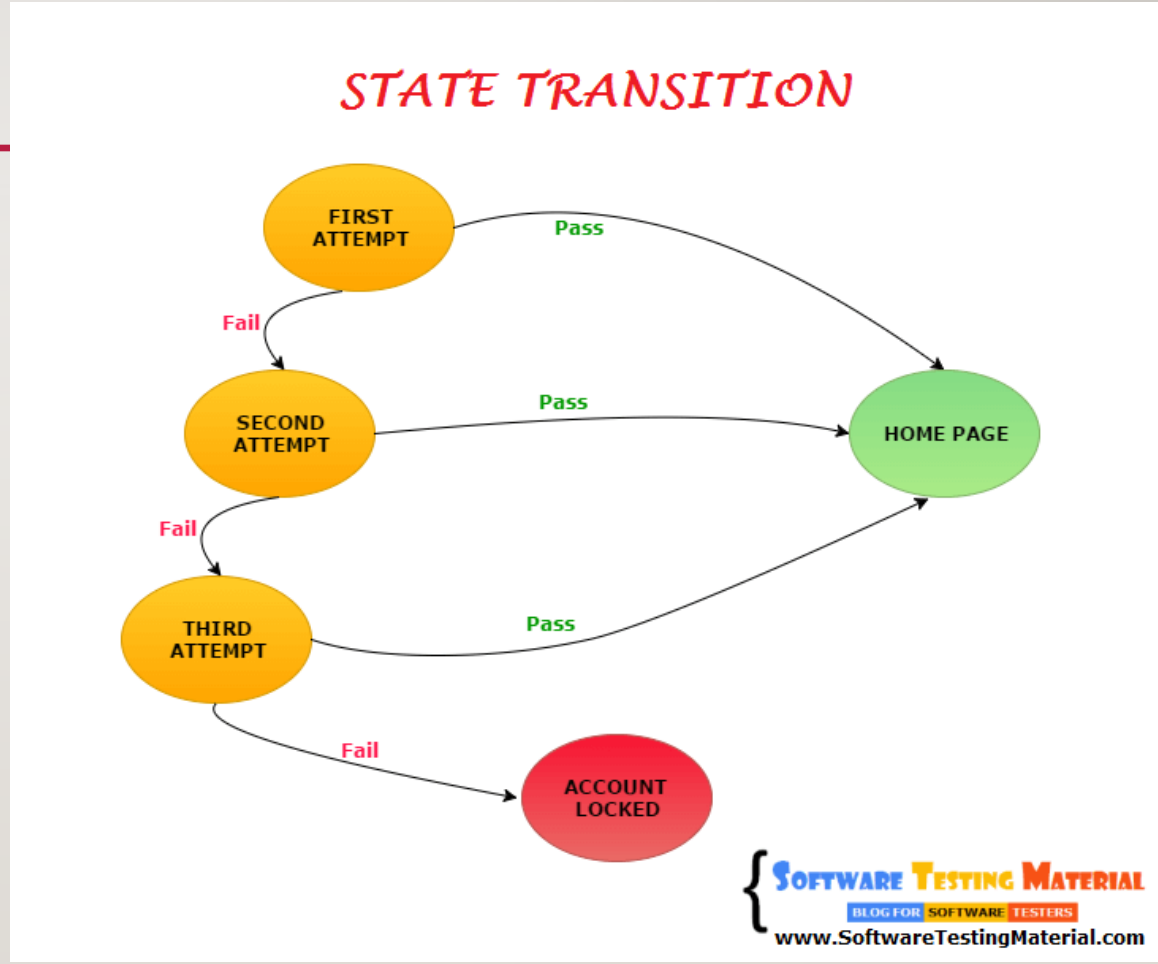
State Transition Testing:

- Using state transition testing, we pick test cases from an application where we need to test different system transitions. We can apply this when an application gives a different output for the same input, depending on what has happened in the earlier state.
- Some examples are Vending Machine, Traffic Lights.
- Vending machine dispenses products when the proper combination of coins is deposited.
- Traffic Lights will change sequence when cars are moving / waiting

BLACK BOX TESTING

Example 1

- Take an example of login page of an application which locks the user name after three wrong attempts of password.
- A finite state system is often shown as a state diagram



BLACK BOX TESTING

- Entering correct password in the first attempt or second attempt or third attempt, user will be redirected to the home page (i.e., State – S4).
- Entering incorrect correct password in the first attempt, a message will be displayed as try again and user will be redirected to state S2 for the second attempt.
- Entering incorrect correct password in the second attempt, a message will be displayed as try again and user will be redirected to state S3 for the third attempt.
- Entering incorrect correct password in the third attempt, user will be redirected to state S5 and a message will be displayed as "Account locked. Consult Administrator".

It works like a truth table. First determine the states, input data and output data.

STATE	LOGIN	CORRENT PASSWORD	INCORRECT PASSWORD
S1	First Attempt	S4	S2
S2	Second Attempt	S4	S3
S3	Third Attempt	S4	S5
S4	Home Page		
S5	Display a message as "Account Locked, please consult Administrator"		

BLACK BOX TESTING

- Example 2:

Withdrawal of money from ATM. ‘User A’ wants to withdraw 30,000 from ATM. Imagine he could take 10,000 per transaction and total balance available in the account is 25,000. In the first two attempts, he could withdraw money. Whereas in the third attempt, ATM shows a message as “Insufficient balance, contact Bank”. Same Action but due to change in the state, he couldn’t withdraw the money in the third transaction.

GREY BOX TESTING

GREY BOX TESTING = WHITE BOX TESTING + BLACK BOX TESTING