SATHYABAMA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ELECTIVE – SOFTWARE TESTING

UNIT 3 – SOFTWARE TESTING STRATEGIES

What is Static Testing?

Static Testing, a software testing technique in which the software is tested without executing the code. It has two parts as listed below:

- Review - Typically used to find and eliminate errors or ambiguities in documents such as requirements, design, test cases, etc.
- Static analysis - The code written by developers are analysed (usually by tools) for structural defects that may lead to defects.

Difference between Static and Dynamic Testing

	Static Testing	Dynamic Testing
1	In this type of testing we don't execute code	In this type of Testing we execute code
2	It means examining and reviewing the software.	It means testing, running, and using the software.
3	In this Testing methods like code review, inspection, walkthrough, reviews are used	In this Testing methods like testing and validations are used.
4	It is done in the phase of verification	It is done in the phase of validation.
5	This testing means "How we prevent" means it always talks about prevention	This testing means "How we cure" means it always talks about cure.
6	It is not a time consuming job because its purpose is to examine the software or code.	It is always a time consuming job because its purpose is to execute the software or code and it may also involve running more test cases.
7	As it can always start early in the life cycle it definitely reduces the cost of product or you can say project	As it not starting early in the life cycle hence it definitely increases the cost of product/project.
8	It is always considered as less cost effective job/task.	It is always considered as more cost effective job/task.
9	It can find errors that dynamic testing cannot find and it is a low level exercise	It can find errors that static testing cannot find and it is a high level exercise
10	It is not considered as a time Consuming job or task.	It is always considered as a time consuming job or task because it requires several test cases to execute.

11	Techniques/methods of static testing are inspections, reviews, and walkthroughs etc.	Technique/method of dynamic testing is always software testing means testing.
12	Static testing is also known by the name Dry Run Testing.	Dynamic Testing is not known by any other name.
13	It definitely comes before dynamic testing	It definitely follows after static testing

Static Analysis - By Tools:

Following are the types of defects found by the tools during static analysis:

- A variable with an undefined value
- Inconsistent interface between modules and components
- Variables that are declared but never used
- Unreachable code (or) Dead Code
- Programming standards violations
- Security vulnerabilities
- Syntax violations

Static Analysis (Static Code Analysis)

Static analysis, also called static code analysis, is a method of computer program debugging that is done by examining the code without executing the program. The process provides an understanding of the code structure, and can help to ensure that the code adheres to industry standards. Automated tools can assist programmers and developers in carrying out static analysis. The process of scrutinizing code by visual inspection alone (by looking at a printout, for example), without the assistance of automated tools, is sometimes called program understanding or program comprehension.

The principal advantage of static analysis is the fact that it can reveal errors that do not manifest themselves until a disaster occurs weeks, months or years after release. Nevertheless, static analysis is only a first step in a comprehensive software quality-control regime. After static analysis has been done, dynamic analysis is often performed in an effort to uncover subtle defects or vulnerabilities. In computer terminology, static means fixed, while dynamic means capable of action and/or change. Dynamic analysis involves the testing and evaluation of a program based on execution. Static and dynamic analyses, considered together, are sometimes referred to as glass-box testing.

What is Dynamic Testing?

Dynamic Testing is a kind of software testing technique using which the dynamic behaviour of the code is analysed.

For Performing dynamic, testing the software should be compiled and executed and parameters such as memory usage, CPU usage, response time and overall performance of the software are analyzed.Dynamic testing involves testing the software for the input values and output values are analyzed. Dynamic testing is the Validation part of Verification and Validation.

Dynamic Testing Techniques

The Dynamic Testing Techniques are broadly classified into two categories. They are:

- Functional Testing
- Non-Functional Testing

Levels of Dynamic Testing

There are various levels of Dynamic Testing Techniques. They are:

- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing

What is Dynamic testing technique?

- This testing technique needs computer for testing.
- It is done during Validation process.
- The software is tested by executing it on computer.
- Example of this **Dynamic Testing Technique**: **Unit testing**,**integration testing**, **system testing**.

What is White Box Testing?

White box testing is a testing technique that examines the program structure and derives test data from the program logic/code. The other names of glass box testing are clear box testing, open box testing, logic driven testing or path driven testing or structural testing.

White Box Testing Techniques:

- **Statement Coverage -** This technique is aimed at exercising all programming statements with minimal tests.

- **Branch Coverage -** This technique is running a series of tests to ensure that all branches are tested at least once.

- **Path Coverage -** This technique corresponds to testing all possible paths which means that each statement and branch is covered.

Advantages of White Box Testing:

- Forces test developer to reason carefully about implementation.

- Reveals errors in "hidden" code.
- Spots the Dead Code or other issues with respect to best programming practices.

Disadvantages of White Box Testing:

- Expensive as one has to spend both time and money to perform white box testing.
- Every possibility that few lines of code are missed accidentally.
- In-depth knowledge about the programming language is necessary to perform white box testing.

What is Software Metric?

Measurement is nothing but quantitative indication of size / dimension / capacity of an attribute of a product / process.Software metric is defined as a quantitative measure of an attribute a software system possesses with respect to Cost, Quality, Size and Schedule.

Example- Measure - No. of Errors, Metrics - No. of Errors found per person

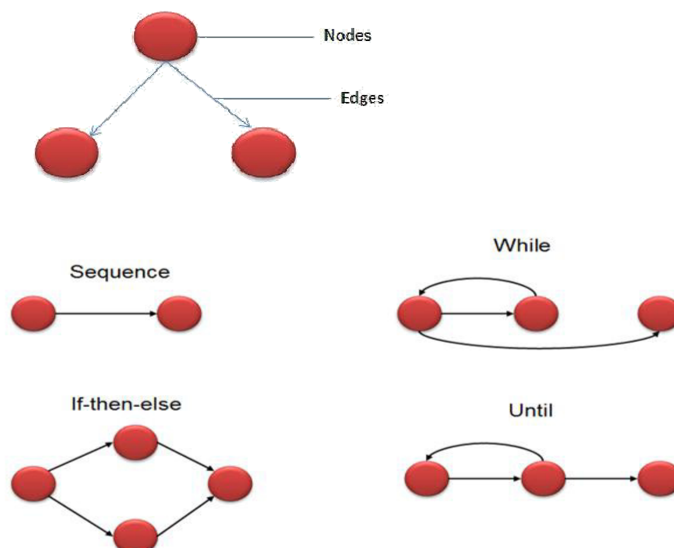
What is Cyclomatic Complexity?

Cyclomatic complexity is a software metric used to measure the complexity of a program. These metric, measures independent paths through program source code. Independent path is defined as a path that has atleast one edge which has not been traversed before in any other paths.

Cyclomatic complexity can be calculated with respect to functions, modules, methods or classes within a program. This metric was developed by Thomas J. McCabe in 1976 and it is based on a control flow representation of the program. Control flow depicts a program as a graph which consists of Nodes and Edges. In the graph, Nodes represent processing tasks while edges represent control flow between the nodes.

Flow graph notation for a program:

Flow Graph notation for a program is defines .several nodes connected through the edges. Below are Flow diagrams for statements like if-else, While, until and normal sequence of flow.



Mathematical representation:

Mathematically, it is set of independent paths through the graph diagram. The complexity of the program can be defined as -

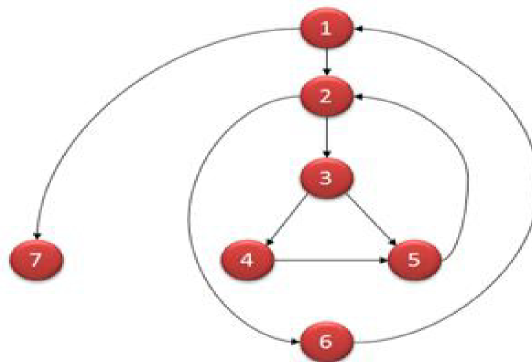
V(G) = E - N + 2   Where, E - Number of edges, N - Number of Nodes

V (G) = P + 1, Where P = Number of predicate nodes (node that contains condition)

Example -

```
1	i = 0;
2	n=4; //N-Number of nodes present in the graph
3	while (i<n-1) do
4	j = i + 1;
5	while (j<n) do
6	if A[i]<A[j] then
7	swap(A[i], A[j]);
8	end do;
9	i=i+1;
10	end do;
```

Flow graph for this program will be



Computing mathematically,

- V(G) = 9 - 7 + 2 = 4
- V(G) = 3 + 1 = 4 (Condition nodes are 1,2 and 3 nodes)

- Basis Set - A set of possible execution path of a program
- 1, 7
- 1, 2, 6, 1, 7
- 1, 2, 3, 4, 5, 2, 6, 1, 7
- 1, 2, 3, 5, 2, 6, 1, 7

Properties of Cyclomatic complexity:

Following are the properties of Cyclomatic complexity:

1. V (G) is the maximum number of independent paths in the graph
2. V (G) >=1
3. G will have one path if V (G) = 1
4. Minimize complexity to 10

How this metric is useful for software testing?

Basis Path testing is one of White box technique and it guarantees to execute atleast one statement during testing. It checks each linearly independent path through the program, which **means number test cases, will be equivalent to the cyclomatic complexity of the program.**

This metric is useful because of properties of Cyclomatic complexity (M) -

1. M can be number of test cases to achieve branch coverage(Upper Bound)
2. M can be number of paths through the graphs.(Lower Bound)

Consider this example -

```
1    If (Condition 1)
2    Statement 1
3    Else
4    Statement 2
5    If (Condition 2)
6    Statement 3
7    else
8    Statement 4
```

Cyclomatic Complexity for this program will be 9-7+2=4.

As complexity has calculated as 4, four test cases are necessary to the complete path coverage for the above example.

Steps to be followed:

The following steps should be followed for computing Cyclomatic complexity and test cases design.

Step 1 - Construction of graph with nodes and edges from the code
Step 2 - Identification of independent paths
Step 3 - Cyclomatic Complexity Calculation
Step 4 - Design of Test Cases
Once the basic set is formed, TEST CASES should be written to execute all the paths.

More on V (G):

Cyclomatic complexity can be calculated manually if the program is small. Automated tools need to be used if the program is very complex as this involves more flow graphs. Based on complexity number, team can conclude on the actions that need to be taken for measure.

Following table gives overview on the complexity number and corresponding meaning of v (G):

Complexity Score	Meaning
10-20	Complex Code, Medium Testability, Cost and effort is Medium
20-40	Very complex Code, Low Testability, Cost and Effort are high
>40	Not at all testable, Very high Cost and Effort

Tools for Cyclomatic Complexity calculation:

Many tools are available for determining the complexity of the application. Some complexity calculation tools are used for specific technologies. Complexity can be found by the number of decision points in a program. The decision points are if, for, for-each, while, do, catch, case statements in a source code.

Examples of tools are

- OCLint - Static code analyzer for C and Related Languages
- devMetrics - Analyzing metrics for C# projects
- Reflector Add In - Code metrics for .NET assemblies
- GMetrics - Find metrics in Java related applications
- NDepends - Metrics in Java applications

Uses of Cyclomatic Complexity:

Cyclomatic Complexity can prove to be very helpful in

- Helps developers and testers to determine independent path executions
- Developers can assure that all the paths have been tested atleast once
- Helps us to focus more on the uncovered paths
- Improve code coverage
- Evaluate the risk associated with the application or program

Control Structure testing.

Control structure testing is a group of white-box testing methods.

- Branch Testing
- Condition Testing
- Data Flow Testing
- Loop Testing

Branch Testing

- also called Decision Testing
- definition: "For every decision, each branch needs to be executed at least once."
- shortcoming - ignores implicit paths that result from compound conditionals.
- Treats a compound conditional as a single statement. (We count each branch taken out of the decision, regardless which condition lead to the branch.)
- This example has two branches to be executed: IF (a equals b) THEN

statement 1
ELSE statement 2
END IF

This examples also has just two branches to be executed, despite the compound conditional:

IF (a equals b AND c less than d) THEN statement 1
ELSE statement 2
END IF

This example has four branches to be executed: IF (a equals b) THEN

statement 1
ELSE
IF (c equals d) THEN statement 2
ELSE statement 3
END IF
END IF

- Obvious decision statements are if, for, while, switch.
- Subtle decisions are return *boolean expression*, ternary expressions, try-catch.
- For this course you don't need to write test cases for IOException and OutOfMemory exception.

Condition Testing

Condition testing is a test construction method that focuses on exercising the logical conditions in a program module.

Errors in conditions can be due to:

- Boolean operator error
- Boolean variable error
- Boolean parenthesis error
- Relational operator error
- Arithmetic expression error

Definition: "For a compound condition C, the true and false branches of C *and every simple condition in C* need to be executed at least once."

Multiple-condition testing requires that all true-false *combinations* of simple conditions be exercised at least once. Therefore, all statements, branches, and conditions are necessarily covered.

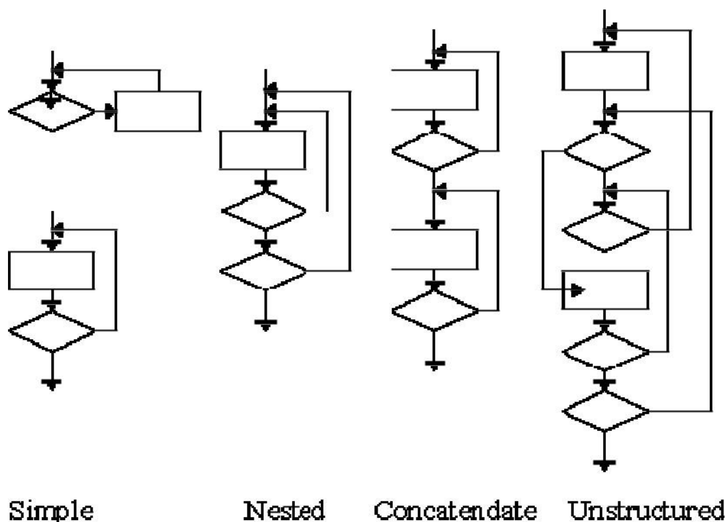
Data Flow Testing

Selects test paths according to the location of definitions and use of variables. This is a somewhat sophisticated technique and is not practical for extensive use. Its use should be targeted to modules with nested if and loop statements.

Loop Testing

Loops are fundamental to many algorithms and need thorough testing.There are four different classes of loops: simple, concatenated, nested, and unstructured.

Examples:



Create a set of tests that force the following situations:

- **Simple Loops,** where *n* is the maximum number of allowable passes through the loop.
    - Skip loop entirely
    - Only one pass through loop
    - Two passes through loop
    - m passes through loop where m<n.
    - (n-1), n, and (n+1) passes through the loop.
- **Nested Loops**
    - Start with inner loop. Set all other loops to minimum values.
    - Conduct simple loop testing on inner loop.
    - Work outwards
    - Continue until all loops tested.
- **Concatenated Loops**
    - If independent loops, use simple loop testing.
    - If dependent, treat as nested loops.
- **Unstructured loops**
    - Don't test - redesign.

What is Black box Testing?

Black-box testing is a method of software testing that examines the functionality of an application based on the specifications. It is also known as Specifications based testing. Independent Testing Team usually performs this type of testing during the software testing life cycle. This method of test can be applied to each and every level of software testing such as unit, integration, system and acceptance testing.

Behavioural Testing Techniques:

There are different techniques involved in Black Box testing.

- Equivalence Class
- Boundary Value Analysis
- Domain Tests
- Orthogonal Arrays
- Decision Tables
- State Models
- Exploratory Testing
- All-pairs testing

Equivalence Partitioning

Equivalence partitioning (EP) is a blackbox testing technique. This technique is very common and mostly used by all the testers informally. Equivalence partitions are also known as equivalence classes.

As the name suggests Equivalence partitioning is to divide or to partition a set of test conditions into sets or groups that can be considered same by the software system.

As you all know that exhaustive testing of the software is not feasible task for complex software's so by using equivalence partitioning technique we need to test only one condition from each partition because it is assumed that all the conditions in one partition will be treated in the same way by the software. If one condition works fine then all the conditions within that partition will work the same way and tester does not need to test other conditions or in other way if one condition fails in that partition then all other conditions will fail in that partition.

These conditions may not always be true however testers can use better partitions and also test some more conditions within those partitions to confirm that the selection of that partition is fine.

Some Examples:

A store in city offers different discounts depending on the purchases made by the individual. In order to test the software that calculates the discounts, we can identify the ranges of purchase values that earn the different discounts. For example, if a purchase is in the range of $1 up to $50 has no discounts, a purchase over $50 and up to $200 has a 5% discount, and purchases of $201 and up to $500 have a 10% discounts, and purchases of $501 and above have a 15% discounts.

Now we can identify 4 valid equivalence partitions and 1 invalid partition as shown below.

Invalid Partition	**Valid Partition(No Discounts)**	**Valid Partition(5%)**	**Valid Partition(10%)**	**Valid Partition(15%)**
$0.01	$1-$50	$51-$200	$201-$500	$501-Above

Boundary Value Analysis

A boundary value is any input or output value on the edge of an equivalence partition.

Example:1

Suppose you have a software which accepts values between 1-1000, so the valid partition will be (1-1000), equivalence partitions will be like:

Invalid Partition	**Valid Partition**	**Invalid Partition**
0	1-1000	1001 and above

And the boundary values will be 1, 1000 from valid partition and 0,1001 from invalid partitions.

Boundary Value Analysis is a black box test design technique where test case is designed by using boundary values; BVA is used in range checking.

Example: 2

A store in city offers different discounts depending on the purchases made by the individual. In order to test the software that calculates the discounts, we can identify the ranges of purchase values that earn the different discounts. For example, if a purchase is in the range of $1 up to $50 has no discounts, a purchase over $50 and up to $200 has a 5% discount, and purchases of $201 and up to $500 have a 10% discounts, and purchases of $501 and above have a 15% discounts.

We can identify 4 valid equivalence partitions and 1 invalid partition as shown below.

Invalid Partition	**Valid Partition(No Discounts)**	**Valid Partition(5%)**	**Valid Partition(10%)**	**Valid Partition(15%)**
$0.01	$1-$50	$51-$200	$201-$500	$501-Above

From this table we can identify the boundary values of each partition. We assume that two decimal digits are allowed.

Boundary values for Invalid partition :0.00
Boundary values for valid partition(No Discounts): 1, 50
Boundary values for valid partition(5% Discount): 51, 200
Boundary values for valid partition(10% Discount): 201,500
Boundary values for valid partition(15% Discount): 501, Max allowed number in the software application

What is Cause-Effect Graph?

Cause Effect Graph is a black box testing technique that graphically illustrates the relationship between a given outcome and all the factors that influence the outcome.It is also known as Ishikawa diagram as it was invented by Kaoru Ishikawa or fish bone diagram because of the way it looks.

Circumstances - under which Cause-Effect Diagram used

- To Identify the possible root causes, the reasons for a specific effect, problem, or outcome.
- To Relate the interactions of the system among the factors affecting a particular process or effect.
- To Analyze the existing problems so that corrective action can be taken at the earliest.

Benefits :

- It Helps us to determine the root causes of a problem or quality using a structured approach.
- It Uses an orderly, easy-to-read format to diagram cause-and-effect relationships.
- It Indicates possible causes of variation in a process.
- It Identifies areas, where data should be collected for further study.
- It Encourages team participation and utilizes the team knowledge of the process.
- It Increases knowledge of the process by helping everyone to learn more about the factors at work and how they relate.

Steps for drawing cause-Effect Diagram:

- **Step 1 :** Identify and Define the Effect
- **Step 2 :** Fill in the Effect Box and Draw the Spine
- **Step 3:** Identify the main causes contributing to the effect being studied.
- **Step 4 :** For each major branch, identify other specific factors which may be the causes of the EFFECT.
- **Step 5 :** Categorize relative causes and provide detailed levels of causes.

What is Syntax Testing?

Syntax Testing, a black box testing technique, involves testing the System inputs and it is usually automated because syntax testing produces a large number of tests. Internal and external inputs have to conform the below formats:

- Format of the input data from users.
- File formats.
- Database schemas.

Syntax Testing - Steps:

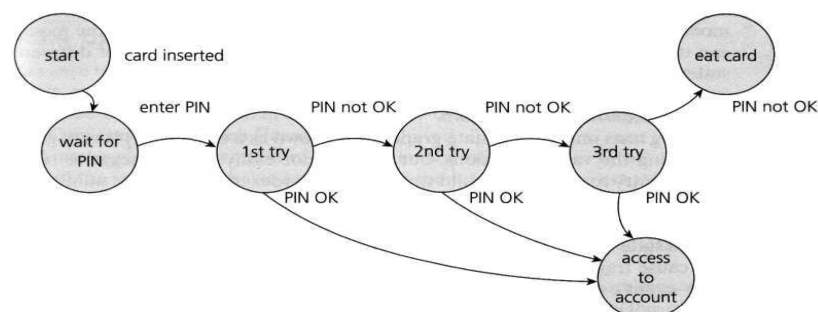
- Identify the target language or format.
- Define the syntax of the language.
- Validate and Debug the syntax. Syntax Testing

Limitations:

- Sometimes it is easy to forget the normal cases.
- Syntax testing needs driver program to be built that automatically sequences through a set of test cases usually stored as data.

State transition testing

State transition testing is used where some aspect of the system can be described in what is called a 'finite state machine'. This simply means that the system can be in a (finite) number of different states, and the transitions from one state to another are determined by the rules of the 'machine'. This is the model on which the system and the tests are based. Any system where you get a different output for the same input, depending on what has happened before, is a finite state system. A finite state system is often shown as a **state diagram**



Error guessing

Error guessing is a technique that should always be used as a complement to other more formal techniques. The success of error guessing is very much dependent on the skill of the tester, as good testers know where the defects are most likely to lurk. Some people seem to be naturally good at testing and others are good testers because they have a lot of experience either as a tester or working with a particular system and so are able to pin-point its weaknesses. This is why an error-guessing approach, used after more formal techniques have been applied to some extent, can be very effective. In using more formal techniques, the tester is likely to gain a better understanding of the system, what it does and how it works. With this better understanding, he or she is likely to be better at guessing ways in which the system may not work properly.

There are no rules for error guessing. The tester is encouraged to think of situations in which the software may not be able to cope. Typical conditions to try include division by zero, blank (or no) input, empty files and the wrong kind of data (e.g. alphabetic characters where numeric are required). If anyone ever says of a system or the environment in which it is to operate 'That could never happen', it might be a good idea to test that condition, as such assumptions about what will and will not happen in the live environment are often the cause of failures. A structured approach to the error-guessing technique is to list possible defects or failures and to design tests that attempt to produce them. These defect and failure lists can be built based on the tester's own experience or that of other people, available defect and failure data, and from common knowledge about why software fails.