**Strategic Approach to Software Testing**

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.

A number of software testing strategies have been proposed in the literature. All provide the software developer with a template for testing and all have the following generic characteristics:

* Testing begins at the component level2 and works "outward" toward the integration of the entire computer-based system.
* Different testing techniques are appropriate 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.

**Unit Testing**

Unit testing is a type of software testing that focuses on individual units or components of a software system. The purpose of unit testing is to validate that each unit of the software works as intended and meets the requirements. Unit testing is typically performed by developers, and it is performed early in the development process before the code is integrated and tested as a whole system.

Unit tests are automated and are run each time the code is changed to ensure that new code does not break existing functionality. Unit tests are designed to validate the smallest possible unit of code, such as a function or a method, and test it in isolation from the rest of the system. This allows developers to quickly identify and fix any issues early in the development process, improving the overall quality of the software and reducing the time required for later testing.

**Unit Testing** is a software testing technique using which individual units of software i.e. group of computer program modules, usage procedures, and operating procedures are tested to determine whether they are suitable for use or not. It is a testing method using which every independent module is tested to determine if there is an issue by the developer himself. It is correlated with the functional correctness of the independent modules. Unit Testing is defined as a type of software testing where individual components of a software are tested. Unit Testing of the software product is carried out during the development of an application. An individual component may be either an individual function or a procedure. Unit Testing is typically performed by the developer. In SDLC or V Model, Unit testing is the first level of testing done before integration testing. Unit testing is a type of testing technique that is usually performed by developers. Although due to the reluctance of developers to test, quality assurance engineers also do unit testing

### Objective of Unit Testing:

The objective of Unit Testing is:

1. To isolate a section of code.
2. To verify the correctness of the code.
3. To test every function and procedure.
4. To fix bugs early in the development cycle and to save costs.
5. To help the developers understand the code base and enable them to make changes quickly.
6. To help with code reuse.

### Objective of Unit Testing:

The objective of Unit Testing is:

1. To isolate a section of code.
2. To verify the correctness of the code.
3. To test every function and procedure.
4. To fix bugs early in the development cycle and to save costs.
5. To help the developers understand the code base and enable them to make changes quickly.
6. To help with code reuse.

### Disadvantages of Unit Testing:

1. The process is time-consuming for writing the unit test cases.
2. Unit Testing will not cover all the errors in the module because there is a chance of having errors in the modules while doing integration testing.
3. Unit Testing is not efficient for checking the errors in the UI(User Interface) part of the module.
4. It requires more time for maintenance when the source code is changed frequently.

# **Integration Testing**

**Integration testing** is the process of testing the interface between two software units or modules. It focuses on determining the correctness of the interface. The purpose of integration testing is to expose faults in the interaction between integrated units. Once all the modules have been unit-tested, integration testing is performed.

Integration testing is a software testing technique that focuses on verifying the interactions and data exchange between different components or modules of a software application. The goal of integration testing is to identify any problems or bugs that arise when different components are combined and interact with each other. Integration testing is typically performed after unit testing and before system testing. It helps to identify and resolve integration issues early in the development cycle, reducing the risk of more severe and costly problems later on.

Integration testing can be done by picking module by module. This can be done so that there should be a proper sequence to be followed. And also if you don’t want to miss out on any integration scenarios then you have to follow the proper sequence. Exposing the defects is the major focus of the integration testing and the time of interaction between the integrated units.

**Top-down Integration**

Top-down integration testing is an incremental approach to construction of program structure. 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. Referring to Figure 18.6, depth-first integration would integrate all components on a major control path of the structure. Selection of a major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the lefthand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and righthand 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. The integration process is performed in a series of five steps:

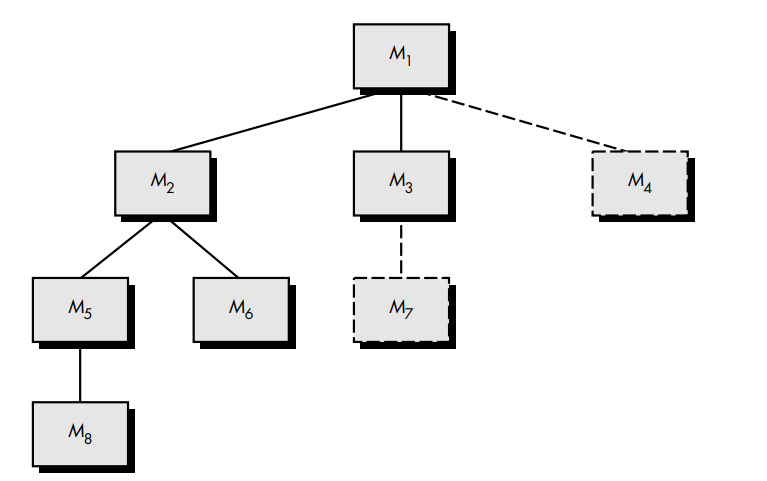
1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.

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.



**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). Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

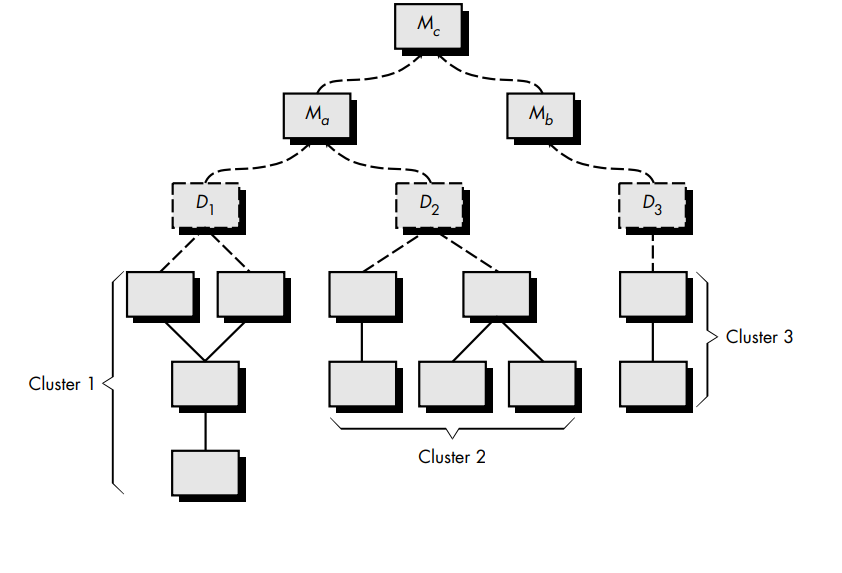
A bottom-up integration strategy may be implemented with the following steps:

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

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



**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. These changes may cause problems with functions that previously worked flawlessly. In the context of an integration test strategy, regression testing is the reexecution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. In a broader context, successful tests (of any kind) result in the discovery of errors, and errors must be corrected.

Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed. Regression testing is the activity that 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 re-executing a subset of all test cases or using automated capture/playback tools. Capture/playback tools enable the software engineer to capture test cases and results for subsequent playback and comparison.

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 “shrinkwrapped” software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis. In essence, 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.

| **Verification** | **Validation** |
| --- | --- |
| Are we building the system right? | Are we building the right system? |
| Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements. | Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements. |
| The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications. | The objective of Validation is to make sure that the product actually meet up the user’s requirements, and check whether the specifications were correct in the first place. |
| Following activities are involved in Verification: Reviews, Meetings and Inspections. | Following activities are involved in Validation: Testing like black box testing, white box testing, gray box testing etc. |
| Verification is carried out by QA team to check whether implementation software is as per specification document or not. | Validation is carried out by testing team. |
| Execution of code is not comes under Verification. | Execution of code is comes under Validation. |
| Verification process explains whether the outputs are according to inputs or not. | Validation process describes whether the software is accepted by the user or not. |
| Verification is carried out before the Validation. | Validation activity is carried out just after the Verification. |
| Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc, | Following item is evaluated during Validation: Actual product or Software under test. |
| Cost of errors caught in Verification is less than errors found in Validation. | Cost of errors caught in Validation is more than errors found in Verification. |
| It is basically manually checking the of documents and files like requirement specifications etc. | It is basically checking of developed program based on the requirement specifications documents & files. |

**WHITE-BOX TESTING**

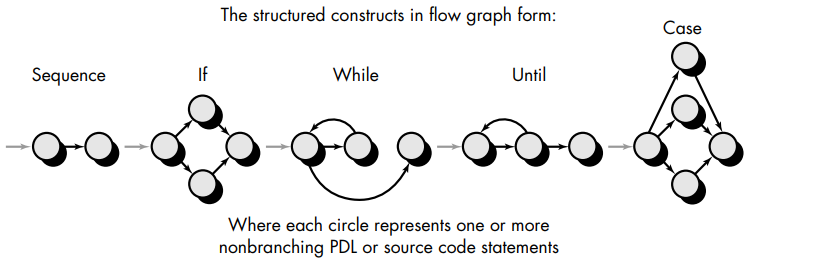
White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that (1) guarantee that all independent paths within a module 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, and (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. 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. 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.3 The flow graph depicts logical control flow using the notation illustrated in Figure



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

**Cyclomatic Complexity**

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used 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. For example, a set of independent paths for the flow graph illustrated in Figure is

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

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph correspond 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.

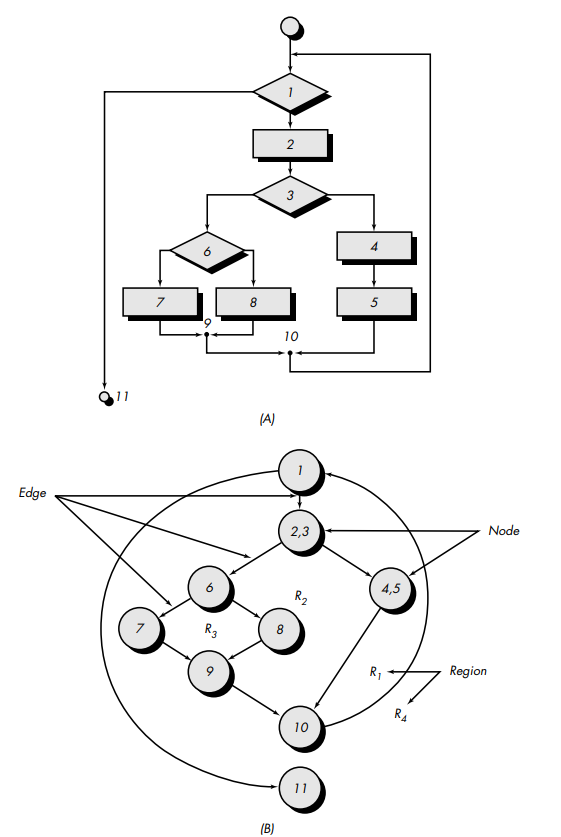
3. Cyclomatic complexity, V(G), for a flow graph, G, is also defined as V(G) = P + 1

where P is the number of predicate nodes contained in the flow graph G. Referring once more to the flow graph in Figure 17.2B, 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.



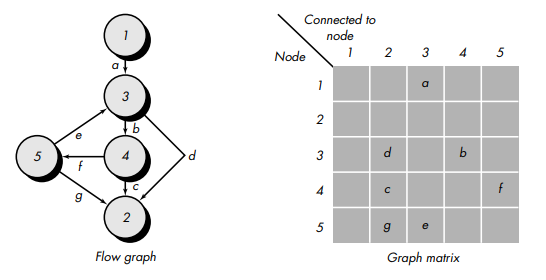
**Deriving Test Cases**

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure average, depicted in PDL in Figure 17.4, will be used as an example to illustrate each step in the test case design method. Note that average, although an extremely simple algorithm, contains compound conditions and loops. 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.

**Graph Matrices**

The procedure for deriving the flow graph and even determining a set of basis paths is amenable to mechanization. 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 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. A simple example of a flow graph and its corresponding graph matrix in Figure

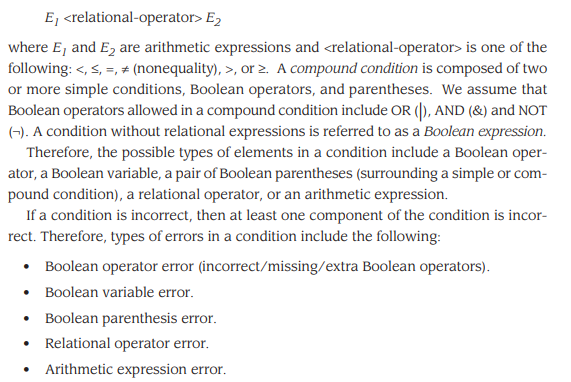


**CONTROL STRUCTURE TESTING**

The basis path testing technique is one of a number of techniques for control structure testing. Although basis path testing is simple and highly effective, it is not sufficient in itself. In this section, other variations on control structure testing are discussed. These broaden testing coverage and improve quality of white-box testing.

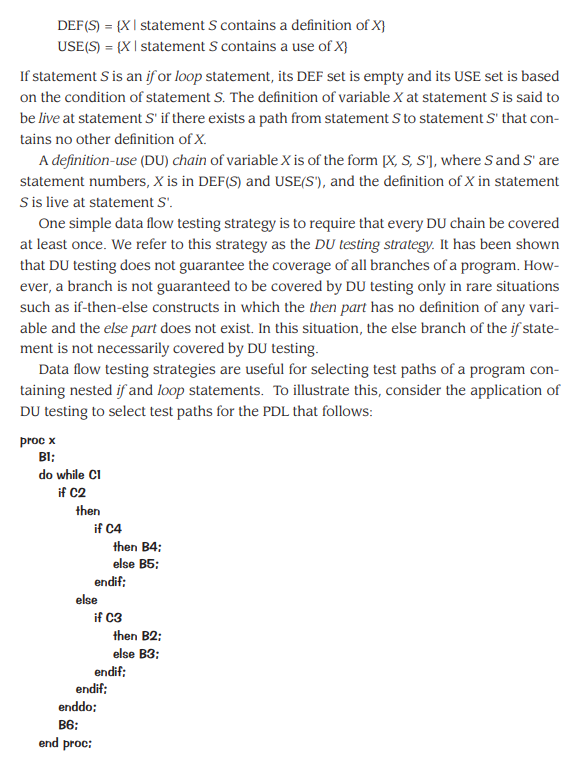
**Condition Testing**

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



**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. A number of data flow testing strategies have been studied and compared. 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,



**Loop Testing**

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. Loop testing 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.

**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**

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. 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.

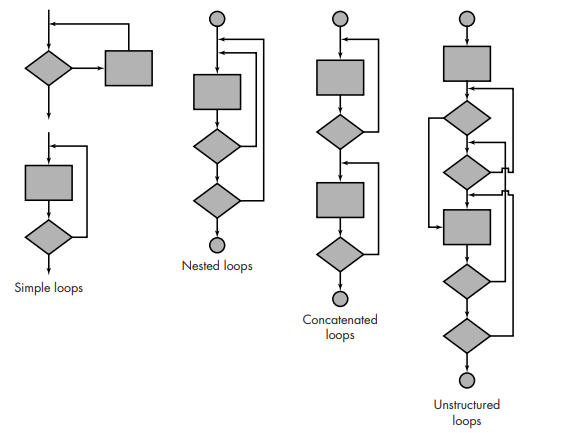
4. Continue until all loops have been tested.

**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 behavioural testing, focuses on the functional requirements of the software. That is, 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 is not an alternative to white-box techniques. Rather, it is a complementary app 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 data base access, (4) behaviour or performance errors, and (5) initialization and termination errors. Unlike white-box testing, which is performed early in the testing process, black box testing tends to be applied during later stages of testing.

Because black-box testing purposely disregards control structure, attention is focused on the information domain.

Tests are designed to answer the following questions:

• How is functional validity tested?

• How is system behavior and performance tested?

• What classes of input will make good test cases?

• Is the system particularly sensitive to certain input values?

• How are the boundaries of a data class isolated?

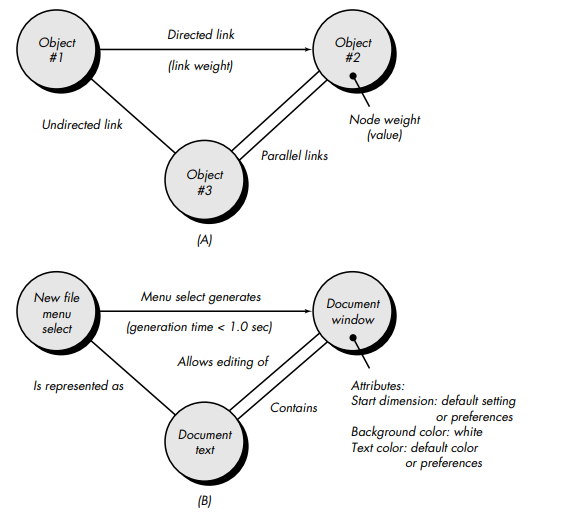
• What data rates and data volume can the system tolerate?

• What effect will specific combinations of data have on system operation?

**Graph-Based Testing Methods**

The first step in black-box testing is to understand the objects6 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.” Stated in another way, 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. To accomplish these steps, the software engineer begins 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.



**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. An ideal test case single-handedly uncovers a class of errors (e.g., incorrect processing of all character data) that might otherwise require many cases to be executed before the general error is observed. Equivalence partitioning strives to define a test case that uncovers classes of errors, thereby reducing the total number of test cases that must be developed.

Test case design for equivalence partitioning is based on an evaluation of equivalence classes for an input condition. Using concepts introduced in the preceding section, 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 either a specific numeric value, a range of values, a set of related values, or a Boolean condition. Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a range, one valid and two invalid equivalence classes are defined. 2. If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.

3. If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined.

4. If an input condition is Boolean, one valid and one invalid class are defined.

**Boundary Value Analysis**

For reasons that are not completely clear, a greater number of errors tends to occur 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. Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:

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

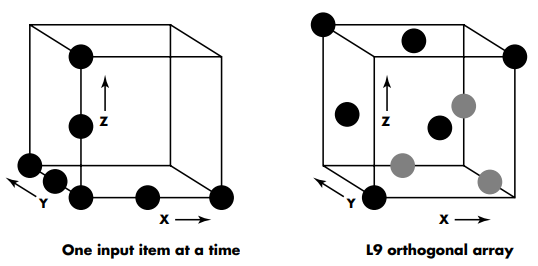
2. If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

3. Apply guidelines 1 and 2 to output conditions. For example, assume that a temperature 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.

**Orthogonal Array**

Testing There are many applications in which the input domain is relatively limited. That is, the number of input parameters is small and the values that each of the parameters may take are clearly bounded. When these numbers are very small (e.g., three input parameters taking on three discrete values each), it is possible to consider every input permutation and exhaustively test processing of the input domain. However, as the number of input values grows and the number of discrete values for each data item increases, exhaustive testing become impractical or impossible. 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 errors associated with region faults—an error category associated with faulty logic within a software component.



## How to do Orthogonal Array Testing: Examples

1. Identify the independent variable for the scenario.
2. Find the smallest array with the number of runs.
3. Map the factors to the array.
4. Choose the values for any “leftover” levels.
5. Transcribe the Runs into test cases, adding any particularly suspicious combinations that aren’t generated.

### Example 1

A Web page has three distinct sections (Top, Middle, Bottom) that can be individually shown or hidden from a user

* No of Factors = 3 (Top, Middle, Bottom)
* No of Levels (Visibility) = 2 (Hidden or Shown)
* Array Type = L4(23)

(4 is the number of runs arrived after creating the OAT array)

If we go for Conventional testing technique, we need test cases like 2 X 3 = 6 Test Cases

