

COURSE NAME

SOFTWARE  
ENGINEERING

CSC 3114

(UNDERGRADUATE)

---

## CHAPTER 10

### SOFTWARE TESTING

---

PROF. DR. KAMRUDDIN NUR  
PROFESSOR, CS, AIUB

# SOFTWARE TESTING

- ❑ Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user
- ❑ Software testability is simply how easily [a computer program] can be tested

## Testing Shows

- Error
- Requirements Conformance
- Performance
- An indication of quality

# WHO TESTS THE SOFTWARE?

## Developer

- Understands the system but, will test "gently" and, is driven by "delivery"
- Experiencing the software operation (known to the developer)

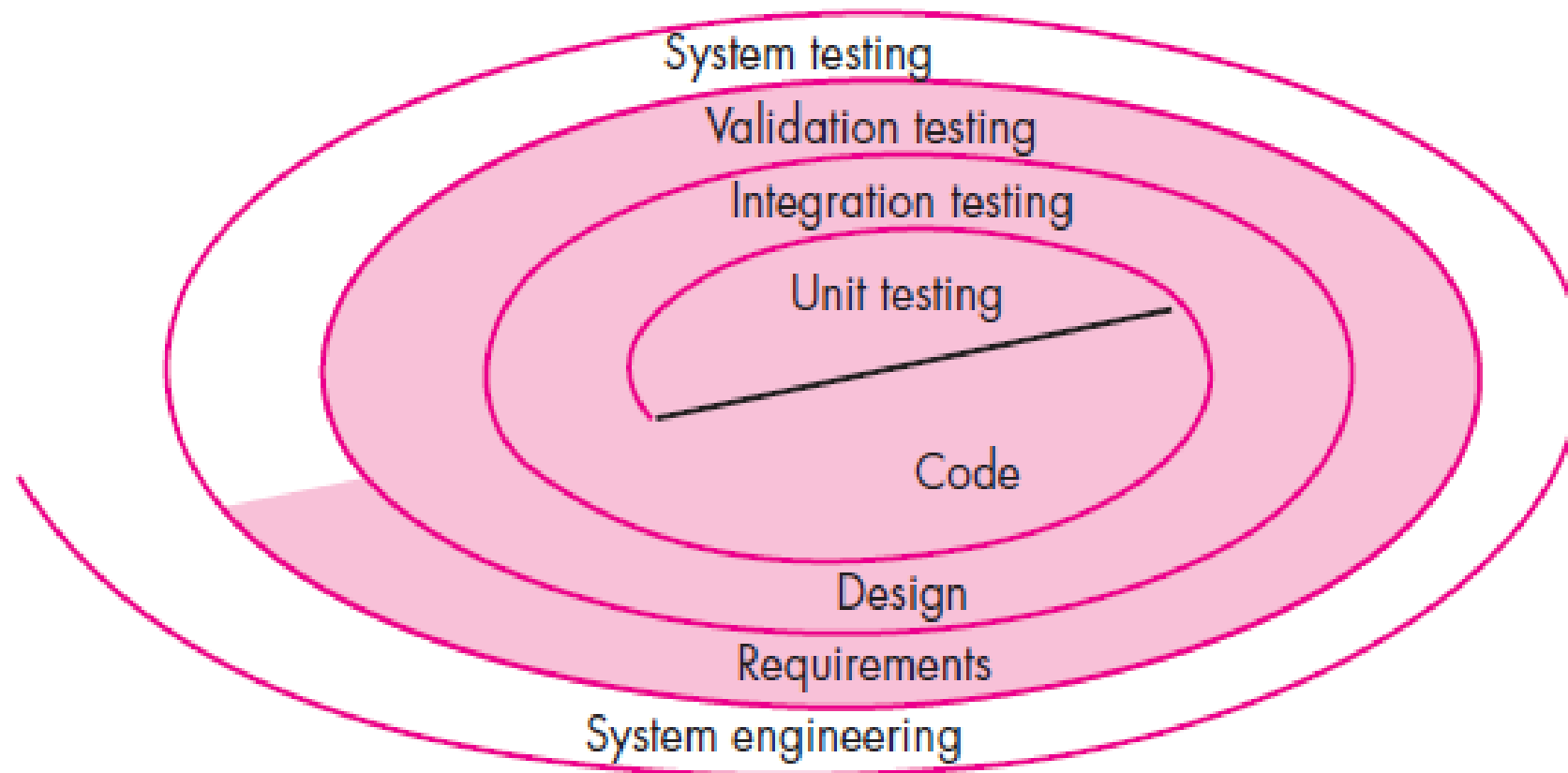
## Independent tester

- Must learn about the system, but, will attempt to break it and, is driven by "quality"
- Exploring the software operation (unknown to the tester)

## V & V

- ❑ *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- ❑ *Verification* refers to the set of tasks that ensure that software correctly implements a specific function/process.
- ❑ Boehm states this another way:
  - *Validation*: "Are we building the right product?"
  - *Verification*: "Are we building the product right?"

# TESTING STRATEGY



# TESTING STRATEGY

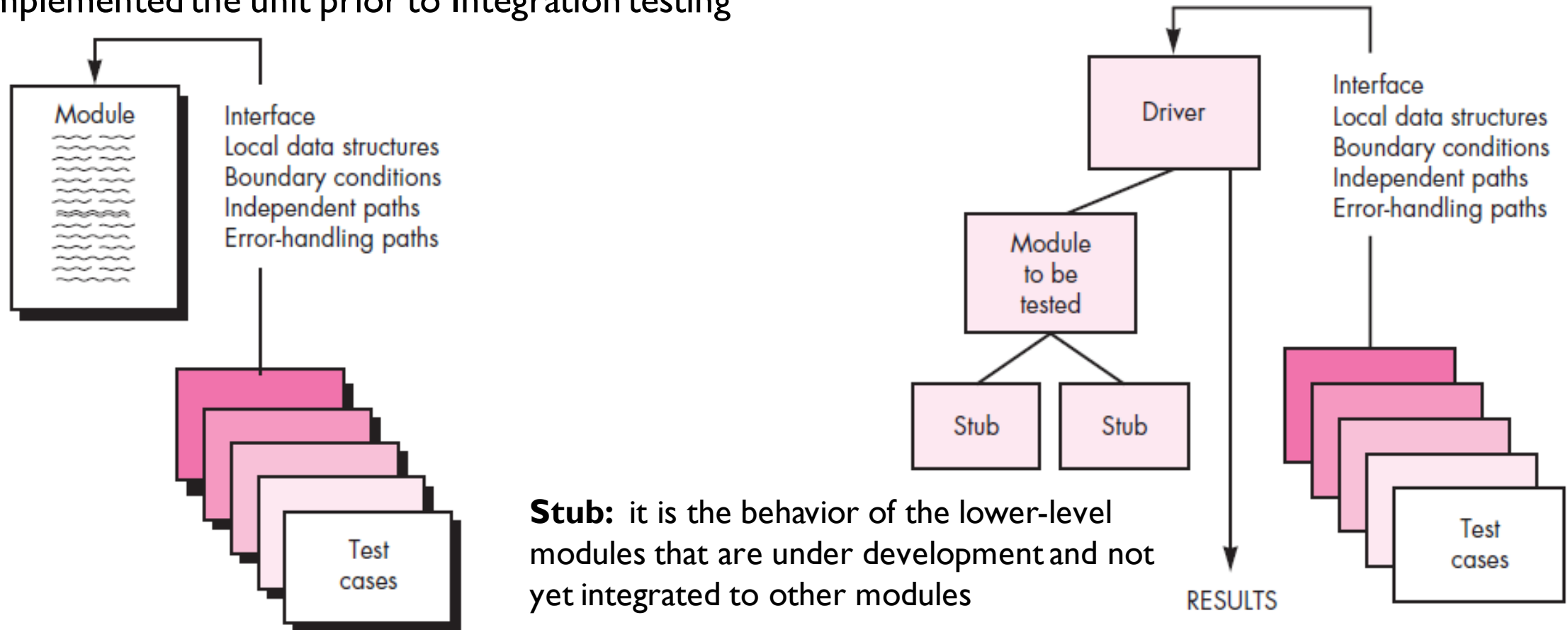
- ❑ We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- ❑ For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- ❑ For OO software
  - Our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

# TESTING STRATEGIC ISSUES

- Specify **product requirements** in a **quantifiable** manner long before testing commences
- State testing objectives explicitly
- Understand the **users of the software** and develop a profile for each user category
- Develop a **testing plan** that emphasizes “**rapid cycle testing**”
- Build “robust” software that is designed to **test itself**
- Use effective **technical reviews as a filter prior to testing**; many errors will be eliminated before testing begins
- Conduct technical reviews to assess the **test strategy and test cases** themselves
- Develop a **continuous improvement** approach for the testing process

# UNIT TESTING

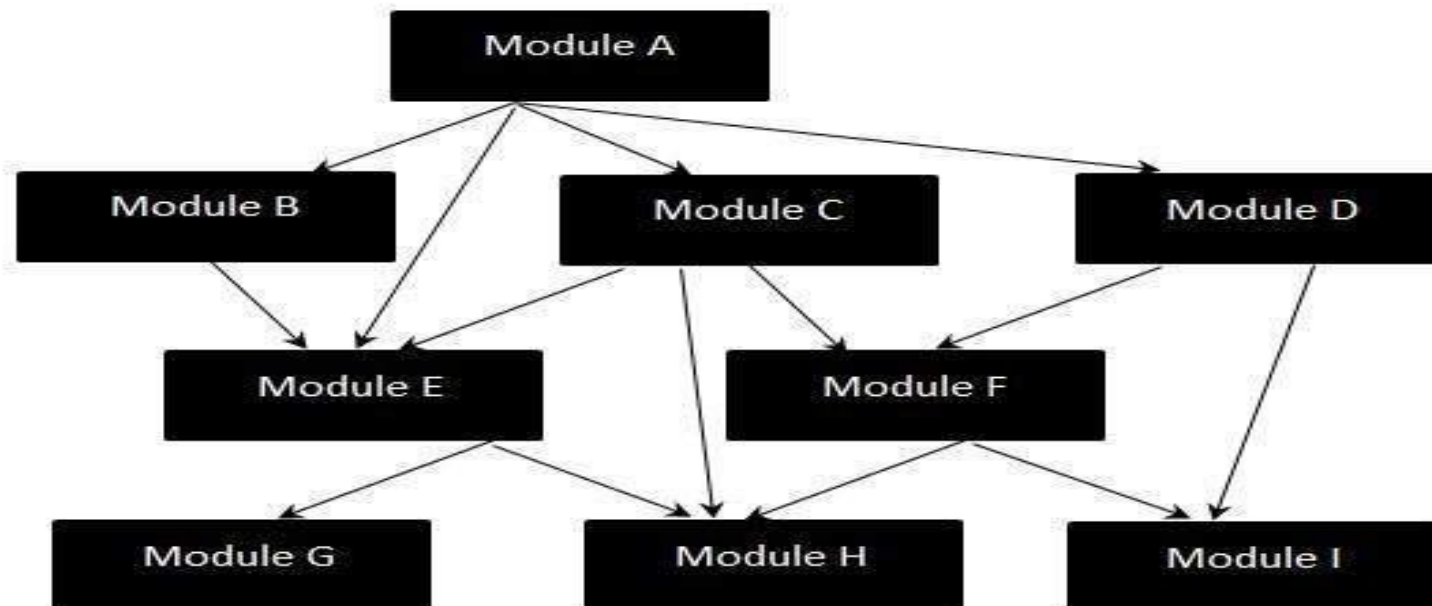
Tests a small software unit at a time, which is typically performed by the individual programmer who implemented the unit prior to Integration testing





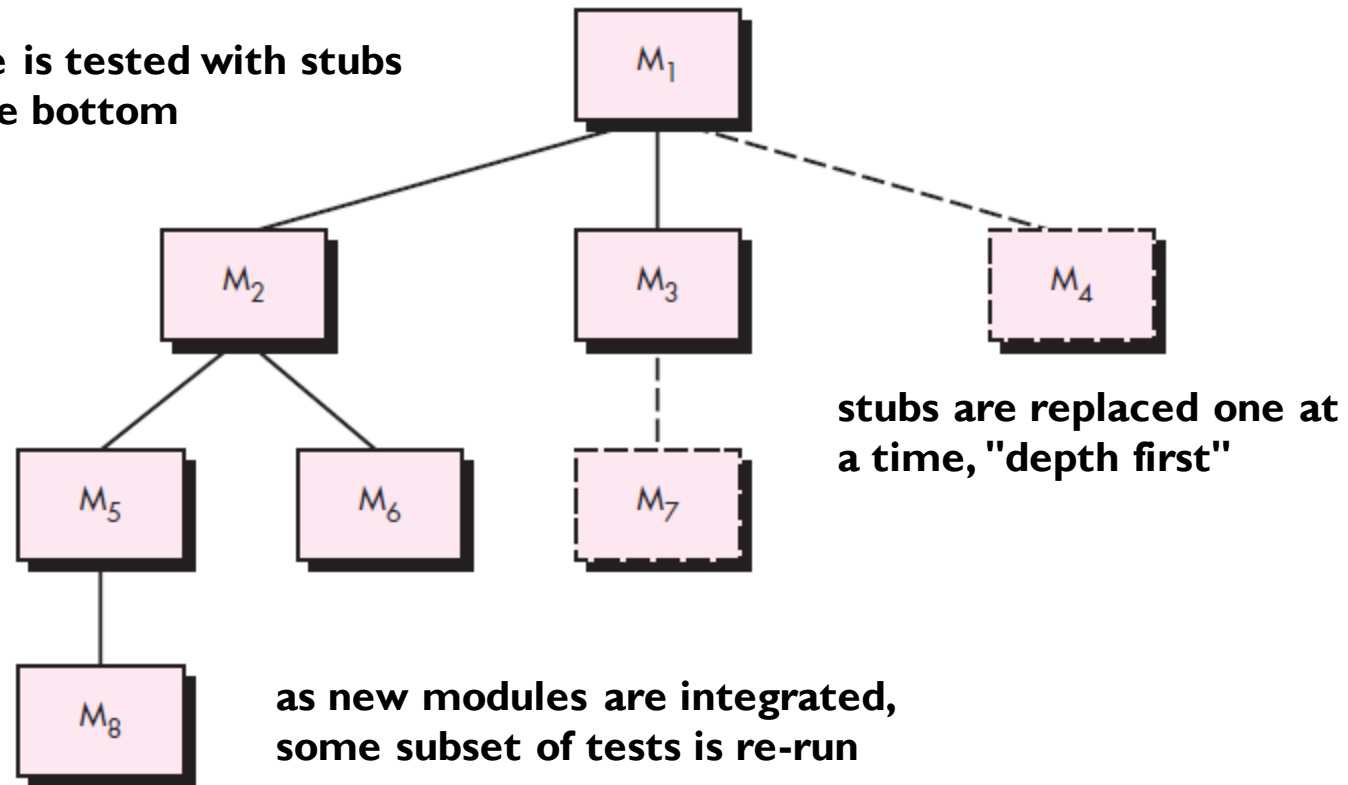
# INTEGRATION TESTING STRATEGIES

- System integration testing (SIT) is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing the modules
- **the “big bang” approach:** Big Bang Integration Testing is an integration testing strategy where all units are linked at once, resulting in a complete system.

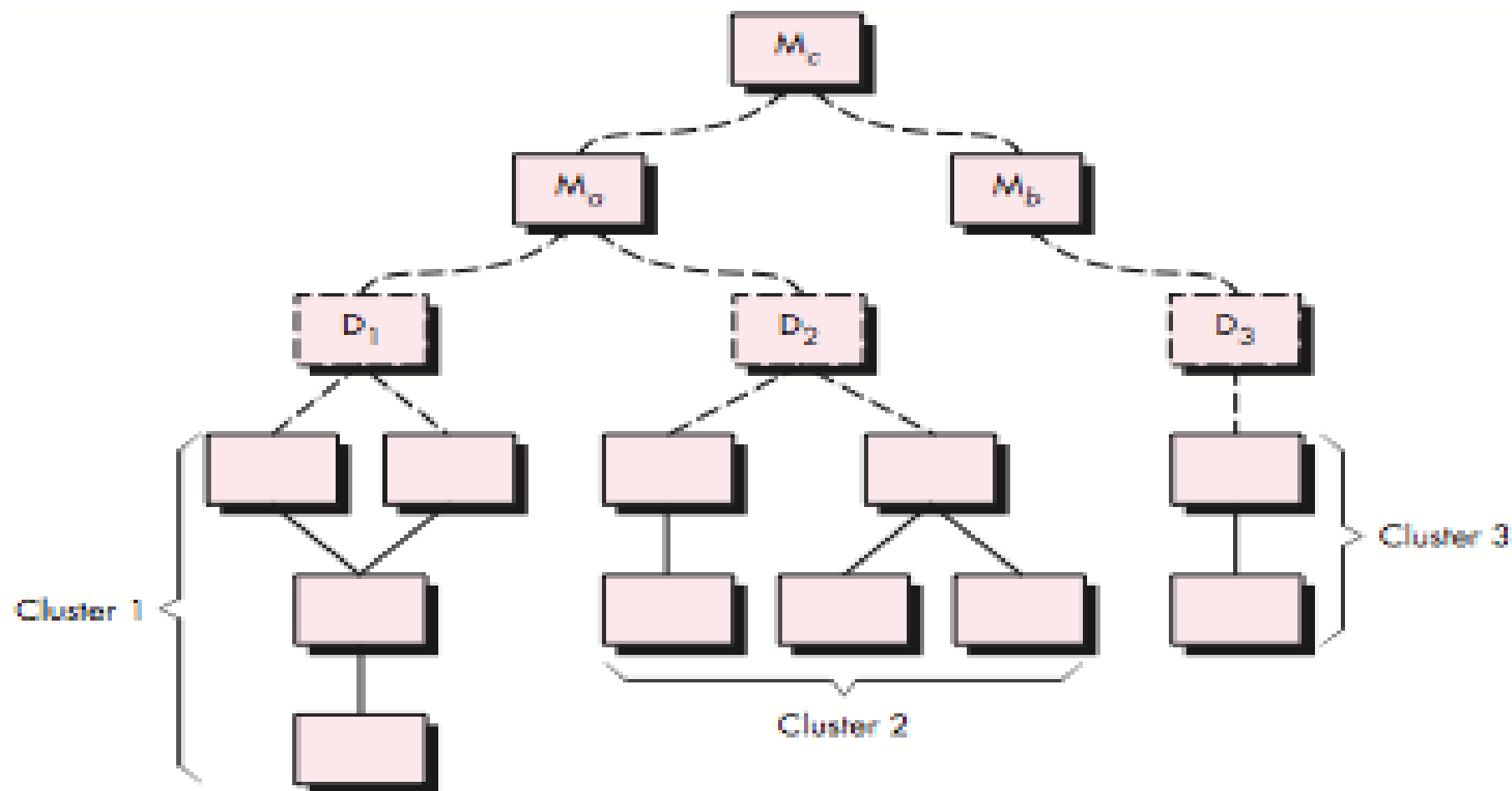


# TOP-DOWN INTEGRATION

top module is tested with stubs  
down to the bottom



# BOTTOM-UP INTEGRATION



# REGRESSION TESTING

- Regression testing is the **re-execution** of some subset of tests that have already been conducted to ensure that **changes** have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce **unintended behavior or additional errors**.
- Regression testing may be conducted manually, by **re- executing a subset of all test cases** or using **automated capture/playback tools**.

# SMOKE TESTING

## Smoke testing steps:

- Software components that have been translated into code are integrated into a “**daily 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 “**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 (in its current form) is smoke tested daily.
  - The integration approach may be top down or bottom up.

# OBJECT-ORIENTED TESTING

- ❑ Class testing is the equivalent of unit testing
  - Operations within the class are tested
  - The state behavior of the class is examined
- ❑ Integration applied three different strategies
  - **Thread-based testing**—integrates the set of classes required to respond one input or event
  - **Use-based testing**—integrates the set of classes required to respond to one use case
  - **Cluster testing**—integrates the set of classes required to demonstrate one collaboration

# HIGHER ORDER TESTING

- **System testing:** focus is on system integration (e.g. hardware integration, OS compatibility)
- **Alpha/Beta testing:** **Alpha testing** is simulated or actual operational **testing** by potential users or an independent **test** team at the developers' site. **Alpha testing** is often employed for off-the-shelf software as a form of internal acceptance **testing**, before the software goes to **beta testing** by users
- **Recovery testing:** forces the software to fail in a variety of ways and verifies that recovery is properly performed
- **Security testing:** verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- **Stress testing:** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance Testing:** test the run-time performance of software within the context of an integrated system (e.g. time required to response a request, compliance with operational constraints)

# DEBUGGING

- ❑ In many cases, the non-corresponding data are a symptom of an underlying cause as yet hidden error
- ❑ The debugging process attempts to match symptom with cause, thereby leading to error correction
  
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes



# DEBUGGING TECHNIQUES

## ❑ Brute force testing

- most common;but least efficient
- memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements (**Dynamic Testing**)

## ❑ Backtracking

- common debugging approach that can be used successfully in small programs
- source code is traced backward (manually) until the cause is found

## ❑ Cause elimination

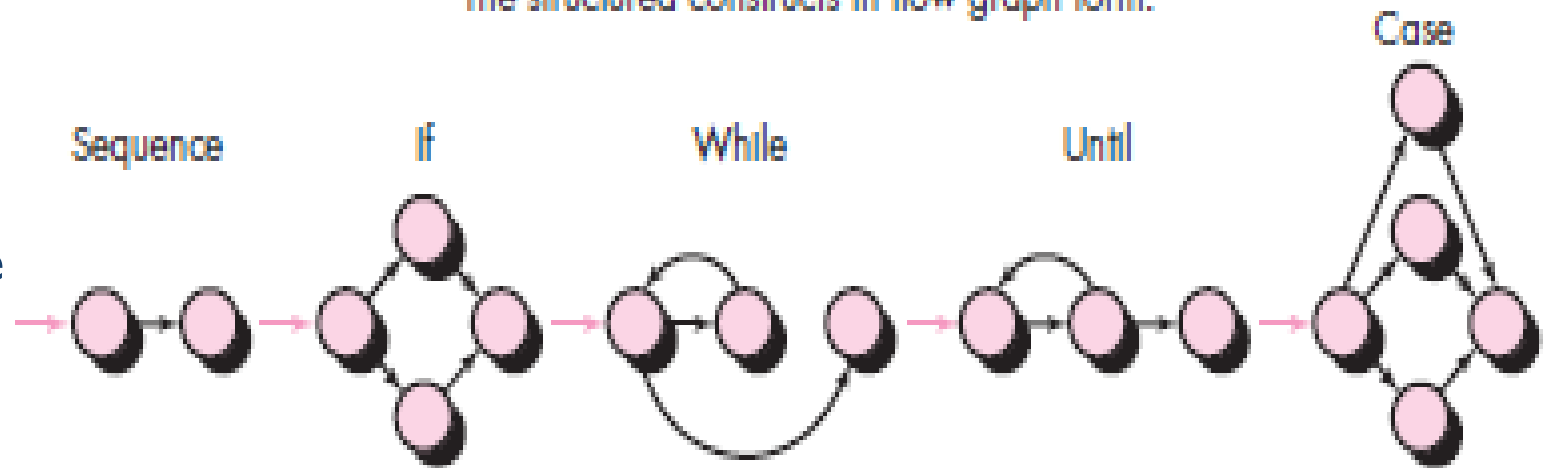
- a “cause hypothesis” is devised
- if initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug (c/a-b where the possibility of a-b is zero)

# BASIS-PATH TESTING

- 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
- McCabe views a program as a directed graph in which lines of program statements are represented by nodes and the flow of control between the statements is represented by the edges

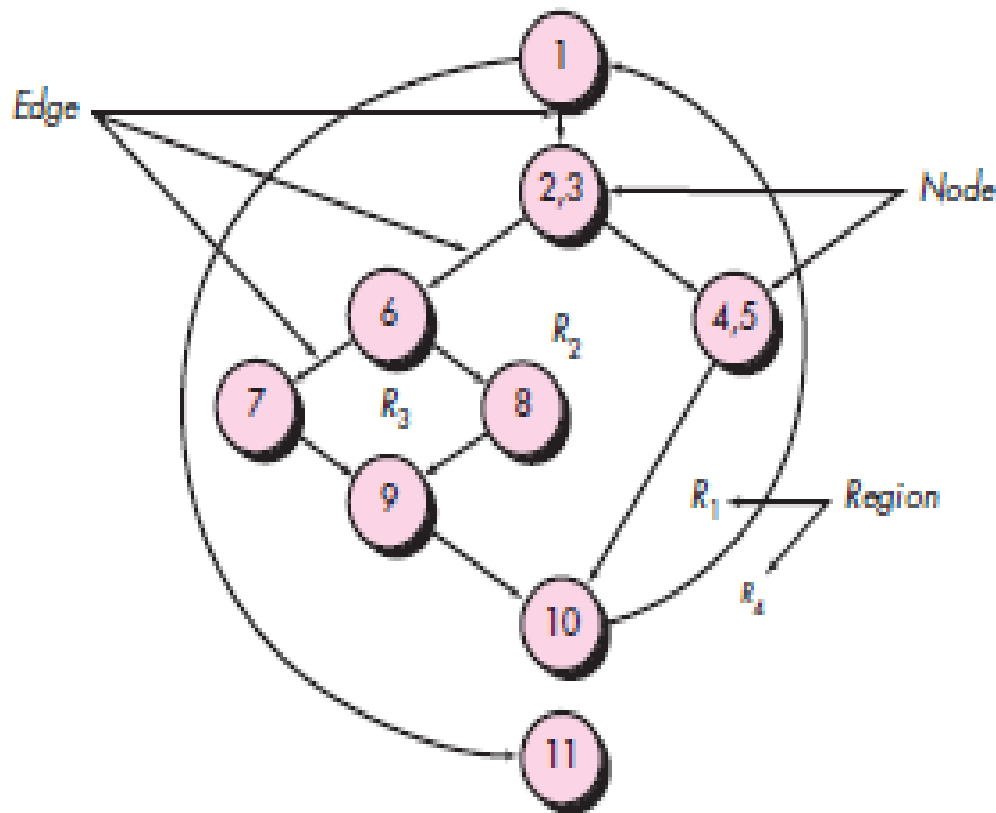
## Flow Graph

The structured constructs in flow graph form:



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

# INDEPENDENT PROGRAM PATHS



Path 1: 1-11

Path 2: 1-2-3-4-5-10-11

Path 3: 1-2-3-6-8-9-10-11

Path 4: 1-2-3-6-7-9-10-11

- Note that each new path introduces a new edge. The path 1-2-3-4-5-10-1-2-3-6-8-9-10-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- How do you know how many paths to look for? The computation of **cyclomatic complexity** provides the answer

# CYCLOMATIC COMPLEXITY

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. Complexity is computed in one of three ways:

1. The number of independent paths
2. The number of regions of the flow graph corresponds to the cyclomatic complexity.  
(in the previous example = 4)
3. Cyclomatic complexity  $V(G)$  for a flow graph  $G$  is defined as  $V(G) = E - N + 2$   
(in the previous example  $11 - 9 + 2 = 4$ )
  - where  $E$  is the number of flow graph edges and  $N$  is the number of flow graph nodes.
4. Cyclomatic complexity  $V(G)$  for a flow graph  $G$  is also defined as  $V(G) = P + 1$   
(in the previous example  $3 + 1 = 4$ ) [condition: 1; 2,3;6]
  - where  $P$  is the number of predicate nodes (containing a condition) contained in the flow graph  $G$

# WHITE-BOX TESTING

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.

# BLACK-BOX TESTING

- Focuses on the functional requirements of the software
- Black-box testing attempts to find errors in the following categories:
  - (1) incorrect or missing functions
  - (2) interface errors
  - (3) errors in external database access (accessibility)
  - (4) behavior or performance errors
  - (5) initialization and termination errors

# REFERENCES

- R.S. Pressman & Associates, Inc. (2010). *Software Engineering: A Practitioner's Approach*.
- Kelly, J. C., Sherif, J. S., & Hops, J. (1992). An analysis of defect densities found during software inspections. *Journal of Systems and Software*, 17(2), 111-117.
- Bhandari, I., Halliday, M. J., Chaar, J., Chillarege, R., Jones, K., Atkinson, J. S., & Yonezawa, M. (1994). In-process improvement through defect data interpretation. *IBM Systems Journal*, 33(1), 182-214.