

USN



Internal Assessment Test I – March. 2019

Sub:	Software Testing and Practices			Sub Code:	17MCA43	Branch:	MCA
Date:	06.02.2019	Duration:	90 mins	Max Marks:	50	Sem	IV
							OBE

Note : Answer FIVE FULL Questions, choosing ONE full question from each part

		MARKS	CO	RBT
PART I				
1	a. Explain the relationship of Human, Errors and Testing with example. b. List out quality attributes of software and explain each of them OR	[5+5]	CO1	L4, L1
2	a. Define the terminologies organizational metrics, project metrics, process metrics and product metrics with example. b. Differentiate between verification and validation	[5+5]	CO1	L1, L3
PART II				
3	a. How is hardware testing different from software testing? b. Discuss the defect life-cycle and draw an appropriate diagram OR	[5+5]	CO1	L3, L2
4.	a. Discuss how levels of testing are associated with levels of software development. Draw a supporting diagram. b. List down the best practices for writing good test case.	[5+5]	CO2	L2, L1
PART III				
5	a. Discuss the six basic principles underlying the analysis and testing techniques. b. Explain the difference between error, fault, failure and incident. OR	[5+5]	CO1	L2, L4
6	a. Discuss defect severity and defect priority with suitable examples. b. Compare specification based testing and code based testing.	[5+5]	CO2	L2, L3
PART IV				
7	a. Explain the Next Date problem and write the pseudo code. b. List down the different fault types and give two examples of each. OR	[5+5]	CO2	L4, L1
8	a. Explain BVA test case for two variables functions and limitations of BVA b. Explain Robustness Testing	[5+5]	CO2	L4, L4
PART V				
9	a. Explain the Test Case Generation Strategies b. Represent Triangle problem in a decision table and generate test cases from that OR	[5+5]	CO2	L4, L4
10	a. Discuss Test-Debug cycle with the help of a diagram b. How is reliability different from correctness? Testing aims at which one of these two and why?	[5+5]	CO2	L2, L3

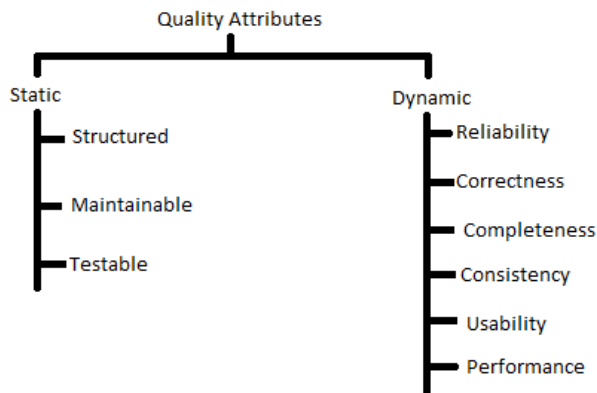
1(a)

(b) Explain the relationship of Human, Errors and Testing with example.

Error is a part of our daily life. Humans make errors in their thoughts, in their actions and in the products that might result from their actions. Errors occur almost everywhere like in observation, in speech, in medical prescription, in surgery, in sport and similarly in software development. An error might be insignificant like a slip of tongue or might lead to a catastrophe like a burst tyre in an airbus. To determine whether there are any errors in our thought, actions and products generated, we resort to the process of testing. The primary goal of testing is to determine if the thoughts, actions and products are as desired, that is they conform to the requirements. Testing of thoughts is usually designed to determine if a concept or method has been understood satisfactorily. Testing of actions is

designed to check if a skill that results in actions has been acquired satisfactorily. Testing of product is designed to check if the product behaves as desired.

2(a) List out quality attributes of software and explain each of them



Static quality attributes: structured, maintainable, testable code as well as the availability of correct and complete documentation.

Dynamic quality attributes: software reliability, correctness, completeness, consistency, usability, and performance

Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of the likelihood of correct function for some “unit” of behavior, which could be a single use or program execution or a period of time.

Correctness will be established via requirement specification and the program text to prove that software is behaving as expected. Though correctness of a program is desirable, it is almost never the objective of testing. To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish. Thus correctness is established via mathematical proofs of programs. While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it. Thus completeness of testing does not necessarily demonstrate that a program is error free.

Completeness refers to the availability of all features listed in the requirements, or in the user manual. Incomplete software is one that does not fully implement all features required.

Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database.

Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing.

Performance refers to the time the application takes to perform a requested task. It is considered as a non-functional requirement. It is specified in terms such as “This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory.”

(b) Define the terminologies organizational metrics, project metrics, process metrics and product metrics with example.

Organizational Metrics: These metrics measure the impact of organizational economics, employee satisfaction, communication and organizational growth factors of the project. Example: Average defect density across all software projects in a company is 1.73 defects/KLOC.

Project Metrics: These are metrics that pertain to Project Quality. They are used to quantify defects, cost, schedule, productivity and estimation of various project resources and deliverables. Example: ratio of number of successful test to number of tests conducted.

Process Metrics: These are metrics that pertain to Process Quality. They are used to measure the efficiency and effectiveness of various processes. Example: measure of defects found in unit test, integration test, system test.

Product Metrics: These are metrics that pertain to Product Quality. They are used to measure cost, quality and the product’s time-to-market. Example: Product complexity metric (cyclomatic

complexity)

3(a)

Differentiate between verification and validation

Verification	Validation
<ul style="list-style-type: none"> Verifying process includes checking documents, design, code and program 	<ul style="list-style-type: none"> It is a dynamic mechanism of testing and validating the actual product
<ul style="list-style-type: none"> It does <i>not</i> involve executing the code 	<ul style="list-style-type: none"> It always involves executing the code
<ul style="list-style-type: none"> Verification uses methods like reviews, walkthroughs, inspections and desk- checking etc. 	<ul style="list-style-type: none"> It uses methods like black box testing ,white box testing and non-functional testing
<ul style="list-style-type: none"> Whether the software conforms to specification is checked 	<ul style="list-style-type: none"> It checks whether software meets the requirements and expectations of customer
<ul style="list-style-type: none"> It finds bugs early in the development cycle 	<ul style="list-style-type: none"> It can find bugs that the verification process can not catch
<ul style="list-style-type: none"> Target is application and software architecture, specification, complete design, high level and data base design etc. 	<ul style="list-style-type: none"> Target is actual product
<ul style="list-style-type: none"> QA team does verification and make sure that the software is as per the requirement in the SRS document. 	<ul style="list-style-type: none"> With the involvement of testing team validation is executed on software code.
<ul style="list-style-type: none"> It comes before validation 	<ul style="list-style-type: none"> It comes after verification

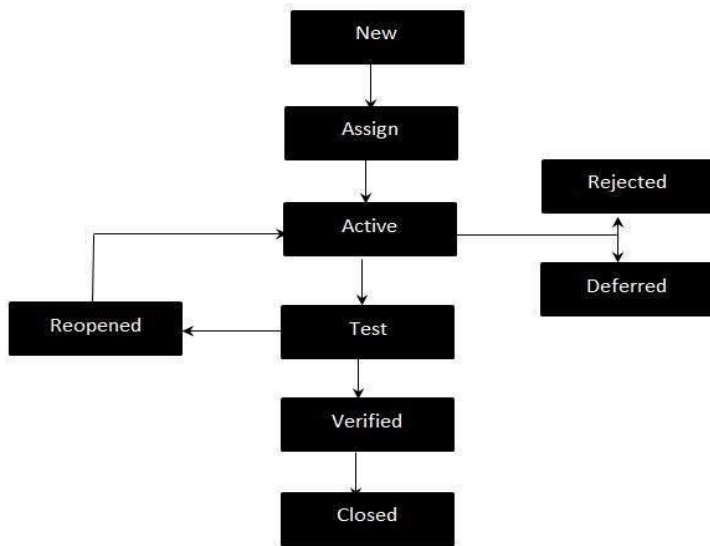
(b)

How is hardware testing different from software testing?

Software Product	Hardware Product
Does not degrade over time	Degrades over time
Fault present in application will remain and no new fault will creep in unless application is changed.	VLST chip might fail over time due to a fault that did not exist at the time chip was manufactured and tested.
Built-in self test meant for hardware product, rarely can be applied to software design and code.	BIST intend to actually test for the correct functioning of a circuit
It only detects faults that were present when the last change was made	Hardware testers generate test based on fault models e.g Stuck-at fault model – one can use a set of input test patterns to test whether a logic gate is functioning as expected

4(a)

Discuss the defect life-cycle and draw an appropriate diagram

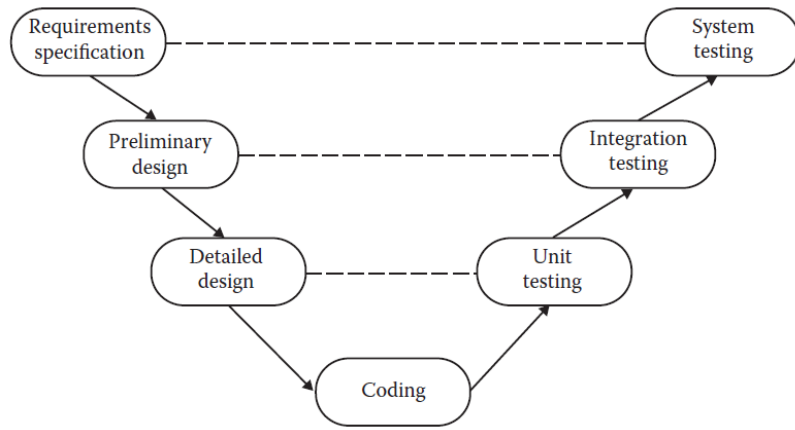


Defect life cycle, also known as **Bug Life cycle** is the journey of a **defect cycle**, which a **defect** goes through during its lifetime. It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.

Defect Life Cycle States:

- **New** - Potential defect that is raised and yet to be validated.
- **Assigned** - Assigned against a development team to address it but not yet resolved.
- **Active** - The Defect is being addressed by the developer and investigation is under progress. At this stage there are two possible outcomes; viz - Deferred or Rejected.
- **Test** - The Defect is fixed and ready for testing.
- **Verified** - The Defect that is retested and the test has been verified by QA.
- **Closed** - The final state of the defect that can be closed after the QA retesting or can be closed if the defect is duplicate or considered as NOT a defect.
- **Reopened** - When the defect is NOT fixed, QA reopens/reactivates the defect.
- **Deferred** - When a defect cannot be addressed in that particular cycle it is deferred to future release.
- **Rejected** - A defect can be rejected for any of the 3 reasons; viz - duplicate defect, NOT a Defect, Non Reproducible.

(b) Discuss how levels of testing are associated with levels of software development. Draw a supporting diagram.



Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing— system, integration, and unit testing. A practical relationship exists between levels of testing versus specification-based and code based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level. This is generally true; however, it is also a likely consequence of the base information produced during the requirements specification, preliminary design, and detailed design phases. The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Chapters 11 through 17 to support code-based testing at the integration and system levels for both traditional and object-oriented software.

5(a) List down the best practices for writing good test case.

Best Practice for writing good Test Case Example.

1. Test Cases need to be simple and transparent:

Create test cases that are as simple as possible. They must be clear and concise as the author of test case may not execute them.

Use assertive language like go to home page, enter data, click on this and so on. This makes the understanding the test steps easy and test execution faster.

2. Create Test Case with End User in Mind

Ultimate goal of any software project is to create test cases that meets customer requirements and is easy to use and operate. A tester must create test cases keeping in mind the end user perspective

3. Avoid test case repetition.

Do not repeat test cases. If a test case is needed for executing some other test case, call the test case by its test case id in the pre-condition column

4. Do not Assume

Do not assume functionality and features of your software application while preparing test case. Stick to the Specification Documents.

5. Ensure 100% Coverage

Make sure you write test cases to check all software requirements mentioned in the specification document. Use Traceability Matrix to ensure no functions/conditions is left untested.

6. Test Cases must be identifiable.

Name the test case id such that they are identified easily while tracking defects or identifying a software requirement at a later stage.

7. Implement Testing Techniques

It's not possible to check every possible condition in your software application. Testing techniques help you select a few test cases with the maximum possibility of finding a defect.

Boundary Value Analysis (BVA): As the name suggests it's the technique that defines the testing of boundaries for specified range of values.

Equivalence Partition (EP): This technique partitions the range into equal parts/groups that tend to have the same behavior.

State Transition Technique: This method is used when software behavior changes from one state to another following particular action.

	<p>Error Guessing Technique: This is guessing/anticipating the error that may arise while testing. This is not a formal method and takes advantages of a tester's experience with the application</p> <p>8. Self cleaning The test case you create must return the test environment to the pre-test state and should not render the test environment unusable. This is especially true for configuration testing.</p> <p>9. Repeatable and self-standing The test case should generate the same results every time no matter who tests it</p> <p>10. Peer Review. After creating test cases, get them reviewed by your colleagues. Your peers can uncover defects in your test case design, which you may easily miss.</p>
(b)	<p>Discuss the six basic principles underlying the analysis and testing techniques.</p> <ul style="list-style-type: none"> • General engineering principles: <ul style="list-style-type: none"> – Partition: divide and conquer – Visibility: making information accessible – Feedback: tuning the development process • Specific A&T principles: <ul style="list-style-type: none"> – Sensitivity: better to fail every time than sometimes – Redundancy: making intentions explicit – Restriction: making the problem easier
6(a)	<p>Explain the difference between error, fault, failure and incident.</p> <p>Error—People make errors. A good synonym is <i>mistake</i>. When people make mistakes while coding, we call these mistakes <i>bugs</i>. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.</p> <p>Fault—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, Unified Modeling Language diagrams, hierarchy charts, and source code. <i>Defect</i> is a good synonym for fault, as is <i>bug</i>. Faults can be elusive. An error of omission results in a fault in which something is missing that should be present in the representation. This suggests a useful refinement; we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.</p> <p>Failure—A failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or perhaps do not execute for a long time? Reviews prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.</p> <p>Incident—When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.</p>
(b)	<p>Discuss defect severity and defect priority with suitable examples.</p> <p>Severity can be of following types:</p> <ul style="list-style-type: none"> • Critical: The defect that results in the termination of the complete system or one or more component of the system and causes extensive corruption of the data. The failed function is unusable and there is no acceptable alternative method to achieve the required results then the severity will be stated as critical. • Major: The defect that results in the termination of the complete system or one or more component of the system and causes extensive corruption of the data. The failed function is unusable but there exists an acceptable alternative method to achieve the required results then the severity will be stated as major. • Moderate: The defect that does not result in the termination, but causes the system to produce incorrect, incomplete or inconsistent results then the severity will be stated as moderate. • Minor: The defect that does not result in the termination and does not damage the usability of the system and the desired results can be easily obtained by working around the defects then the severity is stated as minor. • Cosmetic: The defect that is related to the enhancement of the system where the changes are related to the look and field of the application then the severity is stated as cosmetic. <p>Priority can be of following types:</p> <ul style="list-style-type: none"> • Low: The defect is an irritant which should be repaired, but repair can be deferred until after more serious defect have been fixed. • Medium: The defect should be resolved in the normal course of development activities. It can wait

until a new build or version is created.

- **High:** The defect must be resolved as soon as possible because the defect is affecting the application or the product severely. The system cannot be used until the repair has been done.

Example:

- **High Priority & High Severity:** An error which occurs on the basic functionality of the application and will not allow the user to use the system. (Eg. A site maintaining the student details, on saving record if it, doesn't allow to save the record then this is high priority and high severity bug.)
- **High Priority & Low Severity:** The spelling mistakes that happens on the cover page or heading or title of an application.
- **High Severity & Low Priority:** An error which occurs on the functionality of the application (for which there is no workaround) and will not allow the user to use the system but on click of link which is rarely used by the end user.
- **Low Priority and Low Severity:** Any cosmetic or spelling issues which is within a paragraph or in the report (Not on cover page, heading, title).

7(a) Compare specification based testing and code based testing.

The Differences Between Black Box Testing (or specification based testing) and White Box Testing (or code based testing) are listed below.

Criteria	Black Box Testing	White Box Testing
<i>Definition</i>	Black Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is NOT known to the tester	White Box Testing is a software testing method in which the internal structure/ design/ implementation of the item being tested is known to the tester.
<i>Levels Applicable To</i>	Mainly applicable to higher levels of testing: Acceptance Testing , System Testing	Mainly applicable to lower levels of testing: Unit Testing, Integration Testing
<i>Responsibility</i>	Generally, independent Software Testers	Generally, Software Developers
<i>Programming Knowledge</i>	Not Required	Required
<i>Implementation Knowledge</i>	Not Required	Required
<i>Basis for Test Cases</i>	Requirement Specifications	Detail Design

(b) Explain the Next Date problem and write the pseudo code.

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions (the year range ending in 2012 is arbitrary, and is from the first edition):

- c1. $1 \leq \text{month} \leq 12$
- c2. $1 \leq \text{day} \leq 31$
- c3. $1812 \leq \text{year} \leq 2012$

Pseudo-code:

```

Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Output ("Enter today's date in the form MM DD YYYY")
Input (month, day, year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
If day < 31
Then tomorrowDay = day + 1
Else
tomorrowDay = 1
tomorrowMonth = month + 1
EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months

```

	<pre> If day < 30 Then tomorrowDay = day + 1 Else tomorrowDay = 1 tomorrowMonth = month + 1 EndIf Case 3: month Is 12: 'December If day < 31 Then tomorrowDay = day + 1 Else tomorrowDay = 1 tomorrowMonth = 1 If year = 2012 Then Output ("2012 is over") Else tomorrow.year = year + 1 EndIf Case 4: month is 2: 'February If day < 28 Then tomorrowDay = day + 1 Else If day = 28 Then If ((year is a leap year) Then tomorrowDay = 29 'leap year Else 'not a leap year tomorrowDay = 1 tomorrowMonth = 3 EndIf Else If day = 29 Then If ((year is a leap year) Then tomorrowDay = 1 tomorrowMonth = 3 Else 'not a leap year Output("Cannot have Feb.", day) EndIf EndIf EndIf EndCase Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear) End NextDate </pre>
8 a)	<p>List down the different fault types and give two examples of each.</p> <p>Our definitions of error and fault hinge on the distinction between process and product: process refers to how we do something, and product is the end result of a process. The point at which testing and Software Quality Assurance (SQA) meet is that SQA typically tries to improve the product by improving the process. In that sense, testing is clearly more product oriented. SQA is more concerned with reducing errors endemic in the development process, whereas testing is more concerned with discovering faults in a product. Both disciplines benefit from a clearer definition of types of faults. Faults can be classified in several ways: the development phase in which the corresponding error occurred, the consequences of corresponding failures, difficulty to resolve, risk of no resolution, and so on. My favorite is based on anomaly (fault) occurrence: one time only, intermittent, recurring, or repeatable. For a comprehensive treatment of types of faults, see the IEEE Standard Classification for Software Anomalies (IEEE, 1993). (A software anomaly is defined in that document as "a departure from the expected," which is pretty close to our definition.) The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition.</p> <p>Fault Types:</p> <p>Input/Output Faults Incorrect input not accepted Incorrect input accepted Output Wrong format Wrong result</p>

Cosmetic

Logic Faults

Missing case(s)

Duplicate case(s)

Extreme condition neglected

Wrong operator (e.g., < instead of \leq)

Computation Faults

Incorrect algorithm

Missing computation

Incorrect operand

Incorrect operation

Interface Faults

Incorrect interrupt handling

I/O timing

Call to wrong procedure

Call to nonexistent procedure

Parameter mismatch (type, number)

Incompatible types

Superfluous inclusion

Data Faults

Incorrect initialization

Incorrect storage/access

Wrong flag/index value

Incorrect packing/unpacking

Wrong variable used

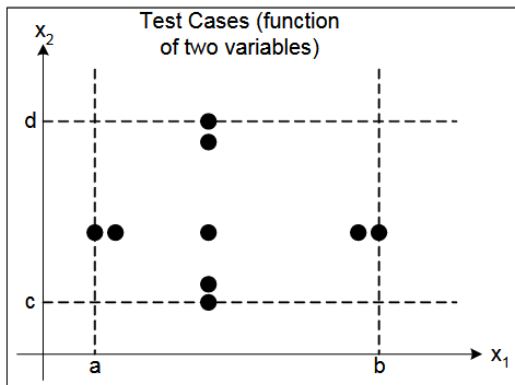
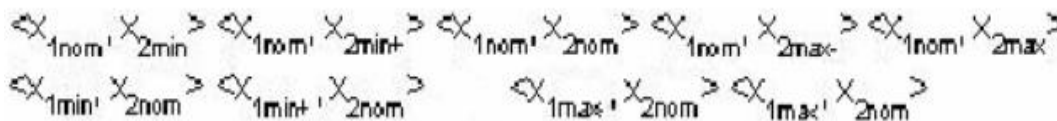
b) Explain BVA test case for two variables functions and limitations of BVA

BVA test case for two variables functions

In the general application of Boundary Value Analysis can be done in a uniform manner.

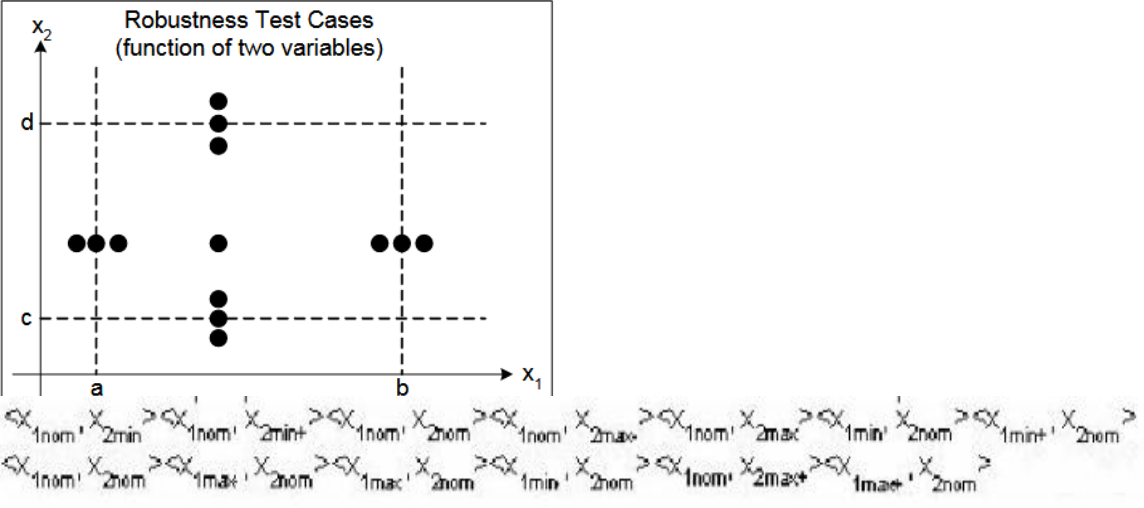
The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

- Min ----- - Minimal
- Min+ ----- - Just above Minimal
- Nom ----- - Average
- Max- ----- - Just below Maximum
- Max ----- - Maximum



Limitations of BVA

Boundary Value Analysis works well when the Program Under Test (PUT) is a “function of several independent variables that represent bounded physical quantities” [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results. For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester’s intuition and common sense shows that we require more emphasis towards the end of February or on leap years. The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables. This lack of intuition or understanding for the variable

	nature means that BVA can be seen as quite rudimentary.
9 a)	<p>Explain Robustness Testing</p> <p>Robustness Testing</p> <p>Robustness testing can be seen as an extension of Boundary Value Analysis. The idea behind Robustness testing is to test for clean and dirty test cases. By clean I mean input variables that lie in the legitimate input range. By dirty I mean using input variables that fall just outside this input domain. In addition to the aforementioned 5 testing values (min, min+, nom, max-, max) we use two more values for each variable (min-, max+), which are designed to fall just outside of the input range. If we adapt our function f to apply to Robustness testing we find the following equation: $f = 6n + 1$</p> <p>Equate this solution by the same reasoning that lead to the standard BVA equation. Each variable now has to assume 6 different values each whilst the other values are assuming their nominal value (hence the $6n$), and there is again one instance whereby all variables assume their nominal value (hence the addition of the constant 1). Robustness testing ensures a sway in interest, where the previous interest lied in the input to the program, the main focus of attention associated with Robustness testing comes in the expected outputs when and input variable has exceeded the given input domain. For example the NextDate problem when we an entry like the 31st June we would expect an error message to the effect of “that date does not exist; please try again”. Robustness testing has the desirable property that it forces attention on exception handling. Although Robustness testing can be somewhat awkward in strongly typed languages it can show up alterations. In Pascal if a value is defined to reside in a certain range then and values that falls outside that range result in the run time errors that would terminate any normal execution. For this reason exception handling mandates Robustness testing.</p> 
b)	<p>Explain the Test Case Generation Strategies</p> <p>The aim of testing is to find bugs in a system or application. Test case generation is the process of building test suites for detecting system errors. A test suite is a group of relevant test cases bundled together. Test case generation is the most important and fundamental process of software testing.</p> <p>There are multiple techniques available for generating test cases:</p> <ul style="list-style-type: none"> • Goal-oriented approach – The purpose of the goal-oriented test case generation approach is to cover a particular section, statement or function. Here the execution path is not important, but testing the goal is the primary objective. • Random approach – The random approach generates test cases based on assumptions of errors and system faults. • Specification-based technique – This model generates test cases based on the formal requirement specifications. • Source-code-based technique – The source-code-based case generation approach follows a control flow path to be tested, and the test cases are generated accordingly. It tests the execution paths. • Sketch-diagram-based approach – This type of case generation approach follows the Unified Modeling Language (UML) diagram to formulate the test cases
10 a)	Represent Triangle problem in a decision table and generate test cases from that

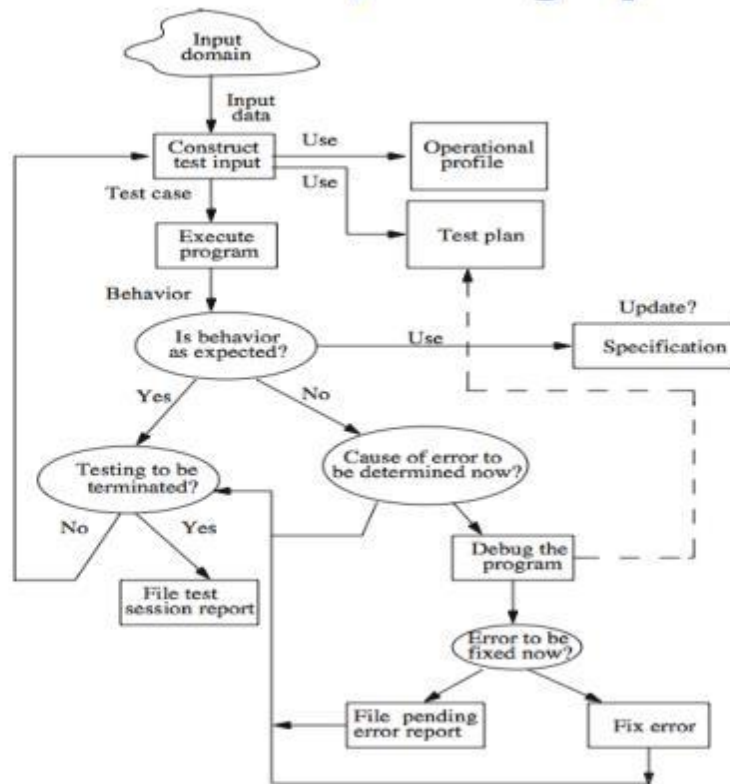
** Write test cases as executed in lab

RULES		R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11
Conditions	C1: $a < b + c$	F	T	T	T	T	T	T	T	T	T	T
	C2: $b < a + c$	-	F	T	T	T	T	T	T	T	T	T
	C3: $c < a + b$	-	-	F	T	T	T	T	T	T	T	T
	C4: $a = b$	-	-	-	T	T	T	T	F	F	F	F
	C5: $a = c$	-	-	-	T	T	F	F	T	T	F	F
	C6: $b = c$	-	-	-	T	F	T	F	T	F	T	F
Actions	1 : Not a triangle	X	X	X								
	2 : Scalene triangle											X
	3 : Isosceles triangle							X		X	X	
	4 : Equilateral triangle				X							
	5 : Impossible					X	X		X			

b) Discuss Test-Debug cycle with the help of a diagram

**Draw the diagram and explain

A test/debug cycle



How is reliability different from correctness? Testing aims at which one of these two and why?

	<p>Reliability and correctness has a very thin line of difference between them. Reliability is a measure of how dependable an application is. If we execute an application for, say, 10 times and 9 times it gives correct output then we can say that the application is reliable. On the contrary, correctness needs the application to be correct all 10 times.</p> <p>Testing aims at reliability. Testing finds out defects in the application, which, when removed, increases the chance of correct output from the application. As a result, the application becomes more reliable to the user. But correctness cannot be achieved even with extensive testing. How much ever an application is tested, the tester can never claim that the application is error free. Thus 100% correctness is never achievable.</p>