SUBJECT NAME: SOFTWARE TESTING METHODOLOGIES

FACULTY NAME: Dr.A.VIJENDAR

MODULE I

What is testing?

Testing is the process of exercising or evaluating a system or system components by manual or automated means to verify that it satisfies specified requirements.

The Purpose of Testing Testing consumes at least half of the time and work required to produce a functional program.

- MYTH: Good programmers write code without bugs. (It's wrong!!!)
- History says that even well written programs still have 1-3 bugs per hundred statements

Productivity and Quality in Software:

- In production of consumer goods and other products, every manufacturing stage is subjected to quality control and testing from component to final stage.
- If flaws are discovered at any stage, the product is either discarded or cycled back for rework and correction.
- Productivity is measured by the sum of the costs of the material, the rework, and the discarded components, and the cost of quality assurance and testing. o There is a tradeoff between quality assurance costs and manufacturing costs: If sufficient time is not spent in quality assurance, the reject rate will be high and so will be the net cost. If inspection is good and all errors are caught as they occur, inspection costs will dominate, and again the net cost will suffer.

Phases in a tester's mental life can be categorized into the following 5 phases:

1. Phase 0: (Until 1956: Debugging Oriented) There is no difference between testing and debugging. Phase 0 thinking was the norm in early days of software development till testing emerged as a discipline.

2. Phase 1: (1957-1978: Demonstration Oriented) the purpose of testing here is to show that software works. Highlighted during the late 1970s. This failed because the probability of showing that software works 'decreases' as testing increases. I.e. the more you test, the more likely you will find a bug.

3. Phase 2: (1979-1982: Destruction Oriented) the purpose of testing is to show that software doesn't work. This also failed because the software will never get released as you will find one bug or the other. Also, a bug corrected may also lead to another bug.

4. Phase 3: (1983-1987: Evaluation Oriented) the purpose of testing is not to prove anything but to reduce the perceived risk of not working to an acceptable value (Statistical Quality Control). Notion is that testing does improve the product to the extent that testing catches bugs and to the extent that those bugs are fixed. The product is released when the confidence on that product is high enough. (Note: This is applied to large software products with millions of code and years of use.)

5. Phase 4: (1988-2000: Prevention Oriented) Testability is the factor considered here. One reason is to reduce the labor of testing. Other reason is to check the testable and non-testable code. Testable code has fewer bugs than the code that's hard to test. Identifying the testing techniques to test the code is the main key here.

Dichotomies:

Testing Versus Debugging:

Many people consider both as same. Purpose of testing is to show that a program has bugs. The purpose of testing is to find the error or misconception that led to the program's failure and to design and implement the program changes that correct the error. Debugging usually

follows testing, but they differ as to goals, methods and most important psychology. The below tab le shows few important differences between testing and debugging.

A Model for Testing:

It includes three models: A model of the environment, a model of the program and a model of the expected bugs.

Environment:

- A Program's environment is the hardware and software required to make it run. For online systems, the environment may include communication lines, other systems, terminals and operators.
- The environment also includes all programs that interact with and are used to create the program under test - such as OS, linkage editor, loader, compiler, utility routines.
- Because the hardware and firmware are stable, it is not smart to blame the environment for bugs.

Program:

- Most programs are too complicated to understand in detail.
- The concept of the program is to be simplified in order to test it.
- If simple model of the program doesn't explain the unexpected behavior, we may have to modify that model to include more facts and details. And if that fails, we may have to modify the program.

Bugs:

- Bugs are more insidious (deceiving but harmful) than ever we expect them to be.
- An unexpected test result may lead us to change our notion of what a bug is and our model of bugs.
- Some optimistic notions that many programmers or testers have about bugs are usually unable to test effectively and unable to justify the dirty tests most programs need.

CONSEQUENCES OF BUGS:

Importance of bugs: The importance of bugs depends on frequency, correction cost,installation cost, and consequences.

1. Frequency: How often does that kind of bug occur? Pay more attention to the more frequent bug types.

2. Correction Cost: What does it cost to correct the bug after it is found? The cost is the sum of 2 factors: (1) the cost of discovery (2) the cost of correction. These costs go up dramatically later in the development cycle when the bug is discovered. Correction cost also depends on system size.

3. Installation Cost: Installation cost depends on the number of installations: small for a single user program but more for distributed systems. Fixing one bug and distributing the fix could exceed the entire system's development cost.

4. Consequences: What are the consequences of the bug? Bug consequences can range from mild to catastrophic.

The consequences of a bug can be measure in terms of human rather than machine. Some consequences of a bug on a scale of one to ten are:

1 Mild: The symptoms of the bug offend us aesthetically (gently); a misspelled output or a misaligned printout.

2 Moderate: Outputs are misleading or redundant. The bug impacts the system's performance.

3 Annoying: The system's behavior because of the bug is dehumanizing. E.g. Names are truncated or arbitrarily modified.

4 Disturbing: It refuses to handle legitimate (authorized / legal) transactions. The ATM won’t give you money. My credit card is declared invalid.

5 Serious: It loses track of its transactions. Not just the transaction itself but the fact that the transaction occurred. Accountability is lost.

6 Very Serious: The bug causes the system to do the wrong transactions. Instead of losing your paycheck, the system credits it to another account or converts deposits to withdrawals.

7 Extreme: The problems aren't limited to a few users or to few transaction types. They are frequent and arbitrary instead of sporadic infrequent) or for unusual cases.

8 Intolerable: Long term unrecoverable corruption of the database occurs and the corruption is not easily discovered. Serious consideration is given to shutting the system down.

9 Catastrophic: The decision to shut down is taken out of our hands because the system fails.

10 Infectious: What can be worse than a failed system? One that corrupt other systems even though it does not fall in itself ; that erodes the social physical environment.

TAXONOMY OF BUGS:

There is no universally correct way categorize bugs. The taxonomy is not rigid.

 A given bug can be put into one or another category depending on its history and the Programmer's state of mind.

 The major categories are: (1) Requirements, Features and Functionality Bugs (2) Structural Bugs (3) Data Bugs (4) Coding Bugs (5) Interface, Integration and System Bugs (6) Test and (7)Test Design Bugs.

MODULE II

BASICS OF PATH TESTING:

- Path Testing is the name given to a family of test techniques based on judiciously selecting a set of test paths through the program.
- If the set of paths are properly chosen then we have achieved some measure of test thoroughness. For example, pick enough paths to assure that every source statement has been executed at least once.
- Path testing techniques are the oldest of all structural test techniques.

- Path testing is most applicable to new software for unit testing. It is a structural technique.
- It requires complete knowledge of the program's structure.
- It is most often used by programmers to unit test their own code.
- The effectiveness of path testing rapidly deteriorates as the size of the software aggregate under test increases.

Control Flow Graphs:

- The control flow graph is a graphical representation of a program's control structure. It uses the elements named process blocks, decisions, and junctions.
- The flow graph is similar to the earlier flowchart, with which it is not to be confused.

Flow Graph Elements: A flow graph contains four different types of elements. (1) Process Block (2) Decisions (3) Junctions (4) Case Statements.

1. Process Block: A process block is a sequence of program statements uninterrupted by either decisions or junctions.
   It is a sequence of statements such that if any one of statement of the block is executed, then all statement thereof are executed. ♣ Formally, a process block is a piece of straight line code of one statement or hundreds of statements
2. Decisions: A decision is a program point at which the control flow can diverge.
   Machine language conditional branch and conditional skip instructions are examples of decisions.
3. Case Statements: A case statement is a multi-way branch or decisions.
   Examples of case statement are a jump table in assembly language, and the PASCAL case statement.

4. Junctions: A junction is a point in the program where the control flow can merge.

TRANSACTION FLOW GRAPHS:

- Transaction flows are introduced as a representation of a system's processing.
- The methods that were applied to control flow graphs are then used for functional testing.

- Transaction flows and transaction flow testing are to the independent system tester what control flows are path testing are to the programmer.
- The transaction flow graph is to create a behavioral model of the program that leads to functional testing.
- The transaction flowgraph is a model of the structure of the system's behavior (functionality).

TRANSACTION FLOW TESTING TECHNIQUES:

- Complicated systems that process a lot of different, complicated transactions should have explicit representations of the transactions flows, or the equivalent.
- Transaction flows are like control flow graphs, and consequently we should expect to have them in increasing levels of detail.
- The system's design documentation should contain an overview section that details the main transaction flows.
- Detailed transaction flows are a mandatory pre requisite to the rational design of a system's functional test.

DATA FLOW TESTING:

- Data flow testing is the name given to a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of data objects.
- For example, pick enough paths to assure that every data object has been initialized prior to use or that all defined objects have been used for something.
- Motivation: It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

MODULE III

A DOMAIN IS A SET:

- An input domain is a set.
- If the source language supports set definitions (E.g. PASCAL set types and C enumerated types) less testing is needed because the compiler does much of it for us.
- Domain testing does not work well with arbitrary discrete sets of data objects.
- Domain for a loop-free program corresponds to a set of numbers defined over the input vector.

Domain Testing is a Software Testing process in which the application is tested by giving a minimum number of inputs and evaluating its appropriate outputs. The primary goal of Domain testing is to check whether the software application accepts inputs within the acceptable range and delivers required output.

It is a Functional Testing technique in which the output of a system is tested with a minimal number of inputs to ensure that the system does not accept invalid and out of range input values. It is one of the most important White Box Testing methods. It also verifies that the system should not accept inputs, conditions and indices outside the specified or valid range.Domain testing differs for each specific domain so you need to have domain specific knowledge in order to test a software system.

While domain testing, you need to consider following things,

1. What domain are we testing?
2. How to group the values into classes?
3. Which values of the classes to be tested?
4. How to determine the result?

LINEAR AND NON LINEAR BOUNDARIES: Nice domain boundaries are defined by linear inequalities or equations. o The impact on testing stems from the fact that it takes only two points to determine a straight line and three points to determine a plane and in general n+ 1 point to

determine an n-dimensional hyper plane. In practice more than 99.99% of all boundary predicates are either linear or can be linearized by simple variable transformations.

COMPLETE BOUNDARIES: Nice domain boundaries are complete in that they span the number space from plus to minus infinity in all dimensions. Such boundaries can come about because the path that hypothetically corresponds to them is unachievable, because inputs are constrained in such a way that such values can't exist, because of compound predicates that define a single boundary, or because redundant predicates convert such boundary values into a null set. The advantage of complete boundaries is that one set of tests is needed to confirm the boundary no matter how many domains it bounds. If the boundary is chopped up and has holes in it, then every segment of that boundary must be tested for every domain it bounds.

Random testing: It is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail. In case of absence of specifications the exceptions of the language are used which means if an exception arises during test execution then it means there is a fault in the program, it is also used as way to avoid biased testing.

Random Testing Characteristics:

- Random testing is performed where the defects are NOT identified in regular intervals.
- Random input is used to test the system's reliability and performance.
- Saves time and effort than actual test efforts.
- Other Testing methods Cannot be used to.

Random Testing Steps:

- Random Inputs are identified to be evaluated against the system.
- Test Inputs are selected independently from test domain.
- Tests are Executed using those random inputs.
- Record the results and compare against the expected outcomes.

- Reproduce/Replicate the issue and raise defects, fix and retest

Types of Software Testing:

Functional Testing types include: Unit Testing

- **Integration Testing**
- **System Testing**
- **Sanity Testing**
- **Smoke Testing**
- **Interface Testing**
- **Regression Testing**
- **Beta/Acceptance Testing**

Non-functional Testing types include: Performance Testing

- **Load Testing**
- **Stress Testing**
- **Volume Testing**
- **Security Testing**
- **Compatibility Testing**
- **Reliability Testing**
- **Usability Testing**
- **Compliance Testing**
- **Localization Testing**

Alpha Testing

- It is the most common type of testing used in the Software industry. The objective of this testing is to identify all possible issues or defects before releasing it into the market or to the user.

- Alpha Testing is carried out at the end of the software development phase but before the Beta Testing. Still, minor design changes may be made as a result of such testing.

Acceptance Testing

- An Acceptance Test is performed by the client and verifies whether the end to end the flow of the system is as per the business requirements or not and if it is as per the needs of the end-user. Client accepts the software only when all the features and functionalities work as expected.
- It is the last phase of the testing, after which the software goes into production. This is also called User Acceptance Testing (UAT).

Ad-hoc Testing

- The name itself suggests that this testing is performed on an Ad-hoc basis i.e. with no reference to the test case and also without any plan or documentation in place for such type of testing.
- The objective of this testing is to find the defects and break the application by executing any flow of the application or any random functionality.

- Ad-hoc Testing is an informal way of finding defects and can be performed by anyone in the project. It is difficult to identify defects without a test case but sometimes it is possible that defects found during ad-hoc testing might not have been identified using existing test cases.

Accessibility Testing

- The aim of Accessibility Testing is to determine whether the software or application is accessible for disabled people or not.
- Here, disability means deaf, color blind, mentally disabled, blind, old age and other disabled groups. Various checks are performed such as font size for visually disabled, color and contrast for color blindness, etc.

Beta Testing

- Beta Testing is a formal type of Software Testing which is carried out by the customer. It is performed in **the Real Environment** before releasing the product to the market for the actual end-users.
- Beta Testing is carried out to ensure that there are no major failures in the software or product and it satisfies the business requirements from an end-user perspective. Beta Testing is successful when the customer accepts the software.

- Usually, this testing is typically done by end-users or others. It is the final testing done before releasing an application for commercial purpose. Usually, the Beta version of the software or product released is limited to a certain number of users in a specific area.

Black Box Testing

- Internal system design is not considered in this type of testing. Tests are based on the requirements and functionality.

Compatibility Testing

- It is a testing type in which it validates how software behaves and runs in a different environment, web servers, hardware, and network environment.

- Ensures that software can run on a different configuration, different database, different browsers, and their versions. Compatibility testing is performed by the testing team.

MODULE IV

PATH PRODUCTS AND PATH EXPRESSION:

- Flow graphs are being an abstract representation of programs.
- Any question about a program can be cast into an equivalent question about an appropriate flowgraph.
- Most software development, testing and debugging tools use flow graphs analysis techniques.

PATH PRODUCTS:

- Normally flow graphs used to denote only control flow connectivity.
- The simplest weight we can give to a link is a name. o Using link names as weights, we then convert the graphical flow graph into an equivalent algebraic like expressions which denotes the set of all possible paths from entry to exit for the flow graph. o Every link of a graph can be given a name.
- The link name will be denoted by lower case italic letters In tracing a path or path segment through a flow graph, you traverse a succession of link names.
- The name of the path or path segment that corresponds to those links is expressed naturally by concatenating those link names.

PATH EXPRESSION:

- Consider a pair of nodes in a graph and the set of paths between those node.
- Denote that set of paths by Upper case letter such as X,Y.
- The + sign is understood to mean "or" between the two nodes of interest, paths ac, or abc, or abbc, and so on can be taken.
- Any expression that consists of path names and "OR"s and which denotes a set of paths between two nodes is called a "Path Expression".

PATH PRODUCTS:

- The name of a path that consists of two successive path segments is conveniently expressed by the concatenation or Path Product of the segment names.
- For example, if X and Y are defined as X=abcde,Y=fghij,then the path corresponding to X followed by Y is denoted by XY=abcdefghij
- Similarly,
- YX=fghijabcde
- aX=aabcde
- Xa=abcdea
- XaX=abcdeaabcde
- o If X and Y represent sets of paths or path expressions, their product represents the
- set of paths that can be obtained by following every element of X by any element
- of Y in all possible ways. For example,
- o X = abc + def + ghi
- o Y = uvw + z
- Then,
- XY = abcuvw + defuvw + ghiuvw + abcz + defz + ghiz

PATH SUMS:

- The "+" sign was used to denote the fact that path names were part of the same set of paths.
- The "PATH SUM" denotes paths in parallel between nodes.
- Links a and b in Figure 5.1a are parallel paths and are denoted by a + b. Similarly,links c and d are parallel paths between the next two nodes and are denoted by c +d.
- The set of all paths between nodes 1 and 2 can be thought of as a set of parallelpaths and denoted by eacf+eadf+ebcf+ebdf.

REDUCTION PROCEDURE:

REDUCTION PROCEDURE ALGORITHM:

- This section presents a reduction procedure for converting a flowgraph whose links are labeled with names into a path expression that denotes the set of all entry/exit paths in that flowgraph. The procedure is a node-by-node removal algorithm.
- The steps in Reduction Algorithm are as follows:

1. Combine all serial links by multiplying their path expressions.

2. Combine all parallel links by adding their path expressions.

3. Remove all self-loops (from any node to itself) by replacing them with a link of the form X*, where X is the path expression of the link in that loop

STEPS 4 - 8 ARE IN THE ALGORIHTM'S LOOP:

4. Select any node for removal other than the initial or final node. Replace it with a set of equivalent links whose path expressions correspond to all the ways you can form a product of the set of inlinks with the set of outlinks of that node.

5. Combine any remaining serial links by multiplying their path expressions.

6. Combine all parallel links by adding their path expressions.

7. Remove all self-loops as in step 3.

8. Does the graph consist of a single link between the entry node and the exit node? If yes, then the path expression for that link is a path expression for the original flowgraph; otherwise, return to step 4.

REGULAR EXPRESSIONS:

The generic flow-anomaly detection problem (note: not just data-flow anomalies, but any flow anomaly) is that of looking for a specific sequence of options considering all possible paths through a routine.

- Let the operations be SET and RESET, denoted by s and r respectively, and we want to know if there is a SET followed immediately a SET or a RESET followed immediately by a RESET (an ss or an rr sequence).
- Some more application examples:

1. A file can be opened (o), closed (c), read (r), or written (w). If the file is read or written to after it's been closed, the sequence is nonsensical. Therefore, cr and cw are anomalous. Similarly, if the file is read before it's been written, just after opening, we may have a bug. Therefore, or is also anomalous. Furthermore, oo and cc, though not actual bugs, are a waste of time and therefore should also be examined.

2. A tape transport can do a rewind (d), fast-forward (f), read (r), write (w), stop (p), and skip (k). There are rules concerning the use of the transport; for example, you cannot go from rewind to fast-forward without an intervening stop or from rewind or fast-forward to read or write without an intervening stop. The following sequences are anomalous: df, dr, dw, fd, and fr. Does the flowgraph lead to anomalous sequences on any path? If so, what sequences and under what circumstances?

3. The data-flow anomalies discussed in Unit 4 requires us to detect the dd, dk, kk, and ku sequences. Are there paths with anomalous data flows?

OVERVIEW OF LOGIC BASED TESTING:

The functional requirements of many programs can be specified by decision tables, which provide a useful basis for program and test design.

- Consistency and completeness can be analyzed by using boolean algebra, which can also be used as a basis for test design. Boolean algebra is trivialized by using Karnaugh-Veitch charts.
- "Logic" is one of the most often used words in programmers' vocabularies but one of their least used techniques.
- Boolean algebra is to logic as arithmetic is to mathematics. Without it, the test or programmer is cut off from many test and design techniques and tools that incorporate those techniques.
- Logic has been, for several decades, the primary tool of hardware logic designers.
- Many test methods developed for hardware logic can be adapted to software logic testing. Because hardware testing automation is 10 to 15 years ahead of software testing automation, hardware testing methods and its associated theory is a fertile ground for software testing methods.
- As programming and test techniques have improved, the bugs have shifted closer to the process front end, to requirements and their specifications. These bugs range from 8% to 30% of the total and because they're first-in and last-out, they're the costliest of all.
- The trouble with specifications is that they're hard to express.
- Boolean algebra (also known as the sentential calculus) is the most basic of all logic systems.
- Higher-order logic systems are needed and used for formal specifications.
- Much of logical analysis can be and is embedded in tools. But these tool incorporate methods to simplify, transform, and check specifications and the methods are to a large extent based on boolean algebra.

MODULE V

GRAPH MATRICES AND APPLICATIONS:

Problem with Pictorial Graphs

- Graphs were introduced as an abstraction of software structure.

- Whenever a graph is used as a model, sooner or later we trace paths through it- to find a set of covering paths, a set of values that willsensitize paths, the logic function that controls the flow, the processing time of the routine, the equations that define the domain, or whether a state is reachable or not.
- Path is not easy, and it's subject to error. You can miss a link here and there or cover some links twice.
- One solution to this problem is to represent the graph as a matrix and to use matrix operations equivalent to path tracing. These methods are more methodical and mechanical and don't depend on your ability to see a path they are more reliable.
- If you build test tools or want to know how they work, sooner or later you will be implementing or investigating analysis routines based on these methods.
- It is hard to build algorithms over visual graphs so the properties or graph matrices are fundamental to tool building.

The Basic Algorithms

The basic tool kit consists of: Matrix multiplication, which is used to get the path expression from every node to every other node. A partitioning algorithm for converting graphs with loops into loop free graphs or equivalence classes. A collapsing process which gets the path expression from any node to any other node.

The Matrix of a Graph

A graph matrix is a square array with one row and one column for every node in the graph. Each row-column combination corresponds to a relation between

the node corresponding to the row and the node corresponding to the column. The relation for example, could be as simple as the link name, if there is a link between the nodes.

Some of the things to be observed: The size of the matrix equals the number of nodes. There is a place to put every possible direct connection or link between any and any other node. The entry at a row and column intersection is the link weight of the link that connects the two nodes in that direction.

1. A connection from node i to j does not imply a connection from node j to node i.

2. If there are several links between two nodes, then the entry is a sum; the "+" sign denotes parallellinks as usual.

Connection Matrix-continued

Each row of a matrix denotes the out links of the node corresponding to that row.

Each column denotes the in links corresponding to that node.

A branch is a node with more than one nonzero entry in its row.

A junction is node with more than one nonzero entry in its column.

A self loop is an entry along the diagonal.

Cyclomatic Complexity

The cyclomatic complexity obtained by subtracting 1 from the total number of entries in each row

and ignoring rows with no entries, we obtain the equivalent number of decisions for each row.

Adding these values and then adding 1 to the sum yields the graph's cyclomatic complexity.

The Powers of a Matrix

- Each entry in the graph's matrix expresses a relation between the pair of nodes that corresponds to that entry.
- Squaring the matrix yields a new matrix that expresses the relation between each pair of nodes via one intermediate node under the assumption that the relation is transitive.
- The square of the matrix represents all path segments two links long.
- The third power represents all path segments three links long.

Partitioning Algorithm

- Consider any graph over a transitive relation. The graph may have loops.
- We would like to partition the graph by grouping nodes in such a way that every loop is contained within one group or another.
- Such a graph is partially ordered.
- There are many used for an algorithm that does that:
- We might want to embed the loops within a subroutine so as to have a resulting graph which is loop free at the top level.
- Many graphs with loops are easy to analyze if you know where to break the loops.

 While you and I can recognize loops, it’s much harder to program a tool to do it unless you have
a solid algorithm on which to base the tool.