UNIT –IV

SOFTWARE TESTING STRATEGIES

SOFTWARE TESTING

- Testing is a method which ensures correctness of an activity.
- Software Testing is a process where a software/product is tested whether it is satisfying all the requirements or not.
- It is a crucial activity which involves activities like creating test cases, evaluating the case, defect logging, communication with developers, retesting.
- Testing is essential to deliver a quality product.

SOFTWARE TESTING STRATEGIES

- Testing strategy is a road map describing the steps to be performed in testing. It’s a plan which gives an overview of types of testing, test case planning, test execution, resource planning.
- A testing plan is developed by project manager, test engineers.
- Testing requires equal effort as much as coding requires to get a quality product output.
- The outcome/work product after applying testing strategies is **Test Specification document**

A Strategic Approach to Software Testing

GENERIC CHARACTERISTICS OF TESTING STRATEGIES

- Conducting effective technical reviews.
- Start with each component and expand outwards towards integration.
- In each step of software life cycle different techniques should be followed.
- Many cases testing is conducted by developers at initial level. For large projects a separate testing team is available .

Verification and Validation

- **Verification**: is a process which checks whether the software/module correctly implements a specific function.

- **Validation :** is a process which checks whether the software that has been built is satisfying to customer requirements

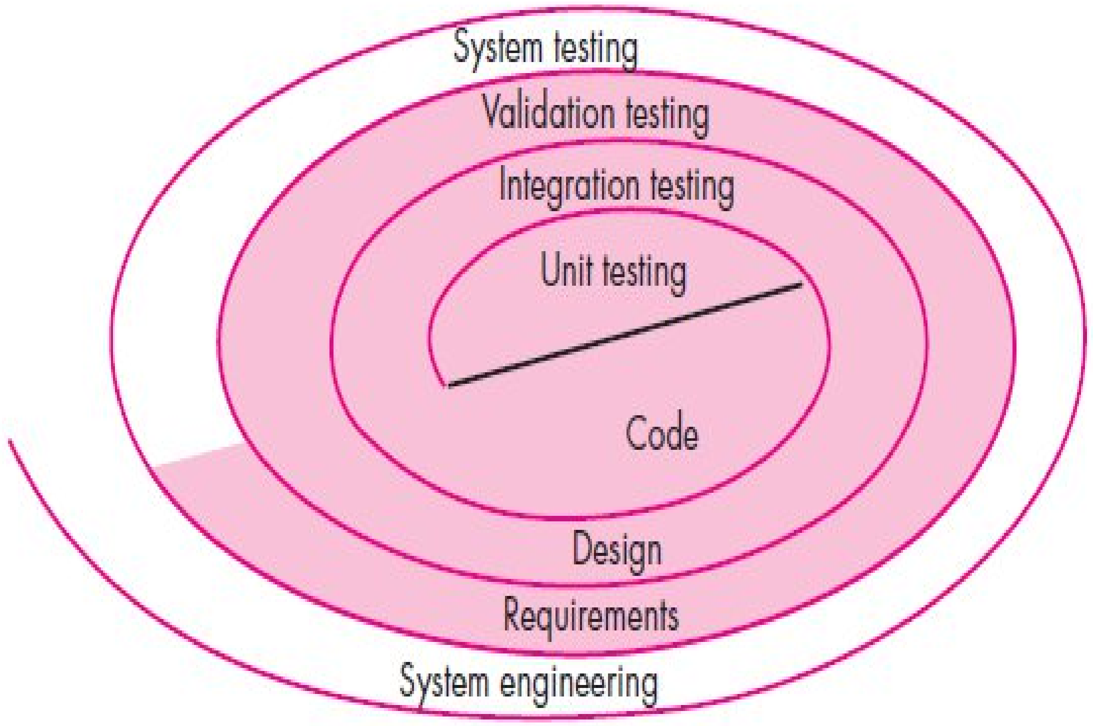
- Verification: “Are we building the product right?”
- Validation: “Are we building the right product?”

- Both the processes include quality assurance activities like
  - Technical reviews
  - quality and configuration audits
  - performance monitoring
  - Simulation
  - Feasibility study
  - documentation review, database review, algorithm analysis
  - development testing, usability testing, qualification testing, acceptance testing, and installation testing.

Software Testing Strategy—The Big Picture

FIGURE 17.1

Testing strategy



Software Testing Strategy—The Big Picture

- ***Unit testing -*** begins at the vortex of the spiral and concentrates on each unit (e.g., component, class, or WebApp content object) of the software as implemented in source code.

- ***Integration testing –*** focuses on design and the construction of the software architecture.

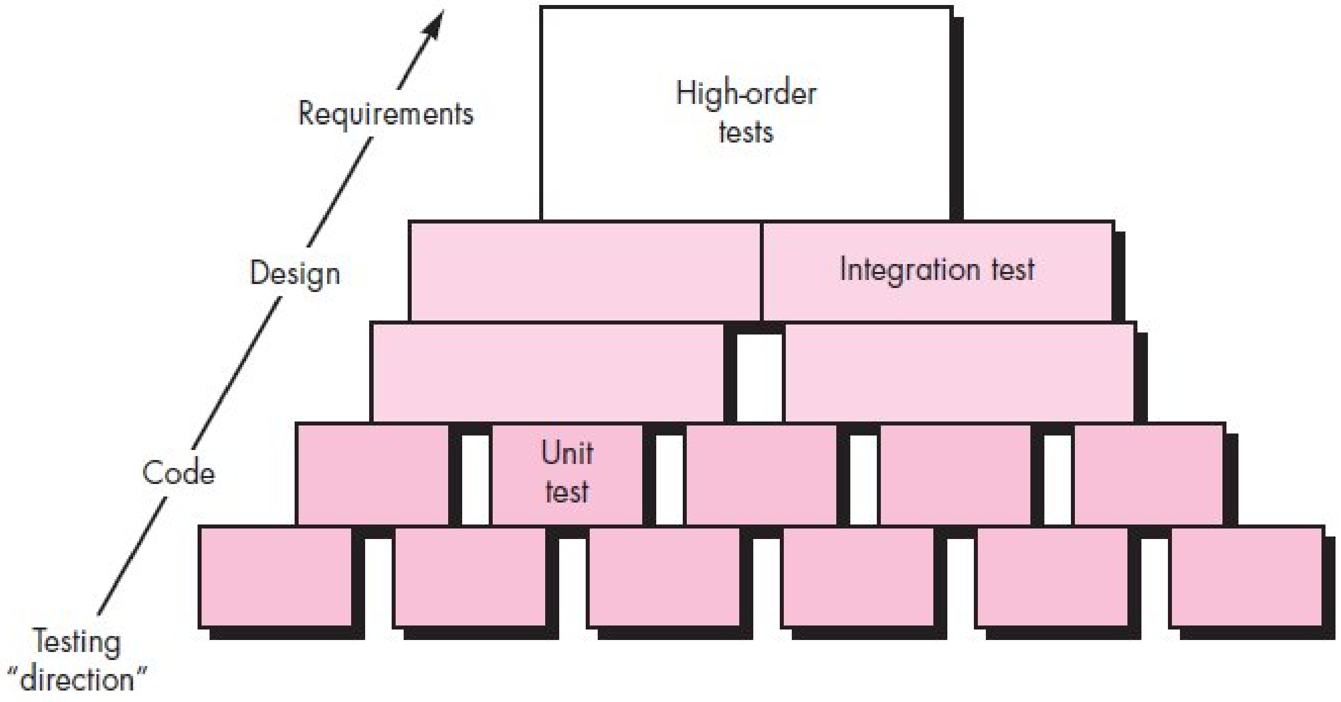
- ***Validation testing -*** *R*equirements established as part of requirements modeling are validated against the software that has been constructed

- ***System testing - S***oftware and other configuration system elements are tested as a whole

Software Testing Strategy—The Big Picture

FIGURE 17.2

Software testing steps



Software Testing Strategy—The Big Picture

- **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.

- ***Integration testing*** addresses the issues associated with the dual problems of verification and program construction.

- ***high-order tests*** *ensures that* Validation criteria is evaluated. *Validation testing* provides final assurance that software meets all
  - informational
  - functional
  - behavioral and
  - performance requirements.

Test strategies for conventional software

- **Conventional software** refers to **traditional, stand-alone software systems** that were developed and deployed before the rise of internet-based and AI-driven applications.

- It is a traditional approach in software engineering that emphasizes structured processes, extensive documentation, and a sequential development model, **often the Waterfall model.**

- There are two broad ways in which we can proceed our testing
  - **Wait until entire software is built ,then perform testing**
  - **Conduct time to time testing as and when each individual module is developed.**

Test strategies for conventional software

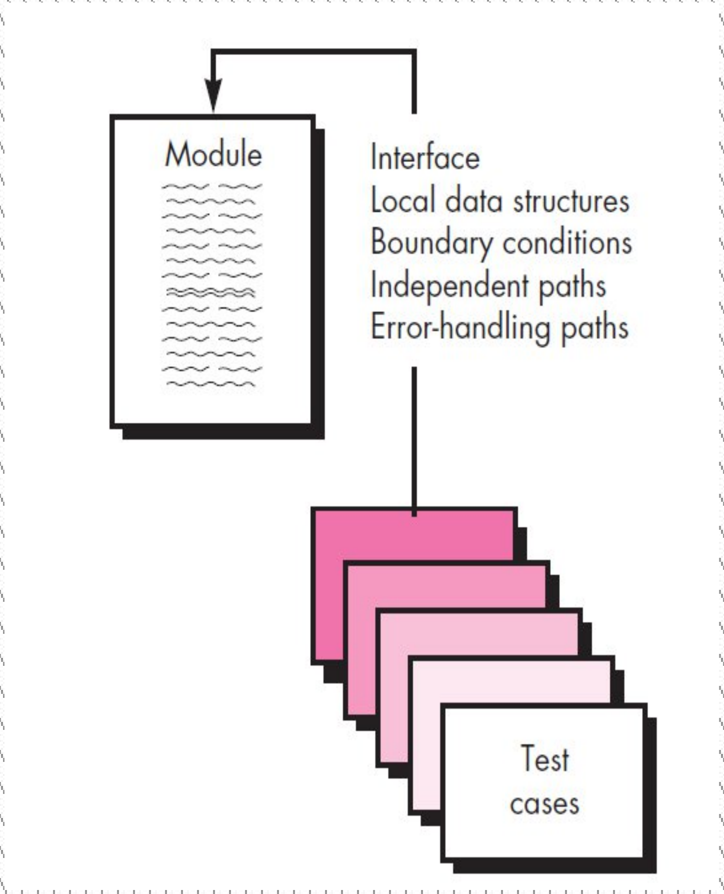
- UNIT TESTING
  - UNIT TEST CONSIDERATIONS
  - UNIT TEST PROCEDURES

- INTEGRATION TESTING
  - APROACHES
  - REGRESSION TESTING
  - SMOKE TESTING

Test strategies for conventional software

UNIT TESTING

- Unit testing focuses on verification of smallest unit in a software.
- It tests every component for end-to-end functionality.
- It focuses on the internal processing logic and data structures within the boundaries of a component.



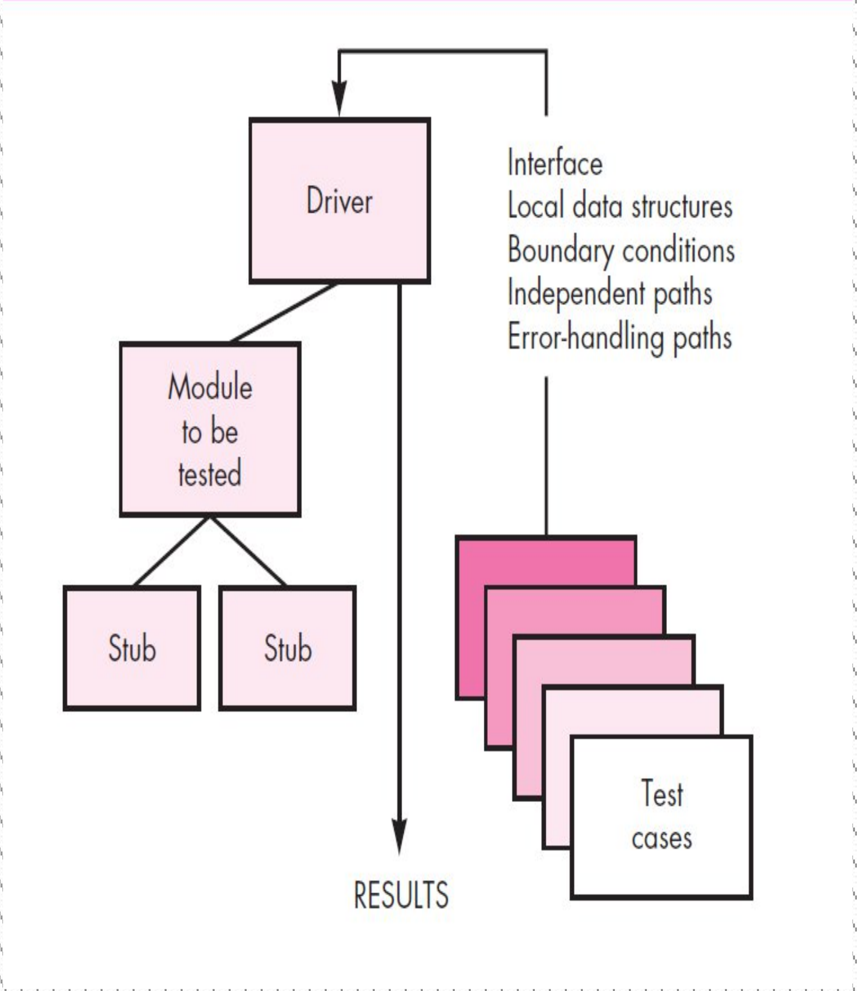
Test strategies for conventional software

- **UNIT TEST CONSIDERATIONS**
- The module interface is tested to ensure that information properly flows into and out of the program unit under test.

- Boundary conditions are tested.
- Data flow is tested.
- Local data and global data usage must be tested.
- Execution path is tested across all features inside a component.
- **Antibugging** : Inbuilt systems to clearly terminate the module when any error occurred.

Test strategies for conventional software

- Review the design model and prepare test cases.
- Driver is a main program which accepts test case data, gives it to module and print results.
- Stub acts as place holder for dependent module. It is a temporary code which simulates the behaviour of immediate module.



Test strategies for conventional software

INTEGRATION TESTING

- Combining the modules to share data and work together is **interfacing**.
- Integration testing is a systematic technique for conducting tests to uncover errors associated with interfacing.
- **Non incremental Integration –**Takes more time and effort and creates more bugs.
- **Incremental Integration –** Limits the time of error resolving and prevents same error from other new modules.

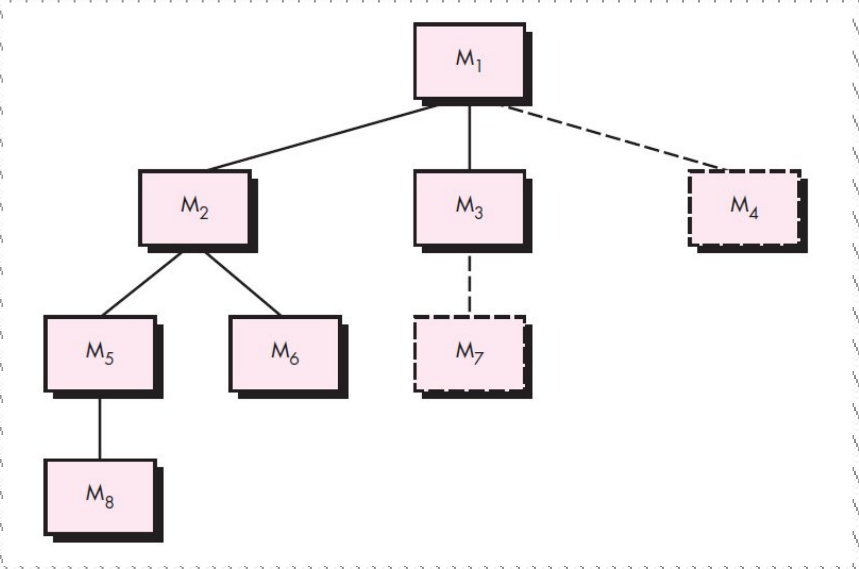
Test strategies for conventional software

Top down integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module.

- Subsequent modules are visited either by following breath first search or depth first search.

- Follows a series of 5 steps.



Test strategies for conventional software

Top down integration (STEPS)

- The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
- Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
- Tests are conducted as each component is integrated.
- On completion of each set of tests, another stub is replaced with the real component.
- Regression testing may be conducted to ensure that new errors have not been introduced.

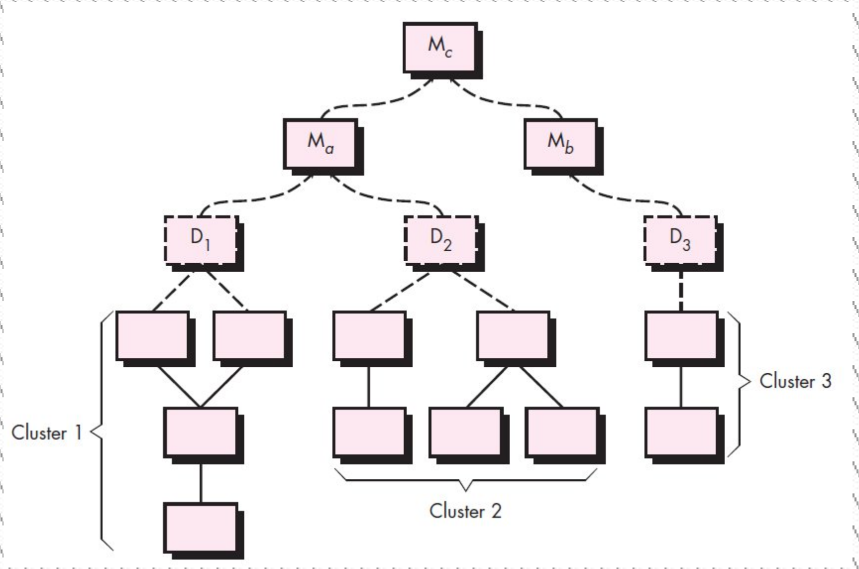
Test strategies for conventional software

BOTTOM UP INTEGRATION

- Testing begins with atomic bottom level modules.
- No need of stub

- **STEPS**

  1. Low level components are combined to form cluster(build).
  2. Driver code will coordinate input and output flow.
  3. Cluster is tested
  4. Driver is removed and cluster moves upward in flow.



Test strategies for conventional software

REGRESSION TESTING

- It is a type of testing which is conducted to unfold the new errors when
  - A new module is added to the system
  - After changing the existing model (because of the errors in original integration)
  - Data flow is altered in the process of previous error resolving.
- Regression testing is done only after the original integration testing is completed.
- It is also done every time a new module or a new change is happening in the system.
- It is done either manually or using **Capture/playback tools.**

- **Regression test suite contains test cases for 3 main issues**
  - Sample test cases on all features.
  - Test cases on the new/changed component
  - Additional cases for features that may be influenced by our change.

Test strategies for conventional software

SMOKE TEST

- Smoke testing is a preliminary software testing process to quickly verify that a new build's most critical functionalities are working before proceeding with more detailed testing.

- It's a rapid, shallow check, sometimes called **build verification testing**, that ensures the application is stable enough for the next stages of quality assurance.

- If the smoke test fails, the build is immediately sent back to developers for fixes, saving time and resources on deeper tests that would be pointless on a broken system.

Test strategies for conventional software

SMOKE TEST

The smoke-testing approach comrpises the following activities:

- Software components that have been translated into code are integrated into a *build.* A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "**showstopper**" errors.
- The build is integrated with other builds, and the entire product is smoke tested daily.

Test strategies for conventional software

Benefits of smoke testing.

1. Integration risk is minimized.
2. The quality of the end product is improved.
3. Error diagnosis and correction are simplified.
4. Progress is easier to assess.

Test Strategies For Object-oriented Software

- Object-oriented software is a programming approach that organizes software around data, or **objects**, instead of logic and functions.

- Entire software is dependent on OOPS principles. (Encapsulation, Abstractions, Polymorphism, Inheritance, Message passing).

- With respect to OO software ,the idea of both Unit and Integration testing differs a lot from conventional software strategies.

Test Strategies For Object-oriented Software

UNIT TESTING

- Encapsulation binds methods and objects together into a unit.(class).

- Operations within the class (methods) are the smallest testable units.

- But methods doesn’t exist in stand alone fashion in OOP. Therefore testing a method completely individually is practically impossible in **unit test context** here**.**

- class testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

Test Strategies For Object-oriented Software

INTEGRATION TESTING

- OO programming doesn’t have a hierarchical structure, therefore top down or bottom up is meaningless.

- Here in OOP we follow
  - **Thread based testing**
  - **Use based testing**

- **Thread based testing -** Classes that are involved when a particular event is occurred are integrated into one build. Each action is treated as **thread** and tested separately. Connecting threads are tested as per data flow.

- **Use based testing –** Testing the classes that are loosely coupled, or less dependent on other class data. Once it is done dependent classes are tested.

Validation Testing

- It is a type of testing where is system is evaluated against user requirements.
- Validation testing begins at the end if Integration testing.
- Validation testing happens after individual units are tested, software is assembled as a package, interfacing errors have been corrected.
- **Software Requirements Specification** describes all user-visible attributes of the software and contains a ***Validation Criteria section*** that forms the basis for a validation-testing approach.
- **Validation test criteria :** Includes a series of tests with test plan and test cases.

Validation Testing

- Test plan contains the strategy and procedure to be followed.
- Test procedure contain test cases which satify,
  - Functional requirements
  - Behavioural characteristics
  - Performance requirements
  - Documentation
  - Usability and configuration
- After the Validation test is done, if test is passed we can proceed to system testing.
- If validation test fails product cannot be delivered at agreed date.

System Testing

- System Testing is a type of software testing that is performed on a completely integrated system to evaluate the compliance of the system with the corresponding requirements. In system testing, software/module which cleared/passed integration testing is taken as input.

- System testing, also known as **end-to-end testing**, that validates the complete and integrated software system against requirements.

- It’s a way in which Validation testing is conducted.

- System testing covers the following sub testing types.
    - Recovery testing
    - Security testing
    - Stress testing
    - Performance testing
    - Deployment testing

System Testing

Recovery testing –

- System should be fault tolerant

- Recovery testing forces the system to fail in many boundary situation and validates whether it handles the failure seamlessly.

- If recovery is automatic - reinitialization, checkpointing 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.

System Testing

Security Testing

- Evaluates a system against intruders attacks.
- Tester will take up the role of hacker and tries to penetrate the system through all possible ways.
- Hackers may enter a system for sensitive information, relieved employees may enter for revenge, common public may steal information for personal gain.
- System testing should ensure the sensitive data in system is end-to-end encrypted.
- It should prevent entry to system. Even if hackers enters, system testing should ensure that no information must be leaked outside the system.

System Testing

Stress Testing

- Evaluates a system against its functional requirements but with unsusual activities.

- Tester should cause stress to the system with test cases/ conditions that confuses the system and makes it unable to react.

- Example :
- Generate 10 interrupts per sec where average is 2/3
- Test cases that require maximum memory
- Test cases which crash the system

- All such cases are evaluated and tests the maximum level of a system behaviour

System Testing

Performance Testing

- Is conducted internally in all testing types.
- Essentially it is done as part of system testing to check resource utilization.
- Test cases are designed to test run time performance of the system

Deployment Testing

- It evaluates a system over different hardware and software platforms.
- Check of installation procedures and configuration issues.
- It is also called as configuration testing

Basis Path Testing

- It is one of the methods of WHITE BOX TESTING. It ensures all the procedures and modules are executed at least once by covering all possible paths.

- To compute all possible paths we need to follow certain metrics while designing the TEST.

- The below 4 ideas are used before planning the test

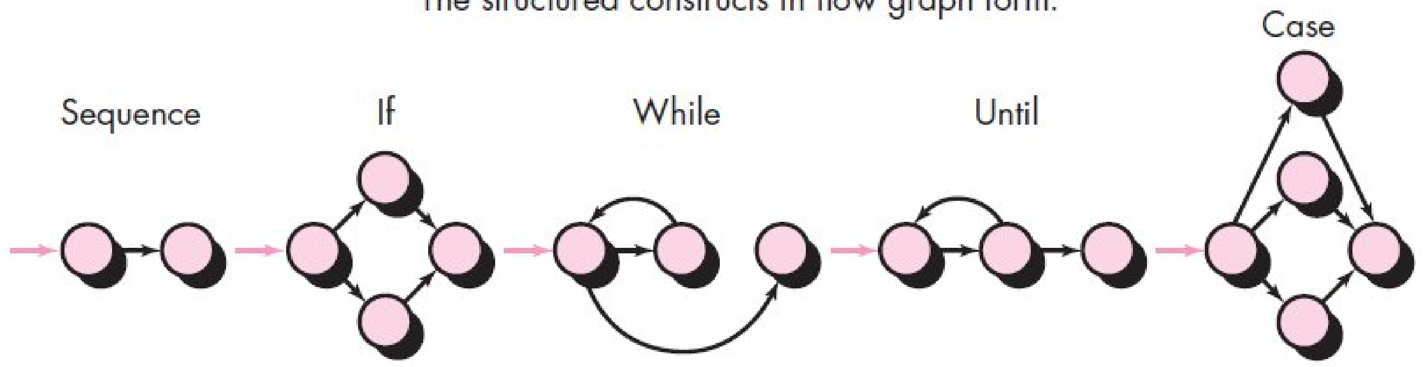
  - Flow Graph Notation
  - Independent Program Paths
  - Deriving Test Cases
  - Graph Matrices

Basis Path Testing

Flow Graph Notation

- It is a flowchart like structure which consists of nodes, edges, regions.
- For every structure in code like if, while, loops, sequences we have graph notations

The structured constructs in flow graph form:

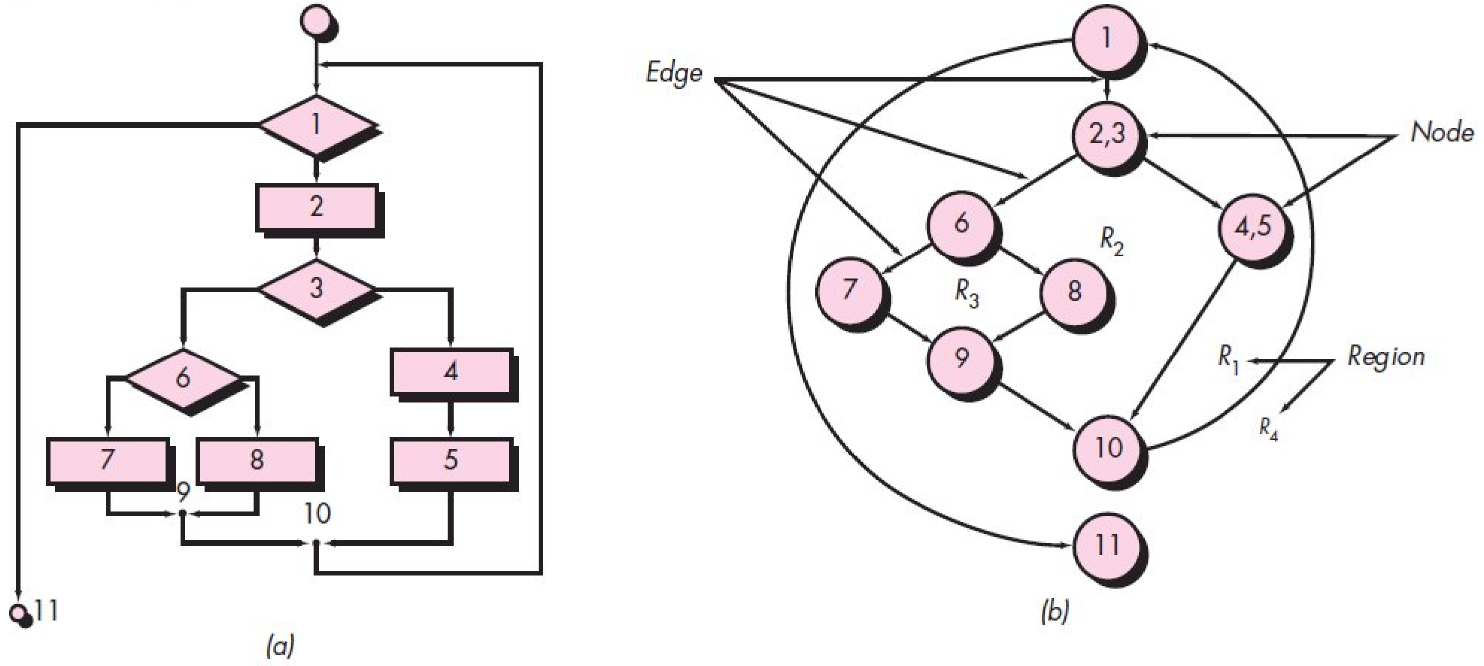


Where each circle represents one or more nonbranching PDL or source code statements

Basis Path Testing

- Sample flow graph

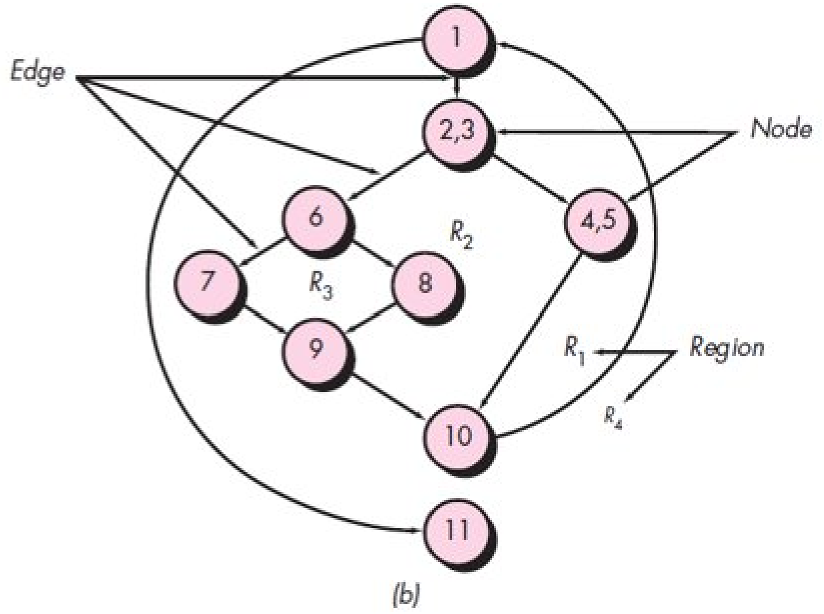
FIGURE 18.2 (a) Flowchart and (b) flow graph



Basis Path Testing

Independent Program Paths

A path that follows a direction from start to end with no nodes being visited more than once is called as independent path.



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

Basis Path Testing

- To determine number of independent paths in a flowgraph, we have a measure called as **Cyclomatic complexity.**

- It is computed by simple formula as

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

$$V(G) = E - N + 2$$

E 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

$$V(G) = P + 1$$

P is the number of predicate nodes contained in the flow graph *G.*

Basis Path Testing

Deriving Test Cases

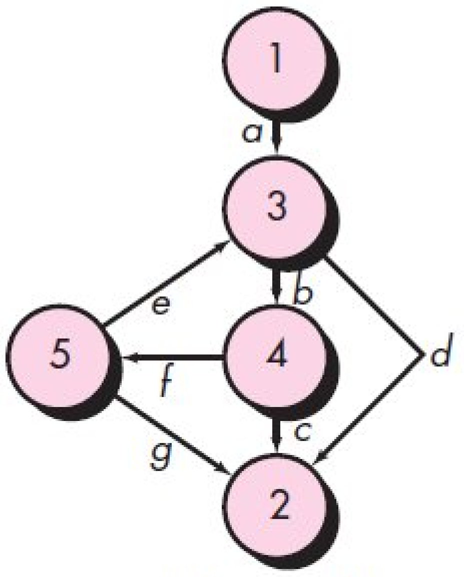
1. Using the design or code as a foundation, draw a corresponding flow graph.
2. Determine the cyclomatic complexity of the resultant flow graph.
   This value gives maximum independent paths than can be formed
3. Determine a basis set of linearly independent paths
4. Prepare test cases that will force execution of each path in the basis set.

Basis Path Testing

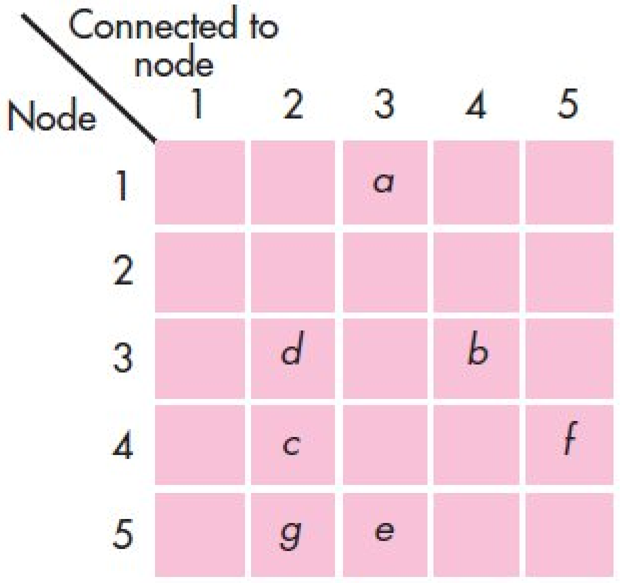
Graph Matrices

- A data structure, called a *graph matrix,* is a great tool used for developing a software tool that assists in basis path testing.
- A graph matrix is a square matrix whose size 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
- We can also add weight called as links to each matrix entry and compute the cost of construction.

Basis Path Testing



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