LAB MANUAL OF

SOFTWARE TESTING AND QUALITY ASSURANCE

ETCS -453

Maharaja Agrasen Institute of Technology, PSP area,
Sector – 22, Rohini, New Delhi – 110085
(Affiliated to Guru Gobind Singh Indraprastha University, New Delhi)
INDEX OF THE CONTENTS

1. Introduction to the lab manual
2. Lab requirements (details of H/W & S/W to be used)
3. List of experiments
4. List of Advance programs
5. Projects to be allotted
6. Format of lab record to be prepared by the students.
7. Marking scheme for the practical exam
8. Details of the each section of the lab along with the examples, exercises & expected viva questions.
1. Introduction to Software Testing and Quality Assurance Lab

The purpose of the laboratory is to evaluate and develop methods of testing software efficiently that aim on discovering security relevant software flaws along with considering core components of quality before the final product is deployed.

The prime goal is to make the students aware about the existing methods of software testing and considering software quality. Course describes a wide range of techniques including mutation testing, slicing, test case coverage determination and Software quality factor, core components of quality, Iso standards and six sigma concept etc. Many of the software testing techniques available are very expensive and time consuming. Therefore, the aim of the lab is to understand, which existing testing techniques are most effective for vulnerability detection, in order to provide software engineers guidelines for the selection of testing methods using software quality methods.
# 2. LAB REQUIREMENTS

<table>
<thead>
<tr>
<th>Hardware Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>HCL Intel CPU (P-IV 3.0GHz, HT)/512 MB RAM/ 80GB HDD/ Intel 865GLC M.B./On board Sound &amp; 3D Graphics card/ Lan Card Key Board/ Mouse/ CDRW Drive/ 15” Color Monitor/ UPS</td>
</tr>
<tr>
<td>24 Nos.</td>
</tr>
<tr>
<td>Dot Matrix Printer</td>
</tr>
<tr>
<td>1 No.</td>
</tr>
<tr>
<td>LaserJet Printer</td>
</tr>
<tr>
<td>1 No.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Software Detail</th>
</tr>
</thead>
<tbody>
<tr>
<td>RedHat Linux &amp; C/C++, Selenium tool</td>
</tr>
</tbody>
</table>

---

**MAIT/CSE**

4 | Page
SOFTWARE TESTING AND QUALITY ASSURANCE LAB

Paper Code: ETCS-453

List of Experiments:

(As prescribed by G.G.S.I.P.U)

Tool Required: Smartbear QA Complete

1. To determine the nature of roots of a quadratic equations, its input is triple of +ve integers (say x,y,z) and values may be from interval[1,100] the program output may have one of the following:-
   [Not a Quadratic equations, Real roots, Imaginary roots, Equal roots] Perform BVA.

2. To determine the type of triangle. Its input is triple of +ve integers (say x,y,z) and the values may be from interval[1,100].The program output may be one of the following [Scalene, Isosceles, Equilateral, Not a Triangle].Perform BVA

3. Perform robust case testing on Problem No. 1.

4. Perform robust case testing on Problem No. 2.

5. Create a test plan document for any application (e.g. Library Management System)

6. Experiment: Study of Any Testing Tool (Win Runner)

7. Experiment: Study of Any Test Management Tool ( QA Complete)

8. Experiment: Automate the Test cases using Test Automation tool(using QA Complete)

9. Experiment: Learn how to raise and report Bugs using Bug tracking tool (Bugzilla,Jira using QA Complete)

10. Experiment: Study of any open source testing tool (Web Performance Analyzer/O STA)

NOTE: At least 8 Experiments out of the list must be done in the semester.
3. LIST OF EXPERIMENTS

Paper Code: ETCS-453
Paper: Software Testing and Quality Assurance Lab

List of Experiments:

Problem Statement 01
Consider an automated banking application. The user can dial the bank from a personal computer, provide a six-digit password, and follow with a series of keyword commands that activate the banking function. The software for the application accepts data in the following form:

<table>
<thead>
<tr>
<th>Area Code</th>
<th>Blank or three-digit number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>Three-digit number, not beginning with 0 or 1</td>
</tr>
<tr>
<td>Suffix</td>
<td>Four-digit number</td>
</tr>
<tr>
<td>Password</td>
<td>Six-character alphanumeric</td>
</tr>
<tr>
<td>Commands</td>
<td>&quot;Check status&quot;, &quot;Deposit&quot;, &quot;Withdrawal&quot;</td>
</tr>
</tbody>
</table>

Design adhoc test cases to test the system

Problem Statement 02
For the above problem design the test cases to test the system using following Black Box testing technique:

- BVA, Worst BVA, Robust BVA, Robust Worst BVA
- Equivalence class testing (Input/Output domain)

Problem Statement 03

- Experiment: Study of any open source testing tool (Web Performance Analyzer/O STA/selenium)

Problem Statement 04
Consider an application that is required to validate a number according to the following simple rules:

1. A number can start with an optional sign.
2. The optional sign can be followed by any number of digits.
3. The digits can be optionally followed by a decimal point, represented by a period.
4. If there is a decimal point, then there should be two digits after the decimal.
5. Any number-whether or not it has a decimal point, should be terminated a blank.

Generate test cases to test valid and invalid numbers.

(HINT) Use Decision table and cause-effect graph to generate test cases.
Problem Statement 05
Generate test cases using Black box testing technique to Calculate Standard Deduction on Taxable Income
The standard deduction is higher for tax payers who are 65 or older or blind.
Use the method given below to calculate tax.
1. The first factor that determines the standard deduction is the filing status. The basic standard deduction for the various filing status are:

<table>
<thead>
<tr>
<th>Filing Status</th>
<th>Standard Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>$4,750</td>
</tr>
<tr>
<td>Married, filing a joint return</td>
<td>$9,500</td>
</tr>
<tr>
<td>Married, filing a separate return</td>
<td>$7,000</td>
</tr>
</tbody>
</table>

2. If a married couple is filing separate returns and one spouse is not taking standard deduction, the other spouse also is not eligible for standard deduction.
3. An additional $1,000 is allowed as standard deduction, if either the filer is 65 yrs or the spouse is 65 yrs or older (the latter case applicable when the filing status is “married” and filing “joint”).
4. An additional $1,000 is allowed as standard deduction, if either the filer is blind or the spouse is blind (the latter case applicable when the filing status is “married” and filing “joint”).

Problem Statement 06
Consider the following program segment:

```java
int max (int i, int j, int k)
{
    int max;
    if (i>j) then
        if (i>k) then max=i;
        else max=k;
    else if (j > k) max=j
    else max=k
    return (max);
}
```

a) Draw the control flow graph for this program segment
b) Determine the cyclomatic complexity for this program
c) Determine the independent paths
Problem Statement 07
Source code of simple insertion sort implementation using array in ascending order in c programming language
#include<stdio.h>
int main(){
    int i,j,s,temp,a[20];
    printf("Enter total elements: ");
    scanf("%d",&s);
    printf("Enter %d elements: ",s);
    for(i=0;i<s;i++)
        scanf("%d",&a[i]);
    for(i=1;i<s;i++){
        temp=a[i];
        j=i-1;
        while((temp<a[j])&&(j>=0)){
            a[j+1]=a[j];
            j=j-1;
        }
        a[j+1]=temp;
    }
    printf("After sorting: ");
    for(i=0;i<s;i++)
        printf(" %d",a[i]);
    return 0;
}

a) Draw the program graph for given program segment
b) Determine the DD path graph
c) Determine the independent paths
d) Generate the test cases for each independent path

Problem Statement 8
Consider a program to input two numbers and print them in ascending order given below. Find all du paths and identify those du-paths that are not feasible. Also find all dc paths and generate the testcases for all paths (dc paths and non dc paths).
#include<stdio.h>
#include<conio.h>
1. void main()
2. {
3   int a,b,t;
4   clrscr();
5   printf("Enter first number");
6   scanf("%d",&a);
Problem Statement 9
Consider the above program and generate possible program slices for all variables. Design at least one test case from every slice.

Problem Statement 10
Consider the code to arrange the nos. in ascending order. Generate the test cases for relational coverage, loop coverage and path testing. Check the adequacy of the test cases through mutation testing and also compute the mutation score for each.

i = 0;
n=4; //N-Number of nodes present in the graph
while (i<n-1) do
    j = i + 1;
    while (j<n) do
        if A[i]<A[j] then
            swap(A[i], A[j]);
        end do;
    end do;
i=i+1; end do;

Problem Statement 11
Create a test plan document for any application (e.g. Library Management System)

Problem Statement 12
Automate the Test cases using Test Automation tool(using selenium tool)
5. PROJECTS TO BE ALLOTED  
(Beyond the syllabus prescribed by G.G.S.I.P.U)

The project is to be submitted at the end of the semester along with a project report by the individual student.

Develop a small project using any language. Perform different types of testing needed during the development of the software and generate corresponding test report.

Students can select project work of his own choice subject to the permission of concern faculty.

NOTE: The project is to be made using any language.
6. FORMAT OF THE LAB RECORD TO BE PREPARED BY THE STUDENTS

1. The front page of the lab record prepared by the students should have a cover page as displayed below.

**NAME OF THE LAB**

Font should be (Size 20”, italics bold, Times New Roman)

Faculty name
Font should be (12”, Times Roman)

Student name
Roll No.:
Semester:
Group:
Font should be (12”, Times Roman)

Maharaja Agrasen Institute of Technology, PSP Area,
Sector – 22, Rohini, New Delhi – 110085
Font should be (18”, Times Roman)
2. The second page in the record should be the index as displayed below.

SOFTWARE TESTING AND QUALITY ASSURANCE

PRACTICAL RECORD

<table>
<thead>
<tr>
<th>PAPER CODE</th>
<th>:</th>
<th>ETCS 453</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of the student</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>University Roll No.</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>Branch</td>
<td>:</td>
<td></td>
</tr>
<tr>
<td>Section/ Group</td>
<td>:</td>
<td></td>
</tr>
</tbody>
</table>

PRACTICAL DETAILS

Experiments according to the lab syllabus prescribed by GGSIPU

<table>
<thead>
<tr>
<th>Exp. no</th>
<th>Experiment Name</th>
<th>Date of performance</th>
<th>Date of checking</th>
<th>Remarks</th>
<th>Marks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PROJECT DETAILS

1. TITLE : 
2. MEMBERS IN THE PROJECT GROUP : 
3. PROJECT REPORT ATTACHED : 
   a) YES    b) NO 
4. SOFT COPY SUBMITTED : 
   a) YES    b) NO 

Signature of the lecturer       Signature of the student

(                                       )

3. Each practical which student is performing in the lab should have the following details:
   a) Topic Detail
   b) AIM
   c) Algorithm
   d) Source Code
   e) Output
   f) Viva questions

4. Project report should be added at last page.
7. MARKING SCHEME FOR THE PRACTICAL EXAMS

There will be two practical exams in each semester.

- Internal Practical Exam
- External Practical Exam

INTERNAL PRACTICAL EXAM

It is taken by the concerned lecturer of the batch.

MARKING SCHEME FOR THIS EXAM IS:

Total Marks: 40

Division of 40 marks is as follows

1. Regularity: 25
   - Performing program in each turn of the lab
   - Attendance of the lab
   - File

2. Viva Voice: 10

3. Project: 5

NOTE: For the regularity, marks are awarded to the student out of 5 for each experiment performed in the lab and at the end the average marks are giving out of 25.
EXTERNAL PRACTICAL EXAM

It is taken by the concerned lecturer of the batch and by an external examiner. In this exam student needs to perform the experiment allotted at the time of the examination, a sheet will be given to the student in which some details asked by the examiner needs to be written and at the last viva will be taken by the external examiner.

MARKING SCHEME FOR THIS EXAM IS:
Total Marks: 60
Division of 60 marks is as follows

1. Sheet filled by the student: 20
2. Viva Voice: 15
3. Experiment performance: 15
4. File submitted: 10

NOTE:

- Internal marks + External marks = Total marks given to the students
  (40 marks)  (60 marks)  (100 marks)

- Experiments given to perform can be from any section of the lab.
8. DETAILS OF EACH SECTION ALONGWITH EXAMPLES, EXERCISES & EXPECTED VIVA QUESTIONS

SECTION 1

What is the purpose of software testing?

The purpose of software testing is
1. To demonstrate that the product performs each function intended;
2. To demonstrate that the internal operation of the product performs according to specification and all internal components have been adequately exercised;
3. To increase our confidence in the proper functioning of the software.
4. To show the product is free from defect.
5. All of the above.

What is software Quality?

Software quality is the degree of conformance to explicit or implicit requirements and expectations.

Explanation:

- Explicit: clearly defined and documented
- Implicit: not clearly defined and documented but indirectly suggested
- Requirements: business/product/software requirements
- Expectations: mainly end-user expectations

Note: Some people tend to accept quality as compliance to only explicit requirements and not implicit requirements. We tend to think of such people as lazy.

Definition by IEEE

- The degree to which a system, component, or process meets specified requirements.
- The degree to which a system, component, or process meets customer or user needs or expectations.

Definition by ISTQB
**Quality**:

The degree to which a component, system or process meets specified requirements and/or user/customer needs and expectations.

**Software Quality**:

The totality of functionality and features of a software product that bear on its ability to satisfy stated or implied needs.

**Cost of Quality (COQ)** is a measure that quantifies the cost of control/conformance and the cost of failure of control/non-conformance. In other words, it sums up the costs related to prevention and detection of defects and the costs due to occurrences of defects.

---

**Testing in the Lifecycle**

![Testing in the Lifecycle Diagram]

**Unit Testing**

Checks each coded module for the presence of bugs. Unit testing's purpose is to ensure that each as-built module behaves according to its specification defined during detailed design.
Integration Testing

Interconnects sets of previously tested modules to ensure that the sets behave as well as they did as independently tested modules. Integration testing’s purpose is to ensure that each as-built component behaves according to its specification defined during preliminary design.

System Testing
Checks that the entire software system embedded in its actual hardware environment behaves according to the requirements document.

Unit Test Planning
Generate plans and procedures to test each module independently and thoroughly.

Integration Test Planning
Generates plans and procedures to effect orderly system integration.

System Test Planning
Development and documentation of test plans and procedures. Examination of the SRD to determine whether or not it is verifiable (is it possible to establish that the software meets the SRD description).

Objectives

Testing cannot show the absence of defects, it can only show that software defects are present.
1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as yet undiscovered error.
3. A successful test is one that uncovers an as yet undiscovered error.

Testing Principles
1. All tests should be traceable to customer requirements.
2. Tests should be planned long before testing begins.
3. 80% of errors are traceable to 20% of the modules.
4. Testing should begin in the small and progress to larger components.
5. Exhaustive testing is NOT possible.
6. Testing is more effective when conducted by an independent party.

Black Box Testing

Black box testing methods focus on the functional requirements of the software. Tests sets are derived that fully exercise all functional requirements. This strategy tends to be applied during the latter part of the lifecycle.

Tests are designed to answer questions such as:
1) How is functional validity tested?

2) What classes of input make good test cases?

3) Is the system particularly sensitive to certain input values?

4) How are the boundaries of data classes isolated?

5) What data rates or volumes can the system tolerate?

6) What effect will specific combinations of data have on system operation?

**a. Equivalence Partitioning**

This method divides the input of a program into classes of data. Test case design is based on defining an equivalent class for a particular input. An equivalence class represents a set of valid and invalid input values.

Guidelines for equivalence partitioning –

There are two distinct steps:

1. **Identifying equivalence classes**
   1) If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
   2) If an input condition requires a *specific value*, one valid and two invalid equivalence classes are defined.
   3) If an input condition specifies a *member of a set*, one valid and one invalid equivalence class are defined.
   4) If an input condition is *boolean*, one valid and one invalid class are defined.

2. **Identifying test cases**
   1) Assign a unique number to each EC.
   2) Until all valid ECs have been covered by test cases, write a new test case covering as many of the uncovered ECs as possible.
   3) Until all invalid ECs have been covered by test cases, write a test case that covers one, and only one, of the uncovered invalid ECs.
   4) If multiple invalid ECs are tested in the same test case, some of those tests may never be executed because the first test may mask other tests or terminate execution of the test case.

Equivalence partitioning significantly reduces the number of input conditions to be tested by identifying classes of conditions that are equivalent to many other conditions. It does not test combinations of input conditions.
Example

Consider an automated banking application. The user can dial the bank from a personal computer, provide a six-digit password, and follow with a series of keyword commands that activate the banking function. The software for the application accepts data in the following form:

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Input Condition</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Code</td>
<td>boolean</td>
<td>The area code may or may not be present</td>
</tr>
<tr>
<td></td>
<td>range</td>
<td>Values between 200 and 999 with area codes requiring 0, 1 in second position</td>
</tr>
<tr>
<td>Prefix</td>
<td>range</td>
<td>Specified value &gt; 200</td>
</tr>
<tr>
<td>Suffix</td>
<td>value</td>
<td>Four-digit length</td>
</tr>
<tr>
<td>Password</td>
<td>boolean</td>
<td>Password may or not be present</td>
</tr>
<tr>
<td></td>
<td>value</td>
<td>Six-character string</td>
</tr>
<tr>
<td>Commands</td>
<td>set</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Input Condition</th>
<th>Equivalence Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Code</td>
<td>boolean</td>
<td>1) no area code given</td>
</tr>
<tr>
<td></td>
<td>range</td>
<td>2) area code given</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1) valid - in range specified</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) invalid - greater</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) invalid - less than</td>
</tr>
<tr>
<td>Prefix</td>
<td>range</td>
<td>1) valid - between 200 and 999 (inclusive)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) invalid - greater than 999</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) invalid - less than 200</td>
</tr>
<tr>
<td>Suffix</td>
<td>value</td>
<td>1) valid - a four-digit number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2) invalid - five-digit number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3) invalid - three-digit number</td>
</tr>
</tbody>
</table>
### Password

<table>
<thead>
<tr>
<th>Password</th>
<th>boolean</th>
</tr>
</thead>
</table>
| value    | 1) no password given  
2) password given |

### Commands

<table>
<thead>
<tr>
<th>Commands</th>
<th>set</th>
</tr>
</thead>
</table>
| value    | 1) valid - command in command set  
2) invalid - command not in command set |

### b. Boundary Value Analysis

Boundary value analysis is complementary to equivalence partitioning. Rather than selecting arbitrary input values to partition the equivalence class, the test case designer chooses values at the extremes of the class. Furthermore, boundary value analysis also encourages test case designers to look at output conditions and design test cases for the extreme conditions in output.

**Guidelines for boundary value analysis -**

1) If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b, and values just above and just below and b.

2) If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values above and below the minimum and maximum are also tested.

3) Apply the above guidelines to output conditions. For example, if the requirement specifies the production of an table as output then you want to choose input conditions that produce the largest and smallest possible table.

4) For internal data structures be certain to design test cases to exercise the data structure at its boundary. For example, if the software includes the maintenance of a personnel list, then you should ensure the software is tested with conditions where the list size is 0, 1 and maximum (if constrained).

### Example

<table>
<thead>
<tr>
<th>Data Item</th>
<th>Input Condition</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Area Code</td>
<td>range</td>
<td>Values between 200 and 999 with area codes requiring 0, 1 in second position</td>
</tr>
</tbody>
</table>
|            |                | **Test Cases:**  
200, 910 (valid end-points) |
<table>
<thead>
<tr>
<th></th>
<th>199, 912 (value below and above)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefix</td>
<td>range Specified value &gt; 200</td>
</tr>
<tr>
<td></td>
<td><strong>Test Cases:</strong></td>
</tr>
<tr>
<td></td>
<td>201 (minimum)</td>
</tr>
<tr>
<td></td>
<td>999 (maximum)</td>
</tr>
<tr>
<td></td>
<td>200 (invalid, just below)</td>
</tr>
<tr>
<td>Suffix</td>
<td>value Four-digit length</td>
</tr>
<tr>
<td></td>
<td><strong>Test Cases:</strong></td>
</tr>
<tr>
<td></td>
<td>0000 (minimum)</td>
</tr>
<tr>
<td></td>
<td>9999 (maximum)</td>
</tr>
<tr>
<td>Password</td>
<td>value Six-character string</td>
</tr>
<tr>
<td></td>
<td><strong>Test Cases:</strong></td>
</tr>
<tr>
<td>Commands</td>
<td>set <strong>Test Cases:</strong></td>
</tr>
</tbody>
</table>

### c. Cause-Effect Graphs

A weakness of the two methods is that do not consider potential combinations of input/output conditions. Cause-effect graphs connect input classes *(causes)* to output classes *(effects)* yielding a directed graph.

Guidelines for cause-effect graphs –

1. Decompose the specification into workable pieces.
2. Identify causes and their effects.
3. Create a (Boolean) cause-effect graph (special symbols are required).
4. Annotate the graph with constraints describing combinations of causes and /or effects that are impossible.
5. The graph is converted to a decision table by methodically tracing state conditions in the graph. Each column in the table represents a test case.
6. Decision table rules are converted to test cases.

### Sample Symbols

<table>
<thead>
<tr>
<th>Identity</th>
<th><img src="image" alt="Sample Symbol" /></th>
</tr>
</thead>
<tbody>
<tr>
<td>Not</td>
<td><img src="image" alt="Sample Symbol" /></td>
</tr>
<tr>
<td>Or</td>
<td>And</td>
</tr>
<tr>
<td>-----</td>
<td>------</td>
</tr>
<tr>
<td><img src="image1.png" alt="Or Diagram" /></td>
<td><img src="image2.png" alt="And Diagram" /></td>
</tr>
</tbody>
</table>
SECTION 2

White-Box Testing

White-box (glass-box) testing is a test case design method that uses the control structure of the procedural design to derive test cases. With white-box methods, tests cases are derived that:

1) guarantee all independent paths in a module have been tested (exercised) at least once;
2) exercise all logical decisions for both true and false conditions;
3) execute all loops at their boundary values and within their operational bounds;
4) exercise internal data structures to ensure their validity.

There are five basic forms of logic coverage:

(1) Statement coverage
(2) Branch coverage
(3) Condition coverage
(4) Relation coverage
(5) Path coverage

Statement coverage: Each statement is executed at least once.
Branch coverage: Each statement is executed at least once, each decision takes on all possible outcomes at least once.
Condition coverage: Each statement is executed at least once; each decision takes on all possible outcomes at least once; each condition in a decision takes on all possible outcomes at least once.
Relational coverage: Each condition is tested for all its possible relational operators.
Path coverage: All independent paths in a module have been at least once.

Path Testing

The basis path method allows for the construction of test cases that are guaranteed to execute every statement in the program at least once. This method can be applied to detailed procedural design or source code.

Method

1. Draw the flow graph corresponding to the procedural design or code.
2. Determine the cyclomatic complexity of the flow graph.
3. Determine the basis set of independent paths. (The cyclomatic complexity indicates the number of paths required.)

4. Determine a test case that will force the execution of each path.

**Flow Graphs**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>IF</th>
<th>While</th>
<th>Repeat</th>
<th>Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Sequence Diagram]</td>
<td>![IF Diagram]</td>
<td>![While Diagram]</td>
<td>![Repeat Diagram]</td>
<td>![Case Diagram]</td>
</tr>
</tbody>
</table>

**Example**

Procedure Average
* This procedure computes the average of 100 or fewer numbers that lie between bounding values. It also computes the sum and the total number of valid entries.
INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;
TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid, minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;

i=1;
total.input=total.valid=0;
sum=0;
DO WHILE value[i]<>-999 and total.input<100
increment total.input by 1;
IF value[i]>=minimum AND value[i]<=maximum
    THEN increment total.valid by 1;
        sum=sum+value[i]
    ELSE skip
ENDIF
increment i by 1;
ENDDO
IF total.valid>0
    THEN average=sum/total.valid;
    ELSE average=-999;
ENDIF
END AVERAGE

Step 1: Construct Flow Graph
a) Identify predicate nodes

Procedure Average
* This procedure computes the average of 100 or fewer numbers that lie between bounding
values. It also computes the sum and the total number of valid entries.
INTERFACE RETURNS average, total.input, total.valid;
INTERFACE ACCEPTS value, minimum, maximum;
TYPE value[1:100] IS SCALAR ARRAY;
TYPE average, total.input, total.valid, minimum, maximum, sum IS SCALAR;
TYPE i IS INTEGER;
i=1;
total.input=total.valid=0; { 1 }
sum=0;
DO WHILE value[i]<>-999 { 2 }
and
total.input<100 { 3}
increment total.input by 1; \{ 4 \}

IF
value[i] >= minimum \{ 5 \}
AND
value[i] <= maximum \{ 6 \}
    THEN increment total.valid by 1; \{ 7 \}
        sum = sum + value[i]
    ELSE skip
ENDIF

increment i by 1; \{ 8 \}
ENDDO \{ 9 \}

IF total.valid > 0 \{ 10 \}
    THEN average = sum / total.valid; \{ 11 \}
    ELSE average = -999; \{ 12 \}
ENDIF \{ 13 \}
END AVERAGE
b) Draw flow graph

Step 2: Determine Cyclomatic Complexity

\[ V(G) = E - N + 2 \]

\[ V(G) = 17 - 13 + 2 = 6 \]

Step 3: Determine the basis set of independent paths.

1-2-10-11-13

1-2-10-12-13

1-2-3-10-11-13

1-2-3-4-5-8-9-2 ...
Step 4: Prepare test cases.

<table>
<thead>
<tr>
<th>Path</th>
<th>Test Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2-10-11-13</td>
<td></td>
</tr>
<tr>
<td>1-2-10-12-13</td>
<td></td>
</tr>
<tr>
<td>1-2-3-10-11-13</td>
<td></td>
</tr>
<tr>
<td>1-2-3-4-5-8-9-2...</td>
<td></td>
</tr>
<tr>
<td>1-2-3-4-5-6-8-9-2...</td>
<td></td>
</tr>
<tr>
<td>1-2-3-4-5-6-7-8-9-2...</td>
<td></td>
</tr>
</tbody>
</table>

**Loop Testing**

**Simple Loops**

The following set of tests should be applied to simple loops, where n is the maximum number of allowable passes:

1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. m passes through the loop where m<n.
5. n-1, n, n+1 passes through the loop

**Nested Loops**

1. Start with the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at this minimum iteration count.
4. Continue until all loop have been tested.

**Concatenated Loops**

Concatenated loops can be tested using the approach defined for simple loops, if the loops are independent. If the loop counter from a loop i is used as the initial value for loop i+1 then the loops are not independent. When loops are not independent use the concatenated loop strategy.

**Unstructured Loops**

Redesign the loops so they are one of the above categories.
SECTION 3

Validation testing (Low-Level testing)
(i) unit (module) testing
(ii) integration testing

Unit (module) testing

Testing a given module(X) in isolation may require:
(1) a driver module which transmits test cases in the form of input arguments to X and either
prints or interprets the results produced by X;
(2) zero or more “stub” modules each of which simulates the function of a module called by
X. It is required for each module that is directly sub-ordinate to X in the execution hierarchy. If
X is a terminal module (i.e., it calls no other modules), then no stubs are required.

Integration testing

The steps in bottom-up integration are:
Begin with the terminal modules(those that do not call other modules) of the hierarchy.
A driver module is produced for every module.
The next module to be tested is any module whose subordinate modules (the modules it calls)
have all been tested.
After a module has been tested, its driver is replaced by an actual module (the next one to be
tested) and its driver.

The steps in top-down integration are:
Begin with the top module in the execution hierarchy.
Stub modules are produced, and some may require multiple versions.
Stubs are often more complicated than they first appear.
The next module to be tested is any module with at least one previously tested superordinate
(calling) module.
After a module has been tested, one of its stubs is replaced by an actual module (the next one to
be tested) and its required stubs.

Drivers

There are several strategies you can take when writing a driver:
• Go through a fixed series of tests. Such a driver is “hard coded” to do exactly the same
  thing every time you run it.
• Prompt the user (the tester) for inputs to test. Such a driver is more general. It can be a
  little more trouble to write than a hard-coded driver, but may be worth the trouble if you expect
  to vary the testing frequently.
• Randomly generate tests.
• Systematically generate tests by looping through a sequence of possibilities.
Stubs

Another strategy is to work from the top down during testing. But how can we test a method before having tested all the methods upon which it depends? This can be accomplished by using "stubs." A stub simulates the behaviour of an untested (or even unwritten) method. There are several ways to implement a stub. The simplest possible stub just prints a message saying that it was called and showing the values of the parameters. Sometimes a stub has to do a little more, for example, if the method must return a value. It may be sufficient for such a stub always return the same, fixed value. If the stub really does need to behave differently in different situations, we can make it do so by asking the user (in this case, the tester), to tell it what answer to return.

Testing need not proceed strictly top-down or strictly bottom-up. A mixture of the two is also possible.

Example

Driver for testing an implementation of algorithm for sorting an array using quicksort.

```java
Public static void main(String [ ] args){
    int [ ] A1={87,98,69,54,65,76,87,89};
    quickSort(A1);
    for ( int i= 0; i<A1.length; i++)
        System.out.println(A1[i]);
    int [ ] A2={1,2,3,4,5,6,7};
    quickSort(A2);
    for (int i=0; i<A2.length; i++)
        System.out.println(A2[i]);
}
```

A hard-coded driver. Only two test cases are shown; usually such a driver would contain a long sequence of tests.
SECTION 4

Total Quality Management (TQM) is an integrative management philosophy for continuous improvement of the quality of an organization's products and processes in order to meet or exceed customer expectations. There are several TQM strategies used to improve business management systems. Considering the practices of TQM as discussed in six empirical studies, Cua, McKone, and Schroeder (2001) identified the nine most common TQM practices as:

1. Cross-functional product design
2. Process management
3. Supplier quality management

Many organizations around the globe develop and implement different standards to improve the quality needs of their software. This chapter briefly describes some of the widely used standards related to Quality Assurance and Testing.

ISO/IEC 9126

This standard deals with the following aspects to determine the quality of a software application:

- Quality model
- External metrics
- Internal metrics
- Quality in use metrics

This standard presents some set of quality attributes for any software such as:

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

The above-mentioned quality attributes are further divided into sub-factors, which you can study when you study the standard in detail.

ISO/IEC 9241-11

Part 11 of this standard deals with the extent to which a product can be used by specified users to achieve specified goals with Effectiveness, Efficiency and Satisfaction in a specified context of use.

This standard proposed a framework that describes the usability components and the relationship between them. In this standard, the usability is considered in terms of user performance and satisfaction. According to ISO 9241-11, usability depends on context of use and the level of usability will change as the context changes.
**ISO/IEC 25000:2005**

ISO/IEC 25000:2005 is commonly known as the standard that provides the guidelines for Software Quality Requirements and Evaluation (SQuaRE). This standard helps in organizing and enhancing the process related to software quality requirements and their evaluations. In reality, ISO-25000 replaces the two old ISO standards, i.e. ISO-9126 and ISO-14598.

**Square** is divided into sub-parts such as:

- ISO 2500n - Quality Management Division
- ISO 2501n - Quality Model Division
- ISO 2502n - Quality Measurement Division
- ISO 2503n - Quality Requirements Division
- ISO 2504n - Quality Evaluation Division

The main contents of SQuaRE are:

- Terms and definitions
- Reference Models
- General guide
- Individual division guides
- Standard related to Requirement Engineering (i.e. specification, planning, measurement and evaluation process)

**ISO/IEC 12119**

This standard deals with software packages delivered to the client. It does not focus or deal with the clients’ production process. The main contents are related to the following items:

- Set of requirements for software packages.
- Instructions for testing a delivered software package against the specified requirements.

**Miscellaneous**

Some of the other standards related to QA and Testing processes are mentioned below:

<table>
<thead>
<tr>
<th>Standard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IEEE 829</td>
<td>A standard for the format of documents used in different stages of software testing.</td>
</tr>
<tr>
<td>IEEE 1061</td>
<td>A methodology for establishing quality requirements, identifying, implementing, analyzing, and validating the process, and the product of software quality metrics.</td>
</tr>
<tr>
<td>IEEE 1008</td>
<td>A standard for unit testing.</td>
</tr>
<tr>
<td>IEEE 1012</td>
<td>A standard for Software Verification and Validation.</td>
</tr>
<tr>
<td>IEEE 1028</td>
<td>A standard for software inspections.</td>
</tr>
<tr>
<td>IEEE 1044</td>
<td>A standard for the classification of software anomalies.</td>
</tr>
<tr>
<td>IEEE 1044-1</td>
<td>A guide for the classification of software anomalies.</td>
</tr>
<tr>
<td>IEEE 830</td>
<td>A guide for developing system requirements specifications.</td>
</tr>
</tbody>
</table>
Difference between Quality Control and Quality Assurance

Quality Control
quality control is considered as a system. It is comprised of routine processes and activities, which are specifically aimed to measure and control the overall quality of the concerned product and service. It also involves accuracy check to ensure zero-error in data calculations and estimating the uncertainties. The quality assurance check should be regular. This is how the QC system can ensure data correctness, completeness and also the integrity. One major part of QC is to distinguish errors and rectify them.

Quality Assurance
As mentioned above QA is a process, which is executed to meet the expectations of customers. There is a set of steps which are followed in order to attain the quality management goals. Six Sigma QA approach and quality infrastructure are gaining sky rocketing popularity in this domain as it is known to make use of a planned and systematic process for quality checks. It is done to prevent defects.
Introduction to Selenium tool

What is Selenium?

Selenium is a free (open source) automated testing suite for web applications across different browsers and platforms. It is quite similar to HP Quick Test Pro (QTP) only that Selenium focuses on automating web-based applications.

Selenium is not just a single tool but a suite of software's, each catering to different testing needs of an organization. It has four components.

- Selenium Integrated Development Environment (IDE)
- Selenium Remote Control (RC)
- WebDriver
- Selenium Grid

Who developed Selenium?

Since Selenium is a collection of different tools, it had different developers as well. Below are the key persons who made notable contributions to the Selenium Project. Primarily, Selenium
was created by Jason Huggins in 2004. An engineer at ThoughtWorks, he was working on a web application that required frequent testing. Having realized that the repetitious manual testing of their application was becoming more and more inefficient, he created a JavaScript program that would automatically control the browser's actions. He named this program as the "JavaScriptTestRunner." Seeing potential in this idea to help automate other web applications, he made JavaScriptRunner open-source which was later re-named as Selenium Core.

Selenium Integrated Development Environment (IDE) is the simplest framework in the Selenium suite and is the easiest one to learn. It is a Firefox plugin that you can install as easily as you can with other plugins. However, because of its simplicity, Selenium IDE should only be used as a prototyping tool. If you want to create more advanced test cases, you will need to use either Selenium RC or WebDriver.

**Introduction to Selenium IDE**

Selenium IDE (Integrated Development Environment) is the simplest tool in the Selenium Suite. It is a Firefox add-on that creates tests very quickly through its record-and-playback functionality. This feature is similar to that of QTP. It is effortless to install and easy to learn.

Because of its simplicity, **Selenium IDE should only be used as a prototyping tool - not an overall solution for developing and maintaining complex test suites.**

Selenium IDE supports autocomplete mode when creating tests. This feature serves two purposes:

- It helps the tester to enter commands more quickly.
- It restricts the user from entering invalid commands.
Features of Selenium IDE

- Menu bar
- Base URL bar
- Toolbar
- Test Case Pane
- Log/Reference/ UI-Element/ Rollup Pane
Test Case Pane

- In Selenium IDE, you can open more than one test case at a time.
- The test case pane shows you the list of currently opened test cases.
- When you open a test suite, the test case pane will automatically list all the test cases contained in it.
- The test case written in bold font is the currently selected test case.
- After playback, each test case is color-coded to represent if it passed or failed.
  - Green color means "Passed."
  - Red color means "Failed."
- At the bottom portion is a summary of the number of test cases that were run and failed.
QUESTION BANK

(Software Testing)

Q1. Explain Cyclomatic Complexity.

Cyclomatic complexity is part of software metrics, by using this the logical complexity of an application can be measured.

(i.e) This is used to find out the minimum no of critical path.

Q2. Explain Boundary value testing and Equivalence testing with some examples.

Boundary value testing is a technique to find whether the application is accepting the expected range of values and rejecting the values which falls out of range.

Ex. A user ID text box has to accept alphabet characters (a-z) with length of 4 to 10 characters.

BVA is done like this, max value:10 pass; max-1: 9 pass;

max+1=11 fail ;min=4 pass;min+1=5 pass;min-1=3 fail;

Like wise we check the corner values and come out with a conclusion whether the application is accepting correct range of values.

Equivalence testing is normally used to check the type of the object.

Ex. A user ID text box has to accept alphabet characters (a-z) with length of 4 to 10 characters.

In +ve condition we have test the object by giving alphabets. i.e a-z char only, after that we need to check whether the object accepts the value, it will pass.

In -ve condition we have to test by giving other than alphabets (a-z) i.e A-Z,0-9,blank etc, it will fail.
Q3. What is Security testing?

It is a process used to look out whether the security features of a system are implemented as designed and also whether they are adequate for a proposed application environment. This process involves functional testing, penetration testing and verification.

Q4. What is Installation testing?

Installation testing is done to verify whether the hardware and software are installed and configured properly. This will ensure that all the system components were used during the testing process. This Installation testing will look out the testing for a high volume data, error messages as well as security testing.

Q5. What is AUT ?

AUT is nothing but "Application Under Test". After the designing and coding phase in Software development life cycle, the application comes for testing then at that time the application is stated as Application Under Test.

Q6. What is Defect Leakage ?

Defect leakage occurs at the Customer or the End user side after the application delivery. After the release of the application to the client, if the end user gets any type of defects by using that application then it is called as Defect leakage. This Defect Leakage is also called as Bug Leak.

Q7. What are the contents in an effective Bug report?

Project, Subject, Description, Summary, Detected By (Name of the Tester), Assigned To (Name of the Developer who is supposed to the Bug), Test Lead (Name), Detected in Version, Closed in Version, Date Detected, Expected Date of Closure, Actual Date of Closure, Priority (Medium, Low, High, Urgent), Severity (Ranges from 1 to 5), Status, Bug ID, Attachment, Test Case Failed (Test case that is failed for the Bug)

Q8. What is Bug Life Cycle?

Bug Life Cycle is nothing but the various phases a Bug undergoes after it is raised or reported.
Q9. What is Error guessing and Error seeding?

Error Guessing is a test case design technique where the tester has to guess what faults might occur and to design the tests to represent them.

Error Seeding is the process of adding known faults intentionally in a program for the reason of monitoring the rate of detection & removal and also to estimate the number of faults remaining in the program.

Q10. What is the difference between Bug, Error and Defect?

Error: It is the Deviation from actual and the expected value.

Bug: It is found in the development environment before the product is shipped to the respective customer.

Defect: It is found in the product itself after it is shipped to the respective customer.

Q11. What is Test bed and Test data?

Test Bed is an execution environment configured for software testing. It consists of specific hardware, network topology, Operating System, configuration of the product to be under test, system software and other applications. The Test Plan for a project should be developed from the test beds to be used.

Test Data is that run through a computer program to test the software. Test data can be used to test the compliance with effective controls in the software.
Q12. What is Negative testing?

**Negative testing** - Testing the system using negative data is called negative testing, e.g. testing the password where it should be minimum of 8 characters so testing it using 6 characters is negative testing.

Q13. Explain Load, Performance and Stress Testing with an Example.

Load Testing and Performance Testing are commonly said as positive testing where as Stress Testing is said to be as negative testing. Say for example there is a application which can handle 25 simultaneous user logins at a time. In load testing we will test the application for 25 users and check how application is working in this stage, in performance testing we will concentrate on the time taken to perform the operation. Where as in stress testing we will test with more users than 25 and the test will continue to any number and we will check where the application is cracking.

Q14. What are SDLC and STLC ? Explain its different phases.

**SDLC**

- Requirement phase
- Designing phase (HLD, DLD (Program spec))
- Coding
- Testing
- Release
- Maintenance

**STLC**

- System Study
- Test planning
- Writing Test case or scripts
- Review the test case
- Executing test case
- Bug tracking
- Report the defect
Q15. What is Ad-hoc testing?

Ad hoc testing is concern with the Application Testing without following any rules or test cases.

For Ad hoc testing one should have strong knowledge about the Application.

Q16. Describe bottom-up and top-down approaches in Regression Testing.

**Bottom-up approach**: In this approach testing is conducted from sub module to main module, if the **main module** is not developed a temporary program called **DRIVERS** is used to simulate the main module.

**Top-down approach**: In this approach testing is conducted from main module to sub module, if the **sub module** is not developed a temporary program called **STUB** is used for simulate the submodule.

Q17. What is the difference between structural and functional testing?

Structural testing is a "white box" testing and it is based on the algorithm or code.

Functional testing is a "black box" (behavioral) testing where the tester verifies the functional specification.

Q18. What is Re-test ? What is Regression Testing ?

**Re-test**: Retesting means we testing only the certain part of an application again and not considering how it will effect in the other part or in the whole application.

**Regression Testing**: Testing the application after a change in a module or part of the application for testing that is the code change will affect rest of the application.

Q19. What is UAT testing? When it is to be done?

**UAT testing**: UAT stands for 'User acceptance Testing. This testing is carried out with the user perspective and it is usually done before the release.
Q20. What are the basic solutions for the software development problems?

- **Basic requirements** - clear, detailed, complete, achievable, testable requirements has to be developed. Use some prototypes to help pin down requirements. In nimble environments, continuous and close coordination with customers/end-users is needed.
- **Schedules should be realistic** - enough time to plan, design, test, bug fix, re-test, change, and document in the given schedule.
- **Adequate testing** – testing should be started early, it should be re-tested after the bug fixed or changed, enough time should be spend for testing and bug-fixing.
- **Proper study on initial requirements** – be ready to look after more changes after the development has begun and be ready to explain the changes done to others. Work closely with the customers and end-users to manage expectations. This avoids excessive changes in the later stages.
- **Communication** – conduct frequent inspections and walkthroughs in appropriate time period; ensure that the information and the documentation is available on up-to-date if possible electronic. More emphasize on promoting teamwork and cooperation inside the team; use prototypes and proper communication with the end-users to clarify their doubts and expectations.

Q21. What are the common problems in the software development process?

- **Inadequate requirements from the Client** - if the requirements given by the client is not clear, unfinished and not testable, then problems may come.
- **Unrealistic schedules** – Sometimes too much of work is being given to the developer and ask him to complete in a Short duration, then the problems are unavoidable.
- **Insufficient testing** – The problems can arise when the developed software is not tested properly.
- **Given another work under the existing process** – request from the higher management to work on another project or task will bring some problems when the project is being tested as a team.
- **Miscommunication** – in some cases, the developer was not informed about the Clients requirement and expectations, so there can be deviations.
Q22. Why does software have bugs?

- **Miscommunication or no communication** – about the details of what an application should or shouldn't do
- **Programming errors** – in some cases the programmers can make mistakes.
- **Changing requirements** – there are chances of the end-user not understanding the effects of changes, or may understand and request them anyway to redesign, rescheduling of engineers, effects of other projects, work already completed may have to be redone or thrown out.
- **Time force** - preparation of software projects is difficult at best, often requiring a lot of guesswork. When deadlines are given and the crisis comes, mistakes will be made.

Q23. What software testing types can be considered?

**Black box testing** – This type of testing doesn’t require any knowledge of the internal design or coding. These Tests are based on the requirements and functionality.

**White box testing** – This kind of testing is based on the knowledge of internal logic of a particular application code. The Testing is done based on the coverage of code statements, paths, conditions.

**Unit testing** – the 'micro' scale of testing; this is mostly used to test the particular functions or code modules. This is typically done by the programmer and not by testers; it requires detailed knowledge of the internal program design and code. It cannot be done easily unless the application has a well-designed architecture with tight code; this type may require developing test driver modules or test harnesses.

**Sanity testing or Smoke testing** – This type of testing is done initially to determine if a new software version is performing well enough to accept it for a major testing effort. For example, if the new software is crashing the systems in every 5 minutes or corrupting databases, the software may not be in a 'sound’ condition to proceed for further testing in its current state.

**Functional testing** – This a commonly used black-box testing geared to check the functional requirements of an application; this type of testing should be done by testers.
**Integration testing** – This testing is combining the ‘parts’ of an application to determine if they function together correctly. The ‘parts’ can be code modules, individual applications, client and server applications on a network, etc. This type of testing is especially relevant to the client/server and distributed systems.

**Incremental Integration testing** – This is continuous testing of an application when a new functionality is added the existing ones; it checks the application functionality by verifying whether it works separately before all parts of the program are completed, in this type it will be checked whether to introduce test drivers or not; this is done by programmers or by testers.

**Regression testing** – This is testing the whole application again after the fixes or the modifications are done on the software. This is mostly done at the end of the Software development life cycle. Mostly Automated testing tools are used for these type of testing.

**System testing** – This is a type of black-box type testing that is based on overall requirements specifications; covers all combined parts of a system.

**End-to-end testing** – This is similar to system testing; this involves testing of a complete application environment such as interacting with a database, using network communications, or interacting with other hardware, applications and so on.

**UAT ( User Acceptance Testing )** – This type of testing comes on the final stage and mostly done on the specifications of the end-user or client.

**Usability testing** – This testing is done to check the 'user-friendliness' of the application. This depends on the targeted end-user or customer. User interviews, surveys, video recording of user sessions, and other techniques can be used. Programmers and testers are usually not appropriate as usability testers.

**Compatibility testing** – Testing how well the software performs in a particular hardware, software, operating system, network etc.

**Comparison testing** – This is nothing comparing the software strengths and weakness with another competing product.
Mutation testing – This is another method for determining if a set of test data or test cases is useful, by purposely introducing various code changes or bugs and retesting with the original test data or cases to determine whether the 'bugs' are detected.

Q24. How do you decide when you have 'tested enough’?

Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
  - Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
  - Bug rate falls below a certain level
  - Beta or alpha testing period ends

Q25. Describe the Software Development Life Cycle

It includes aspects such as initial concept, requirements analysis, functional design, internal design, documentation planning, test planning, coding, document preparation, integration, testing, maintenance, updates, retesting, phase-out, and other aspects.

Q26. Describe the difference between validation and verification

Verification is done by frequent evaluation and meetings to appraise the documents, policy, code, requirements, and specifications. This is done with the checklists, walkthroughs, and inspection meetings.

Validation is done during actual testing and it takes place after all the verifications are being done.

Q27. What is the difference between QA and testing?

Testing involves operation of a system or application under controlled conditions and evaluating the results. It is oriented to 'detection'.
Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

Q28. What is quality assurance?

Software QA involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with. It is oriented to 'prevention'.

Q29. What is the purpose of the testing?

Software testing is the process used to help identify the Correctness, Completeness, Security and Quality of the developed Computer Software.

Software Testing is the process of executing a program or system with the intent of finding errors.

Q30. You are shown the code for another module in the Zachariah system, this time for calculating the cost of wine orders. The wine is sold in boxes of 12 bottles, and is of three types: A. Shiraz, B. Cabernet Sauvignon, and C. Lambrusco. The code is as follows:

```plaintext
  Begin
    cost_of_boxA = 60;
    cost_of_boxB = 80;
    cost_of_boxC = 100;
    if (no_of_boxA == 2)
      discount = 5;
    elseif (no_of_boxA == 1 AND no_of_boxB == 1)
      discount = 8;
    elseif (no_of_boxC == 3)
      discount = 10;
    else
      discount = 0;
    ifend
```
return (cost_to_customer);
end Procedure CostOfWine;

Write a **test driver** for this module.

Test driver for the module is given below:

Procedure CostOfWine_driver is
Begin
  Write “enter no. of box A”;
  Read no_of_BoxA;
  Write “enter no. of box B”;
  Read no_of_BoxB;
  Write “enter no. of box C”;
  Read no_of_boxC;
  Cost_to_customer = CostOfWine(no_of_BoxA, no_of_BoxB, no_of_boxC);
  write Cost_to_Customer;
end procedure CostOfWine_driver;

Q31. **Design test cases to force execution down each independent path.**
An independent path is a path that has some unique part.
The aim is to execute all statements at least once.
All conditional statements must therefore be tested for both true / false cases.

<table>
<thead>
<tr>
<th>Input sequence</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 0, 0</td>
<td>cost_to_customer = 106 (5% discount)</td>
</tr>
<tr>
<td>1, 1, 0</td>
<td>cost_to_customer = 126 (8% discount)</td>
</tr>
<tr>
<td>0, 0, 3</td>
<td>cost_to_customer = 255 (10% discount)</td>
</tr>
<tr>
<td>0, 0, 0</td>
<td>cost_to_customer = 0 (no discount)</td>
</tr>
</tbody>
</table>

note: where there is a zero in the input column, this can be any number but will change the cost_to_customer figure.

Q32. You are asked to test a module, using Black Box testing, which calculates the cost of buying cans of beer from Zachariah’s Liquor Emporium. The basic cost per can is £1, but discounts are available for bulk orders. If you order between 10 and 99 cans you qualify for a 10% discount, while if you order 100 cans or more you qualify for a 20% discount. The module should take as input the required number of cans, and output the price of the order.
Display your test suite using the standard five column format: test case number, function tested, input, expected output, and actual output (leave the final column blank at this stage).

There are three equivalence classes: 0 - 9 cans, the no discount range; 10 - 99 cans, the 10% discount range; 100+ cans, the 20% discount range. For each range, test the lower bound, mid range and upper bound values. For the 0% range these would be 0, 4 and 9; for the 10% range these would be 10, 50 and 99; and for the 20% range these would be 100, 200 (there is no upper bound). Also test one negative number, and one non-integer, e.g. 2.5.

The expected outputs then would be 0, 4 and 9 for the 0% range; 9, 45 and 89.10 for the 10% range, 80 and 160 for the 20% range, we would need white box testing to find the expected output for a negative input, and inputting a non-integer should result in an error message.

Q32. What's an 'inspection'?

An inspection is more formalized than a 'walkthrough', typically with 3-8 people including a moderator, reader, and a recorder to take notes. The subject of the inspection is typically a document such as a requirements spec or a test plan, and the purpose is to find problems and see what's missing, not to fix anything. Attendees should prepare for this type of meeting by reading thru the document; most problems will be found during this preparation. The result of the inspection meeting should be a written report. Thorough preparation for inspections is difficult, painstaking work, but is one of the most cost effective methods of ensuring quality.

Q33. What are 5 common problems in the software development process?

- poor requirements - if requirements are unclear, incomplete, too general, and not testable, there will be problems.
- unrealistic schedule - if too much work is crammed in too little time, problems are inevitable.
- inadequate testing - no one will know whether or not the program is any good until the customer complains or systems crash.
- featuritis - requests to pile on new features after development is underway; extremely common.
- miscommunication - if developers don't know what's needed or customer's have erroneous expectations, problems are guaranteed.

Q34. What's a 'test plan'?

A software project test plan is a document that describes the objectives, scope, approach, and focus of a software testing effort. The process of preparing a test plan is a useful way to think through the efforts needed to validate the acceptability of a software product. The completed document will help people outside the test group understand the 'why' and 'how' of product validation. It should be thorough enough to be useful but not so thorough that no one outside the test group will read it. The following are some of the items that might be included in a test plan, depending on the particular project:
- Title
- Identification of software including version/release numbers
- Revision history of document including authors, dates, approvals
- Table of Contents
- Purpose of document, intended audience
- Objective of testing effort
- Software product overview
- Relevant related document list, such as requirements, design documents, other test plans, etc.
- Relevant standards or legal requirements
- Traceability requirements
- Relevant naming conventions and identifier conventions
- Overall software project organization and personnel/contact-info/responsibilities
- Test organization and personnel/contact-info/responsibilities
- Assumptions and dependencies
- Project risk analysis
- Testing priorities and focus
- Scope and limitations of testing
- Test outline - a decomposition of the test approach by test type, feature, functionality, process, system, module, etc. as applicable
- Outline of data input equivalence classes, boundary value analysis, error classes
- Test environment - hardware, operating systems, other required software, data configurations, interfaces to other systems
- Test environment validity analysis - differences between the test and production systems and their impact on test validity.
- Test environment setup and configuration issues
- Software migration processes
- Software CM processes
- Test data setup requirements
- Database setup requirements
- Outline of systemlogging/error logging/other capabilities, and tools such as screen capture software, that will be used to help describe and report bugs
- Discussion of any specialized software or hardware tools that will be used by testers to help track the cause or source of bugs
- Test automation - justification and overview
- Test tools to be used, including versions, patches, etc.
- Test script/test code maintenance processes and version control
- Problem tracking and resolution - tools and processes
- Project test metrics to be used
- Reporting requirements and testing deliverables
- Software entrance and exit criteria
- Initial sanity testing period and criteria
- Test suspension and restart criteria
- Personnel allocation
- Personnel pre-training needs
- Test site/location
Outside test organizations to be utilized and their purpose, responsibilities, deliverables, contact persons, and coordination issues
- Relevant proprietary, classified, security, and licensing issues.
- Open issues
- Appendix - glossary, acronyms, etc.

Q35. How can it be known when to stop testing?

This can be difficult to determine. Many modern software applications are so complex, and run in such an interdependent environment, that complete testing can never be done. Common factors in deciding when to stop are:

- Deadlines (release deadlines, testing deadlines, etc.)
- Test cases completed with certain percentage passed
- Test budget depleted
- Coverage of code/functionality/requirements reaches a specified point
- Bug rate falls below a certain level
- Beta or alpha testing period ends