CMR INSTITUTE OF TECHNOLOGY	USN	

Improvement Test

Sub:	Software Testing and Practices					Code:			13MCA444
Date:	26.05.2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA

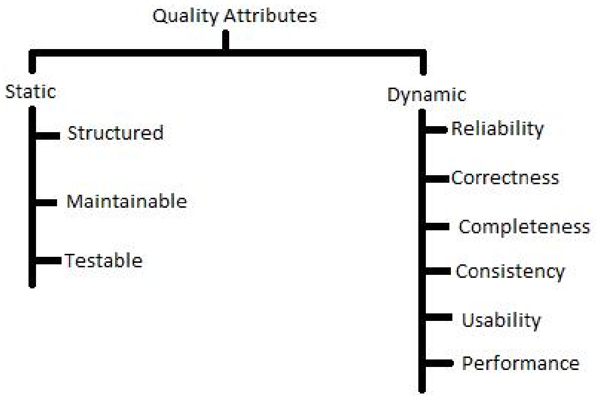
Answer Any **FIVE FULL** Questions

		Marks	OBE	
			CO	RBT
1(a)	List and explain the quality attributes of software.	[5]	CO1	L4
(b)	How is software testing different from hardware testing?	[5]	CO2	L1
2(a)	What do you understand by adequacy criteria? What are its uses?	[6]	CO3	L1
(b)	Discuss the six basic principles of software testing	[4]	CO1	L2
3(a)	Write the pseudo-code for the implementation of the NextDate function	[5]	CO1	L4
(b)	Discuss fault taxonomy and give two examples for each fault type.	[5]	CO4	L2
4(a)	Apply Boundary Value Testing to the triangle problem and list down the test cases	[5]	CO1	L2
(b)	Represent the triangle problem in a decision table format.	[5]	CO2	L1
5(a)	Define the following: i) path testing ii) DD path iii) test coverage metric.	[6]	CO2	L2
(b)	What are metric based testing and slice based testing?	[4]	CO1	L4
6(a)	What is system testing? Differentiate between integration testing and system testing.	[6]	CO4	L2
(b)	Explain the types of integration testing.	[4]	CO1	L4
7(a)	Differentiate between generic and specific scaffolding.	[6]	CO4	L4
(b)	Explain ‘self-checks as oracles’ and ‘capture and replay’.	[4]	CO6	L1

CMR INSTITUTE OF TECHNOLOGY	USN										

Improvement Test									
Sub:	Software Testing and Practices						Code:	13MCA444	
Date:	26.05.2017	Duration:	90 mins	Max Marks:	50	Sem:	IV	Branch:	MCA
Answer Any **FIVE FULL** Questions									

		Marks	OBE CO	OBE RBT
1(a)	List and explain the quality attributes of software.	[5]	CO1	L4



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)	How is software testing different from hardware testing?	[5]	CO2	L1

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

2(a)	What do you understand by adequacy criteria? What are its uses?	[6]	CO3	L1

A software test adequacy criterion is a predicate that defines what properties of a program must be exercised to constitute a thorough test. If the system passes an adequate suite of test cases, then it must be correct (or dependable). But determining an adequate suite of test case is hypothetical.

Use of adequacy criteria:

- Specify a software testing requirement

  -Determine test cases to satisfy requirement
- Determine observations that should be made during testing
- Control the cost of testing

  -Avoid redundant and unnecessary tests
- Help assess software dependability

Build confidence in the integrity estimate

(b)	Discuss the six basic principles of software testing The six basic principles of software testing are: • General engineering principles: – Partition: divide and conquer – Visibility: making information accessible – Feedback: tuning the development process • Specific A&T principles: – Sensitivity: better to fail every time than sometimes – Redundancy: making intentions explicit – Restriction: making the problem easier **Partition**: Hardware testing and verification problems can be handled by suitably partitioning the input space **Visibility**: The ability to measure progress or status against goals. X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?" **Feedback**: The ability to measure progress or status against goals X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?" **Sensitivity**: A test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria). Run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine **Redundancy**: Redundant checks can increase the capabilities of catching specific faults early or more efficiently. e.g, Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently. **Restriction**: Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems	[4]	CO1	L2

3(a)	Write the pseudo-code for the implementation of the NextDate function
	
	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
	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
		[5]	CO1	L4

```
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
EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
End NextDate
```

(b) Discuss fault taxonomy and give two examples for each fault type. [5] CO4 L2

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. The IEEE standard defines a detailed anomaly resolution process built around four phases (another life cycle): recognition, investigation, action, and disposition.

Fault Types:

Input/Output Faults
ICorrect input not accepted
Incorrect input accepted
Output Wrong format
Wrong result
Cosmetic
Logic Faults
Missing case(s)
Duplicate case(s)
Extreme condition neglected
Wrong operator (e.g., < instead of ≤)
1.3 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

4(a) Apply Boundary Value Testing to the triangle problem and list down the test cases [5] CO1 L2

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper
bound. For each side, the test values are {1, 2, 100, 199, 200}. Robust boundary value test cases will add {0, 201}. Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted. Further, there is no test case for scalene triangles. The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table below only lists the first 25 worst-case boundary value test cases for the triangle problem. You can picture them as a plane slice through the cube (actually it is a rectangular parallelepiped) in which a = 1 and the other two variables take on their full set of cross-product values.

Case	a	b	c	Expected Output
1	100	100	1	Isosceles
2	100	100	2	Isosceles
3	100	100	100	Equilateral
4	100	100	199	Isosceles
5	100	100	200	Not a triangle
6	100	1	100	Isosceles
7	100	2	100	Isosceles
8	100	100	100	Equilateral
9	100	199	100	Isosceles
10	100	200	100	Not a triangle
11	1	100	100	Isosceles
12	2	100	100	Isosceles
13	100	100	100	Equilateral
14	199	100	100	Isosceles
15	200	100	100	Not a triangle

(b) Represent the triangle problem in a decision table format. [5] CO2 L1

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
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

5(a) Define the following: i) path testing ii) DD path iii) test coverage metric. [6] CO2 L2

Path Testing: **path testing**, or structured **testing**, is a white box method for designing **test** cases. The method analyzes the control flow graph of a program to find a set of linearly independent **paths** of execution.

Given a program written in an imperative programming language, its *program graph* is a directed graph in which nodes are statement fragments, and edges represent flow of control. (A complete statement is a "default" statement fragment.)If *i* and *j* are nodes in the program graph, an edge exists from node *i* to node *j* if and only if the statement fragment corresponding to node *j* can be executed immediately after the statement fragment corresponding to node *i*.

(b) What are metric based testing and slice based testing? [4] CO1 L4

Metric Based Testing: In software **testing**, **Metric** is a quantitative measure of the degree to which a system, system component, or process possesses a given attribute. In other

	words, **metrics** helps estimating the progress, quality and health of a software **testing** effort. Metric based testing is efficient because: - Metrics help the Project Management/Team to effectively manage the various activities across the SDLC and achieve a single view, understanding of the progress of the deliverables and also to quickly analyze and identify the impact of any change across the deliverables. - Metrics assist early detection and correction of errors or changes in the requirements gathered. - Multiple metrics are needed for comprehensive evaluation of requirements, testing and their trace-ability to do a Gap analysis, Change Impact analysis, compliance verification of code, regression test selection, and requirements verification and validation for the project team to achieve the best of best deliverables. - Metric collection, with a combination of tool based approach and other methods, is cheaper, faster and more reliable. Slice Based Testing: A program slice is a set of program statements that contributes to or affects the value of a variable at some point in a program Backward slices S(v, n): refer to statement fragments that contribute to the value of v at statement n • Statement n is a Use node of variable v, Use (v, n) Forward slices S(v, n): refer to all the program statements that are affected by the value o v and statement n • Refers to the predicate uses and computation uses of the variable v			
6(a)	What is system testing? Differentiate between integration testing and system testing. **System Testing** (ST) is a black box **testing** technique performed to evaluate the complete **system** the **system's** compliance against specified requirements. In **System testing**, the functionalities of the **system** are tested from an end-to-end perspective. ... It includes both functional and Non-Functional **testing**.	[6]	CO4	L2

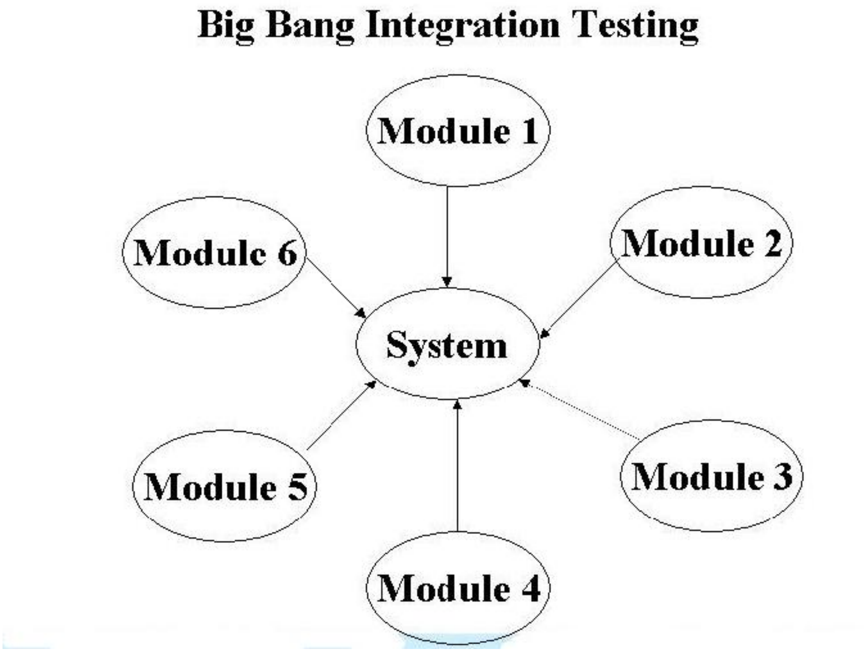
System Testing	Integration Testing
1. Testing the completed product to check if it meets the specification requirements.	1. Testing the collection and interface modules to check whether they give the expected result
2. Both functional and non-functional testing are covered like sanity, usability, performance, stress an load .	2.Only Functional testing is performed to check whether the two modules when combined give correct outcome.
3. It is a high level testing performed after integration testing	3. It is a low level testing performed after unit testing
4. It is a black box testing technique so no knowledge of internal structure or code is required	4. It is both black box and white box testing approach so it requires the knowledge of the two modules and the interface
5. It is performed by test engineers only	5. Integration testing is performed by developers as well test engineers
6. Here the testing is performed on the system as a whole including all the external interfaces, so any defect found in it is regarded as defect of whole system	6. Here the testing is performed on interface between individual module thus any defect found is only for individual modules and not the entire system
7. In System Testing the test cases are developed to simulate real life scenarios	7. Here the test cases are developed to simulate the interaction between the two module

	8. The System testing covers many different testing types like sanity, usability, maintenance, regression, retesting and performance	8. Integration testing techniques includes big bang approach, top bottom , bottom to top and sandwich approach.		

(b) Explain the types of integration testing. [4] CO1 L4

1. Big Bang integration testing:

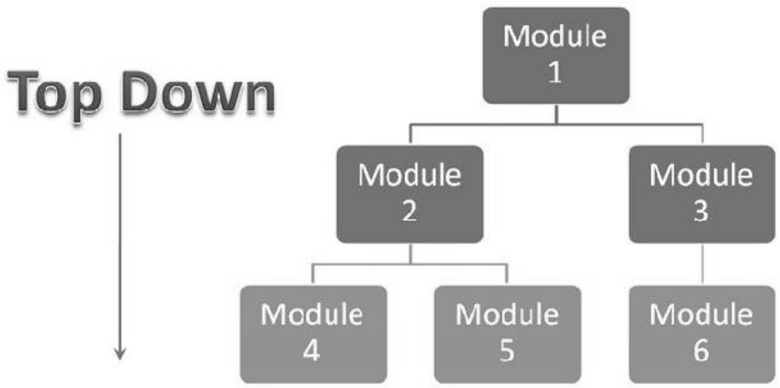
In Big Bang integration testing all components or modules are integrated simultaneously, after which everything is tested as a whole. As per the below image all the modules from 'Module 1' to 'Module 6' are integrated simultaneously then the testing is carried out.



Advantage: Big Bang testing has the advantage that everything is finished before integration testing starts.

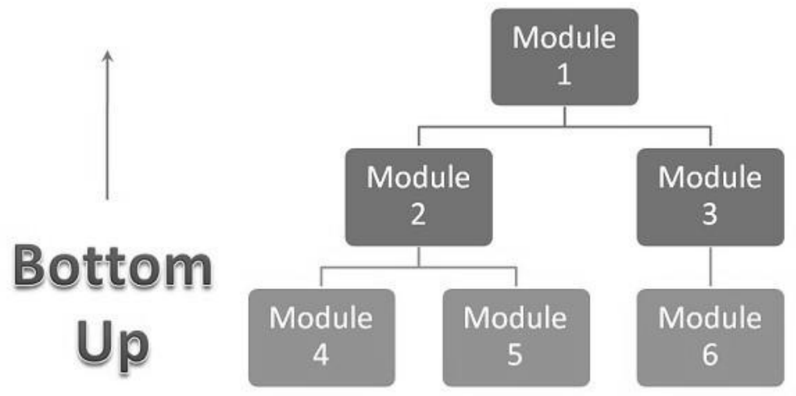
Disadvantage: The major disadvantage is that in general it is time consuming and difficult to trace the cause of failures because of this late integration.

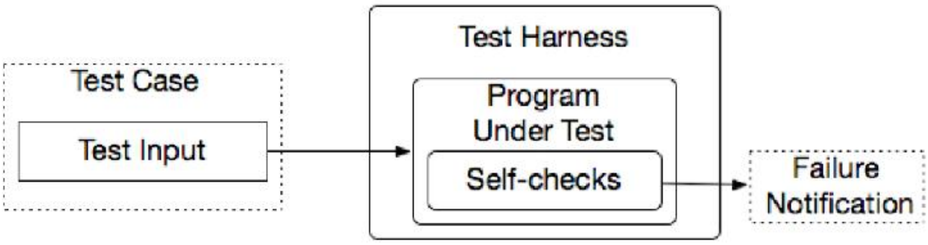
2. **Top-down integration testing:** Testing takes place from top to bottom, following the control flow or architectural structure (e.g. starting from the GUI or main menu). Components or systems are substituted by stubs. Below is the diagram of 'Top down Approach':



Advantages of Top-Down approach:

- The tested product is very consistent because the integration testing is basically

	performed in an environment that almost similar to that of reality • Stubs can be written with lesser time because when compared to the drivers then Stubs are simpler to author. **Disadvantages of Top-Down approach:** • Basic functionality is tested at the end of cycle 3. **Bottom-up integration testing:** Testing takes place from the bottom of the control flow upwards. Components or systems are substituted by drivers. Below is the image of 'Bottom up approach':  **Advantage of Bottom-Up approach:** • In this approach development and testing can be done together so that the product or application will be efficient and as per the customer specifications. **Disadvantages of Bottom-Up approach:** • We can catch the Key interface defects at the end of cycle • It is required to create the test drivers for modules at all levels except the top control **Incremental testing:** • Another extreme is that all programmers are integrated one by one, and a test is carried out after each step. • The incremental approach has the advantage that the defects are found early in a smaller assembly when it is relatively easy to detect the cause. • A disadvantage is that it can be time-consuming since stubs and drivers have to be developed and used in the test. • Within incremental integration testing a range of possibilities exist, partly depending on the system architecture. **Functional incremental:** Integration and testing takes place on the basis of the functions and functionalities, as documented in the functional specification.			
7(a)	Differentiate between generic and specific scaffolding. How general should scaffolding be? To answer We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit) ... or further factor out some common support code, to drive a large number of test cases from data... or further, generate the data automatically from a more abstract model (e.g., network traffic model) Fully generic scaffolding may suffice for small numbers of handwritten test cases The simplest form of scaffolding is a driver program that runs a single,specific test case. It is worthwhile to write more generic test drivers that essentially interpret test case	[6]	CO4	L4

	specifications. A large suite of automatically generated test cases and a smaller set of handwritten test cases can share the same underlying generic test scaffolding Scaffolding to replace portions of the system is somewhat more demanding and again both generic and application-specific approaches are possible A simplest stub– ***mock*** – can be generated automatically by analysis of the source code The balance of quality, scope and cost for a substantial piece of scaffolding software can be used in several projects The balance is altered in favour of simplicity and quick construction for the many small pieces of scaffolding that are typically produced during development to support unit and small-scale integration testing A question of costs and reuse – Just as for other kinds of software			
(b)	Explain ‘self-checks as oracles’ and ‘capture and replay’. **SELF-CHECKS AS ORACLES** An oracle can also be written as self checks -Often possible to judge correctness without predicting results. Typically these self checks are in the form of assertions, but designed to be checked during execution. It is generally considered good design practice to make assertions and self checks to be free of side effects on program state. Self checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specification rather than all program behaviour. Devising the program assertions that correspond in a natural way to specifications poses two main challenges: Bridging the gap between concrete execution values and abstractions used in specification Dealing in a reasonable way with quantification over collection of values Structural invariants are good candidates for self checks implemented as assertions They pertain directly to the concrete data structure implementation It is sometimes straight-forward to translate quantification in a specification statement into iteration in a program assertion A run time assertion system must manage ghost variables They must retain “before” values They must ensure that they have no side effects outside assertion checking *Advantages:* -Usable with large, automatically generated test suites. *Limits:* -often it is only a partial check. -recognizes many or most failures, but not all.  **CAPTURE AND REPLAY** Sometimes it is difficult to either devise a precise description of expected behaviour or adequately characterize correct behaviour for effective self checks. Example: even if we separate testing program functionally from GUI, some testing of the GUI is required. If one cannot completely avoid human involvement test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error. The principle is simple: The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated testing. The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it, before it is invalidated by some change to the program. Mapping from concrete state to an abstract model of interacting sequences is some time possible but is generally quite limited.	[4]	CO6	L1