22CSC51 - AGILE METHODOLOGIES

Prepared By,

Mr.M.Muthuraja,

Assistant Professor

Department.of CSE

Kongu Engineering College

Unit - IV

Software Testing Fundamentals

SOFTWARE TESTING

- Testing begins **“in the small”** and progresses **“to the large.”**
- Testing is a **set of activities** that can be planned in advance and conducted systematically.
- For this reason a template for software testing—**a set of steps** into which we can place specific test-case design techniques and testing methods— should be defined for the software process.
- Software is **tested to uncover errors** that were made inadvertently as it was designed and constructed.

Generic Characteristics

- **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.

Strategic Approach to Software Testing

- Verification and Validation
- Organizing for Software Testing
- Software Testing Strategy—The Big Picture
- Criteria for Completion of Testing

Verification and Validation (V&V)

- 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**.
- Boehm states this another way:
- **Verification: "Are we building the product right?"**
- **Validation: "Are we building the right product?"**
- Verification and validation includes a wide array of SQA activities: **technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, usability testing, qualification testing, acceptance testing, and installation testing**.

Organizing for Software Testing

- There are often a number of misconceptions,

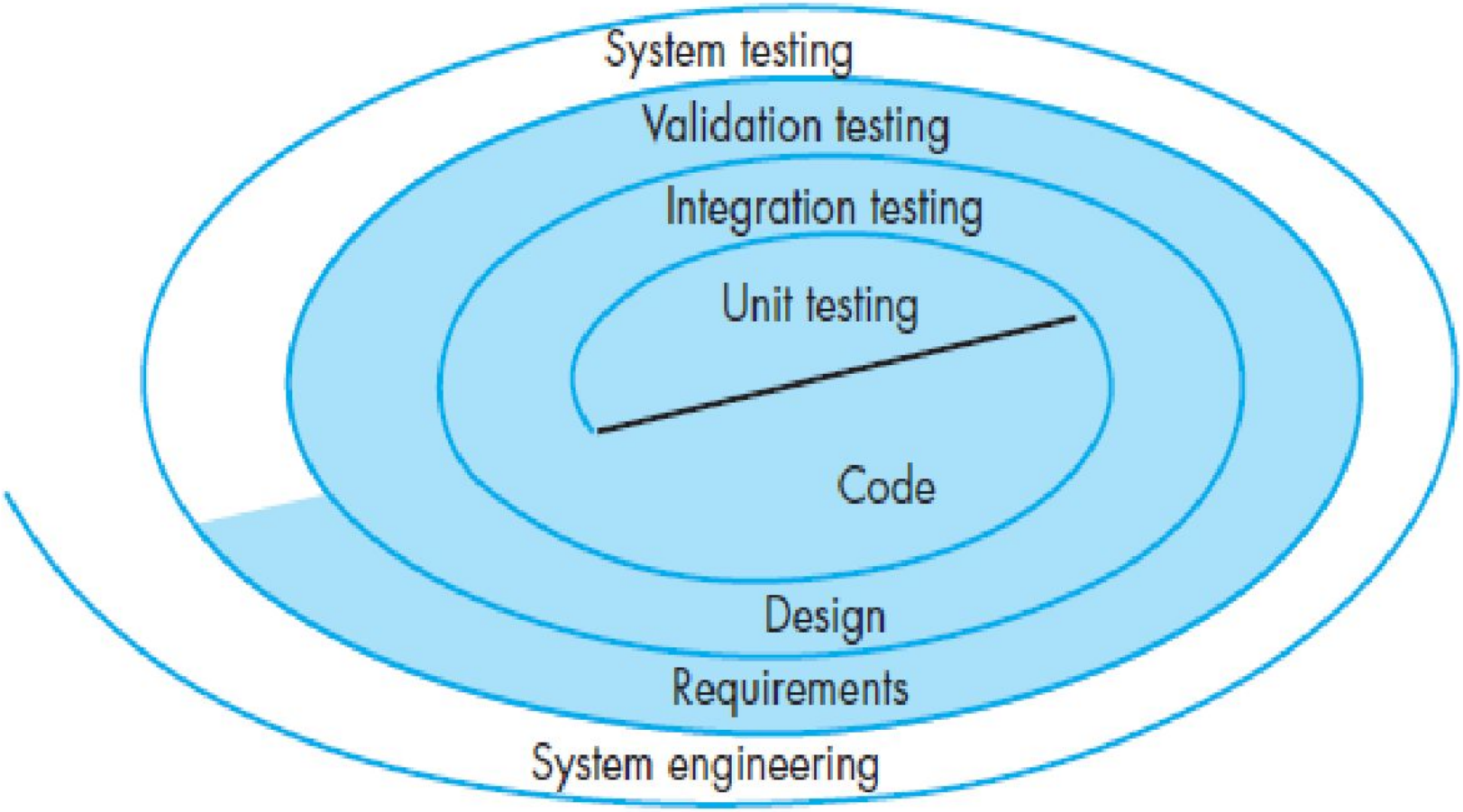
(1) that the **developer of software** should do **no testing at all**,

(2) that the software should be **"tossed over the wall"** to strangers who will test it mercilessly,

(3) that testers get involved with the project only when the **testing steps are about to begin.**

- Each of these statements is **incorrect.**
- The software **developer is always responsible** for testing the individual units (components) of the program, ensuring that each performs the function or exhibits the behavior 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 software architecture.
- Only after the software architecture is complete 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.

Software Testing Strategy -The Big Picture



Software Testing Strategy -The Big Picture

- **Unit testing** focuses **verification** effort on the smallest unit of software design— the software component or module.
- **Integration testing** is a **systematic technique** for constructing the software architecture while at the same time conducting tests to **uncover errors** associated with interfacing.
- **Validation** can be defined in many ways, but a simple (albeit harsh) definition is that **validation succeeds when software functions in a manner** that can be reasonably **expected by the customer**.
- **System testing** verifies that all elements **mesh properly** and that overall system function/performance is achieved.

Criteria for Completion of Testing

A **classic question arises** every time software testing is discussed:

- **One response to the question** is: “You're never done testing; the burden simply shifts from you (the software engineer) to the end user.”
- Every time the user **executes a computer program**, the program is being **tested**.
- By collecting **metrics during software testing** and making use of existing statistical models, it is possible to Test.

STRATEGIC ISSUES

Software testing strategy will succeed only when software testers:

(1) **specify product requirements** in a quantifiable manner long before **testing commences**,

(2) state testing **objectives explicitly**,

(3) understand the **users of the software** and develop a **profile for each user category,**

(4) develop a testing plan that emphasizes **"rapid cycle testing,"**

(5) build **"robust"** software that is designed to test itself

(6) use effective **technical reviews** as a filter prior to testing,

(7) conduct **technical reviews to assess** the test strategy and test cases themselves, and

(8) develop a **continuous improvement approach** for the testing process.

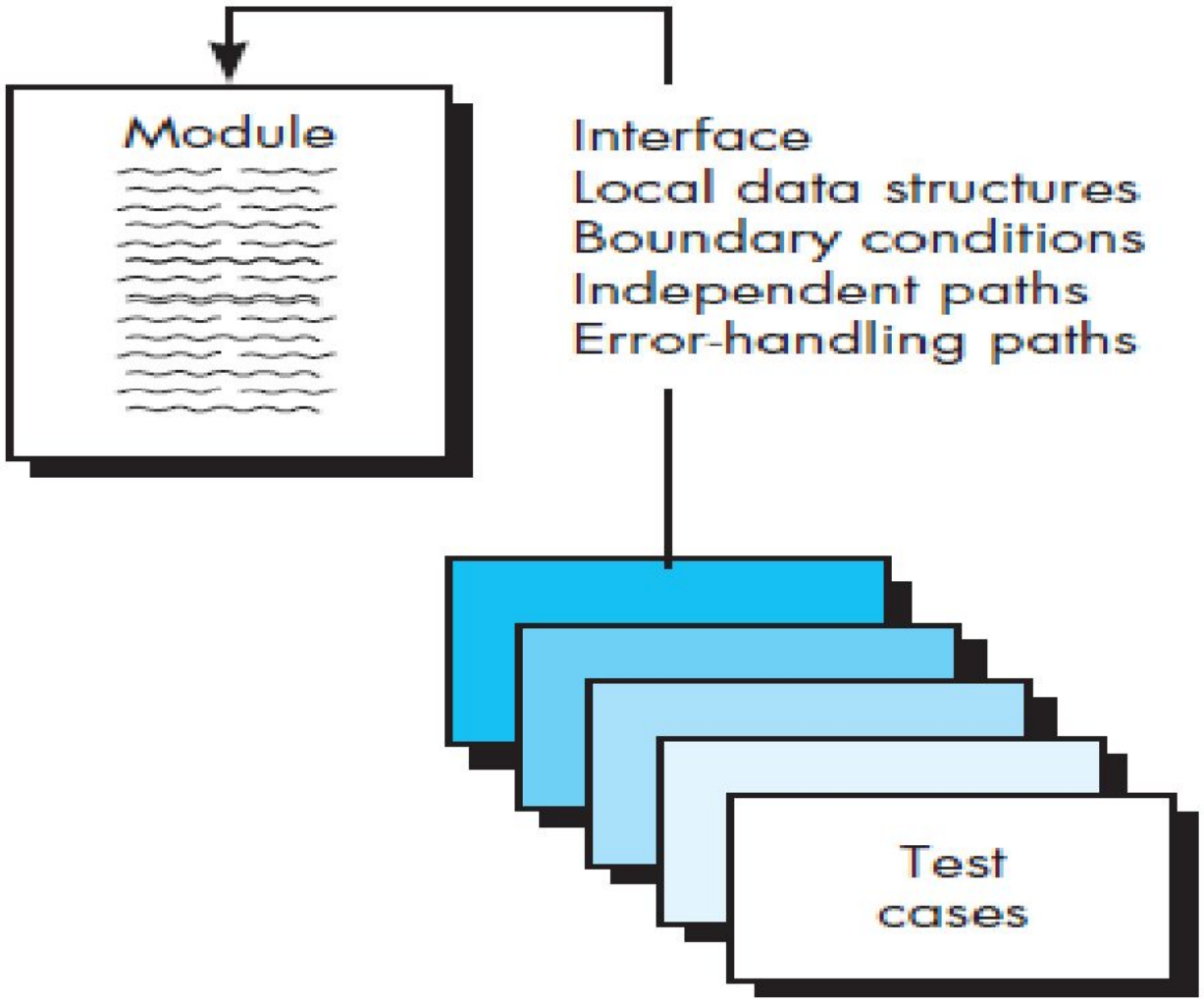
TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

- Many strategies can be used to **test software.**
- At one extreme, you can **wait until the system is fully constructed** and then conduct tests on the overall system in the hope of finding errors.
- This approach, although appealing, **simply does not work.**
- It will result in **buggy software that disappoints** all stakeholders.
- At the other extreme, you **could conduct tests on a daily basis,** whenever any part of the system is constructed.
- A testing strategy that is chosen by many software teams falls **between the two extremes**.
- It takes an incremental view of testing, beginning with the testing of **individual program units**, moving to tests designed to facilitate the integration of the units (sometimes on a daily basis), and culminating with tests that **exercise the constructed system**.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

- **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 focuses on the **internal processing logic and data structures** within the boundaries of a component.
- This type of testing can be **conducted in parallel for multiple components.**

Unit Test Considerations



Unit Test Considerations

- The module **interface is tested** to ensure that information properly **flows into and out of the program unit** under test.
- Local **data structures are examined** to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution.
- All **independent paths** through the control structure are exercised to ensure that all statements in a module have been **executed at least once**.
- **Boundary conditions** are tested to **ensure that the module operates properly** at boundaries established to limit or restrict processing.
- And finally, all **error-handling paths** are tested.

Unit Test Considerations

Among the potential errors that should be tested when error handling is evaluated are:

(1) **error description** is unintelligible,

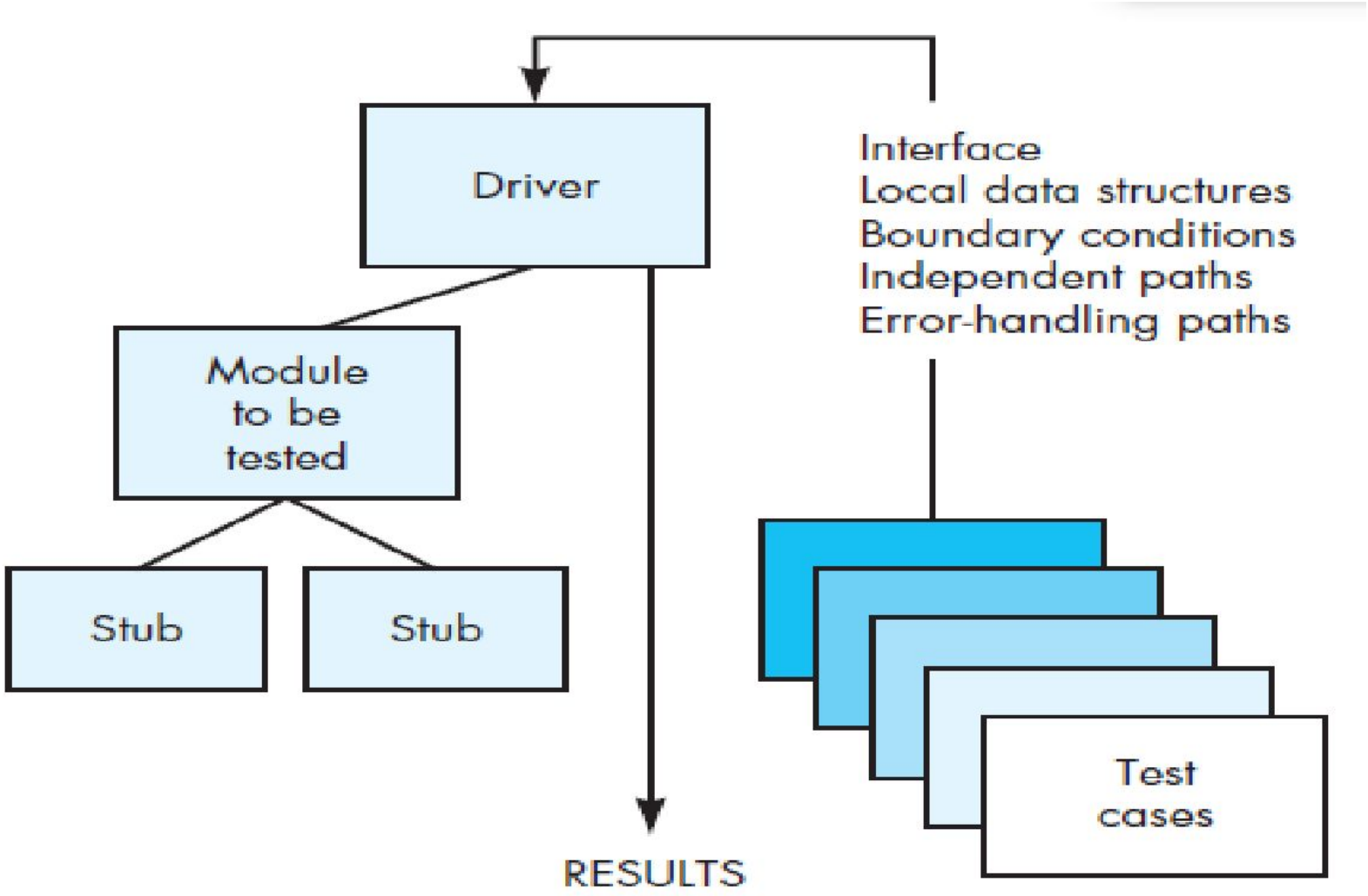
(2) error noted does not **correspond to error encountered,**

(3) error condition **causes system intervention** prior to error handling,

(4) **exception-condition** processing is incorrect, or

(5) error description does not **provide enough information to assist** in the location of the cause of the error

Unit-Test Procedures



Unit-Test Procedures

- Unit testing is normally **considered as an adjunct to the coding step.**
- The design of unit tests can occur before **coding begins or after source code has been generated.**
- A review of design information provides guidance for establishing test cases that are likely to **uncover errors in each of the categories** discussed earlier.
- Each test case should be **coupled with a set of expected results**.
- Because a component is not a stand-alone program, driver and/or stub software must **often be developed for each unit test**.
- 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 (invoked by) the component to be tested.

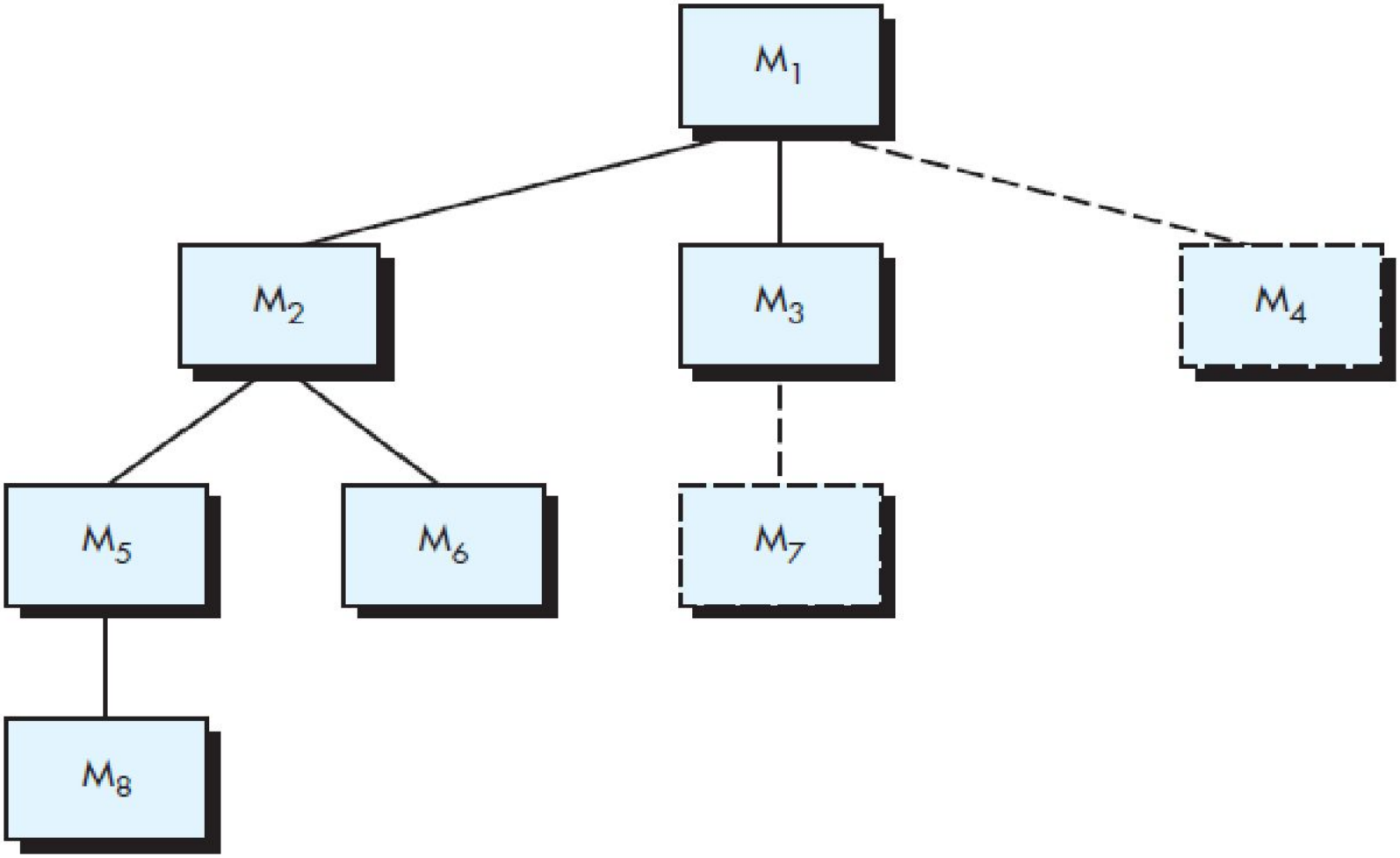
Integration Testing

- 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.
- A number of different **incremental integration** strategies are available. They are,
  - **Top-Down Integration**
  - **Bottom-Up Integration**
  - **Regression Testing**
  - **Smoke Testing**

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** (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a **depth first or breadth-first** manner.

Top-Down Integration



Top-Down Integration

- Referring to the Figure, **depth-first integration** integrates all components on a major control path of the program structure.
- Selection of a major path is somewhat arbitrary and depends on **application-specific** characteristics.
- For example, selecting the left-hand path, components **M1, M2, M5 would be integrated first**. Next, M8 or M6 would be integrated.
- Then, the 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** would be integrated first.
- The next control level, M5, M6, and so on, follows.

Top-Down Integration

The integration process is performed in a series of five 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.

5. **Regression testing** may be conducted to ensure that new errors have not been introduced.

Top-Down Integration

- The **top-down integration strategy** verifies major control or decision points early in the test process.
- If major control problems do exist, early recognition is essential.
- If **depth-first integration is selected**, a complete function of the software may be implemented and demonstrated.
- **Early demonstration of functional capability** is a confidence builder for all stakeholders.

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

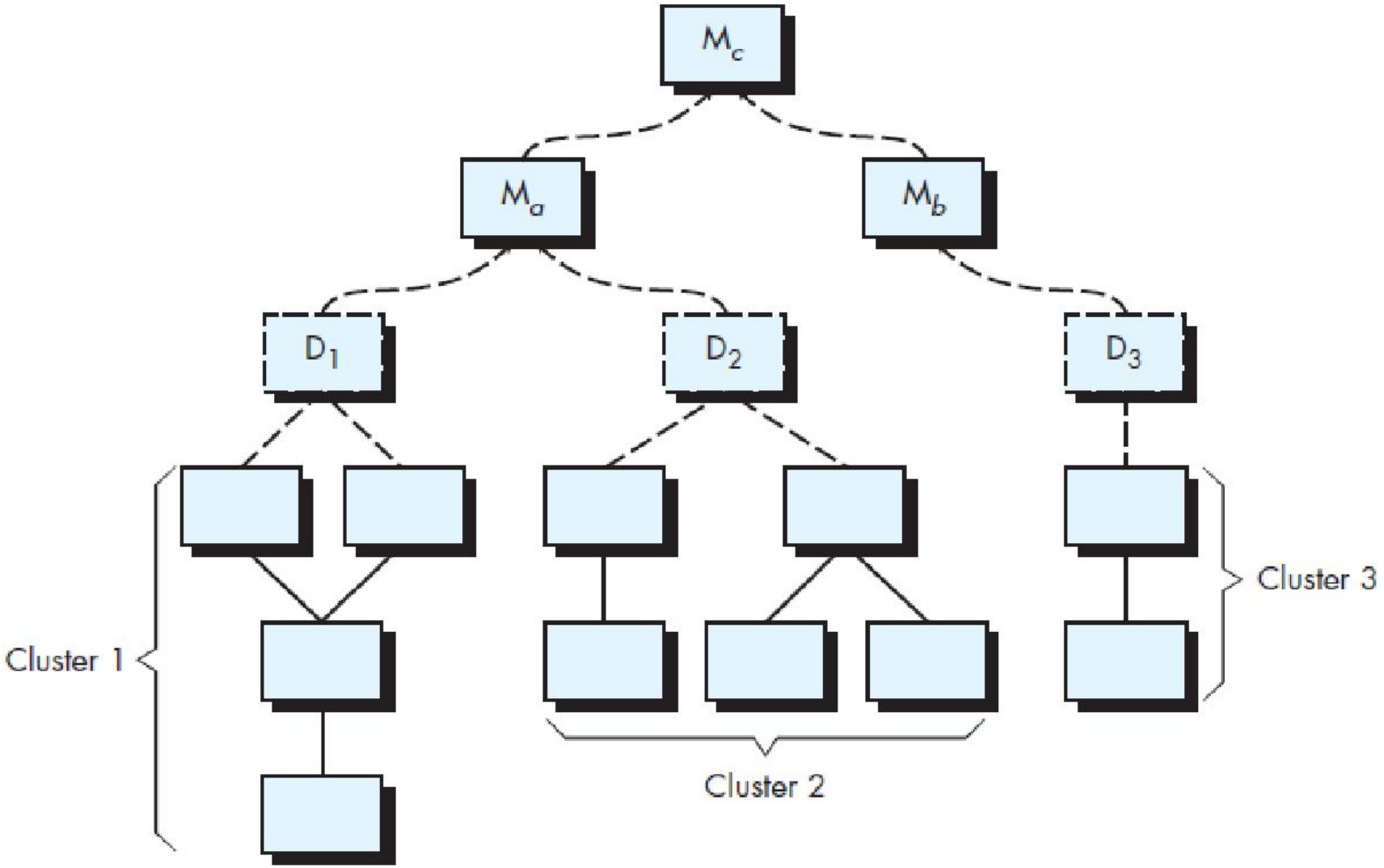
1. Low-level components are **combined into clusters** (sometimes called builds) that perform a specific software sub function.

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**.

Bottom-Up Integration



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.
- As **integration moves upward**, the need for **separate test drivers lessens**.
- Infact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

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.**
- Side effects associated with these changes may cause problems with functions that previously worked flawlessly.
- In the context of an **integration test strategy**, 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.
- Regression testing **helps to ensure that changes** (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be **conducted manually**, by **reexecuting a subset of all test cases or using automated capture/playback tools**.

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 test should exercise the entire system from end to end.
- It does not have to be exhaustive, but it should be **capable of exposing major problems**.

Benefits:

✔ Integration **risk is minimized**

✔ The quality of the **end product** is improved

✔ Error **diagnosis and correction** are simplified

✔ Progress is easier to assess

TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

Unit Testing in the OO Context:

- When **object-oriented software** is considered, the concept of the unit changes.
- **Encapsulation** drives the definition of classes and objects.
- This means that each class and each instance of a class packages attributes (data) and the operations that manipulate these data.
- An encapsulated class is usually the **focus of unit testing**.
- To illustrate, consider a class hierarchy in which an **operation X** is defined for the **superclass** and is inherited by a **number of subclasses**.
- Each subclass uses operation X , but it is applied within the context of the private attributes and operations that have been **defined for the subclass**.
- Because the context in which operation X is used varies in subtle ways, it is necessary to **test operation X** in the context of each of the subclasses.
- Unlike unit testing of conventional software, which tends to focus on the **algorithmic detail of a module** and the data that flow across the module interface, class testing for OO software is driven by the operations **encapsulated by the class and the state behavior of the class**.

TEST STRATEGIES FOR OBJECT-ORIENTED SOFTWARE

Integration Testing in the OO Context:

- There are **two different strategies** for integration testing of OO systems
- The first, **thread-based testing**, integrates the set of classes required to respond to one input or event for the system.
- Each thread is **integrated and tested individually**. **Regression testing** is applied to ensure that no side effects occur.
- The second integration approach, **use-based testing,** begins the construction of the system by **testing those classes** (called independent classes) that use very few (if any) server classes.
- After the independent classes are tested, the next layer of classes, called **dependent classes,** that use the independent classes are tested.
- This sequence of testing layers of dependent classes continues until the entire **system is constructed.**
- **Cluster testing is one step** in the integration testing of OO software.
- Here, a cluster of collaborating classes (determined by examining the CRC and objectrelationship model) is exercised by **designing test cases** that attempt to uncover errors in the collaborations.

VALIDATION TESTING

- Validation testing begins at the **culmination of integration testing**, when individual components have been exercised, the **software is completely assembled as a package**, and interfacing errors have been uncovered and corrected.
- At the **validation or system level,** the distinction between different software categories disappears.
- Validation can be **defined in many ways,** but a simple (albeit harsh) definition is that 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 tests that demonstrate conformity with requirements.
- A **test plan outlines the classes of tests** to be conducted, and a test procedure defines specific test cases that are designed to ensure that all functional requirements are satisfied, **all behavioral characteristics are achieved**, all content is accurate and properly presented, all performance requirements are attained, **documentation is correct, and usability and other requirements are met** (e.g., transportability, compatibility, error recovery, maintainability).
- If a deviation from **specification is uncovered**, 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 activities.**
- The confi guration review, sometimes called an **audit.**

Alpha and Beta 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.**
- The beta test is conducted at **one or more end-user sites.**
- Unlike alpha testing, the **developer generally is not present.**
- Therefore, the beta test is a **"live" application of the software in an environment that cannot be** controlled by the developer.
- The customer **records all problems** (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals.
- A variation on beta testing, called **customer acceptance testing** , is sometimes performed when custom **software is delivered to a customer under contract.**

Alpha Testing	Beta Testing
It is done by internal testers of the organization.	It is done by real users.
It is an internal test, performed within the organization.	It is an external test, carried out in the user's environment.
Alpha Testing uses both black box and white box testing techniques	Beta Testing only uses the black box testing technique.
Identifies possible errors.	Checks the quality of the product.
Developers start fixing bugs as soon as they are identified.	Errors are found by users and feedback is necessary.
Long execution cycles.	It only takes a few weeks.
It can be easily implemented as it is done before the near end of development.	It will be implemented in the future version of the product.
It is performed before Beta Testing.	It is the final test before launching the product on the market.
It answers the question: Does the product work?	It answers the question: Do customers like the product?
Functionality and usability are tested.	Usability, functionality, security and reliability are tested with the same depth.

SYSTEM TESTING

- **System testing** verifies that all elements mesh properly and that overall **system function/performance** is achieved.
- It includes
  - **Recovery Testing**
  - **Security Testing**
  - **Stress Testing**
  - **Performance Testing**
  - **Deployment Testing**

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 is automatic** (performed by the system itself), reinitialization, check pointing mechanisms, data recovery, and **restart are evaluated for correctness.**
- 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, in fact, 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."

Stress Testing

- **Stress tests** are designed to confront programs with abnormal situations.
- 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.**
- In some situations (the most common occur in mathematical algorithms), a **very small range of data** contained within the bounds of valid data for a program may cause extreme and even **erroneous processing or profound** performance degradation.

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.**
- Performance tests are often **coupled with stress testing** and usually require both hardware and software instrumentation.
- That is, it is often necessary to **measure resource utilization** (e.g., processor cycles) in an exacting fashion.

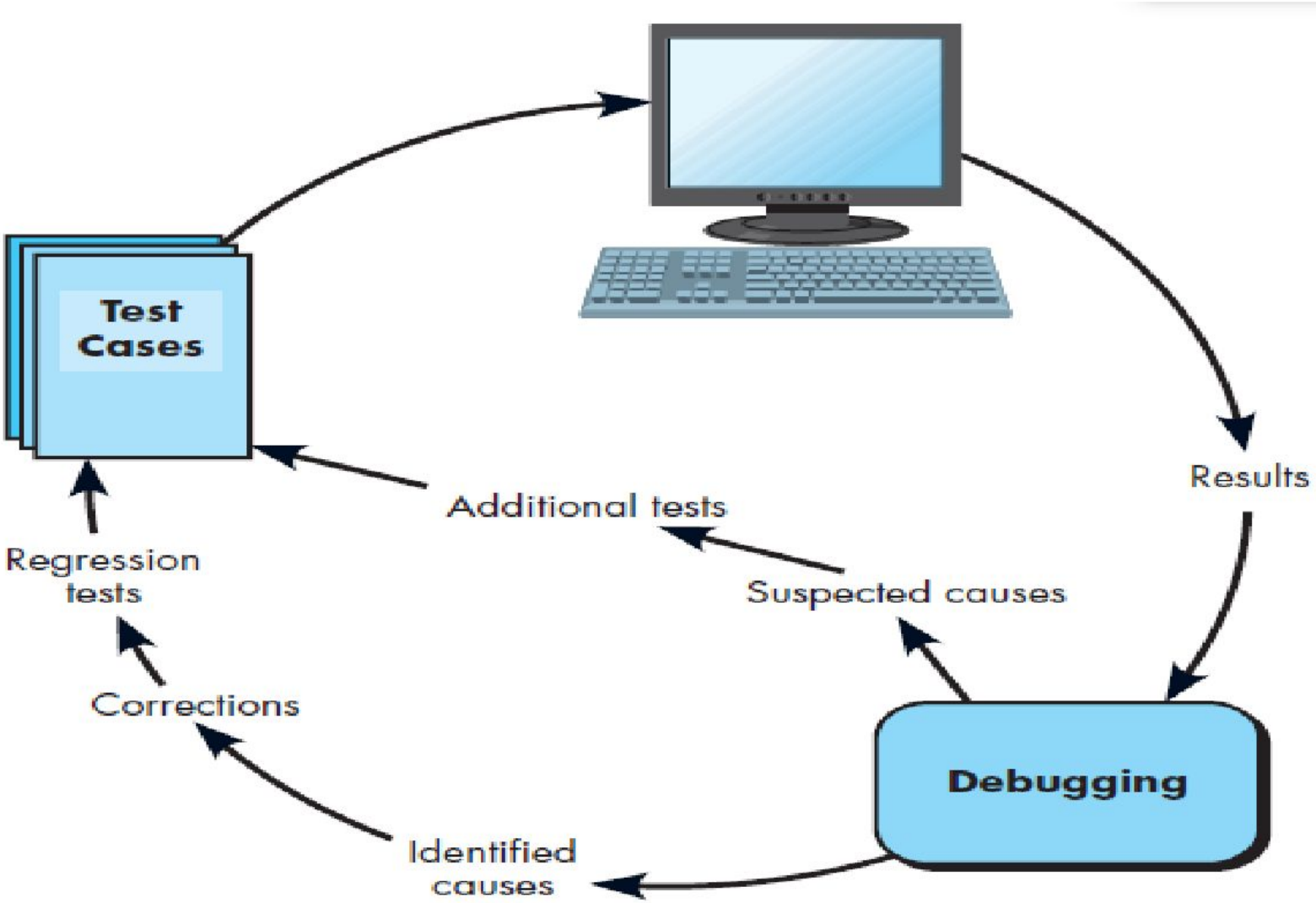
Deployment Testing

- In many cases, **software must execute on a variety of platforms** and under more than one **operating system** environment.
- Deployment testing, sometimes called **configuration testing**, exercises the software in each environment in which it is to operate.
- In addition, **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.

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.
- Debugging is **not testing** but often occurs as a consequence of testing.
- The debugging process begins with the **execution of a test case.**
- During debugging, we encounter **errors that range from mildly annoying** (e.g., an incorrect output format) to **catastrophic** (e.g., the system fails, causing serious economic or physical damage).
- As the consequences of an **error increase**, the amount of pressure to **find the cause also increases**.

Debugging Process



Debugging Process

- Results are assessed and a lack of correspondence between **expected and actual performance** is encountered.
- In many cases, the non corresponding 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 usually have one of **two outcomes**:

(1) the cause will be **found and corrected** or

(2) the cause **will not be found**.

In the latter case, the person performing **debugging may suspect a cause,** design a test case to help **validate that suspicion**, and work toward error correction in an iterative fashion.

Debugging Strategies

- In general, three debugging strategies have been proposed:
  - **Brute force,**
  - **Backtracking, and**
  - **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**.
- You apply brute force debugging methods when **all else fails**.
- **Backtracking** is a fairly common debugging approach that can be **used successfully in small programs**.
- **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.

WHITE-BOX TESTING

- White-box testing, sometimes called **glass-box testing or structural testing or clear 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

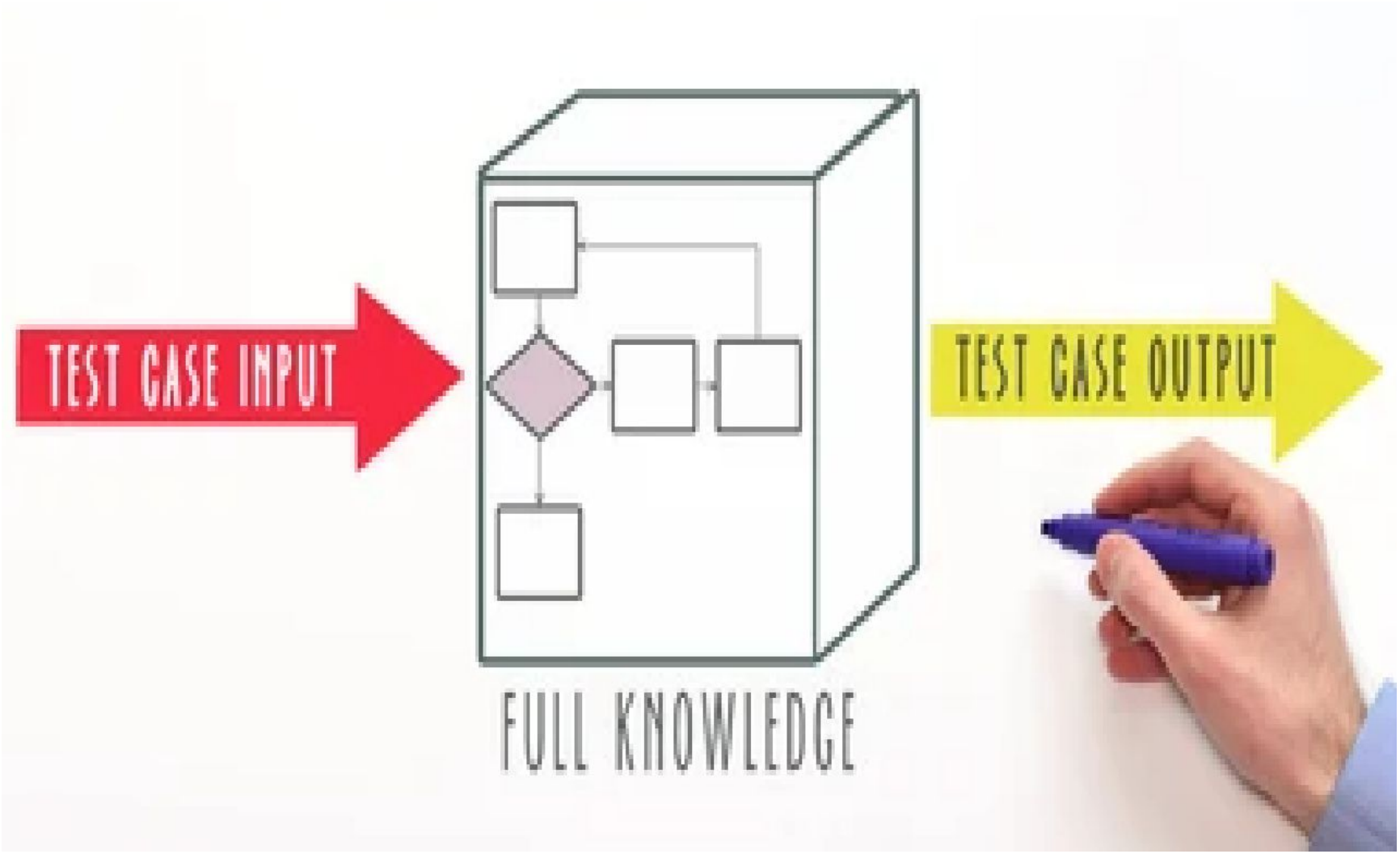
(1) guarantee that **all independent paths** within a module have been exercised at least once,

(2) exercise **all logical decisions** on their true and false sides,

(3) execute **all loops at their boundaries** and within their operational bounds, and

(4) exercise **internal data structures** to ensure their validity.

WHITE-BOX TESTING



BASIS PATH TESTING

- Basis path testing is a **white-box testing** technique.
- 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**

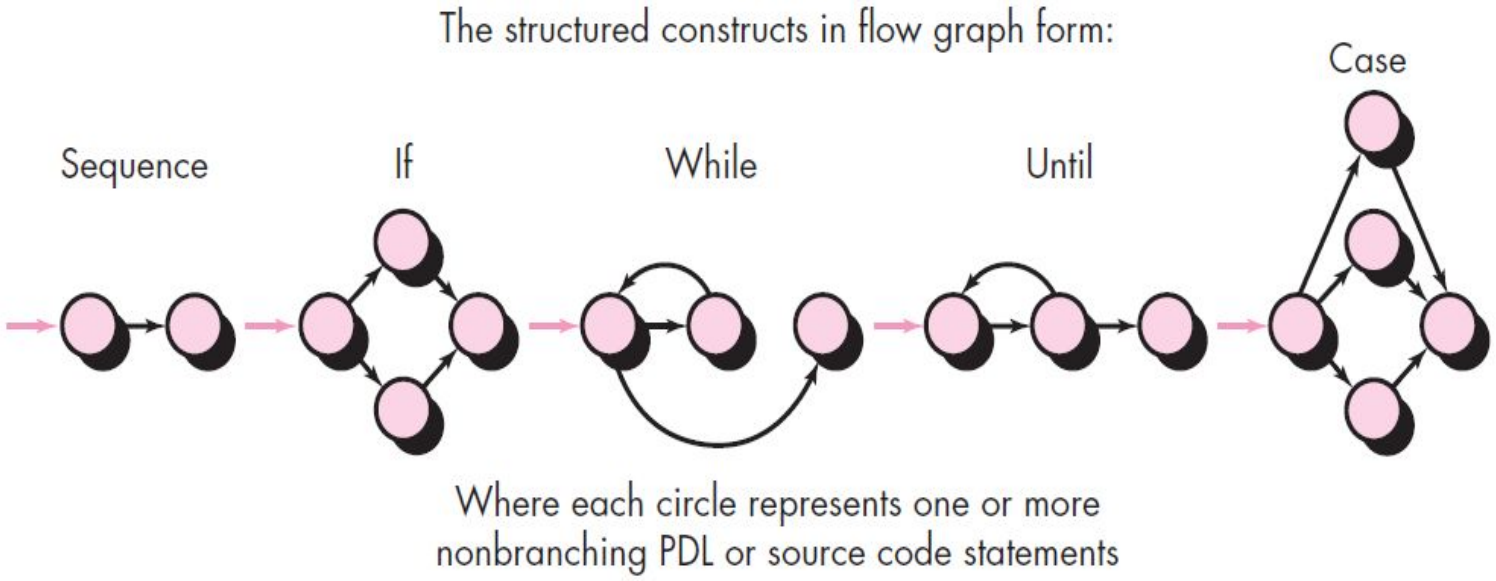
✔ **Independent Program Paths**

✔ **Deriving Test Cases**

✔ **Graph Matrices**

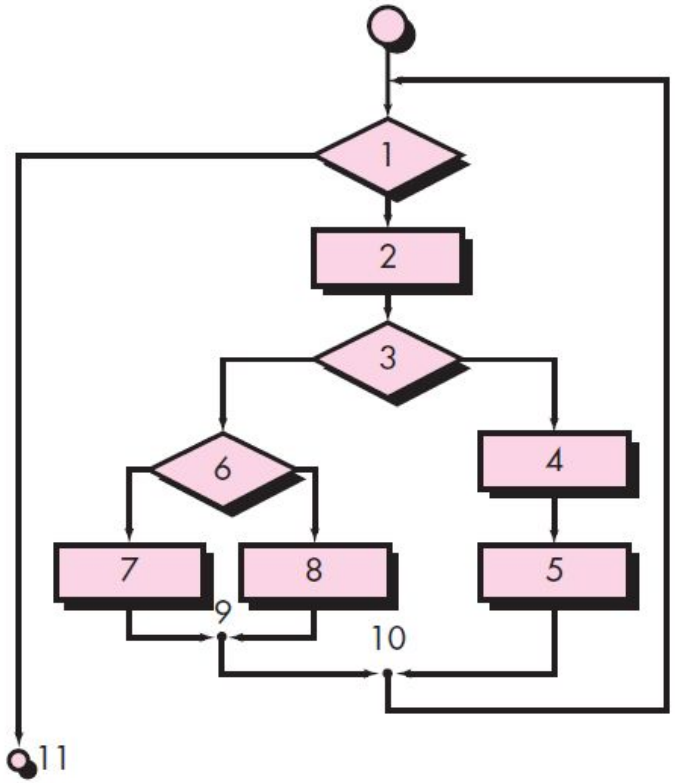
Flow Graph Notation

- 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.
- The flow graph depicts **logical control flow**.



Flow Graph Notation

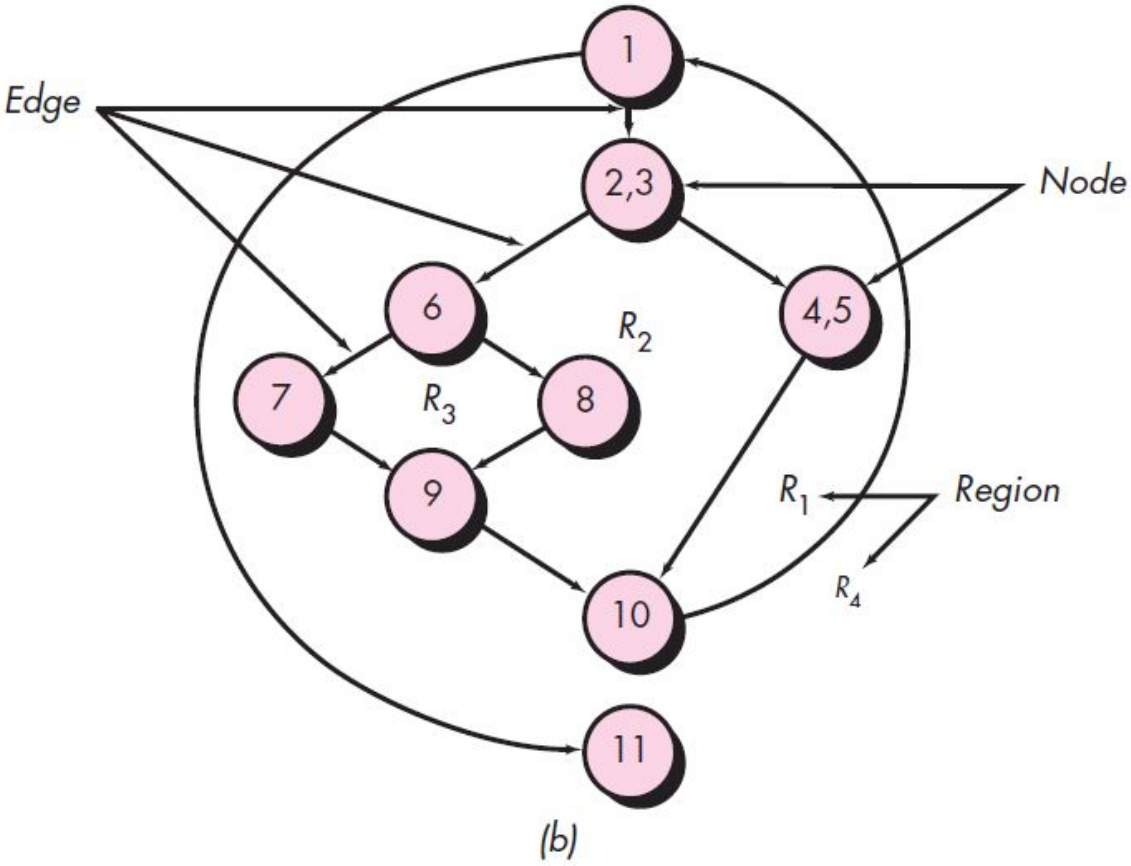
- To illustrate the use of a flow graph, consider the procedural design representation in Figure.
- Here, a **flowchart** is used to depict **program control structure**.



(a)

Flow Graph Notation

- The figure maps the **flowchart** into a corresponding **flow graph**



Flow Graph Notation

- Each **circle**, called a **flow graph node**, represents one or more procedural statements.
- A sequence of **process boxes and a decision diamond** can map into a **single node**.
- The **arrows** on the flow graph, called **edges or links**, represent **flow of control**.
- An edge must **terminate at a node**.
- **Areas bounded** by edges and nodes are called **regions**.

Independent Program Paths

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

  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

Cyclomatic complexity

- Cyclomatic complexity is a **software metric** that provides a **quantitative measure** of the logical complexity of a program.
- Cyclomatic complexity has a **foundation in graph theory** and provides you with an extremely useful software metric.
- The number of **regions of the flow graph** corresponds to the cyclomatic complexity.
- 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**.

The cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has **four regions**.
2. V (G)=E-N+2 **11 edges - 9 nodes +2 = 4**

Deriving Test Cases

- The **basis path testing** method can be applied to a **procedural design or to source code.**
- The following steps can be applied to derive the basis set:

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.

- 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.

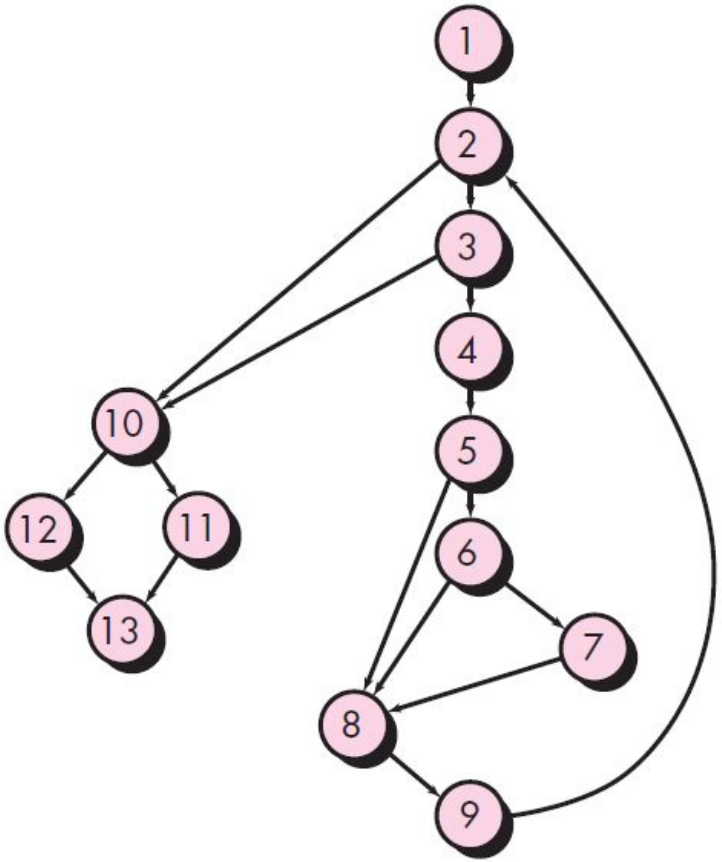
```
PROCEDURE average;

      *  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;
      i = 1;
      total.input = total.valid = 0;
      sum = 0;
      DO WHILE value[i] <> -999 AND total.input < 100
         increment total.input by 1;
         IF value[i] > = minimum AND value[i] < = maximum
            THEN increment total.valid by 1;
                 sum = s sum + value[i]
            ELSE skip
         ENDIF
         increment i by 1;
      ENDDO
      IF total.valid > 0
         THEN average = sum / total.valid;
         ELSE average = -999;
      ENDIF
END average
```

Determine a basis set of linearly 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-. . .

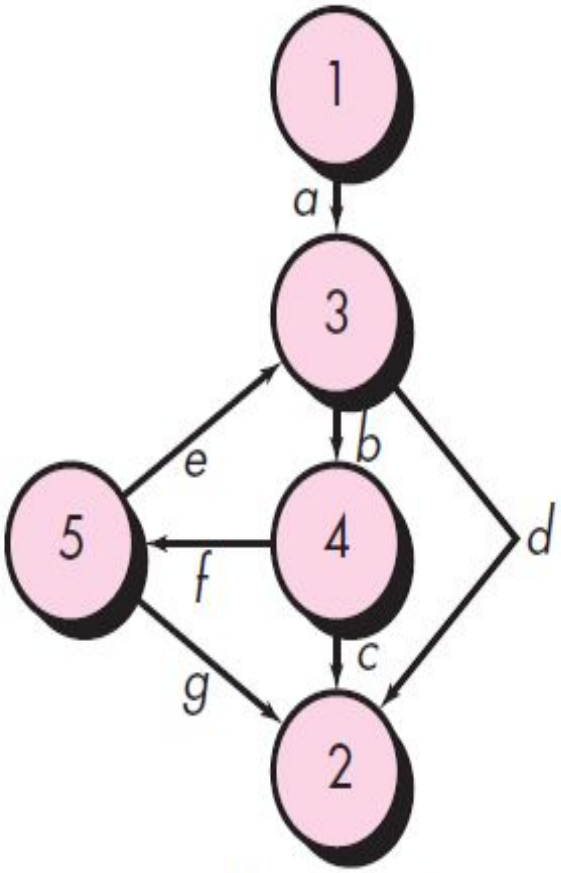
Path 5: 1-2-3-4-5-6-8-9-2-. . .

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

Graph Matrices

- The procedure for **deriving the flow graph** and even determining a **set of basis paths is** amenable to mechanization.
- A data structure, called a **graph matrix**, can be quite useful for developing a software tool that assists in **basis path testing**.
- A graph matrix is a **square matrix** whose size (i.e., number of rows and columns) is equal to the **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.
- Using these techniques, the **analysis required to design test cases** can be **partially or fully automated**.

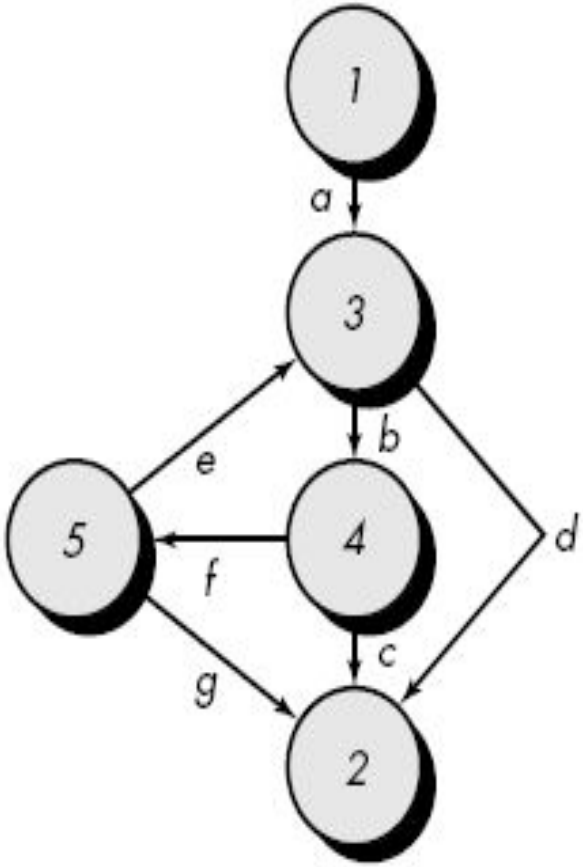
Graph Matrices



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

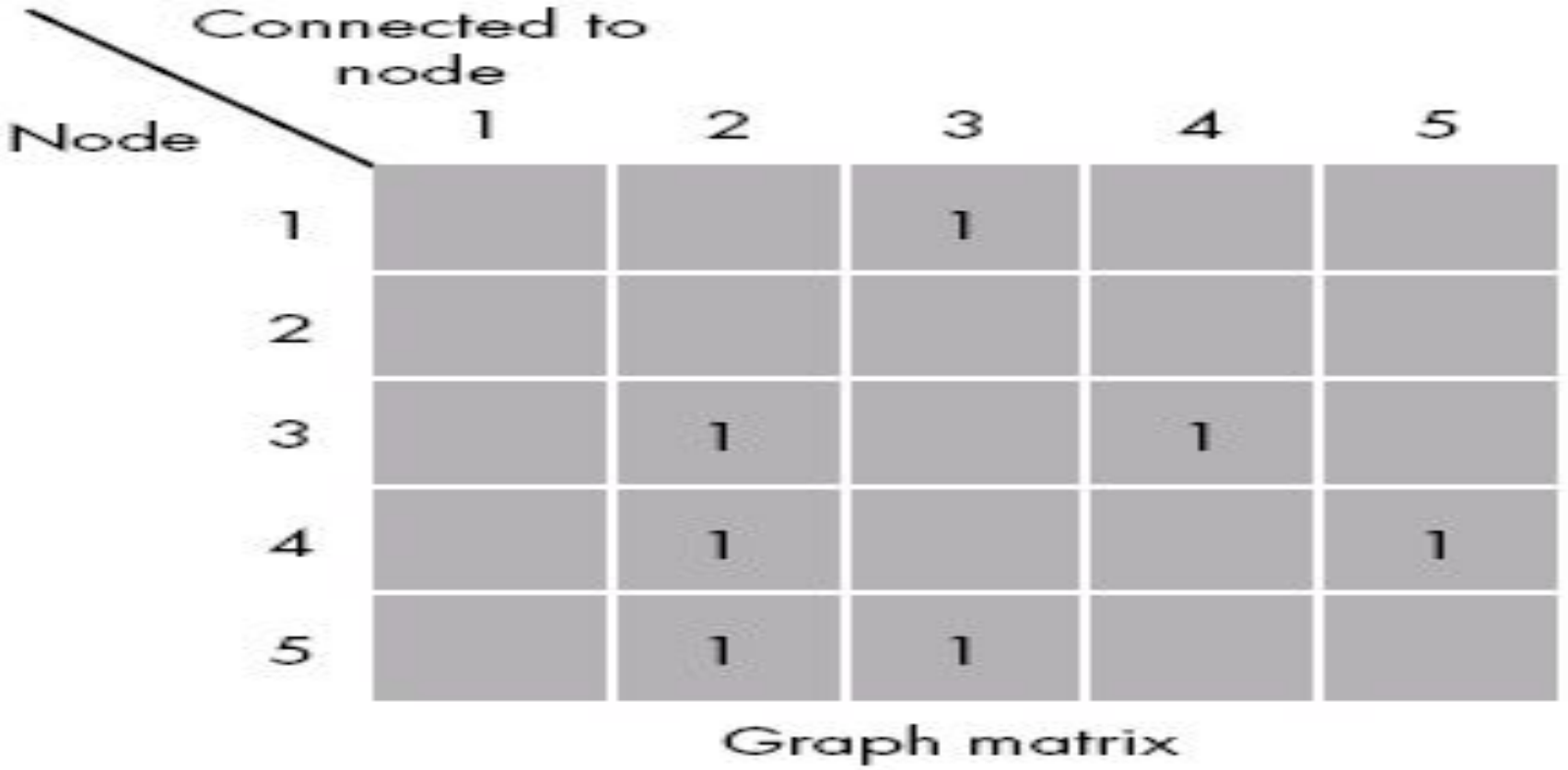


Flow graph

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

Graph matrix

Connection matrix



Graph matrix

- Each letter has been replaced with a 1, indicating that a connection exists (this graph matrix is called a *connection matrix).*
- In fig.(connection matrix) each row with two or more entries represents a predicate node.
- We can directly measure cyclomatic complexity value by performing arithmetic operations
- Connections = Each row Total no. of entries – 1.
- V(G)= Sum of all connections + 1

CONTROL STRUCTURE TESTING

- The basis path testing technique is one of a number of techniques for **control structure testing**.
- It includes,

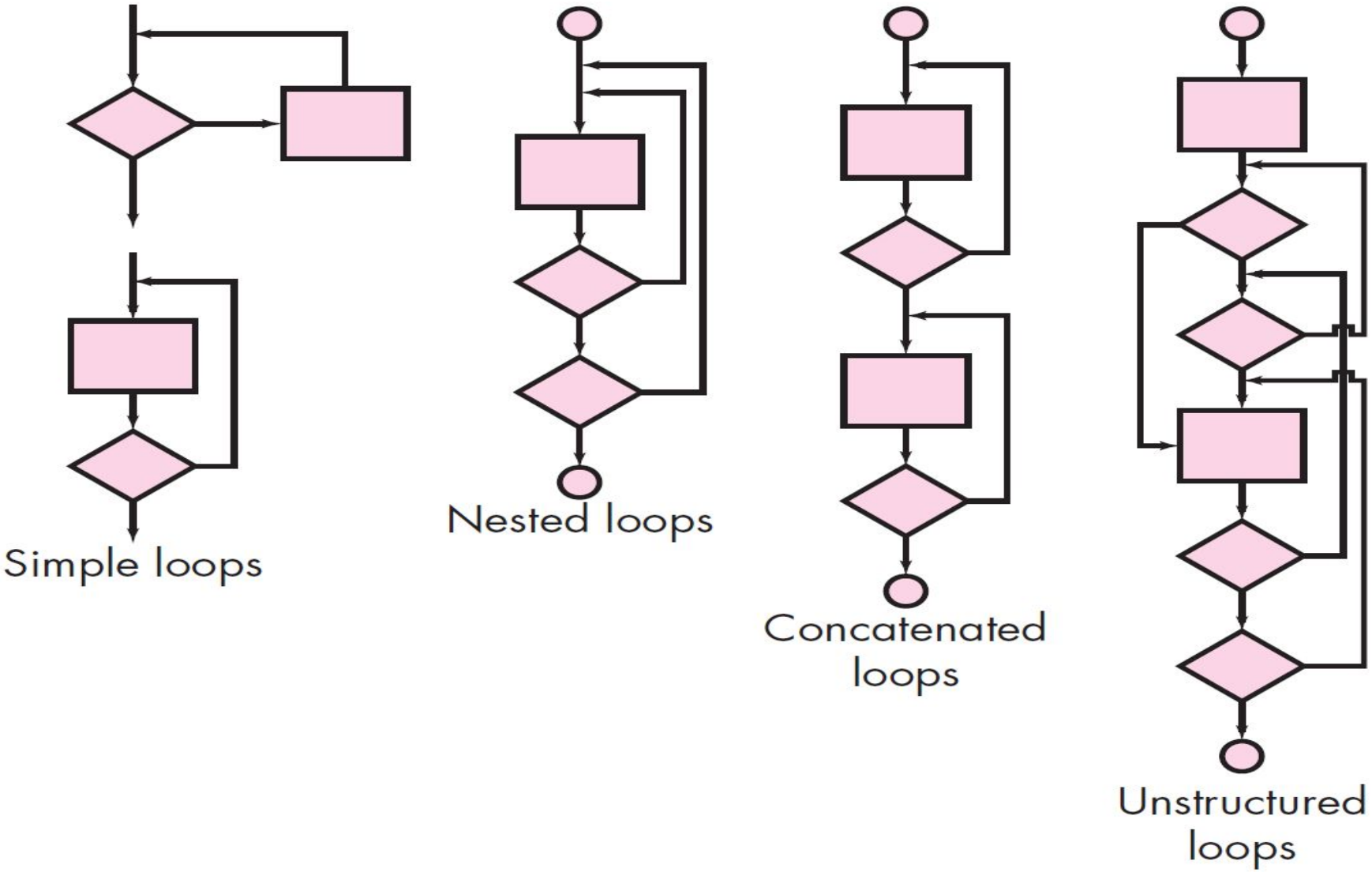
✔ **Condition testing**

✔ **Data flow testing**

✔ **Loop testing**

- **Condition testing** is a test-case design method that exercises the **logical conditions** contained in a program module.
- **Data flow testing** selects **test paths of a program** according to the **locations of definitions** and uses of variables in the program.
- **Loop testing** is a white-box testing technique that focuses exclusively on the **validity of loop constructs**.

Classes of Loops



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.***

Nested loops

Approach that 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. 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

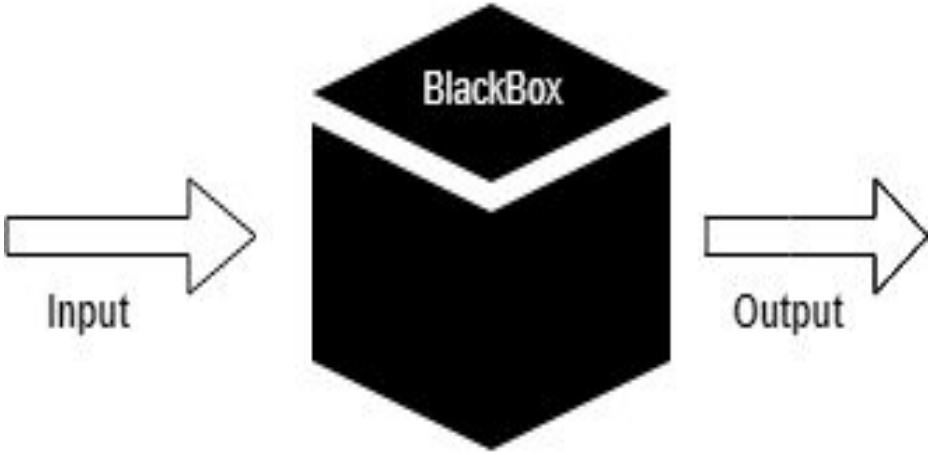
- Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
- However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
- When the loops are not independent, the approach applied to nested loops is recommended.

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 or functional testing**, focuses on the **functional requirements of the software**.
- That is, black-box testing techniques enable you to **derive sets of input conditions** that will fully exercise all **functional requirements** for a program.



BLACK-BOX TESTING

- Black-box testing attempts to **find errors** in the following categories:

(1) **Incorrect or missing** functions,

(2)Interface errors,

(3) Errors in **data structures or external** database access, (4) Behavior or **performance errors**, and

(5) Initialization and termination errors.

BLACK-BOX TESTING

Tests are **designed to answer** the following questions:

- How is **functional validity** tested?
- How are **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**?

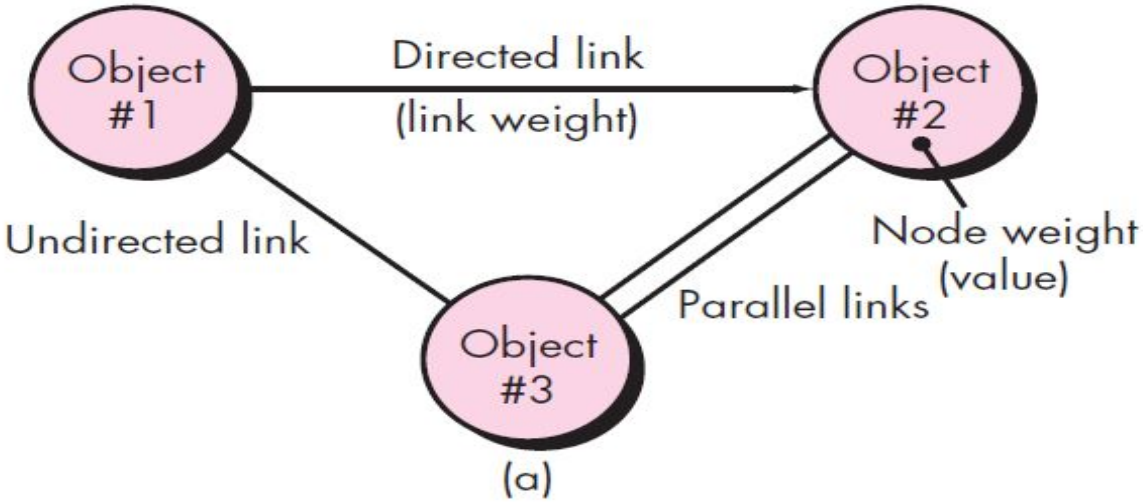
BLACK-BOX TESTING

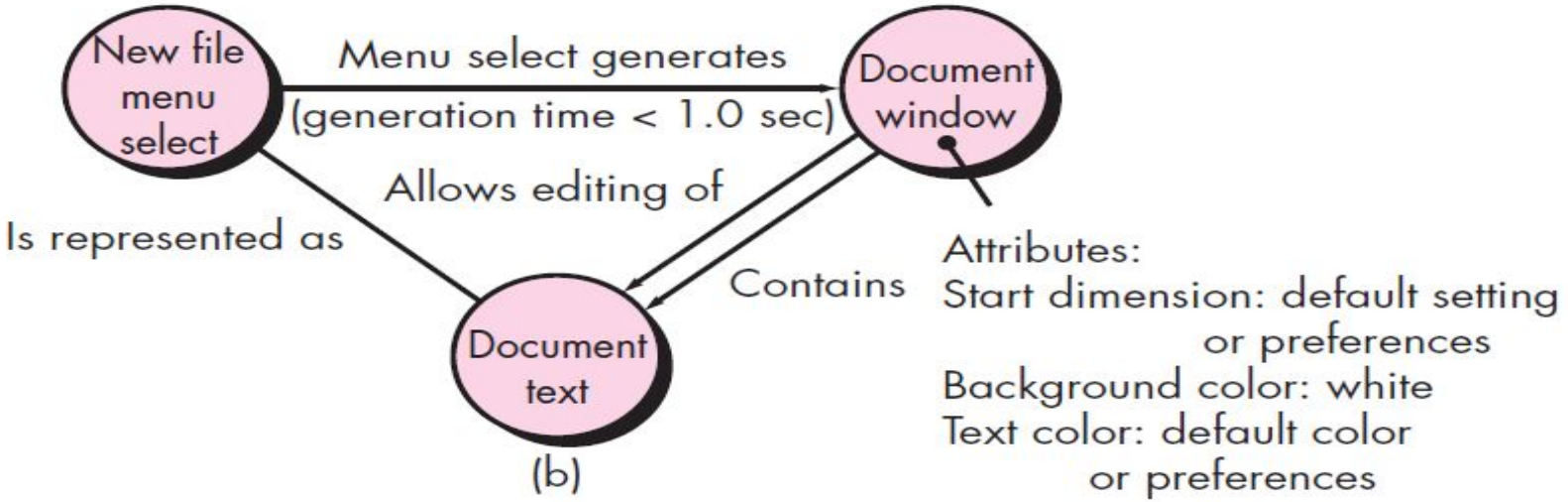
- Graph-Based Testing Methods
- Equivalence Partitioning
- Boundary Value Analysis
- Orthogonal Array Testing

Graph-Based Testing Methods

- The first step in black-box testing is to **understand the objects** that are modeled in software and the relationships that connect these objects.
- Once this has been accomplished, the next step is to **define a series of tests that verify** "all objects have the **expected relationship to one another**".
- To accomplish these steps, you begin by **creating a graph** —a **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**.

Graph Notation





Graph-Based Testing Methods

- **Nodes** are represented as **circles** connected by links that take a number of different forms.
- 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.

Graph-Based Testing Methods

- **Transaction flow modeling:**

The **nodes represent steps** in some transaction and the **links represent the logical** connection between steps.

- **Finite state modeling:**

The **nodes represent** different user-observable states of the software and the **links represent the transitions** that occur to move from state to state.

- **Data flow modeling:**

The **nodes are data objects**, and the **links are the transformations** that occur to translate one data object into another.

- **Timing modeling:**

The **nodes are program objects**, and the **links are the sequential connections** between those objects.

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.
- In this method, the **input domain data is divided** into different **equivalence data classes**.
- This method is typically used to **reduce the total number of test cases** to a finite set of testable test cases, still covering maximum requirements.
- In short, it is the **process of taking all possible test cases** and placing them into classes. One test value is picked from each class while testing.
- 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**.

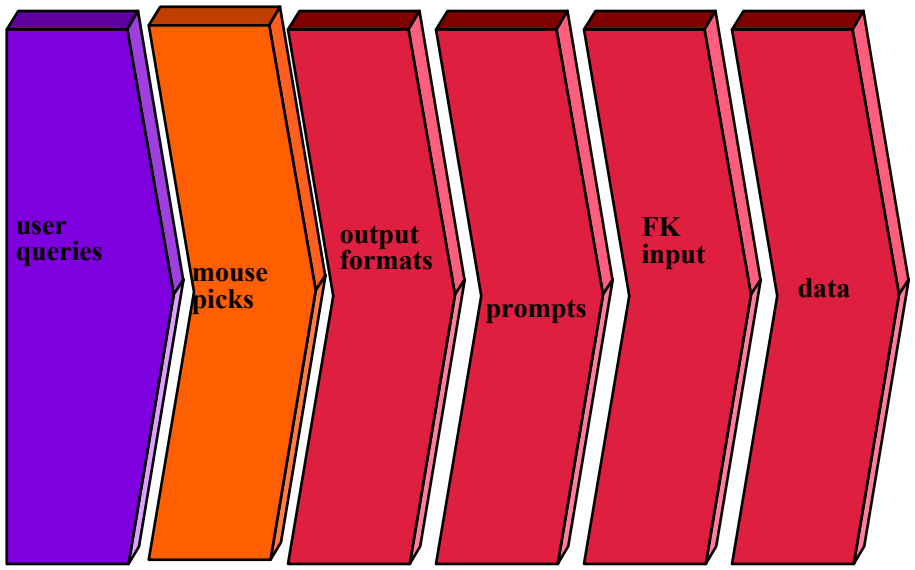
Equivalence Partitioning

- 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.

Equivalence Partitioning



Example

- If you are **testing for an input box accepting** numbers from **1 to 1000** then there is no use in writing thousands of test cases for all **1000 valid input numbers** plus other test cases for invalid data.
- Using the **Equivalence Partitioning** method above, test cases can be **divided into three sets of input data called classes**. Each test case is a representative of the respective class.
- So in the above example, we can divide our test cases into three equivalence classes of **some valid and invalid inputs.**

Test Cases

#1) One input data **class with all valid inputs**. Pick a single value from the range of 1 to 1000 as a **valid test case**. If you select other values between **1 and 1000**, then the result is going to be the same. So one test case for **valid input data should be sufficient.**

#2) Input data **class with all values below the lower limit**. i.e., any value below 1, as an **invalid input data test case**.

#3) Input **data with any value greater than 1000** to represent the **third invalid input class**.

- So using Equivalence Partitioning,we have **categorized all possible test cases into three classes**. Test cases with other values from any class should give the same result.
- Equivalence Partitioning uses the **fewest test cases to cover the maximum** requirements.

Boundary Value Analysis

- A greater number of errors occurs at the **boundaries of the input domain** rather than in the **"center."**
- It is for this reason that boundary value analysis (BVA) has been **developed as a testing technique**.
- Boundary value analysis leads to a selection of test cases that **exercise bounding values**.
- 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.

Boundary Value Analysis

- 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.**

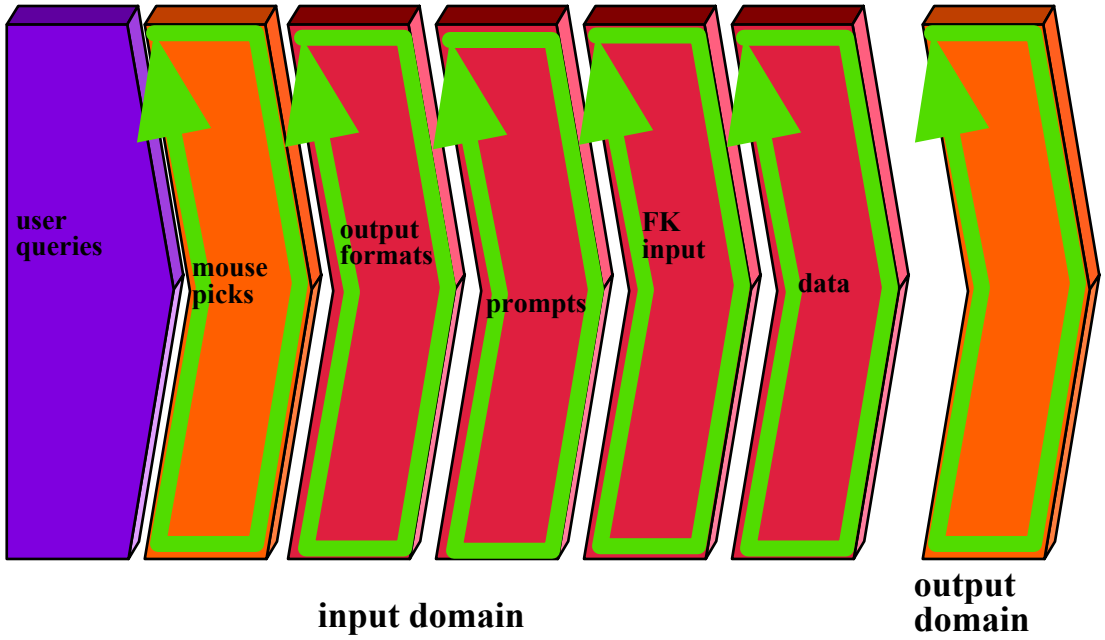
2. If an **input condition specifies a number of values**, test cases should be developed that exercise the **minimum and maximum** numbers. Values just **above and below** minimum and maximum are also tested.

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.

Boundary Value Analysis



Example

- Test cases for input box accepting numbers between 1 and 100 using Boundary value analysis:

#1) Test cases with test data exactly as the **input boundaries of input domain** i.e. values 1 and 100 in our case.

#2) Test data with values just **below the extreme edges of input domains** i.e. values 0 and 99.

#3) Test data with values just a**bove the extreme edges of the input domain** i.e. values 2 and 101.

- Instead of using all the values from 1 to 100, we just use 0, 1, 2, 99, 100, and 101.

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.
- 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 **$3^3 = 27$ possible test cases**.

Orthogonal Array Testing

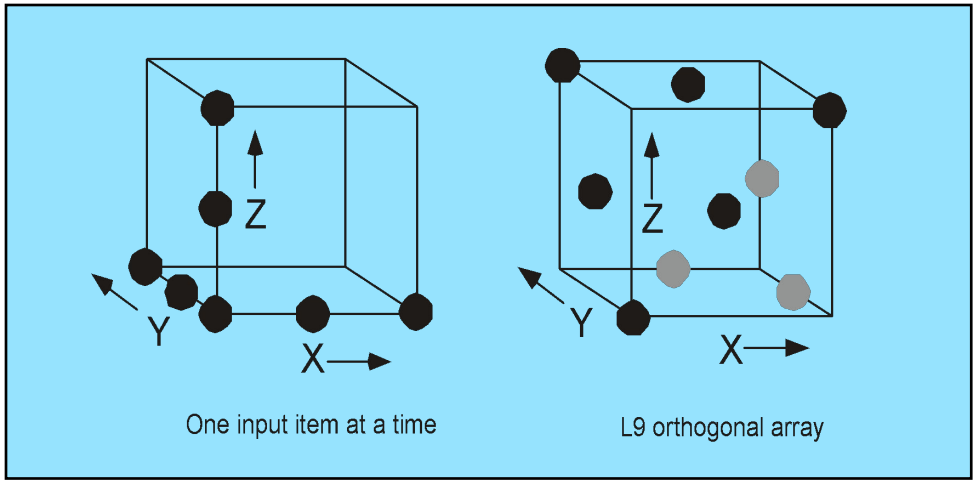
- The orthogonal Array Testing technique is a **statistical approach for testing pair-wise interactions**.
- Most of the defects have observed are caused due to **interaction and integration.**
- This interaction or integration can be within **different objects, elements, options on a screen of the application**, or configuration settings in a file.
- It is obvious that some of the **combinations are missed to test** which thereby results in insufficient tests.
- Hence in order to **cover the entire functionality** in the testing scope with the correct amount of combinations to be tested, Orthogonal Array Testing is used.
- The beauty of this technique is that it **maximizes the coverage by a comparatively lesser number of test cases**.
- The point here is to identify the **correct pair of input parameters**.

Terminologies for Orthogonal Array Testing

- **Runs**: It is the number of **rows which represents the number of test conditions** to be performed.
- **Factors:** It is the **number of columns which represents in the number of variable** to be tested
- **Levels:** It represents the **number of values for a Factor**
- As the rows represent the number of test conditions (experiment test) to be performed, the goal is to **minimize the number of rows** as much as possible.
- Factors indicate the **number of columns**, which is the number of variables.
- Levels represent the **maximum number of values** for a factor

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



Black Box Testing	White Box Testing
It is a testing method without having knowledge about the actual code or internal structure of the application.	It is a testing method having knowledge about the actual code and internal structure of the application.
This is a higher level testing such as functional testing.	This type of testing is performed at a lower level of testing such as Unit Testing, Integration Testing.
It concentrates on the functionality of the system under test.	It concentrates on the actual code – program and its syntax's.
Black box testing requires Requirement specification to test.	White Box testing requires Design documents with data flow diagrams, flowcharts etc.
Black box testing is done by the testers.	White box testing is done by Developers or testers with programming knowledge.