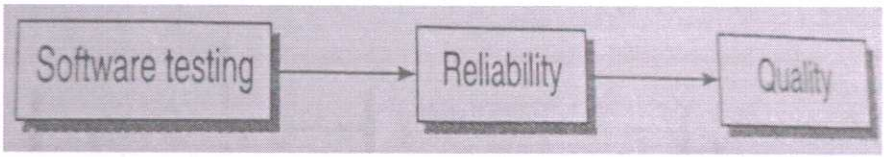
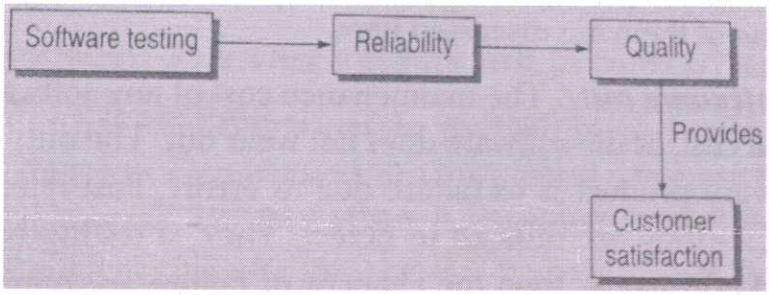
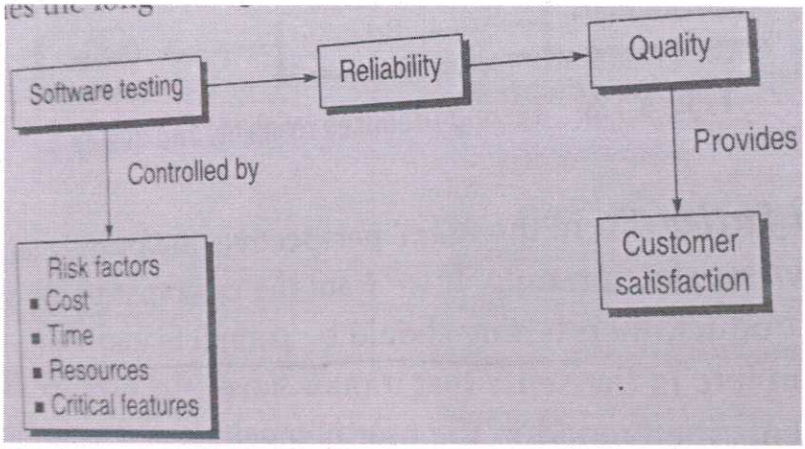
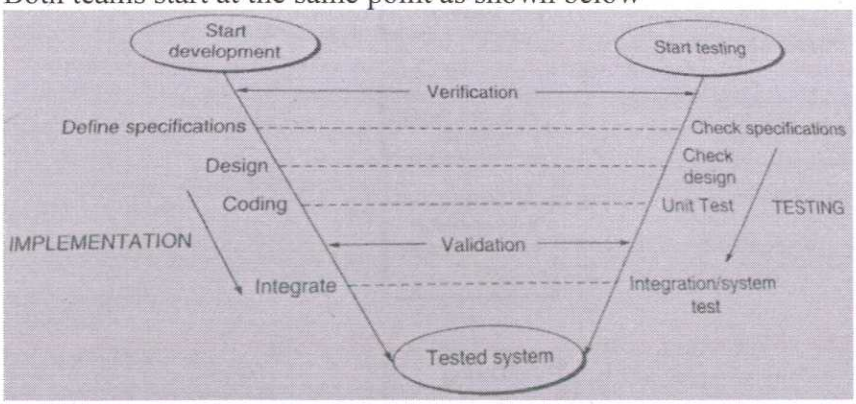
SCHEME OF EVALUATION

(Scoring Indicator)

Revision :2015		Course Code: 6136		
Course Title: Software Testing				
Qst. No	Scoring Indicator	Split up score	Sub total	Total
I (1).	PART A Software testing is a process that detects important bugs with the objective of having better quality software.	2	2	10
I (2)	➢ Software structure or software behaviour can be modeled as a Finite State Machine (FSM). ➢ An FSM is a behavioural model whose outcome depends upon both previous and current inputs. ➢ FSM models can be used as a tool for functional testing.	2	2	
I (3)	Boundary Value Analysis, Equivalence class Partitioning, State table-based testing, Decision Table-based testing, Cause-effect graphing technique, Error guessing (write any two)	2*1	2	
I (4)	Reduction of testing effort, Reduces the tester's involvement in executing tests, facilitates regression testing, avoids human mistakes, Reduces overall cost of the software, Simulated testing, Internal testing, Test enablers, Test case design. (write any four)	4*0.5	2	
I (5)	If output of testing shows some failure, it is required to find the bugs that caused the failure and remove the errors present in the software. This is called debugging.	2	2	

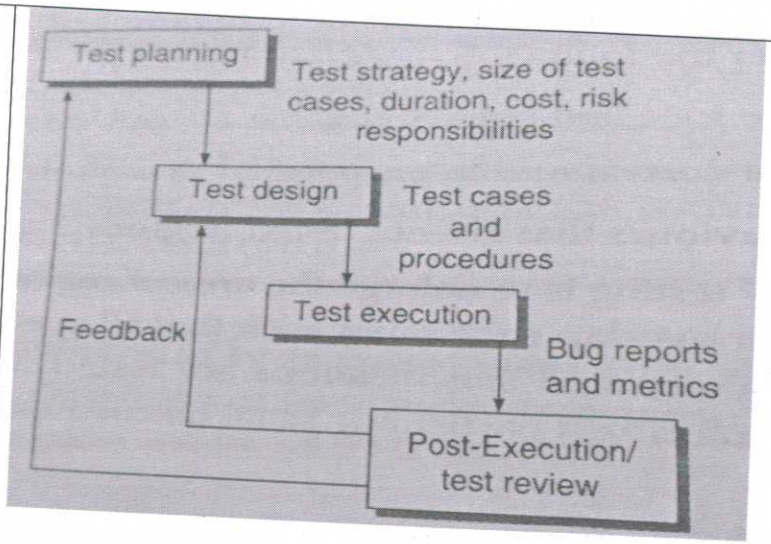
I	PART B			6
1	**Long-term goals** These goals affect the product quality in the long run. ***Quality:*** Software testing ensures product quality, which is the primary need of users. ***Reliability*** is the major factor to achieve quality. Reliability is a matter of confidence that the product will not fail. Quality also depends on various factors like *correctness, integrity, efficiency* etc. The confidence in reliability increases quality as shown below:  ***Customer Satisfaction*** It is the prime concern of testing. All specified and unspecified requirements must be tested for customer satisfaction i.e. testing must be complete. A complete testing process achieves reliability, reliability enhances quality, and quality in turn increases customer satisfaction as shown below:  ***Risk Management*** Risk is the probability that undesirable events will occur in a system which will prevent the organisation from successfully implementing their business. Software testing may act as a control, which can help in eliminating or minimizing risks as shown below  Testers has to evaluate business risks such as *cost, time, resources and other critical features* of the system being developed and make those risks as a	4*1.5	6	

	basis for testing choices. Testers can categorize risks as *high-risk, moderate-risk, low-risk* and can consider this analysis as a basis for testing activities.			
II (2)	**Testing Life cycle model - (V-Testing model)** • Verification and validation are the building blocks of a testing process • V&V can be modelled as Testing Life Cycle Model for best understanding • In V-testing concept, the development team attempts to implement the software, the testing team concurrently starts checking the software. • When the project starts, the team that is developing the system begins the system development process and the team that is conducting the system test begins planning the system test process. • Both teams start at the same point as shown below  • On the left arm of the V, development cycle is progressing and on the right arm, the	3 3	6	6

	corresponding testing stages are moving. In early stages of SDLC, testing comprises of more verification activities and later stages comprises of validation activities. • V&V process involves ○Verification of every step of SDLC ○ Validation of the verified system at the end			
II (3)	**Mutation Testing Process** • Construct the mutants of a test program • Add test cases to the mutation system and check the output of the original program on each test case to see if it is correct • If the output of original program is incorrect, a fault has been found and the program must be modified and restart the process. • If the output is correct, that test case is executed against each live mutant. • If the output of the mutant differs from that of the original program on the same test case, the mutant is assumed killed. • After executing all test cases, the remaining live mutants belongs to one of two categories. Mutants which are functionally equivalent to the original program so that they cannot be killed. Killable mutants, but test cases are not sufficient to kill them. • Mutation score of a set of test data is the percentage of non-equivalent mutants killed by the test data. If score is 100%, then test data is called mutation-adequate	6	6	6
II (4)	**Progressive Vs. Regressive Testing** In progressive testing, the testing process progresses from verification to validation towards the release of product. • All testing techniques black-box testing, white-box testing, static testing, validation testing etc. are progressive in nature. A system under test (SUT) is said to regress if: ○ A modified component fails ○ A new component causes failure in the unchanged components by generating side effects. • Therefore , all systems will have three versions : ○ ***Baseline version***: The version of a system that has passed a test suite. ○ ***Delta Version*** : A changed version that has not passed a regression test ○ ***Delta build*** : An executable configuration of the SUT that contains all the delta and baseline components. • Most test cases begin as progressive test cases and eventually become regression test cases • Regression testing can be defined as *the software maintenance task performed on a modified program to in still confidence that*	6	6	6

	changes are correct and have not adversely affected the unchanged portions of the program.			
II (5)	**GUIDELINES FOR AUTOMATED TESTING** • Testing tools can never replace the analytical skills required to conduct testing and manual testing. • So automation should be adopted carefully after deciding which tool and how many tools are required, how much resources are required including cost of the tool and time spent on training • Listed below are few guideline to be followed while planning automation in testing : ***1. Consider building a tool instead of buying one, if possible*** • If the requirement is small and enough resources are available, go for building a tool instead of buying. • Whether to buy or build a tool requires management commitment including budget and resource approvals. ***2. Test the tool on an application prototype*** • While purchasing a tool, it is important to verify that the tool works properly with the system being developed. So it is suggested that the development team can build a system prototype for evaluating the testing tool. ***3. Not all the tests should be automated*** • Automated testing is an enhancement of manual testing . • Some tests are impossible to automate such as verifying a print out. It has to be done manually. ***4. Select the tools according to organizational need*** • Do not buy the tools just for their popularity or to compete with other organizations. • Focus on the needs of the organization and know the resources (budget, schedule) before choosing the automation tool. ***5. Use proven test-script development techniques*** • Automation can be effective if proven techniques are used to produce efficient, maintainable, and reusable test scripts • Following are some hints : ***i.*** Read the data values from either spreadsheets or tool-provided data pools, rather than being hard coded into the test script because this prevent test cases from being reused. ***ii.*** Use modular script development for maintainability and readability ***iii.*** Build a shared script library of reusable functions which is usable by all test engineers ***iv.*** All test scripts should be stored in a version control tool ***6. Automate the regression tests whenever feasible*** • Since regression tests need a lot of time, automate the regression test cases whenever possible. Thus, testing time can be reduced to a greater extent.	6*1	6	6
II (6)	**C Unit Test System (CUT or CuTest)** • CUT is a simple unit testing system. It's different from other unit test packages in that it follows the KISS (Keep it simple, Stupid) principle. It's designed for C testing, not designed to emulate SUnit (**SUnit** is a unit testing framework for the programming language Smalltalk). • CUT is OS independent	6	6	6

	It is a unit testing library for the C language. • It can be used to do Extreme Programming and Test-First Development in the C language. • It's a fun and cute library that will make your programming fun and productive. • **Benefits** i. ***Lower Defects***. The tests ensure that your code keeps working as you make small changes in it. ii. ***Faster Debugging***. The tests tell you which subroutine is broken. You avoid spending hours trying to figure out what's broken. iii. ***Development Speed.*** You trust your old code and can keep adding to it without worrying about bad interactions. If there is a bad interaction the tests will catch it. iv. ***Permanent Bug Fixes***. If every time a bug is reported you write a quick test, you will guarantee that the bug never reappears again. v. ***Fun.*** As your bug count drops you will begin to enjoy programming like you've never done before. Running the tests every few minutes and seeing them pass feels good. • **Features** i. ***Small.*** Consists of a single .c and .h file. ii. ***Easy to Deploy.*** Just drop the two files into your source tree. iii. ***Highly Portable***. Works with all major compilers on Windows (Microsoft, Borland), Linux, Unix, PalmOS. iv. ***Open Source***. You can extend it to add more functionality. The source can be invaluable if you are trying to trace a test failure. v. ***Cuteness.*** Of all the testing frameworks CuTest has the cutest name			
II (7)	***Kernel debugger***: • It is for dealing with problems with an OS kernel on its own or for interactions between OS-dependent applications and the OS. • A kernel debugger might be a stub implementing low-level operations, with a full blown debugger such as gdb, running on another machine, sending commands to the stub over a serial line or a network connection, or it might provide a command line that can be used directly on the machine being debugged. • The Windows NT family includes a kernel debugger named **KD**, which can act as a local debugger with limited capabilities	3*2	6	6
III	**Software Testing Life Cycle (STLC)** Software testing process is divided into a well-defined sequence of steps known as software testing life cycle (STLC). • STLC allows testers to involve at early stages of development which has significant benefit in the project cost and schedule • STLC helps management in measuring specific milestones. • STLC consists of ***four phases*** as shown below : ***Test Planning , Test Design, Test Execution & Post-Execution/ test review***	15	15	15



Test Planning

o Following are the activities during test planning

Define the test strategy

Estimate the number of test cases, their duration and cost

Plan the resources like manpower, tools, documents required.

Identify areas of risks

Define test completion criteria

Identify techniques and tools needed for various test cases

Identify reporting procedures, bug classification, databases for testing etc.

o Output of planning is the test plan document

o After analyzing issues of planning, the following activities are performed:

Develop test case format

Develop test case plan for every SDLC phase

Identify test cases to be automated

Prioritize test cases according to importance and criticality

Define areas of stress and performance testing

Plan the test cycles required for regression testing

Test design

It is a well-planned process of designing test cases.

o Test design includes the following critical activities:

Determining the test objectives and their prioritization:

• Based on SRS and design documentations, a team of experts compile a list of objectives. This objective list is then prioritized based on scope and risk.

Preparing list of items to be tested :

• Here, each objective is converted to list of items to be tested

Mapping items to test cases :

• Identify the test cases for each item identified. Prepare a matrix to identify which test case will be covered by which item. The matrix will help in :

o Identifying the major test scenarios

o Identifying and reducing the redundant test cases

o Identifying the absence of a test case for a particular objective

• A prior analysis of the program at functional and structural level is needed for designing test cases

The main rule for designing test cases is " Cover all features, but do not make too many test cases"

Attributes for a good test case are :

o A good test case is designed to test the most important functions which are critical due to high-risk requirements

o A good test case should have designed with high probability of finding an error.

Test cases should not overlap or redundant

o A good test case should be designed with a modular approach so that there is no complexity and it can be reused and recombined to execute various functional paths

o A good test case avoids masking of errors and duplication of test creation efforts

o A successful test case should discover errors that are not yet discovered

Selection of test case design techniques :

There are two broad categories of test case design techniques:

o *Black-box testing* : Generates test cases without knowing the internal working of a system

o *White-box testing*

o Techniques are selected such that there is more coverage and the system detects more bugs

Creating test cases and test data

Create test cases based on objectives identified

Test cases mention objectives, inputs required and expected outputs

Setting up the test environment and supporting tools :

Details like hardware configurations, testers, interfaces, operating systems, and manuals must be specified during this phase

Creating test procedure specification :

This is a description of how test case will be run. It is in the form a sequenced steps that can be used by a tester at the time of execution of test cases

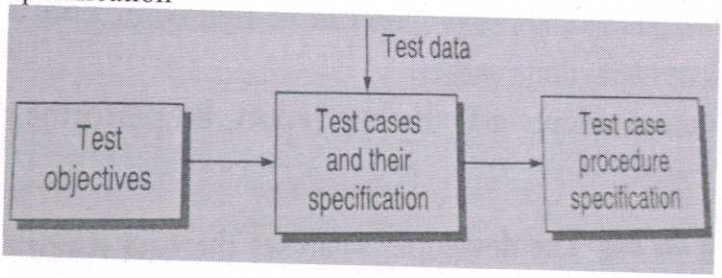
o The hierarchy for test design phase includes :

Developing test objectives

Identifying test cases and creating their specifications

Developing test case procedure specifications

o Details of test design phase are documented in the test design specification



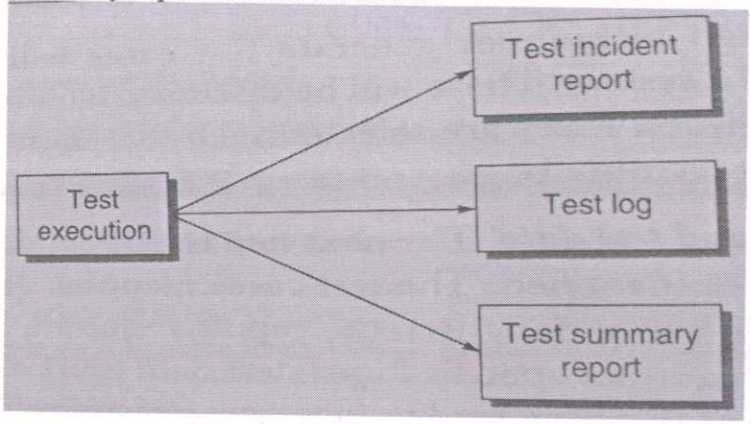
Test Execution

o In this phase all test cases are executed including verification and validation

o Verification test cases are started at the end of each phase of SDLC

o Validation test cases are started after the completion of a module

o Test results are documented in the test incident reports, test logs, and test summary reports as shown below



Responsibilities at various levels of execution of the test cases are shown below:

Test Execution Level	Person Responsible
Unit	Developer of the module
Integration	Testers and Developers
System	Testers, Developers, end-users
Acceptance	Testers, end-users

Post-Execution / Test Review

o This phase is to analyse bug-related issues and get feedback so that maximum number of bugs can be removed

o As soon as the developer gets the bug report, he performs the following activities

Understanding the bug

Reproducing the bug : to see the failure that exists

Analyse the nature and cause of the bug :

Here the developer performs debugging (Identify failure symptoms, track back to error location, correct the errors and report the corrections to testing team)

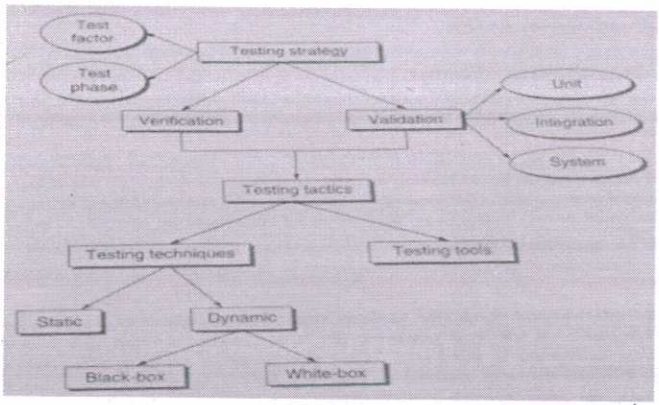
Finally the following activities can be done:

o *Reliability analysis*: to ensure that the software meets the predefined reliability goals or not

o *Coverage analysis*: as an alternate criteria to stop testing

o *Overall defect analysis*: to identify risk areas and to improve quality.

IV	**Software Testing Methodology** Software testing methodology is the organization of software testing by means of which the ***test strategy*** and ***test tactics*** are achieved.	15	15	15



Software Testing Strategy

o Testing strategy is the planning of the whole testing process into a well-planned series of steps

o The components of testing strategy are :

Test Factors

• Test factors are risk factors or issues related to the system under development

• Risk factors should be selected ranked

• Testing should reduce these test factors to a prescribed level

Test Phase

• It refers to the phases of SDLC where testing will be performed. Strategy may be different for different SDLC models.

Test Strategy Matrix

o A test strategy matrix can be used while developing testing strategy.

o This matrix is prepared using test factors and test phases as shown below:

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test

Steps to prepare test strategy matrix

Select and rank test factors : These are rows of the matrix

Identify system development phases : These are the columns of the matrix

Identify risks associated with the system under development : The purpose is to identify the risks to be addressed under each test phase. Risks may include events, actions, or circumstances such as late budget, approvals, delayed arrival of equipments, late availability of software etc. Risks may prevent implementation of software according to planned schedule

o Example for creating a test strategy : Test strategy for designing a new operating system

Test Factors	Test Phase					
	Requirements	Design	Code	Unit test	Integration test	System test
Portability	Is portability feature mentioned in specifications according to different hardware?					Is system testing performed on MIPS and INTEL platforms?
Service Level	Is time frame for booting mentioned?	Is time frame incorporated in design of the module?				

Steps :

Select and rank test factors : Portability & Service level

• ***Identify the test phases*** : Below matrix shows the affected test phases according to test factors

• ***Identify risks associated with each test factor and its corresponding phase*** : Risks are expressed as questions in the matrix.

• ***Plan the strategy for every risk identified*** : After identifying the risks, plan strategy to tackle them.

Development of Test Strategy

o The rule for development of test strategy is that " testing begins from the smallest unit and progresses to enlarge"

o Test strategy begins at the component level and finish at the integration of the entire system

o Testing process is a combination of Verification & Validation. These are the two basic strategies for testing any type of software

o ***Verification:***

Check the software with its specification at every development phase such that any defect can be detected at an early stage of testing and will not be allowed to propagate further

Verification can be applied to all stages of SDLC

Verification is checking the work at intermediate stages to confirm that the project is moving in the right direction, towards the set goal.

o ***Validation***

As the system development progresses down to completion, the scope of verification decreases.

The validation process starts replacing verification at the later stages of SDLC

After building individual modules of a system, the following stages need to be tested which is known as validation testing :

• The module as a whole

• Integration of modules

• The system built after integration

• **Testing Life cycle model - (V-Testing model)**

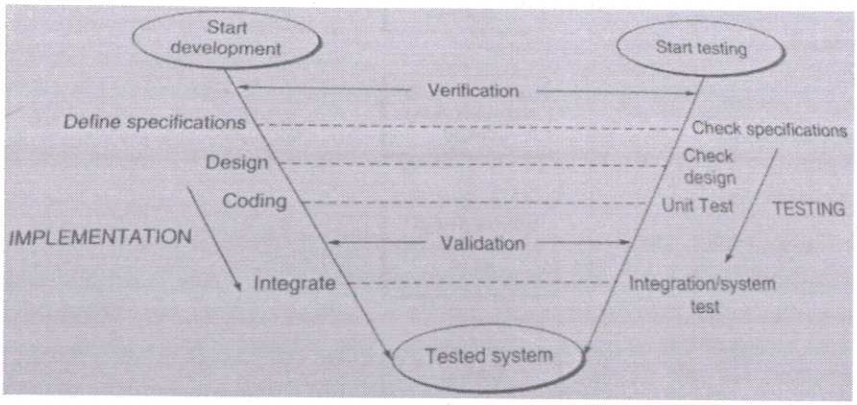
• Verification and validation are the building blocks of a testing process

• V&V can be modelled as Testing Life Cycle Model for best understanding

• In V-testing concept, the development team attempts to implement the software, the testing team concurrently starts checking the software

When the project starts, the team that is developing the system begins the system development process and the team that is conducting the system

test begins planning the system test process.
• Both teams start at the same point as shown in fig.



On the left arm of the V, development cycle is progressing and on the right arm, the corresponding testing stages are moving.
• In early stages of SDLC, testing comprises of more verification activities and later stages comprises of validation activities
• V&V process involves
o Verification of every step of SDLC
o Validation of the verified system at the end
• **Validation Activities**
o Validation has the following three levels of testing
Unit Testing : Validation performed on the smallest unit or module of the system. It confirms the behaviour of a single module according to its functional specification
Integration Testing : It is a validation testing which combines all unit tested modules and performs a test on their aggregation. Mainly the interfacing between modules are tested
System Testing : Testing the entire integrated system against the requirement specification. The purpose is to test the validity for specific users and environments
• **Testing Tactics**
Complete and exhaustive testing is not possible
o Our effort should be for effective testing
o The main objectives of effective test case design are:
Coverage of testing domain
Detection of maximum number of bugs
o The technique used to design effective test case is called Software Testing Technique
o These techniques can be categorized into two parts : *Static testing & Dynamic Testing*
Static Testing
• It is a technique for assessing the structural characteristics of source code, design specifications etc.
• It is called static because we never execute the code in this technique
Dynamic Testing
• In this technique, the code is run on a number of inputs provided by the user and the corresponding results are checked.
• Dynamic testing is further divided in to two parts : ***black-box testing & white-box testing***

	• ***Black-box testing***: o This technique takes care of inputs given to a system and the output received after processing from the system o It is not concerned with the operations happening in the system o It is also known as "*functional testing*" o It is used for *system testing* under validation • ***White-box testing*** o Here the system is not a black-box. o Every design feature and its corresponding code is checked logically with every possible path execution o It takes care of the structural paths instead of just outputs. o It is also known as "*structural testing*" o It is used for *unit testing* under verification • **Testing Tools** o Testing tools provide the option to automate the selected testing technique with the help of tools. o A tool is a resource to perform test process Tester firs understand the testing techniques and then choose the tool that can be used with each technique. **Considerations in developing Testing Methodologies** o *Determine Project Risks* o *Determine the type of development project* o *Identify test activities according to SDLC phase* o *Build the test plan*			

V (a)	**Logic Coverage Criteria**
	Test cases for structural testing are designed based on logic of the program. Every element of the logic must be covered by the test cases.
	• Basic forms of logic coverage are :
	o *Statement coverage*
	o *Decision or Branch coverage*
	o *Condition coverage*
	o *Decision/Condition coverage*
	o *Multiple Condition Coverage*
	1)Statement coverage
	In this form, test cases are designed to cover (or execute) all statements in the module under testing.
	o For example, Consider the following sample code,
	
	scanf("%d", &x);
	2) scanf("%d", &y);
	3) while (x!=y)
	4) {
	5) If (x > y)
	6) x = x – y;
	7) else
	8) y = y – x;
	9) }
	10) printf("x=",x);
	11) printf("y=",y);
	
	The following test cases are enough for statement coverage. These test cases will cover every statement in the code segment. But these test cases are not enough to test all conditions and paths exists in the code and errors might exist undetected.
	Test Case 1 | **x > y**
	Test Case 2 | **x < y**
	o **Test Case 3 : x = y,** is also needed to execute the path of skipping the loop and directly jump to printf statement.
	o Statement coverage is necessary but not a sufficient criteria.
	
	2)Decision or Branch Criteria
	• In this form, test cases are designed to include all possible outcomes of each decision at least once.
	• In other words, each branch path must be executed at least once
	• Example:
	o In the sample code mentioned above, the following test cases will consider all possible outcomes by **while** and **if** conditions
	Test Case 1 | x == y
	Test Case 2 | x != y
	Test Case 3 | x < y
	Test Case 4 | x > y
	
	3)Condition Coverage
	In this form, each condition in a decision will take all possible outcomes at least once
	• Example:
		5*2	10	15

	o Consider the **while** condition :
	while (I <= 5 && (J < Count))
	o Here the following test cases are enough to cover both conditions
	Test Case 1 | I < 5 (*True*) , J < Count (*True*)
	Test Case 2 | I > 5 (*False*), J > Count (*False*)
	
	4)Decision / Condition coverage
	• Condition coverage does not ensure the coverage of decisions.
	• Consider the following decision
	if (A && B) , test cases for condition coverage are :
	Test Case 1 | A is True , B is False
	Test Case 2 | A is False, B is True
	But these two test cases will not execute the **THEN** clause of **if** statement
	• So, decision / condition coverage considers all possible outcomes of each condition at least once and all possible outcomes of each decision at least once
	• There for test cases can be designed as follows to cover both conditions and decision
	Test Case 1 | A is True , B is True
	Test Case 2 | A is False, B is False
	
	5)Multiple condition coverage
	• Multiple condition coverage requires that , we should design sufficient test cases such that all possible combinations of condition outcomes in each decision and all point of entry are invoked at least once
	• Example :
	For the decision *if (A && B)* ,the following test cases can be designed :
	Test Case 1 | A is True , B is True
	Test Case 2 | A is True, B is False
	Test Case 3 | A is False, B is True
	Test Case 4 | A is False, B is False			

V (b)	**Describe about Validation Testing** • By validation, we mean that the module or software that has been prepared till now is in conformance with the requirements initially specified in the SRS. • As in V&V activity diagram, every validation testing focuses on a particular SDLC. • There is a one-to-one correspondence between development and testing processes • The advantage of this structure of validation testing is that it avoids redundant testing and prevents one from overlooking large classes of error • Validation is achieved through a series of black-box testing which demonstrates conformity with requirements • A test plan decides the classes of tests to be conducted and a test procedure defines the test cases. • Both the plan & procedure are designed to ensure : o All functional requirements are satisfied o All behavioral characteristic are achieved o All performance requirements are attained o Documentation is correct and human-engineered	5	5	

VI	**Boundary Value Analysis (BVA)** BVA is a black-box testing technique that uncovers bugs at the boundary of input values • Boundary means maximum or minimum value taken by the input domain. For example, if A is an integer variable between 10 and 250, then boundary checking can be on 10 (9, 10, 11) and on 250 (249, 250, 251) • BVA offers **four** methods to design test cases : o ***Boundary Value checking (BVC)*** o ***Robustness Testing method*** o ***Worst-case testing method*** o ***Robust worst-case testing method***	3 4*3	15	15

• ***Method1 : Boundary Value check (BVC)***

o Here, test cases are designed by holding one variable at its extreme value and other variables at their nominal values.

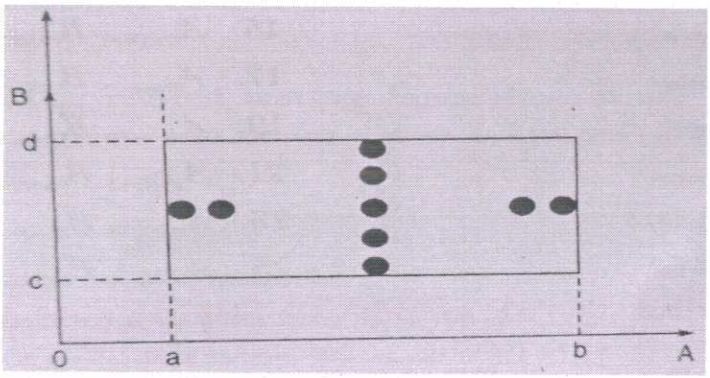
o The extreme values can be selected at :

Minimum value (Min)
Value just above minimum value (Min+)
Maximum value (Max)
Value just below maximum values (Max-)

o Example: Consider two variables A & B , the test cases that can be designed are:

1. Anom, Bmin	2. Anom, Bmin+
3. Anom, Bmax	4. Anom, Bmax-
5. Amin, Bnom	6. Amin+, Bnom
7.Amax, Bnom	8. Amax-, Bnom
9. Anom, Bnom	

In general , **4n+1** test cases can be designed for '**n**' variables in a module.



Method 2 : Robustness Testing method

o Here, BVC is extended by adding two more boundary values :

A value just greater than Max (Max+)
A value just less than Min (Min-)

o Eg:- The following test cases can also be added with 9 test cases mentioned in the above example

10. Amax+, Bnom	11. Amin-, Bnom
12. Anom, Bmax+	13. Anom, Bmin

o In general, **6n+1** test cases can be designed with '**n**' variables in a module

Method 3: Worst-case Testing

o In this method, BVC is extended by considering more than on variable at its extreme values or boundary.

o For example, we can add the following test cases along with the 9 test cases designed in BVC :

10. Amin, Bmin	11. Amin, Bmin+	12. Amin, Bmax	13. Amin, Bmax-
14. Amin+, Bmin	15. Amin+, Bmin+`	16. Amin+,Bmax	17. Amin+,Bmax-
18. Amax, Bmin	19. Amax, Bmin+	20. Amax, Bmax	21. Amax,Bmax-
22. Amax-, Bmin	23. Amax-, Bmin+	24. Amax-, Bmax	25. Amax,Bmax

o In general, **5n** test cases can be designed with worst-case testing for n-input variables

• ***Method 4 : Robust Worst-case Testing***

o This is the combined form of Robustness testing and worst-case testing. i.e. extend BVC
by considering extreme values (min, min-, min+, max, max+ & max-) of more than one variables.

o *(Write all 49 test cases by yourself , for the previous two variable example)*

o In general for **n**-input variables in a module, **7n** test cases can be designed with robust worst-case testing

Example :- A program reads an integer number within the range [1-100] and determines whether it is prime or not. Design test cases usinf BVC, worst-case testing method.

Solution :-

a) Test cases using BVC

• Since there is only one variable, No. of test cases = 4n+1 = 4x1+1 – 5

• Min, Max and Nominal values can be :

o Min Value = 1

Min+ Value =2

o Max Value =100

o Max- Value =99

o Nom Value = 50 – 55

• 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	53	Prime Number

b) Test Cases using Robustness testing = 6n+1 = 6

• The set of boundary values are :

o Min value = 1

o Min- value = 0

o Min + value = 2

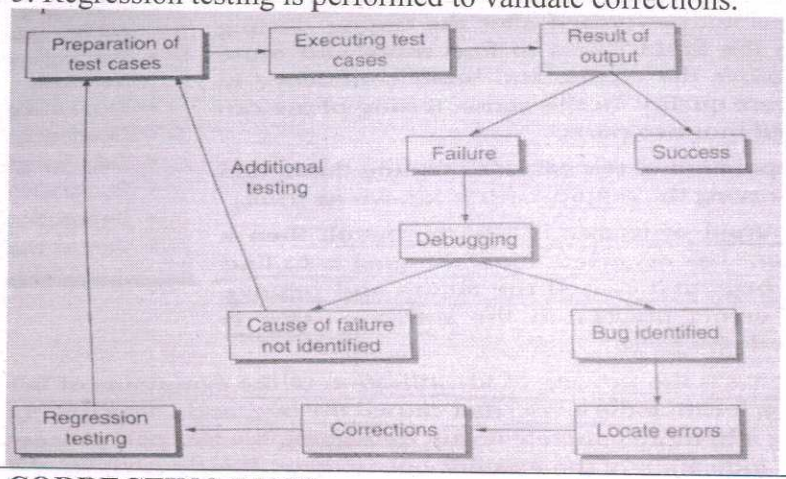
	o Max Value = 100 o Max+ Value = 101 o Max- Value = 99 o Nominal Value = 50-55 • Test cases can be designed as shown below: c) Test case using worst-case testing 5n = 51 = 5 (In this example , it is same as BVC)			

Test Case ID	Integer variable	Expected Output
1	0	Invalid input
2	1	Not a prime number
3	2	Prime number
4	100	Not a prime number
5	99	Not a prime number
6	101	Invalid input
7	53	Prime number

VII	**COMMERCIAL TESTING TOOLS (write any five)** **1)WinRnner** • It is a tool used for performing functional / regression testing • It is a product of Mercury Interactive • It automatically creates the test scripts by recording the user interactions on GUI of the software. These scripts can be run repeatedly whenever needed without any manual intervention. • It support the Test Script Language (TSL, a C-like language) using which test scripts can be modified. • If any problem occurred during automatic testing, there is provision to bring the application to a new state. It has a default interleaving of one second between statement execution. If some activities take more time, then test cases automatically synchronizes by waiting for the completion of current operation. **2) SilkTest** • It is a product from Segue Software • It is used for regression/functional testing • It supports the object-oriented scripting language 4Test • SilkTest has a provision for customized in-built recovery system which helps in continuing the automated testing even if there is some failure in between **3) LoadRunner** • It is a product of Mercury Interactive • This tool is used for performance and load testing of a system • This tool is helpful for client/server applications of various parameters with their actual load like response time, the number of users etc. • The major benefit of this tool is that it creates virtual users on a single machine and tests the system on various parameters. Thus performance and load testing is done with minimum infrastructure. **4) Jmeter** • This is an open source software tool used for performance and load testing • The Apache JMeter application is open source software, a 100% pure Java application designed to load test functional behavior and measure performance. It was originally designed for testing Web Applications but has since expanded to other test functions.	5*3	15	15

	• Apache JMeter may be used to test performance both on static and dynamic resources, Web dynamic applications. • It can be used to simulate a heavy load on a server, group of servers, network or object to test its strength or to analyze overall performance under different load types. • Apache JMeter features include: Ability to load and performance test many different applications / server / protocol types: Web - HTTP, HTTPS (Java, NodeJS, PHP, ASP.NET, …) SOAP / REST Webservices FTP Database via JDBC LDAP Message-oriented middleware (MOM) via JMS Mail - SMTP(S), POP3(S) and IMAP(S) Native commands or shell scripts TCP Java Objects • Full featured Test IDE that allows fast Test Plan recording (from Browsers or native applications), building and debugging • Command-line mode to load test from any Java compatible OS (Linux, Windows, Mac OSX, …) • A complete and ready to present dynamic HTML report • Easy correlation through ability to extract data from most popular response formats, HTML, JSON , XML or any textual format • Complete portability and 100% Java purity. • Full multi-threading framework allows concurrent sampling by many threads and simultaneous sampling of different functions by separate thread groups. • Caching and offline analysis/replaying of test results. • Data analysis and visualization plugins allow great extensibility as well as personalization. • Functions can be used to provide dynamic input to a test or provide data manipulation. **5) TestDirector** • TestDirector is a test management tool. • It is a software product from Mercury Interactive • It is a web-based tool with the advantage of managing the testing if two teams are at different locations • It manages the test process with four phases : Specifying requirements Planning tests Running tests Tracking defects • Tests are planned as per the requirements. Once the plan is ready, test cases are executed. • Defect tracking can be done in any phase of test process. During defect-tracking, new bug can be reported assign responsibility to someone for bug repair Assign priority to the bug			

	Bug repair status can be analysed • This tool can also be integrated with LoadRunner or WinRunner **6) IBM Rational SQA Robot** • It is another powerful tool for functional/Regression testing • Synchronization of test cases with a default delay of 20 second is also available			
VIII (a)	**Challenges or Issues in Testing Web-based software** 1. **Diversity and Complexity** : • Web applications interact with many components that run on diverse hardware and software platforms • They are written in diverse languages and they are based on different programming approaches such as procedural, OO, and hybrid languages such as JSP • The client side includes browsers, HTML, embedded scripting languages and applets • The server side includes CGI, JSPs, Java Servlets and .NET technologies • They all interact with diverse back-end engines found on web server or other servers 2. **Dynamic Environment**: • The key aspect of web applications is its dynamic nature caused by uncertainty in the program behavior, changes in application requirements, rapidly evolving web technology etc. • It is difficult to statically determine the control flow of applications because the control flow is highly dependent on user inputs, trends of user behavior over time or user location • This dynamic nature creates challenges for analysis, testing, and maintenance of these systems **3. Very short development time** • Clients of web-based systems impose very short development time, compared to other software projects **4. Continuous evolution** • Demand for more functionality and capacity after the system has been designed and deployed i.e. scalability issues **5. Compatibility and Interoperability** • Web applications are often affected by factors that may cause incompatibility and interoperability issues • The problem of incompatibility may exists on both client as well as server side. • Server components can be distributed to different operating systems • Various versions of browsers running under a variety of operating systems on the client side. • Graphics and other objects on a website have to be tested on multiple browsers • There are different versions of HTML Web applications are diverse, complex, dynamic and face the problems of incompatibility and interoperability.	5*2	10	15
VIII (b)	**ISSUES ON OBJECTED ORIENTED TESTING (write any five)** ***1. Basic unit of testing*** • Class is the natural unit for test case design • Other units are aggregation of classes such as class clusters and application systems	5*1	5	

	• There are different types of classes : application-specific vs general-purpose, abstract classes, parameterized or template classes . This demands different test requirements • Testing an object (class instance) verify a class. When objects are used in applications, the entire system must be tested as a whole . ***2. Implications of inheritance*** • The inherited features require retesting because these features are now in a new context of usage ***3. Polymorphism*** • It is difficult to find out all polymorphic bindings which requires a separate test to validate all of them ***4. White-box testing*** • Conventional white-box testing cannot be applied on testing a class. ***5. Black-box testing*** • Retesting of inherited features require examination of class structure. This limits the use of black-box testing technique for OO software ***6. Integration strategies*** • There is no clear hierarchy of methods and classes. So conventional methods of incremental integration testing cannot be adopted. Therefore it is required to devise new integration methods.			
IX (a)	**Debugging Process** • The debugging process is explained in following steps: 1. Check the result of testing process output. If the actual output and expected output are same, then the results are successful. Otherwise, the output is a failure which needs to be analyzed. 2. Debugging is performed for the analysis of failure. Here, we identify the cause of problem and correct it. If the symptoms of failure are not sufficient to find the bug, additional testing is performed to get more clues to analyze the cause of failure. 3. If the failure symptoms are sufficient to provide clues about bug, then the cause of failure is identified. The bug is traced to find the actual location of error. 4. Once the actual location of error is traced, the bug is removed with corrections. 5. Regression testing is performed to validate corrections.  	5*1 4	9	15
IX (b)	**CORRECTING BUGS** • This is the second phase of debugging process. • The design of the software should not be affected by correcting the bug.	6	6	

	• Consider the following points before correcting the errors : o Evaluate coupling of the logic and data structure where corrections are to be made. Correction of highly coupled module can introduce other bugs. Low-coupled module is easy to debug. o Recognize the influence of corrections on other modules and then plan regression test cases to perform regression testing o Perform regression testing with every correction in the software to ensure that the corrections have not created bugs in other parts.			
X (b)	**Debugging Techniques** **1. Debugging with memory dump** • In this technique, a printout of all registers and relevant memory locations is obtained and studied. The relevant data and program stored in these registers and memory locations are observed for any bug in the program • This method is used as the last option • Following are some drawbacks of this method: i. It is difficult to establish the correspondence between storage locations and the variables in a source program. ii. Memory dump might contain massive amount of irrelevant data iii. It is limited to static state of the program as it shows state of the program at only one instant of time. **2. Debugging with watch points** • Programs final output may not give sufficient clues about the bug. But, intermediate execution points may provide sufficient cause of the problem. These points are known as watch points. • Debugging with watch points can be implemented with the following methods: ***i. Output statements*** • In this method, output or print statements are inserted at various watch points to check the status of variables. • The program is compiled and executed with these output statements. Execution of output statements may give some clues to find the bug. • **Drawbacks** of this method are : a. It may require many changes in the program which may mask an error or introduce new errors in the program. b. After bug analysis, we may forget to remove the added statements which may cause misinterpretations in the result ***ii. Breakpoint Execution*** • This is an advanced form of the watch point used with an automated debugger program. • Break point is a watch point inserted at various places in the program. Program is executed up to the breakpoint inserted so that we can examine the status at that point. Afterwards, the program will resume and will be executed further until next breakpoint. • Breakpoints allows controlled execution of program with necessary stops for intermediate examinations. • Advantages of breakpoints over output statements : ***a.*** There is no need to compile the program after inserting breakpoints ***b.*** Removing the breakpoints after their requirement is easy ***c.*** The status of variable or particular conditions can be observed at the breakpoint. No need to wait for complete execution of the program as in	3*5	15	15

	the case of output statements. • Breakpoints can be categorized as follows : a. ***Unconditional breakpoint*** :- It is a simple breakpoint without any condition to be evaluated. Its execution stops the execution of the program b. ***Conditional breakpoint*** :- At his breakpoint, one Boolean expression is evaluated and if true, the breakpoint will cause a stop; otherwise execution will continue. c. ***Temporary breakpoint*** : This breakpoint is used only once in the program. After executing once, the temporary breakpoint is automatically removed. d. ***Internal breakpoint*** : These breakpoints are added by the debugger itself for its own purposes and these are invisible to the user ***iii. Single stepping*** • Single stepping allows the user to watch the status or condition of variables after execution of every single instruction • Single stepping is implemented with the help of internal breakpoints. ***Step-into*** • It means execution proceeds into any function in the current source statement and stops at the first executable source line in that function ***iv. Step-over*** • It is also called skip, instead of step. It treats function calls as an atomic operation without single stepping through the called function. **3. BackTracking** • This is a logical approach for debugging a program. Following are the steps for backtracking process : i. Observe the symptoms of the failure at the output and reach the site where the symptoms can be uncovered. For example, one value is not getting displayed on the output due to non-working of function X. ii. From that site, trace the source code starting backwards and move to the highest level of abstraction of design. The bug will be found in this path. iii. Slowly isolate the module where bug resides using data flow diagrams of the module. iv. Logical backtracking in the isolated module will lead to the actual bug and error can thus be removed. • The person doing debugging using backtracking method must have knowledge regarding design of the system in order to understand the logical flow of the program. • This is more effective as compared to other methods to locate errors quickly.			