Unit VI

SOFTWARE TESTING FUNDAMENTALS: STRATEGIC APPROACH TO SOFTWARE TESTING, UNIT TESTING, INTEGRATION TESTING, VERIFICATION, VALIDATION TESTING, SYSTEM TESTING, TEST STRATEGIES FOR WEBAPPS

SOFTWARE TESTING TECHNIQUES: WHITE BOX TESTING, BASIS PATH TESTING, CONTROL STRUCTURE TESTING AND BLACK BOX TESTING. TDD

6TH EDITION
Chapter 13: Testing Strategies
Chapter 14: Testing Tactics

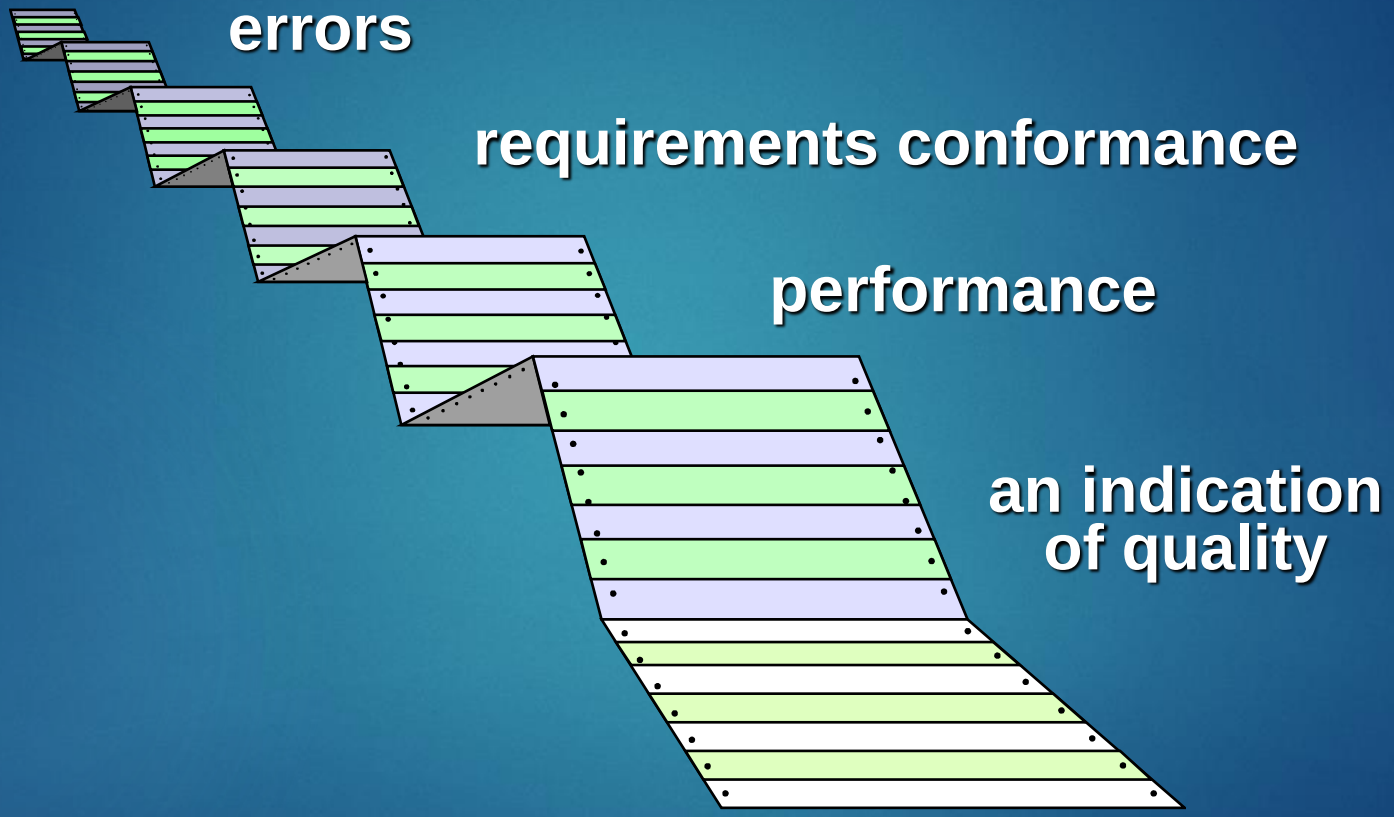
7TH EDITION
Chapter 17: Testing Strategies
Chapter 18: Testing Tactics

Software Testing

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Verification:- Are we building the product right?
Validation:- Are we building the right product?

What Testing Shows



Who Tests the Software?

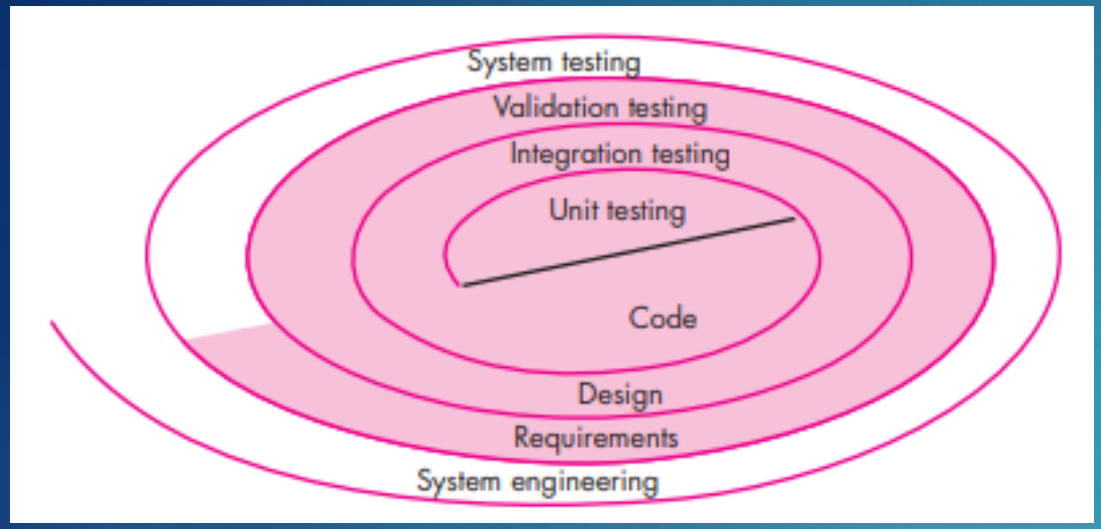
developer

Understands the system
but, will test "gently"
and, is driven by "delivery"

independent tester

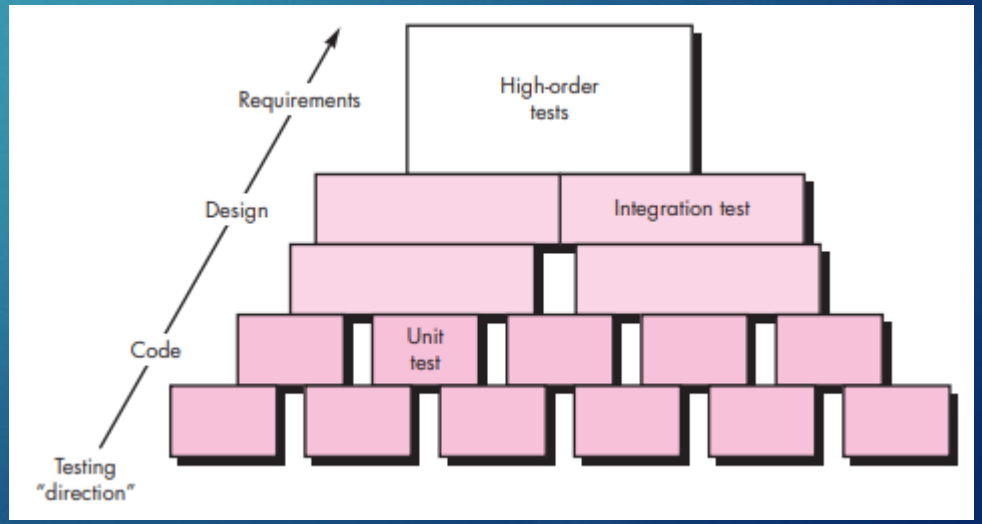
Must learn about the system,
but, will attempt to break it
and, is driven by quality

Software testing for conventional architecture



Software testing strategy

Software testing steps



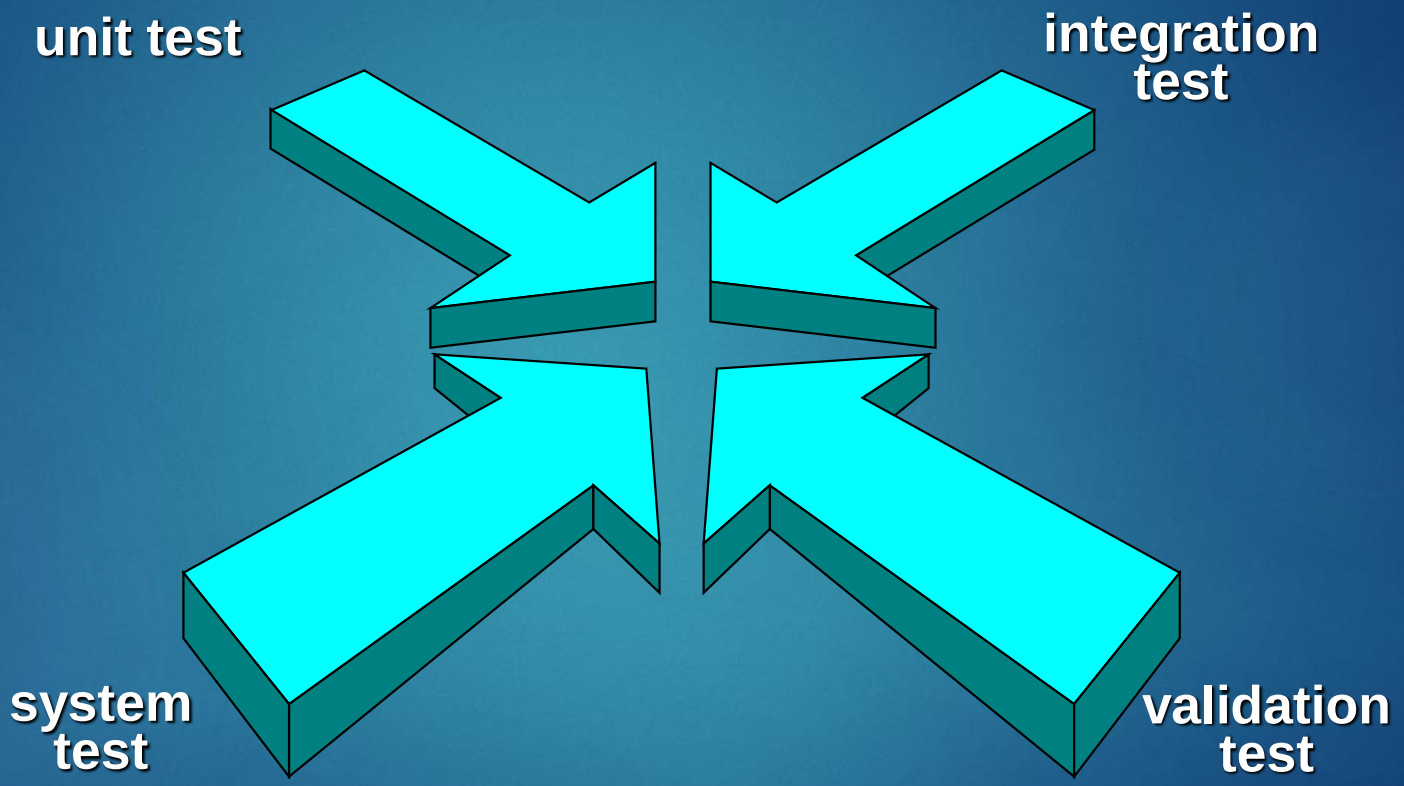
Testing Strategy

- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

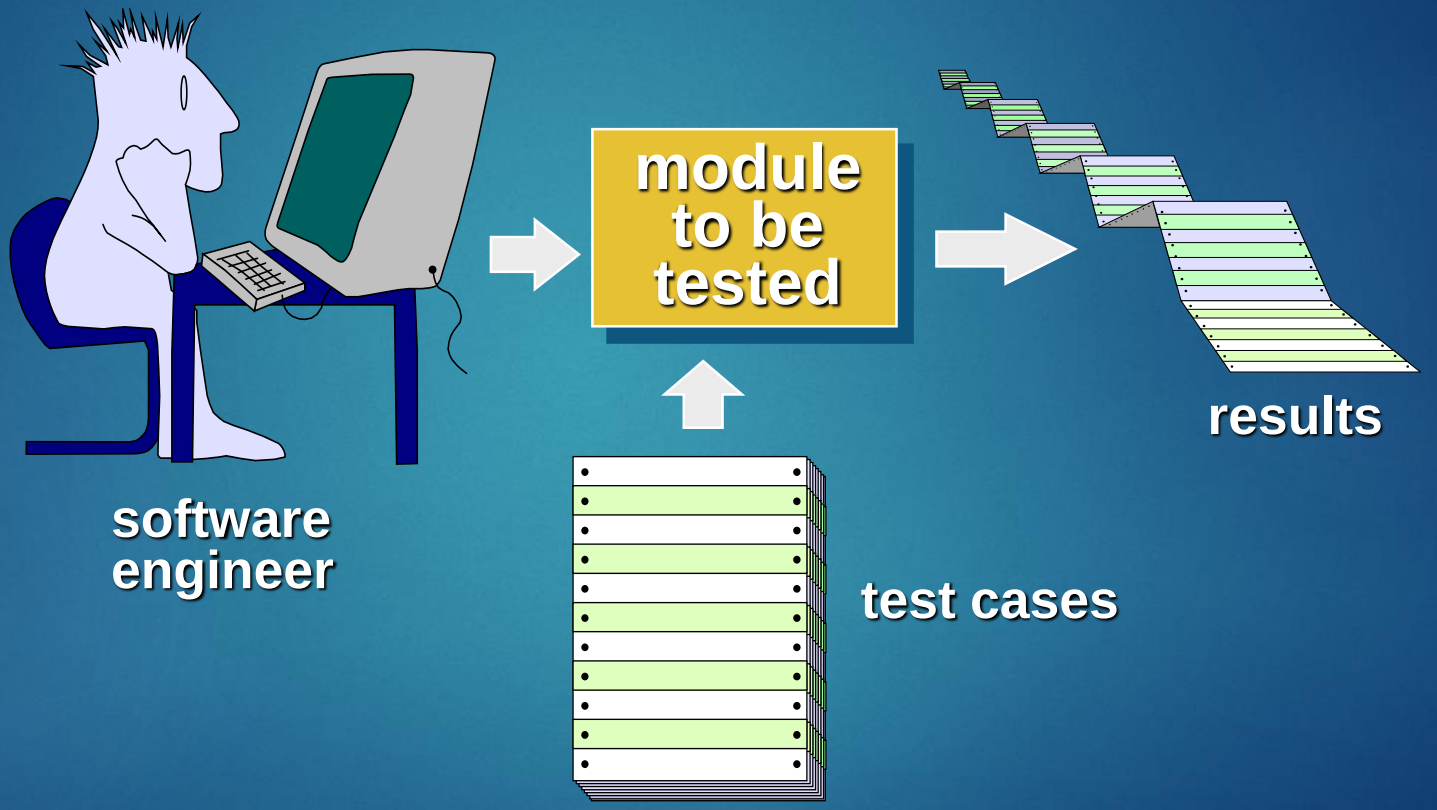
Strategic Issues

- Issues to be addressed for successful implementation of software testing strategy
  - State testing objectives explicitly.
  - Understand the users of the software and develop a profile for each user category.
  - Develop a testing plan that emphasizes “rapid cycle testing.”
  - Build “robust” software that is designed to test itself
  - Use effective formal technical reviews as a filter prior to testing
  - Conduct formal technical reviews to assess the test strategy and test cases themselves.
  - Develop a continuous improvement approach for the testing process.

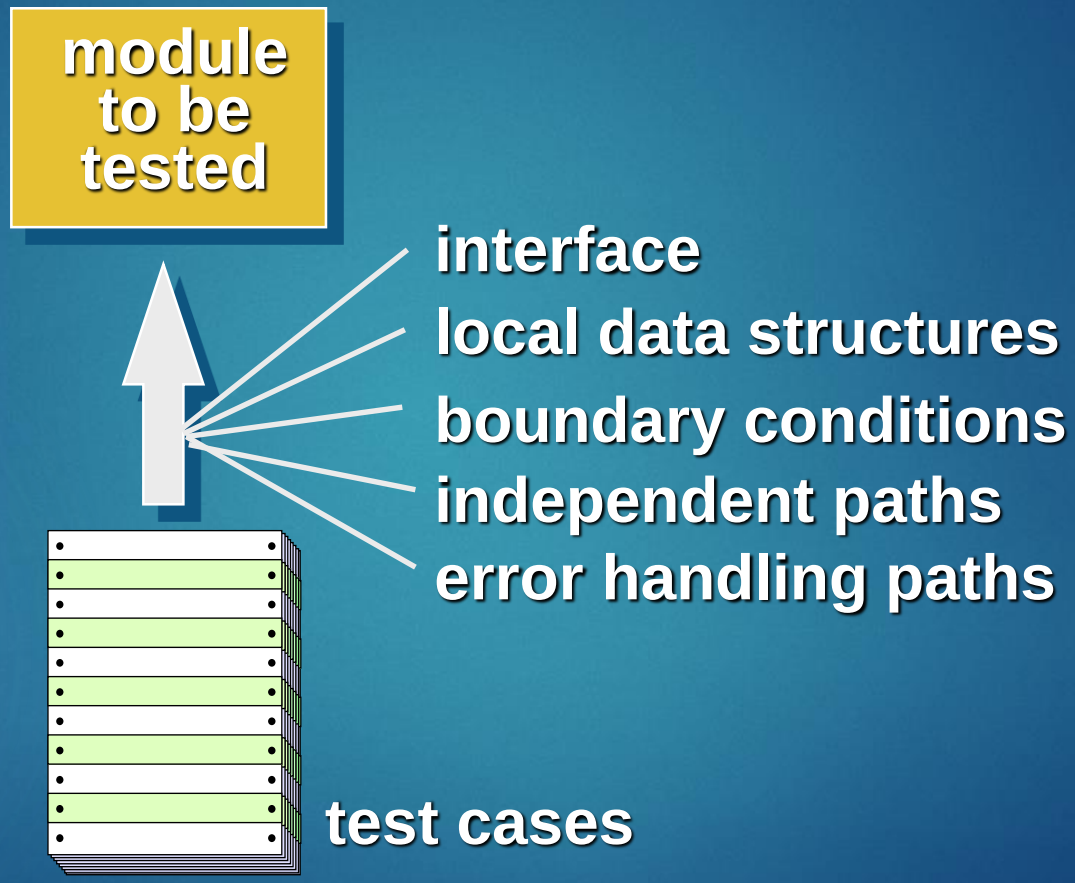
Testing Strategy



Unit Testing



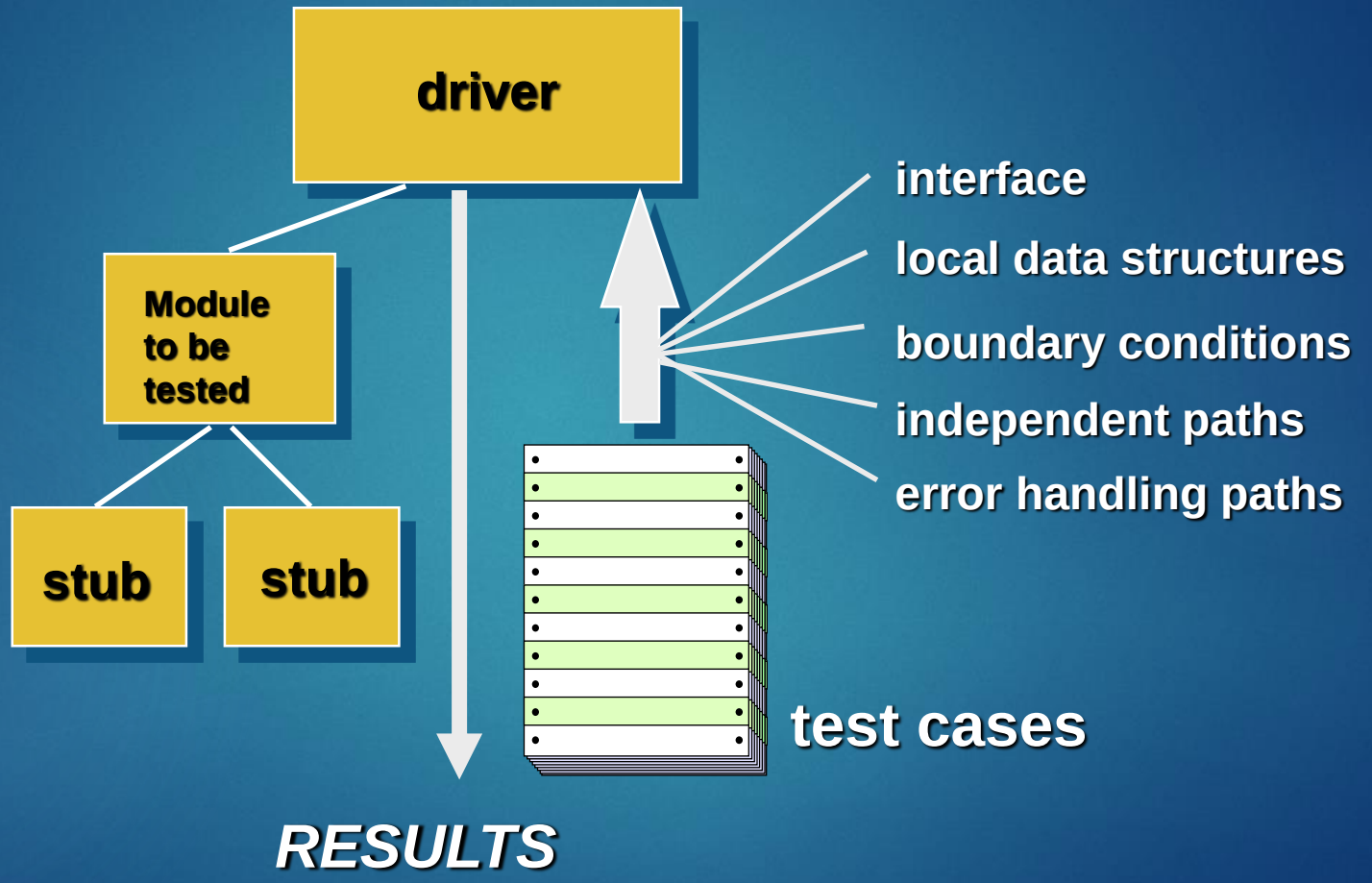
Unit Testing



Unit testing

- Common errors in computations are
  - Misunderstood or incorrect arithmetic precedence
  - Mixed mode operations
  - Incorrect initialization
  - Precision accuracy
  - Incorrect symbolic representation
- Test cases should uncover errors due to
  - Erroneous computations – incorrect logical operators or precedence
  - Incorrect comparisons - Comparison of different data types,
  - Improper control flow – non-existent loop termination, failure to exit in divergent iterations, improperly modified loop variables
  - Error handling – error description is intelligible, error noted should correspond to error encountered, error condition should not cause OS to intervene, exception handling should be managed, error description should provide cause of error

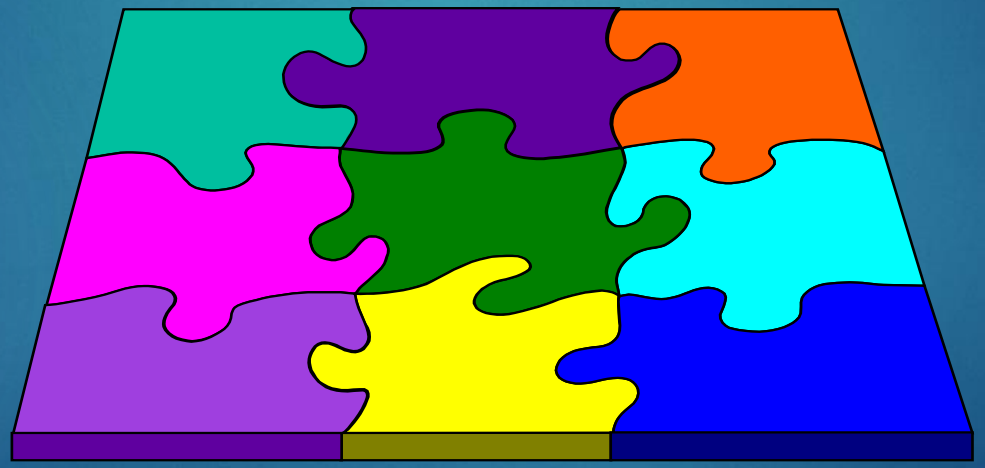
Unit Test Environment



Integration Testing Strategies

A systematic technique for constructing the software architecture while at the same conducing tests to uncover errors associated with interfacing.

Using unit tested components build a program structure dictated by design.



Integration Testing Strategies

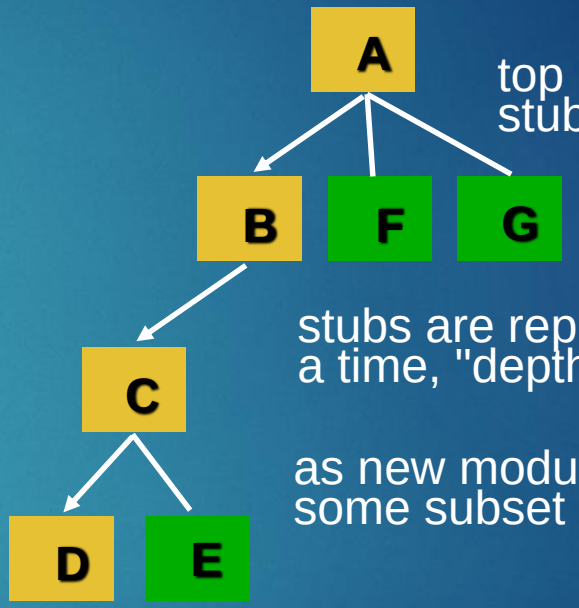
Approaches:

- the “big bang” approach – chaos

- an incremental integration
  - Modules are constructed and tested in small increments
  - Errors are easier to isolate and correct
  - Interfaces are tested
  - Approaches
    - ✓ Top down approach
      - Modules are integrated by moving downward through the control hierarchy
      - Begin with main control module
      - Subordinate modules are incorporated with depth first or breadth first integration
    - ✓ Bottom up approach

Top Down Integration

- Advantages
  - Verifies major control or decision points early in the test process
  - In depth first a complete functionality tested
  - Early demonstration of functionality
- Disadvantages
  - Stubs replace low-level modules, so no significance data can flow upward
    - Delay tests till stubs replace actual modules
    - Develop stubs that perform limited functions to simulate actual module
    - Integrate the software using bottom-up approach



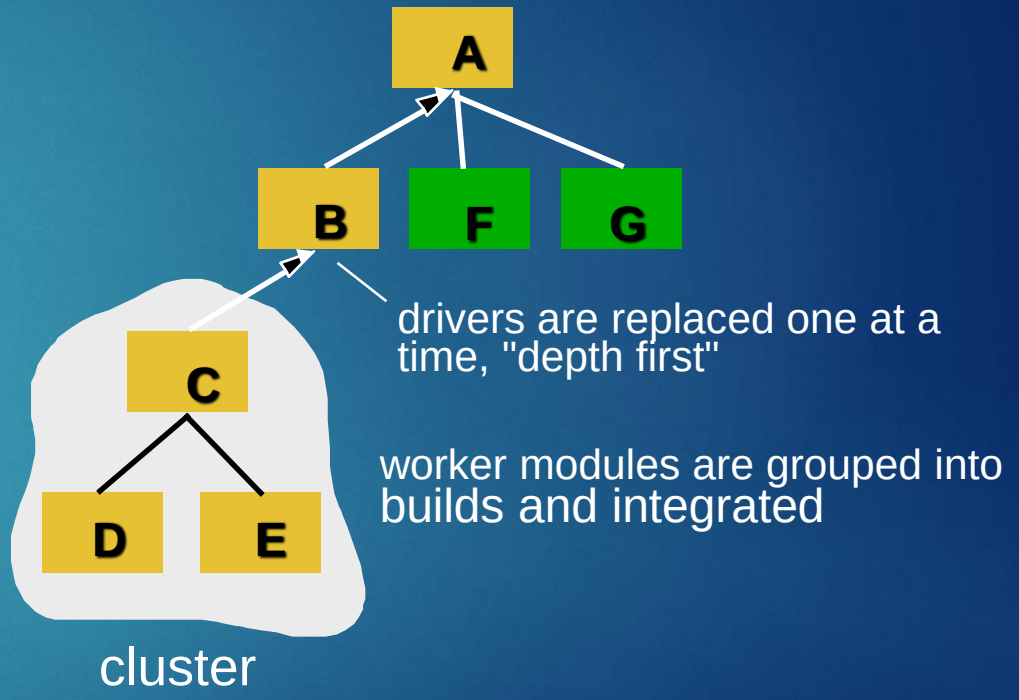
top module is tested with stubs

stubs are replaced one at a time, "depth first"

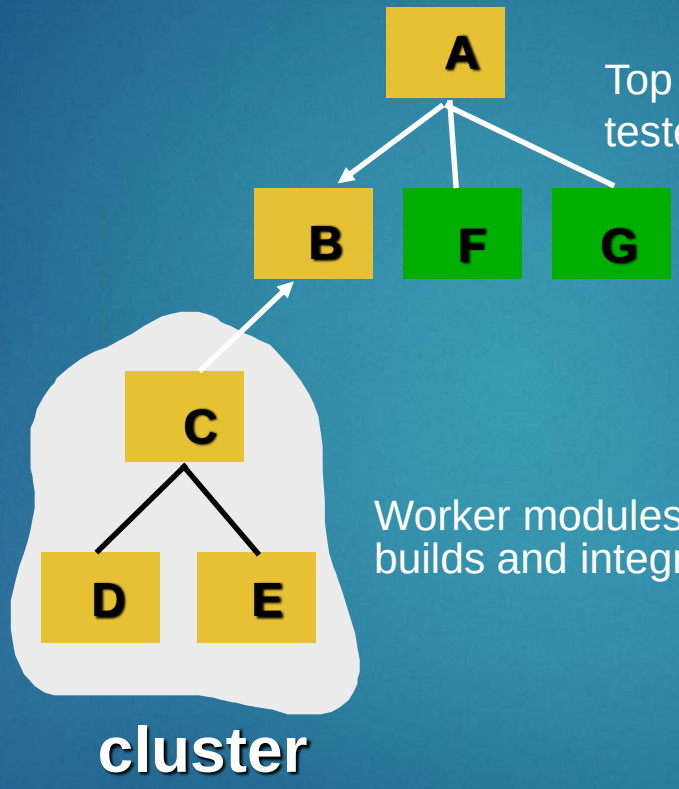
as new modules are integrated, some subset of tests is re-run

Bottom-Up Integration

- Advantages
  - Stubs not needed - Processing required for subordinate components is always available
  - Driver has limited functionality and simple to create



Sandwich Testing



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

Regression testing

- Testing after adding each module
- New data flow paths are established
- New I/o may occur
- New control logic is invoked
- Regression testing – re-execution of some subset of tests that have been done already to ensure that changes have not propagated unintended side effects.

Smoke Testing

- Is an integration testing approach that is commonly used when s/w products are being developed.
- Used in time critical projects and designed as a pacing mechanism allowing the s/w teams to assess its projects on frequent basis.
- A common approach for creating "daily builds" for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a "build."
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover "show stopper" errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

High Order Testing

- Validation testing
  - Focus is on software requirements
  - Alpha/Beta testing
  - Focus is on customer usage
- System testing
  - Focus is on system integration
  - Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Security testing
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
  - Stress testing
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
  - Performance Testing
  - test the run-time performance of software within the context of an integrated system. To test resource utilization, monitor execution levels of external instruments, log events like interrupts, state of machines etc. Uncover situations that can lead to degradation and possible system failure.

Elaborate from textbook

Testing for WebApps

1. The content model for the WebApp is reviewed to uncover errors.
2. The interface model is reviewed to ensure that all use cases can be accommodated.
3. The design model for the WebApp is reviewed to uncover navigation errors.
4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.
5. Each functional component is unit tested.
6. Navigation throughout the architecture is tested.
7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
9. Performance tests are conducted.
10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance

Software Testing Techniques

Software Testability

Characteristics of testable software:

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

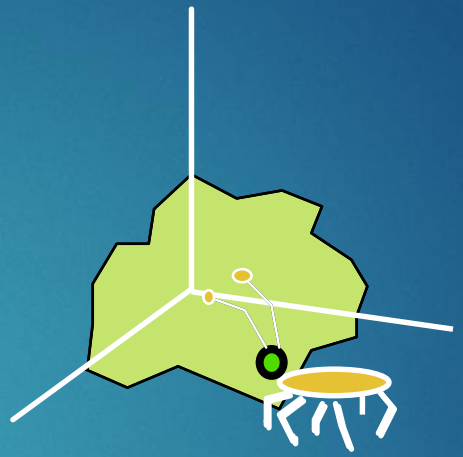
What is a "Good" Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

Test Case Design

"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer

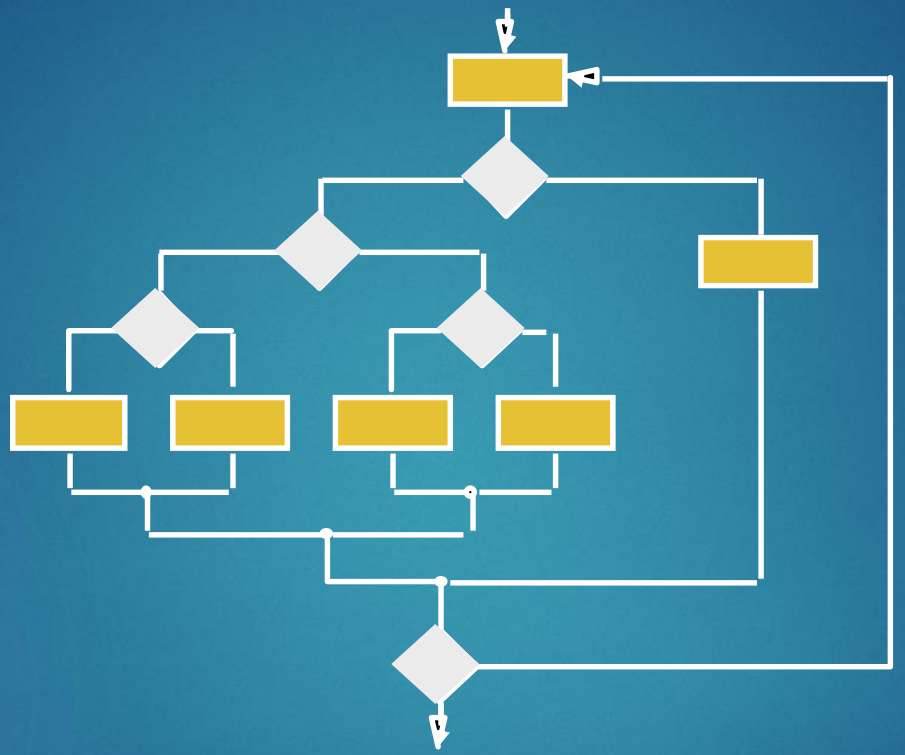


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

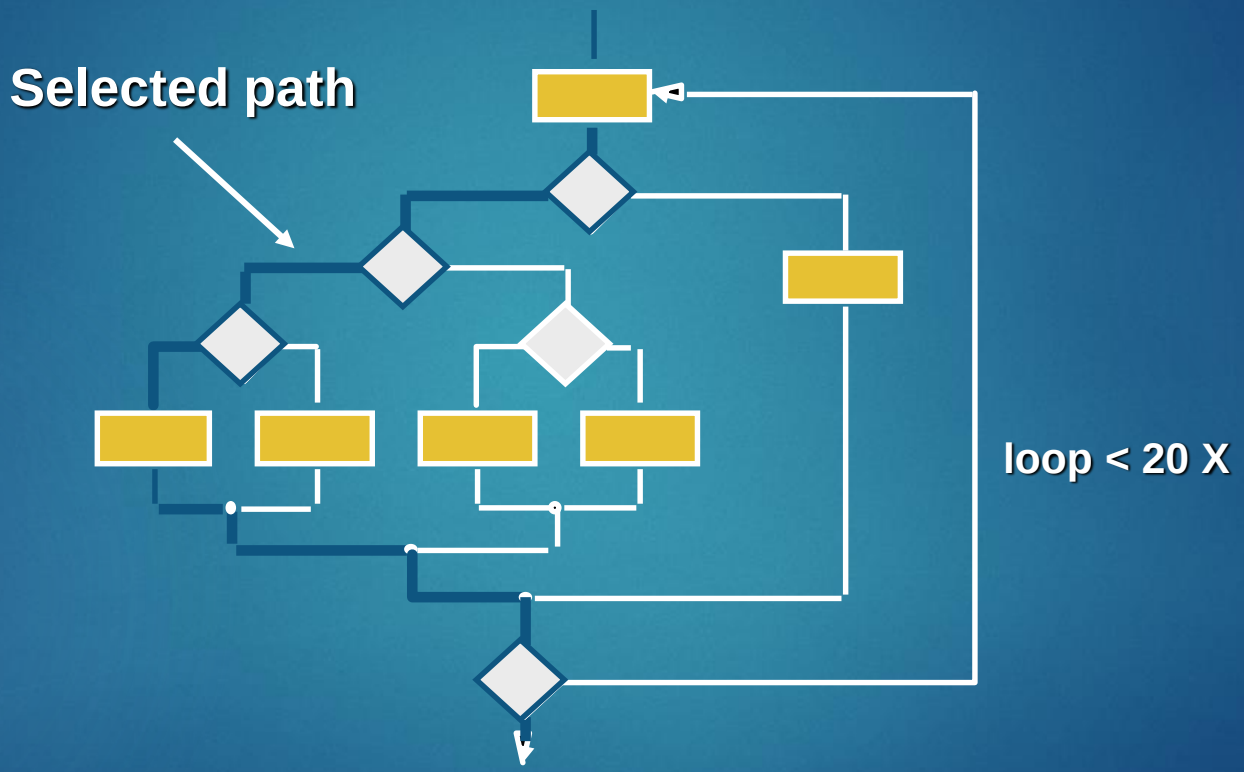
CONSTRAINT **with a minimum of effort and time**

Exhaustive Testing

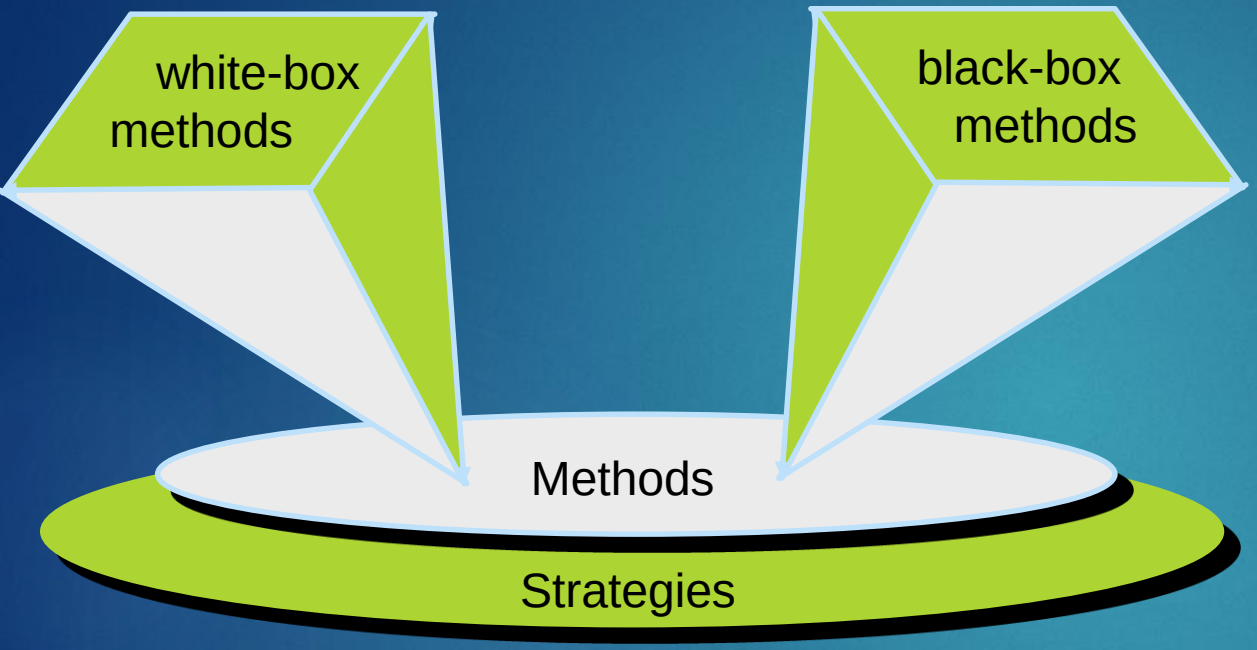


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



Software Testing



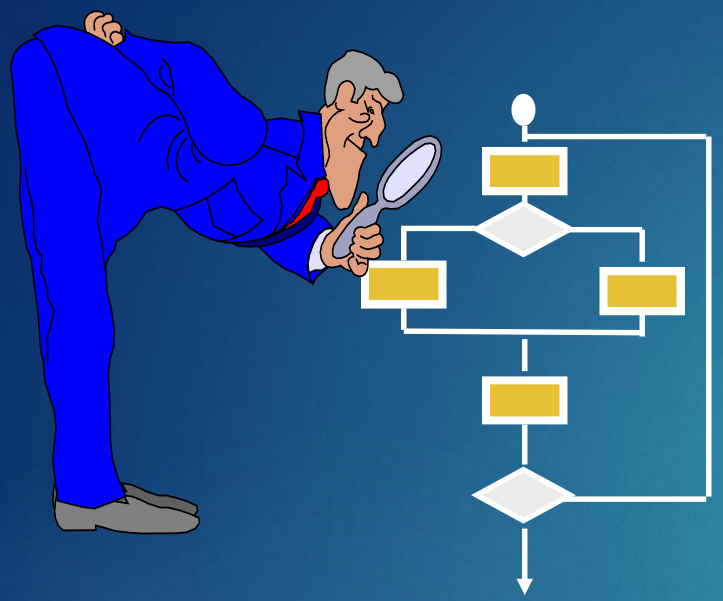
Two ways of testing a product:

1. Tests that demonstrate each function is fully operational
2. Tests that ensures all internal operations are performed according to specifications

Black-box testing alludes to tests that are conducted at the software interface. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software and collaborations between components are tested by exercising specific sets of conditions and/or loops.

White-Box Testing



White-box testing, sometimes called *glass-box testing,* is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.

Using white-box testing methods, you can derive test cases that

- Guarantee All independent paths are executed at least once
- Exercise All logical decisions are checked on their true and false sides
- Execute All loops are executed at their boundaries and within operational bounds
- Exercise internal data structures to ensure their validity

Basis path testing – enables the test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

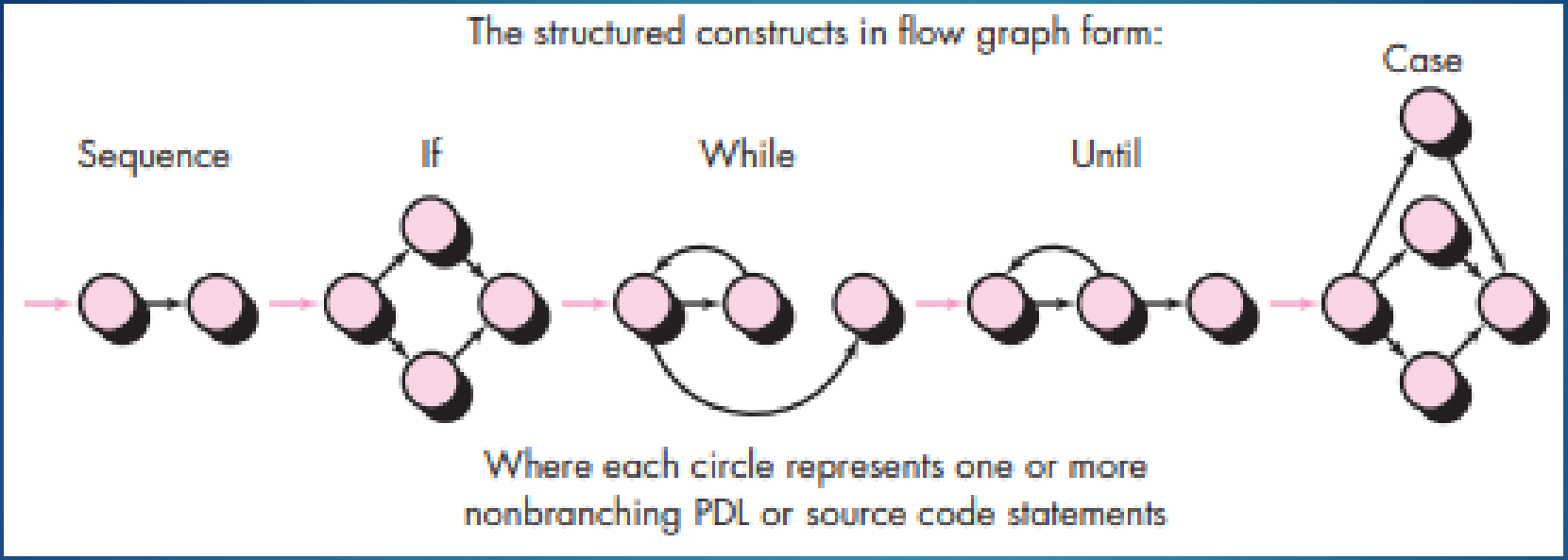
Control structure testing – enables the test case designer to broaden the testing coverage by considering variations on control structures and improve quality of white box testing.

BASIS PATH TESTING

- Flow graph notation
- Independent Program paths
- Deriving Test cases
- Graph Matrices

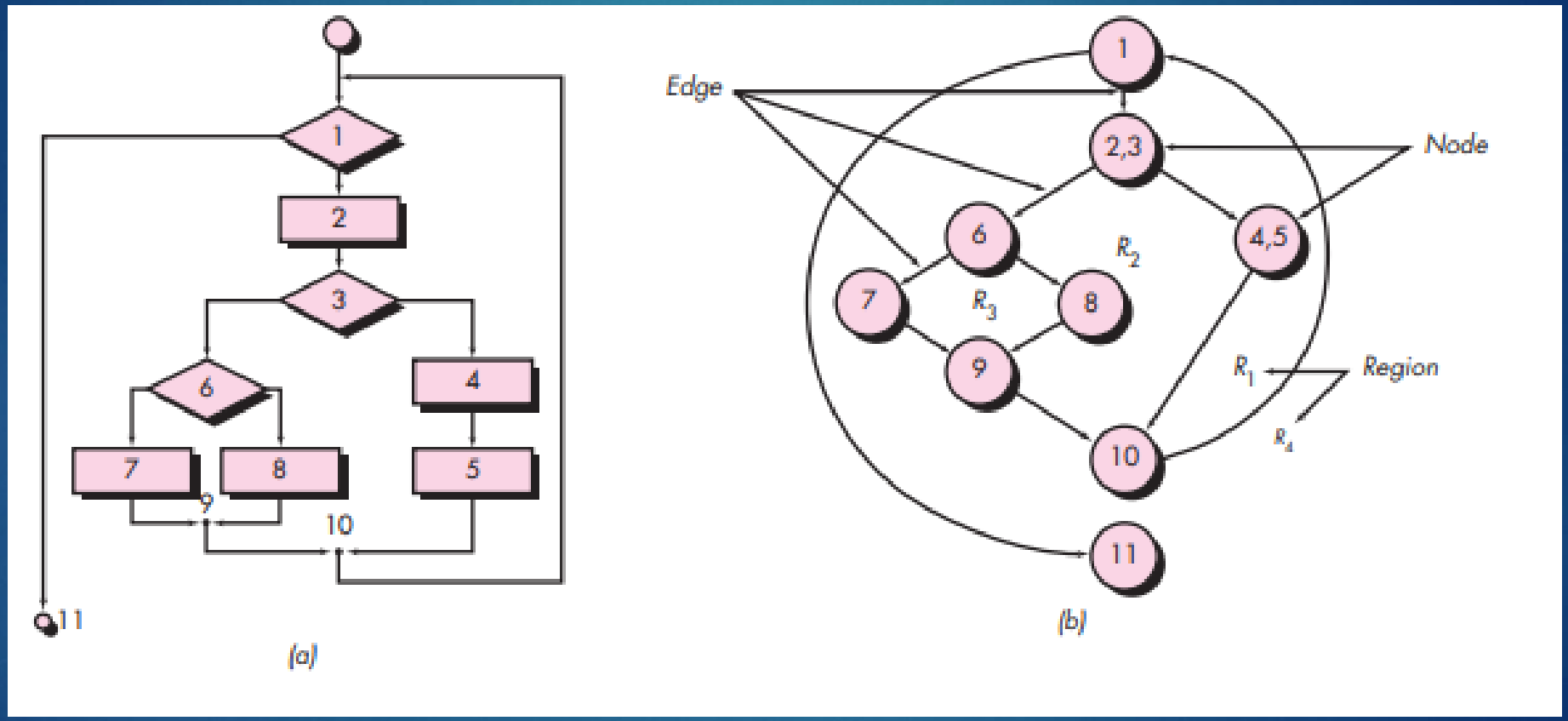
Flow graph notation

- Flow graph depicts logical control flow

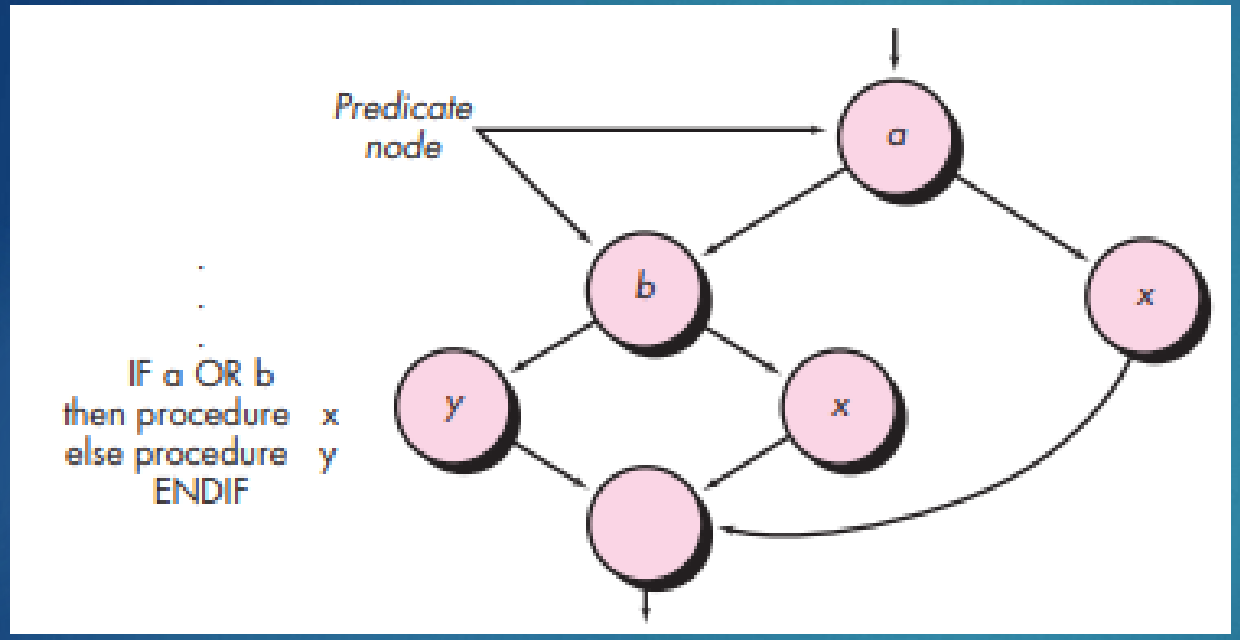


- Each node represents one or more procedural statements.
- A sequence of statements and decision diamonds can be mapped into a single node.
- The arrows are called as edges or links.

Flowchart and flow graph

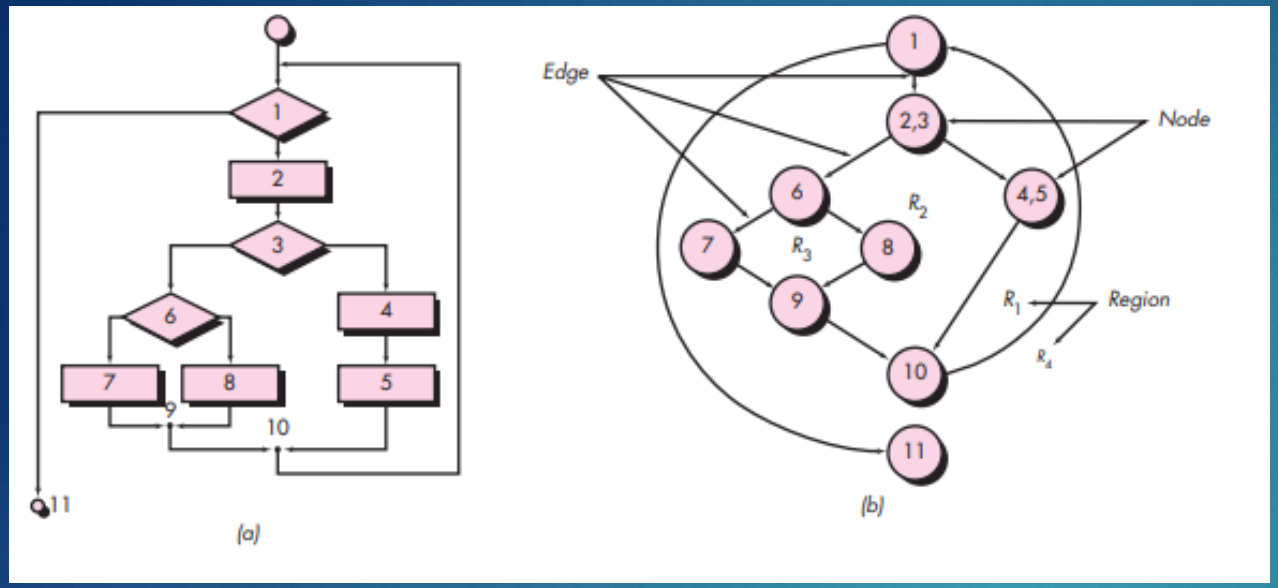


Compound logic



A compound condition occurs when one or more Boolean operators is present in a conditional statement

Independent Program Paths



An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition

Each new path introduces a new edge

Path 1: 1-11
Path 2: 1-2-3-4-5-10-1-11
Path 3: 1-2-3-6-8-9-10-1-11
Path 4: 1-2-3-6-7-9-10-1-11

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 ← Not an independent path

Cyclomatic Complexity

- How many paths to look for??

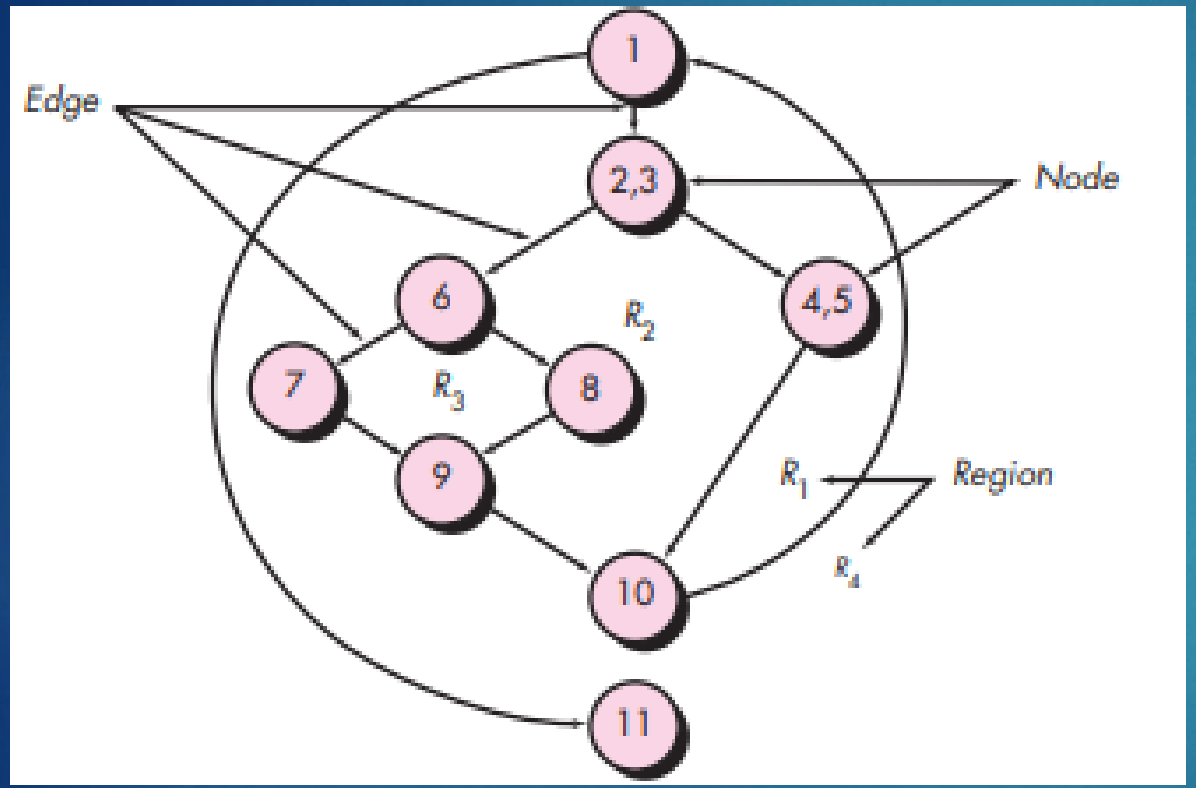
- Cyclomatic complexity – a software metric that provides a quantitative measure of the logical complexity of a program.

- In the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

- Computing cyclomatic complexity

1. V(G) = No. of regions
2. V(G) = E – N + 2 (E- no. of edges, N – no. of nodes)
3. V(G) = P + 1 (P – no. of predicate nodes)

Cyclomatic Complexity



1. **R = 4**
2. **V(G) = 11 edges – 9 nodes + 2 = 4**
3. **V(G) = 3 predicate nodes + 1 = 4**

Deriving Test Cases

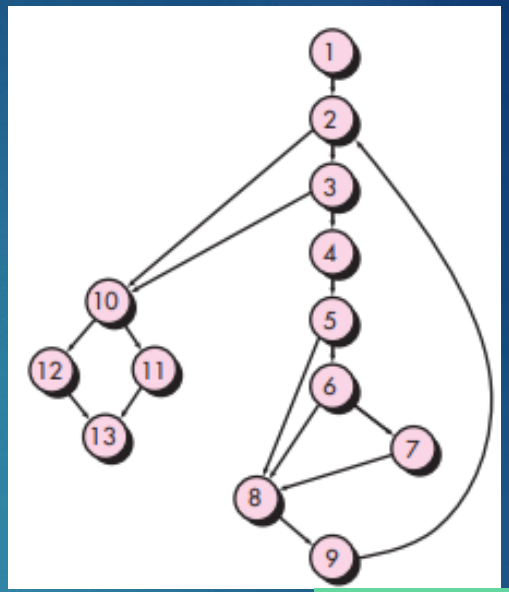
1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
3. Determine a basis set of linearly independent paths
4. Prepare test cases that will force execution of each path in the basis set.

```
PROCEDURE average;

      *   This procedure computes the average of 100 or fewer
          numbers that lie between bounding values; it also computes the
          sum and the total number valid.

      INTERFACE RETURNS average, total.input, total.valid;
      INTERFACE ACCEPTS value, minimum, maximum;

      TYPE value[1:100] IS SCALAR ARRAY;
      TYPE average, total.input, total.valid;
          minimum, maximum, sum IS SCALAR;
      TYPE i IS INTEGER;
1     i = 1;
      total.input = total.valid = 0;                    2
      sum = 0;
      DO WHILE value[i] <> -999 AND total.input < 100   3
        4  increment total.input by 1;
           IF value[i] > = minimum AND value[i] < = maximum   6
      5        THEN increment total.valid by 1;
      7                 sum = s sum + value[i]
               ELSE skip
      8     ENDIF
            increment i by 1;
      9  ENDDO
         IF total.valid > 0   10
        11  THEN average = sum / total.valid;
      12    ELSE average = -999;
      13  ENDIF
END average
```



Cyclomatic complexity

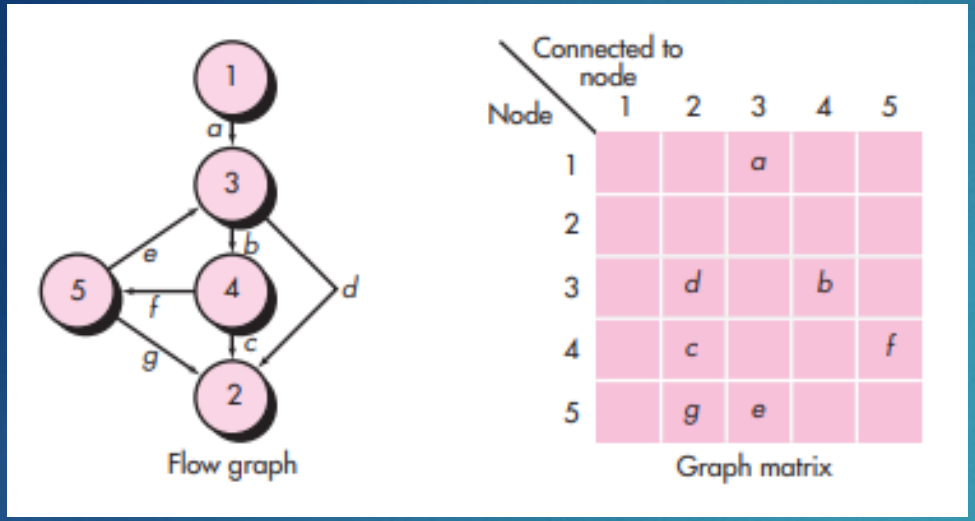
R = 6

V(G) = 17 – 13 + 2 = 6

V(G) = 5 + 1 = 6

Independent paths:
Path 1: 1-2-10-11-13
Path 2: 1-2-10-12-13
Path 3: 1-2-3-10-11-13
Path 4: 1-2-3-4-5-8-9-2-…
Path 5: 1-2-3-4-5-6-8-9-2-…
Path 6: 1-2-3-4-5-6-7-8-9-2-…

Graph Matrices



- A data structure, called a *graph matrix,* can be quite useful for developing a software tool that assists in basis path testing

- A square matrix whose size is equal to the number of nodes on the flow graph.
- Each row and column is a node
- Matrix entry corresponds to connections between nodes
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing
- Link weight can depict
  - Probability that a link will be executed
  - Processing time expended during traversal of link
  - Memory required during traversal of link
  - Resources required during traversal of link

Control Structure Testing

Although basis path testing is simple and highly effective, it is not sufficient in itself. Other variations on control structure testing are discussed. These broaden testing coverage and improve the quality of white-box testing.

- Condition testing
- Data flow testing
- Loop testing

Condition Testing

- Is a test case design method that exercises the logical conditions contained in a program module
- Simple condition – Boolean variable or relational expression
  - Boolean variable - Not operator
  - Relational expression

    E1<relational-operator>E2
- Compound condition – two or more simple conditions, Boolean operators and parentheses
- If a condition is incorrect then at least one component of condition is incorrect
- Types of errors in a condition
  - Boolean operator errors (incorrect/missing/extra operators)
  - Boolean variable errors
  - Boolean parenthesis errors
  - Relational operator errors
  - Arithmetic expression errors

Data Flow Testing

- selects test paths of a program according to the locations of definitions and uses of variables in the program
- assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables
- For a statement with *S* as its statement number

  DEF(S) = {X | statement S contains a definition of X}

  USE (S) = {X | statement S contains a use of X}
- The definition of variable X at statement S is said to be live at statement S' if there exists a path from statement S to statement S' that contains no other definition of X.
- If statement S is an if or loop statement, its DEF set is empty and its USE set is based on the condition of statement S.

Data Flow Testing

- A *definition-use (DU) chain* of variable X is of the form [X, S, S'] where S and S' are statement numbers, X is in DEF(S) and USE(S') and the definition of X in statement S is live at statement S'
- DU Testing Strategy: is to require that every DU chain be covered at least once.
- *Does not guarantee all branches are covered in the DU chain*
  - *E.g. in if-then-else statement, the then part has no definition and else part doesn't exist. In this situation, the else branch of the if statement is not necessarily covered by DU testing*

Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.

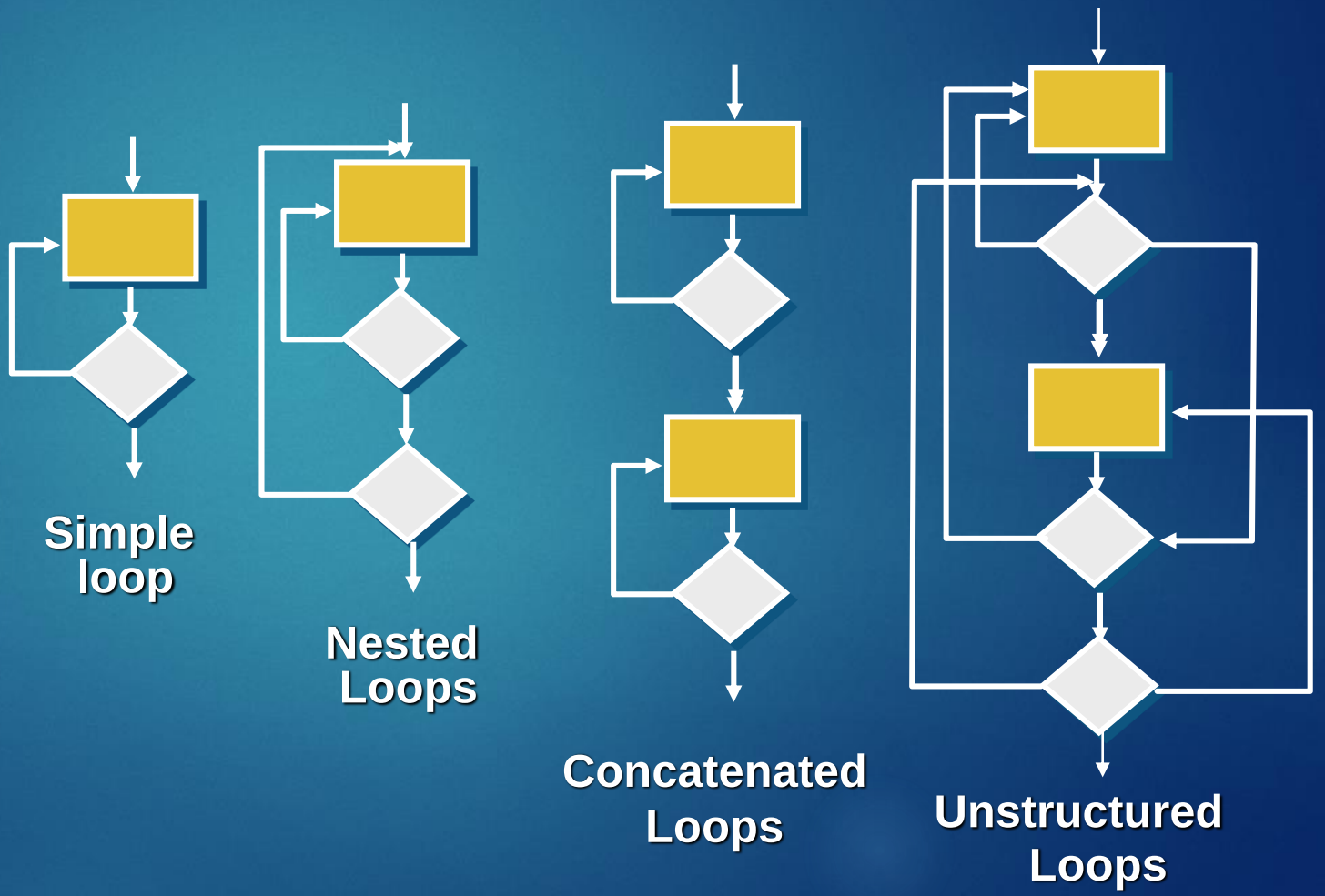
Four different classes of loops:

Simple loops,

concatenated loops,

nested loops, and

unstructured loops

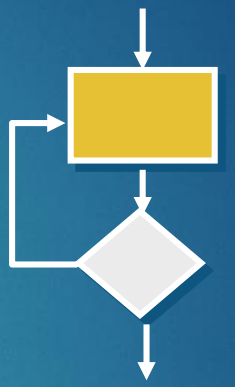


Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

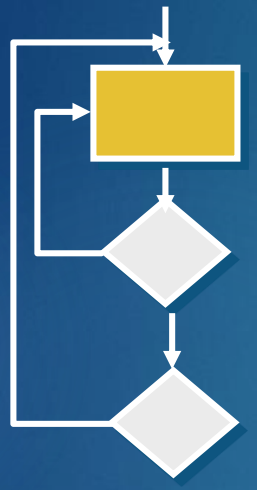


Simple loop

Nested Loops

1. Start at the innermost loop. Set all other loops to minimum values.

2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of-range or excluded values.



3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to “typical” values.

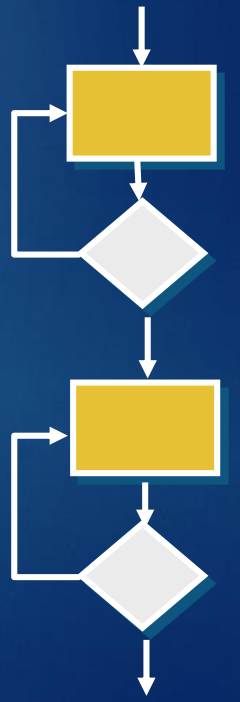
4. Continue until all loops have been tested.

Concatenated Loops

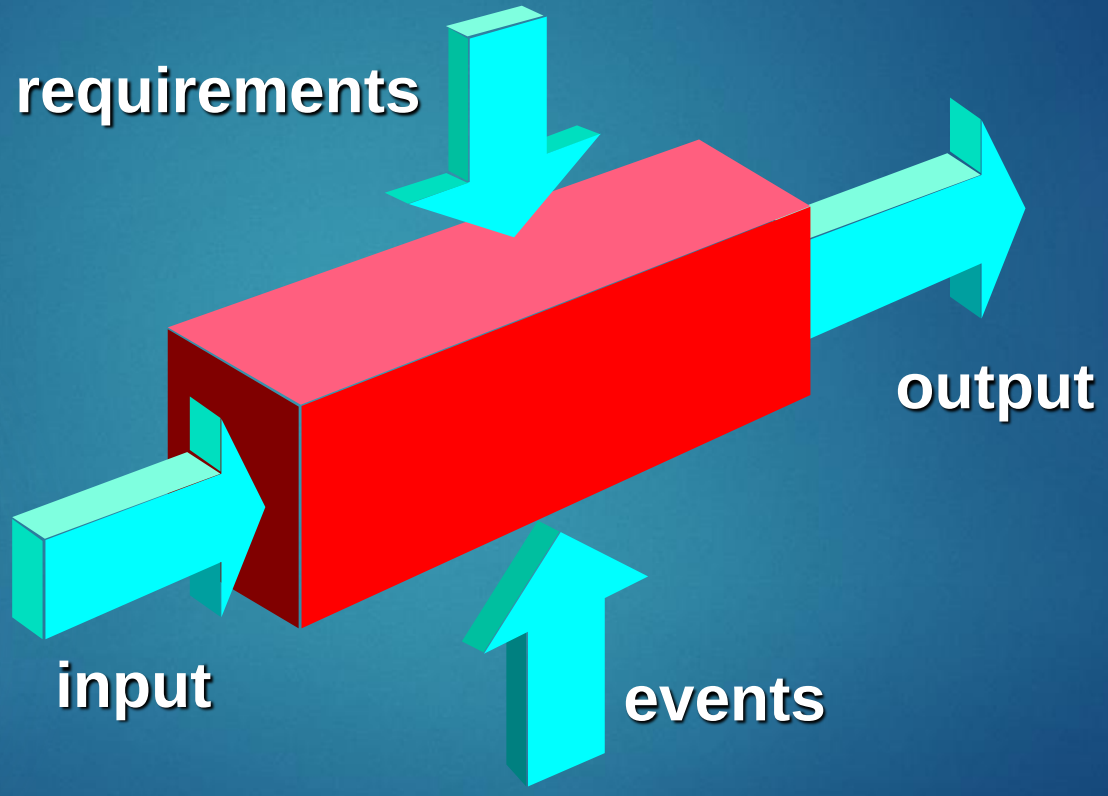
Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.

If two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.

When the loops are not independent, the approach applied to nested loops is recommended.



Black-Box Testing



Black-Box Testing

- Also called Behavioral testing
- Focuses on the functional requirements of the software
- Enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program
- Black box testing attempts to find errors in
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external data base access
  - Behaviour or performance errors
  - Initialization and termination errors

Black-Box Testing

- Tests are designed to answer the following
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?

- Benefits of Black-Box Testing: This technique helps to derive test cases that
  - Reduce the number of additional test cases that must be designed to achieve reasonable testing
  - Notifies about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Graph-Based Methods

To understand the objects that are modeled in software and the relationships that connect these objects

Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that each object and relationship is exercised and errors are uncovered.

In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.

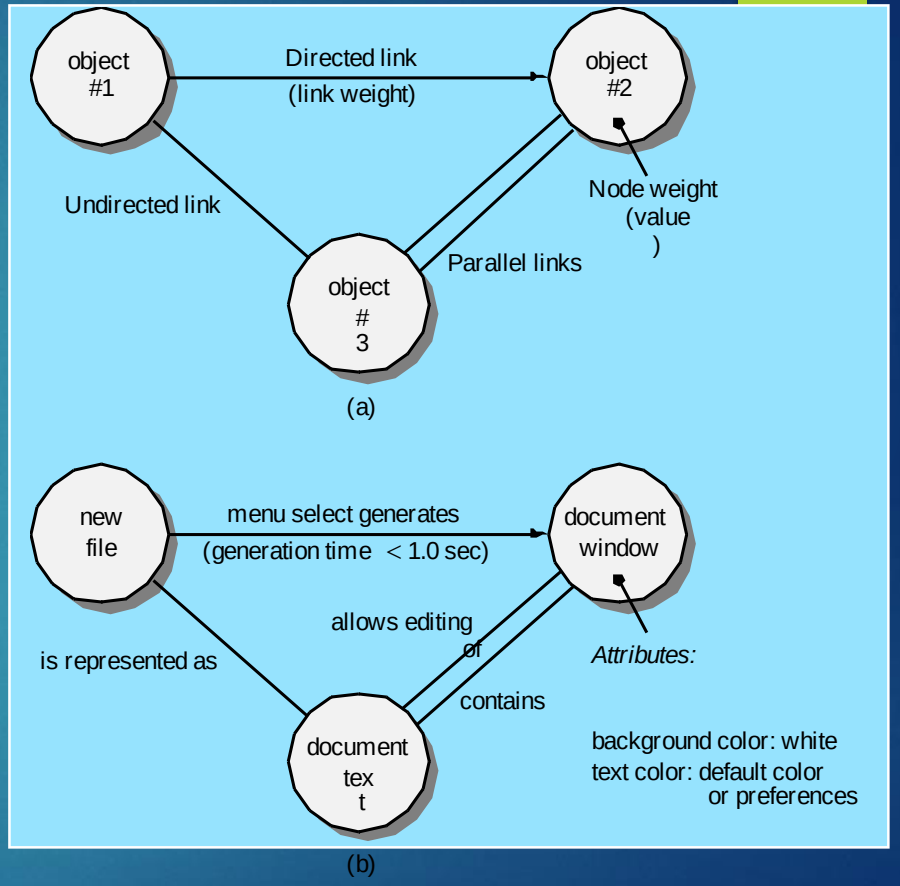
Nodes – objects

Links – relationships between objects

Node weights – properties of a node

Link weights – characteristics of a link

Directed link
Bidirectional link
Parallel links



Graph-Based Methods

- Software engineer derives test cases by traversing the graph
- Behavioral testing can make use of graphs
  - Transaction flow modelling – e.g. steps in airline reservation
  - Finite state modelling – e.g. phone order entry
  - Data flow modelling – e.g. transformation to translate one object into another
  - Timing modelling – e.g. sequential connections

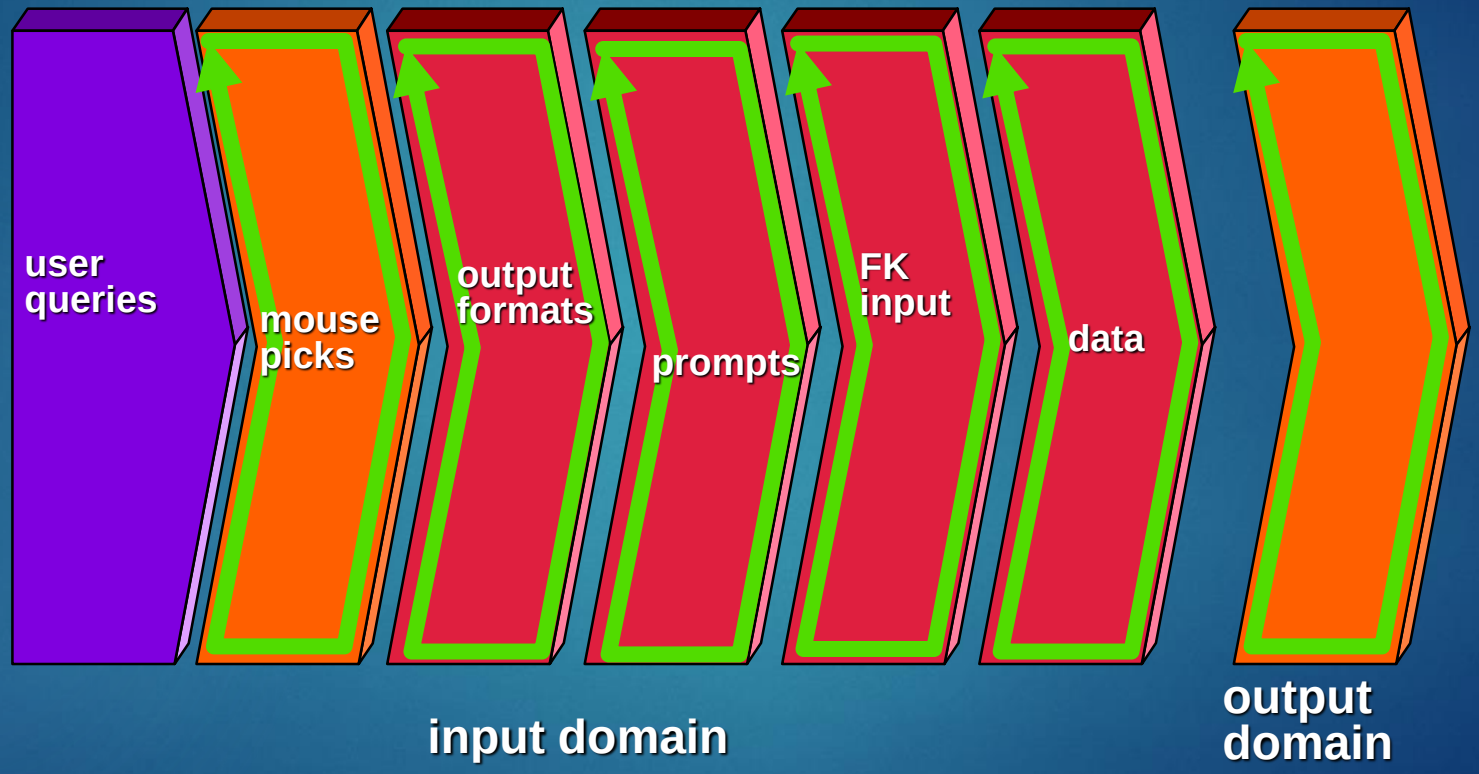
Equivalence Partitioning

- Dividing the test input data into a range of values and selecting one input value from each range is called **Equivalence Partitioning**.
- This is a black box test design technique used to calculate the effectiveness of test cases and which can be applied to all levels of testing from unit, integration, system testing and so forth.
- We cannot test all the possible input domain values, because if we attempted this, the number of test cases would be too large.
- In this method, input data is divided into different classes, each class representing the input criteria from the equivalence class. We then select one input from each class.
- This technique is used to reduce an infinite number of test cases to a finite number, while ensuring that the selected test cases are still effective test cases which will cover all possible scenarios.

Equivalence Partitioning

- Test-case design for equivalence partitioning is based on an evaluation of *equivalence classes* for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions. Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition
- Typically, an input condition is either a specific numeric value, a range of values, a set of related values, or a Boolean condition.
- Equivalence classes may be defined according to the following guidelines:
  1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.
  4. If an input condition is Boolean, one valid and one invalid class are defined.
- By applying the guidelines for the derivation of equivalence classes, test cases for each input domain data item can be developed and executed.
- Test cases are selected so that the largest number of attributes of an equivalence class are exercised at once.

Boundary Value Analysis



Boundary Value Analysis

- For the most part, errors are observed in the extreme ends of the input values, so these extreme values like start/end or lower/upper values are called Boundary values and analysis of these Boundary values is called "Boundary value analysis". It is also sometimes known as 'range checking'.
- Boundary value analysis is another black box test design technique and it is used to find the errors at boundaries of input domain rather than finding those errors in the center of input.
- Equivalence Partitioning and Boundary value analysis are linked to each other and can be used together at all levels of testing. Based on the edges of the equivalence classes, test cases can then be derived.
- Each boundary has a valid boundary value and an invalid boundary value. Test cases are designed based on the both valid and invalid boundary values. Typically, we choose one test case from each boundary.
- Finding defects using Boundary value analysis test design technique is very effective and it can be used at all test levels. You can select multiple test cases from valid and invalid input domains based on your needs or previous experience but remember you do have to select at least one test case from each input domain.

Boundary Value Analysis

- Boundary value analysis is a test-case design technique that complements equivalence partitioning.
- Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.
- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* and just above and just below *a* and *b*.

2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature versus pressure table is required as output from an engineering analysis program. Test cases should be designed to create an output report that produces the maximum (and minimum) allowable number of table entries.

4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Why Equivalence & Boundary Analysis Testing

- This testing is used to reduce very large number of test cases to manageable chunks.
- Very clear guidelines on determining test cases without compromising on the effectiveness of testing.
- Appropriate for calculation-intensive applications with large number of variables/inputs

Orthogonal Array Testing

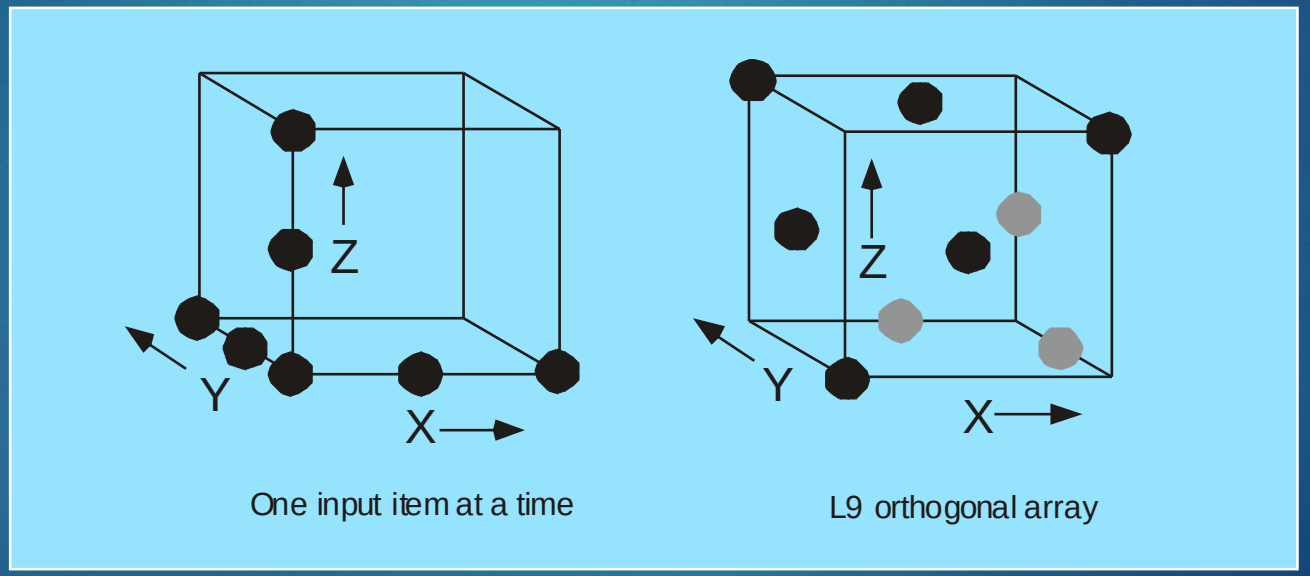
- In the present scenario, delivering a quality software product to the customer has become challenging due to the complexity of the code.
- In the conventional method, test suites include test cases that have been derived from all combination of input values and pre-conditions. As a result n number of test cases has to be covered.
- But in a real scenario, the testers won't have the leisure to execute all the test cases to uncover the defects as there are other processes such as documentations, suggestions, and feedbacks from the customer that has to be taken into account while in the testing phase.
- Hence, the test managers wanted to optimize the number and quality of the test cases to ensure maximum Test coverage with minimum effort. This effort is called Test Case Optimization.
- Let see below how Test Case Optimization can be achieved by OAT (Orthogonal Array Testing).

Orthogonal Array Testing

- Orthogonal Array Testing strategy is one of the Test Case Optimization techniques.
- Orthogonal Testing is a Black Box Testing - test cases optimization technique used when the system to be tested has huge data inputs.
- For example, when a train ticket has to be verified, the factors such as - the number of passengers, ticket number, seat numbers and the train numbers has to be tested, which becomes difficult when a tester verifies input one by one. Hence, it will be more efficient when he combines more inputs together and does testing. Here, we can use the Orthogonal Array testing method.
- This type of pairing or combining of inputs and testing the system to save time is called Pairwise testing. OATS technique is used for pairwise testing.

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



$$L_{Runs}(Levels^{Factors})$$

- Runs (N) – Number of rows in the array, which translates into a number of test cases that will be generated.
- Factors (K) – Number of columns in the array, which translates into a maximum number of variables that can be handled.
- Levels (V) – Maximum number of values that can be taken on any single factor.

Example 1
A Web page has three distinct sections (Top, Middle, Bottom) that can be individually shown or hidden from user
•No of Factors = 3 (Top, Middle, Bottom)
•No of Levels (Visibility) = 2 (Hidden or Shown)

If we go for Conventional testing technique, we need test cases like: 2 X 3 = 6 Test Cases
If we go for OAT Testing we need: 4 Test cases as shown below:

Test Cases	Scenarios	Values to be tested
Test #1	HIDDEN	Top
Test #2	SHOWN	Top
Test #3	HIDDEN	Bottom
Test #4	SHOWN	Bottom
Test #5	HIDDEN	Middle
Test #6	SHOWN	Middle

Conventional testing

Test Cases	TOP	Middle	Bottom
Test #1	Hidden	Hidden	Hidden
Test #2	Hidden	Visible	Visible
Test #3	Visible	Hidden	Visible
Test #4	Visible	Visible	Hidden

Orthogonal Array testing

- A microprocessor's functionality has to be tested:
- Temperature: 100C, 150C and 200C.
- Pressure : 2 psi,5psi and 8psi
- Doping Amount :4%,6% and 8%
- Deposition Rate : 0.1mg/s , 0.2 mg/s and 0.3mg/s

By using the Conventional method we need = 81 test cases to cover all the inputs. Let's work with the OATS method:

No. of factors = 4 (temperature, pressure, doping amount and Deposition rate)

Levels = 3 levels per factor (temperature has 3 levels-100C, 150C, and 200C and likewise other factors too have levels)

1. Columns with the No. of factors

Test case #	Temperature	Pressure	Doping amount	Deposition rate

2. Enter the number of rows is equal to levels per factor. i.e temperature has 3 levels. Hence, insert 3 rows for each level for temperature,

https://www.guru99.com/orthogonal-array-testing.html

3. Now split up the pressure, doping amount and the deposition rates in the columns.

For eg: Enter 2 psi across temperatures 100C,150C and 200C likewise enter doping amount 4% for 100C,150C and 200C and so on.

Test case #	Temperature	Pressure	Doping amount	Deposition rate
1	100C	2 psi	4%	0.1 mg/s
2	100C	5 psi	6%	0.2 mg/s
3	100C	8 psi	8%	0.3 mg/s
4	150C	2 psi	4%	0.1 mg/s
5	150C	5 psi	6%	0.2 mg/s
6	150C	8 psi	8%	0.3 mg/s
7	200C	2 psi	4%	0.1 mg/s
8	200C	5 psi	6%	0.2 mg/s
9	200C	8 psi	8%	0.3 mg/s

- **OAT Advantages**
  - Guarantees testing of the pair-wise combinations of all the selected variables.
  - Reduces the number of test cases
  - Creates fewer Test cases which cover the testing of all the combination of all variables.
  - A complex combination of the variables can be done.
  - Is simpler to generate and less error-prone than test sets created by hand.
  - It is useful for Integration Testing.
  - It improves productivity due to reduced test cycles and testing times.
- **OAT Disadvantages**
  - As the data inputs increase, the complexity of the Test case increases. As a result, manual effort and time spent increases. Hence, the testers have to go for Automation Testing.
  - Useful for Integration Testing of software components.
- **Mistakes or errors while performing OAT**
  - The testing effort should not be focused on the wrong area of the application.
  - Avoid picking the wrong parameters to combine
  - Avoid using Orthogonal Array Testing for minimal testing efforts.
  - Applying Orthogonal Array Testing manually
  - Applying Orthogonal Array Testing for high-risked applications

Test Driven Development