UNIT- IV

SOFTWARE TESTING STRATEGIES:

A Strategic Approach to Software Testing, Test Strategies for Conventional Software and Object Oriented Software, Validation Testing, White- Box Testing, Basis Path Testing, Black-Box Testing, System Testing.

SOFTWARE TESTING STRATEGIES:

A STRATEGIC APPROACH TO SOFTWARE TESTING

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

• 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 develope**r 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 AND VALIDATION:

Verification is a process of checking documents, design, code, and program in order to check if the software has been built according to the requirements or not. The main goal of verification process is to ensure quality of software application, design, architecture etc. The verification process involves activities like reviews, walk-throughs and inspection.

Validation in Software Engineering is a dynamic mechanism of testing and validating if the software product actually meets the exact needs of the customer or not. The process helps to ensure that the software fulfills the desired use in an appropriate environment. The validation process involves activities like unit testing, integration testing, system testing and user acceptance testing..

Boehm states this another way:

Verification: "Are we building the product right?"

Validation: "Are we building the right product?"

Organizing for Software Testing

For every software project, there is an inherent conflict of interest that occurs as testing begins. The people who have built the software are now asked to test the software.

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 does an independent test group become involved.

The role of an ***independent test group*** **(ITG)** is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present. The developer and the ITG work closely throughout a software project to ensure that thorough tests will be conducted. While testing is conducted, the developer must be available to correct errors that are uncovered.

Software Testing Strategy—The Big Picture

The software process may be viewed as the spiral illustrated in following figure. Initially, system engineering defines the role of software and leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for software are established. Moving inward along the spiral, you come to design and finally to coding. To develop computer software, you spiral inward (counter clockwise) along streamlines that decrease the level of abstraction on each turn.

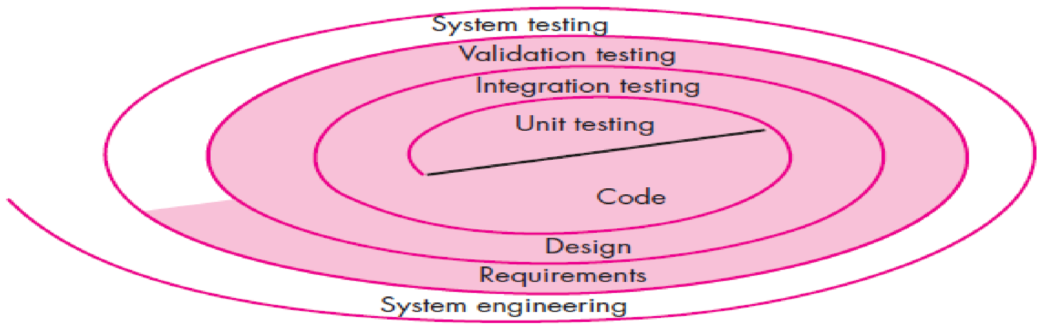


Fig : Testing Strategy

A strategy for software testing may also be viewed in the context of the spiral. ***Unit testing*** begins at the vortex of the spiral and concentrates on each unit of the software as implemented in source code. Testing progresses by moving outward along the spiral to ***integration testing***, where the focus is on design and the construction of the software architecture. Taking another turn outward on the spiral, you encounter ***validation testing***, where requirements established as part of requirements modeling are validated against the software that has been constructed. Finally, you arrive at ***system testing***, where the software and other system elements are tested as a whole.

Considering the process from a procedural point of view, testing within the context of software engineering is actually a series of **four** steps that are implemented sequentially. The steps are shown in following figure. Initially, tests focus on each component individually, ensuring that it functions properly as a unit. Hence, the name ***unit testing***. Unit testing makes heavy use of testing techniques that exercise specific paths in a component's control structure to ensure complete coverage and maximum error detection.

Next, components must be assembled or integrated to form the complete software package. ***Integration testing*** addresses the issues associated with the dual problems of verification and program construction. Test case design techniques that focus on inputs and outputs are more prevalent during integration, although techniques that exercise specific program paths may be used to ensure coverage of major control paths. After the software has been integrated (constructed), a set of *high-order tests* is conducted. Validation criteria must be evaluated. *Validation testing* provides final assurance that software meets all informational, functional, behavioral, and performance requirements.

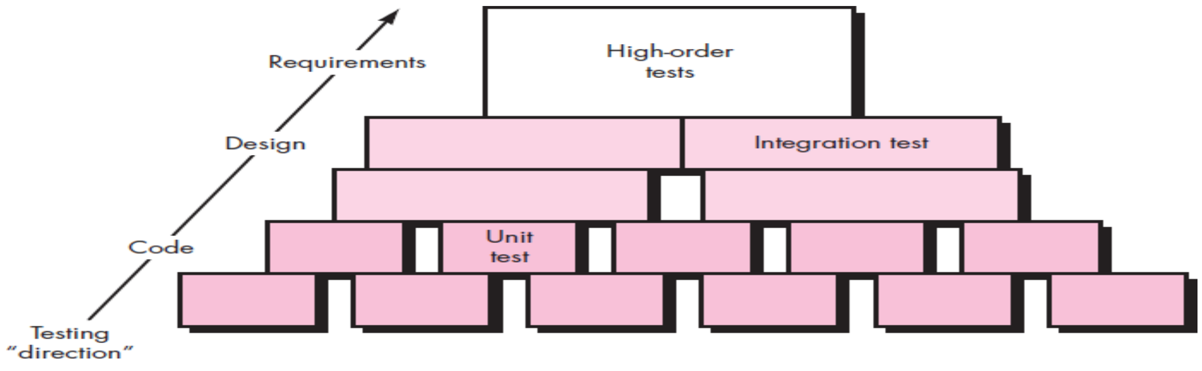


Fig : Software testing steps

The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, databases). ***System testing*** verifies that all elements mesh properly and that overall system function/performance is achieved.

Criteria for Completion of Testing

"When are we done testing—how do we know that we've tested enough?" Sadly, there is no definitive answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

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.

Although few practitioners would argue with these responses, you need more rigorous criteria for determining when sufficient testing has been conducted. The ***clean room software engineering*** approach suggests statistical use techniques that execute a series of tests derived from a statistical sample of all possible program executions by all users from a targeted population.

. By collecting metrics during software testing and making use of existing software reliability models, it is possible to develop meaningful guidelines for answering the question: "When are we done testing?"

<u>STRATEGIC ISSUES</u>

Tom Gilb argues that a software testing strategy will succeed when software testers:

- ***Specify product requirements in a quantifiable manner long before testing commences***. Although the overriding objective of testing is to find errors, a good testing strategy also assesses other quality characteristics such as portability, maintainability, and usability. These should be specified in a way that is measurable so that testing results are unambiguous.

- ***State testing objectives explicitly***. The specific objectives of testing should be stated in measurable terms.
- ***Understand the users of the software and develop a profile for each user category***. Use cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
- ***Develop a testing plan that emphasizes "rapid cycle testing*.**"** Gilb recommends that a software team "learn to test in rapid cycles The feedback generated from these rapid cycle tests can be used to control quality levels and the corresponding test strategies.
- ***Build "robust" software that is designed to test itself***. Software should be designed in a manner that uses anti bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
- ***Use effective technical reviews as a filter prior to testing***. Technical reviews can be as effective as testing in uncovering errors.
- ***Conduct technical reviews to assess the test strategy and test cases themselves***. Technical reviews can uncover inconsistencies, omissions, and outright errors in the testing approach. This saves time and also improves product quality.
- ***Develop a continuous improvement approach for the testing process***. The test strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for software testing.

TEST STRATEGIES FOR CONVENTIONAL SOFTWARE

A testing strategy that is chosen by most 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, and culminating with tests that exercise the constructed system. Each of these classes of tests is described in the sections that follow.

Unit Testing

Unit testing focuses verification effort on the smallest unit of software design. 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 tests are illustrated schematically in following figure. 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.

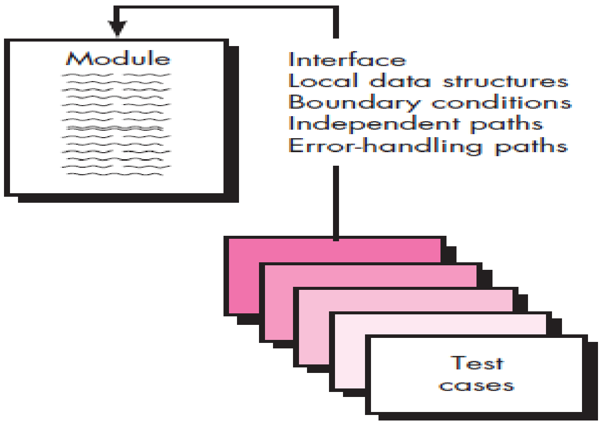


Fig : Unit Test

Selective testing of execution paths is an essential task during the unit test. Test cases should be designed to uncover errors due to erroneous computations, incorrect comparisons, or improper control flow.

Boundary testing is one of the most important unit testing tasks. Software often fails at its boundaries. That is, errors often occur when the *n*th element of an *n*-dimensional array is processed, when the *i*th repetition of a loop with *I* passes is invoked, when the maximum or minimum allowable value is encountered.

A good design anticipates error conditions and establishes error-handling paths to reroute or cleanly terminate processing when an error does occur. Yourdon calls this approach ***anti bugging***.

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 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.

The unit test environment is illustrated in following figure.. 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.

Unit testing is simplified when a component with high cohesion is designed. When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.

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.

There is often a tendency to attempt non incremental integration; that is, to construct the program using a "**big bang**" approach. All components are combined in advance. The entire program is tested as a **whole**. If a set of errors is encountered. Correction is difficult because isolation of causes is complicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied. There are **two** different incremental integration strategies:

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 to the main control module are incorporated into the structure in either a **depth-first or breadth-first** manner. Referring to the following figure, ***depth-first integration*** integrates all components on a major control path of the program structure. 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.

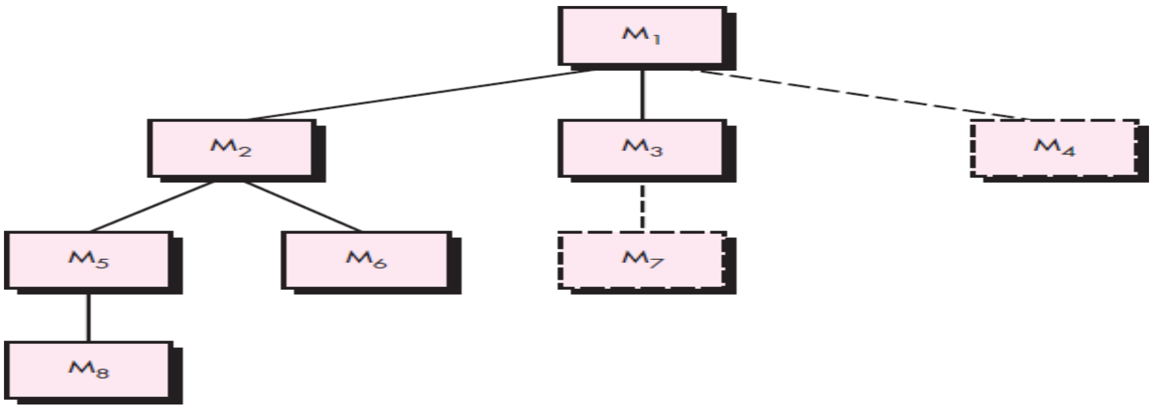


Fig : Top-down integration

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 (discussed later in this section) 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, the functionality provided by 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:

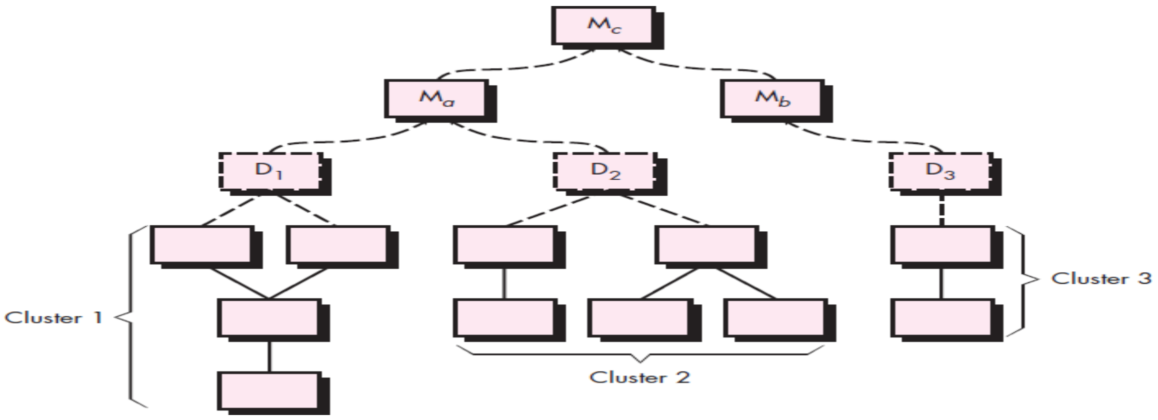
1. Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.

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.

Integration follows the pattern illustrated in following figure. 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 M*a*. Drivers D1 and D2 are removed and the clusters are interfaced directly to M*a*. Similarly, driver D3 for cluster 3 is removed prior to integration with module M*b*. Both M*a* and M*b* will ultimately be integrated with component M*c*,



and so forth.

Fig : Bottom-up integration

As integration moves upward, the need for separate test drivers lessens. In fact, 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. R*egression 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 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.

As integration testing proceeds, the number of regression tests can grow quite large.

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. 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 "showstopper" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds, and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

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

- ***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.
- ***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.

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**.

Class testing for OO software is the equivalent of unit testing for conventional software. 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.

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.

Cluster testing is one step in the integration testing of OO software. Here, a cluster of collaborating classes is exercised by designing test cases that attempt to uncover errors in the collaborations.

TEST STRATEGIES FOR WEBAPPS

The strategy for WebApp testing adopts the basic principles for all software testing and applies a strategy and tactics that are used for object-oriented systems. The following steps summarize the approach:

1. The content model for the WebApp is reviewed to un cover errors.

2. The interface model is reviewed to ensure that all use cases can be accommodated.

3. The design model for the WebApp is reviewed to uncover navigation errors.

4. The user interface is tested to uncover errors in presentation and/or navigation mechanics.

5. Each functional component is unit tested.

6. Navigation throughout the architecture is tested.

7. The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.

8. Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.

9. Performance tests are conducted.

10. The WebApp is tested by a controlled and monitored population of end users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

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.

Validation can be defined in many ways, but a simple 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. After each validation test case has been conducted, one of two possible conditions exists: (1) The function or performance characteristic conforms to specification and is accepted or (2) a deviation from specification is uncovered and a deficiency list is created.

Configuration Review

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

Alpha and Beta Testing

When custom software is built for one customer, a series of **acceptance tests** are conducted to enable the customer to validate all requirements. Conducted by the end user rather than software engineers, an acceptance test can range from an informal “test drive” to a planned and systematically executed series of tests. In fact, acceptance testing can be conducted over a period of weeks or months, thereby uncovering cumulative errors that might degrade the system over time.

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 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. 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. Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.

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), re initialization, 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. During security testing, the tester plays the role(s) of the individual who desires to penetrate the system. Good security testing will ultimately penetrate a system. 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 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. For example, (1) special tests may be designed that generate ten interrupts per second, when one or two is the average rate, (2) input data rates may be increased by an order of magnitude to determine how input functions will respond, (3) test cases that require maximum memory or other resources are executed, (4) test cases that may cause thrashing in a virtual operating system are designed, (5) test cases that may cause excessive hunting for disk-resident data are created.

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. Even at the unit level, the performance of an individual module may be assessed as tests are conducted. Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

Deployment Testing

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.

INTERNAL AND EXTERNAL VIEWS OF TESTING

Any engineered product can be tested in one of **two** ways: (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function. (2) Knowing the internal workings of a product.

The first test approach takes an **external view** and is called **black-box testing.** The second requires an **internal view** and is termed **white-box testing**.

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

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

WHITE-BOX TESTING

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

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

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

A simple notation for the representation of control flow, called a *flow graph* (or *program graph*). The flow graph depicts logical control flow using the notation illustrated in following figure.

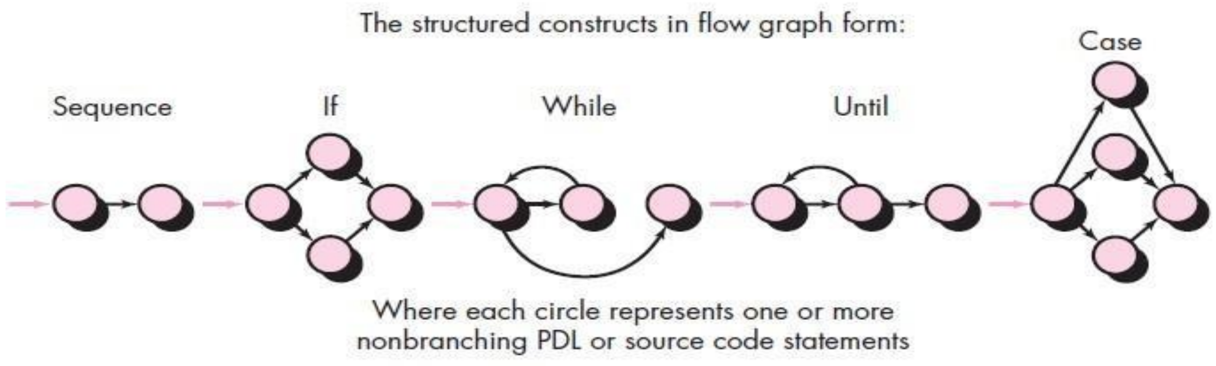
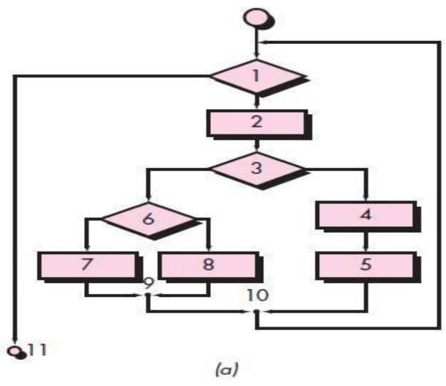

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Fig : Flow Graph Notation

To illustrate the use of a flow graph, consider the procedural design representation in following figure (a). Here, a flowchart is used to depict program control structure. Figure (b) maps the flowchart into a corresponding flow graph.

Referring to figure (b), 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 and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements. Areas bounded by edges and nodes are called ***regions***. When counting regions, we include the area outside the graph as a region Each node that contains a condition is called a ***predicate node*** and is characterized by two or more edges emanating from it



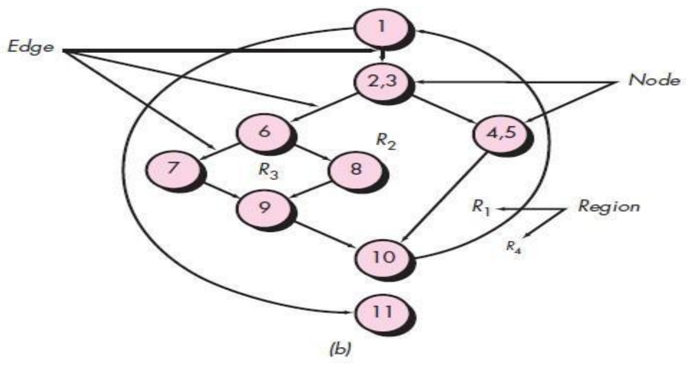


Fig : (a) Flowchart and (b) flow graph

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. 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 (b) 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 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any newedges.

How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer. ***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 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.

Cyclomatic complexity has a foundation in graph theory and provides you with an extremely useful software metric. 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.

Referring once more to the flow graph in figure (b), 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 + =4.
3. $V(G)$ = 3 predicate nodes + 1 =4.

Therefore, the cyclomatic complexity of the flow graph in figure (b) is 4.

Deriving Test Cases

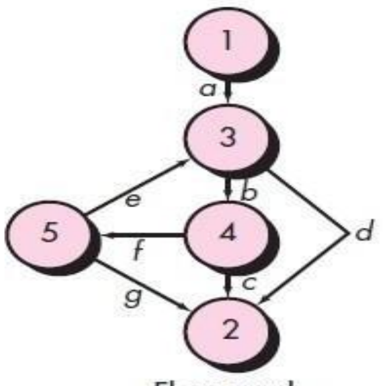
The basis path testing method can be applied to a procedural design or to source code. The following steps can be applied to derive the basis set:

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.

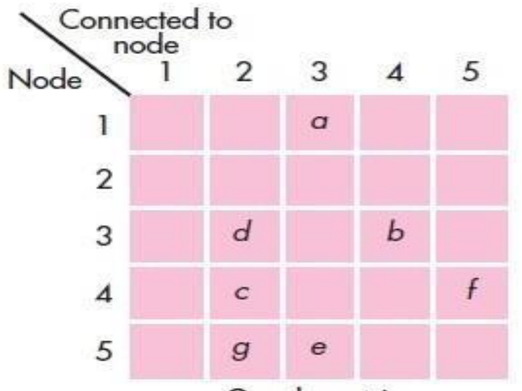
Graph Matrices

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



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

Fig : Graph Matrix

Referring to the figure, each node on the flow graph is identified by numbers, while each edge is identified by letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example, node 3 is connected to node 4 by edge *b*. To this point, the graph matrix is nothing more than a tabular representation of a flow graph. However, by adding a ***link weight*** to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.

The link weight provides additional information about control flow. In its simplest form, the link weight is 1 (a connection exists) or 0 (a connection does not exist). But link weights can be assigned other, more interesting properties:

- The probability that a link (edge) will be execute.
- The processing time expended during traversal of a link
- The memory required during traversal of a link
- The resources required during traversal of a link.

BLACK-BOX TESTING

Black-box testing, also called ***behavioral 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 is not an alternative to white-box techniques. Rather, it is a complementary approach that is likely to uncover a different class of errors than white-box methods. 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.

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?

By applying black-box techniques, you derive a set of test cases that satisfy the following criteria

(1) test cases that reduce, by a count that is greater than one, the number of additional test cases that must be designed to achieve reasonable testing, and (2) test cases that tell you something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

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". 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, you begin by creating a ***graph***, it is a collection of ***nodes*** that represent objects, ***links*** that represent the relationships between objects, ***node weights*** that describe the properties of a node, and ***link weights*** that describe some characteristic of a link.

The symbolic representation of a graph is shown in following figure. 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 graphnodes.

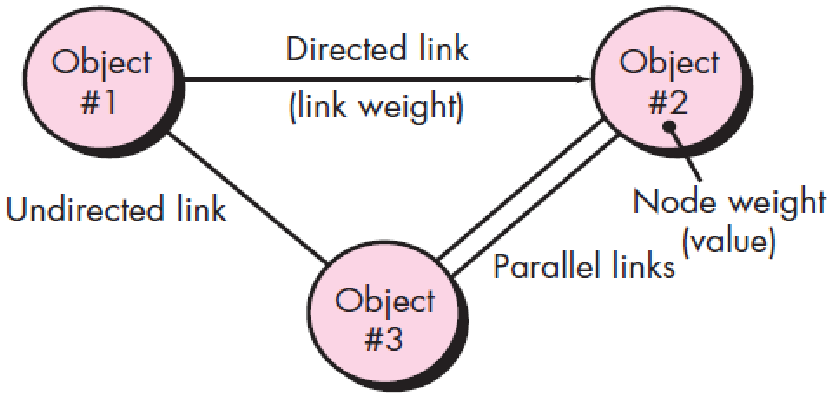


Fig : Graph Notation

Beizer describes a number of behavioral testing methods that can make use of graphs:

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. The state diagram can be used to assist in creating graphs of thistype.

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. Link weights are used to specify the required execution times as the program executes.

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. 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.

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

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. 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.

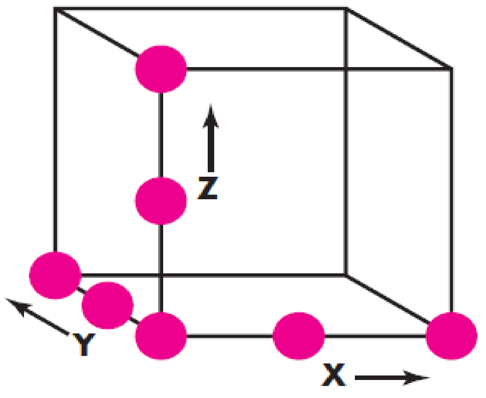
3. Apply guidelines 1 and 2 to output conditions.

4. If internal program data structures have prescribed boundaries, be certain to design a test case to exercise the data structure at its boundary. Most software engineers intuitively perform BVA to some degree.

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.

Orthogonal array testing enables you to design test cases that provide maximum test coverage with a reasonable number of test cases



One input item at a time