04 Structure-based test design

PowerPoint by Wanida Khamprapai

Test design techniques

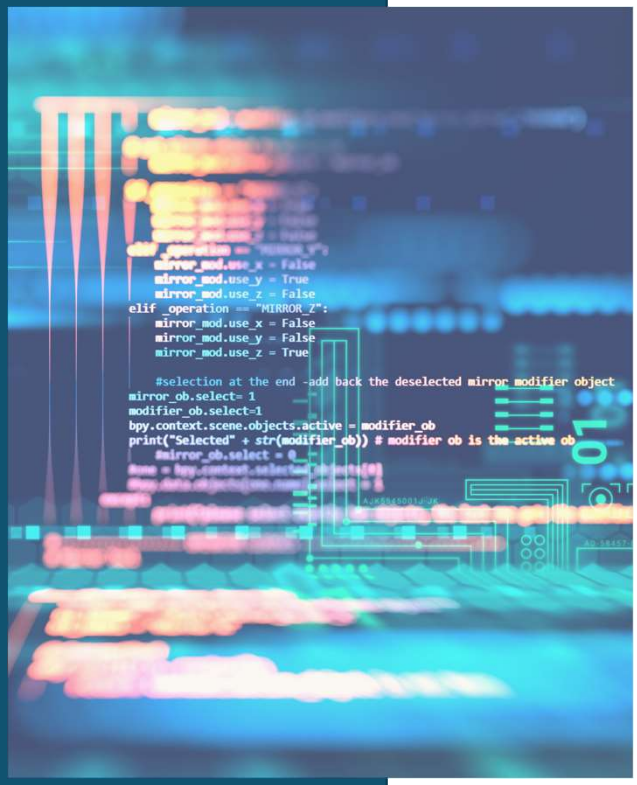
Goal: Select test cases based on test objectives.

Specification-based testing

- SUT: Black-box testing
- Only specification is known
- Testing specified functionality

Structure-based testing

- SUT: White-box testing
- internal structure known
- Testing based on internal behavior



Structure-based testing

Structure-based testing, also known as white box testing is a code-based testing technique in which the internal structure is being known to the tester who is going to test the software. Here, the test cases are calculated after analyzing the internal structure of the system based on code, branch, path, and condition coverage.

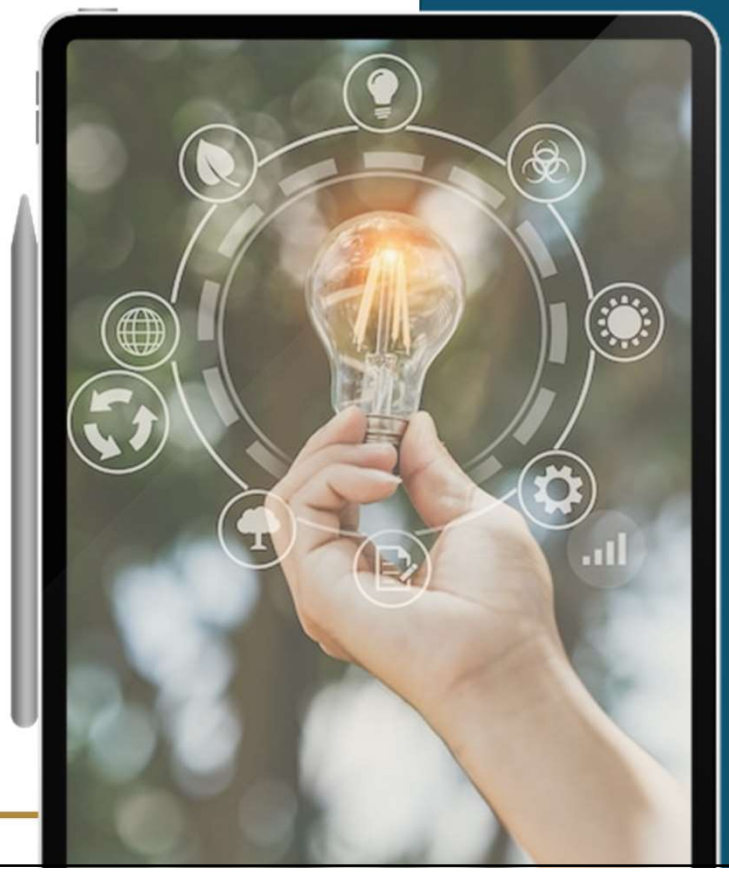
Structure-based testing basically involves the following steps

- Verifying security loops in the code
- Verifying the broken paths in the code
- Verifying the specified flow structure
- Verifying the desired output
- Verifying the condition loop to check the functionality
- Verifying each line and section

Structure-based testing

Every program uses sequential statements, condition statements (If-then-else or Switch), and loop structures. Typically control statements break the normal sequence of the statement execution as even simple if-then-else statements will result in two possible paths, one with a true value and the other with a false value.

Each input action or input data set is considered a test case. So, the primary objective of the white box testing technique is to ensure that all the statements of the code, all the paths of the decision statement, and all the results (true & false) of each control statements used are covered at least once during testing.



Structure-based testing

Coverage

A measure of the degree to which the test cases exercise or cover the logic (source code) of the program -% of code covered

The coverage goal is to ensure that

- Every statement is executed at least once
- All independent paths/decisions within a module are traversed at least once
- All logical conditions are exercised on their true and false sides
- All loops at their boundaries and within operational bounds are executed

This can be achieved by using following techniques

- Statement Coverage
- Path Coverage
- Modified Condition Decision Coverage
- Loop Testing

Structure-based testing

Basic statements

Two kinds of basic statements in a program unit are assignment statements and control statements.

- Assignment statement is explicitly represented by using an assignment symbol, “ = ”, such as x = 2*y;, where x and y are variables.
- Control statements are at the core of conditional statements, such as if(), switch..case, for() loop, while() loop, and go to. As an example, in if(x! = y), we are testing for the inequality of x and y.

Structure-based testing

```
import java.util.Scanner;
class IfStatement {
     public static void main(String[] args) {
          Scanner myObj = new Scanner(System.in);
          int number = myObj.nextInt();
          if (number < 0) {
               System.out.println("The number is negative.");
          }
          System.out.println("Statement outside if block");
     }
}
```

```
#include <stdio.h>
int main() {
     double number, sum = 0;
     do {
            printf("Enter a number: ");
            scanf("%lf", &number);
            sum += number;
     } while(number != 0.0);
     printf("Sum = %.2lf",sum);
     return 0;
}
```

```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
```

Statement coverage

- As part of this technique, one needs to use test cases (or data inputs) such that all statements of the program are executed at least once.
- If the program or a component uses only simple sequential statements without any control structure (condition branches), all the statements will get executed once the first statement is executed.

$$Statement\ coverage = \frac{Total\ number\ of\ statements\ exercised}{Total\ number\ of\ executable\ statements\ in\ the\ program\ section\ under\ test} \times 100$$

Statement coverage

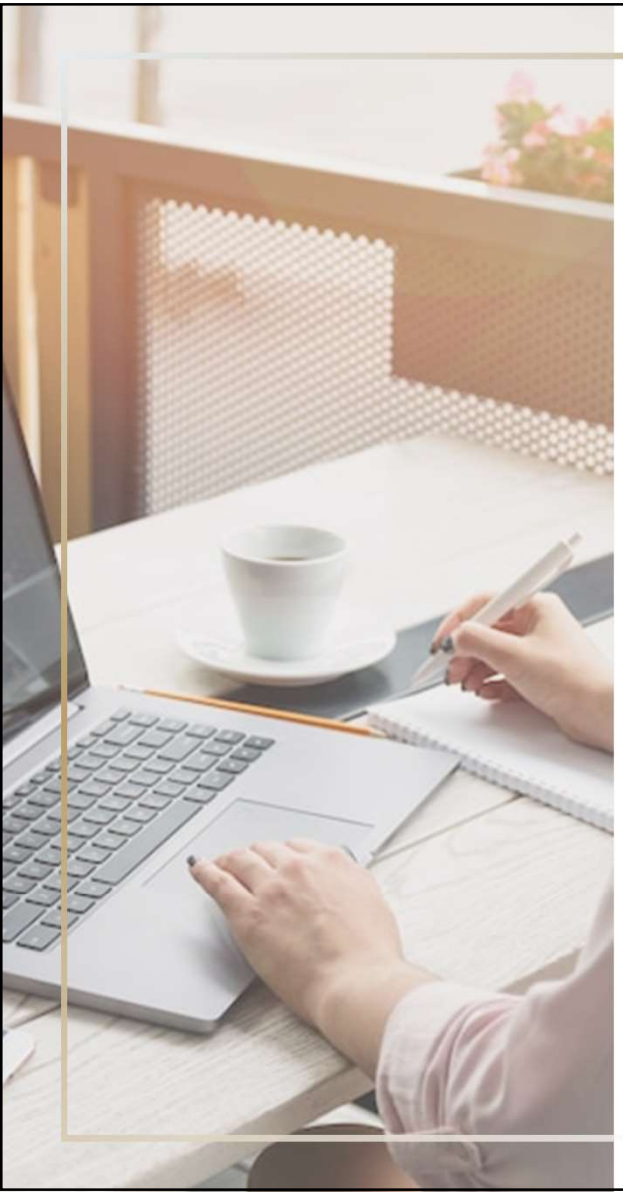
```
# Store input numbers
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')

# Add two numbers
sum = float(num1) + float(num2)

# Display the sum
print('The sum of {0} and {1} is
{2}'.format(num1, num2, sum))
```

Test case 1: num1 = 4 and num2 = 6

Total of statements = 4 statements

Total of exercised statements = 4 statements

So, The statement coverage would be (4 / 4) * 100 = 100%



Statement coverage

```
import java.util.Scanner;
class IfStatement {
     public static void main(String[] args) {
          Scanner myObj = new Scanner(System.in);
          int number = myObj.nextInt();
          if (number < 0) {
               System.out.println("The number is negative.");
          }
          System.out.println("Statement outside if block");
     }
}
```

Test case 1: number = 3

Total of statements = 8 statements

Total of exercised statements = ? statements

The statement coverage would be (? / 8) * 100 = ?%

Test case 2: number = ?

Total of statements = 8 statements

Total of exercised statements = ? statements

The statement coverage would be (? / 8) * 100 = ?%

Statement coverage

```
#include <stdio.h>
int main() {
      double number, sum = 0;
      do {
             printf("Enter a number: ");
             scanf("%lf", &number);
             sum += number;
      } while(number != 0.0);
      printf("Sum = %.2lf",sum);
      return 0;
}
```

Test case 1: number = ?

Total of statements = ? statements

Total of exercised statements = ? statements

The statement coverage would be (? / ?) * 100 = ?%

Test case 2: number = ?

Total of statements = ? statements

Total of exercised statements = ? statements

The statement coverage would be (? / ?) * 100 = ?%

Path Coverage

Decision statements allow programmers to choose which statement should be executed under different conditions. At runtime, the decision statements determine which statement to execute based on the test expression. Every decision has two paths – one with a TRUE value and the other with a FALSE value. So, we need to have additional test cases. Decision statements are used to create test cases that cover all the paths of the decision.

Decision statements

There are four types:

- if statement
- if-else statement
- if else if statement
- switch case statement

Test cases are designed to cover all the paths, every statement in the program will be executed at least once and every decision will have been executed on its TRUE and FALSE sides.

$$Path\ coverage = \frac{Total\ paths\ exercised}{Total\ number\ of\ paths\ in\ the\ program} \times 100$$

Path coverage

```
x = int(input('Enter number: '))
if x > 95:
   print('Student is brilliant')
elif x < 30:
   print('Student is poor')
elif x < 95 and int(x) > 30:
   print('Student is average')
print('The end')
```

Test case 1: x = 45

Decision	(True/False)
x > 95	False
x < 30	False
x < 95 and x > 30	True

Test case 2: x = ?

Decision	(True/False)
x > 95	
x < 30	
x < 95 and x > 30	

Test case 3: x = ?

Decision	(True/False)
x > 95	
x < 30	
x < 95 and x > 30	

Path coverage

```
// Program to create a simple calculator
#include <stdio.h>
int main() {
    char operation;
    double n1, n2;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%lf %lf",&n1, &n2);
    switch(operation) {
        case '+': printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
            break;
        case '-': printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
            break;
        case '*': printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
            break;
        case '/': printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
            break;
        // operator doesn't match any case constant +, -, *, /
        default: printf("Error! operator is not correct");
    }
    return 0;
}
```

Test case 1: operation = /

The exercised switch..case statement is case '/'

Test case 2: operation = ?

The exercised switch..case statement is ?

Test case 3: operation = ?

The exercised switch..case statement is ?

Test case 4: operation = ?

The exercised switch..case statement is ?

Test case 5: operation = ?

The exercised switch..case statement is ?

Test case 6: operation = ?

The exercised switch..case statement is ?

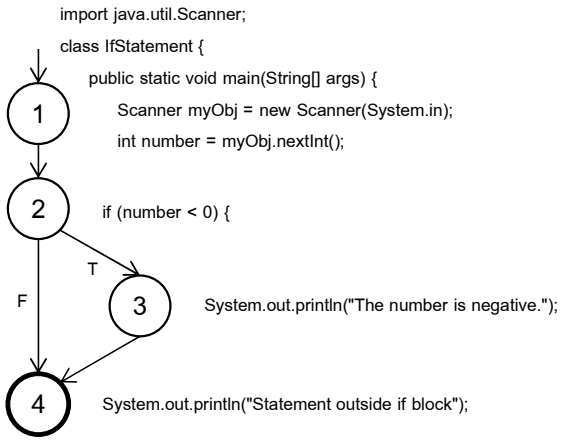
Path coverage

Control flow graph

For complex programs, the graphs are further simplified by using nodes (drawn as circles) and links (drawn as arrows).

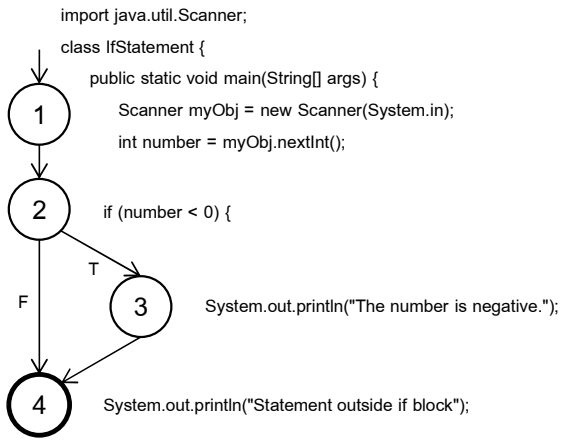
Nodes are generally numbered. Links indicate the direction of flow.

```
import java.util.Scanner;
class IfStatement {
     public static void main(String[] args) {
          Scanner myObj = new Scanner(System.in);
          int number = myObj.nextInt();
          if (number < 0) {
               System.out.println("The number is negative.");
          }
          System.out.println("Statement outside if block");
     }
}
```



Path coverage

Paths can be either represented using a sequence of nodes or using a sequence of links as given below. Length of the path is measured by the number of links in that path.



#Path	Path using nodes	Length of path
1	1-2-3-4	3
2	1-2-4	2

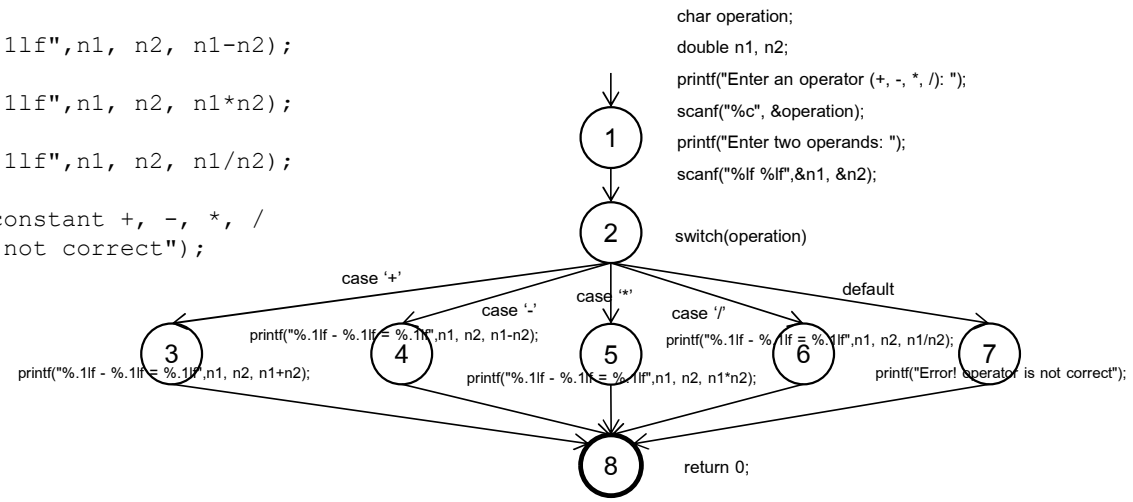
Test case	#Path	Coverage (%)
Number = ?		

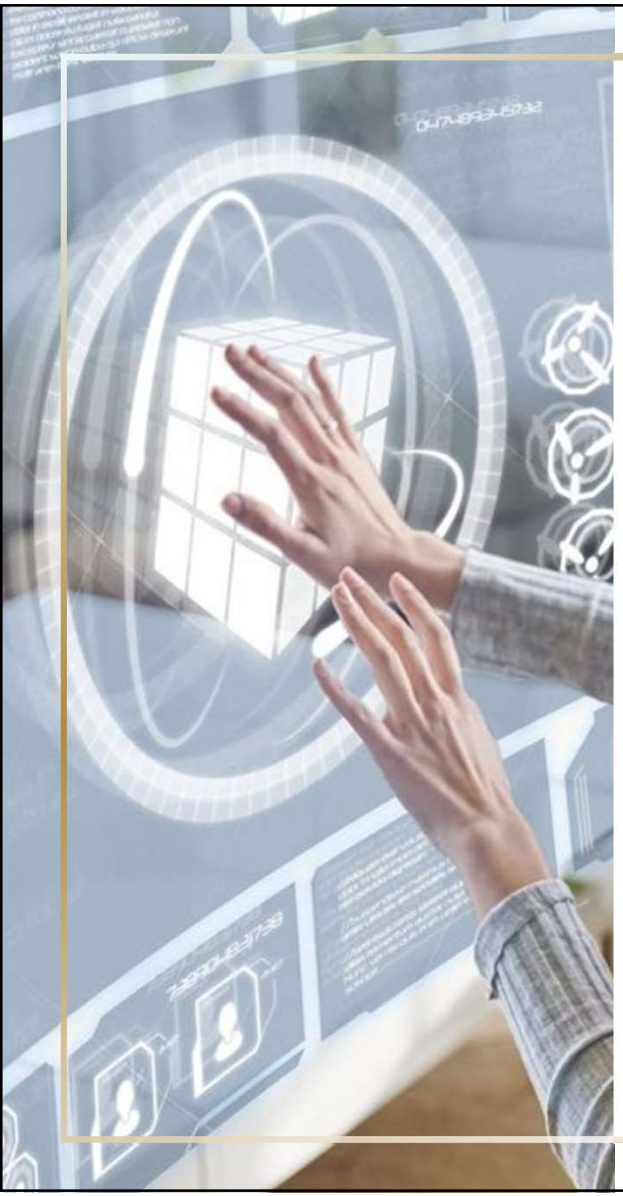
Path coverage

```
x = int(input('Enter number: '))
if x > 95:
   print('Student is brilliant')
elif x < 30:
   print('Student is poor')
elif x < 95 and int(x) > 30:
   print('Student is average')
print('The end')
```

Path coverage

```
// Program to create a simple calculator
#include <stdio.h>
int main() {
    char operation;
    double n1, n2;
    printf("Enter an operator (+, -, *, /): ");
    scanf("%c", &operation);
    printf("Enter two operands: ");
    scanf("%lf %lf",&n1, &n2);
    switch(operation) {
        case '+': printf("%.1lf + %.1lf = %.1lf",n1, n2, n1+n2);
            break;
        case '-': printf("%.1lf - %.1lf = %.1lf",n1, n2, n1-n2);
            break;
        case '*': printf("%.1lf * %.1lf = %.1lf",n1, n2, n1*n2);
            break;
        case '/': printf("%.1lf / %.1lf = %.1lf",n1, n2, n1/n2);
            break;
        // operator doesn't match any case constant +, -, *, /
        default: printf("Error! operator is not correct");
    }
    return 0;
}
```





Path coverage

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main () {
   /* local variable definition */
   int a = 100;
   int b = 200;
   int ret;
   /* calling a function to get max value */
   ret = max(a, b);
   printf( "Max value is : %d\n", ret );
   return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2) {
   /* local variable declaration */
   int result;
   if (num1 > num2)
      result = num1;
   else
      result = num2;
   return result;
}
```

Path coverage

```
#include <stdio.h>
int main () {
   int num;
   printf("Enter a number : ");
   scanf("%d",&num);
   if(num > 0){
      if (num%2 == 0)
          printf(" %d is positive even number \n", num);
      else
          printf(" %d is positive odd number \n", num);
   } else if(num < 0){
      if (num%2 == 0)
          printf(" %d is negative even number \n", num);
      else
          printf(" %d is negative odd number \n", num);
   } else
          printf(" %d is zero \n", num);
   printf("Bye...");
   return 0;
}
```

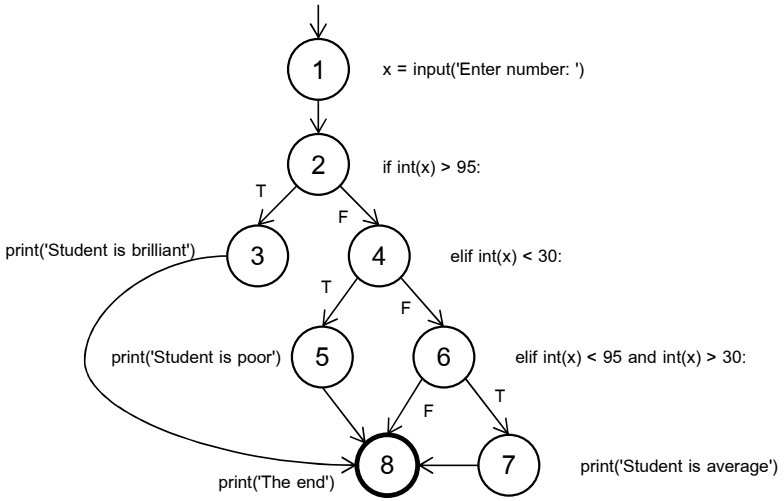
Modified condition decision coverage (MC/DC)

Modified condition decision coverage: MC/DC

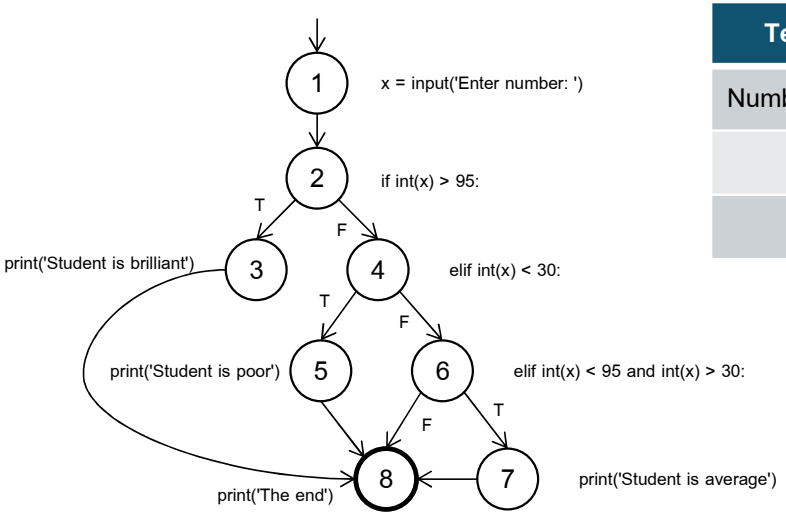
MC/DC is a coverage type that makes sure every condition within a decision determines every possible outcome of that decision.

This coverage type is a good combination of effectiveness (good coverage) and efficiency (not too many test cases).

```
x = int(input('Enter number: '))
if x > 95:
   print('Student is brilliant')
elif x < 30:
   print('Student is poor')
elif x < 95 and int(x) > 30:
   print('Student is average')
print('The end')
```



Modified condition decision coverage (MC/DC)



Test case	x > 95	Decision
Number = ?		

Test case	x < 30	Decision
Number = ?		

Test case	x < 95	x > 30	Decision
Number = ?			

Modified condition decision coverage (MC/DC)

```
public class LeapYearExample {
    public static void main(String[] args) {
        int year;
        if(((year % 4 == 0) && (year % 100 != 0)) || (year % 400 == 0)){
            System.out.println("LEAP YEAR");
        } else {
            System.out.println("COMMON YEAR");
        }
    }
}
```

Test case	year % 4 == 0	year % 100 != 0	year % 400==0	Decision
year = ?				

Modified condition decision coverage (MC/DC)

```
a = int(input('Enter a: '))
b = int(input('Enter b: '))
c = int(input('Enter c: '))
if((a>b and a>c) and (a != b and a != c)):
   print(a, " is the largest")
elif((b>a and b>c) and (b != a and b != c)):
   print(b, " is the largest")
elif((c>a and c>b) and (c != a and c != b)):
   print(c, " is the largest")
else:
   print("entered numbers are equal")
```

Test case	a>b	a>c	a != b	a != c	Decision
a = ? b = ? c = ?					

Loop Testing

Loop testing focuses on the validity of loop constructs. Loop is a commonly used construct required in most programs. Loops make path testing difficult due to the significantly increased number of possible paths, and they often contain bugs within the loop condition at the boundary values that specify the first iteration and the last iteration of the loop.

For loop

The for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

While loop

The while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

Do-while loop

The do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once.

Loop Testing

Simple loop testing

If the loop can only be traversed a maximum of n times, the following checks should be taken into consideration:

- The loop is skipped entirely, having undergone exactly zero iterations.
- Just one loop iteration
- Two loop iterations
- m iterations of the loop where m < n
- n-1 iterations of the loop
- N+1 iterations of the loop (if possible)

```
public class ForExample {
  public static void main(String[] args){
    //Code of Java for loop
    for(int i=1;i<=10;i++){
        System.out.println(i);
    }
  }
}
```

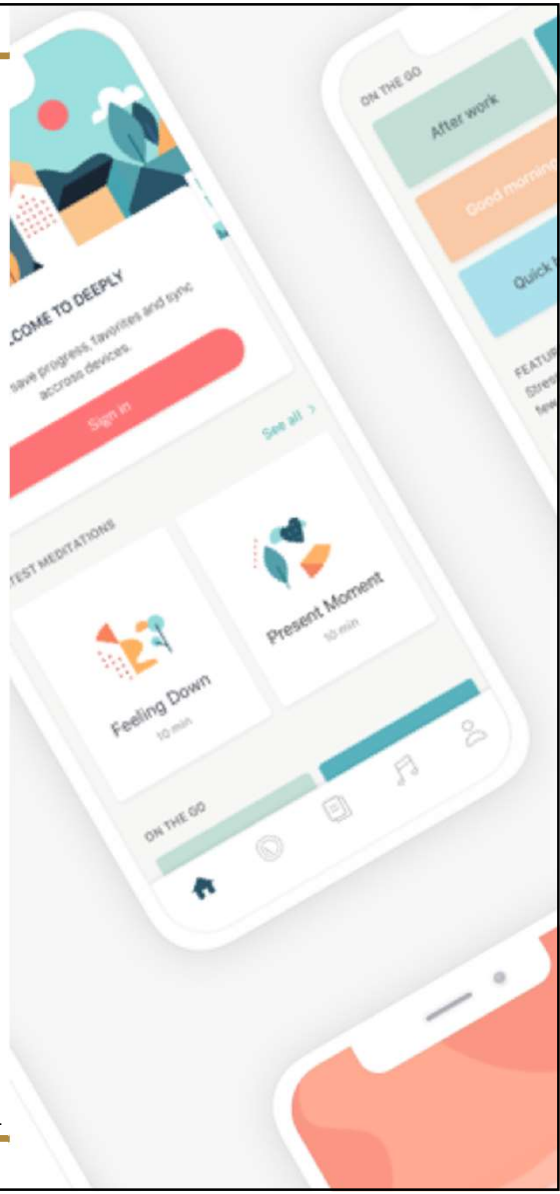
Loop Testing

Nested loop testing

Nested loops can result in an impractical number of tests if the nesting increases. Beizer (2008) suggests an approach that will help to reduce the number of tests:

- Start at 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 other tests for out-of-range or excluded values.
- Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.
- Continue until all loops have been tested.

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]
for x in adj:
  for y in fruits:
    print(x, y)
```



Information from Beizer, B. 2008. Software Testing Techniques. India: Wiley.

Loop Testing

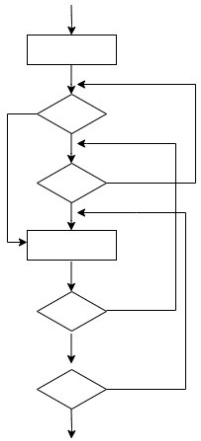
Concatenated loop testing

If each loop is independent of the other. So, test them as simple loops, else test them as nested loops

Spaghetti loops testing

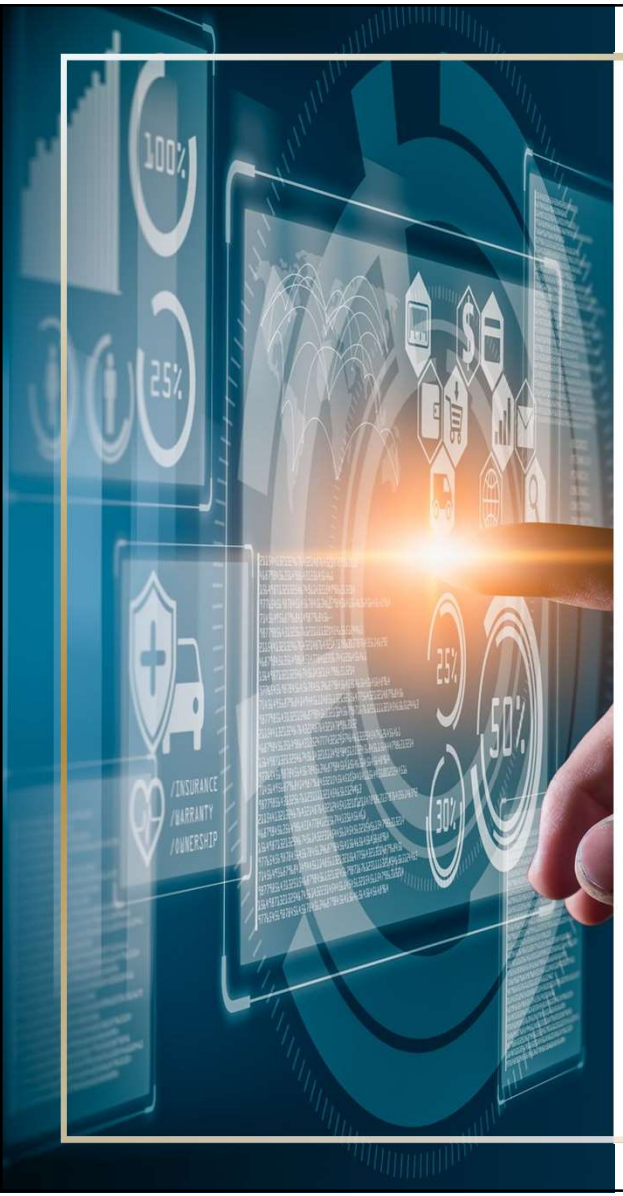
Unstructured loops are not good for any design and should not be accepted. So, it should be reported as a defect and suggested to redesign using structured constructs

```
while(condition 1){
    statement(s);
}

while(condition 2){
   statement(s);
}
```



Loop Testing

```
 /**
 * C program to find maximum and minimum element in array
 */

#include <stdio.h>
#define MAX_SIZE 100   // Maximum array size
int main(){
    int arr[MAX_SIZE];
    int i, max, min, size;

    /* Input size of the array */
    printf("Enter size of the array: ");
    scanf("%d", &size);

    /* Input array elements */
    printf("Enter elements in the array: ");
    for(i=0; i<size; i++){
        scanf("%d", &arr[i]);
    }
/* Assume first element as maximum and minimum */
    max = arr[0];
    min = arr[0];

    /*
     * Find maximum and minimum in all array elements.
     */
    for(i=1; i<size; i++){
        /* If current element is greater than max */
        if(arr[i] > max){
            max = arr[i];
        }

        /* If current element is smaller than min */
        if(arr[i] < min){
            min = arr[i];
        }
    }

    /* Print maximum and minimum element */
    printf("Maximum element = %d\n", max);
    printf("Minimum element = %d", min);

    return 0;
}
```



Loop Testing

```
i = int(input('Enter i: '))
while i <= 4 :
    j = 0
    while  j <= 3 :
        print(i*j, end=" ")
        j += 1
    print()
    i += 1
```

Test case 1: ?

Test case 2: ?

Test case 3: ?

Advantages and Limitations

Structure-based testing is best suited for unit testing and done by developers as it requires knowledge of programming and the internal structure of the program.

Advantages

- The entire program need not be run in order to check independent paths. Instead, think of each section of the code as only involving one component or decision at a time.
- Additionally, it works well for reusable technological components. All business functions that call technical components will produce false results if they include any errors.
- The method of structural testing doesn't take much time.
- It is simple to find any early defect.
- It makes it simple to delete dead code (excessive code) or statements.

Limitations

- Its inability to show missing logic or misunderstanding of business requirements.
- Since there are many paths, it is not possible to evaluate each one individually in order to find defects that could be problematically buried along the way.

Questions & Answers