Software Engineering

Presented by
B.Pranalini

Unit 4 Part 2: Software Testing Strategies

Topics Covered

- A Strategic Approach to Software Testing
- Test strategies for Conventional Software
- Test strategies for Object-Oriented Software
- Validation testing
- White box testing
- Basic path testing
- Black box testing
- System testing

A Strategic Approach to Testing

- To perform effective testing, a software team should conduct effective formal technical reviews
- Testing begins at the component level and work 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) by an independent test group
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

Verification and Validation:

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- Verification (Are the algorithms coded correctly?)
  - The set of activities that ensure that software correctly implements a specific function or algorithm
- Validation (Does it meet user requirements?)
  - The set of activities that ensure that the software that has been built is traceable to customer requirements.
- Verification: "Are we building the product right?"
- Validation: "Are we building the right product?"

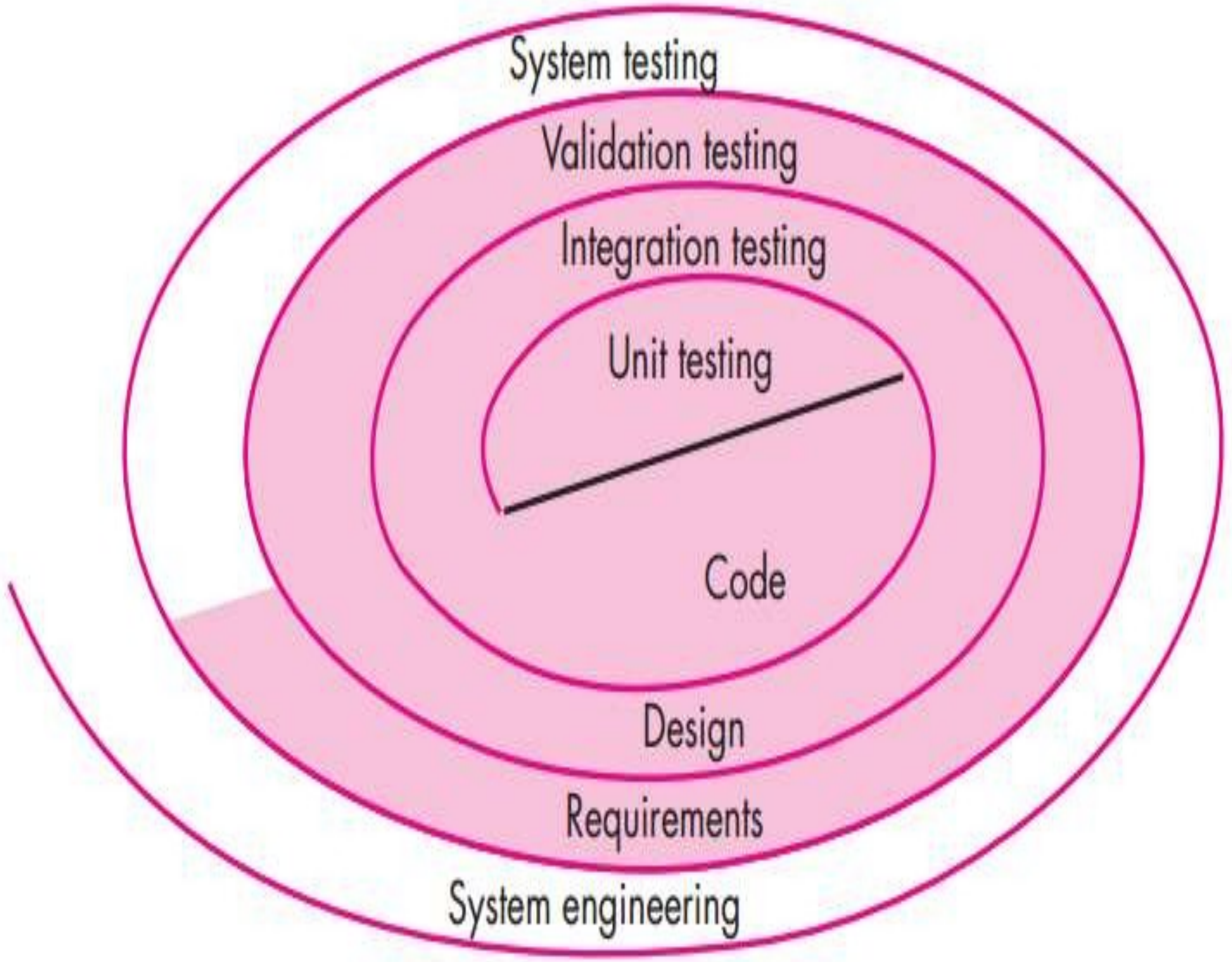
Organizing for Software Testing:

- Testing should aim at "breaking" the software
- Common misconceptions
  - The developer of software should do no testing at all
- that the software should be "tossed over the wall" to strangers who will test it mercilessly,
- that testers get involved with the project only when the testing steps are about to begin.
- 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.

Software testing strategy:

- Unit testing
  - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
  - Focuses on the design and construction of the software architecture
- Validation testing
  - Requirements are validated against the constructed software
- System testing
  - The software and other system elements are tested as a whole

Testing strategy:



Criteria for Completion of testing:

when is testing completed ??

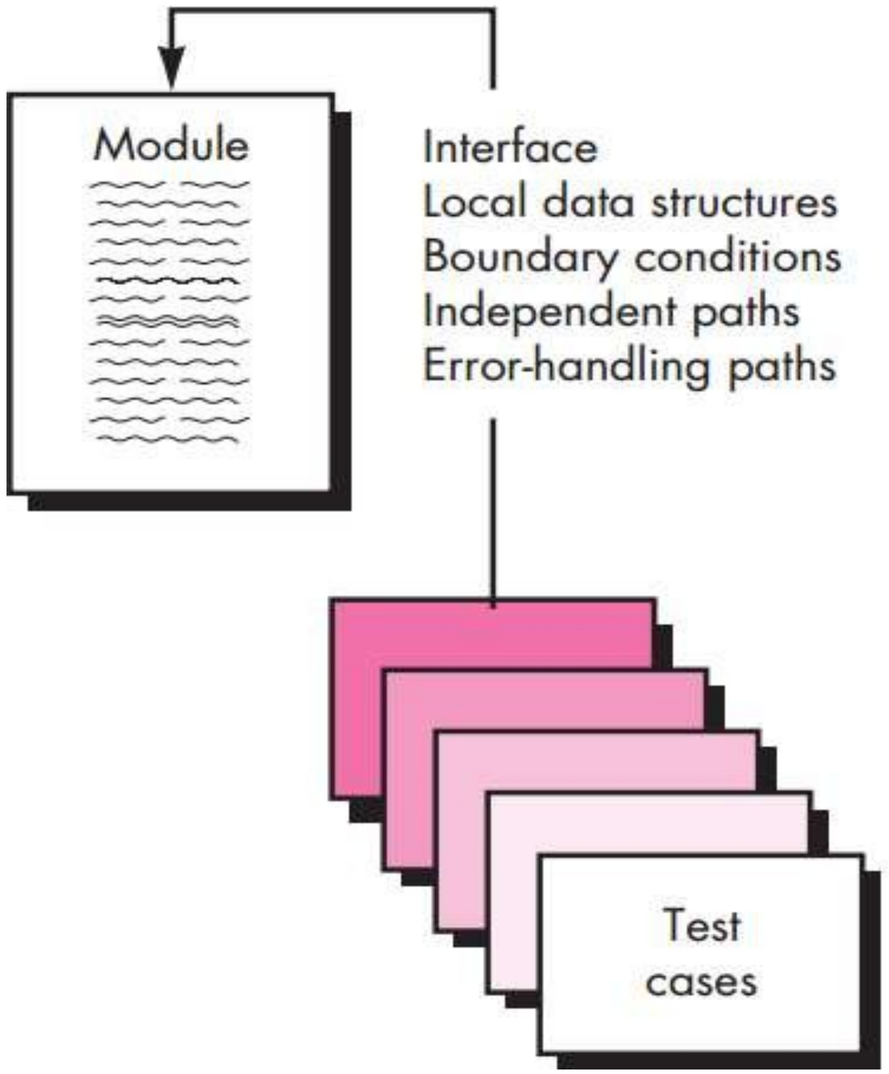
- A classic question arises every time software testing is discussed: “When are we done testing—how do we know that we’ve tested enough?” Sadly, there is no definitive,
- answer to this question, but there are a few pragmatic responses and early attempts at empirical guidance.
- . By collecting metrics 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?”

Test Strategies for Conventional Software

Unit testing:

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
  - Reduces the number of test cases
  - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Unit testing:



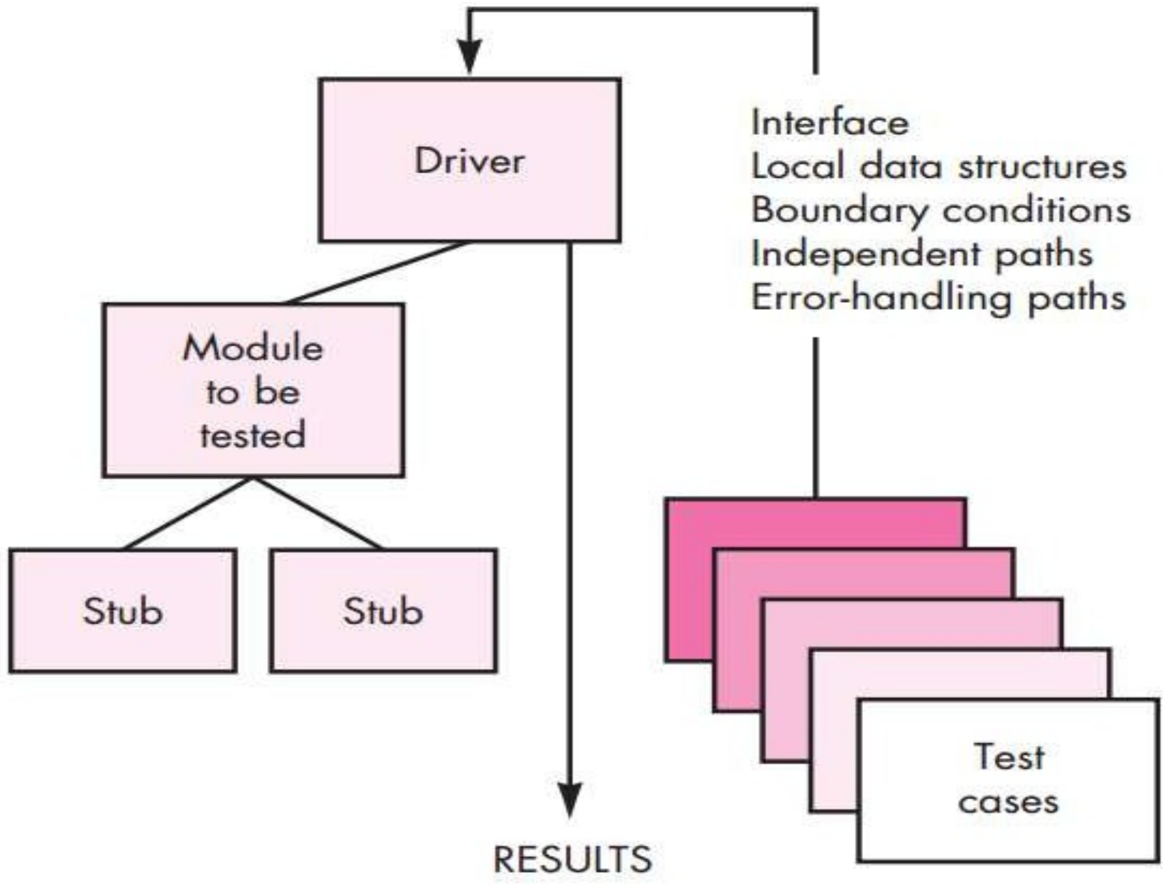
Unit testing considerations

- Module interface
  - Ensure that information flows properly into and out of the module
- Local data structures
  - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- Boundary conditions
  - Ensure that the module operates properly at boundary values established to limit or restrict processing
- Independent paths (basis paths)
  - Paths are exercised to ensure that all statements in a module have been executed at least once
- Error handling paths
  - Ensure that the algorithms respond correctly to specific

Unit test procedures

- Driver
  - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- Stubs
  - Serve to replace modules that are subordinate to (called by) the component to be tested
  - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- Drivers and stubs both represent testing overhead.
  - Both must be written but don't constitute part of the installed software product

Unit-test environment



Integration testing:

- Defined as a systematic technique for constructing the software architecture
  - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
  - Non-incremental Integration Testing
  - Incremental Integration Testing

Non-incremental Integration Testing

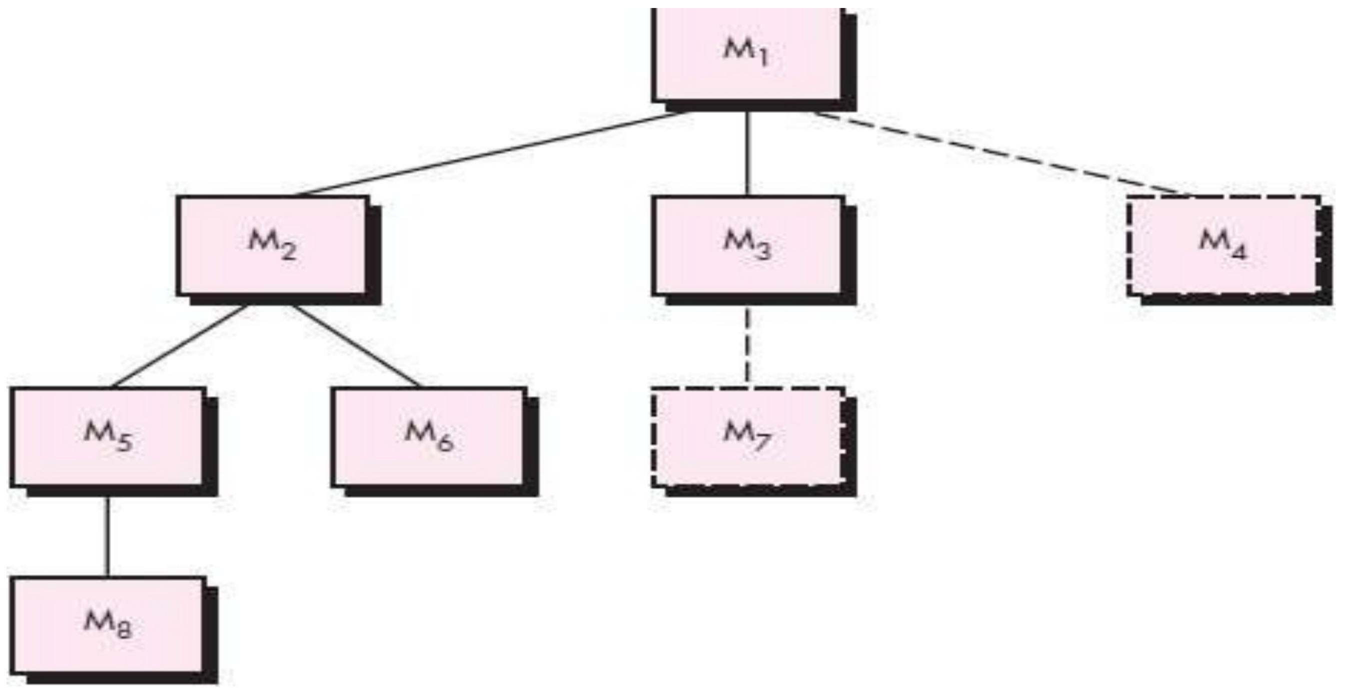
- Uses “Big Bang” approach
- All components are combined in advance
- The entire program is tested as a whole Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

Incremental Integration Testing

- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied
- Different incremental integration strategies
  - Top-down integration
  - Bottom-up integration
  - Regression testing
  - Smoke testing

Top-down Integration

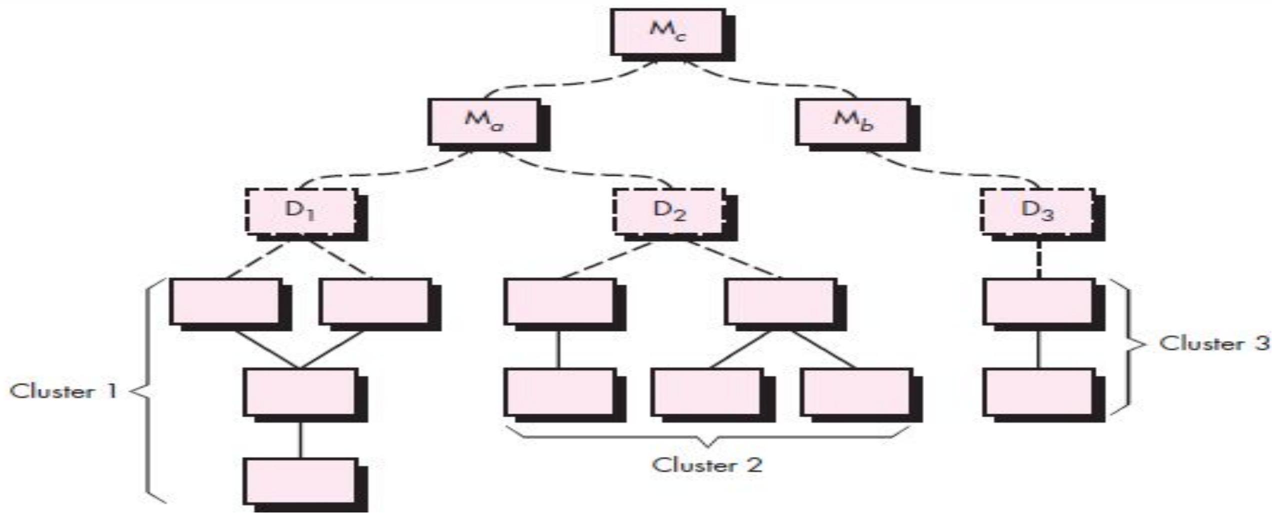
- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in two ways :
  - depth-first : All modules on a major control path are integrated
  - breadth-first : All modules directly subordinate at each level are integrated
- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  integration/testing process
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the



For example,
Selecting the left-hand 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 right-hand control paths are built.

Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
  - This approach verifies low-level data processing early in the testing process
  - Need for stubs is eliminated
- Disadvantages
  - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
  - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available



Integration follows the pattern illustrated in 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 D1and 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.

Regression Testing

- Each new addition or modification of data may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
  - Ensures that changes have not propagated unintended side effects
  - Helps to ensure that changes do not introduce unintended behavior or additional errors
  - May be done manually or through the use of automated capture/playback tools
- Regression test suite 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 actual software components that

Smoke testing

- Designed as a pacing mechanism for time-critical projects
  - Allows the software team to assess its project on a frequent basis
- Includes the following activities
  - The software components that have been translated into code and linked into a build
  - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
    - The goal is to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule
  - The build is integrated with other builds and the entire product is smoke tested daily

Benefits of Smoke Testing

- Integration risk is minimized
  - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
  - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
  - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
  - As integration testing progresses, more software has been integrated and more has been demonstrated to work
  - Managers get a good indication that progress is being made

Comparison b/w conventional & OO s/w Unit Testing

- In conventional software we are testing the individual units/modules…
- In object oriented software testing the class individual classes & subclasses..
- which tends to focus on the algorithmic detail of a module and the data that flow across the module interface.
- OO software is driven by the operations encapsulated by the class and the state behavior of the class

Test Strategies for Object-Oriented Software

Unit testing:

- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class

Integration testing in OO context:

- Two different object-oriented integration testing strategies are
  - 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
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

Validation Testing

- Validation testing follows integration testing
- The distinction between conventional and object-oriented software disappears and Focuses on user-visible actions and user-recognizable output from the system

Validation test criteria :

- Demonstrates conformity with requirements
- Designed to ensure that All functional requirements are satisfied,All behavioral characteristics are achieved,All performance requirements are attained
- Documentation is correct
- Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)
- After each validation test
  – The function or performance characteristic conforms to specification and is accepted
  – A deviation from specification is uncovered and

Configuration review:

- The intent of this 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

Alpha and beta testing :

- Alpha testing conducted at the developer's site by end users
  - Software is used in a natural setting with developers watching intently
  - Testing is conducted in a controlled environment
- Beta testing conducted at end-user sites
  - Developer is generally not present
  - It serves as a live application of the software in an environment that cannot be controlled by the developer
  - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

Alpha and Beta Testing:

- Most software product builders use a process called alpha and beta testing to uncover errors that only the
end user seems able to find.
  - 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.
  - A variation on beta testing, called *customer acceptance testing,* is sometimes performed when custom software is delivered to a customer under contract.

White box testing

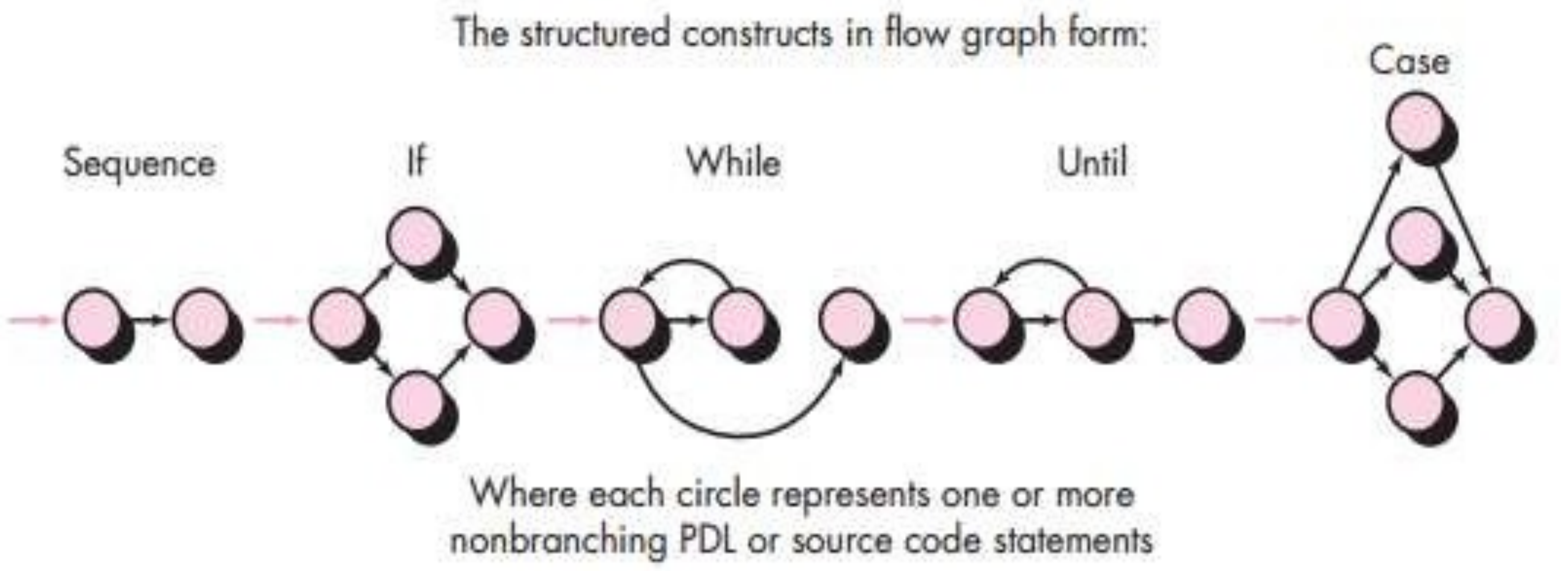
- White box testing is also called as glass-box testing
- Using white-box testing methods can derive test cases that
  - guarantee that all independent paths within a module have been exercised at least once
  - exercise all logical decisions on their true and false sides
  - execute all loops at their boundaries and within their operational bounds
  - exercise internal data structures to ensure their validity

Basis path testing

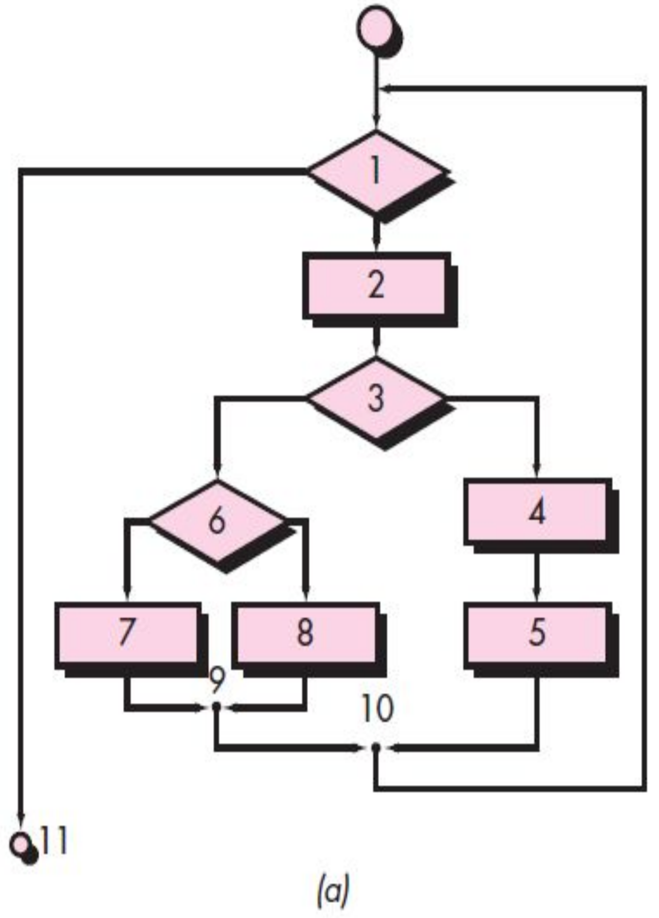
- Basis path testing is a white-box testing technique
- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths

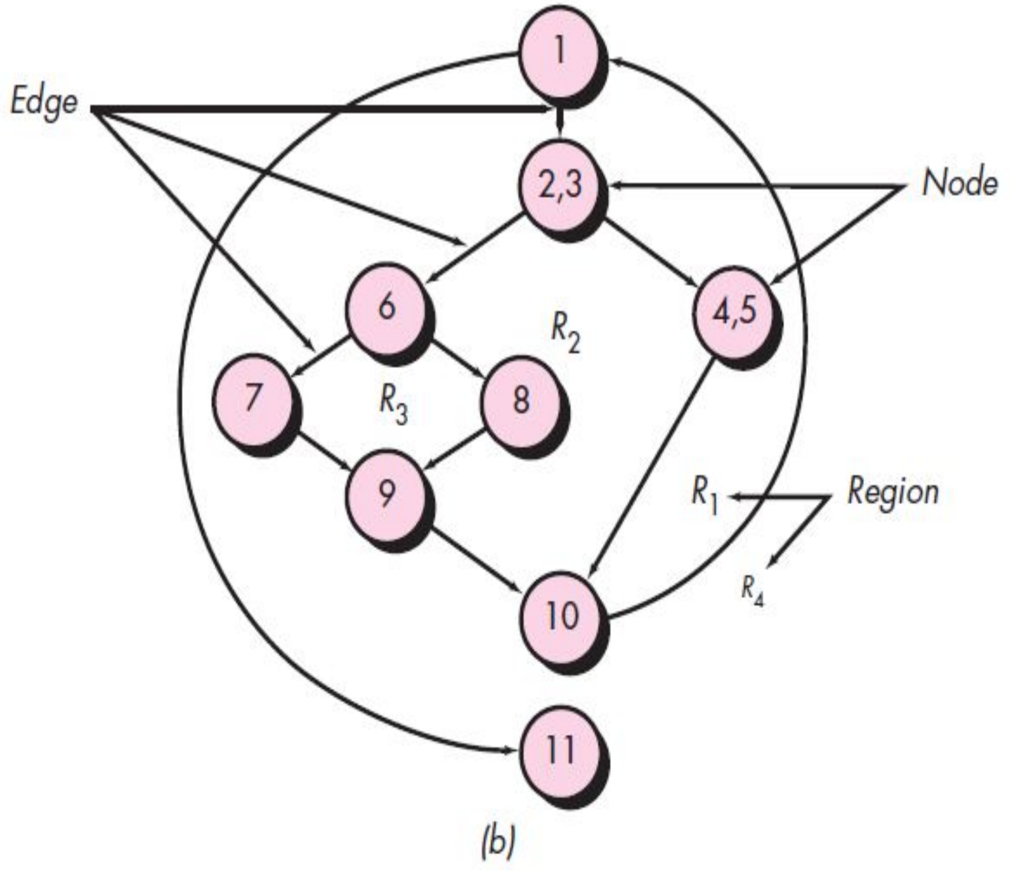
Flow Graph Notation:

- A simple notation for the representation of control flow, called a flow graph
- It also know as program graph


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

- Arrows called edges or links represent flow of control
- Circles called floe graph nodes represent one or more actions
- Areas bounded by edges and nodes called regions
- A predicate node is a node containing a condition





(a) Flowchart and (b) flow graph

Flowchart is used to depict program control structure.

Maps the flowchart into a corresponding flow graph

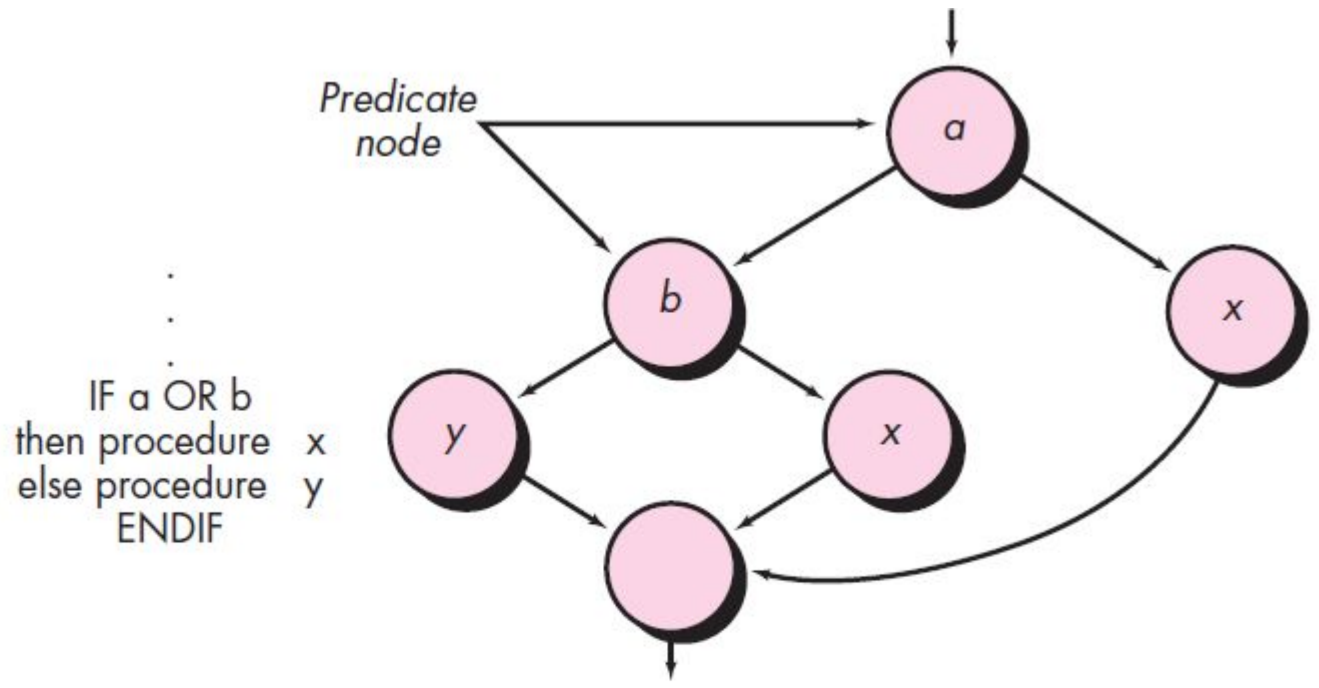


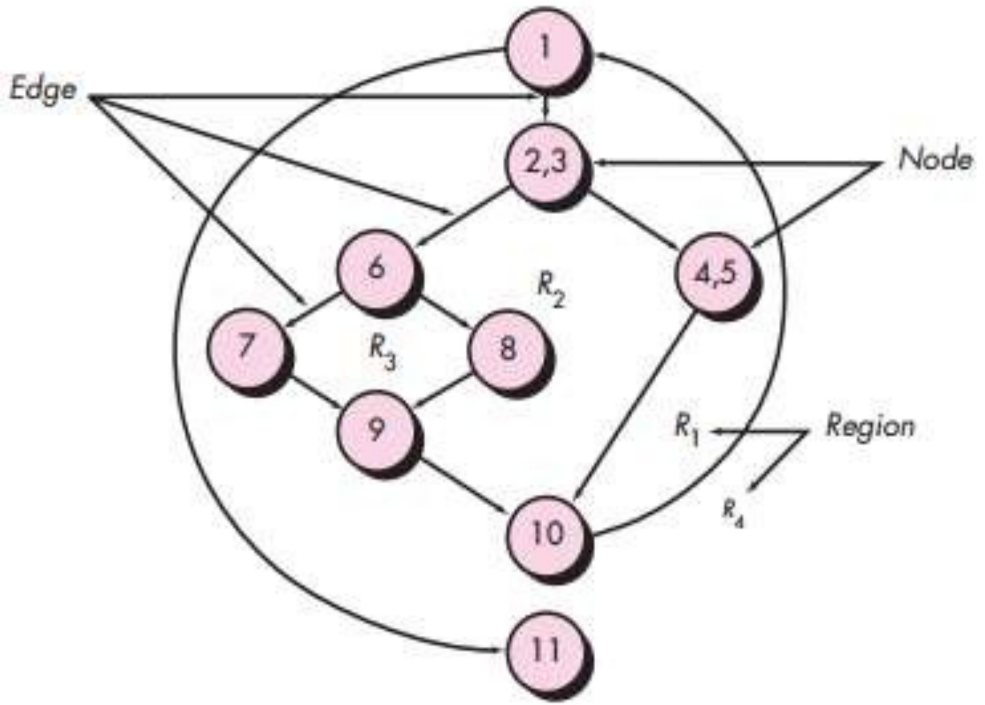
Fig: Compound logic

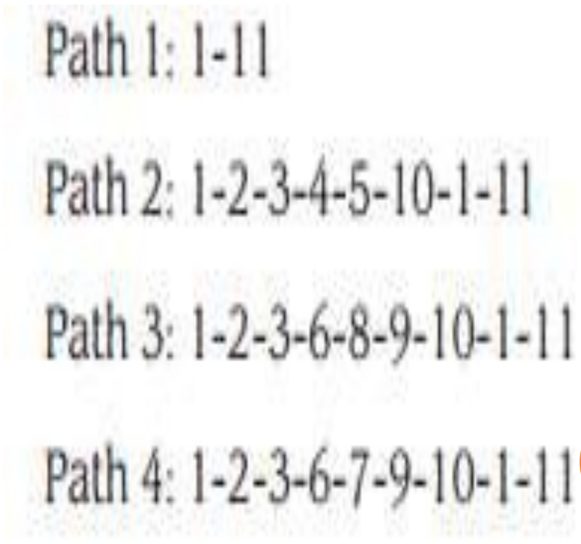
The program design language (PDL) segment translates into the flow graph shown. Note that a separate node is created for each of the conditions *a and b in the statement IF a OR b. Each node that* contains a condition is called a *predicate node and is characterized by two or more* edges emanating from it.

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
- independent path must move along at least one edge that has not been traversed before the path is defined

- An independent path is any path through the program that introduces at least one new set of processing statements or a new condition
- independent path must move along at least one edge that has not been traversed before the path is defined
- Example:




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

Deriving test cases:

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

```
    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;
1 { total.input = total.valid = 0;      2
  { sum = 0;
  { DO WHILE value[i] <> -999 AND total.input < 100   3
4    increment total.input by 1;
5 →  IF value[i] > = minimum AND value[i] < = maximum   6
       { THEN increment total.valid by 1;
7      {        sum = s sum + value[i]
       { ELSE skip
8  { ENDIF
   { increment i by 1;
9 ENDDO
   IF total.valid > 0   10
11  THEN average = sum / total.valid;
12 → ELSE average = -999;
13 ENDIF
END average
```

$$\begin{tikzcd}
 & & & & 1 \arrow[d] & \\
 & & & & 2 \arrow[d] \arrow[ddlll] & \\
 & & & & 3 \arrow[d] \arrow[dlll] & \\
 & 10 \arrow[dl] \arrow[dr] & & & 4 \arrow[d] & \\
12 \arrow[dr] & & 11 \arrow[dl] & & 5 \arrow[d] \arrow[ddl] & \\
 & 13 & & & 6 \arrow[dr] \arrow[dl] & \\
 & & & 8 \arrow[dr] & & 7 \arrow[ll] \\
 & & & & 9 \arrow[uuuuuu, bend right] &
\end{tikzcd}$$

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-. . .

Path 1 test case:

value(k) = valid input, where $k < i$ for $2 \le i \le 100$

value(i) = −999 where $2 \le i \le 100$

Expected results: Correct average based on k values and proper totals.

Note: Path 1 cannot be tested stand-alone but must be tested as part of path 4, 5, and 6 tests.

Path 2 test case:

value(1) = −999

Expected results: Average = −999; other totals at initial values.

Path 3 test case:

Attempt to process 101 or more values.

First 100 values should be valid.

Expected results: Same as test case 1.

Path 4 test case:

value(i) = valid input where $i < 100$

value(k) < minimum where $k < i$

Expected results: Correct average based on k values and proper totals.

Path 5 test case:

value(i) = valid input where $i < 100$

value(k) > maximum where $k <= i$

Expected results: Correct average based on n values and proper totals.

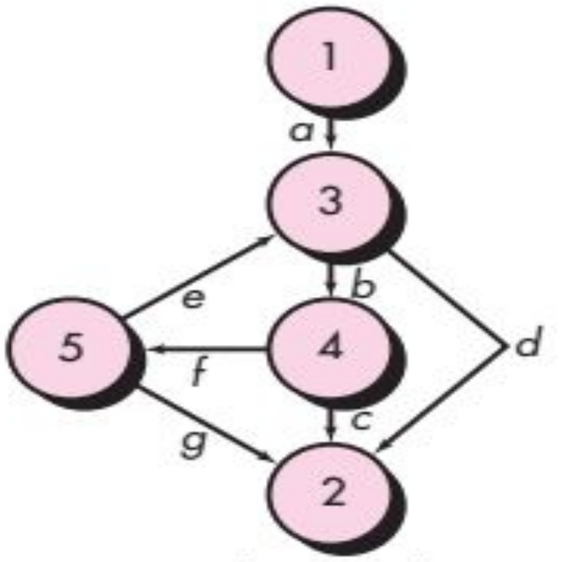
Path 6 test case:

value(i) = valid input where $i < 100$

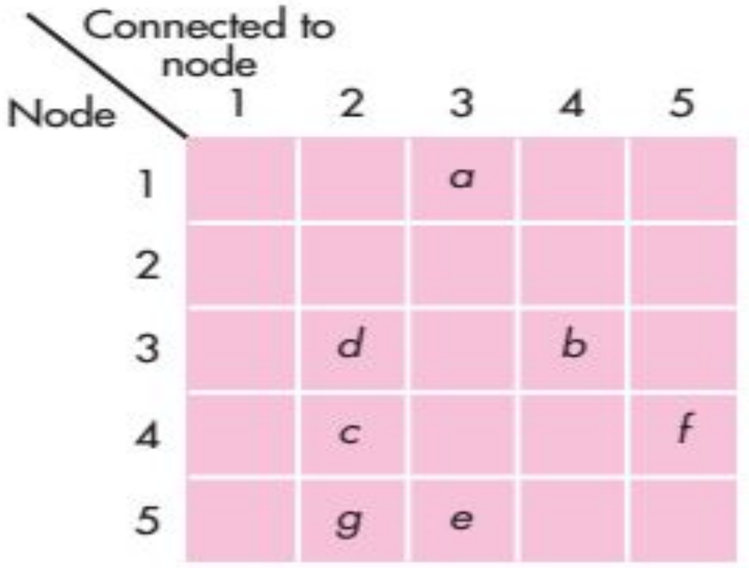
Expected results: Correct average based on n values and proper totals.

Graph matrices:

- 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



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

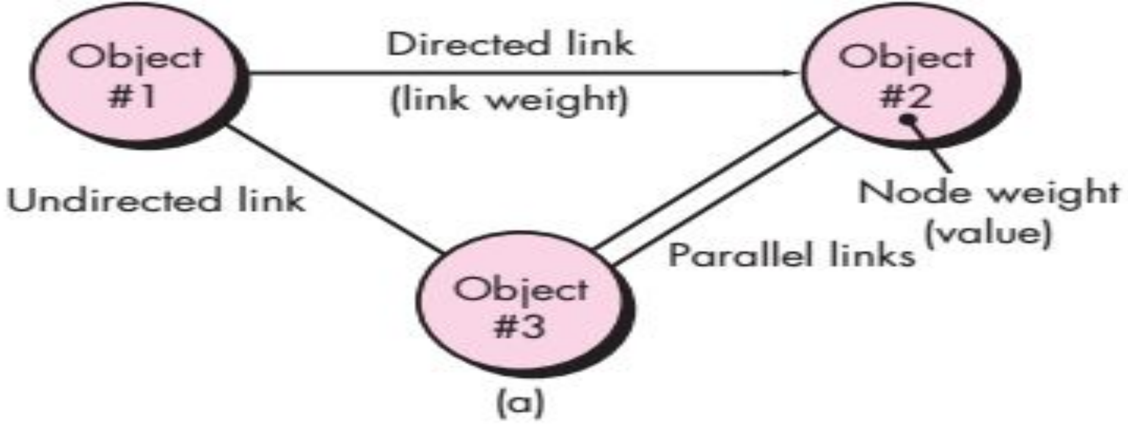
Black box testing

- Black-box testing, also called behavioral testing

- Black-box testing attempts to find errors in the following categories
  - incorrect or missing functions
  - interface errors
  - errors in data structures or external database access
  - behavior or performance errors
  - initialization and termination errors.

Graph-based testing methods

- The first step is to understand the objects that are modeled in software and the relationships that connect these objects.
- Next step is to define a series of tests that verify "all objects have the expected relationship to one another"
  (1) creating a *graph* of important objects and their relationships
  (2) devising a series of *tests* that will cover the graph so that each object and relationship is exercised and errors are uncovered.
- A *graph* is 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);
  ● *link weights* that describe some characteristic of a link.



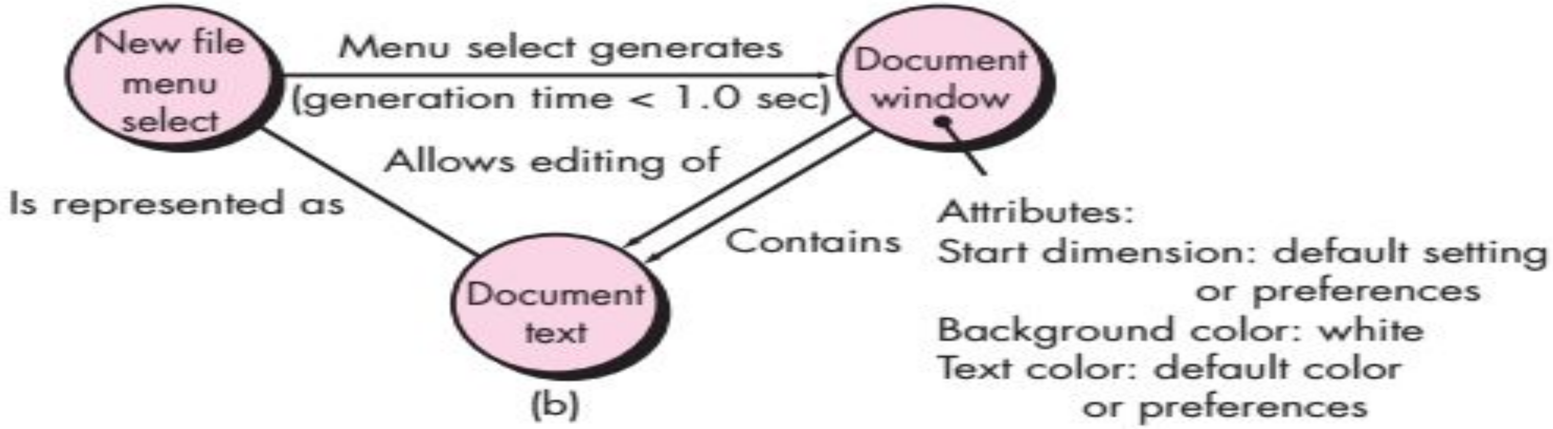


Fig: (a) Graph notation; (b) simple example

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.

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
- 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
- Guidelines for BVA are similar in many respects to those provided for equivalence partitioning:
  - 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.
  - 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.
  - Apply guidelines 1 and 2 to output conditions
  - If internal program data structures have prescribed boundaries (e.g., a

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
- a single parameter value makes the software malfunction. These faults are called single mode faults
- If there exists a consistent problem when specific levels of two parameters occur together, it is called a double mode fault

System testing

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

- Recovery testing
  - Tests for recovery from system faults
  - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
  - Tests reinitialization, check pointing mechanisms, data recovery, and restart for correctness
- Security testing
  - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- Stress testing
  - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

- Performance testing
  - Tests the run-time performance of software within the context of an integrated system
  - Often coupled with stress testing and usually requires both hardware and software instrumentation
  - Can uncover situations that lead to degradation and possible system failure
- Deployment testing
  - Also known as configuration testing
  - It examines all installations procedures that will be used by customers