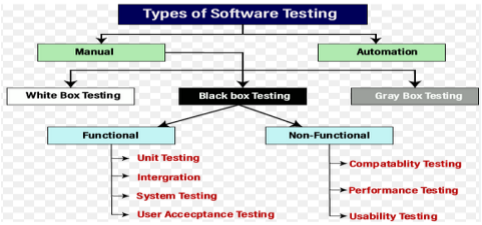
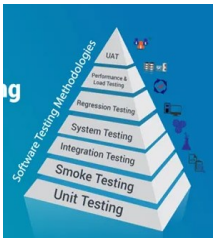


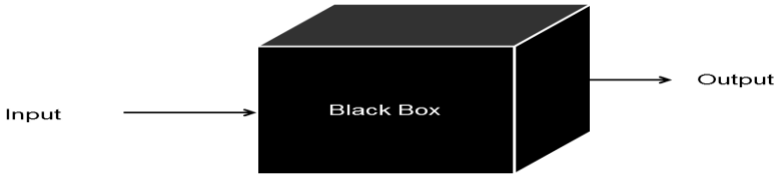


Software Engineering And Testing

Software Testing

Black Box Testing

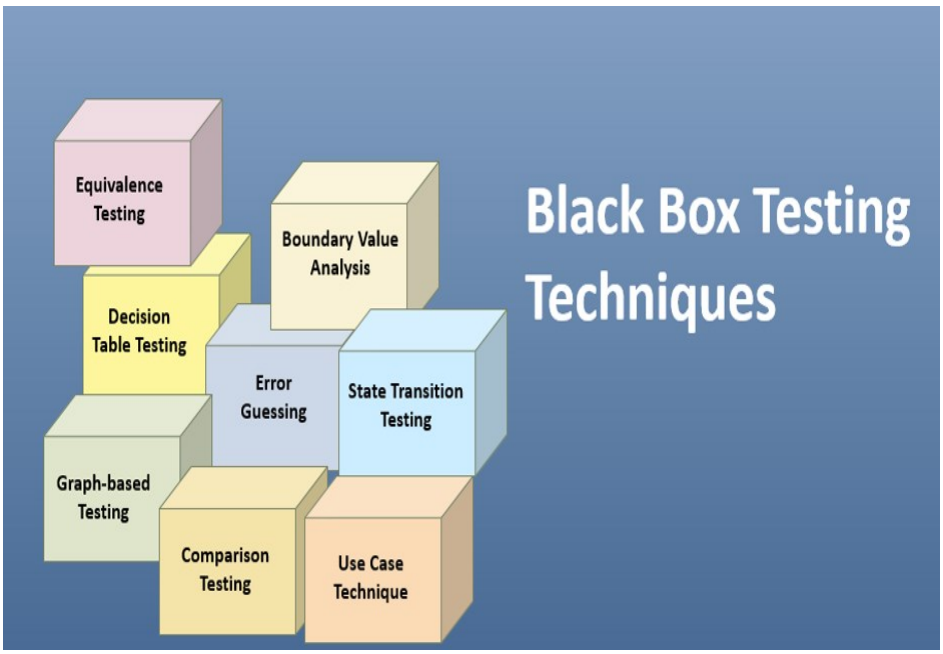
- The term 'Black Box' refers to the software, which is treated as a black box.
- By treating it as a black box, we mean that the system or source code is not checked at all.
- It is done from customer's viewpoint.
- The test engineer engaged in black box testing only knows the set of inputs and expected outputs and is unaware of how those inputs are transformed into outputs by the software.



Internal behavior of the code is unknown

Black Box Testing

- Black-box testing attempts to find errors in the following categories:
  - To test the modules independently.
  - To test the functional validity of the software
  - Interface errors are detected.
  - To test the system behavior and check its performance.
  - To test the maximum load or stress on the system.
  - Customer accepts the system within defined acceptable limits.



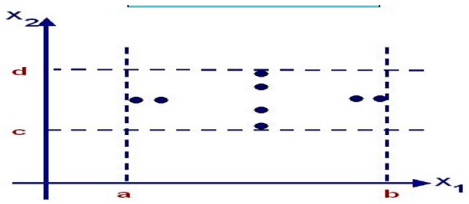
Boundary Value Analysis (BVA)

- This Black Box testing technique believes and extends the concept that the density of defect is more towards the boundaries. This is done due to the following reasons
  - Usually the programmers are not able to decide whether they have to use <= operator or < operator when trying to make comparisons.
  - Different terminating conditions of For-loops, While loops and Repeat loops may cause defects to move around the boundary conditions.
  - The requirements themselves may not be clearly understood, especially around the boundaries, thus causing even the correctly coded program to not perform the correct way.

5

Boundary Value Analysis (BVA)

- The basic idea of BVA is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum and at their maximum. Meaning thereby (min, min+, nom, max-, max), as shown in the following figure

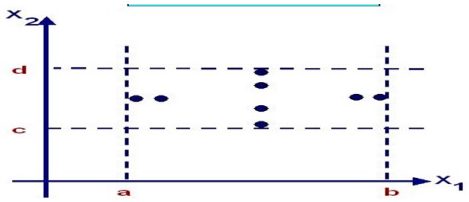


- Example: A textbox can input integer numbers from 1 to 99
- Select value for test case: 1; 2; 98; 99

Boundary Value Analysis (BVA)

- The variable at its extreme value can be selected as :

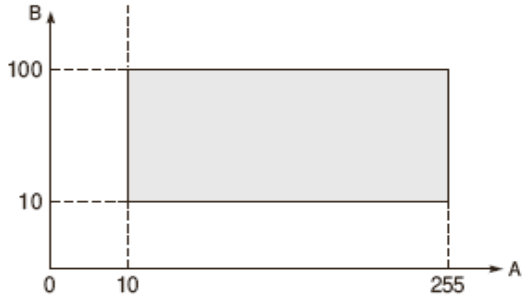
1. Minimum value (min)
2. Value just above the minimum value (min^+)
3. Maximum Value (max)
4. Value just below the maximum value (max^-)
5. Nominal Value (nom)



7

Boundary Value Analysis (BVA)

- Example : if A is an integer between 10 and 255, then boundary checking can be on 10(9,10,11) and on 255(256,255,254).
- Similarly, B is another integer variable between 10 and 100, then boundary checking can be on 10(9,10,11) and 100(99,100,101), as shown in Fig.



8

Boundary Value Analysis (BVA)

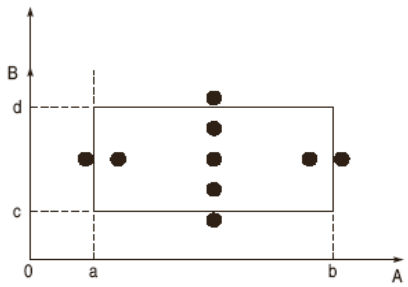
- BVA is based upon a critical assumption that is known as "**Single fault assumption theory**". According to this assumption, we derive the test cases on the basis of the fact that failures are not due to simultaneous occurrence of two (or more) faults. So, we derive test cases by holding the values of all but one variable at their nominal values and allowing that variable assume its extreme values.
- If we have a function of **n-variables**, we hold all but one at the nominal values and let the remaining variable assume the **min, min+, nom, max-and max values**, repeating this for each variable.
- Thus, for a function of **n** variables, BVA yields **(4n + 1)** test cases.

9

Boundary Value Analysis (BVA)

- Consider 2 inputs to a program

1. A_{nom}, B_{min}
2. A_{nom}, B_{min+}
3. A_{nom}, B_{max}
4. A_{nom}, B_{max-}
5. A_{min}, B_{nom}
6. A_{min+}, B_{nom}
7. A_{max}, B_{nom}
8. A_{max-}, B_{nom}
9. A_{nom}, B_{nom}



- 4n+1 test cases can be designed with boundary value checking method.

10

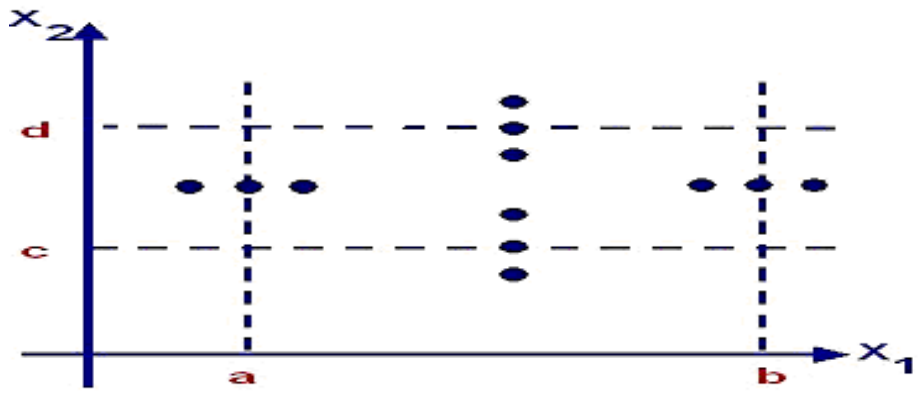
Robustness Testing

- Robustness Testing is another variant of BVA
- In BVA, we remain within the legitimate boundary of our range i.e. for testing we consider values like **(min, min+, nom, max-, max)** whereas in Robustness testing, we try to cross these legitimate boundaries as well.
- Thus for testing here we consider the values like **(min-, min, min+, nom, max-, max, max+)**
- Again, with robustness testing, we can focus on exception handling. With strongly typed languages, robustness testing may be very awkward. For example, in PASCAL, if a variable is defined to be within a certain range, values outside that range result in run-time errors thereby aborting the normal execution.

11

Robustness Testing

- For a program with **n**-variables, robustness testing will yield **(6n + 1)** test-cases. Thus we can draw the following Robustness Test Cases graph



12

Robustness Testing

- A value just greater than the Maximum value (Max^+)
- A value just less than Minimum value (Min^-)
- When test cases are designed considering above points in addition to BVC, it is called Robustness testing.
- For n=2 we have 13 testcases.

10. A_{max+}, B_{nom}
11. A_{min-}, B_{nom}
12. A_{nom}, B_{max+}
13. A_{nom}, B_{min-}

13

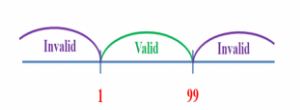
Limitations of Boundary Value Analysis (BVA)

- Boolean and logical variables present a problem for Boundary Value Analysis
- BVA assumes the variables to be truly independent which is not always possible.
- BVA test cases have been found to be rudimentary because they are obtained with very little insight and imagination.

14

Equivalence Partitioning

- Inputs to the software are divided into groups that are expected to exhibit similar behavior, so they are likely to be processed in the same way.
- Therefore, instead of taking every value in one domain, only one test case is chosen from one class.
- Equivalence partitions can be found for
  - Valid data – values that should be accepted.
  - Invalid data – values that should be rejected.
- Example: A textbox can input integer numbers from 1 to 99
  - Partition 1 – invalid: < 1
  - Partition 2 – valid: 1 <= & <=99
  - Partition 3 – invalid: > 99



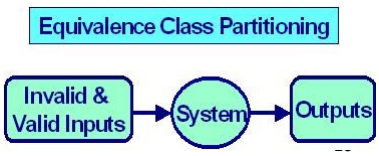
- Equivalence Partitioning is also know.. as Equivalence Class Partitioning.

15

Equivalence Partitioning

- The goal of equivalence partitioning method is to reduce these redundant test cases.
- To use equivalence partitioning, one needs to perform two steps:

1. Identify equivalence classes
2. Design test cases



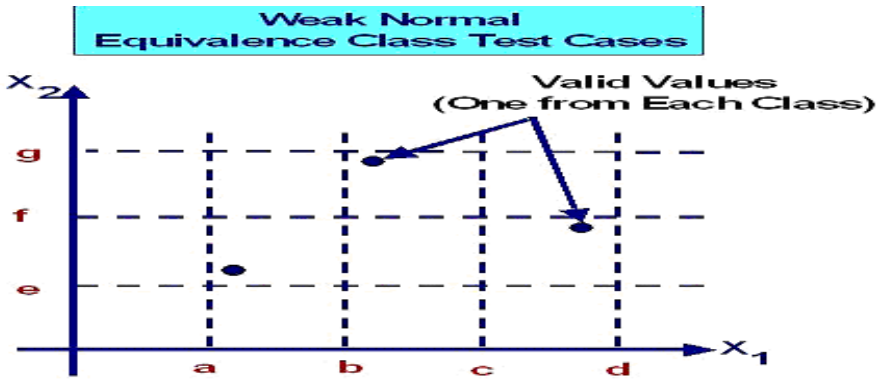
Types of Equivalence Partitioning

1. Weak Normal Equivalence Class Testing.
2. Strong Normal Equivalence Class Testing.
3. Weak Robust Equivalence Class Testing.
4. Strong Robust Equivalence Class Testing.

17

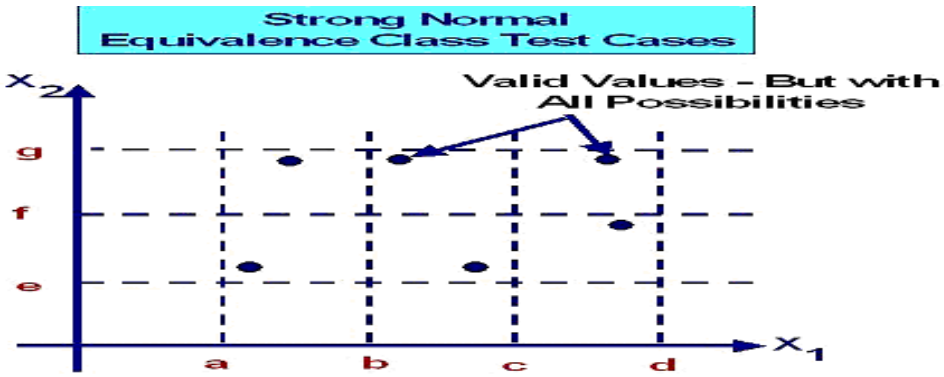
Weak Normal Equivalence Class Testing

- The word 'weak' means 'single fault assumption'.
- This type of testing is accomplished by using one variable from each equivalence class in a test case. [valid values]
- We would, thus, end up with the weak equivalence class test cases as shown in the following figure



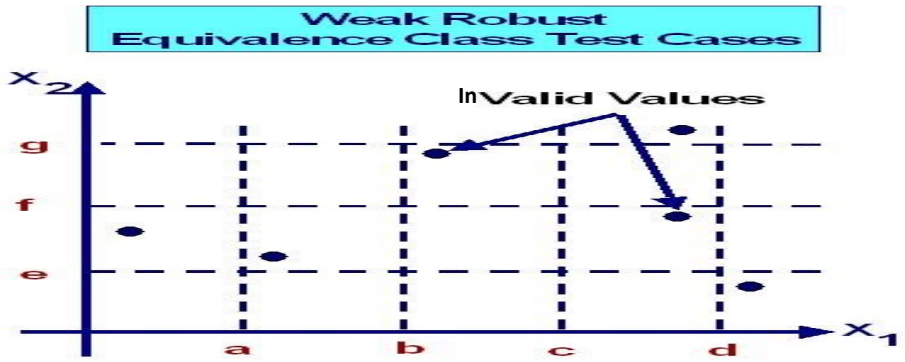
Strong Normal Equivalence Class Testing

- This type of testing is based on the multiple fault assumption theory. So, now we need test cases from each element of the Cartesian product [like truth table] of the equivalence classes, as shown in the following figure.



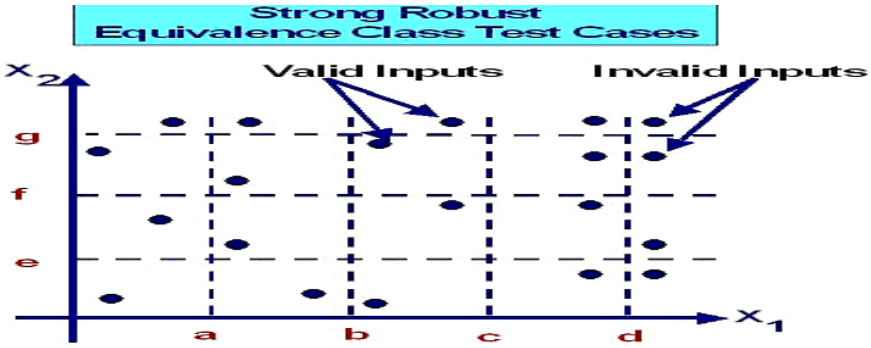
Weak Robust Equivalence Class Testing

- The word' weak' means single fault assumption theory and the word 'Robust' refers to invalid values.
- The test cases resulting from this strategy are shown in the following figure



Strong Robust Equivalence Class Testing

- As explained earlier also, 'robust' means consideration of invalid values and the 'strong' means multiple fault assumption.
- We obtain the test cases from each element of the Cartesian product of all the equivalence classes as shown in the following figure



Guidelines for Equivalence Class Testing

1. The weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed and invalid values cause run-time errors then there is no point in using the robust form.
3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is approximate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing (BVA).

Guidelines for Equivalence Class Testing

6. Equivalence class testing is used when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes.
7. Strong equivalence class testing makes a presumption that the variables are independent and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate "error" test cases.
8. Several tries may be needed before the "right" equivalence relation is established.
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

23

Difference between ECT & BVA

Equivalence Class Testing	*Boundary Value Analysis*
1. Equivalence Class Testing is a type of black box technique.	1. Next part of Equivalence Class Partitioning/Testing.
2. It can be applied to any level of testing, like unit, integration, system, and more.	2. Boundary value analysis is usually a part of **stress** & **negative testing**.
3. A **test case design technique** used to divide input data into different equivalence classes.	3. This test case design technique used to test boundary value between partitions.
4. Reduces the time of testing, while using less and effective test cases.	4. Reduces the overall time of test execution, while making **defect detection** faster & easy.
5. Tests only one from each partition of the equivalence classes.	5. Selects test cases from the edges of the equivalence classes.

Some of tools for Black Box

Auto Hotkey	JMeter	Appium
OWASP ZEBEDIA	Auto Italia Online	Selenium IDE
Ranorex	Katalon	MbUnit
QTP/ UFT	SilkTest	Water
Selendroid	Gremlins	IBM Rational Functional tester
Squish by froglogic	Aegis Web	Applitools
Microsoft Coded UI	HP QTP	

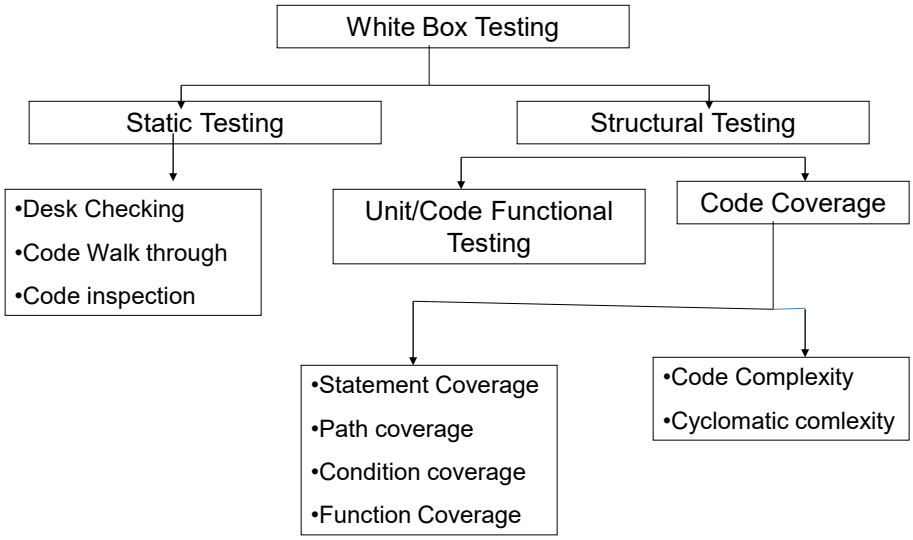
25

Introduction

- White-box or structural or development testing is another effective testing technique in dynamic testing.
- The structure means the logic of the program which has been implemented in the language code.
- Every software product is realized by means of a program code. White box testing is a way of testing the external functionality of the code by examining and testing the program code that realizes the external functionality.
- White box testing takes into account the program code, code structure, and internal design flow.
- It is also known as **glass-box** testing, as everything that is required to implement the software is visible.
- This testing is also called **structural** or **development** testing.
- A number of defects come about because of incorrect translation of requirements and design into program code.

26

Types of White Box Testing



27

Need Of White-box Testing

- White-box testing techniques are used for testing the module for initial stage testing. Black-box testing is the second stage for testing the software.
  - Though test cases for black box can be designed earlier than white-box test cases, they cannot be executed until the code is produced and checked using white-box testing techniques.
  - Thus, white-box testing is not an alternative but an essential stage.
- There are categories of bugs which can be revealed by white-box testing, but not through black-box testing. There may be portions in the code which are not checked when executing functional test cases, but these will be executed and tested by white-box testing.

28

Logic Coverage Criteria

- Structural testing considers the program code, and test cases are designed based on the logic of the program such that every element of the logic is covered.
- Structural testing takes account of code, code structure, internal designs, and how they are coded. In structural testing tests are actually run by the computer on the built product.
- The knowledge of internal structure of the code can be used to find the number of test cases required to guarantee the given level of test coverage.
- Structural testing entails the actual product against some pre-designed test cases to exercise as much as of the code as possible or necessary. A given portion of the code is exercised if a test case causes the program to execute that portion of the code when running the test.

29

Logic Coverage Criteria

- Since a product is realized in terms of program code, if we can run test cases to exercise the different parts of the code, then that part of the product is realized by the code gets tested. Code coverage testing involves designing and executing test cases and finding out the percentage of code that is covered by the testing.
- This leads to the notation of code coverage
- Divide the program into elements(e;g; statements)
- Define the coverage of a test suite to be

$$\frac{\#\ of\ elements\ executed\ by\ suite}{\#\ of\ element\ in\ the\ program} * 100$$

30

Statement Coverage

- In most of the programming languages, the program construct may be a sequential control flow, a two-way decision statement like if – then – else, a multi-way decision statement like switch or even loops like while, do, repeat until and for.
- Statement coverage refers to writing test cases that execute each of the program statements. We assume that "more the code covered, the better is the testing of the functionality".
- For a set of sequential statements (i.e., with no conditional branches), test cases can be designed to run through from top to bottom.
- The statement coverage for a program, which is an indication of the percentage of statements actually executed in a set of tests, can be calculated by the following formula
- **Statement Coverage = (Total Statements Exercised) / (Total Number of Executable Statements in Program) x 100**

31

Statement Coverage

- Draw the flow in the following way:
  - Nodes represent entries, exits, decisions and each statement of code.
  - Edges represent non-branching and branching links between nodes.

32

Branch Coverage

- Here test cases are designed such that the different branch condition are given true and false value in turn.
- Here test requirements: Branches in a program.

$$\frac{\#\ of\ executed\ branches}{\#\ of\ branches\ in\ program} * 100$$

- It is stronger testing criteria than the statement coverage based testing.
- Example : Take the above problem of balance
- Construct the control flow graph of it.

33

How to draw Control flow graph?

- Number all the statements of a program.
- Numbered statements: represent nodes of the control flow graph.
- An edge from one node to another node exists: if execution of the statement representing the first node can result in transfer of control to the other node.
- Elements of CFG
- Nodes: (3 types)
  - Statement nodes : single- entry, single exit sequence of statements
  - Predicate (decision) nodes: Conditions for branching.
  - Auxiliary nodes (optional) : for easier understanding (e.g. “merge point” for if)
- Edges : Possible flow of control

34

Condition Coverage

- For example, in if-then-else, there are 2^2 or 4 possible True / False conditions.
- The condition coverage which indicates the percentage of conditions covered by a set of test cases, is defined by the following formula

  Condition Coverage = (Total Decisions Exercised) / (Total Number of Decisions in Program) x 100
- Thus it becomes quite clear that this technique of condition coverage is much stronger criteria than path coverage, which in turn is a much stronger criteria than statement coverage.

35

Path testing

- In path coverage technique, we split a program into a number of distinct paths. A program or a part of a program can start from the beginning and take any of the paths to its completion. The path coverage of a program may be calculated based on the following formula

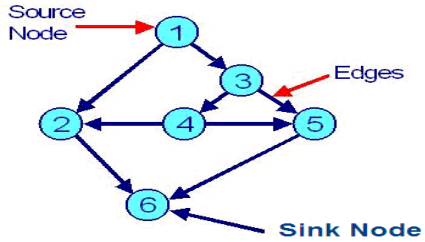
  Path Coverage = (Total Path Exercised) / (Total Number of Paths in Program) X 100
- In path coverage based testing strategy requires us to design test cases such that all linearly independent paths in the program are executed at least once.
- A linearly independent path is defined in terms of the control flow graph (CFG) of a program.
- A control flow graph (CFG) describes: the sequence in which different instructions of a program get executed.
  - the way control flows through the program.

36

Path testing

- Consider the following flow graph having different paths.

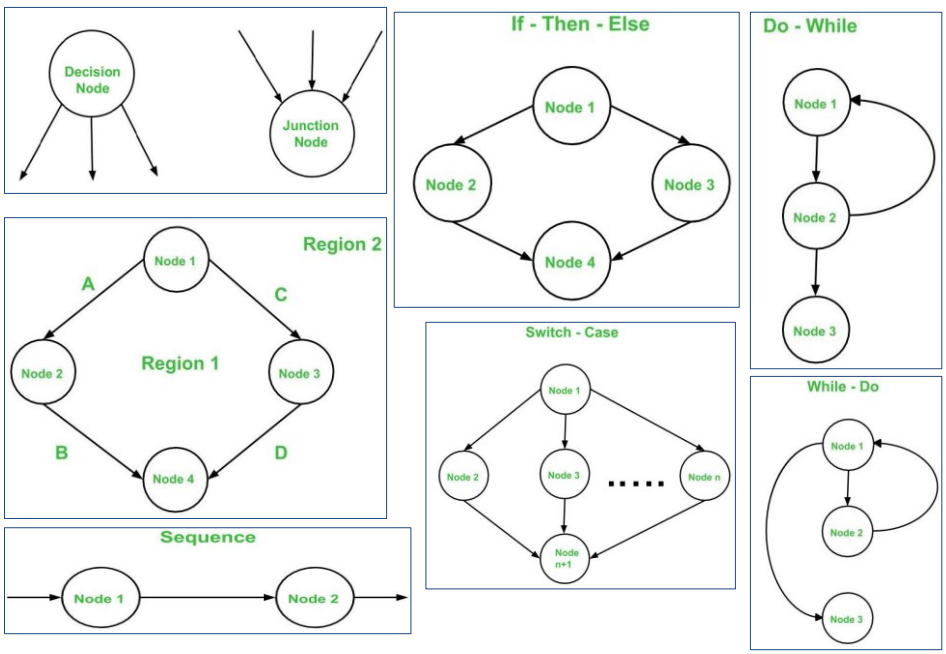
Some of the paths are
Path-1: 1 > 2 > 6
Path-2: 1 > 3 > 5 > 6
Path-3: 1 > 3 > 4 > 5 > 6
Path-4: 1 > 3 > 4 > 2 > 6



- Regardless of the number of statements in each of these paths, if we can execute these paths, then we would have covered most of the typical scenarios.
- Path coverage provides a stronger condition of coverage compared to statement coverage as it relates to the various logical paths in the program rather than just program statements.

37

Standard notations used in constructing a flow graph are as under



Cyclomatic Complexity

- McCabe's Cyclomatic metric, V(G) of a graph "G" with "n" vertices and "e" edges is given by the following formula
- V(G) = e – n + 2
- Steps to compute the complexity measure, V(G) are as under

1. Construct the flow graph from the source code or flow charts.
2. Identify independent paths.
3. Calculate Cyclomatic Complexity, V(G).
4. Design the test cases.

39

Meaning of V(G)

Complexity No.	Corresponding Meaning of V(G)
1-10	1) Well-written code, 2) Testability is high, 3) Cost / effort to maintain is low
10-20	1) Moderately complex code, 2) Testability is medium, 3) Cost / effort to maintain is medium.
20-40	1) Very complex code, 2) Testability is low, 3) Cost / effort to maintain is high.
> 40	1) Not testable, 2) Any amount of money / effort to maintain may not be enough.

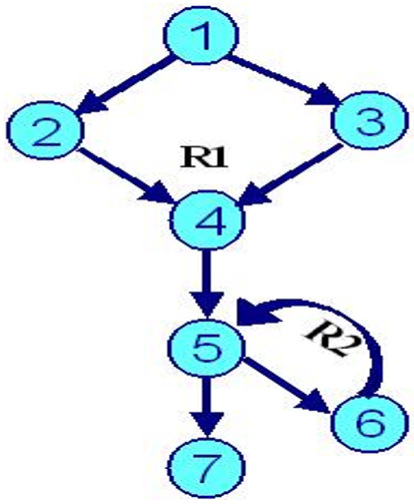
40

Cyclomatic Complexity Example

```
void foo (float y, float a *, int n)

{
                                /*1*/
    float x = sin (y) ;         /*1*/
    if (x > 0.01)               /*1*/
            z = tan (x) ;       /*2*/
    else

            z = cos (x) ;       /*3*/
    for (int i = 0 ; i < x ; + + i)/*4*/
    {                           /*5*/
            a[i] = a[i] * z ;   /*6/
            cout < < a [i] ;    /*6*/
    }                           /*6*/
}                               /*7*/
```



41

Cyclomatic Complexity Example Calculation

- **Method – 1**: V(G) = e – n + 2 (Where “e” are edges & “n” are nodes) V(G) = 8 – 7 + 2=3
- **Method – 2**: V(G) = P + 1 (Where P- predicate nodes with out degree = 2) V(G) = 2 + 1 = 3 (Nodes 1 and 5 are predicate nodes)
- **Method – 3**: V(G) = Number of enclosed regions + 1 = 2+1=3
  - V(G) = 3 and is same by all the three methods.
- **Conclusion – 1**: By getting a value of V(G) = 3 means that it is a “well written” code, its “testability” is high and cost / effort to maintain is low.
- **Conclusion – 2**: There are 3 paths in this program which are independent paths and they form a basis-set.
  - Path 1: 1 – 2 – 4 – 5 - 7
  - Path 2: 1 – 3 – 4 – 5 - 7
  - Path 3: 1 – 3 – 4 – 5 – 6 - 7

42

Data Flow testing

- Data flow testing is another form of structural testing. Here, we concentrate on the usage of variables and the focus points are:
  1. Statements where variables receive values
  2. Statements where these values are used or referenced
- Flow graphs are also used as a basis for the data flow testing as in the case of path testing.
- With a program data definition faults are nearly as frequent (22% of all faults) as control flow faults (24 percent).
- As we know variables are defined and referenced throughout the program. We may have few define/reference anomalies:
  1. A variable is defined but not used/referenced
  2. A variable is used but never defined
  3. A variable is defined twice before it is used

43

State of a Data Object

- **Defined (d)** : A data object is called defined when it is initialized, i.e. when it is on the left side of assignment statement. Defined state can also be used to mean that a file has been opened, a dynamically allocated objects has been allocated, something is pushed onto the stack, a record is written, and so on.
- **Killed/Undefined/Released (k)** : When the data has been reinitialized or the scope of a loop control variable finishes, i.e. existing the loop or memory is released dynamically or a file has been closed.
- **Usage (u)** : When the data object is on the right side of assignment or used as a control variable in a loop, or in an expression used to evaluate the control flow of a case statement, or as a pointer to an object, etc. In general, we say that the usage is either computational use **(c-use)** or predicate use **(p-use)**.

44

Terminology used in Data Flow Testing

- **Definition node**: Defining a variable means assigning value to a variable for the very first time in the program. For example, input statements, assignment statements, loop control statements, procedure calls, etc.
- **Usage node:** It means the variable has been used in some statement of the program. Node n that belongs to G(P) is a usage node of variable v, if the value of variable v is used at the statement corresponding to node n. for example, output statements, assignment statements (right), conditional statements , loop control statements, etc.
- A usage node can be of the following two types:
  - **Predicate usage node (p-use)** : if usage node n is predicate node, then n is predicated usage node. The predicate use or p-use is assigned to both branches out of the decision statements.

45

Terminology used in Data Flow Testing

  - **Computation usage node (c-use):** If usage node n corresponds to a computation statement in a program other than predicated, then it is called a computation usage node. c-use is when the variable appears on the RHS of an assignment statement. A c-use is said to occur on the assignment statement.
- **Loop free path segment** : It is path segment for which every node is visited once at most.
- **Simple path segment** : It is path segment in which at most one node is visited twice. A simple path segment is either loop-free or if there is a loop, only one node is involved.
- **Definition-use path (du-path)** : A du-path with respect to a variable v is a path between the definition node and the usage node of that variable. Usage node can be either p-use or c-use node.

46

Terminology used in Data Flow Testing

- **Definition-clear path (dc-path)** : A dc-path with respect to a variable v is a path between the definition node and the usage node such that no other node in the path is defining node of variable v.
- The du-path which are not dc-paths are important from testing view-point, as these are potential problematic spots for testing persons.
- Steps followed for DFT are given below:

1. Draw the CFG
2. Prepare a table for define/use status of all variables used in your program.
3. Find all du-paths
4. Identify du-paths that are not dc-paths.

47

White Box Testing Tools

VERACODE Securing the Software That Runs the World

EclEmma Java Code Coverage

RCUNIT

CppUnit

JUnit 5

JSUnit.net
Java Script Testing Tool

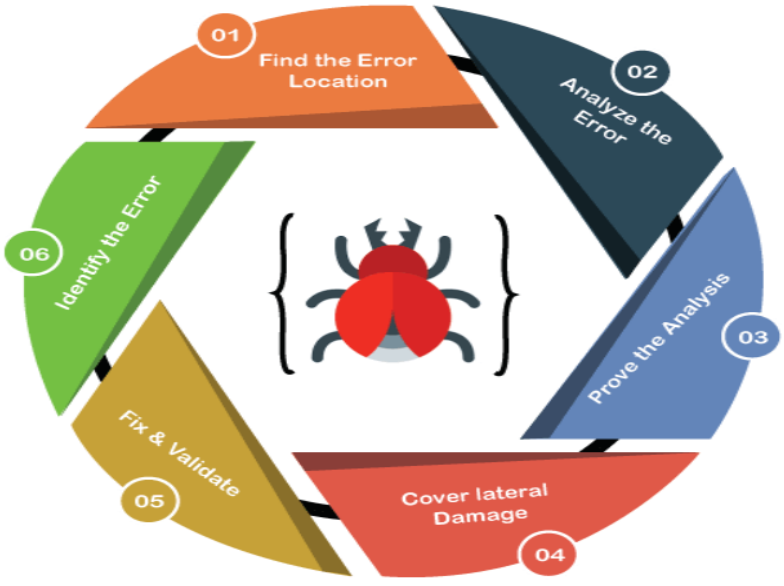
48

Debugging

- Debugging is the process of finding and resolving coding errors or “bugs” in a software program. Bugs (logical errors, runtime errors, syntax errors and others) can lead to crashes, incorrect or inaccurate outputs, security vulnerabilities, data loss and more.
- Debugging is the process of identifying and resolving errors, or bugs, in a software system. It is an important aspect of software engineering because bugs can cause a software system to malfunction, and can lead to poor performance or incorrect results. Debugging can be a time-consuming and complex task, but it is essential for ensuring that a software system is functioning correctly.

19

Steps involved in Debugging



50

Debugging Approaches/Strategies

- **Brute Force**: Study the system for a longer duration to understand the system. It helps the debugger to construct different representations of systems to be debugged depending on the need. A study of the system is also done actively to find recent changes made to the software.
- **Backtracking**: Backward analysis of the problem which involves tracing the program backward from the location of the failure message to identify the region of faulty code. A detailed study of the region is conducted to find the cause of defects.
- **Cause elimination**: it introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes.
- **Static analysis**: Analyzing the code without executing it to identify potential bugs or errors. This approach involves analyzing code syntax, data flow, and control flow.

51

Debugging Approaches/Strategies

- **Dynamic analysis**: Executing the code and analyzing its behavior at runtime to identify errors or bugs. This approach involves techniques like runtime debugging and profiling.
- **Collaborative debugging**: Involves multiple developers working together to debug a system. This approach is helpful in situations where multiple modules or components are involved, and the root cause of the error is not clear.
- **Automated Debugging**: The use of automated tools and techniques to assist in the debugging process. These tools can include static and dynamic analysis tools, as well as tools that use machine learning and artificial intelligence to identify errors and suggest fixes.

52

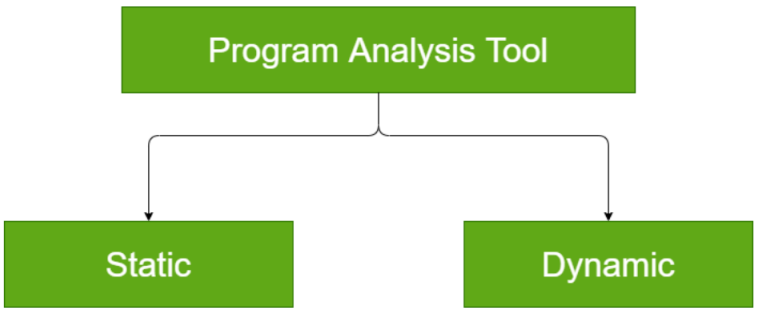
Examples of error during debugging

- Syntax error
- Logical error
- Runtime error
- Stack overflow
- Index Out of Bound Errors
- Infinite loops
- Concurrency Issues
- I/O errors
- Environment Dependencies
- Integration Errors
- Reference error
- Type error

53

Program Analysis Tools

- Program Analysis Tool is an automated tool whose input is the source code or the executable code of a program and the output is the observation of characteristics of the program.
- It gives various characteristics of the program such as its size, complexity, adequacy of commenting, adherence to programming standards and many other characteristics.



54

Static Program Analysis Tools

- Is such a program analysis tool that evaluates and computes various characteristics of a software product without executing it.
- Some structural properties are analyzed using static program analysis tools are:
  - Whether the coding standards have been fulfilled or not.
  - Some programming errors such as uninitialized variables.
  - Mismatch between actual and formal parameters.
  - Variables that are declared but never used.
- Code walkthroughs and code inspections are considered as static analysis methods but static program analysis tool is used to designate automated analysis tools. Hence, a compiler can be considered as a static program analysis tool.

55

Dynamic Program Analysis Tools

- Is such type of program analysis tool that require the program to be executed and its actual behavior to be observed.
- A dynamic program analyzer basically implements the code. It adds additional statements in the source code to collect the traces of program execution.
- When the code is executed, it allows us to observe the behavior of the software for different test cases.
- Once the software is tested and its behavior is observed, the dynamic program analysis tool performs a post execution analysis and produces reports which describe the structural coverage that has been achieved by the complete testing process for the program.

56


THANKS