CH. DEVI LAL STATE INSTITUTE OF ENGINEERING AND TECHNOLOGY

LAB MANNUL

OF

SOFTWARE TESTING AND QUALITY ASSURANCE

PROGRAM -1

Program:- Consider a problem for the determination of the nature of the roots of a quadratic equation where the inputs are 3 variables (a, b, c) and their values may be from the interval [0, 100].

Boundary value analysis is a software testing technique in which tests are designed to include representatives of boundary values in a range. A boundary value analysis has a total of 4*n+1 distinct test cases, where n is the number of variables in a problem.

Here we have to consider all the three variables and design all the distinct possible test cases. We will have a total of 13 test cases as n = 3.

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

How do we design the test cases ?

For each variable we consider below 5 cases:

- $a_{min} = 0$
- $a_{min+1} = 1$
- $a_{nominal} = 50$
- $a_{max-1} = 99$
- $a_{max} = 100$

When we are considering these 5 cases for a variable, rest of the variables have the nominal values, like in the above case where the value of ‘a’ is varying from 0 to 100, the value of ‘b’ and ‘c’ will be taken as the nominal or average value. Similarly, when the values of variable ‘b’ are changing from 0 to 100, the values of ‘a’ and ‘c’ will be nominal or average i.e 50

The possible test cases for the nature of roots of a Quadratic Equation in a Boundary Value Analysis can be:

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

Below is the program that verifies the test cases considered in the table shown above.

```
# Python3 program to check the nature of the roots

# BVA for nature of roots of a quadratic equation

def nature_of_roots(a, b, c):

    # If a = 0, D/2a will yield exception

    # Hence it is not a valid Quadratic Equation

        if (a == 0):

                print("Not a Quadratic Equation");

                return;

        D = b * b - 4 * a * c;
```

```
        # If D > 0, it will be Real Roots
        if (D > 0):
                print("Real Roots");
        # If D == 0, it will be Equal Roots
        elif(D == 0):
                print("Equal Roots");
        # If D < 0, it will be Imaginary Roots
        else:
                print("Imaginary Roots");
        # Function to check for all testcases
def checkForAllTestCase():
        print("Testcase\ta\tb\tc\tActual Output");
        print();
        a = b = c = 0;
        testcase = 1;
        while (testcase <= 13):
                if (testcase == 1):
                        a = 0;
                        b = 50;
                        c = 50;
                elif(testcase == 2):
                        a = 1;
                        b = 50;
                        c = 50;
```

```
elif(testcase == 3):
        a = 50;
        b = 50;
        c = 50;
elif(testcase == 4):
        a = 99;
        b = 50;
        c = 50;
elif(testcase == 5):
        a = 100;
        b = 50;
        c = 50;
elif(testcase == 6):
        a = 50;
        b = 0;
        c = 50;
elif(testcase == 7):
        a = 50;
        b = 1;
        c = 50;
elif(testcase == 8):
        a = 50;
        b = 99;
        c = 50;
```

```
elif(testcase == 9):
        a = 50;
        b = 100;
        c = 50;
elif(testcase == 10):
        a = 50;
        b = 50;
        c = 0;
elif(testcase == 11):
        a = 50;
        b = 50;
        c = 1;
elif(testcase == 12):
        a = 50;
        b = 50;
        c = 99;
elif(testcase == 13):
        a = 50;
        b = 50;
        c = 100;
print("\t" , testcase , "\t" , a , "\t" , b , "\t" , c , "\t", end="");
nature_of_roots(a, b, c);
print();
testcase += 1;
```

```
# Driver Code

if __name___== '__main__':

        checkForAllTestCase();
```

Output:

```
Testcase     a     b     c     Actual Output
 1     0     50     50     Not a Quadratic Equation
 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
```

PROGRAM -2

Program:- Write a program to calculate cyclomatic complexity.

Cyclomatic Complexity may be defined as-

- It is a software metric that measures the logical complexity of the program code.
- It counts the number of decisions in the given program code.
- It measures the number of linearly independent paths through the program code.

Calculating Cyclomatic Complexity-

There are 3 commonly used methods for calculating the cyclomatic complexity-

Method-01:

Cyclomatic Complexity = Total number of closed regions in the control flow graph + 1

Method-02:

Cyclomatic Complexity = E – N + 2

Here-, E = Total number of edges in the control flow graph

And N = Total number of nodes in the control flow graph

Method-03:

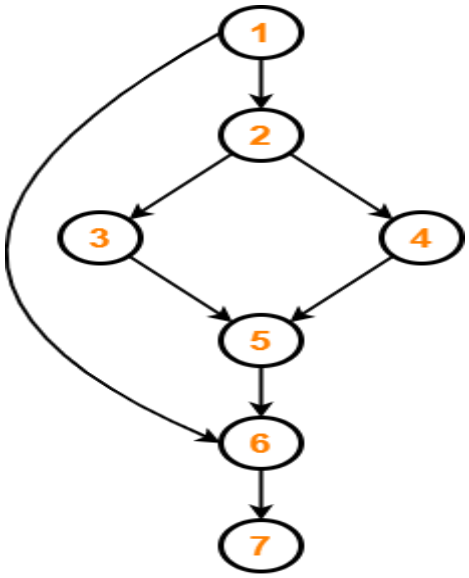
Cyclomatic Complexity = P + 1

Here, P = Total number of predicate nodes contained in the control flow graph

Calculate cyclomatic complexity for the given code-

```
IF A = 354
THEN IF B > C
THEN A = B
ELSE A = C
END IF
END IF
```

We draw the following control flow graph for the given code-



Control Flow Graph

Method-01:

Cyclomatic Complexity

= Total number of closed regions in the control flow graph + 1

= 2 + 1

= 3

Method-02:

Cyclomatic Complexity

= E – N + 2

= 8 – 7 + 2

= 3

Method-03:

Cyclomatic Complexity

= P + 1

= 2 + 1

= 3

PROGRAM -3

Program:- Write a program to perform binary search and generate test cases using equivalence class testing and decision table based testing.

Equivalence Partitioning :

• Divide your input conditions into groups (classes) – Input in the same class should behave similarly in the program

• Be sure to test a mid-range value from each class

Binary Search Equivalence Partitions

• Pre-conditions satisfied, key element in array

• Pre-conditions satisfied, key element not in array

• Pre-conditions unsatisfied, key element in array

• Pre-conditions unsatisfied, key element not in array

• Input array has a single value

• Input array has an even number of values

• Input array has an odd number of values

Program

```
int search ( int key, int [] elemArray)  {
 int bottom = 0;
int top = elemArray.length - 1;
 int mid;
 int result = -1;
 while ( bottom <= top ) {
mid = (top + bottom) / 2;
 if (elemArray [mid] == key) {
result = mid;
 return result; }
```

```
// if part
 Else {
 if (elemArray [mid] < key)
 bottom = mid + 1;
else
top = mid - 1; }
 } //while loop
 return result;
 } // search
```

Binary Search Test Cases:-

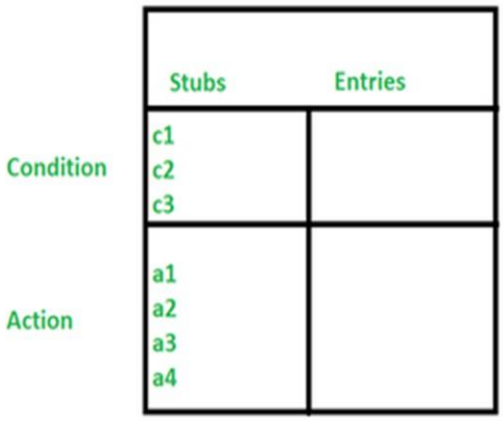
Input Array	Key	Output
17	17	True,1
17	0	False,??
17,21,23,29	17	True,1
9,16,18,30,31,41,45	45	True,7
17,18,21,23,29,38,41	23	True,4
17,18,21,23,29,33,38	21	True,3
12,18,21,23,32	23	True,4
21,23,29,33,38	25	False,??

Decision Table Based Testing

A Decision Table is a tabular representation of inputs versus rules/cases/test conditions. It is a very effective tool used for both complex software testing and requirements management. Decision table helps to check all possible combinations of conditions for testing and testers can also identify missed conditions easily. The conditions are indicated as True(T) and False(F) values.

Parts of Decision Tables :

In software testing, the decision table has 4 parts which are divided into portions and are given below :



1. **Condition Stubs :** The conditions are listed in this first upper left part of the decision table that is used to determine a particular action or set of actions.
2. **Action Stubs :** All the possible actions are given in the first lower left portion (i.e, below condition stub) of the decision table.
3. **Condition Entries :** In the condition entry, the values are inputted in the upper right portion of the decision table. In the condition entries part of the table, there are multiple rows and columns which are known as Rule.
4. **Action Entries :** In the action entry, every entry has some associated action or set of actions in the lower right portion of the decision table and these values are called outputs.

Decision Table in test designing:

CONDITIONS	STEP 1	STEP 2	STEP 3	STEP 4
Condition 1	Y	Y	N	N
Condition 2	Y	N	Y	N
Condition 3	Y	N	N	Y
Condition 4	N	Y	Y	N

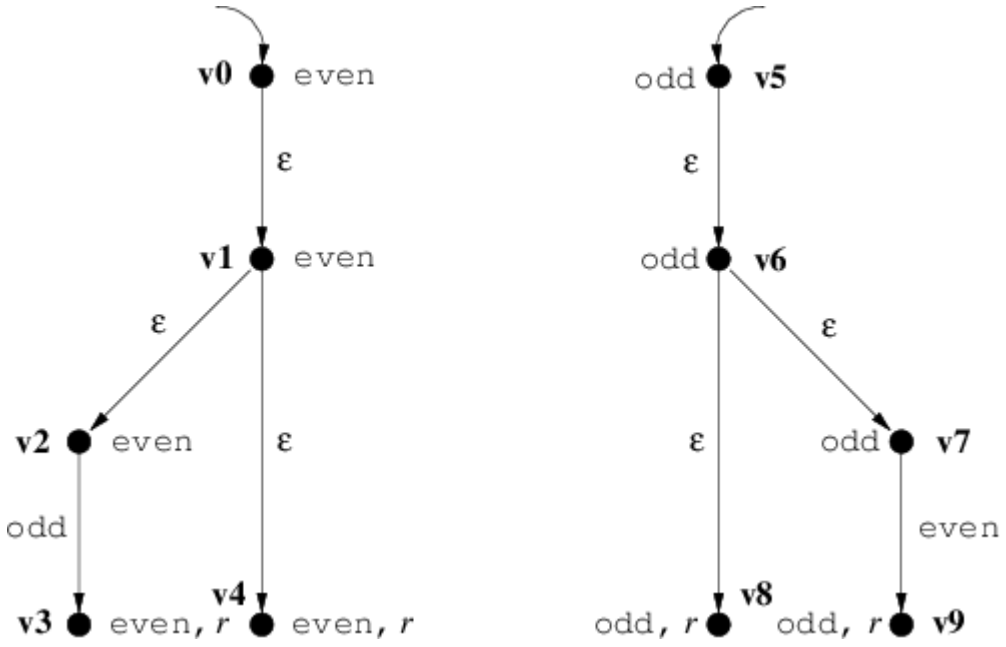
PROGRAM -4

Program:- Write a program to determine whether a number is even or odd. Draw the program graph and DD Path graph. Find the independent paths.

Program

```
#include <stdio.h>
void main()
{
 int num1, rem1;
 printf("Input an integer : ");
 scanf("%d", &num1);
rem1 = num1 % 2;
 if (rem1 == 0)
    printf("%d is an even integer\n", num1);
 else
    printf("%d is an odd integer\n", num1);
  }
```

Program Graph



Find Cyclomatic Complexity

Cyclomatic Complexity = E – N + 2

Here-

- E = Total number of edges in the control flow graph
- N = Total number of nodes in the control flow graph
    - Cyclomatic Complexity = E – N + 2

=5-5+2

=2

Independent Path= Cyclomatic Complexity

PROGRAM -5

Program:- Write a program for classification of triangle. Consider all variables and generate possible program slices. Design at least for one test case from every slice.

Triangle Problem accepts three integers – a, b, c as three sides of the triangle .We also define a range for the sides of the triangle as [l, r] where l>0. It returns the type of triangle (Scalene, Isosceles, Equilateral, Not a Triangle) formed by a, b, c.

For a, b, c to form a triangle the following conditions should be satisfied –

a < b+c

b < a+c

c < a+b

If any of these conditions is violated output is Not a Triangle.

- For Equilateral Triangle all the sides are equal.
- For Isosceles Triangle exactly one pair of sides is equal.
- For Scalene Triangle all the sides are different.
  - The table shows the Test Cases Design for the Triangle Problem. The range value [l, r] is taken as [1, 100] and nominal value is taken as 50.
  - The total test cases is,
  - 4n+1 = 4*3+1 = 13

Test Case ID	a	b	c	Expected Output
T1	1	50	50	Isosceles
T2	2	50	50	Isosceles
T3	99	50	50	Isosceles
T4	100	50	50	Not a Triangle
T5	50	50	50	Equilateral

Test Case ID	a	b	c	Expected Output
T6	50	1	50	Isosceles
T7	50	2	50	Isosceles
T8	50	99	50	Isosceles
T9	50	100	50	Not a Triangle
T10	50	50	1	Isosceles
T11	50	50	2	Isosceles
T12	50	50	99	Isosceles
T13	50	50	100	Not a Triangle

PROGRAM -6

Program:- Consider the problem for statement of a University Student Registration System. Prepare the software requirement checklist with the details of faults in the given SRS.

Definition: A Checklist is a catalog of items/tasks that are recorded for tracking. This list could be either ordered in a sequence or could be haphazard.

As is the general practice we will talk about the "Why" and "How" aspects.

- **Why do we need Checklists?** : For tracking and assessing completion (or non-completion). To make a note of tasks, so that nothing is overlooked.
- **How do we create Checklists? :** Well, this could not be simpler. Simply, write everything down point by point.

SRS Domain Description :-

- As Head of the component system engineering team for a software company, that are tasked with modeling a reusable component system for a Student Registration System (SRS) domain that should cover requirements capture, robustness analysis, and design.
- The company would like the component system to be highly effective to be reusable and customizable for developing a new student registration system that would be able to deploy on any operation system platform.
- The newly customized system will allow students to register for courses and view their semester results from any type of computer device that attaches to any type of campus networks.
- Lecturers will be able to access the system to sign in to access to the courses that have been assigned to them as well as to record grades.
- The component system should allow all courses offered by any educational institutions to be maintained up-to-date.
- The registrar's office will maintain course information and grades through dedicated user interfaces.
- At the beginning of each semester, students may request/access a list of course offerings for the semester. Information about each course, such as lecturers, departments, and any prerequisites, will be included to help students make informed decisions.

- Lecturers must be able to see which students who had signed up for their course offerings. In addition, the lecturers will be able to record and view the grades of courses taken by the students in their classes.
- By using the various modeling constructs of UML/OOSE for component modeling analysiz these tasks as follows:

SRS Use Case Diagram

Model a complete use case diagram for the student reiteration system by including, the use cases and requirements described in the system.

- The generalization and include/exclude relationships must be considered in your diagram based on SRS domain.
- Use case description for each use case must be converted to the flow of events "paths" which shows the actor actions and system responses and set the priority of each use case.

SRS Class Diagram

- Model a domain class diagram to show the structure of student registration system.
- Show the explicitly the class attributes and operations and also the relationships between the classes, multiplicity specifications, and other model elements that you find appropriate.

Use Case Components

- Analyze the problem domain of the SRS, including investigating and analyzing the most common cases of the specific student registration system from other sources,
- Model, a domain component to show the structure of the student registration system that should be traceable to between all use case components.
- Model each use case with identifying the common functionally and variable functionality into a use case component system model that is to an analysis model component system model.

Robustness Analysis

- Continue with the analysis phase from (Part A*) for the purpose of documenting the analysis component system model using relevant analysis techniques, considering also the development and communication platforms.
- Include traceability between the use case component system model and the analysis component system model, which should be detailed to the level of component traceability between the two models.

- Variable features specific to the various cases of the SRS should be taken into account in both the use case component system and analysis component system models for modeling variants and variability points.
- Illustrate the modeling of developing two separate student registration application systems of different educational institutions by reuse suitable components (with or without variability points) of the analysis component system model through façades and using the UML/OOSE import statement. Demonstrate appropriate customization/adaptation of components and variable features into these application systems.

Design Phase

- Continue with the design phase from (Part B) for the purpose of documenting a design component system model using relevant design techniques, considering also the development and communication platforms.
- Include also traceability between the analysis component system model and the design component system model, which should be detailed to the level of components
- traceability between the two models.

PROGRAM -7

Experiment:- Write a program to generate, minimize and prioritize test cases using any programming language/Matlab Tool/ Software Testing tool.

The test cases are an extremely important part of any “Software Testing Process” The following are the programs to generate test cases.

Generating Random Numbers

```
# A Python3 Program to generate test cases

# for random number

import random

# the number of runs

# for the test data generated

requiredNumbers = 5;

# minimum range of random numbers

lowerBound = 0;

# maximum range of random numbers

upperBound = 1000;

# Driver Code

if __name___== '__main__':

        for i in range(requiredNumbers):

                a = random.randrange(0, upperBound -

                                        lowerBound) + lowerBound;

                    print(a);
```

Test-case reduction

- **Reduce execution time.** This is important when running a huge regression suite for a large and complex program. Testing Chrome with a single SELECT tag is much more efficient than testing it with a multi-gigabyte file. And it's especially helpful in slow execution environments, such as Android emulators.
- **Improve performance of mutation-based fuzzers.** When fuzzers use existing tests to generate new test cases, they have a higher chance of exploring interesting paths when the test case doesn't contain a lot of irrelevant garbage. That's why AFL and libFuzzer like to fuzz only the smallest and fastest-running inputs for each coverage element.
- **Solve symbolic execution constraints more easily.** Generated constraints are easier to solve if there is less uninteresting execution involved. More details can be found in this paper by Zhang et al.
- **Avoid flaky tests.** Tests that pass sometimes, and fail other times, without changing the code under test, are called "flaky tests," and they are one of the most critical problems in testing software at Google scale. Test-case reduction is a core part of a recently proposed algorithm to automatically fix some flaky tests.
- **Deduplicate fuzzing bugs.** Automated "fuzzer taming"—finding the set of actual bugs in a large pile of mostly duplicate test cases produced by a fuzzer—is more effective when test cases are reduced. Test cases for the same bug are more similar if irrelevant parts are removed.
- **Improve fault localization tool performance.** A core problem in fault localization is that failing test cases execute a lot of non-faulty code. Figuring out where the bad code is hiding is easier when the failed tests run less non-buggy code. Reducing test cases reduces the amount of non-faulty code executed, and makes life easier for fault-localization algorithms.
- **Help with future-proof testing.** In some cases, reduced test cases may be more effective at finding bugs in future software versions than the original tests; reduced tests may be less overfitted to the current code.
- **Create new tests from existing ones**.
- **Adapt software to environments with fewer resources.** Reducers can also take programs as input, and this turns out to be the heart of a recently proposed method for automatically adapting programs to environments with fewer available resources.
- **Provide an alternative way to fuzz.** Finally, as John Regehr has pointed out, "reducers are fuzzers." Potentially, test-case reduction can be used to help discover previously unknown bugs in a software system, though as yet it's not clear exactly how effective this new approach to fuzzing really is.

Prioritization Techniques :

1. Coverage – based Test Case Prioritization :

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

- **Total Statement Coverage Prioritization –**
  In this technique, total number of statements covered by test case is used as factor to prioritize test cases. For example, test case covering 10 statements will be given higher priority than test case covering 5 statements.
- **Additional Statement Coverage Prioritization –**
  This technique involves iteratively selecting test case with maximum statement coverage, then selecting test case which covers statements that were left uncovered by previous test case. This process is repeated till all statements have been covered.
- **Total Branch Coverage Prioritization –**
  Using total branch coverage as factor for ordering test cases, prioritization can be achieved. Here, branch coverage refers to coverage of each possible outcome of condition.
- **Additional Branch Coverage Prioritization –**
  Similar to additional statement coverage technique, it first selects text case with maximum branch coverage and then iteratively selects test case which covers branch outcomes that were left uncovered by previous test case.
- **Total Fault-Exposing-Potential Prioritization –**
  Fault-exposing-potential (FEP) refers to ability of test case to expose fault. Statement and Branch Coverage Techniques do not take into account fact that some bugs can be more easily detected than others and also that some test cases have more potential to detect bugs than others. FEP depends on :
  1. Whether test cases cover faulty statements or not.
  2. Probability that faulty statement will cause test case to fail.

2. 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, following steps are performed :

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

After performing above steps, 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.

3. 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.

4. 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.

PROGRAM -8

Experiment:- Write the outline of test plan document as per IEEE Std. 829-1998.

What is a test plan?

"A test plan is a document that outlines the strategy for ensuring that a product or system is built in accordance with its specifications and requirements."

Making a test plan offers multiple benefits:

- It is a manual for the testing procedure. It guides your testing strategy and outlines the testing procedures to follow.
- It contains information about the testing scope, which prohibits the team from testing functionality that is "out of scope."
- It assists in determining the amount of time and effort required to test the product.
- It clearly outlines each team member's tasks and responsibilities, ensuring that everyone on the testing team understands what is expected of them.
- It establishes a timetable for testing activities. As a result, you'll have a baseline plan to control and track the testing progress of your team.
- It lays out the resources and equipment requirements that are required to complete the testing process.
- It can be shared with your client to provide them with information about your testing method and to build their trust.

How to create/write a good test plan?

You're certain that a test strategy is a key to a successful testing procedure at this point. Now you're probably wondering, "How can I write a good test plan?" A test plan software can be used to generate and compose an effective test plan. Also, by following the methods below, we may design an effective software test plan:

1. Analyze the Product

To acquire a better understanding of the product, its features, and functionalities, the first step in establishing a test plan is to examine it. Investigate the client's business needs and what he

or she expects from the finished product. To build the ability to test the product from the user's perspective, and understand the users and use cases.

2. Develop Test Strategy

After you've completed your product analysis, you'll be able to create a test plan for various test levels. Several testing methodologies might be included in your test plan. You choose which testing approaches to utilize based on the use cases and business needs.

If you're constructing a website with thousands of users, for example, you'll include 'Load Testing' in your test strategy. Similarly, if you're working on an e-commerce site that involves online financial transactions, you'll place a premium on security and penetration testing.

3. Define Scope

A solid test plan spells forth the scope of the test and its limitations. You can utilize the requirements specifications document to determine what is included and what is omitted from the scope. Make a list of 'to be tested' and 'not to be tested' features. Your test plan will become more specific and useful as a result of this. You may also need to specify a list of deliverables as a testing process output.

The term 'scope' refers to both functionalities and testing procedures. If any testing technique, such as security testing, is out of scope for your product, you may need to specify it clearly. Similarly, while running load testing on an application, you must define the maximum number of users.

4. Develop a Schedule

You can create a testing schedule if you have a good understanding of the testing strategy and scope. Divide the work into testing activities and calculate the amount of time it will take. You can also estimate how much time each task will take. You may now add a test schedule to your testing plan, which will help you keep track of the testing process' progress.

5. Define Roles and Responsibilities

The tasks and responsibilities of the testing team and team manager are clearly defined in a good test plan. The 'Roles and Responsibilities' section, as well as the 'schedule' portion, explain to everyone what to do and when to do it.

6. Anticipate Risks

Without expected risks, mitigation measures, and risk actions, your test strategy is incomplete. In software testing, there are numerous types of risks, including scheduling, budget, experience, and knowledge. You must outline the risks associated with your product, as well as risk responses and mitigation measures, in order to reduce their severity.

IEEE 829 Standard for Test Plan

IEEE is an international organization that creates globally accepted standards and template documents. The IEEE 829 standard for system and software documentation has been established. It defines the format of a set of documents that must be submitted at each level of software and system testing.

IEEE has specified eight stages in the documentation process, producing a separate document for each stage

According to IEEE 829 test plan standard, the following sections go into creating a testing plan:

1. Test plan identifier

'Test Plan Identifier,' as the name implies, uniquely identifies the test plan. It contains information on the project and, in certain cases, version information. Companies may employ a test plan identifier convention in some instances. The type of test plan is also included in the test plan identification. The following are examples of test plans:

A single high-level test plan for a project or product that incorporates all other test plans. Testing Level Specific Test Plans: Each level of testing, i.e., unit level, integration level, system level, and acceptance level, can have its own test plan.

Specific Test Plans for Specific Testing Forms: Plans for major types of testing, such as Performance Testing Plans.

2. Introduction

The testing plan is detailed in the introduction. It establishes the test plan's goal, scope, goals, and objectives. It also has budget and resource limits. It will also detail any test plan constraints and limitations.

3. Test items

The artifacts that will be tested are listed as test items. It could be one or more project/product modules, as well as their versions.

4. Features to be tested

All of the features and functionalities that will be tested are detailed in this section. It must also provide references to the requirements specifications documents, which describe the features to be tested in-depth.

5. Features not to be tested

This section lists the features and functions that are not included in the testing. It must include justifications for why certain features will not be tested.

6. Approach

The approach to testing will be defined in this section. It explains how the testing will be carried out. It includes details on the test data sources, inputs and outputs, testing procedures, and priorities. The technique will specify the guidelines for requirements analysis, scenario development, acceptance criteria development, and test case construction and execution.

7. Item pass/fail criteria

This section explains how to evaluate the test results using success criteria. It specifies the success criteria for each functionality to be tested in great detail.

8. Suspension criteria and resumption requirements

It shall detail any criteria that may lead to the suspension of testing activities, as well as the conditions to resume testing.

9. Test deliverables

The documents that the testing team will give at the end of the testing process are known as test deliverables. Test cases, sample data, test reports, and issue logs are all examples of this.

10. Testing tasks

Testing tasks are defined in this section. It will also detail any job dependencies, as well as the resources necessary and task completion times. Creating test scenarios, test cases, test scripts, executing test cases, reporting defects, and creating an issue log are all examples of testing tasks.

11. Environmental needs

The prerequisites for the test environment are described in this section. It might be hardware, software, or any other type of testing environment. What test equipment is already in place and what has to be procured should be identified in the plan.

12. Responsibilities

In this section of the test plan, roles and responsibilities are assigned to the testing team.

13. Staffing and training needs

This section outlines the staff training requirements for successfully completing the planned testing activities.

14. Schedule

Testing activities are given dates, which are then used to generate the timetable. This timeline must coincide with the development schedule in order to provide a realistic test strategy.

15. Risks and contingencies

It's critical to understand the dangers, their likelihood, and their consequences. The test strategy must also include ways of mitigating the risks that have been identified. The test strategy should also incorporate contingencies.

16. Approvals

This section provides the stakeholders' signatures of approval.