Software Testing and Reliability

Dr. Mritunjay Shall Peelam
Assistant Professor-SG, UPES

Course Introduction and Objectives

UNIVERSITY OF PETROLEUM & ENERGY STUDIES

2021-22 Batch

CSEG 4014P	**Software Testing and Reliability**	**L**	**T**	**P**	**C**
Version 1.0		3	0	0	3
Pre-requisites/Exposure	Software Engineering				
Co-requisites	–				

Course Objectives

1. To apply methods and techniques to analyze requirements, and design and code software artifacts to assess and model software reliability.
2. To select and apply in autonomy appropriate technologies and techniques for different types of testing all over the software process development in different domains and contexts.

Course Introduction and Objectives

Course Outcomes

On completion of this course, the students will be able to

CO1. know principles and methods of software testing.
CO2. Discuss the testing approaches and test management.
CO3. Apply methods to develop software artifacts to assess and model software reliability.
CO4. Use the analysis tools that semi-automatically generate tests to validate a specification.

Catalog Description

This course provides in-depth coverage of testing and other software validation techniques intended to produce reliable, correct software. Topics include formal and informal specification, core testing techniques, approaches to comparing quality of test suites, principles of reasoning about software correctness, designing software to support validation, and using tools to automatically validate or find bugs in software.

Course Introduction and Objectives

Course Content

Unit 1: Introduction **8 lecture hours**

Human and errors, Defects, Faults, Failures, Defect Rate and Reliability, Defect Prevention, Reduction and Containment, Testing and Debugging, Software Quality, Requirement Behavior and Correctness, Fundamentals of Test Process, Psychology of Testing, General Principles of Testing, Test Metrics, Agile Methodology and Its Impact on testing, Test Levels: Unit, Component, Module, Integration, System, Acceptance, Generic

Course Introduction and Objectives

Unit 2: Testing approaches and test management **12 lecture hours**

Static Testing, Structured Group Examinations, Static Analysis, Control flow & Data flow, Determining Metrics, Dynamic Testing, Black Box Testing: Equivalence Class Partitioning, Boundary Value Analysis, State Transition Test, Cause Effect Graphing and Decision Table Technique. Used Case Testing and Advanced black box techniques, White Box Testing: Statement Coverage, Branch Coverage, Path Coverage, System integration, Deployment testing, Beta testing, Scalability testing, Reliability testing, Stress testing, Acceptance testing: Acceptance criteria, test cases selection and execution.

Course Introduction and Objectives

UNIVERSITY OF PETROLEUM & ENERGY STUDIES

2021-22 Batch

Test Organization, Test teams, tasks and Qualifications, Test Planning, Quality Assurance Plan, Test Plan, Prioritization Plan, Test Exit Criteria, Cost and economy Aspects, Test Strategies. Test Activity Management, Incident Management, Configuration Management, Test Progress Monitoring and Control.

Unit 3: Software Reliability **12 lecture hours**

Defining Software Reliability, Software Reliability Attributes and Specification, Basics of Reliability Theory, Software Reliability Problem, Modeling Process, Software Reliability Models (SRGM), preliminary Concepts of Reliability Engineering. Software reliability growth models- Execution Time Models, Calendar Time Models, Erlang Model, Modeling Fault Dependency and Debugging Time Lag, Testing Effort Dependent Modeling, Software reliability allocation models.

Course Introduction and Objectives

Unit 4: Software Verification, Validation and Testing 4 lecture hours

Verification and Validation, Evolutionary Nature of Verification and Validation, Impracticality of Testing all Data and Paths, Proof of Correctness, Software Testing, static and Dynamic Testing Tools, Characteristics of Modern Testing Tools.

Course Text Book and Reference Book

Text Books

Andreas Spillner, Tilo Linz and Hans Schaefer; "Software Testing Foundations", Shroff Publishers and Distributors

References

1. D Srinivasan and R Gopalswamy; "Software Testing: Principles and Practices", Pearson Education, 2006.
2. Claude Y. Laporte and Alain April. Software Quality Assurance, First Edition, Wiley.

Modes of Evaluation: Quiz/Assignment/ presentation/ extempore/ Written Examination

Examination Scheme:

Components	MSE	Presentation/Assignment/ etc	ESE
Weightage (%)	20%	30%	50%

Course Text Book and Reference Book

Relationship between the Course Outcomes (COs) and Program Outcomes (POs) and Program Specific Outcomes (PSOs)

PO/CO	PO1	PO2	PO3	PO 4	PO 5	PO6	PO 7	PO8	PO9	PO 10	PSO1	PSO2	PSO1	PSO2
CO1			1		1									2
CO2			1		1									2
CO3			1	1	2									2
CO4			1	1	3									2

1 = weak 2 = moderate 3 = strong

Joining What'sApp Group

Unit 2: Testing approaches and test management **12 lecture hours**

Static Testing, Structured Group Examinations, Static Analysis, Control flow & Data flow, Determining Metrics, Dynamic Testing, Black Box Testing: Equivalence Class Partitioning, Boundary Value Analysis, State Transition Test, Cause Effect Graphing and Decision Table Technique. Used Case Testing and Advanced black box techniques, White Box Testing: Statement Coverage, Branch Coverage, Path Coverage, System integration, Deployment testing, Beta testing, Scalability testing, Reliability testing, Stress testing, Acceptance testing: Acceptance criteria, test cases selection and execution.

- 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.

A more appropriate definition is:

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

- # 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.

- 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.

Some Terminologies

➢ Error, Mistake, Bug, Fault and Failure

People make **errors**. A good synonym is **mistake**. This may be a syntax error or misunderstanding of specifications. Sometimes, there are logical errors.

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

A **fault** is the representation of an error, where representation is the mode of expression, such as narrative text, data flow diagrams, ER diagrams, source code etc. Defect is a good synonym for fault.

A **failure** occurs when a fault executes. A particular fault may cause different failures, depending on how it has been exercised.

➢ 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.

➢ Verification and Validation

Verification is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

Validation is the process of evaluating a system or component during or at the end of development process to determine whether it satisfies the specified requirements .

Testing= Verification+Validation

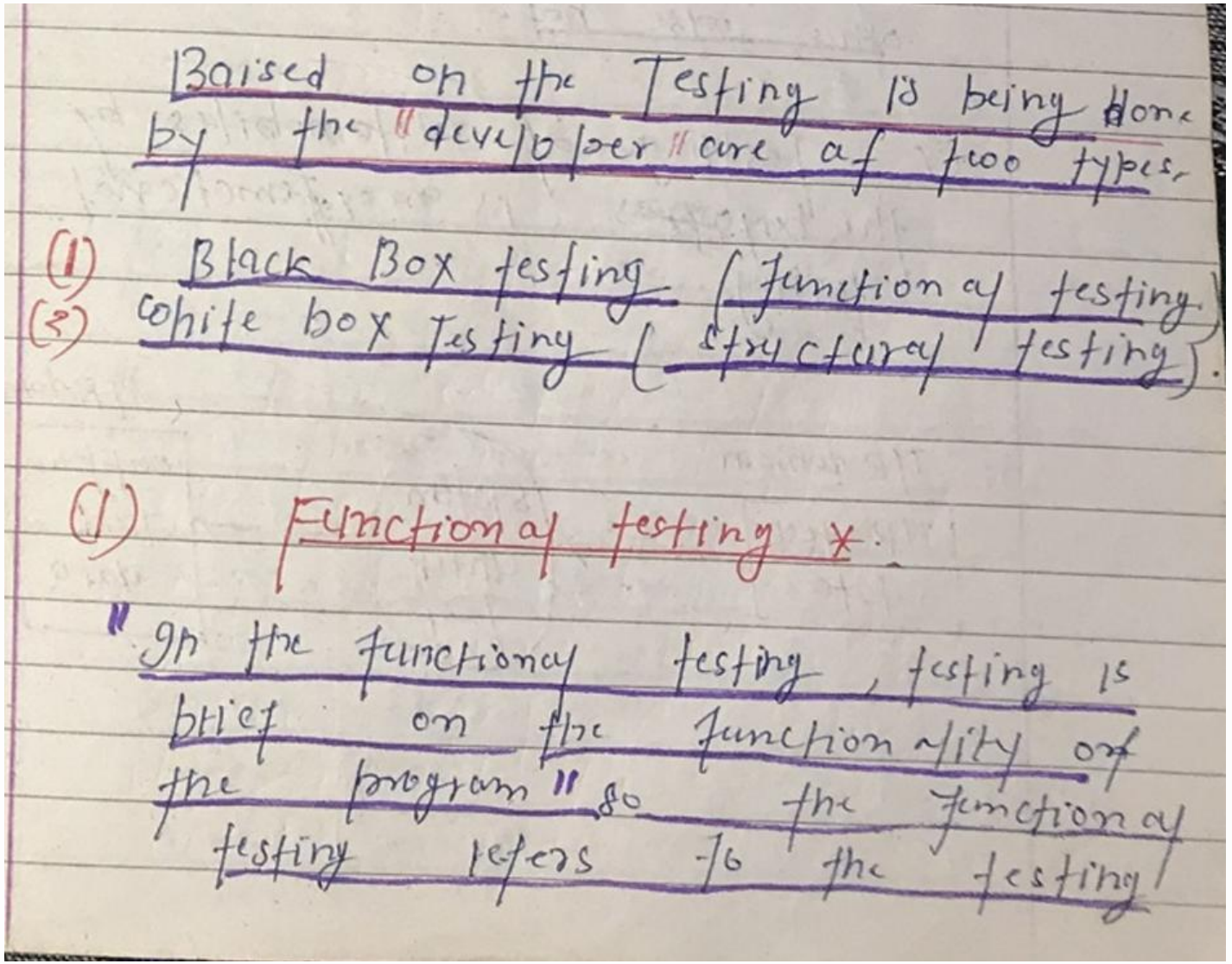
➢ Alpha, Beta and Acceptance Testing

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.

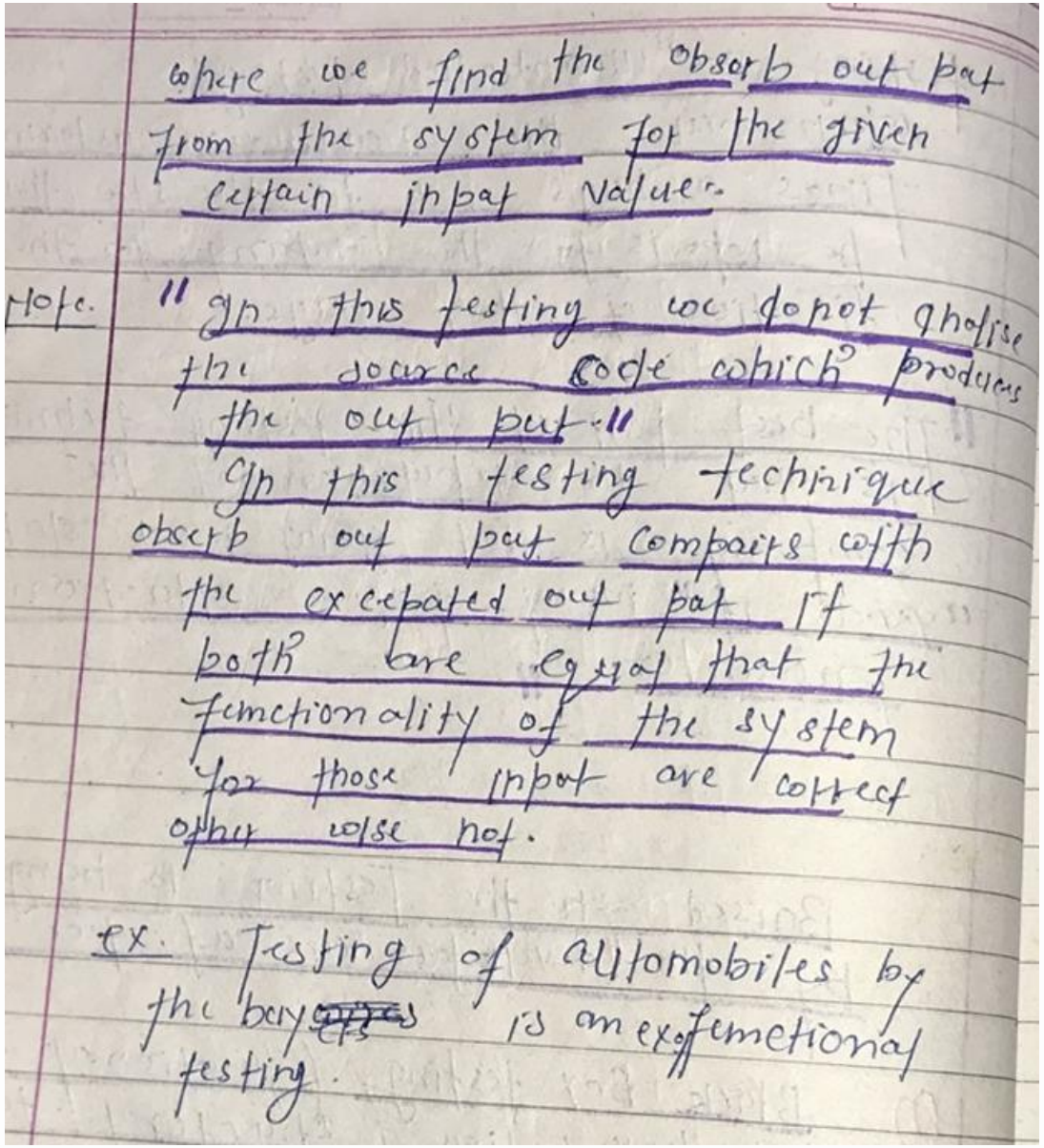
Beta Tests are conducted by the customers / end users at their sites. Unlike alpha testing, developer is not present here. Beta testing is conducted in a real environment that cannot be controlled by the developer.


Baised on the Testing is being done by the "developer" are of two types,

(1) Black Box testing (function al testing)
(2) White box Testing (structural testing).

(1) Functional testing *.

"In the functional testing, testing is biief on the functionality of the program" so the functional testing refers to the testing


where we find the observ out put from the system for the given certain input value.

Note. "In this testing we do not analyse the source code which produces the out put."

In this testing technique observ out put compairs with the excepated out put If both are equal that the functionality of the system for those input are correct other wise not.

ex. Testing of automobiles by the buyers is an exofunctional testing.

Functional Testing

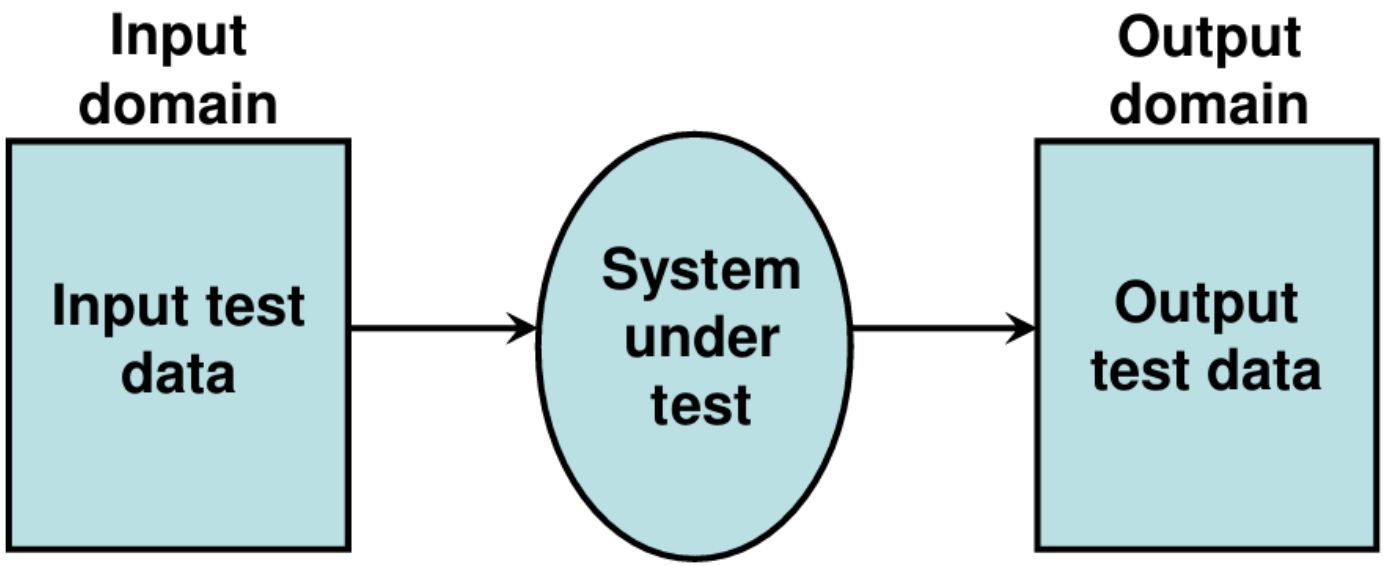
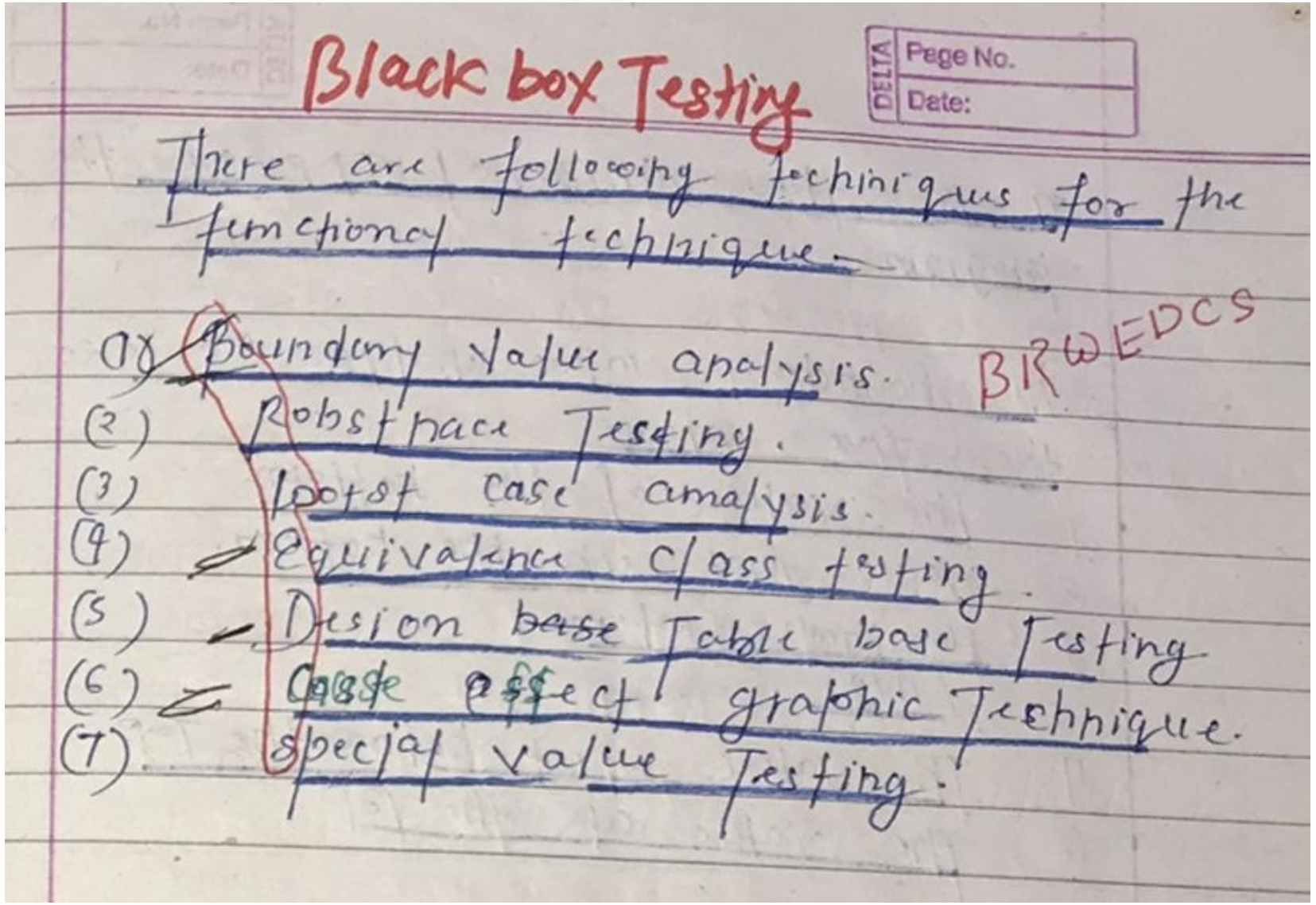
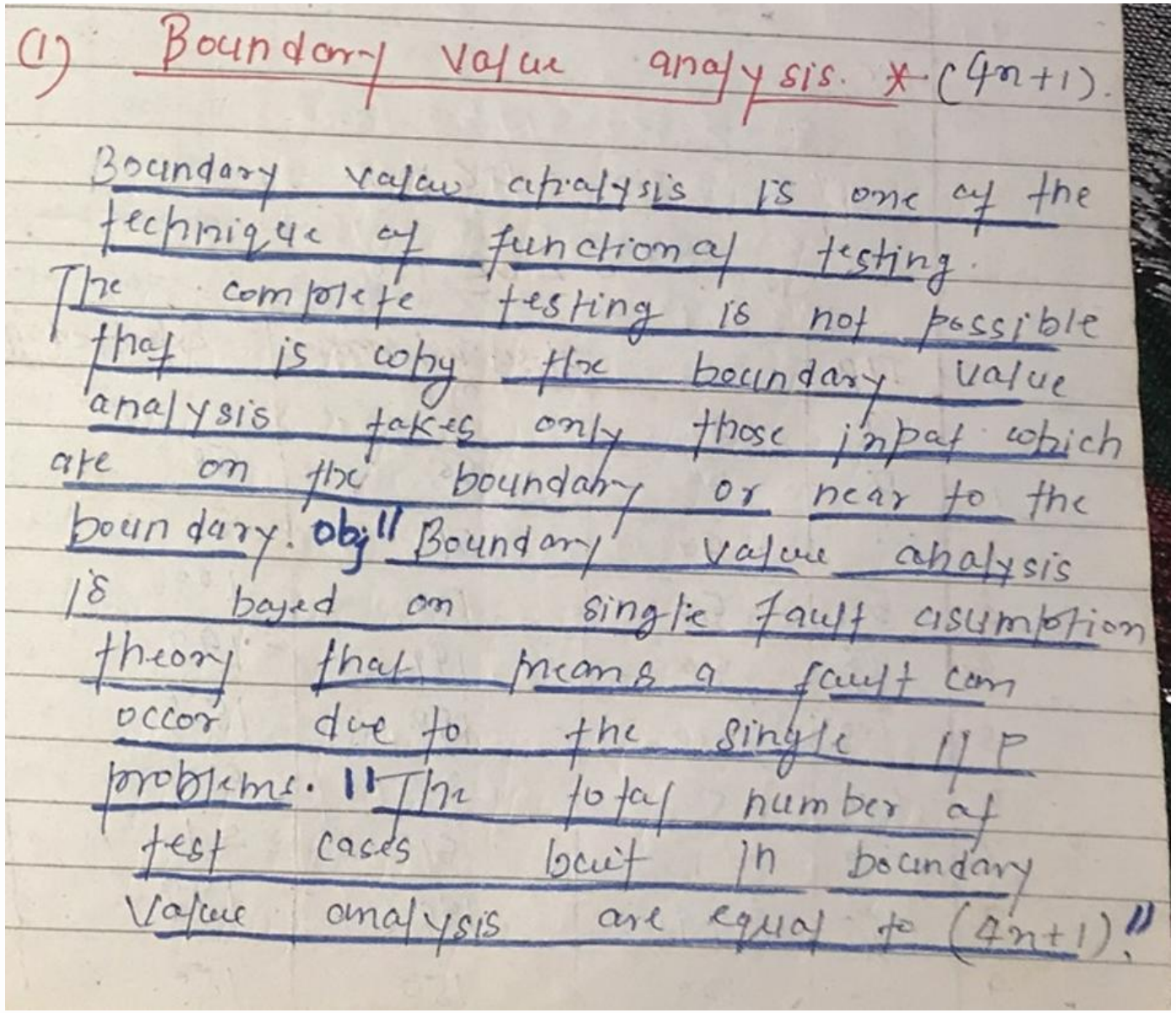
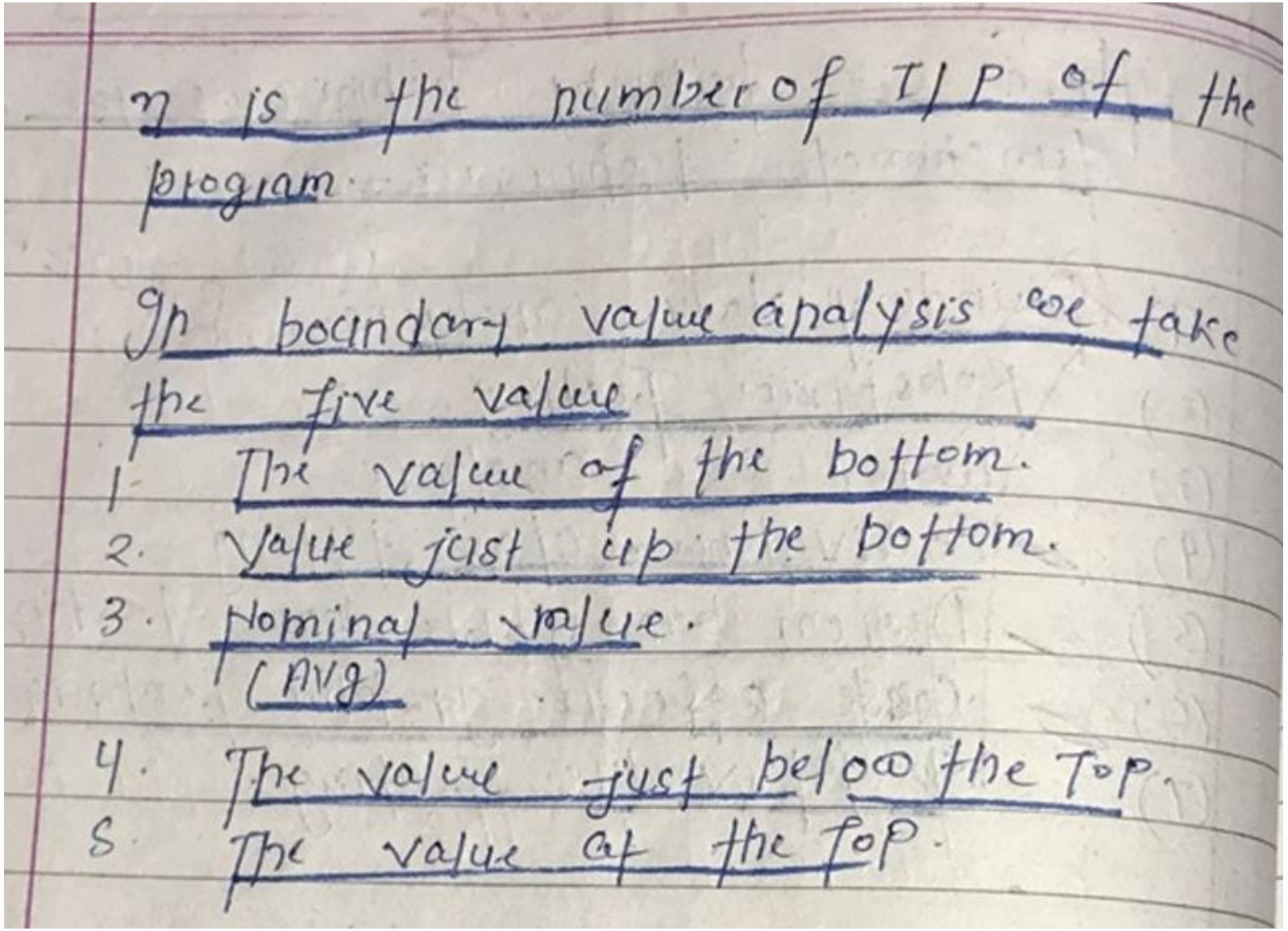


Fig. 3: Black box testing


Black box Testing
DELTA
Page No.
Date:
There are following techniques for the functional technique.
(1) Boundary value analysis.
(2) Robustness Testing.
(3) Worst case analysis.
(4) Equivalence class testing.
(5) Decision base Table base Testing
(6) Cause effect graphic Technique.
(7) special value Testing.
BRWEDCS


(1) Boundary value analysis. * (4n+1).

Boundary value analysis is one of the technique of functional testing. The complete testing is not possible that is why the boundary value analysis takes only those input which are on the boundary or near to the boundary. obj" Boundary value analysis is based on single fault assumption theory that means a fault can occur due to the single I/P problems. " The total number of test cases built in boundary value analysis are equal to (4n+1)."


n is the number of I/P of the program.

In boundary value analysis we take the five value

1. The value of the bottom.
2. Value just up the bottom.
3. Nominal value. (AVG)
4. The value just below the Top.
5. The value at the Top.

Boundary Value Analysis

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

$$a \leq x \leq b$$

$$c \leq y \leq d$$

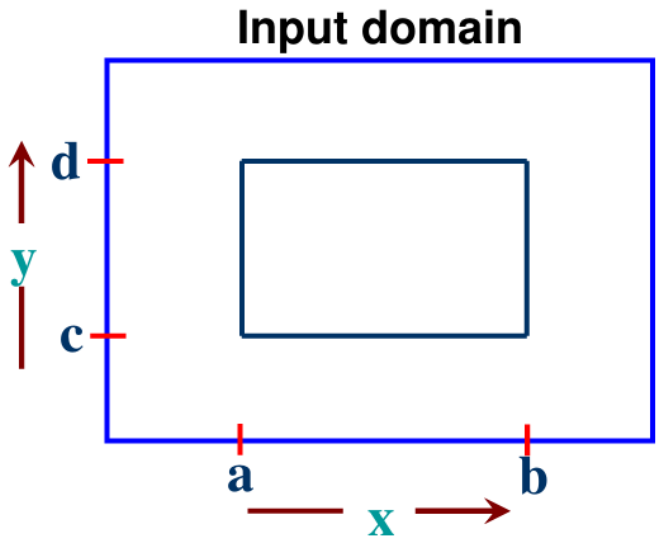
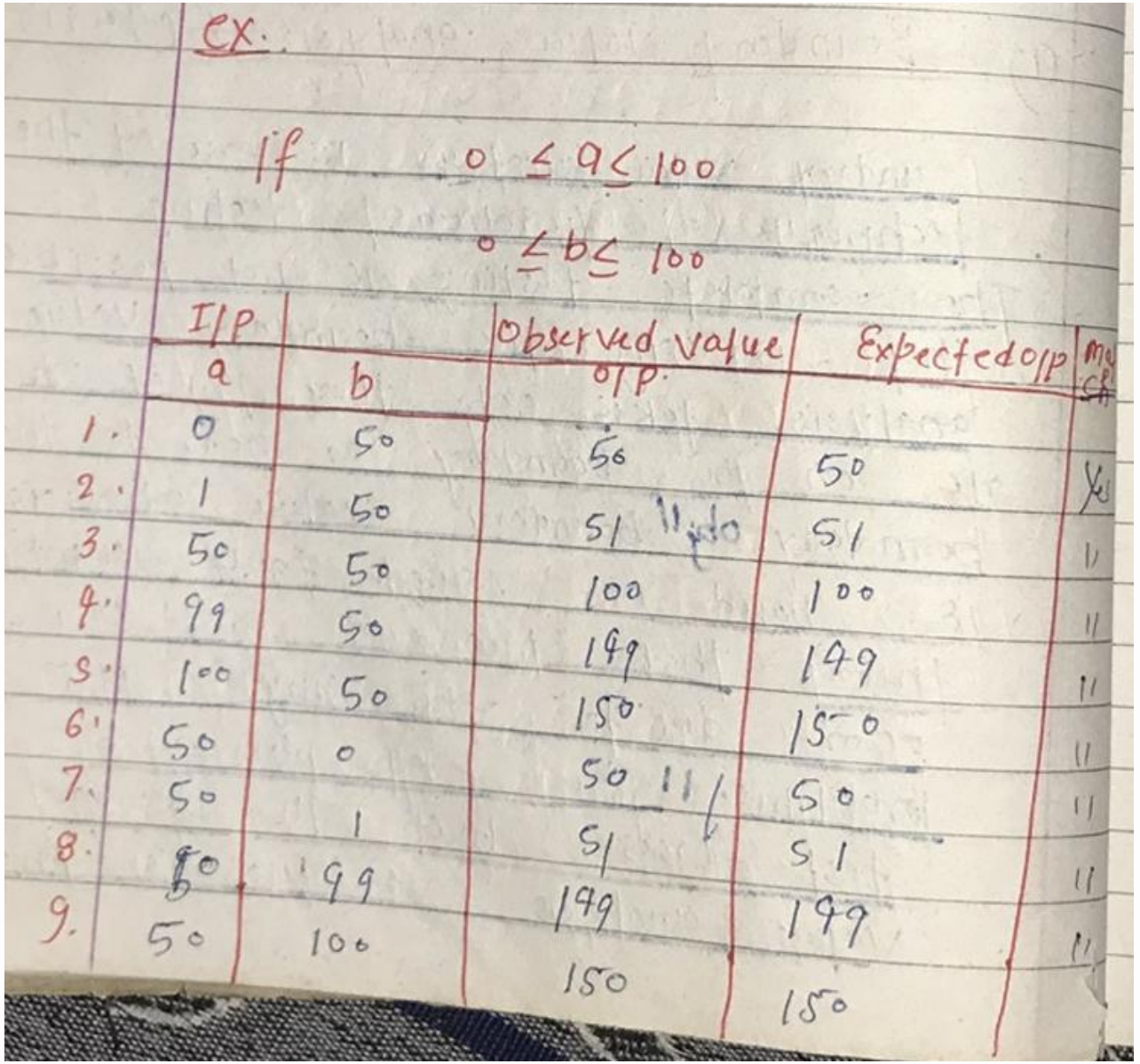


Fig.4: Input domain for program having two input variables



ex.

if $0 \leq a \leq 100$

$0 \leq b \leq 100$

	I/P a	I/P b	observed value o/p	Expected o/p	Mo. CR
1.	0	50	50	50	
2.	1	50	51	51	
3.	50	50	100	100	
4.	99	50	149	149	
5.	100	50	150	150	
6.	50	0	50	50	
7.	50	1	51	51	
8.	50	99	149	149	
9.	50	100	150	150	

Example- 8.I

Consider a program for the determination of the nature of roots of a quadratic equation. Its input is a triple of positive integers (say a,b,c) and values may be from interval [0,100]. The program output may have one of the following words.

[Not a quadratic equation; Real roots; Imaginary roots; Equal roots]

Design the boundary value test cases.

Solution

Quadratic equation will be of type:

$$ax^2+bx+c=0$$

Roots are real if $(b^2-4ac)>0$

Roots are imaginary if $(b^2-4ac)<0$

Roots are equal if $(b^2-4ac)=0$

Equation is not quadratic if $a=0$

The boundary value test cases are :

Test Case	a	b	c	Expected output
1	0	50	50	Not Quadratic
2	1	50	50	Real Roots
3	50	50	50	Imaginary Roots
4	99	50	50	Imaginary Roots
5	100	50	50	Imaginary Roots
6	50	0	50	Imaginary Roots
7	50	1	50	Imaginary Roots
8	50	99	50	Imaginary Roots
9	50	100	50	Equal Roots
10	50	50	0	Real Roots
11	50	50	1	Real Roots
12	50	50	99	Imaginary Roots
13	50	50	100	Imaginary Roots

Example – 8.2

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

$1 \leq month \leq 12$

$1 \leq day \leq 31$

$1900 \leq year \leq 2025$

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

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.

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

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. 6

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)

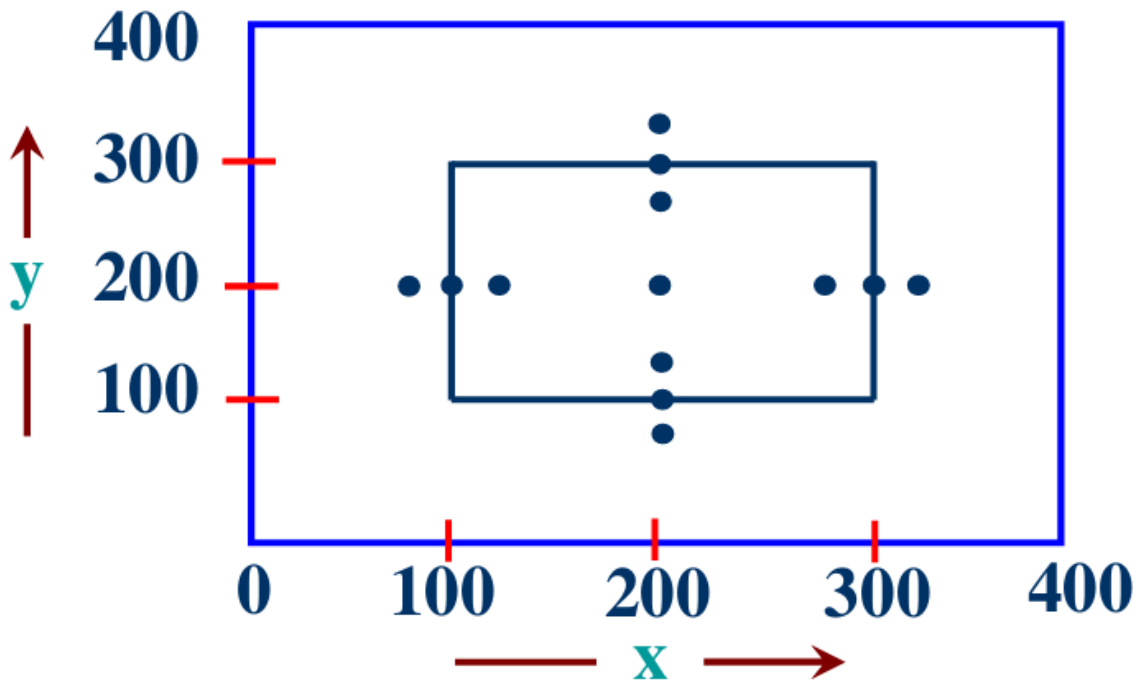


Fig. 8.6: Robustness test cases for two variables x and y with range [100,300] each

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.

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	--		

Example - 8.4

Consider the program for the determination of nature of roots of a quadratic equation as explained in example 8.1. Design the Robust test case and worst test cases for this program.

Solution

Robust test cases are 6n+1. Hence, in 3 variable input cases total number of test cases are 19 as given on next slide:

Test case	a	b	c	Expected Output
1	-1	50	50	Invalid input`
2	0	50	50	Not quadratic equation
3	1	50	50	Real roots
4	50	50	50	Imaginary roots
5	99	50	50	Imaginary roots
6	100	50	50	Imaginary roots
7	101	50	50	Invalid input
8	50	-1	50	Invalid input
9	50	0	50	Imaginary roots
10	50	1	50	Imaginary roots
11	50	99	50	Imaginary roots
12	50	100	50	Equal roots
13	50	101	50	Invalid input
14	50	50	-1	Invalid input
15	50	50	0	Real roots
16	50	50	1	Real roots
17	50	50	99	Imaginary roots
18	50	50	100	Imaginary roots
19	50	50	101	Invalid input

In case of worst test case total test cases are 5^n. Hence, 125 test cases will be generated in worst test cases. The worst test cases are given below:

Test Case	a	b	c	Expected output
1	0	0	0	Not Quadratic
2	0	0	1	Not Quadratic
3	0	0	50	Not Quadratic
4	0	0	99	Not Quadratic
5	0	0	100	Not Quadratic
6	0	1	0	Not Quadratic
7	0	1	1	Not Quadratic
8	0	1	50	Not Quadratic
9	0	1	99	Not Quadratic
10	0	1	100	Not Quadratic
11	0	50	0	Not Quadratic
12	0	50	1	Not Quadratic
13	0	50	50	Not Quadratic
14	0	50	99	Not Quadratic

(Contd.)...

Test Case	A	b	c	Expected output
15	0	50	100	Not Quadratic
16	0	99	0	Not Quadratic
17	0	99	1	Not Quadratic
18	0	99	50	Not Quadratic
19	0	99	99	Not Quadratic
20	0	99	100	Not Quadratic
21	0	100	0	Not Quadratic
22	0	100	1	Not Quadratic
23	0	100	50	Not Quadratic
24	0	100	99	Not Quadratic
25	0	100	100	Not Quadratic
26	1	0	0	Equal Roots
27	1	0	1	Imaginary
28	1	0	50	Imaginary
29	1	0	99	Imaginary
30	1	0	100	Imaginary
31	1	1	0	Real Roots

(Contd.)...

Test Case	A	b	C	Expected output
32	1	1	1	Imaginary
33	1	1	50	Imaginary
34	1	1	99	Imaginary
35	1	1	100	Imaginary
36	1	50	0	Real Roots
37	1	50	1	Real Roots
38	1	50	50	Real Roots
39	1	50	99	Real Roots
40	1	50	100	Real Roots
41	1	99	0	Real Roots
42	1	99	1	Real Roots
43	1	99	50	Real Roots
44`	1	99	99	Real Roots
45	1	99	100	Real Roots
46	1	100	0	Real Roots
47	1	100	1	Real Roots
48	1	100	50	Real Roots

(Contd.)...

Test Case	A	b	C	Expected output
49	1	100	99	Real Roots
50	1	100	100	Real Roots
51	50	0	0	Equal Roots
52	50	0	1	Imaginary
53	50	0	50	Imaginary
54	50	0	99	Imaginary
55	50	0	100	Imaginary
56	50	1	0	Real Roots
57	50	1	1	Imaginary
58	50	1	50	Imaginary
59	50	1	99	Imaginary
60	50	1	100	Imaginary
61	50	50	0	Real Roots
62	50	50	1	Real Roots
63	50	50	50	Imaginary
64	50	50	99	Imaginary
65	50	50	100	Imaginary

(Contd.)...

Test Case	A	b	C	Expected output
66	50	99	0	Real Roots
67	50	99	1	Real Roots
68	50	99	50	Imaginary
69	50	99	99	Imaginary
70	50	99	100	Imaginary
71	50	100	0	Real Roots
72	50	100	1	Real Roots
73	50	100	50	Equal Roots
74	50	100	99	Imaginary
75	50	100	100	Imaginary
76	99	0	0	Equal Roots
77	99	0	1	Imaginary
78	99	0	50	Imaginary
79	99	0	99	Imaginary
80	99	0	100	Imaginary
81	99	1	0	Real Roots
82	99	1	1	Imaginary

Test Case	A	b	C	Expected output
83	99	1	50	Imaginary
84	99	1	99	Imaginary
85	99	1	100	Imaginary
86	99	50	0	Real Roots
87	99	50	1	Real Roots
88	99	50	50	Imaginary
89	99	50	99	Imaginary
90	99	50	100	Imaginary
91	99	99	0	Real Roots
92	99	99	1	Real Roots
93	99	99	50	Imaginary Roots
94	99	99	99	Imaginary
95	99	99	100	Imaginary
96	99	100	0	Real Roots
97	99	100	1	Real Roots
98	99	100	50	Imaginary
99	99	100	99	Imaginary
100	99	100	100	Imaginary

Test Case	A	b	C	Expected output
101	100	0	0	Equal Roots
102	100	0	1	Imaginary
103	100	0	50	Imaginary
104	100	0	99	Imaginary
105	100	0	100	Imaginary
106	100	1	0	Real Roots
107	100	1	1	Imaginary
108	100	1	50	Imaginary
109	100	1	99	Imaginary
110	100	1	100	Imaginary
111	100	50	0	Real Roots
112	100	50	1	Real Roots
113	100	50	50	Imaginary
114	100	50	99	Imaginary
115	100	50	100	Imaginary
116	100	99	0	Real Roots
117	100	99	1	Real Roots
118	100	99	50	Imaginary

Test Case	A	b	C	Expected output
119	100	99	99	Imaginary
120	100	99	100	Imaginary
121	100	100	0	Real Roots
122	100	100	1	Real Roots
123	100	100	50	Imaginary
124	100	100	99	Imaginary
125	100	100	100	Imaginary

Example – 8.5

Consider the program for the determination of previous date in a calendar as explained in example 8.2. Design the robust and worst test cases for this program.

4.1.1 Statement Coverage

We want to execute every statement of the program in order to achieve 100% statement coverage. Consider the following portion of a source code along with its program graph given in Figure 4.1.

```
#include<stdio.h>
#include<conio.h>

1.   void main()
2.   {
3.   int a,b,c,x=0,y=0;
4.   clrscr();
5.   printf("Enter three numbers:");
6.   scanf("%d %d %d",&a,&b,&c);
7.   if((a>b)&&(a>c)){
8.            x=a*a+b*b;
9.   }
10.  if(b>c){
11.           y=a*a-b*b;
12.  }
13.  printf("x= %d y= %d",x,y);
14.  getch();
15.  }
```

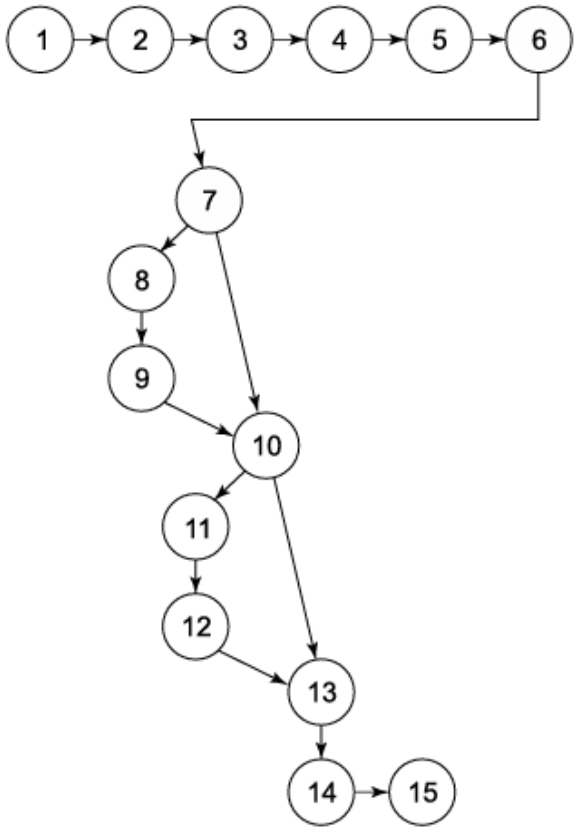


Figure 4.1. Source code with program graph

If, we select inputs like:

a=9, b=8, c=7, all statements are executed and we have achieved 100% statement coverage by only one test case. The total paths of this program graph are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

The cyclomatic complexity of this graph is:

$$V(G) = e - n + 2P = 16 - 15 + 2 = 3$$

$$V(G) = \text{no. of regions} = 3$$

$$V(G) = \Pi + 1 = 2 + 1 = 3$$

Hence, independent paths are three and are given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15

Only one test case may cover all statements but will not execute all possible four paths and not even cover all independent paths (three in this case).

The objective of achieving 100% statement coverage is difficult in practice. A portion of the program may execute in exceptional circumstances and some conditions are rarely possible, and the affected portion of the program due to such conditions may not execute at all.

4.1.2 Branch Coverage

We want to test every branch of the program. Hence, we wish to test every 'True' and 'False' condition of the program. We consider the program given in Figure 4.1. If we select a = 9, b = 8, c = 7, we achieve 100% statement coverage and the path followed is given as (all true conditions):

Path = 1–15

We also want to select all false conditions with the following inputs:

a = 7, b = 8, c = 9, the path followed is

Path = 1–7, 10, 13–15

These two test cases out of four are sufficient to guarantee 100% branch coverage. The branch coverage does not guarantee 100% path coverage but it does guarantee 100% statement coverage.

4.1.3 Condition Coverage

Condition coverage is better than branch coverage because we want to test every condition at least once. However, branch coverage can be achieved without testing every condition.

Consider the seventh statement of the program given in Figure 4.1. The statement number 7 has two conditions (a>b) and (a>c). There are four possibilities namely:

(i) Both are true
(ii) First is true, second is false
(iii) First is false, second is true
(iv) Both are false

If a > b and a > c, then the statement number 7 will be true (first possibility). However, if a < b, then second condition (a > c) would not be tested and statement number 7 will be false (third and fourth possibilities). If a > b and a < c, statement number 7 will be false (second possibility). Hence, we should write test cases for every true and false condition. Selected inputs may be given as:

(i) a = 9, b = 8, c = 7 (first possibility when both are true)
(ii) a = 9, b = 8, c = 10 (second possibility – first is true, second is false)
(iii) a = 7, b = 8, c = 9 (third and fourth possibilities- first is false, statement number 7 is false)

Hence, these three test cases out of four are sufficient to ensure the execution of every condition of the program.

4.1.4 Path Coverage

In this coverage criteria, we want to test every path of the program. There are too many paths in any program due to loops and feedback connections. It may not be possible to achieve this goal of executing all paths in many programs. If we do so, we may be confident about the correctness of the program. If it is unachievable, at least all independent paths should be executed. The program given in Figure 4.1 has four paths as given as:

(i) 1–7, 10, 13–15
(ii) 1–7, 10–15
(iii) 1–10, 13–15
(iv) 1–15

Execution of all these paths increases confidence about the correctness of the program. Inputs for test cases are given as:

Some paths are possible from the program graph, but become impossible when we give inputs as per logic of the program. Hence, some combinations may be found to be impossible to create.

Path testing guarantee statement coverage, branch coverage and condition coverage. However, there are many paths in any program and it may not be possible to execute all the paths. We should do enough testing to achieve a reasonable level of coverage. We should execute at least (minimum level) all independent paths which are also referred to as basis paths to achieve reasonable coverage. These paths can be found using any method of cyclomatic complexity.

We have to decide our own coverage level before starting control flow testing. As we go up (statement coverage to path coverage) in the ladder, more resources and time may be required.

Deployment Testing?

- Testing a software project before and after deploying it on production is not that difficult. But too often, major bugs appear on production server after the deployment process.
- To avoid situations in which your production environment is threatened by these bugs, you should use a streamlined deployment and testing flow.
- Master of Code is an Agile software development and Conversational AI company that specializes in crafting world-class web and mobile software products.
- The testing flow consists of three phases: pre-deploy, deploy and post-deploy.

Deployment Testing?

During the Pre-Deploy testing phase, both the web-development team and the QA engineer should be tasked with the following items:

- **Ask developers to make Production and Stage environment backups.**
- **Ask developers to copy database from Production to Stage.**
- **Announce a Code Freeze for the entire development team.**
- **Retest new features and bug fixes.**
- **Perform general smoke testing using checklists.**
- **Report all bugs, categorizing them with a consistent metric for urgency.**
- **Test urgent bug fixes and engage in regression testing of this functionality.**
- **When Stage is tested and has no major and/or minor bugs (depending on the project), – announce deployment.**

Deployment Testing?

Note: When the Quality Assurance (QA) engineer announces the Code Freeze, developers cannot push, commit, or deploy anything. But they can still continue working on their local environment, so no time is wasted. When the Code Freeze is cancelled, developers can push, commit, and deploy their work to Stage.

Note: After finishing the pre-deploy testing cycle using automated software testing tools, the manager has to approve the deploy process before the developer who is responsible for this process puts it into motion. Having one developer be responsible for deploys is always beneficial, as it avoids misunderstandings like incorrectly merged branches.

Deployment Testing?

After the deployment is done, QA begins the Post-Deploy phase, which consists of the following tasks:

- **Retesting new features and bug fixes.**
- **Perform general smoke testing using checklists.**
- **Report all bugs, categorizing them with a consistent metric for urgency.**
- **Urgent bug fixes should first be deployed on the Stage environment. Here, QA can test them, and engage in regression testing before deploying them on production.**
- **Then, the testing cycle – including bug fix testing and regression testing – should be repeated. When production is tested and has no issues, – deployment is complete.**
- **Cancel Code Freeze, or wait for client feedback as needed before cancelling Code Freeze.**

Different Types of Deployment Testing?

Compatibility testing:

- This type of testing ensures that the application is compatible with various hardware, operating systems, browsers, and other software components it interacts with. It verifies that the application functions seamlessly across different platforms.

Regression testing:

- Regression testing is conducted to ensure that the deployment of new features or changes does not introduce any unintended side effects or break existing functionality. It helps maintain the stability and reliability of the application.

Different Types of Deployment Testing?

API testing:

- **API tests validate the APIs of the application function in accordance with their contracts. Passing API tests indicate compatibility with other applications that depend on this one.**

Performance testing:

- **Performance testing evaluates the application's response time, scalability, and resource utilization under different load conditions. It helps identify any performance bottlenecks or issues that may affect the application's performance in the production environment.**

Different Types of Deployment Testing?

Resilience Testing (Chaos Testing):

- **For complex systems that are expected to withstand partial failures, resilience testing switches off services or reduces the performance of some components to ensure the rest of the system continues to operate.**

Security testing:

- **Dynamic application security testing (DAST) aims to identify vulnerabilities and ensure that the application is secure against potential threats. It includes testing for authentication, authorization, data encryption, and other security measures to protect sensitive information.**

Different Types of Deployment Testing?

Data migration testing:

- When migrating data from an existing system to a new one during deployment, data migration testing ensures the accuracy and integrity of the migrated data. It verifies that the data is transferred correctly and remains consistent throughout the migration process.

Rollback testing:

- Rollback testing verifies the ability to revert the deployment in case of any issues or failures. It ensures that the application can be rolled back to the previous version without data loss or adverse effects on the system.

Different Types of Deployment Testing?

User acceptance testing (UAT):

- **UAT involves involving end-users in the testing process to validate the application's functionality, usability, and overall user experience before deployment. It helps ensure that the application meets the users' requirements and expectations.**

Smoke testing:

- **Smoke testing is performed to quickly assess the basic functionality of the application after deployment. It aims to identify any critical issues or errors that may prevent further testing or usage of the application.**

Different Types of Deployment Testing?

Disaster recovery testing:

- Disaster recovery testing evaluates the ability to recover the application and its data in the event of a disaster or system failure. It ensures that appropriate backup and recovery mechanisms are in place.

Scalability Testing

- Scalability Testing is a type of non-functional testing in which the performance of a software application, system, network or process is tested in terms of its capability to scale up or scale down the number of user request load or other such performance attributes.
- It can be carried out at a hardware, software or database level.
- Scalability Testing is defined as the ability of a network, system, application, product or a process to perform the function correctly when changes are made in the size or volume of the system to meet a growing need.
- It ensures that a software product can manage the scheduled increase in user traffic, data volume, transaction counts frequency and many other things.
- It tests the system, processes or database's ability to meet a growing need.

Scalability Testing

- **Scalability Testing is to measure at what point the software product or the system stops scaling and identify the reason behind it.**
- **The parameters used for this testing differs from one application to another.**
- **For example, scalability testing of a web page depends on the number of users, CPU usage, network usage while scalability testing of a web server depends on the number of requests processed.**

Objective of Scalability Testing:

- To determine how the application scales with increasing workload.
- To determine the user limit for the software product.
- To determine client-side degradation and end user experience under load.
- To assess the system's performance under various network circumstances, such as latency and bandwidth fluctuations, in order to guarantee dependable operation in a range of settings.
- To determine whether the system is capable of withstanding scenarios of high usage, making sure that unexpected spikes in traffic can be accommodated without causing performance issues.
- To guarantee that the system's scalability prevents performance decline and maintains acceptable response times, both of which improve user experience.
- To determine server-side robustness and degradation.
- To help developers improve the system design or code by pointing out locations that could become bottlenecks when the load grows.

Objective of Scalability Testing:

- To evaluate the effective use of system resources, including CPU, memory and network bandwidth, in relation to the system's increasing load, in order to guarantee resource management.
- To make that the system satisfies performance criteria and offers a satisfying user experience, assess the system's response time under various loads.

Scalability Testing Attributes:

- **Response Time:** Response time is the time consumed between the user's request and the application's response. Response time may increase or decrease based on different user load on the application. Basically, the response time of an application decreases as the user load increases. Application having the lesser response time is considered as the higher performance application.
- **Throughput:** Throughput is the measurement of number of requests processed in a unit time by the application. It differs from one application to another as in web application it is measured in number of user requests processed in a unit time whereas in database application it is measured in number of queries processed in a unit time.

Scalability Testing Attributes:

- **Performance measurement with number of users:** Depending on the application type, it is always tested for the number of users that it can support without its breakdown or busy standby situation.
- **Threshold load:** Threshold load is the number of requests or transactions the application can process with desired throughput.
- **CPU Usage:** CPU Usage is the measurement of the CPU utilization while executing application code instructions. It is basically measured in terms of the unit Megahertz.
- **Memory Usage:** Memory usage is the measurement of the memory consumed for performing a task by an application. It is basically measured in terms of the unit bytes.
- **Network Usage:** Network usage is the measurement of the bandwidth consumed by an application under test. It is measured in terms of bytes received per second, frames received per second, segments received and sent per second etc.

Advantages of Scalability Testing:

- It provides more accessibility to the product.
- It detects issues with web page loading and other performance issues.
- It finds and fixes the issues earlier in the product which saves a lot of time.
- It ensures the end user experience under the specific load. It provides customer satisfaction.
- It helps in effective tool utilization tracking.

Disadvantages of Scalability Testing:

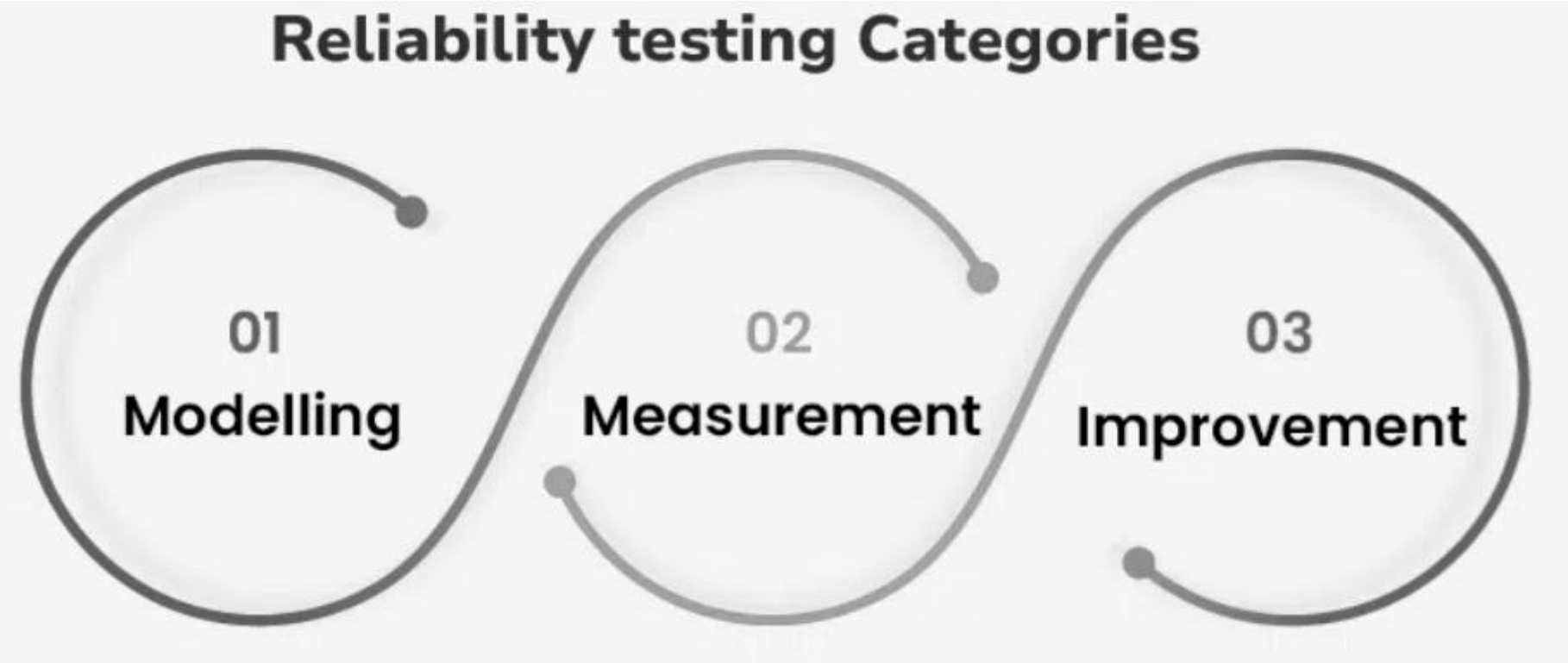
- Sometimes, it fails to find the functional errors or issues in the product.
- Some automation tools used for Scalability testing is costlier which ultimately increases the budget of the product.
- Team members involve in this testing technique should have high level of testing skills.
- The time spent on testing some parts of product may consume more time than expected time.
- Unexpected results may also be raised after launching the product in the customer environment.

Reliability testing

- Reliability testing is a Type of software testing that evaluates the ability of a system to perform its intended function consistently and without failure over an extended period.

- Reliability testing aims to identify and address issues that can cause the system to fail or become unavailable.
- It is defined as a type of software testing that determines whether the software can perform a failure-free operation for a specific period in a specific environment.
- It ensures that the product is fault-free and is reliable for its intended purpose.
- It is an important aspect of software testing as it helps to ensure that the system will be able to meet the needs of its users over the long term.
- It can also help to identify issues that may not be immediately apparent during functional testing, such as memory leaks or other performance issues.

Reliability testing



Reliability testing

Modelling

- Modelling in reliability testing involves creating mathematical or statistical representations of how a product or system might fail over time.
- It's like making an educated guess about the product's lifespan based on its design and components.

- This helps predict when and how failures might occur without actually waiting for the product to fail in real life.

- Example: Engineers might create a model to estimate how long a new smartphone battery will last before it degrades significantly.

Reliability testing

Measurement

- Measurement focuses on collecting real-world data about a product's performance and failures. This involves testing products under various conditions and recording when and how they fail. It's about gathering concrete evidence of reliability rather than just predictions.

- **Example:** A car manufacturer might test drive hundreds of cars for thousands of miles, recording any issues that arise during these tests.

Reliability testing

Improvement

- Improvement uses the insights gained from modelling and measurement to enhance the reliability of a product or system.
- This involves identifying weak points, redesigning components, or changing manufacturing processes to make the product more reliable.
- **Example:** After finding that a particular part in a washing machine fails frequently, engineers might redesign that part or choose a more durable material to improve its lifespan.

Different Ways to Perform Reliability Testing

Stress testing: Stress testing involves subjecting the system to high levels of load or usage to identify performance bottlenecks or issues that can cause the system to fail.

Endurance testing: Endurance testing involves running the system continuously for an extended period to identify issues that may occur over time.

Recovery testing: Recovery testing is testing the system's ability to recover from failures or crashes.

Environmental Testing: Conducting tests on the product or system in various environmental settings, such as temperature shifts, humidity levels, vibration exposure or shock exposure, helps in evaluating its dependability in real-world circumstances.

Different Ways to Perform Reliability Testing

Performance Testing: In Performance Testing It is possible to make sure that the system continuously satisfies the necessary specifications and performance criteria by assessing its performance at both peak and normal load levels.

Regression Testing: In Regression Testing After every update or modification, the system should be tested again using the same set of test cases to help find any potential problems caused by code changes.

Fault Tree Analysis: Understanding the elements that lead to system failures can be achieved by identifying probable failure modes and examining the connections between them.

Objective of Reliability Testing

- To find the perpetual structure of repeating failures.
- To find the number of failures occurring is the specific period of time.
- To discover the main cause of failure.
- To conduct performance testing of various modules of software product after fixing defects.
- It builds confidence in the market, stakeholders and users by providing a dependable product that meets quality criteria and operates as expected.
- Understanding the dependability characteristics and potential mechanisms of failure of the system helps companies plan and schedule maintenance actions more efficiently.
- It evaluates whether a system or product can be used continuously without experiencing a major loss in dependability, performance or safety.
- It confirms that in the absence of unexpected shutdown or degradation, the system or product maintains constant performance levels under typical operating settings.

Measurement of Reliability Testing

- **Mean Time Between Failures (MTBF): Measurement of reliability testing is done in terms of mean time between failures (MTBF).**
- **Mean Time To Failure (MTTF): The time between two consecutive failures is called as mean time to failure (MTTF).**
- **Mean Time To Repair (MTTR): The time taken to fix the failures is known as mean time to repair (MTTR).**
- **MTBF = MTTF + MTTR**

Measurement of Reliability Testing

$$\text{MTTF} = \frac{\text{Total Uptime}}{\text{Number of Failures}}$$

$$\text{MTTR} = \frac{\text{Total Repair Time}}{\text{Number of Failures}}$$

$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

Measurement of Reliability Testing

Q. A software system runs for 1000 hours and experiences 8 failures during this period. The total repair time recorded is 80 hours.

Calculate:
a) Mean Time To Failure (MTTF)
b) Mean Time To Repair (MTTR)
c) Mean Time Between Failures (MTBF)

Solution:

Calculate Total Uptime

$$\text{Total } Uptime = Total\ Time - Total\ Repair\ Time$$

$$\text{Total } Uptime = 1000 - 80 = 920\ hours$$

Measurement of Reliability Testing

Q. A software system runs for 1000 hours and experiences 8 failures during this period. The total repair time recorded is 80 hours.

Calculate:
a) Mean Time To Failure (MTTF)
b) Mean Time To Repair (MTTR)
c) Mean Time Between Failures (MTBF)

Solution:

Calculate Total Uptime

$$\text{Total } Uptime = Total\ Time - Total\ Repair\ Time$$

$$\text{Total } Uptime = 1000 - 80 = 920\ hours$$

Measurement of Reliability Testing

Q. A software system runs for 1000 hours and experiences 8 failures during this period. The total repair time recorded is 80 hours.

Calculate:
a) Mean Time To Failure (MTTF)
b) Mean Time To Repair (MTTR)
c) Mean Time Between Failures (MTBF)

Solution:

$$\text{MTTF} = \frac{\text{Total Uptime}}{\text{Number of Failures}} \qquad \text{MTTR} = \frac{\text{Total Repair Time}}{\text{Number of Failures}}$$

$$\text{MTTF} = \frac{920}{8} = 115 \text{ hours} \qquad \text{MTTR} = \frac{80}{8} = 10 \text{ hours}$$

Measurement of Reliability Testing

Q. A software system runs for 1000 hours and experiences 8 failures during this period. The total repair time recorded is 80 hours.

Calculate:
a) Mean Time To Failure (MTTF)
b) Mean Time To Repair (MTTR)
c) Mean Time Between Failures (MTBF)

Solution:

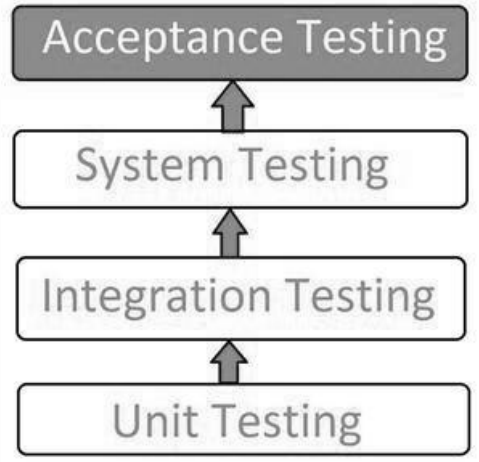
$$\text{MTBF} = \text{MTTF} + \text{MTTR}$$

$$\text{MTBF} = 115 + 10 = 125 \text{ hours}$$

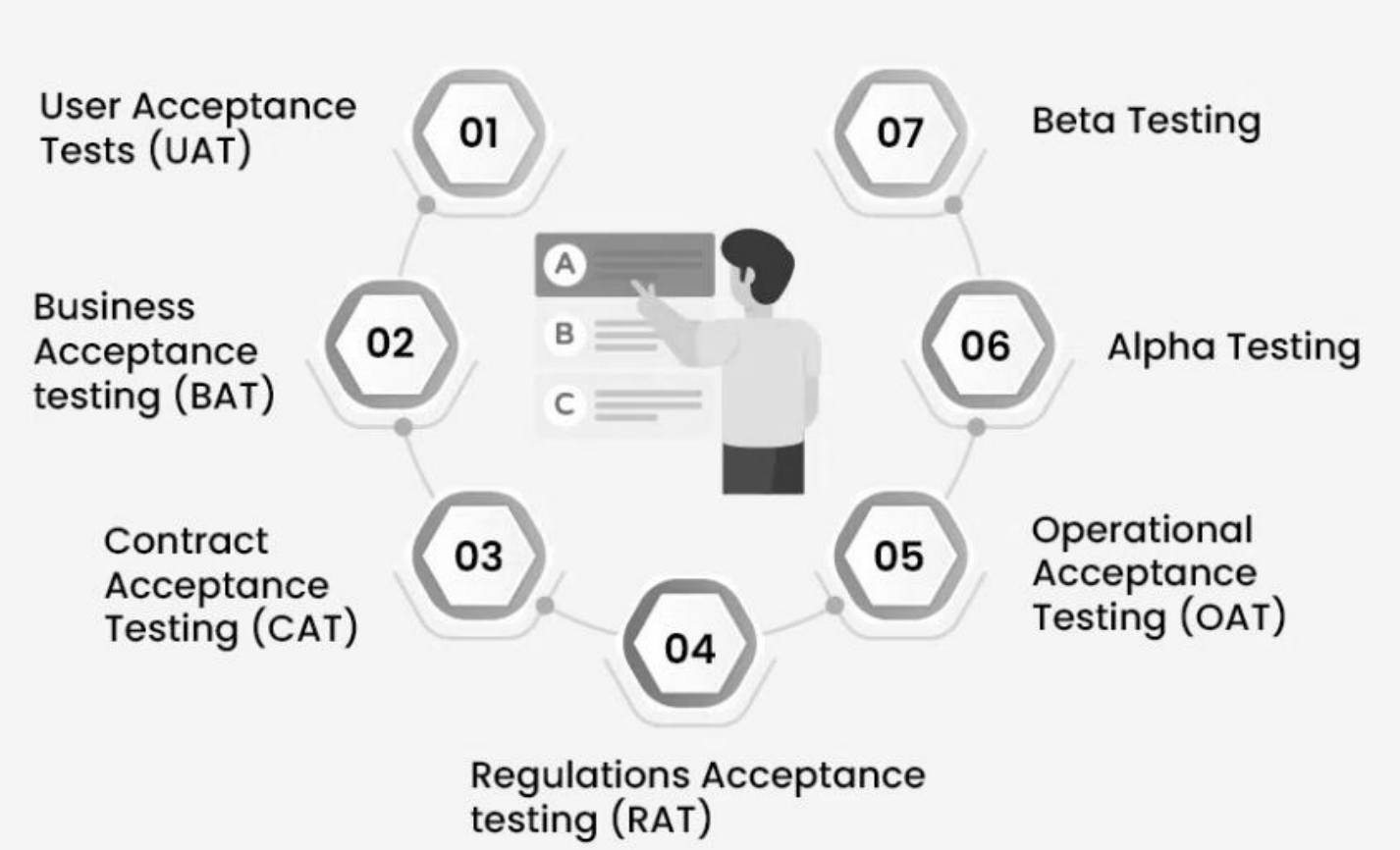
Acceptance testing

- **Acceptance Testing is a formal testing according to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria or not and to enable the users, customers, or other authorized entities to determine whether to accept the system or not.**

- **Acceptance Testing is the last phase of software testing performed after System Testing and before making the system available for actual use.**



Types of Acceptance Testing



Types of Acceptance Testing

User Acceptance Testing (UAT):

- User acceptance testing is used to determine whether the product is working for the user correctly.
- Specific requirements which are quite often used by the customers are primarily picked for testing purposes. This is also termed as End-User Testing.

Business Acceptance Testing (BAT):

- BAT is used to determine whether the product meets the business goals and purposes or not.
- BAT mainly focuses on business profits which are quite challenging due to the changing market conditions and new technologies, so the current implementation may have to being changed which results in extra budgets.

Types of Acceptance Testing

Contract Acceptance Testing (CAT):

- **CAT is a contract that specifies that once the product goes live, within a predetermined period, the acceptance test must be performed, and it should pass all the acceptance use cases.**
- **Here is a contract termed a Service Level Agreement (SLA), which includes the terms where the payment will be made only if the Product services are in-line with all the requirements, which means the contract is fulfilled.**
- **Sometimes, this contract happens before the product goes live.**
- **There should be a well-defined contract in terms of the period of testing, areas of testing, conditions on issues encountered at later stages, payments, etc.**

Types of Acceptance Testing

Regulations Acceptance Testing (RAT):

- **RAT is used to determine whether the product violates the rules and regulations that are defined by the government of the country where it is being released.**
- **This may be unintentional but will impact negatively on the business. Generally, the product or application that is to be released in the market, has to go under RAT, as different countries or regions have different rules and regulations defined by its governing bodies.**
- **If any rules and regulations are violated for any country then that country or the specific region then the product will not be released in that country or region.**
- **If the product is released even though there is a violation then only the vendors of the product will be directly responsible.**

Types of Acceptance Testing

Operational Acceptance Testing (OAT)

- OAT is used to determine the operational readiness of the product and is non-functional testing.
- It mainly includes testing of recovery, compatibility, maintainability, reliability, etc. OAT assures the stability of the product before it is released to production.

Alpha Testing

- Alpha testing is used to determine the product in the development testing environment by a specialized testers team usually called alpha testers.

Beta Testing

- Beta testing is used to assess the product by exposing it to the real end-users, typically called beta testers in their environment.
- Feedback is collected from the users and the defects are fixed. Also, this helps in enhancing the product to give a rich user experience

Use of Acceptance Testing

- To find the defects missed during the functional testing phase.
- How well the product is developed.
- A product is what actually the customers need.
- Feedback help in improving the product performance and user experience.
- Minimize or eliminate the issues arising from the production.

Three major objectives of acceptance testing:

- Confirm that the system meets the agreed upon criteria
- Identify and resolve discrepancies, if there is any
- Determine the readiness of the system for cut-over to live operations

Acceptance Criteria

- The acceptance criteria are defined on the basis of the following attributes:

  - Functional Correctness and Completeness
  - Accuracy
  - Data Integrity
  - Data Conversion
  - Backup and Recovery
  - Competitive Edge
  - Usability
  - Performance
  - Start-up Time
  - Stress
  - Reliability and Availability
  - Maintainability and Serviceability
  - Robustness
  - Timeliness
  - Confidentiality and Availability
  - Compliance
  - Installability and Upgradability
  - Scalability
  - Documentation

Selection of Acceptance Criteria

The acceptance criteria are defined on the basis of the following attributes:

- **The acceptance criteria discussed are too many and very general**
- **The customer needs to select a subset of the quality attributes**
- **The quality attributes are prioritize them to specific situation**
- **IBM used the quality attribute list CUPRIMDS for their products**
- **– Capability, Usability, Performance, Reliability, Installation, Maintenance, Documentation, and Service**
- **Ultimately, the acceptance criteria must be related to the business goals of the customer's organization**

Acceptance Test Plan

1. Introduction
2. Acceptance test category For each category of acceptance criteria (a) Operation environment (b) Test case specification (i) Test case Id# (ii) Test title (iii)Test objective (iv) Test procedure
3. Schedule
4. Human resources

Table 14.1: An outline of an acceptance test plan.

Acceptance Test Execution

- The acceptance test cases are divided into two subgroups
  - The first subgroup consists of basic test cases, and
  - The second consists of test cases that are more complex to execute
- The acceptance tests are executed in two phases
  - In the first phase, the test cases from the basic test group are executed
  - If the test results are satisfactory then the second phase, in which the complex test cases are executed, is taken up.
  - In addition to the basic test cases, a subset of the system-level test cases are executed by the acceptance test engineers to independently confirm the test results
- Acceptance test execution activity includes the following detailed actions:
  - The developers train the customer on the usage of the system
  - The developers and the customer co-ordinate the fixing of any problem discovered during acceptance testing
  - The developers and the customer resolve the issues arising out of any acceptance criteria discrepancy

Acceptance Test Execution

- The acceptance test engineer may create an Acceptance Criteria Change (ACC) document to communicate the deficiency in the acceptance criteria to the supplier
- A representative format of an ACC document is shown in Table 14.2.
- An ACC report is generally given to the supplier's marketing department through the on-site system test engineers

1. ACC Number:	A unique number
2. Acceptance Criteria Affected:	The existing acceptance criteria
3. Problem/Issue Description:	Brief description of the issue
4. Description of Change Required:	Description of the changes needed to be done to the original acceptance criterion
5. Secondary Technical Impacts:	Description of the impact it will have on the system
6. Customer Impacts:	What impact it will have on the end user
7. Change Recommended by:	Name of the acceptance test engineer(s)
8. Change Approved by:	Name of the approver(s) from both the parties

Table 14.2: Acceptance criteria change document information.

Acceptance Test Report

- The acceptance test activities are designed to reach at a conclusion:
  - accept the system as delivered
  - accept the system after the requested modifications have been made
  - do not accept the system
- Usually some useful intermediate decisions are made before making the final decision.
  - A decision is made about the continuation of acceptance testing if the results of the first phase of acceptance testing is not promising
  - If the test results are unsatisfactory, changes be made to the system before acceptance testing can proceed to the next phase
- During the execution of acceptance tests, the acceptance team prepares a test report on a daily basis
- A template of the test report is given in **Table 14.3**
- At the end of the first and the second phases of acceptance testing an acceptance test report is generated which is outlined in **Table 14.4**

Acceptance Test Report

1. Date:	Acceptance report date
2. Test case execution status:	Number of test cases executed today Number of test cases passing Number of test cases failing
3. Defect identifier:	Submitted defect number Brief description of the issue
4. ACC number(s):	Acceptance criteria change document number(s), if any
5. Cumulative test execution status:	Total number of test cases executed Total number of test cases passing Total number of test cases failing Total number of test cases not executed yet

Table 14.3: Structure of the acceptance test status report.

Acceptance Test Report

1. Report identifier
2. Summary
3. Variances
4. Summary of results
5. Evaluation
6. Recommendations
7. Summary of activities
8. Approval

Table 14.4: Structure of the acceptance test summary report.

Acceptance Testing in eXtreme Programming

- In XP framework the user stories are used as acceptance criteria
- The user stories are written by the customer as things that the system needs to do for them
- Several acceptance tests are created to verify the user story has been correctly implemented
- The customer is responsible for verifying the correctness of the acceptance tests and reviewing the test results
- A story is incomplete until it passes its associated acceptance tests
- Ideally, acceptance tests should be automated, either using the unit testing framework, before coding
- The acceptance tests take on the role of regression tests

Test Organization

TEST MANAGEMENT

Test management is concerned with both test resource and test environment management. It is the role of test management to ensure that new or modified service products meet the business requirements for which they have been developed or enhanced. The key elements of test management are

Test organization: - It is the process of setting up and managing a suitable test organizational structure and defining explicit roles. The project framework under which the testing activities will be carried out is reviewed, high-level test phase plans are prepared, and resource schedules are considered. Test organization also involves the determination of configuration standards and defining the test environment.

Test Organization

TEST ORGANIZATION: -

Since testing is viewed as a process, it must have an organization such that a testing group works for better testing and high quality software. The testing group is responsible for the following activities:

- Maintenance and application of test policies „
- Development and application of testing standards „
- Participation in requirement, design, and code reviews „
- Test planning „
- Test execution „
- Test measurement „
- Test monitoring „
- Defect tracking „
- Acquisition of testing tools „
- Test reporting

Test Organization

The staff members of such a testing group are called test specialists or test engineers or simply testers. A tester is not a developer or an analyst. He does not debug the code or repair it. He is responsible for ensuring that testing is effective and quality issues are being addressed. The skills a tester should possess are

Test Organization

STRUCTURE OF TESTING GROUP: -

Testing is an important part of any software project. One or two testers are not sufficient to perform testing, especially if the project is too complex and large. Therefore, many testers are required at various levels. Figure 9.1 shows different types of testers in a hierarchy.

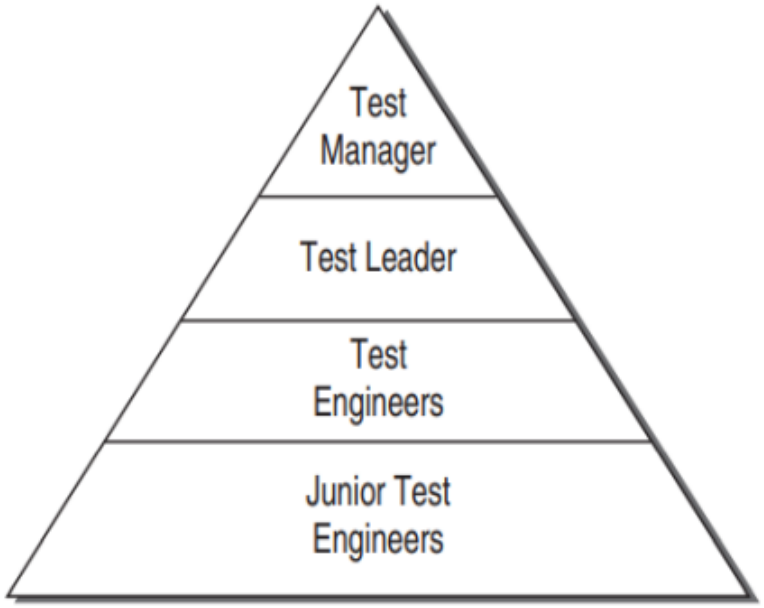


Figure 9.1 Testing Group Hierarchy

Test Organization

TEST PLANNING: -

There is a general human tendency to 'get on with the next thing', especially under pressure. This is true for the testers who are always short of time.

However, if resources are to be utilized intelligently and efficiently during the earlier testing phases and later phases, these are repaid many times over. The time spent on planning the testing activities early is never wasted and usually the total time cycle is significantly shorter.

According to the test process as discussed in STLC, testing also needs planning as is needed in SDLC. Since software projects become uncontrolled if not planned properly, the testing process is also not effective if not planned earlier. Moreover, if testing is not effective in a software project, it also affects the final software product. Therefore, for a quality software, testing activities must be planned as soon as the project planning starts.

A test plan is defined as a document that describes the scope, approach, resources, and schedule of intended testing activities. Test plan is driven with the business goals of the product. In order to meet a set of goals, the test plan identifies the following:

- Test items
- Features to be tested
- Testing tasks
- Tools selection
- Time and effort estimate
- Who will do each task
- Any risks
- Milestones

Quality Assurance Plan

A quality assurance plan is a document, constructed by the project team, meant to ensure the final products are of the utmost quality. A quality assurance plan contains a set of documented activities meant to ensure that customers are satisfied with the goods or services a company provide. The plan provide a layout of the whole project execution processes, roles of each stakeholder, materials, standards and equipment to be used. The plan also provides the steps necessary to maintain and assure quality in a way that it is easily understandable and beneficial to the client, engineer and contractor alike.

Quality Assurance Plan as an essential part of quality control has advantage that the steps are documented and data is kept so that all issues can be traced back and dealt with accordingly, this too is outlined and addressed. With documentation comes communication, **checks and balances** maintained and the dialogue between individuals regarding quality is accurate and informative.

Major Items to be included in the Quality Assurance Plan

When developing a Quality assurance plan, the following should be considered;

7.3.1 Project Quality Organization

- Define who are project stakeholders ; personnel and independent quality testing Institutions,
- Assess qualifications of employees ,
- Assess qualifications of subcontractors and suppliers and independent Laboratories,
- Define what are the responsibilities and interaction between stakeholders,
- Develop Project Organization chart.

7.3.2Quality Communication

- Define types and frequency of meetings (project start-up meeting, routine production meetings, final closure meetings) that will discuss quality issues
- Define type of reports, test results, minutes and/or inspection forms to be submitted to the client.

Major Items to be included in the Quality Assurance Plan

7.3.3 Major phases/Definable Work Tasks

- Define project major work tasks (eg concreting, excavation, pipe work),
- For each Definable Work Tasks prepare a method statement of construction,
- For each Definable Work Tasks develop Inspection Check List,
- For each Definable Work Tasks Develop Risk Assessment Matrix,

7.3.4 Documented Quality standards and Specifications

- Define which quality standards and specification are to be used for each task.

Major Items to be included in the Quality Assurance Plan

List of attachments

The Quality Assurance plan contains other documents and reports, these are developed at the site and should be submitted to the client.

- Major Definable features of Work
- Methods Statements of Constructions
- Inspection checklist
- Inspection Test Plan
- Risk assessment matrix
- Material tests reports
- Reports of nonconformance's

Implementing a Prioritization Plan

- **Identify all tasks:** Begin by creating a master list of all test cases and tasks to be performed.
- Define clear criteria: Establish the factors that will determine priority, such as potential business impact, risk level, and user frequency.
- **Assign priorities:** Using a chosen framework (e.g., MoSCoW method, priority matrix), assign priority levels (e.g., critical, high, medium, low) to each test case.
- **Involve stakeholders:** Collaborate with product owners, developers, and business analysts to ensure consensus on priorities and gather different perspectives on risk and importance.
- Execute high-priority tests first: Follow the established order to run the most critical tests as early as possible in the development cycle.
- **Regularly review and update:** Test prioritization is not a one-time activity. Continuously revisit and adjust priorities based on new information, changing requirements, and ongoing feedback.
- **Use a test management tool:** Utilize a test management platform to help organize, track, and manage the prioritized test cases effectively.

Why should test cases be prioritized?

- As the size of software increases, the test suite also grows bigger and requires more effort to maintain the test suite. To detect bugs in software as early as possible, it is important to prioritize test cases so that important test cases can be executed first.

- **Types of Test Case Prioritization :**

- **General Prioritization:** In this type of prioritization, test cases that will be useful for the subsequent modified versions of the product are prioritized. It does not require any information regarding modifications made to the product.
- **Version-Specific Prioritization:** Test cases can also be prioritized such that they are useful on a specific version of the product. This type of prioritization requires knowledge about changes that have been introduced in the product.

Prioritization Techniques

- **Coverage-based Test Case Prioritization :**

  This type of prioritization is based on code coverage i.e. test cases are prioritized based on their code coverage.

  - **Total Statement Coverage Prioritization -** In this technique, the total number of statements covered by a test case is used as a factor to prioritize test cases. For example, a test case covering 10 statements will be given higher priority than a test case covering 5 statements.

  - **Additional Statement Coverage Prioritization -** This technique involves iteratively selecting a test case with maximum statement coverage, then selecting a test case that covers statements that were left uncovered by the previous test case. This process is repeated till all statements have been covered.

  - **Total Branch Coverage Prioritization -** Using total branch coverage as a factor for ordering test cases, prioritization can be achieved. Here, branch coverage refers to coverage of each possible outcome of a condition.

Prioritization Techniques

Risk-based Prioritization

This technique uses risk analysis to identify potential problem areas which if failed, could lead to bad consequences. Therefore, test cases are prioritized keeping in mind potential problem areas. In risk analysis, the following steps are performed :

- List potential problems.
- Assigning probability of occurrence for each problem.
- Calculating the severity of impact for each problem.

After performing the above steps, a risk analysis table is formed to present results. The table consists of columns like Problem ID, Potential problem identified, Severity of Impact, Risk exposure, etc.

Prioritization Techniques

Prioritization using Relevant Slice

In this type of prioritization, slicing technique is used – when program is modified, all existing regression test cases are executed in order to make sure that program yields same result as before, except where it has been modified. For this purpose, we try to find part of program which has been affected by modification, and then prioritization of test cases is performed for this affected part. There are 3 parts to slicing technique :

Execution slice - The statements executed under test case form execution slice.

Dynamic slice - Statements executed under test case that might impact program output.

Relevant Slice - Statements that are executed under test case and don't have any impact on the program output but may impact output of test case.

Prioritization Techniques

Requirements - based Prioritization

Some requirements are more important than others or are more critical in nature, hence test cases for such requirements should be prioritized first. The following factors can be considered while prioritizing test cases based on requirements :

Customer assigned priority - The customer assigns weight to requirements according to his need or understanding of requirements of product.

Developer perceived implementation complexity - Priority is assigned by developer on basis of efforts or time that would be required to implement that requirement.

Requirement volatility - This factor determines frequency of change of requirement.

Fault proneness of requirements - Priority is assigned based on how error-prone requirement has been in previous versions of software.

Metric for measuring Effectiveness of Prioritized Test Suite

- **For measuring how effective prioritized test suite is, we can use metric called APFD (Average Percentage of Faults Detected).**

$$APFD = 1 - ((TF_1 + TF_2 + ... + TF_m) / nm) + 1 / 2n$$

where,
TF_i = position of first Test case in Test suite T that exposes Fault i
m = total number of Faults exposed under T
n = total number of Test cases in T

Note: AFPD value can range from 0 to 100. The higher APFD value, faster faults detection rate. So simply put, APFD indicates of how quickly test suite can identify faults or bugs in software. If test suite can detect faults quickly, then it is considered to be more effective and reliable.

Metric for measuring Effectiveness of Prioritized Test Suite

- **For measuring how effective prioritized test suite is, we can use metric called APFD (Average Percentage of Faults Detected).**

$$APFD = 1 - ((TF_1 + TF_2 + ... + TF_m) / nm) + 1 / 2n$$

where,
TF_i = position of first Test case in Test suite T that exposes Fault i
m = total number of Faults exposed under T
n = total number of Test cases in T

Metric for measuring Effectiveness of Prioritized Test Suite

Question: A software system has n = 10 test cases and m = 4 faults.
The positions of the first test case that detects each fault are:
$TF_1 = 1$
$TF_2 = 3$
$TF_3 = 4$
$TF_4 = 8$

Metric for measuring Effectiveness of Prioritized Test Suite

Solution:

Since we know that:

$$APFD = 1 - \frac{\sum TF_i}{nm} + \frac{1}{2n}$$

∑TFi=1+3+4+8=16

$$nm = 10 \times 4 = 40$$

$$\frac{\sum TF_i}{nm} = \frac{16}{40} = 0.4$$

$$\frac{1}{2n} = \frac{1}{20} = 0.05$$

$$APFD = 1 - 0.4 + 0.05$$

$$APFD = 0.65$$

Metric for measuring Effectiveness of Prioritized Test Suite

Solution:

Since we know that:

$$APFD = 1 - \frac{\sum TF_i}{nm} + \frac{1}{2n}$$

APFD (Average Percentage of Faults Detected) measures how quickly faults are detected in a prioritized test suite.

Range of APFD:

$$0 \leq APFD \leq 1$$

Higher APFD means:

Faults are detected earlier → better prioritization.

TEST EXIT CRITERIA

Test Exit Criteria are the conditions that must be satisfied before stopping the testing process. They ensure that the software has achieved the required quality level.

Purpose of the Test Exit Criteria:

- **Ensure adequate test coverage**
- **Confirm required quality level**
- **Prevent premature release**
- **Control project cost and schedule**

Test Exit Criteria, Cost and economy Aspects

Common Exit Criteria:

1.Test Case Execution

1. All planned test cases executed
2. Example: 95% test cases passed

2.Defect Status

1. No open Critical or High severity defects
2. Remaining defects are low severity and acceptable

3.Requirements Coverage

1. 100% requirements covered by test cases

4.Code Coverage

1. Minimum predefined coverage achieved (e.g., 80–85%)

5.Performance Criteria

1. Response time within acceptable limit
2. Load and stress testing completed

6. Approval

1. QA sign-off
2. Client or stakeholder approval

Test Exit Criteria, Cost and economy Aspects

- The cost aspects of software testing refer to the different types of expenses involved in ensuring software quality.
- These costs are generally classified into prevention cost, appraisal cost, and failure cost.
- Prevention costs include expenses related to training, quality planning, and process improvement activities aimed at avoiding defects.
- Appraisal costs involve test design, execution, reviews, and automation efforts used to detect defects.
- Failure costs occur when defects are found, and they are divided into internal failure costs, such as debugging and rework before release, and external failure costs, such as maintenance, customer complaints, and loss of reputation after release.
- It is widely accepted that the cost of fixing defects increases significantly as the project progresses, with defects found after deployment being far more expensive to correct than those identified during the requirement or design phase.

Test Exit Criteria, Cost and economy Aspects

- The economy aspects of testing focus on achieving an optimal balance between the cost of testing and the benefits gained from defect detection.
- Testing should be economically justified, meaning that the cost of testing should be less than the potential cost of failures in production.
- It is neither practical nor economical to aim for completely defect-free software. Instead, organizations adopt risk-based testing approaches, concentrating more effort on high-risk and critical components.
- Automation may involve higher initial investment but provides long-term savings for repetitive testing. Ultimately, testing is considered economical when it reduces overall project risk and ensures acceptable quality at a reasonable cost.

Test Management Contd…


thank
you