UNIT IV

Testing Conventional Applications: Software Testing Fundamentals, Internal and External Views of Testing, White-Box Testing, Basis Path Testing, Control Structure Testing, Black–Box Testing.
Software Testing Strategies: A Strategic Approach to Software Testing, Strategic Issues, Test Strategies for Conventional Software, Validation Testing, System Testing, The Art of Debugging.

Testing Conventional Applications:

SOFTWARE TESTING FUNDAMENTALS

The goal of testing is to find errors, and a good test is one that has a high probability of finding an error. The product should be testable. "Software testability is simply how easily a computer program can be tested." **Testability** exhibits following characteristics:-

a) **Operability:-** If a system is designed and implemented with quality in mind, relatively few bugs will block the execution of tests.
b) **Observability:-** "What you see is what you test." Inputs provided as part of testing produce distinct outputs.
c) **Controllability:-** "The better we can control the software, the more the testing can be automated and optimized." All possible outputs can be generated through some combination of input.
d) **Decomposability:-** By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.
e) **Simplicity:-** "The less there is to test, the more quickly we can test it." The program should exhibit functional simplicity; structural simplicity, and code simplicity.
f) **Stability:-** "The fewer the changes, the fewer the disruptions to testing."
g) **Understandability:-** "The more information we have, the smarter we will test."

Test Characteristics:- The following attributes of a "good" test:

i. A good test has a high probability of finding an error.
ii. A good test is not redundant. There is no point in conducting a test that has the same purpose as another test.
iii. A good test should be "best of breed".
iv. A good test should be neither too simple nor too complex.

INTERNAL AND EXTERNAL VIEWS OF TESTING

Any software can be tested in one of two ways:

(1) **Knowing the specified function that a product has been designed to perform**, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function.

(2) **Knowing the internal workings of a product,** that is, internal operations are performed according to specifications and all internal components have been adequately exercised.

The first test approach takes an **external view** and is called **black-box testing**. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

The second requires an **internal view** and is termed **white-box testing.** White-box testing of software is predicated on close examination of procedural detail. White-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results.

WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing. Our goal is to ensure that all statements and conditions have been executed at least once.

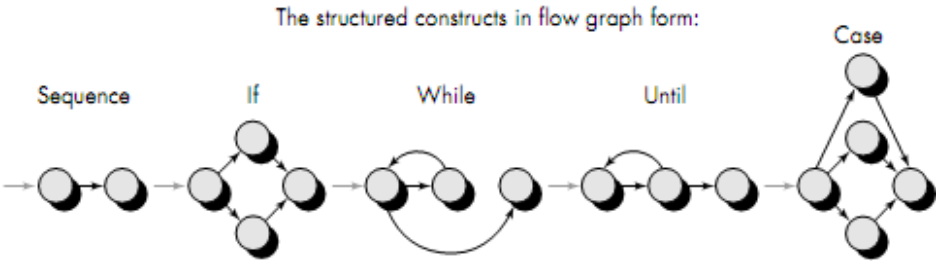
Using white-box testing methods, the software engineer can derive test cases that:

1. Guarantee that all independent paths within a module have been exercised at least once.
2. Exercise all logical decisions on their true and false sides.
3. Execute all loops at their boundaries and within their operational bounds.
4. Exercise internal data structures to ensure their validity.

BASIS PATH TESTING: Basis path testing is a white-box testing technique first proposed by Tom McCabe. Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

- **Flow Graph Notation:** Before the basis path method can be introduced, a simple notation for the representation of control flow, called a flow graph (or program graph) must be introduced.

FIGURE 17.1 Flow graph notation

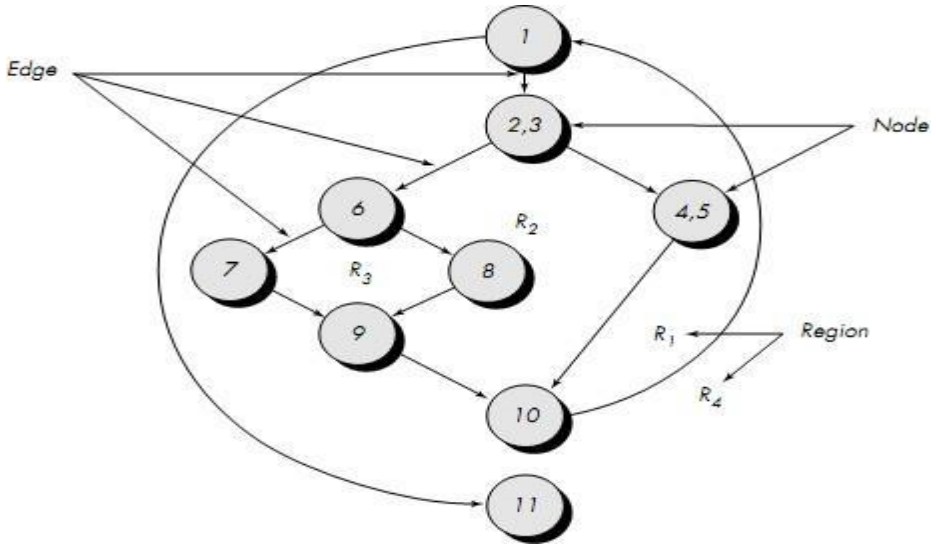


Each circle, called a flow graph node, represents one or more procedural statements. The arrows on the flow graph, called edges or links, represent flow of control. Each node that contains a condition is called a **predicate node** and is characterized by two or more edges emanating from it.

Cyclomatic Complexity: Cyclomatic complexity is 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 us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

An independent path is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a **flow graph,** an independent path must move along at least one edge that has not been traversed before the path is defined.



In the above fig the set of independent paths are as follows:

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

Note that each new path introduces a new edge.

The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

Is not considered an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Paths 1, 2, 3, and 4 constitute a basis set for the above flow graph

Cyclomatic complexity

1. The number of regions of the flow graph corresponds to the Cyclomatic complexity.
2. Cyclomatic complexity, V(G), for a flow graph, G, is defined as V(G) = E - N + 2 Where E is the number of flow graph edges, N is the number of flow graph nodes.
3. Cyclomatic complexity, V (G), for a flow graph, G, is also defined as V(G) =

P + 1 Where P is the number of predicate nodes contained in the flow graph G.

Referring once more to the above flow graph the Cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. V(G) = 11 edges 9 nodes + 2 = 4.

3. V(G) = 3 predicate nodes + 1 = 4.cyclomatic complexity for above flow graph.

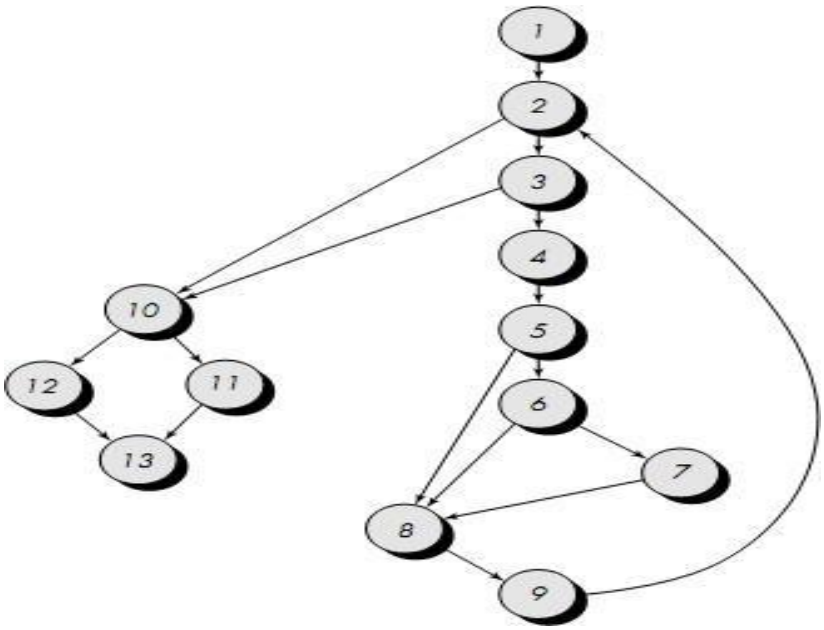
Deriving Test Cases

FIGURE 17.4 PDL for test case design with nodes identified

```
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
```

Using the above PDL let us describe how to derive the test cases The following steps are used to derive the set of 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.

V(G) = 6 regions
V(G) = 17 edges -13 nodes + 2 = 6
V(G) = 5 predicate nodes + 1 = 6

3. **Determine a basis set of linearly independent paths**.

The value of V(G) provides the number of linearly independent paths through the program control structure. In the case of procedure average, we expect to specify six 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-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes

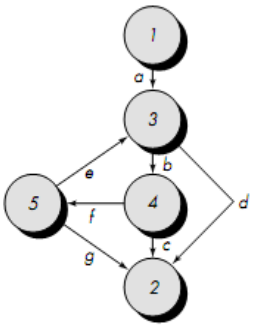
4.Prepare test cases that will force execution of each path in the basis set.

Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

Graph Matrices: To develop a software tool that assists in basis path testing, a data structure, called a graph matrix, can be quite useful. A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between

nodes. A simple example of a flow graph and its corresponding graph matrix is shown in below figure.

FIGURE 17.6
Graph matrix



Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

FIGURE 17.7
Connection matrix

Node \ Connected to node	1	2	3	4	5	Connections
1			1			1 – 1 = 0
2						
3		1		1		2 – 1 = 1
4		1			1	2 – 1 = 1
5		1	1			2 – 1 = 1

Graph matrix

3 + 1 = 4 ← Cyclomatic complexity

What is a graph matrix and how do we extend it use for testing? The graph matrix is nothing more than a tabular representation of a flow graph. By adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). Represented in this form, the graph matrix is called a connection matrix. Referring to above figure each row with two or more entries represents a predicate node.

Performing the arithmetic shown to the right of the connection matrix provides us with still another method for determining Cyclomatic complexity.

CONTROL STRUCTURE TESTING: It broadens testing coverage and improve quality of white-box testing

Condition testing is a test case design method that exercises the logical conditions contained in a program module. A simple condition is a Boolean variable or a relational expression, possibly preceded with one NOT (¬) operator. A relational expression takes the form

E1 <relational-operator> E2

where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following: <, ≤, =, ≠ (nonequality), >, or ≥.

A compound condition is composed of two or more simple conditions, Boolean operators, and parentheses. A condition without relational expressions is referred to as a Boolean expression.

Types of errors in a condition include the following:

• Boolean operator error (incorrect/missing/extra Boolean operators), Boolean variable error, Boolean parenthesis error, Relational operator error and Arithmetic expression error.

The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program

Data Flow Testing: The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program.

To illustrate the data flow testing approach, 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}

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. 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.

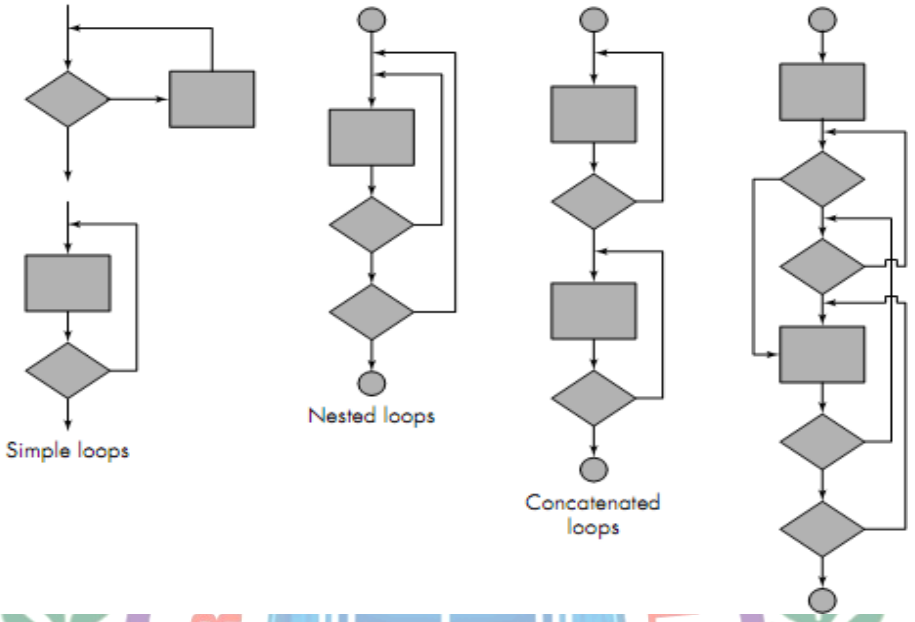
A **definition-use (DU) chain** of variable X is of the form [X, S, S'], where S and S' are statement numbers, X is in DEF(S) and USE(S'), and the definition of X in statement S is live at statement S'. One simple data flow testing strategy is to require that every DU chain be covered at least once.

MATRUSRI ENGINEERING COLLEGE HYDERABAD
Sponsored by MATRUSRI EDUCATION SOCIETY ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA

We refer to this strategy as the **DU testing strategy.**

Loop Testing: It is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops (Figure 17.8)

FIGURE 17.8
Classes of loops



1. **Simple loops:** The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
    1. Skip the loop entirely.
    2. Only one pass through the loop.
    3. Two passes through the loop.
    4. m passes through the loop where m < n.
    5. n - 1, n, n + 1 passes through the loop.
2. **Nested loops:** If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. Beizer suggests an approach that will help to reduce the number of tests:

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.
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.

3. **Concatenated loops:** Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other. However, 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.
4. **Unstructured loops:** Whenever possible, this class of loops should be redesigned to reflect the use of

the structured programming constructs.

BLACK-BOX TESTING

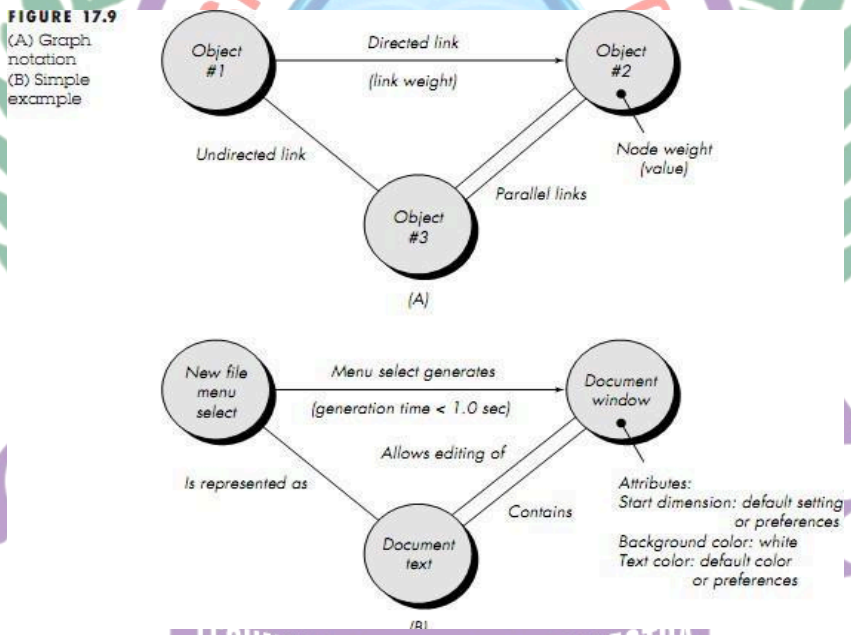
Black-box testing, also called behavioral testing, focuses on the functional requirements of the software. Black -box testing enables the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

Black-box testing attempts to find errors in the following categories: (1) incorrect or missing functions,

(1) interface errors, (3) errors in data structures or external data base access, (4) behavior or performance errors, and (5) initialization and termination errors.

***White-box testing is performed early in the testing process; Black-box testing tends to be applied* during later stages of testing.**

GRAPH-BASED TESTING METHODS: 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.



Collection of nodes that represent **objects;** links that represent the **relationships between objects;** node weights that describe the **properties of a node** (e.g., a specific data value or state behavior); and link weights that describe some **characteristic of a link**.

A **directed link** (represented by an arrow) indicates that a relationship moves in only one direction.
A bidirectional link, also called a symmetric link, implies that the relationship applies in both directions.
Parallel links are used when a number of different relationships are established between graph nodes.

Referring to the figure, a menu select on **new file** generates a **document window.** The node weight of **document window** provides a list of the window attributes that are to be expected when the

window is generated. The link weight indicates that the window must be generated in less than 1.0 second. An undirected link establishes a symmetric relationship between the **new file menu select** and **document text,** and parallel links indicate relationships between **document window** and **document text.**

The software engineer then derives test cases by traversing the graph and covering each of the relationships shown. These test cases are designed in an attempt to find errors in any of the relationships.

EQUIVALENCE PARTITIONING: It is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived. Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition.

If a set of objects can be linked by relationships that are symmetric, transitive, and reflexive, an **equivalence class** is present. An **equivalence class** represents a set of valid or invalid states for input conditions.

Typically, an input condition is 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.

Test cases are selected so that largest numbers of attributes of an equivalence class are exercised at once.

BOUNDARY VALUE ANALYSIS: Number of errors tends to occur at the boundaries of the input domain rather than in the "center". **Boundary value analysis** leads to a selection of test cases that exercise bounding values.

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

How do I create BVA test cases?

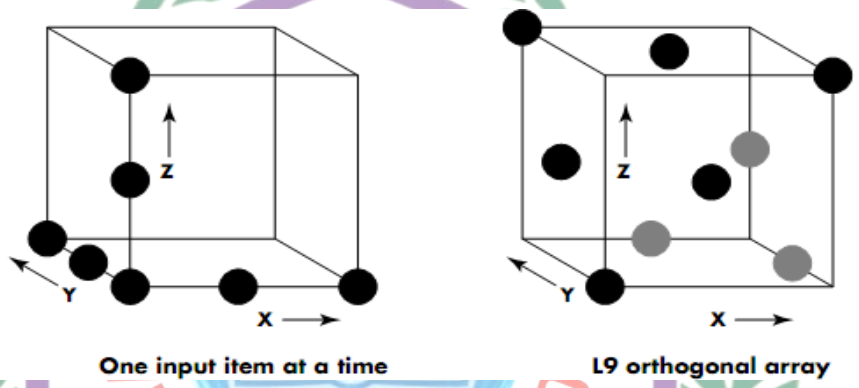
1. If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b and just above and just below a and b.
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.
3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature vs. 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., an array has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

MATRUSRI ENGINEERING COLLEGE HYDERABAD
Sponsored by MATRUSRI EDUCATION SOCIETY ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA

ORTHOGONAL ARRAY TESTING: The orthogonal array testing method is particularly useful in finding errors associated with region faults—an error category associated with faulty logic within a software component.

To illustrate the difference between orthogonal array testing and more conventional "one input item at a time" approaches, consider a system that has three input items, X, Y, and Z. Each of these input items has three discrete values associated with it. There are 33 = 27 possible test cases. Phadke suggests a geometric view of the possible test cases associated with X, Y, and Z illustrated in Figure. Referring to the figure, one input item at a time may be varied in sequence along each input axis. This results in relatively limited coverage of the input domain (represented by the left-hand cube in the figure).

FIGURE 17.10 A geometric view of test cases [PHA97]



When orthogonal array testing occurs, an L9 orthogonal array of test cases is created. The L9 orthogonal array has a balancing property that is; test cases (represented by black dots in the figure) are "dispersed uniformly throughout the test domain,"

To illustrate the use of the L9 orthogonal array, consider the send function for a fax application. Four parameters, P1, P2, P3, and P4, are passed to the send function. Each takes on three discrete values. For example, P1 takes on values:

P1 = 1, send it now
P1 = 2, send it one hour later P1 = 3, send it after midnight
P2, P3, and P4 would also take on values of 1, 2 and 3, signifying other send functions.

Sponsored by
MATRUSRI EDUCATION SOCIETY
ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA

If a "one input item at a time" testing strategy were chosen, the following sequence of tests (P1, P2, P3, P4) would be specified: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1,1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1),
(1, 1, 1, 2), and (1, 1, 1, 3).

Phadke assesses these test cases in the following manner:

FIGURE 17.11 An L9 orthogonal array

Test case	Test parameters			
	P_1	P_2	P_3	P_4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3

SOFTWARE TESTING STRATEGIES

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user. Testing shows errors, requirements conformance, performance & indication of quality.

A STRATEGIC APPROACH TO SOFTWARE TESTING:

Test Strategy incorporates test planning, test case design, test execution and resultant data collection and execution.

All provide the software developer with a template for testing and all have the following **generic characteristics:**

- Perform Formal Technical reviews(FTR) to uncover errors during software development
- Begin testing at component level and move outward to integration of entire component based system.
- Adopt testing techniques relevant to stages of testing
- Testing can be done by software developer and independent testing group
- Testing and debugging are different activities. Debugging follows testing

Verification & Validation

Verification refers to the set of activities that ensure that software correctly implements a specific function.

Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.

Boehm [BOE81] states this way:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

MATRUSRI ENGINEERING COLLEGE HYDERABAD
Sponsored by MATRUSRI EDUCATION SOCIETY ESTD 1980
TECHNOLOGY TRAVERSES TERRESTRA

The definition of V&V encompasses many of the activities that we have referred to as software quality assurance (SQA). Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing. Although testing plays an extremely important role in V&V, many other activities are also necessary.

Organizing for Software Testing: The people who have built the software are now asked to test the software. This seems harmless in itself; after all, who knows the program better than its developers do? Unfortunately, these same developers have a vested interest in demonstrating that the program is error free, that it works according to customer requirements.

From a psychological point of view, software analysis and design (along with coding) are constructive tasks. From the point of view of the builder, testing can be considered to be (psychologically) destructive.

The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function for which it was designed. In many cases, the developer also conducts integration testing—a testing step that leads to the construction (and test) of the complete program structure. Only after the software architecture is complete does an independent test group become involved.

The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present.

However, the software engineer does not turn the program over to ITG and walk way. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

The ITG is part of the software development project team in the sense that it becomes involved during the specification activity and stays involved (planning and specifying test procedures) throughout a large project.

A Software Testing Strategy for Conventional Software Architecture:

The software engineering process may be viewed as the spiral illustrated in Figure18.1. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established.

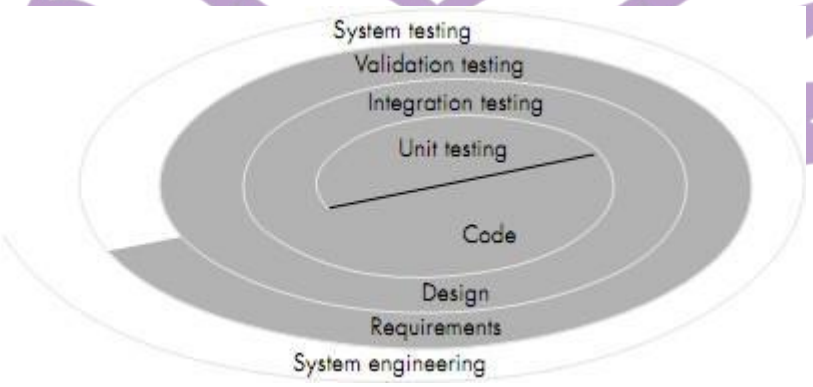


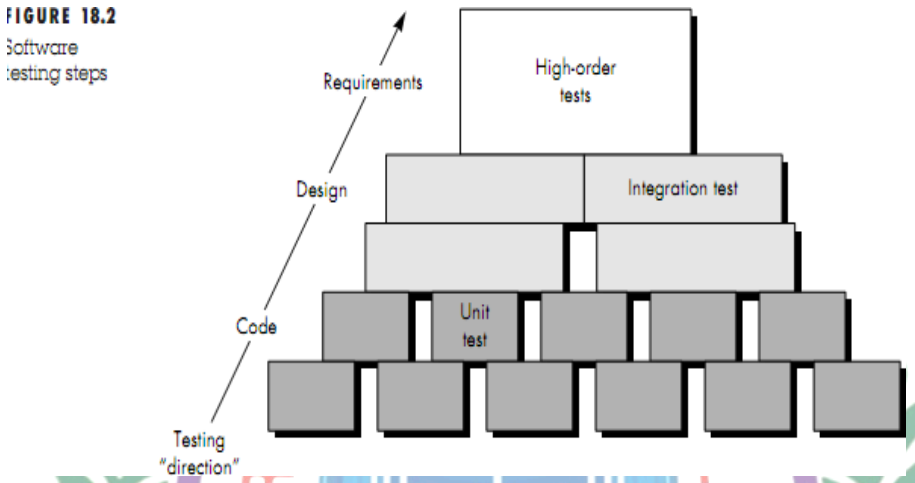
FIGURE 18.1
Testing strategy

Unit testing begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code. Testing progresses by moving outward along the spiral to **integration testing,** where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, we encounter **validation testing,** where requirements

established as part of software requirements analysis are validated against the software that has

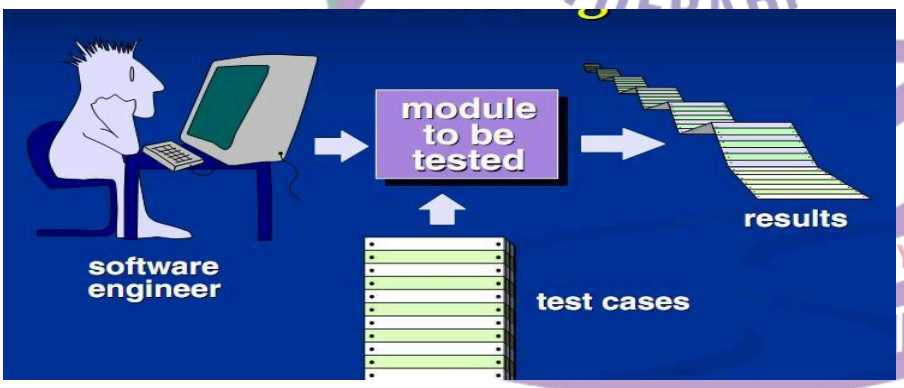
been constructed. Finally, we arrive at **system testing,** where the software and other system elements are tested as a whole.

Unit testing makes heavy use of white-box testing techniques. Black-box test case design techniques are the most prevalent **during integration,** although a limited amount of white-box testing may be used to ensure coverage of major control paths. Black-box testing techniques are used exclusively during **validation.** Software, once validated, must be combined with other system elements (e.g., hardware, people, and databases). **System testing** verifies that all elements mesh properly and that overall system function/performance is achieved.



TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

Unit Testing:



This is a figure just to understand unit testing

Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The unit test is white-box oriented.

Unit Test Considerations

The tests that occur as part of unit tests are illustrated schematically in Figure 18.4. The module

interface is tested to ensure that information properly flows into and out of the program unit under test. The local

data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

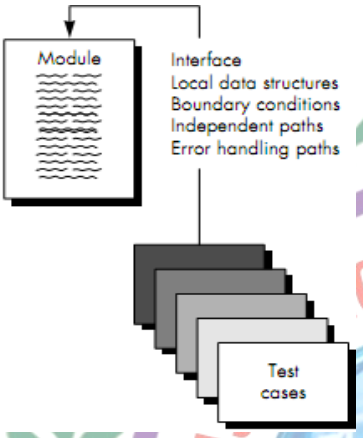


FIGURE 18.4
Unit test

What errors are commonly found during Unit Testing?

(1) Misunderstood or incorrect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, (5) incorrect symbolic representation of an expression. Comparison and control flow are closely coupled to one another (i.e., change of flow frequently occurs after a comparison).

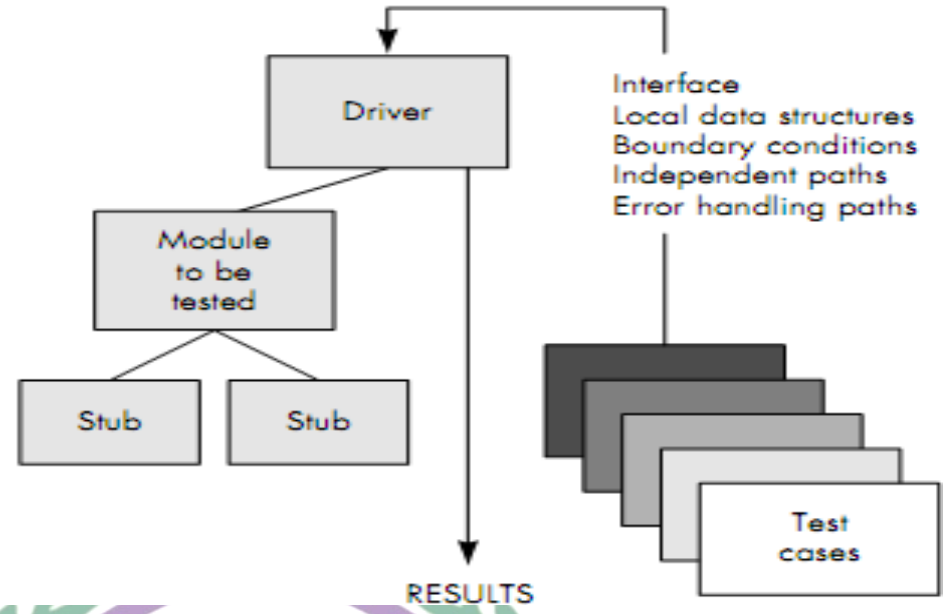
Test cases should uncover errors such as

(1) comparison of different data types, (2) incorrect logical operators or precedence,(3) expectation of equality when precision error makes equality unlikely, (4) incorrect comparison of variables, (5) improper or nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and
(7) improperly modified loop variables.

Unit Test Procedures: After source level code has been developed, reviewed, and verified for correspondence to component-level design, unit test case design begins

MATRUSRI ENGINEERING COLLEGE HYDERABAD
Sponsored by
MATRUSRI EDUCATION SOCIETY
ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA

FIGURE 18.5
Unit test environment



MATRUSRI ENGINEERING COLLEGE
HYDERABAD
Sponsored by
MATRUSRI EDUCATION SOCIETY
ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in Figure 18.5. In most applications a driver is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results. Stubs serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing. ***Unit testing is simplified when a component with high cohesion is designed***.

Integration Testing: The objective is to take unit tested components and build a program structure that has been dictated by design**.**

Approaches:

"Big bang" approach: All components are combined in advance. Entire program is tested as a whole.

Incremental integration: The program is constructed and tested in small increments, where errors are easier to isolate and corrected.

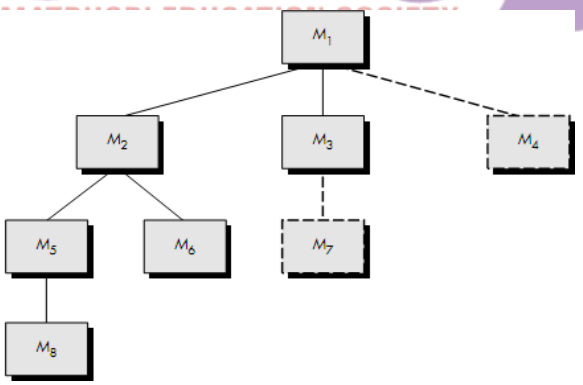
Number of different incremental integration strategies are discussed:

1. **Top-Down Integration**

Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure **in either a depth-first or a breadth-first manner.**

Referring to figure ***Depth-first integration*** would integrate all components on a major control path of the structure. Selection of a major path and depends on application-specific characteristics. For example, selecting left hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for

FIGURE 18.6
Top-down integration



proper functioning of M2) M6 would be integrated. Then, central and right-hand control paths are built.

Breadth-first integration incorporates all components directly subordinate at each

level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

Steps for Top-Down Integration:

1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced. The process continues from step 2 until the entire program structure is built.

What problems are encountered when top-down integration strategy is chosen?

The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing; therefore, no significant data can flow upward in the program structure

2. Bottom-Up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

Steps for Bottom-Up Integration:

1. Low-level components are combined into clusters (sometimes-called builds) that perform a specific software sub function.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.

MATHUSR ENGINEERING COLLEGES HYDERABAD
Sponsored by
MATHUSRI EDUCATION SOCIETY
ESTD, 1980
TECHNOLOGY TRAVERSES TERRESTRA

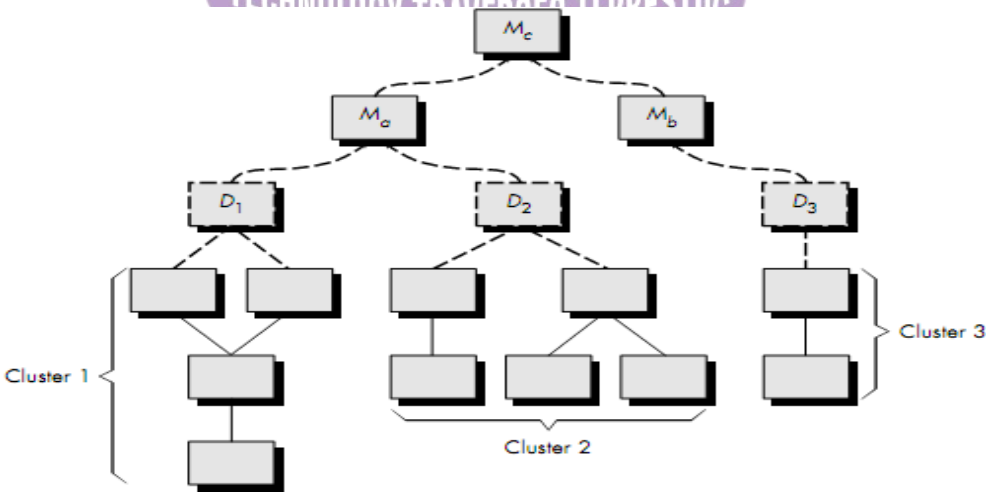


FIGURE 18.7 Bottom-up integration

Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

Regression Testing: Each time a new module is added as part of integration testing, the software changes. New data flow paths are established, new I/O may occur, and new control logic is invoked. **Regression testing** is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.

The regression test suite (the subset of tests to be executed) contains three different classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

Smoke Testing: Smoke testing is an integration testing approach that is commonly used when "shrink- wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects. The smoke testing approach encompasses the following activities:

1. ***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.
2. ***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.
3. ***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

Smoke testing provides a number of benefits when it is applied on complex, time-critical software engineering projects:

- Integration risk is minimized.
- The quality of the end-product is improved.
- Progress is easier to assess.

Comments on Integration Testing: The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them. The major disadvantage of bottom-up integration is that "the program as an entity does not exist until the last module is added"

MATRUSRI ENGINEERING COLLEGE HYDERABAD Sponsored by MATRUSRI EDUCATION SOCIETY ESTD: 1980 TECHNOLOGY TRAVERSES TERRESTRA

Selection of an integration strategy depends upon software characteristics and, sometimes, project schedule. In general, a combined approach (sometimes-called ***sandwich testing***) that uses top-down tests for upper levels of the program structure, coupled with bottom-up tests for subordinate levels may be the best compromise.

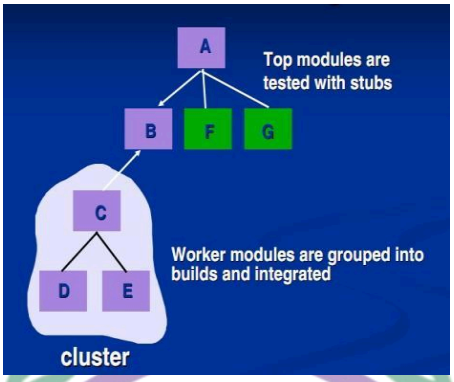


Fig: Sandwich Testing What is a critical module and why should we identify it?

- Address several software requirements
- Has a high level of control (high in the program structure) structure)
- Is complex or error prone
- Has definite performance requirements

VALIDATION TESTING

Validation succeeds when software functions in a manner that can be reasonably expected by the customer.

Validation Test Criteria: Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. Both plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct and other requirements are met.

After each validation test case has been conducted, one of two possible conditions exist:

1. The function or performance characteristics conform to specification and are accepted
2. A deviation from specification is uncovered and a deficiency list is created.

Configuration Review: An important element of the validation process is a configuration review. The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to bolster the support phase of the software life cycle. The configuration review, sometimes called an audit.

MVSR ENGINEERING COLLEGE HYDERABAD Sponsored by CMR EDUCATIONAL SOCIETY ESTD. 1980 TECHNOLOGY TRAVERSES TERRESTRA

Alpha and Beta Testing: A customer conducts the **alpha test** at the developer's site.

The **beta test** is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present

SYSTEM TESTING

Software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted. These tests fall outside the scope of the software process and are not conducted solely by software engineers. A classic system-testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem.

Recovery Testing: Recovery testing is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.

Security Testing: Security testing attempts to verify that protection mechanisms built into a system. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system.

Stress Testing: Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, Essentially, the tester attempts to break the program.

A variation of stress testing is a technique called **sensitivity testing.** Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing

Performance Testing: Performance testing is designed to test the run-time performance of software within context of an integrated system. Performance testing occurs throughout all steps in testing process. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

THE ART OF DEBUGGING

Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error,

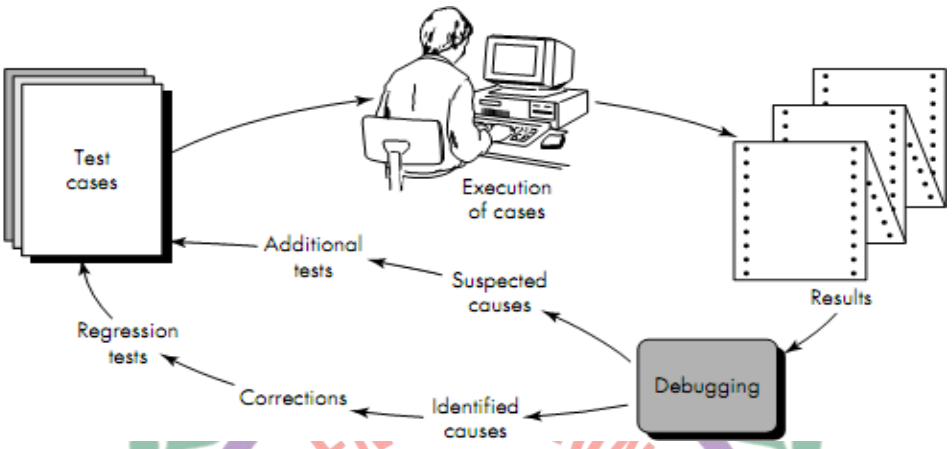
debugging is the process that results in the removal of the error.

The Debugging Process: The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the noncorresponding data are a

symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction.

The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found.

FIGURE 18.8
The debugging process



Debugging Approaches: In general, three categories for debugging approaches may be proposed

(2) brute force, (2) backtracking, and (3) cause elimination.

The **brute force category** of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails

Backtracking is a common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found.

The third approach to debugging — **cause elimination** — is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis.

Once a bug has been found, it must be corrected. However, as we have already noted, the correction of a bug can introduce other errors and therefore do more harm than good.

Van Vleck [VAN89] suggests three simple questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

- Is the cause of the bug reproduced in another part of the program?
- What "next bug" might be introduced by the fix I am about to make?
- What could we have done to prevent this bug in the first place?

MATRUSRI ENGINEERING COLLEGE HYDERABAD
Sponsored by MATRUSRI EDUCATION SOCIETY ESTD. 1980
TECHNOLOGY TRAVERSES TERRESTRA