SOFTWARE TESTING

CONTENTS

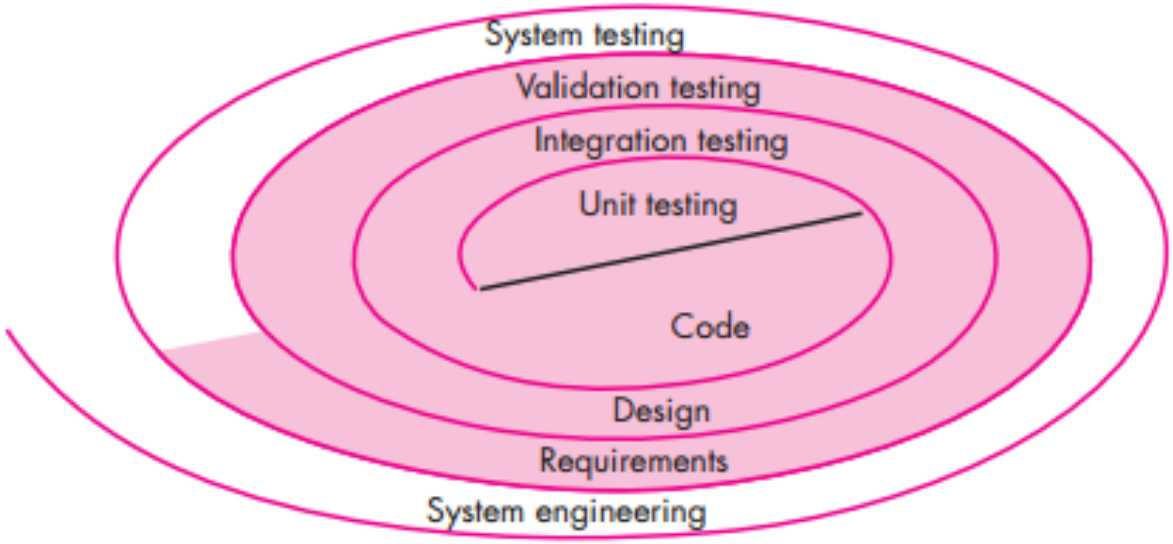
- **Software Testing Strategies**
  - Introduction
  - Verification vs. Validation
  - Testing Strategies for Conventional Software
    1. Unit Testing
    2. Integration Testing
    3. Regression Testing
    4. Smoke Testing
  - Validation Testing
  - System Testing

CONTENTS

SOFTWARE TESTING STRATEGIES

- **Introduction:**

A strategy for software testing provides a road map that describes the steps to be conducted as part of testing, when these steps are planned and then undertaken, and how much effort, time, and resources will be required.



TEMPLATES FOR TESTING

- To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

VERIFICATION VS. VALIDATION

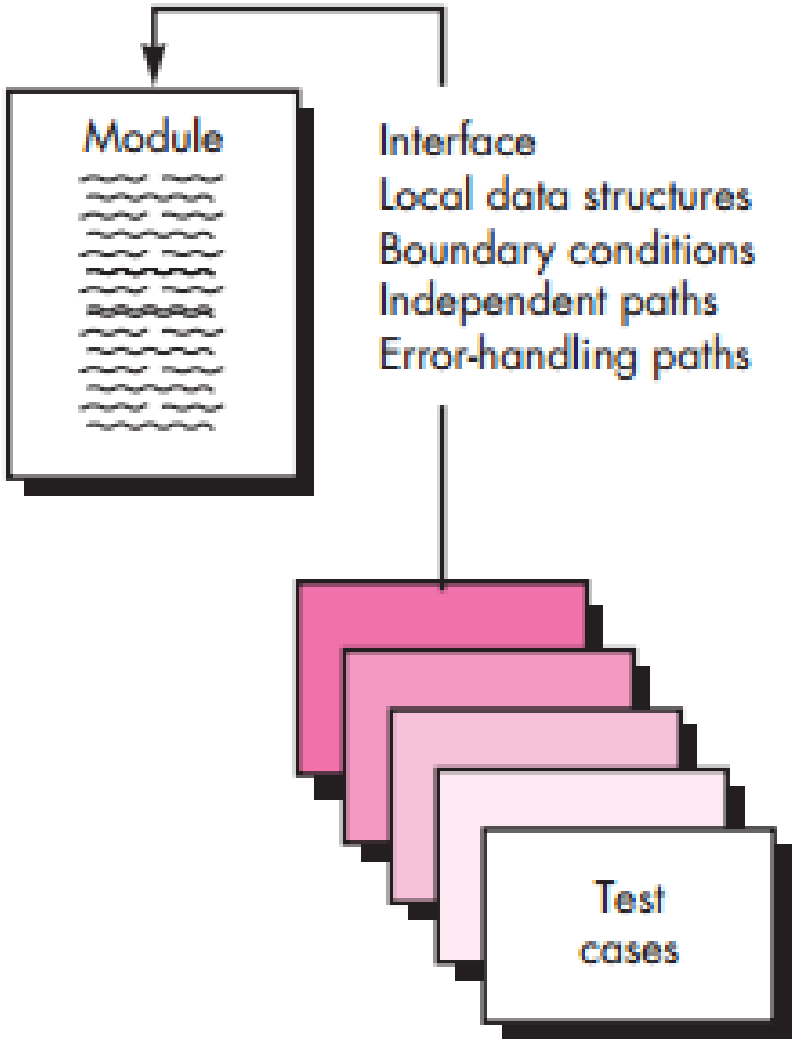
- **Verification** refers to the set of tasks that ensure that software correctly implements a specific function.

- **Validation** refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- According to Boehm :
  - Verification: “Are we building the product right?”
  - Validation: “Are we building the right product?”

TESTING STRATEGIES FOR CONVENTIONAL SOFTWARE

- **Unit Testing :**
  - Unit testing focuses verification effort on the smallest unit of software design—the software component or module.
  - The unit test focuses on the internal processing logic and data structures within the boundaries of a component.

Unit –Test Considerations :



- **Unit Test Procedures :**

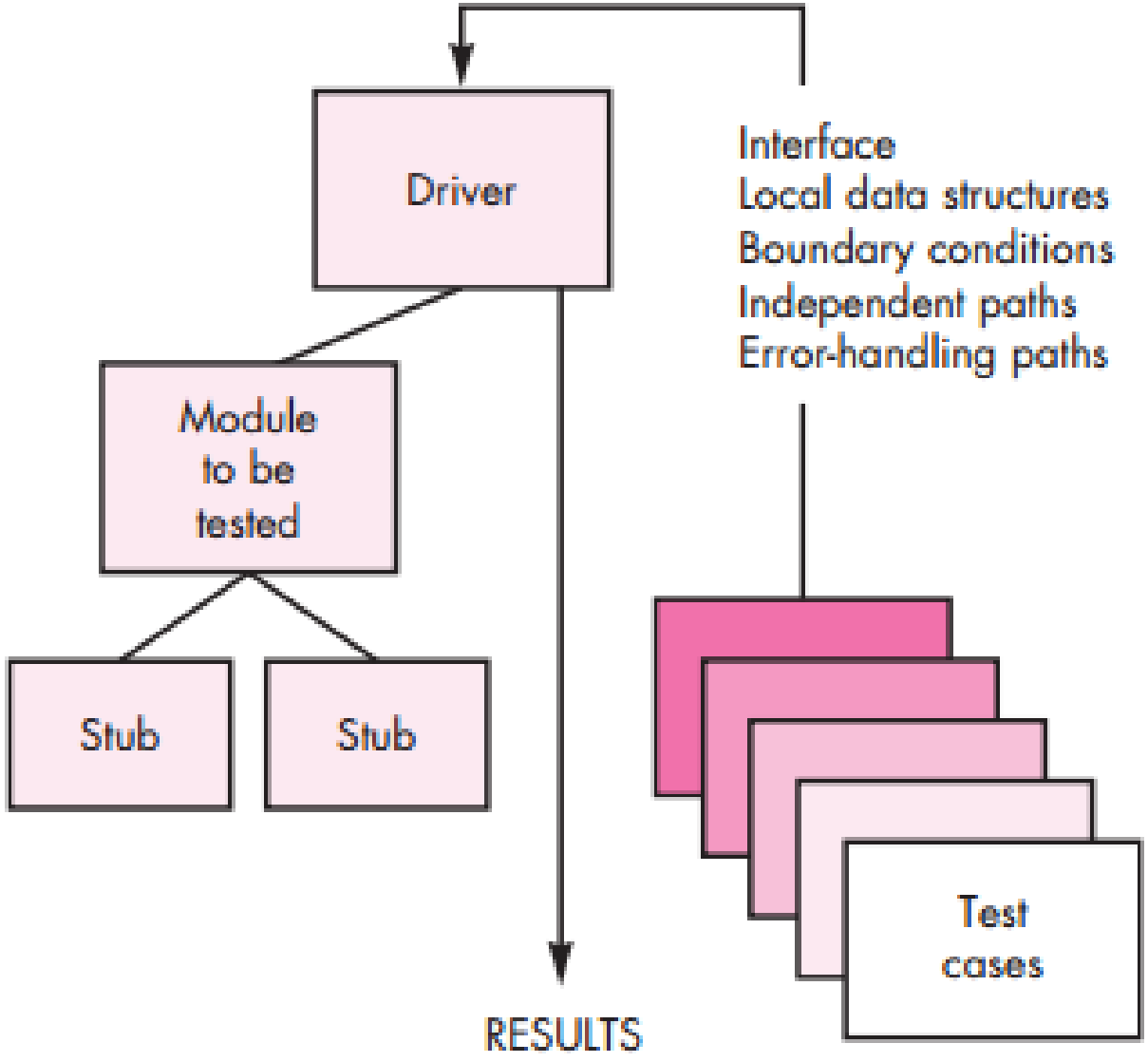
  - A component is not a stand-alone program. Hence, driver and/or stub software must be developed for each unit test.

  - **Driver** :

    It 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

  - **Stub** :

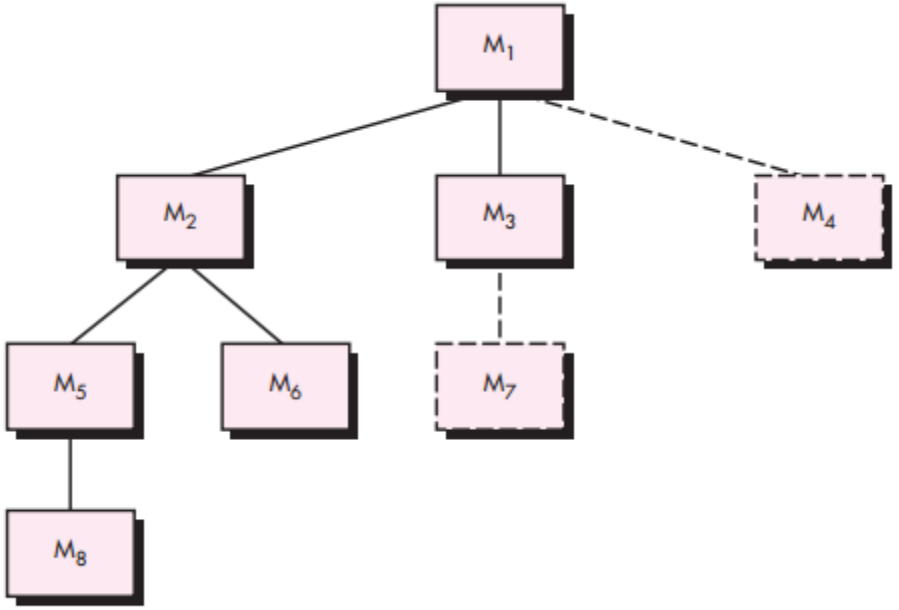
    They serve to replace modules that are subordinate (invoked by) the component to be tested.


Driver
Interface
Local data structures
Boundary conditions
Independent paths
Error-handling paths
Module to be tested
Stub
Stub
Test cases
RESULTS

INTEGRATION TESTING

- Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing.
- The objective is to take unit-tested components and build a program structure that has been dictated by design
- It is of two types :
  1. Top-down integration
  2. Bottom-up Integration

- **Top-down integration**:
  - Top-down integration testing is an incremental approach to construction of the software architecture
  - Modules are integrated by moving downward through the control hierarchy, beginning with the main control module.
  - Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.



The integration process is performed in a series of four steps:

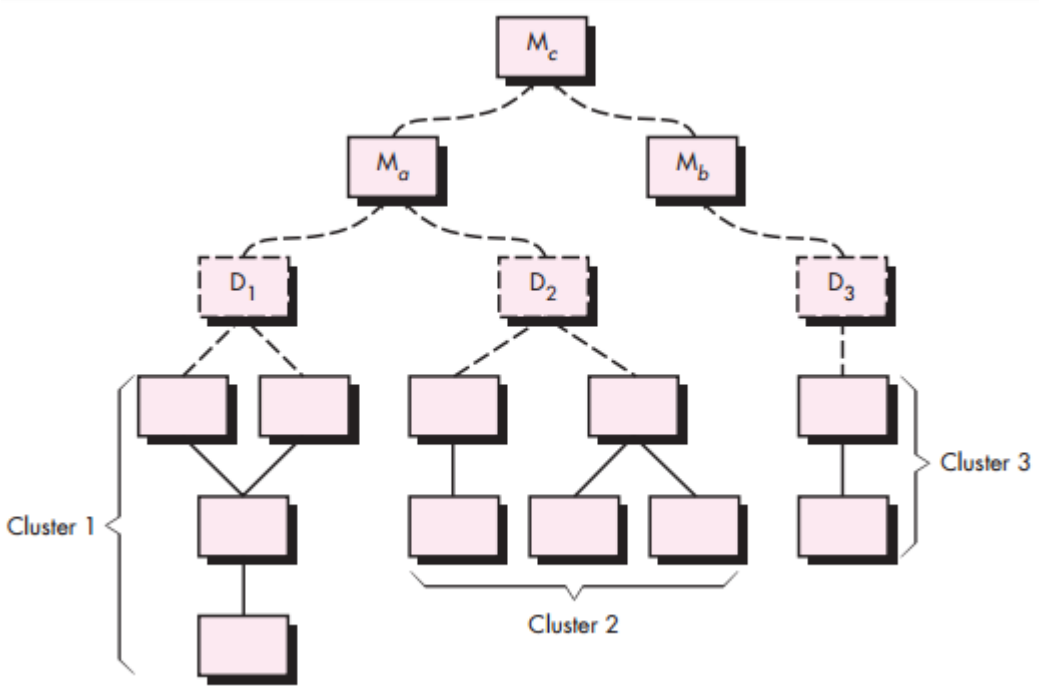
1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

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.

- **Bottom-up integration :**
  - Bottom-up integration testing begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure)
  - The components are integrated from the bottom up. So the functionality provided by components subordinate to a given level is always available and hence the need for stubs is eliminated.



A bottom-up integration strategy may be implemented with the following steps:

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.

TOP-DOWN VS. BOTTOM-UP INTEGRATION

- The major disadvantage of the top-down approach is the need for stubs and the attendant testing difficulties that can be associated with them.
- Problems associated with stubs may be offset by the advantage of testing major control functions early
- The major disadvantage of bottom-up integration is that “the program as an entity does not exist until the last module is added
- This drawback is tempered by easier test case design and a lack of stubs.

- **Regression Testing :**
  - 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 :

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

  - As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

- **Smoke Testing :**
  - Smoke testing is an integration testing approach that is commonly used when product software is developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess the project on a frequent basis.

The smoke-testing approach encompasses the following activities :

- 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 “showstopper” 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

Benefits of Smoke Testing :

- Integration risk is minimized. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.

- The quality of the end product is improved. Because the approach is construction (integration) oriented, smoke testing is likely to uncover functional errors as well as architectural and component-level design errors. If these errors are corrected early, better product quality will result.

Benefits of Smoke Testing (Cont.) :

- Error diagnosis and correction are simplified. Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with “new software increments”—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.

- Progress is easier to assess. With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

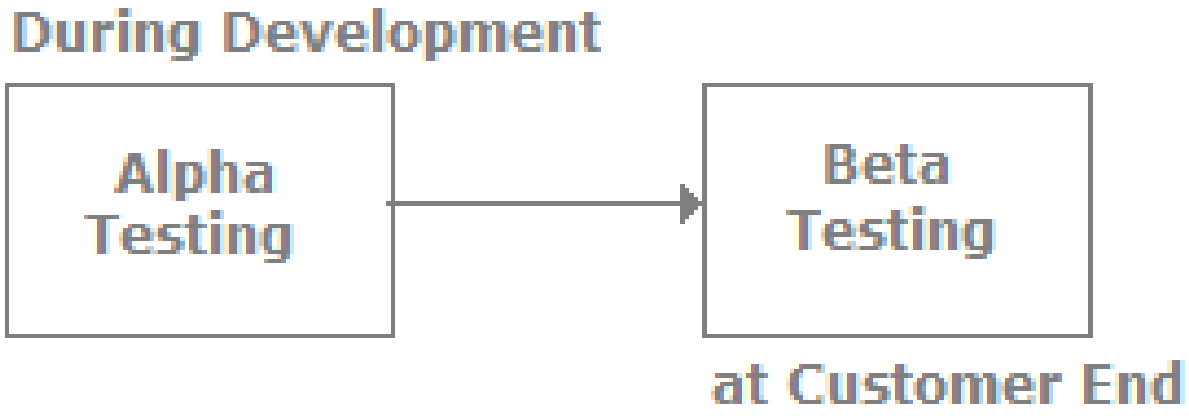
VALIDATION TESTING

- Validation is successful when software functions in a manner that can be reasonably expected by the customer.
- Validation testing begins at the culmination of integration testing
- Testing focuses on user-visible actions and user-recognizable output from the system

- **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 activities.
  - The configuration review, sometimes called an audit.

- **Alpha Testing:**
  - The alpha test is conducted at the developer's site by a representative group of end users.
  - The software is used in a natural setting with the developer "looking over the shoulder" of the users and recording errors and usage problems.
  - Alpha tests are conducted in a controlled environment.

- **Beta Testing:**
  - The beta test is conducted at one or more end-user sites.
  - The developer generally is not present.
  - The beta test is a “live” application of the software in an environment that cannot be controlled by the developer.
  - A variation on beta testing, called customer acceptance testing, is sometimes performed when custom software is delivered to a customer under contract. The customer performs a series of specific tests in an attempt to uncover errors before accepting the software from the developer.



SYSTEM TESTING

System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system

- **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.
  - If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

- **Security Testing :**
  - Security testing attempts to verify that protection mechanisms built into a system will protect it from improper penetration.
  - “The system’s security must, of course, be tested for invulnerability from frontal attack—but must also be tested for invulnerability from flank or rear attack.” – Beizer
  - The role of the system designer is to make penetration cost more than the value of the information that will be obtained

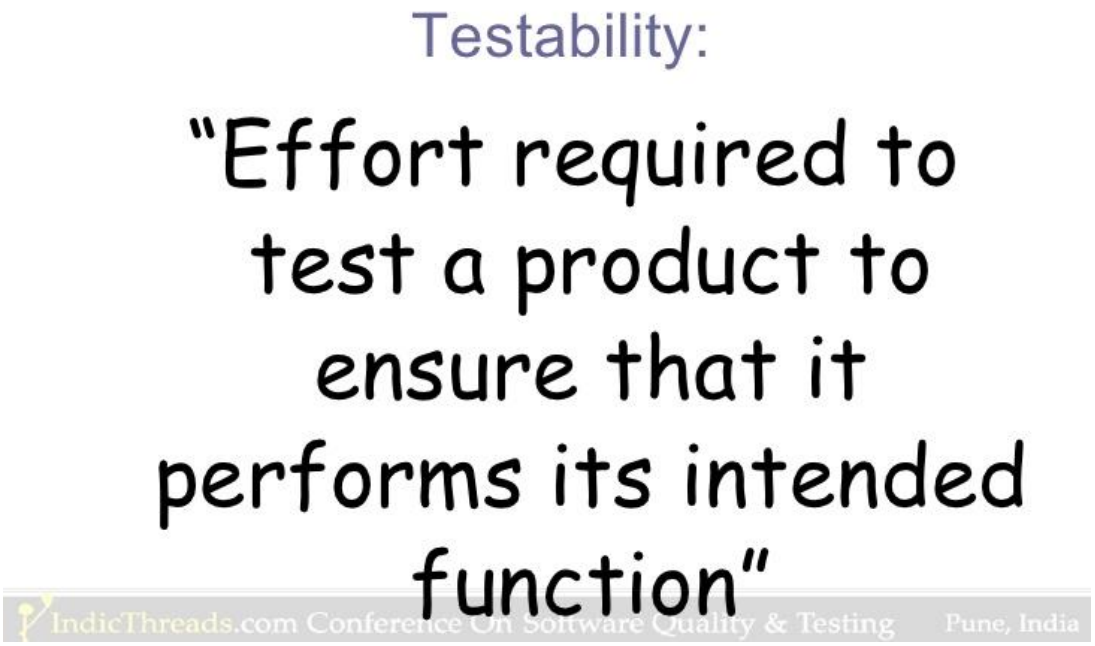
- **Stress Testing :**
  - Stress testing executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
  - A variation of stress testing is a technique called sensitivity testing
  - Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

- **Performance Testing :**
  - Performance testing is designed to test the run-time performance of software within the context of an integrated system.
  - Performance testing occurs throughout all steps in the testing process.

- **Deployment Testing :**
  - Deployment testing, sometimes called configuration testing, exercises the software in each environment in which it is to operate.
  - Deployment testing examines all installation procedures and specialized installation software (e.g., “installers”) that will be used by customers, and all documentation that will be used to introduce the software to end users.

TESTING CONVENTIONAL APPLICATIONS

- **Software Testing Fundamentals :**
  - Testability - "Software testability is simply how easily [a computer program] can be tested." – James Bach



The following characteristics lead to testable software:

- Operability. - “The better it works, the more efficiently it can be tested.”
- Observability -“What you see is what you test.”
- Controllability -“The better we can control the software, the more the testing can be automated and optimized.”
- Decomposability - “By controlling the scope of testing, we can more quickly isolate problems and perform smarter retesting.”
- Simplicity - “The less there is to test, the more quickly we can test it.”
- Stability - “The fewer the changes, the fewer the disruptions to testing.”
- Understandability - “The more information we have, the smarter we will test.”

- **Attributes of 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.

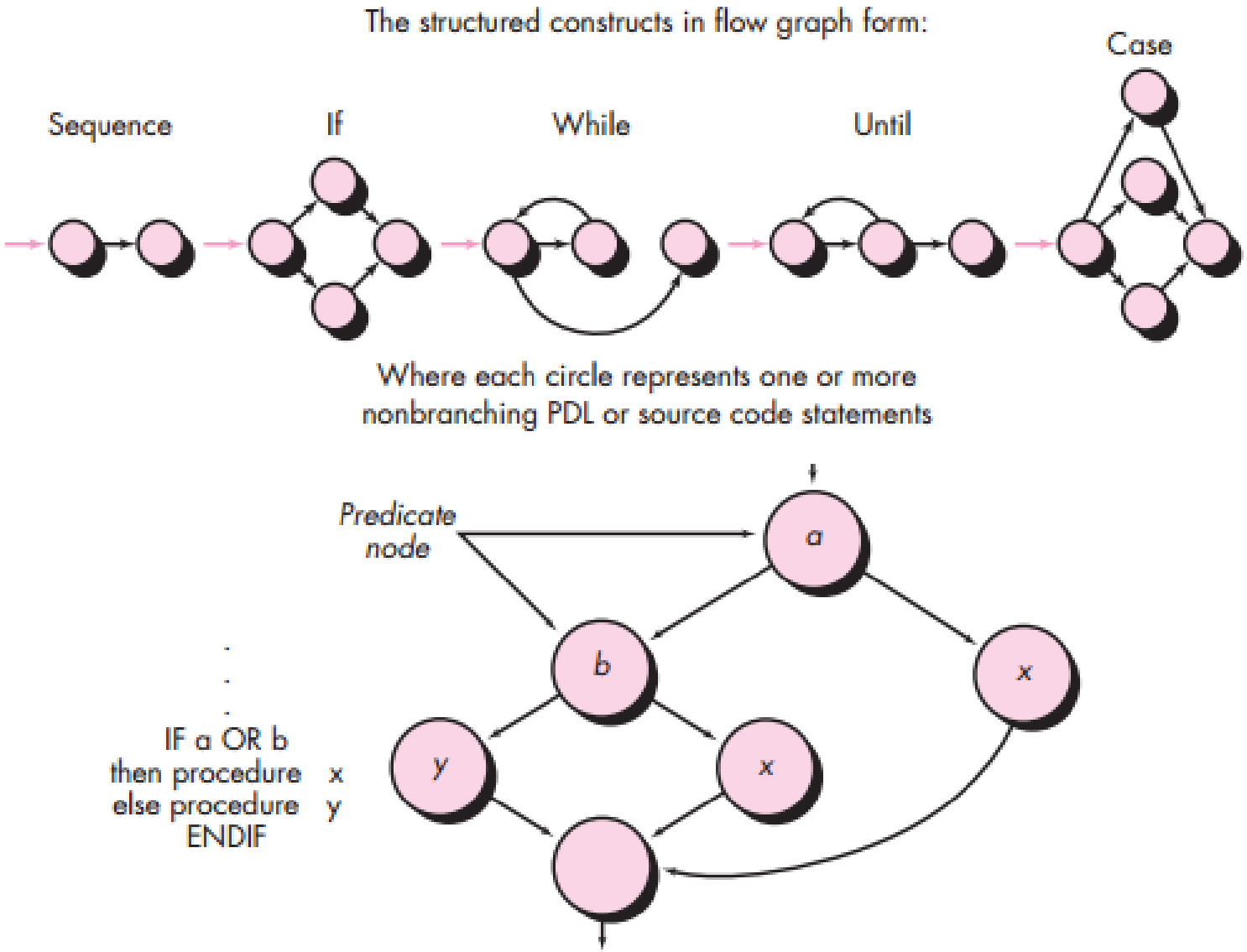
WHITE BOX TESTING

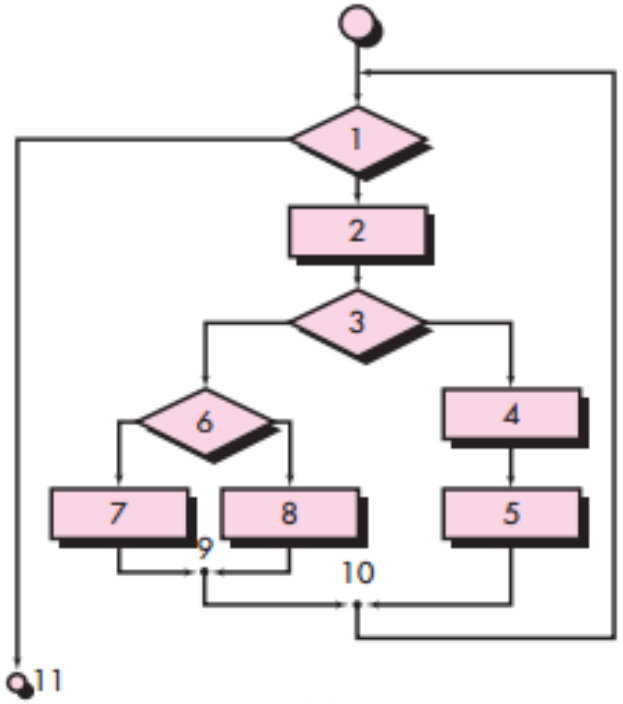
- White-box testing (glass-box testing), is a test-case design philosophy that uses the control structure described as part of component-level design to derive test cases.
- White Box Testing guarantees that :
  - All independent paths within a module have been exercised at least once.
  - All logical decisions have been exercised on their true and false sides.
  - All loops have been executed at their boundaries and within their operational bounds.
  - All internal data structures have been exercised to ensure their validity

BASIS PATH TESTING

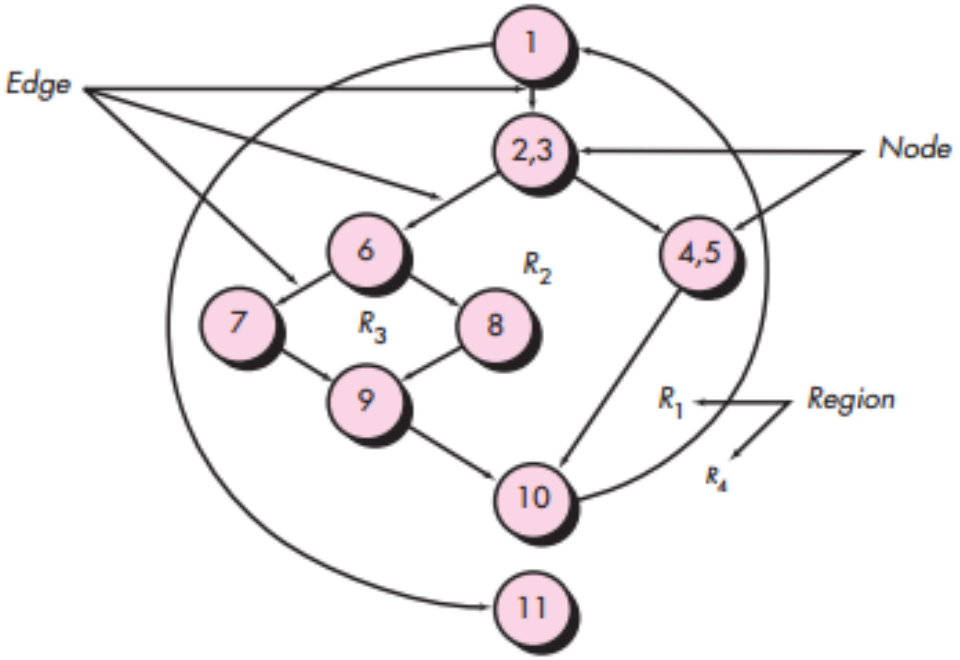
- Basis path testing is a white-box testing technique first proposed by Tom McCabe
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.

- **Flow Graph Notation :**





(a)



(b)

- **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.
  - For example, a set of independent paths for the flow graph in the previous slide
    1. Path 1: 1-11
    2. Path 2: 1-2-3-4-5-10-1-11
    3. Path 3: 1-2-3-6-8-9-10-1-11
    4. Path 4: 1-2-3-6-7-9-10-1-11

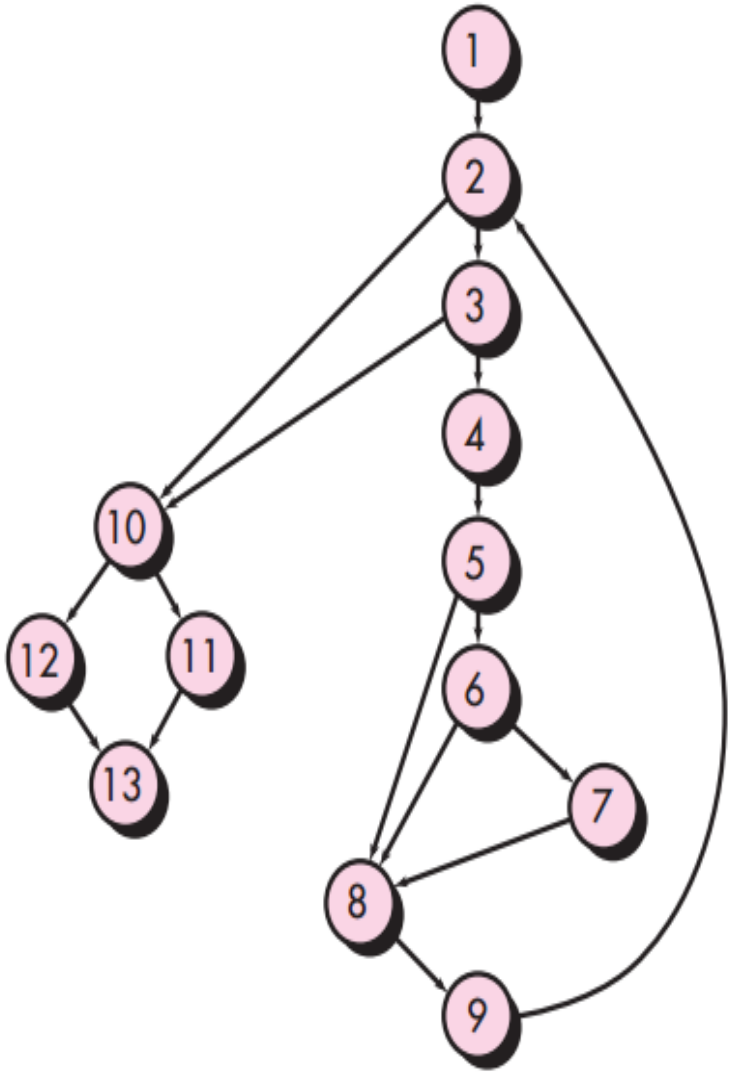
- **Cyclomatic Complexity :**
  - Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program.
  - The value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides you with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.

- Complexity is computed in one of three ways:
  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 and 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.

- The cyclomatic complexity for the flow graph in previous slide :
  - The flow graph has four regions.
  - V(G) = 11 edges - 9 nodes + 2 = 4.
  - V(G) = 3 predicate nodes + 1 = 4.

- **Deriving Test Cases :**
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

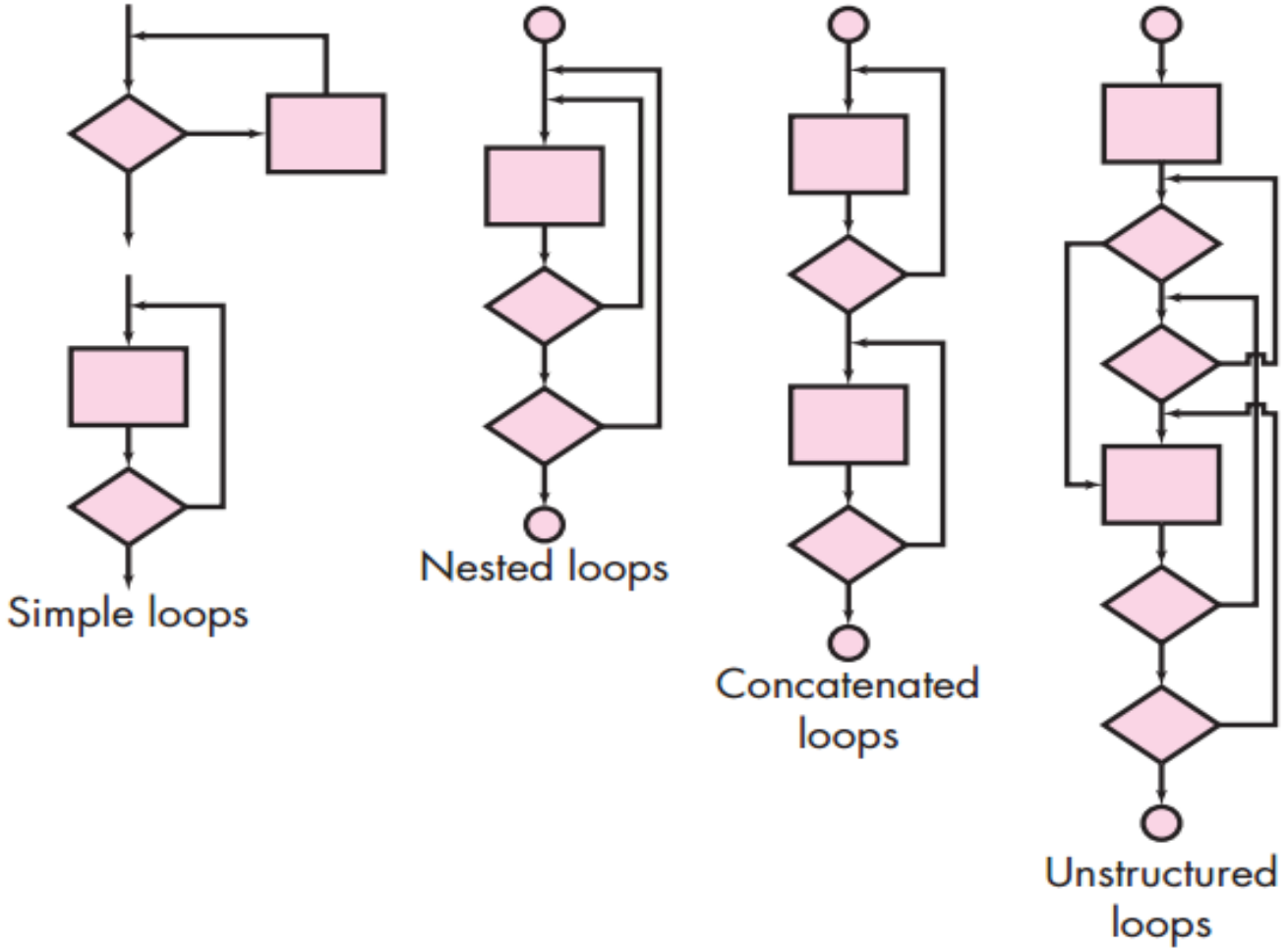
```
PROCEDURE average;

    *  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
```



CONTROL STRUCTURE TESTING

- **Condition Testing :**
  - Condition testing is a test-case design method that exercises the logical conditions contained in a program module.
- **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.
  - DEF(S) = {X | statement S contains a definition of X}
  - USE(S) = {X | statement S contains a use 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'.

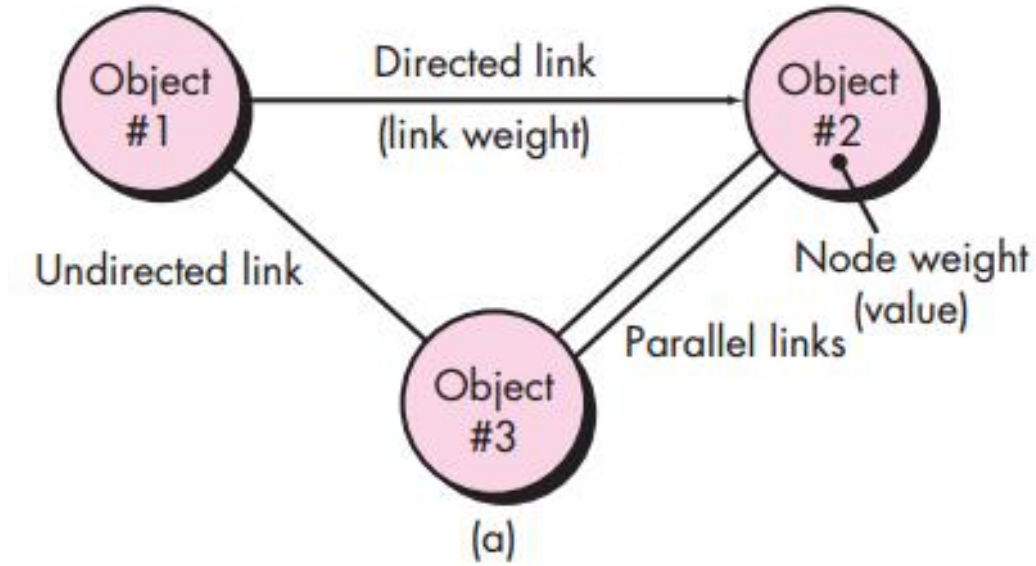
- **Loop Testing :**
  - Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs.

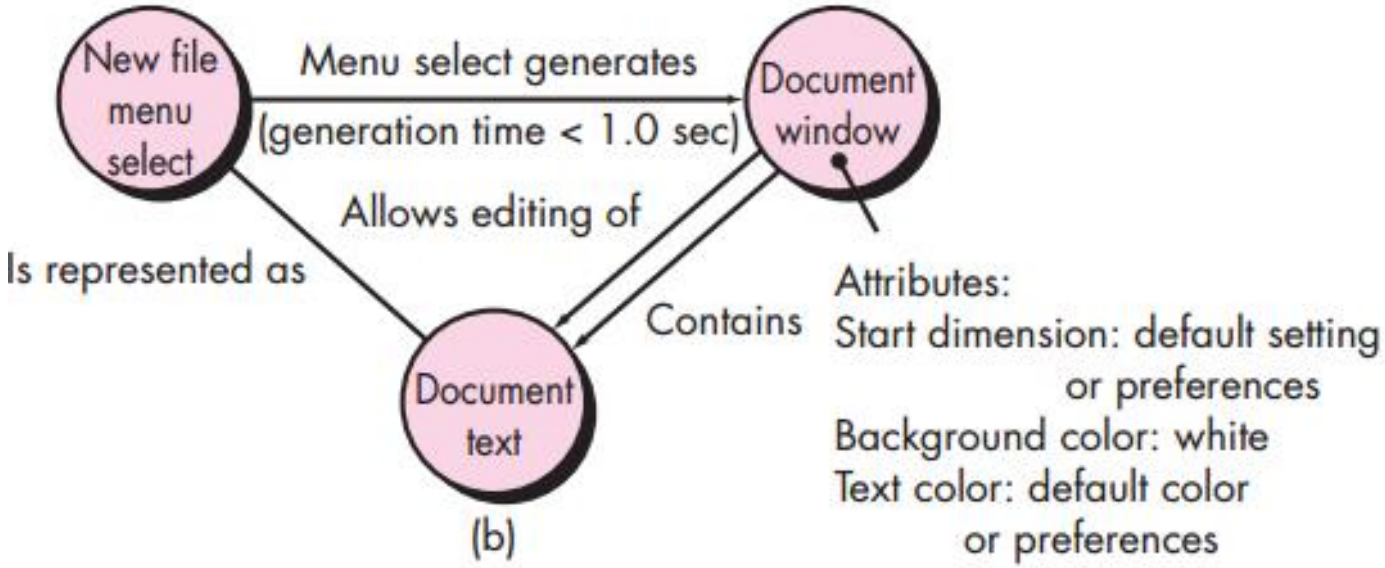


BLACK-BOX TESTING

- Black-box testing, also called behavioral testing, focuses on the functional requirements of the software.
- Black-box testing techniques enable you to derive sets of input conditions that will fully exercise all functional requirements for a program.
- It is a complementary approach that is likely to uncover a different class of errors than white box methods.

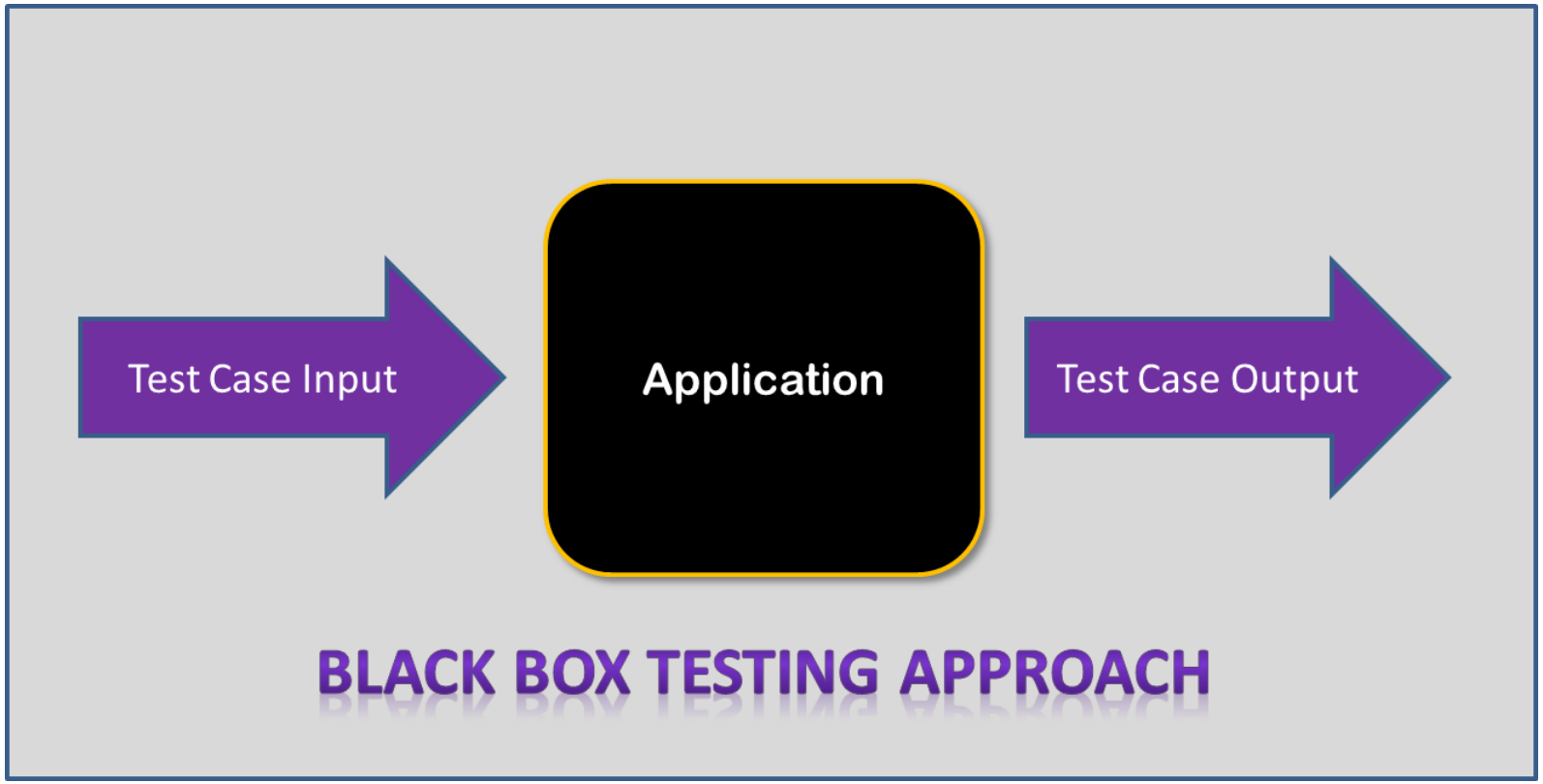
- **Graph-Based Testing Methods :**





Behavioral Testing methods that make use of graphs :

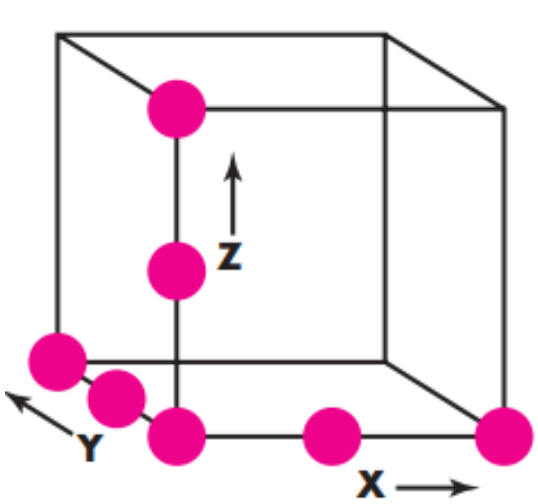
- Transaction flow modeling.
- Finite state modeling.
- Data flow modeling.
- Timing modeling.



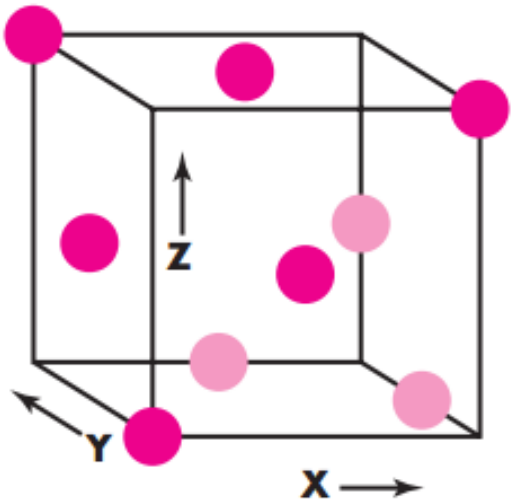
- **Equivalence Partitioning :**
  - Equivalence partitioning is a black-box testing method that divides the input domain of a program into classes of data from which test cases can be derived.
  - Test-case design for equivalence partitioning is based on an evaluation of equivalence classes for an input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

- **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.
  - Guidelines for BVA :
    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.
    3. Apply guidelines 1 and 2 to output conditions.
    4. If internal program data structures have prescribed boundaries be certain to design a test case to exercise the data structure at its boundary

- **Orthogonal Array Testing :**
  - Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
  - The orthogonal array testing method is particularly useful in finding region faults—an error category associated with faulty logic within a software component.



One input item at a time



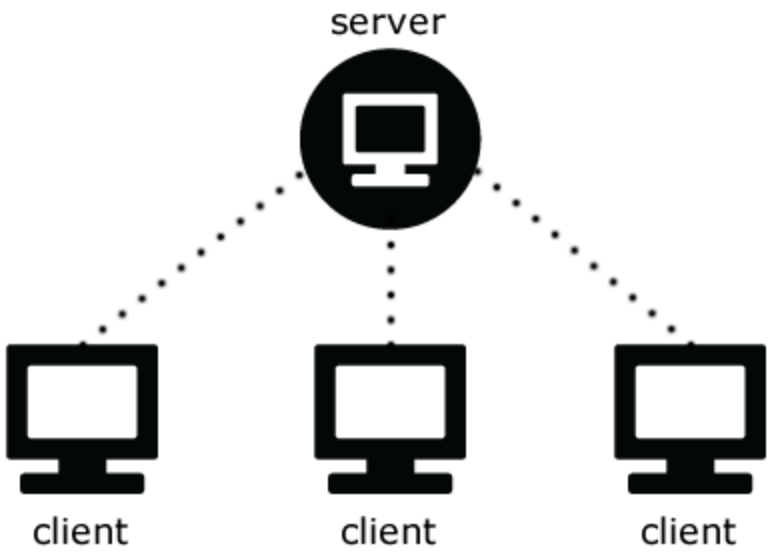
L9 orthogonal array

TESTING FOR SPECIALIZED ENVIRONMENTS

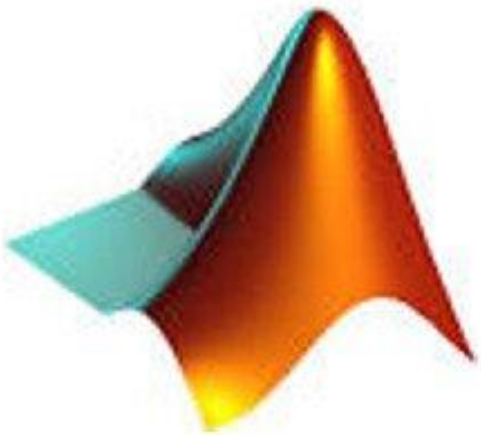
- **Testing GUI :**

Name	Supported platforms (testing system)	Supported platforms (tested system)
Ranorex	Windows, iOS, Android	Unknown
Rational Functional Tester	Windows, Linux	Windows, Swing, .NET, HTML
Robot Framework	Unknown	Unknown
Sahi	Web (cross-browser)	Web

- **<u>Testing of Client-Server Architectures :</u>**
- **The following testing approaches are commonly encountered for client-server applications:**
  - Application function tests.
  - Server tests.
  - Database tests.
  - Transaction tests.
  - Network communication tests



- **Testing Documentation and Help :**



vs.

REFERENCE

Roger S. Pressman, “Software Engineering A Practitioner’s Approach Seventh Edition ”, McGraw-Hill, 2010.

THANKS FOR LISTENING!

ANY QUESTIONS?

NO? SUPER!

BYE!