Unit 5.2
Software Engineering

Software Testing

Software Testing

- What is Testing?

Many people understand many definitions of testing :

1 Testing is the process of demonstrating that errors are not present

2 The purpose of testing is to show that a program performs its intended functions correctly

3 Testing is the process of establishing confidence that a program does what it is supposed to do

These definitions are incorrect

Software Testing

A more appropriate definition is:

"Testing is the process of executing a program with the intent of finding errors"

Error/ Mistake/ Bug, Fault/Defect and Failure

Errors:

The ***Error*** is a human mistake. An Error appears not only due to the logical mistake in the code made by the developer. Anyone in the team can make mistakes during the different phases of software development. For instance,

- *BA (business analyst) may misinterpret or misunderstand requirements.*
- *The customer may provide insufficient or incorrect information.*
- *The architect may cause a flaw in software design.*

When developers make mistakes while coding, we call these mistakes "**bugs**"

Software Testing

Defect/ Fault:

A ***Defect*** is a variance between expected and actual results. An Error that the tester finds is known as Defect. A Defect in a software product reflects its inability or inefficiency to comply with the specified requirements and criteria and, subsequently, prevent the software application from performing the desired and expected work. The defect is also known as ***Fault***.

Failure:

Failure is a consequence of a Defect. It is the observable incorrect behavior of the system. Failure occurs when the software fails to perform in the real environment.

In other words, after the creation & execution of software code, if the system does not perform as expected, due to the occurrence of any defect; then it is termed as Failure.

Software Testing

- Why should We Test ?

Although software testing is itself an expensive activity, yet launching of software without testing may lead to cost potentially much higher than that of testing, specially in systems where human safety is involved

In the software life cycle the earlier the errors are discovered and removed, the lower is the cost of their removal

Software Testing

- Who should Do the Testing ?

  - Testing requires the developers to find errors from their software
  - It is difficult for software developer to point out errors from own creations
  - Many organisations have made a distinction between development and testing phase by making different people responsible for each phase

Software Testing

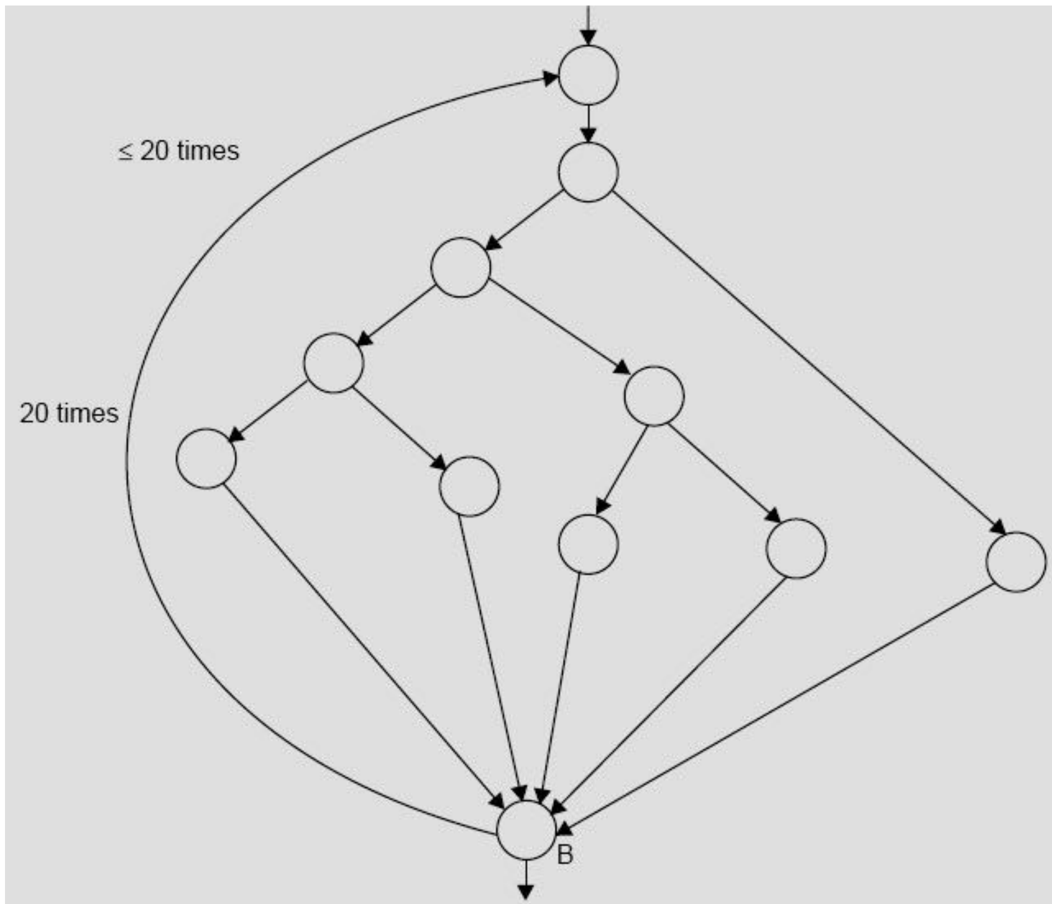

Fig 1: Control flow graph

Software Testing

The number of paths in the example of Fig 1 are 10^{14} or 100 trillions It is computed from $5^{20} + 5^{19} + 5^{18} + \ldots\ldots + 5^{1}$; where 5 is the number of paths through the loop body If only 5 minutes are required to test one test path, it may take approximately one billion years to execute every path

Software Testing

Test, Test Case and Test Suite

Test and **Test case** terms are used interchangeably. In practice, both are same and are treated as synonyms. Test case describes an input description and an expected output description.

Test Case ID	
Section-I (Before Execution)	Section-II (After Execution)
Purpose :	Execution History:
Pre condition: (If any)	Result:
Inputs:	If fails, any possible reason (Optional);
Expected Outputs:	Any other observation:
Post conditions:	Any suggestion:
Written by:	Run by:
Date:	Date:

Fig 2: Test case template

The set of test cases is called a **test suite**. Hence any combination of test cases may generate a test suite.

Software Testing

_ Alpha, Beta and Acceptance Testing

The term **Acceptance Testing** is used when the software is developed for a specific customer A series of tests are conducted to enable the customer to validate all requirements These tests are conducted by the end user / customer and may range from adhoc tests to well planned systematic series of tests

The terms alpha and beta testing are used when the software is developed as a product for anonymous customers

Alpha Tests are conducted at the developer"s site by some potential customers These tests are conducted in a controlled environment Alpha testing may be started when formal testing process is near completion

Beta Tests are conducted by the customers / end users at their sites Unlike alpha testing, developer is not present here Beta testing is conducted in a real environment that cannot be controlled by the developer

Software Testing

Boundary Value Analysis

Consider a program with two input variables x and y These input variables have specified boundaries as:

$$a \le x \le b$$

$$c \le y \le d$$

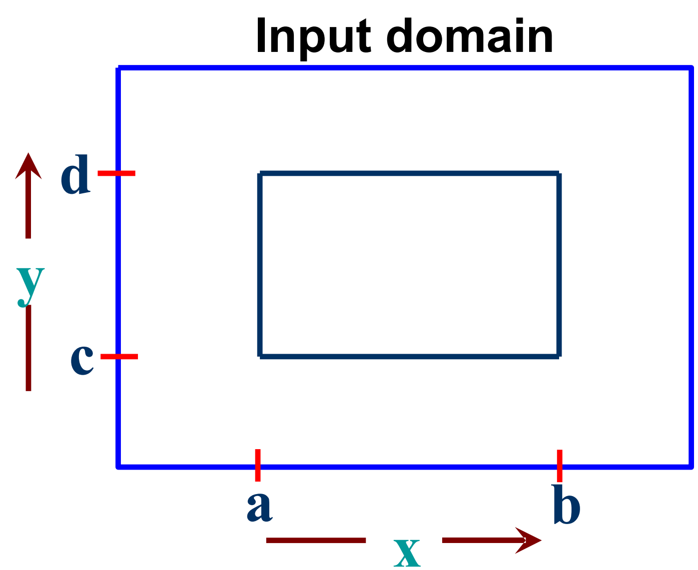

Fig 3: Input domain for program having two input variables

Software Testing

The boundary value analysis test cases for our program with two inputs variables (x and y) that may have any value from 100 to 300 are: (200,100), (200,101), (200,200), (200,299), (200,300), (100,200), (101,200), (299,200) and (300,200) This input domain is shown in Fig 4 Each dot represent a test case and inner rectangle is the domain of legitimate inputs Thus, for a program of n variables, boundary value analysis yield **4n + 1** test cases

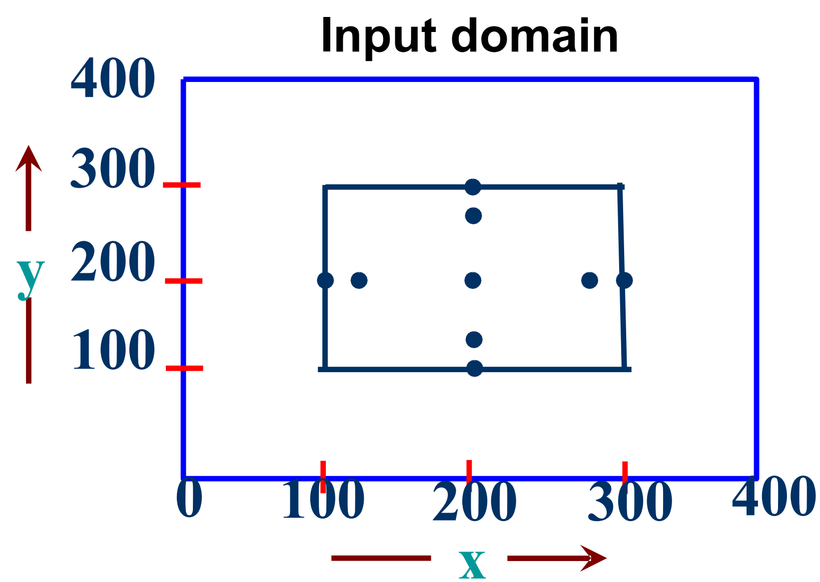

Fig 4: Input domain of two variables x and y with boundaries [100,300] each

Software Testing

Example

Consider a program for determining the Previous date. Its input is a triple of day, month and year with the values in the range

$1 \le month \le 12$

$1 \le day \le 31$

$1900 \le year \le 2025$

The possible outputs would be Previous date or invalid input date Design the boundary value test cases

Software Testing

Solution

The Previous date program takes a date as input and checks it for validity
If valid, it returns the previous date as its output

With single fault assumption theory, 4n+1 test cases can be designed and which are equal to 13

Software Testing

The boundary value test cases are:

Test Case	*Month*	*Day*	*Year*	*Expected output*
1	6	15	1900	14 June, 1900
2	6	15	1901	14 June, 1901
3	6	15	1962	14 June, 1962
4	6	15	2024	14 June, 2024
5	6	15	2025	14 June, 2025
6	6	1	1962	31 May, 1962
7	6	2	1962	1 June, 1962
8	6	30	1962	29 June, 1962
9	6	31	1962	Invalid date
10	1	15	1962	14 January, 1962
11	2	15	1962	14 February, 1962
12	11	15	1962	14 November, 1962
13	12	15	1962	14 December, 1962

Software Testing

Robustness testing

It is nothing but the extension of boundary value analysis Here, we would like to see, what happens when the extreme values are exceeded with a value slightly greater than the maximum, and a value slightly less than minimum It means, we want to go outside the legitimate boundary of input domain This extended form of boundary value analysis is called robustness testing and shown in Fig 5

There are four additional test cases which are outside the legitimate input domain Hence total test cases in robustness testing are 6n+1, where n is the number of input variables So, 13 test cases are:

(200,99), (200,100), (200,101), (200,200), (200,299), (200,300)

(200,301), (99,200), (100,200), (101,200), (299,200), (300,200), (301,200)

Software Testing

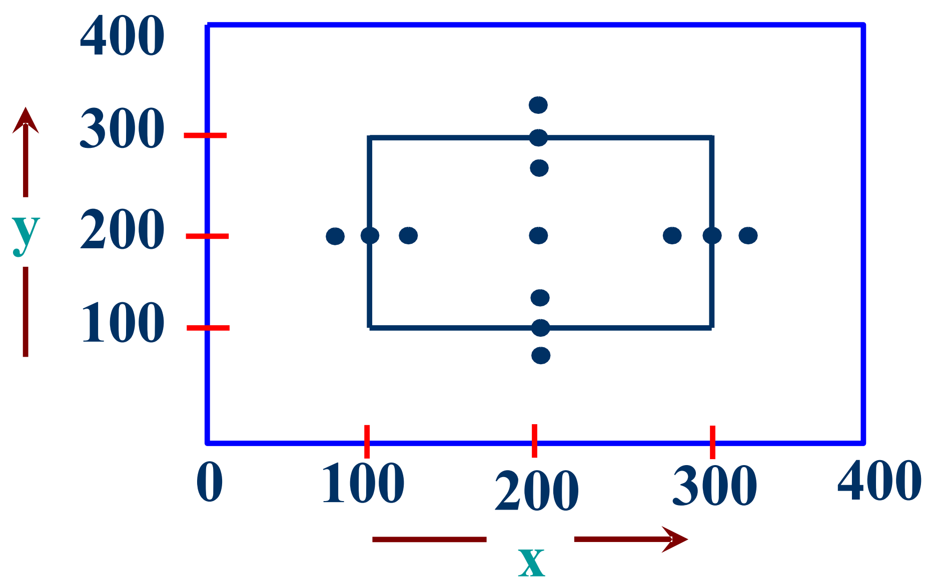

Fig 5: Robustness test cases for two variables x and y with range [100,300] each

Software Testing

Worst-case testing

If we reject "single fault" assumption theory of reliability and may like to see what happens when more than one variable has an extreme value In electronic circuits analysis, this is called "worst case analysis" It is more thorough in the sense that boundary value test cases are a proper subset of worst case test cases It requires more effort Worst case testing for a function of *n* variables generate 5^n test cases as opposed to $4n+1$ test cases for boundary value analysis Our two variables example will have $5^2=25$ test cases and are given in table 1

Software Testing

Table 1: Worst cases test inputs for two variables example

Test case number	Inputs		Test case number	Inputs	
	x	y		x	y
1	100	100	14	200	299
2	100	101	15	200	300
3	100	200	16	299	100
4	100	299	17	299	101
5	100	300	18	299	200
6	101	100	19	299	299
7	101	101	20	299	300
8	101	200	21	300	100
9	101	299	22	300	101
10	101	300	23	300	200
11	200	100	24	300	299
12	200	101	25	300	300
13	200	200	—		

Software Testing

Cyclomatic complexity

Cyclomatic complexity is a source code complexity measurement that is being correlated to a number of coding errors. It is calculated by developing a Control Flow Graph of the code that measures the number of linearly-independent paths through a program module.

Software Testing

Cyclomatic Complexity

McCabe‟s cyclomatic metric V(G) = *e* – *n* + 2P

For example, a flow graph shown in fig with entry node ‚*a*‟ and exit node ‚*f*‟

Cyclomatic complexity = E - N + 2*P where, E = number of edges in the flow graph. N = number of nodes in the flow graph. P = number of nodes that have exit points

A **cyclomatic complexity** below 4 is considered **good**; a **cyclomatic complexity** between 5 and 7 is considered medium **complexity**, between 8 and 10 is high **complexity**, and above that is extreme **complexity**.

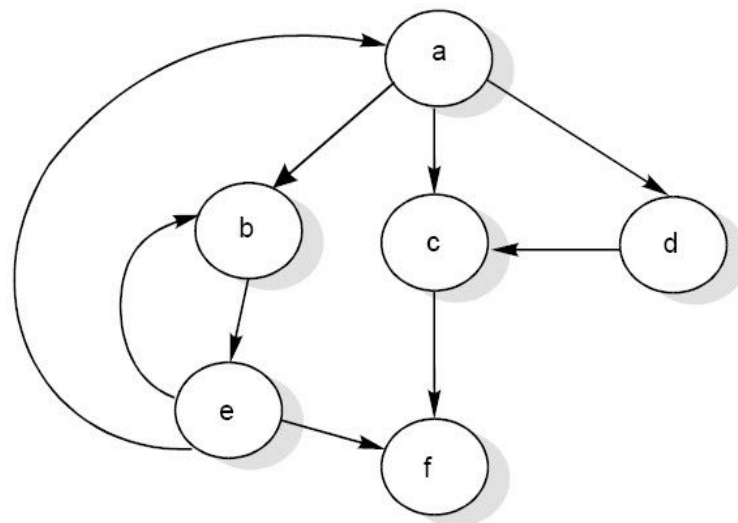

Fig :2 Flow graph

Software Testing

The value of cyclomatic complexity can be calculated as :

$$V(G) = 9 – 6 + 2 = 5$$

Here $e = 9$, $n = 6$ and $P = 1$

There will be five independent paths for the flow graph illustrated in Fig 2

Path 1 : *a c f*

Path 2 : *a b e f*

Path 3 : *a d c f*

Path 4 : *a b e a c f* **or** *a b e a b e f*

Path 5 : *a b e b e f*

```
function fizzBuzz($start, $max) {

    $return = array();

    // sanity check
    if($max < $start) {
        return false;
    }

    for($i = $start; $i < $max; $i++) {
        $result = '';
        if($i % 3 == 0) {
            $result .= 'fizz';
        }

        if($i % 5 == 0) {
            $result .= 'buzz';
        }

        if(!$result) {
            $result = $i;
        }

        $return[] = $result;
    }

    return implode($return, ',');
}
```

Complexity = 6

```
function fizzBuzz($start, $max) {

    $return = array();

    // sanity check
    if($max < $start) {
        return false;
    }

    for($i = $start; $i < $max;
$i++) {
        $return[] =
determineFizzandBuzz($i);
    }

    return implode($return, ',');

}
```

```
function determineFizzandBuzz($value) {
    $result = '';

    if($value % 3 == 0) {
        $result .= 'fizz';
    }

    if($value % 5 == 0) {
        $result .= 'buzz';
    }

    if(!$result) {
        $result = $value;
    }

    return $result;
}
```