**STQA**

**ASSIGNMENT NO. 1**

# **Q1. Differentiate between Software Testability and Reliability with their relative importance.**

Importance of Software Testability:

1. **Helps in saving money**: The cost-effectiveness of the project happens to be one of the top reasons why companies go for software testing Services. The testing of software comprises a bunch of projects. In case you find any bug in the early phases, fixing them costs a reduced amount of money.
2. **Security**: It is another crucial point why software testing should not be taken into consideration. It is considered to be the most vulnerable and sensitive part. There are a bunch of situations in which the information and details of the users are stolen and they are used for the benefits.
3. **Quality of the product**: For ensuring that the specific product comes to life, it should work in accordance with the following. Following the needs of the product is a prerequisite as it is helpful in getting the prerequisite results. Products should be serving the user in one way or the other. It is a must that it is going to bring value, as per the promise.
4. **Satisfaction of the customer**: The primary objective of the owner of the products is offering the best satisfaction of the customers. The reasons why it is necessary to opt for software testing is due to the fact that it offers the prerequisite and perfect user experience. As you opt for the best project in the saturated project, you will be capable of earning the reputation of reliable clients.
5. **Enhancing the development process**: With the aid of Quality Assurance, you can find a wide array of scenarios and errors, for the reproduction of the error.
6. **Easy while adding new features**: The more interconnected and older the code, the more difficult it is to change. Tests counteract this calcification tendency by allowing developers to confidently add new features. As a new developer, changing older parts of your codebase can be terrifying, but with tests, you’ll at least know if you’ve broken anything important.
7. **Determining the performance of the software**: If you find software or application that has low or reduced performance, you will find that it brings your reputation down in the market. Users are not going to trust any people. There are chances that the reputation of your organization is going to suffer.

Importance of Software Reliability:

1. **Computers are now cheap and fast**: There is little need to maximize equipment usage. Paradoxically, however, faster equipment leads to increasing expectations on the part of the user so efficiency considerations cannot be completely ignored.
2. **Unreliable software is liable to be discarded by users**: If a company attains a reputation for unreliability because of a single unreliable product, it is likely to affect future sales of all of that company’s products.
3. **System failure costs may be enormous**: For some applications, such as a reactor control system or an aircraft navigation system, the cost of system failure is orders of magnitude greater than the cost of the control system.
4. **Unreliable systems are difficult to improve**: It is usually possible to tune an inefficient system because most execution time is spent in small program sections. An unreliable system is more difficult to improve as unreliability tends to be distributed throughout the system.
5. **Inefficiency is predictable**: Programs take a long time to execute and users can adjust their work to take this into account. Unreliability, by contrast, usually surprises the user. Software that is unreliable can have hidden errors which can violate system and user data without warning and whose consequences are not immediately obvious. For example, a fault in a CAD program used to design aircraft might not be discovered until several plane crashes occur.
6. **Unreliable systems may cause information loss**: Information is very expensive to collect and maintain, it may sometimes be worth more than the computer system on which it is processed. A great deal of effort and money is spent duplicating valuable data to guard against data corruption caused by unreliable software.

# **Q2. Discuss relative advantages and disadvantages of Top-down and Bottom-up Integration testing approaches.**

Bottom-up Integration testing: In the bottom-up approach, system integration begins with the integration of the lowest level modules. Bottom-up integration testing also uses test drivers to drive and pass appropriate data to the lower level modules. As and when the code for the other module gets ready, these drivers are replaced with the actual module.

Advantages of bottom-up integration testing:

1. If the low level modules and their combined functions are often invoked by other modules, then it is more useful to test them first so that meaningful effective integration of other modules can be done.
2. Appropriate for applications where bottom up design methodology is used.
3. Advantageous if major flaws occur towards the bottom of the program.
4. Test conditions are easier to create.
5. Observation of test results is easier.
6. Always starting at the bottom of the hierarchy again means that the critical modules are generally built and tested first and therefore any errors or mistakes in these forms of modules are identified early in the process.

Disadvantages of bottom-up integration testing:

1. Test engineers cannot observe system level functions from a partly integrated system. They cannot observe the system level functions until the top level test driver is in place.
2. The program as an entity does not exist until the last module is added.
3. One big disadvantage of bottom up strategy is that in this sort of testing no working model can be represented as far as several modules have been built.

Top-down Integration testing: Top-down integration testing is an integration testing technique used in order to simulate the behavior of the lower-level modules that are not yet integrated. Stubs are the modules that act as temporary replacement for a called module and give the same output as that of the actual product.

Advantages of top-down integration testing:

1. Isolation of interface errors becomes easier because of the incremental nature of top down integration.
2. Test cases designed to test the integration of a module are reused during the regression tests performed after integrating other modules.
3. Advantageous if major flaws occur toward the top of the program.
4. It provides an early working module of the program and so design defects can be found and corrected early.

Disadvantages of top-down integration testing:

1. Until a certain set of modules have been integrated, it may not be possible to observe meaningful system functions because of an absence of lower level modules and the presence of stubs.
2. Test case selection and stub design become increasingly difficult when stubs lie far away from the top level module.
3. Observation of test output is more difficult.
4. Stubs have to be written with utmost care as they will simulate setting of output parameters.

# **Q3. Generate a Cause Effect Graph for a web based company selling Computers (CPU), Printers (PR), Monitors (M) and additional Memory (RAM) of the purchase order:**

* M20 and M23 can be purchased with any CPU or as a standalone item but M30 can only purchase with CPU3
* Purchase of CPU1 gets RAM256 upgrade and PR1 and RAM512 are available free with the purchase of CPU2 or CPU3.
* The RAM 1GB upgrade and a free PR2 is available when CPU3 is purchased with monitor M30.
* Monitors and Printers except for M30, can also be purchased with purchasing any CPU.

Sample configurations and contents of the free window are given below:

| Items Purchased | Free Window |
| --- | --- |
| CPU 1 | RAM 256 |
| CPU 2 | PR 1, RAM 512 |
| CPU 3 | PR 1, RAM 512 |
| CPU 3, M30 | PR 2, RAM 1G |

Causes we have:

C1: Purchase of CPU 1

C2: Purchase of CPU 2

C3: Purchase of CPU 3

C4: Purchase of PR 1

C5: Purchase of PR 2

C6: Purchase of M 20

C7: Purchase of M 23

C8: Purchase of M 30

Set of Effects we have:

EF1: RAM 256

EF2: RAM 512 and PR 1

EF3: RAM 1G and PR 2

EF4: No giveaway

Cause-Effect graph formed is:



# **Q4. A program reads three numbers from -100 to 100 and prints the smallest number. Design test cases for this program using Equivalence Class Partitioning technique.**

Equivalence Class Partitioning: Equivalence partitioning or equivalence class partitioning (ECP) is a software testing technique that divides the input data of a software unit into partitions of equivalent data from which test cases can be derived. In principle, test cases are designed to cover each partition at least once.

Code for the program will be:

| n1 = int(input("Enter your first number:")) n2 = int(input("Enter your second number: ")) n3 = int(input("Enter your third number: "))  if (n1 < n2 and n1 < n3):  print("{} is smallest".format(n1)) elif (n2 < n1 and n2 < n3):  print("{} is smallest".format(n2)) elif (n3 < n1 and n3 < n2):  print("{} is smallest".format(n3))  else:  print("All numbers are equal") |
| --- |

Test cases for this program using equivalence class testing technique:

| Test Case No. | Test Value | Expected Value | EC Being Tested |
| --- | --- | --- | --- |
| TC1 | n1 = 6  n2 = 8  n3 = 10 | 6 | EC1 |
| TC2 | n1 = 10  n2 = 12  n3 = 2 | 2 | EC2 |
| TC3 | n1 = 12  n2 = 1  n3 = 5 | 1 | EC3 |
| TC4 | n1 = 10  n2 = 10  n3 = 10 | 10 | EC4 |

# **Q5. Draw a Control flow graph and find Cyclomatic Complexity for the following:**

*if (c1 or c2 and c3) s1;*

*else s2;*

*while(c4) s3;*

*s4;*

*do s5; while(c5);*

*S6*

Cyclomatic Complexity: Cyclomatic Complexity in Software Testing is a testing metric used for measuring the complexity of a software program. It is a quantitative measure of independent paths in the source code of a software program. Cyclomatic complexity can be calculated by using control flow graphs or with respect to functions, modules, methods or classes within a software program.

Mathematically, it is a set of independent paths through the graph diagram. The Code complexity of the program can be defined using the formula:

**V(G) = E - N + 2**

Where, E is the number of edges and N is the number of nodes.

Control flow graph:



Cyclomatic Complexity = E - N + 2 = 11 - 9 + 2 = **4**

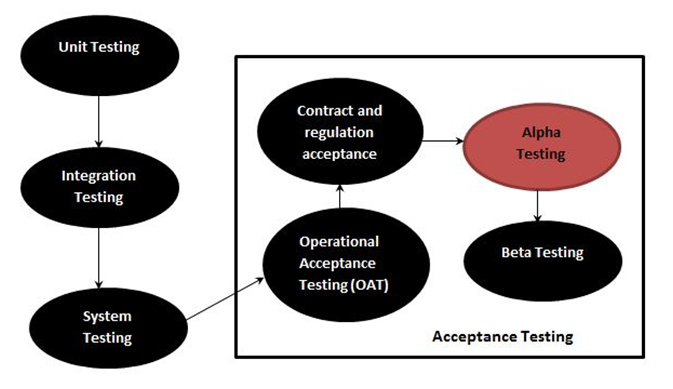
# **Q6. Design test cases to find a maximum of 4 numbers.**

| Steps | Description | Input Data | Expected Result | Actual Result |
| --- | --- | --- | --- | --- |
| 1 | Enter 4 numbers | 0 to 9 | Inputs Accepted | 10, 15, 2, 8 |
| 2 | Enter numbers and alphabets | 0 to 9  A to Z  a to z | Invalid Inputs | Invalid |
| 3 | Enter numbers and special characters | 0 to 9  ~, !, @, #, $, %, ^, &, \*, (, ), \_, - | Invalid Inputs | Invalid |
| 4 | Enter numbers and spaces | 0 to 9  space | Invalid Inputs | Invalid |
| 5.1 | Comparing 1st pair of accepted inputs | (10 > 15) ? 10 : 15 | 15 | 15 |
| 5.2 | Comparing 2nd pair of accepted inputs | (15 > 2) ? 15 : 2 | 15 | 15 |
| 5.3 | Comparing 3rd pair of accepted inputs | (2 > 8) ? 2 : 8 | 8 | 8 |
| 6.1 | Comparing obtained 1st output pairs | (15 => 15) ? 15 : 15 | 15 | 15 |
| 6.2 | Comparing obtained 2nd output pairs | (15 => 8) ? 15 : 8 | 15 | 15 |

# **Q7. Explain Entry and Exit Criteria for Alpha and Beta Testing. State of Alpha testing differs from Beta testing.**

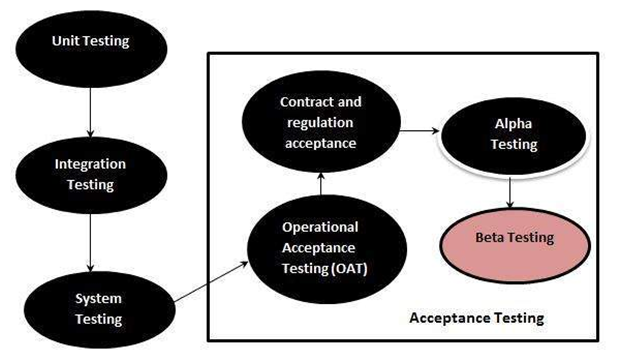
Alpha Testing: Alpha Testing is a type of acceptance testing; performed to identify all possible issues and bugs before releasing the final product to the end users. Alpha testing is carried out by the testers who are internal employees of the organization. The main goal is to identify the tasks that a typical user might perform and test them.

The following diagram explains the fitment of Alpha testing in the software development life cycle:



Beta Testing: Beta Testing is performed by “real users” of the software application in “real environment” and it can be considered as a form of external User Acceptance Testing. It is the final test before shipping a product to the customers. Direct feedback from customers is a major advantage of Beta Testing. This testing helps to test products in the customer's environment.

The following diagram explains the fitment of Beta testing in the software development life cycle:



Entry criteria for Alpha Testing:

1. All features are complete/testable(no urgent bugs).
2. High bugs on primary platforms are fixed/verified.
3. 50% of minimum bugs on primary platforms are fixed/verified.
4. All the features have been tested on primary platforms.
5. Performance has been measured/compared with the previous releases (user function).
6. Usability testing & feedback(ongoing).
7. Aloha sites are ready for installation.

Exit criteria for Alpha Testing:

1. Get responses/feedback from the customers.
2. Create a report of any serious bugs.
3. Notify bug-fixing issues to the developers.

Entry criteria for Beta Testing:

1. Positive responses from the alpha sites.
2. Customer bugs in the alpha testing have been addressed.
3. There are no such fatal errors that can affect the functionality of the software.
4. Secondary platform compatibility testing is complete.
5. Regression tests corresponding to the hug fixes have been done.
6. Beta sites are ready for installation.

Exit criteria for Beta Testing:

1. Get responses/feedback from the beta testers.
2. Create a report of all serious bugs.
3. Notify bug-fixing issues to the developers.

| Alpha Testing | Beta Testing |
| --- | --- |
| Alpha testing is conducted by the end user at developers side. | Beta testing is conducted by the end user at his own side. |
| During alpha testing the developer is present. | During alpha testing the developer is not present. |
| Since the environment is controlled, the end user cannot use software in his own way. | Since the environment is controlled, the end user can use software in his own way. |
| Real time bugs cannot be identified. | Real time bugs can be identified. |
| This testing is conducted by people within the company itself. | This testing is conducted by people outside the company or by another company. |
| This testing is conducted before software release to the general public. | This testing is conducted after alpha testing. |
| This testing is not done to do advertisement of a software. | This testing is done to do advertisement of a software. |
| It requires testing costs. | It does not require testing cost. |
| It uses black box testing and gray box testing. | It uses black box testing. |
| Examples: Windows XP, Adobe Photoshop | Examples: Media classic player, Avast Antivirus |

# **Q8. A program reads an integer number within the range [1,100] and determines whether it is a prime number or not. Design test cases for this program using Boundary Value Analysis.**

Boundary Value Analysis: Boundary Value Analysis is based on testing the boundary values of valid and invalid partitions. The behavior at the edge of the equivalence partition is more likely to be incorrect than the behavior within the partition, so boundaries are an area where testing is likely to yield defects.

It checks for the input values near the boundary that have a higher chance of error. Every partition has its maximum and minimum values and these maximum and minimum values are the boundary values of a partition.

No. of variables (n) = **1**

Total no. of test cases: **4n+1= 5**

The set of minimum and maximum values is shown below:

Minimum = **1**

Minimum + = **2**

Maximum = **100**

Maximum - = **99**

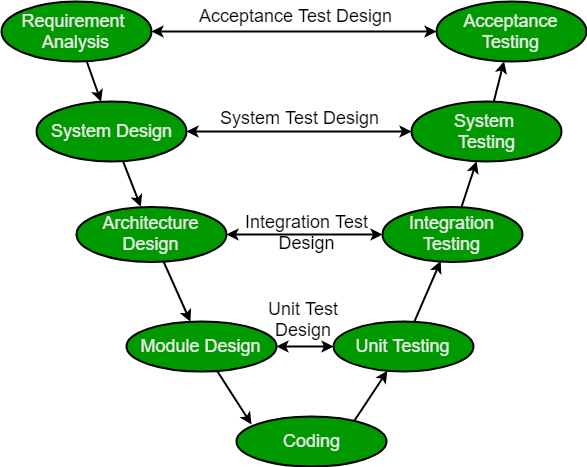
Nominal = **40 to 45**

Using these values, test cases can be designed as shown below:

| Test Case ID | Integer Variable | Expected Output |
| --- | --- | --- |
| 1 | 1 | Not a prime number |
| 2 | 2 | Prime number |
| 3 | 100 | Not a prime number |
| 4 | 99 | Not a prime number |
| 5 | 43 | Prime number |

# **Q9. Explain V-Model in detail.**

The V-model is a type of SDLC model where the process executes in a sequential manner in V-shape. It is also known as the Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.



Verification: It involves static analysis technique (review) done without executing code. It is the process of evaluation of the product development phase to find whether specified requirements meet.

Validation: It involves dynamic analysis technique (functional, non-functional), testing done by executing code. Validation is the process to evaluate the software after the completion of the development phase to determine whether software meets the customer expectations and requirements.

Design Phase:

1. **Requirement Analysis**: This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
2. **System Design**: This phase contains the system design and the complete hardware and communication setup for developing the product.
3. **Architectural Design**: System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
4. **Module Design**: In this phase the system breaks down into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

Testing Phases:

1. **Unit Testing**: Unit Test Plans are developed during the module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
2. **Integration testing**: After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
3. **System Testing**: System testing tests the complete application with its functionality, inter dependency, and communication.It tests the functional and nonfunctional requirements of the developed application.
4. **User Acceptance Testing (UAT)**: UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets the user's requirement and the system is ready for use in the real world.

Principles of V-Model:

1. **Large to Small**: In V-Model, testing is done in a hierarchical perspective, For example, requirements identified by the project team, create High-Level Design, and Detailed Design phases of the project. As each of these phases is completed the requirements become more and more refined and detailed.
2. **Data/Process Integrity**: This principle states that the successful design of any project requires the incorporation and cohesion of both data and processes. Process elements must be identified at each and every requirement.
3. **Scalability**: This principle states that the V-Model concept has the flexibility to accommodate any IT project irrespective of its size, complexity or duration.
4. **Cross Referencing**: Direct correlation between requirements and corresponding testing activity is known as cross-referencing.
5. **Tangible Documentation**: This principle states that every project needs to create a document. This documentation is required and applied by both the project development team and the support team. Documentation is used to maintain the application once it is available in a production environment.

Advantages of the V-Model are:

* This is a highly disciplined model and Phases are completed one at a time.
* V-Model is used for small projects where project requirements are clear.
* Simple and easy to understand and use.
* This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
* It enables project management to track progress accurately.

Disadvantages of the V-Model are:

* High risk and uncertainty.
* It is not good for complex and object-oriented projects.
* It is not suitable for projects where requirements are not clear and contains high risk of changing.
* This model does not support iteration of phases.
* It does not easily handle concurrent events.

# **Q10. Explain Verification of high level design and low level design.**

All the requirements mentioned in SRS documents are addressed in this phase and work in direction of designing the solution. High level design takes the second place in SDLC, wherein there is a high probability of finding bugs. Therefore high level/design must be verified as the next step in early testing. If a bug goes undetected in the high level design phase, then its cost of fixing increases with every phase.

Verification of High level design: It is divided into three parts.

1. Data design
2. Architectural design
3. Interface design

Data Design: It creates a model of data that is represented at a high level of abstraction. At the program component level, the design of data structure and associated algorithms required to manipulate them is essential to create high quality applications.

Verification of Data Design:

* Check for sizes of data structure have been estimated properly.
* Check the provisions of overflow.
* Check consistency of data format with requirements.
* Check data usage is consistent with its declaration.
* Check for relationships among data objects in the data dictionary.
* Check consistency of database and data warehouse with requirements specified in SRS.

Architectural Design: It focuses on representation of structure of s/w components, their properties and interactions.

Verification of Architectural Design:

* Check for every functional requirement from SRS that is included in design.
* Check all exception handling conditions.
* Verify process of transform mapping and transaction mapping transition from requirement model to architectural design.
* Check functionality of each module according to requirements specified.
* Check interdependence and interface between modules.
* Testers should also verify coupling and cohesion. A good design should have low coupling and high cohesion.

Interface Design: It creates an effective communication medium between interfaces of different software modules, interfaces between software systems and any other external entity and interface between user and software system.

Verification of Interface Design:

* Check all interfaces between modules according to architectural design.
* Check all interfaces between software and non-human producer and consumer information.
* Check interfaces between human and computer.
* Check interfaces for their consistency.
* Check response time for all interfaces within required ranges.
* Check for help facility and error handling messages.
* For typed command interactions, check mapping between every menu option and their corresponding command.

Verification of Low level design: In low level design a detailed design of modules and data are prepared such that an operational software is ready. For this SOD is preferred where all modules and their interfaces are defined. For verification of low level design SRS and SDD of individual modules are referred.

1. 4 verify SRS of individual modules.
2. 4 verify SDD of each individual module.
3. 4 in Low level design, data structure, interfaces and algorithms are represented by design notations, which verify the consistency of every item with their design notations.