# Testing

Chap. 11

CSCI 335

# Verification and Validation (V & V)

- Verification checks that the software meets it functional and non-functional requirements.
- Validation ensures that the software meets the customer's expectations.

- The goal of V & V is to establish confidence that the software is 'fit for purpose'.

# Testing

- In order to show that a program does what it is intended to do and to discover defects before it is released.
- Goals
  - Show that the system operates as intended.
  - Expose any defect in the system.
- "Testing can only show the presence of bugs, not their absence" (Dijkstra).
  - Testing is not free.

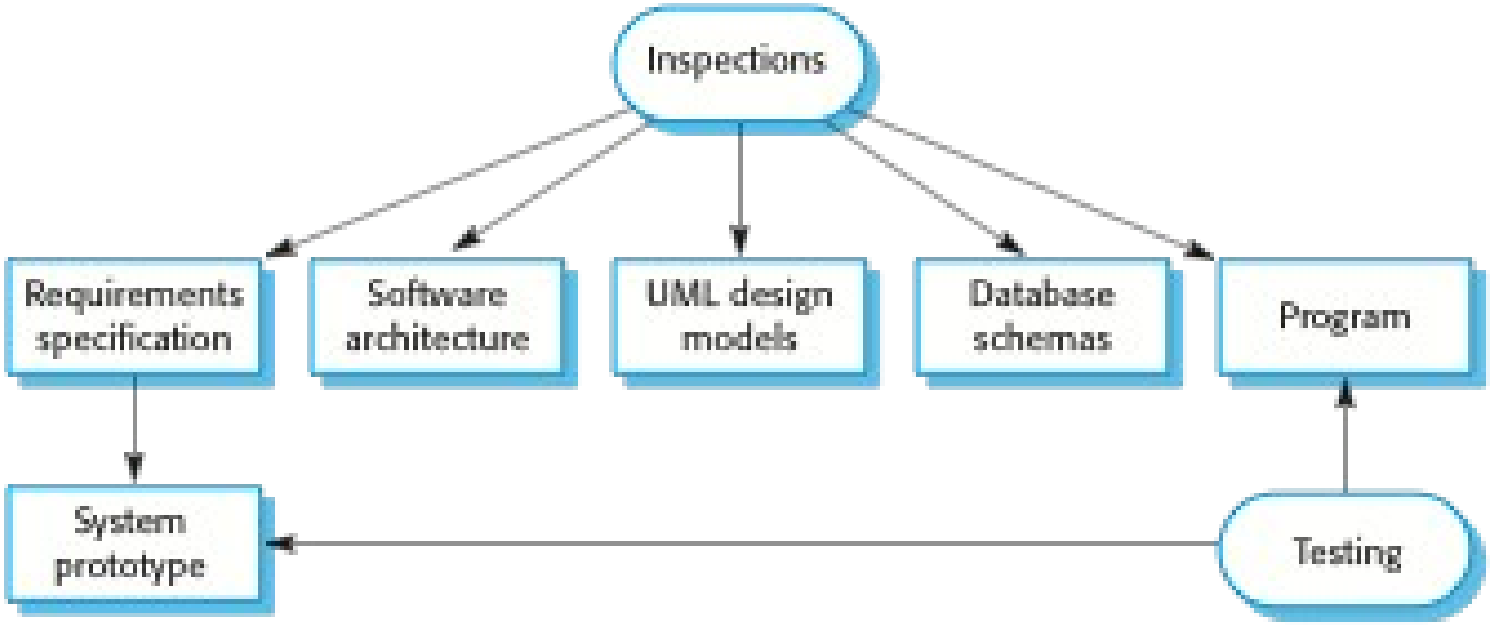
# Independent Testing

- Testing is done best by independent testers.
  - Programmer often stick to the data set that makes the program work.
    - "Don't mess up my code!"
  - A program often does not work when tried by somebody else.
- For an effective test
  - Detailed understanding of the system
  - Knowledge of the testing techniques
  - Skill to apply these techniques in an effective and efficient manner

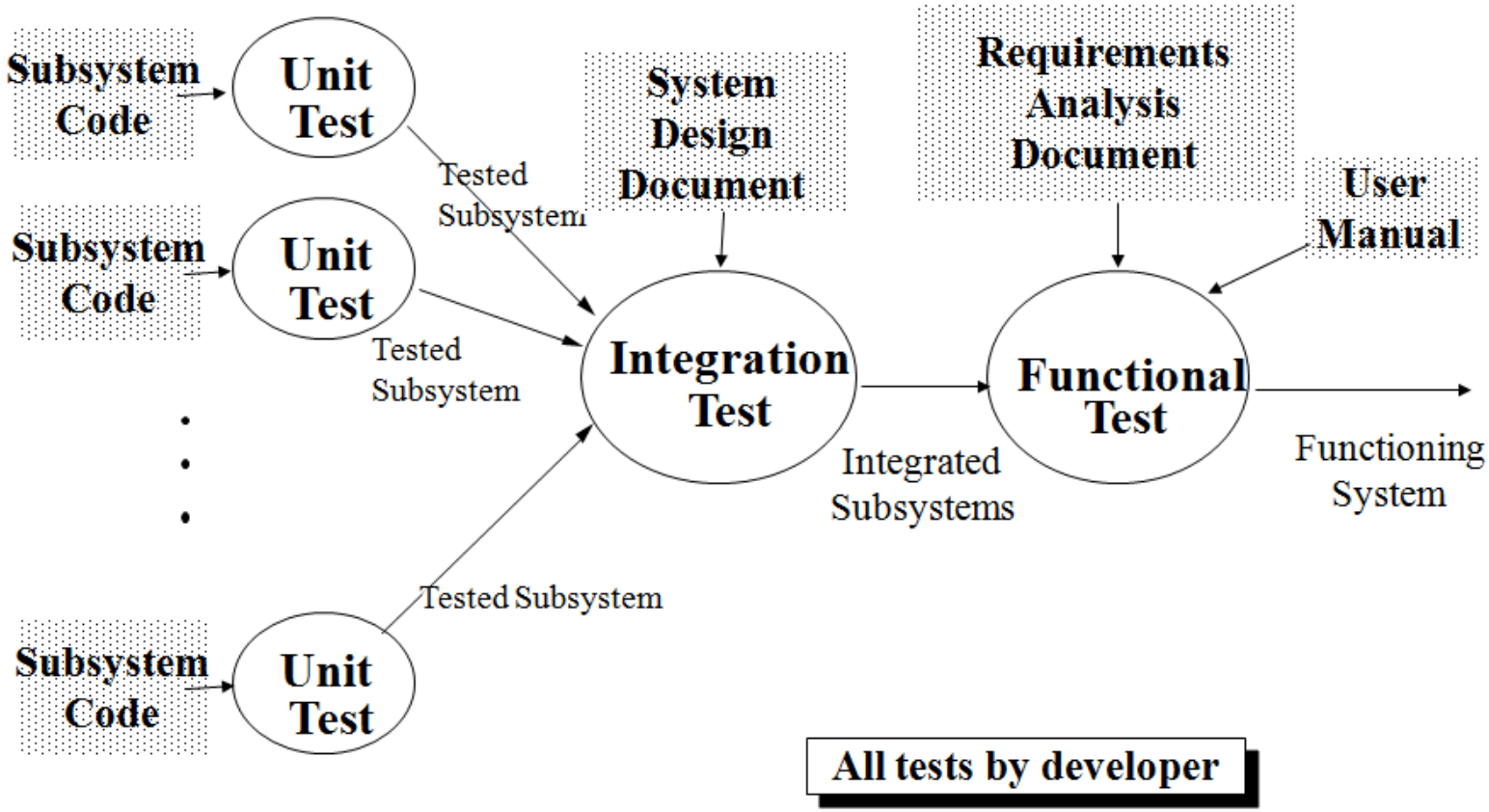
# Software Inspection and Testing

- Inspection
  - Concerned with analysis of the static system representation to discover problems (static verification)
- Testing
  - Concerned with exercising and observing product behavior (dynamic verification)

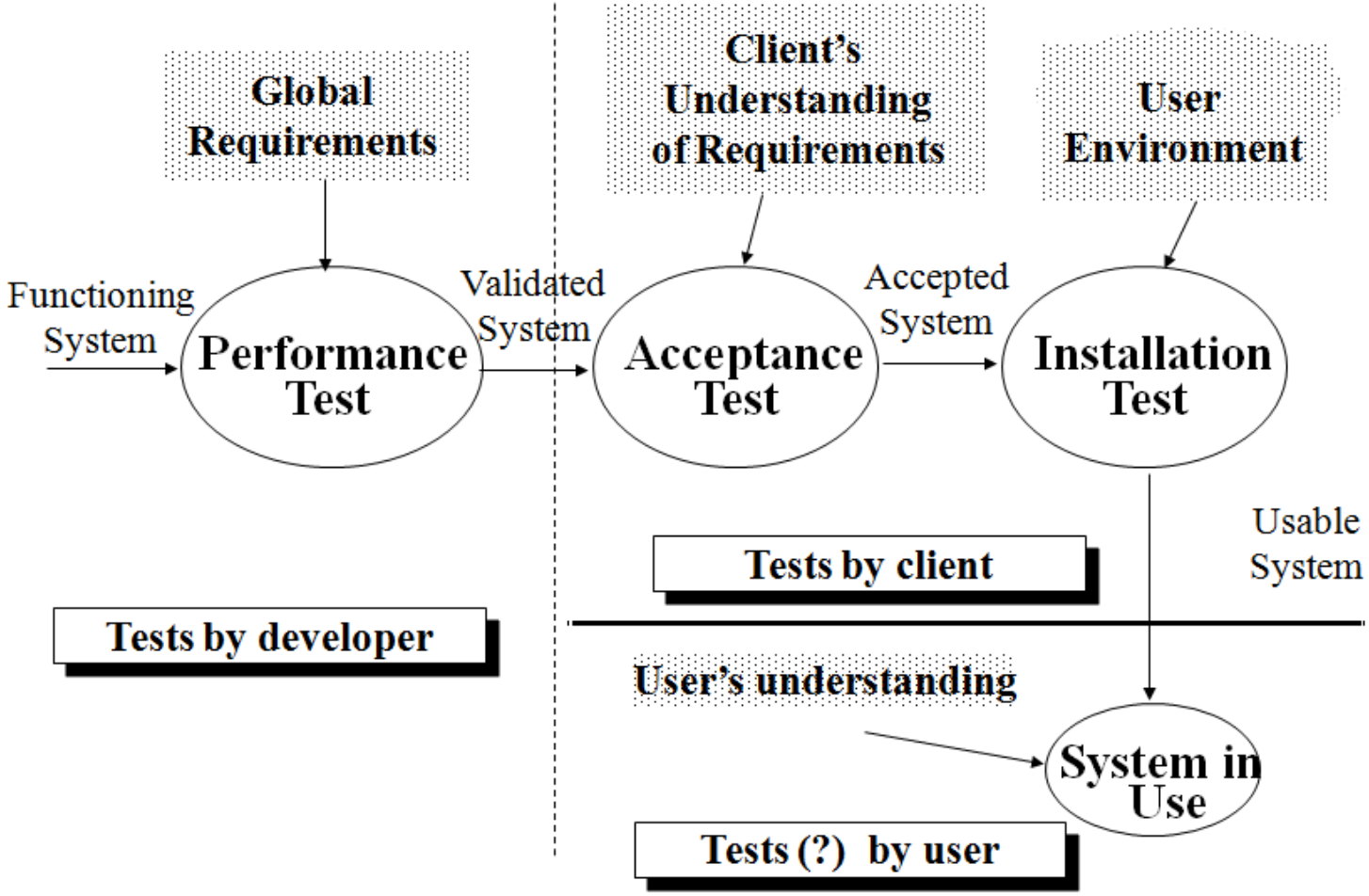
# Software Inspection and Testing (cont.)



# Types of Testing



# Types of Testing (cont.)



# Unit Testing

- Informal: incremental coding
- Static testing
  - Static Analysis
  - Review: walk-through; code inspection
- Dynamic testing
  - White-box testing
    - Testing the internal logic of the subsystem or object
  - Black-box testing
    - Testing the input/output behavior

# Appendix: The Static Analyzer

Use the static analyzer to find bugs in your code before you even run your app. The static analyzer tries out thousand
few seconds, reporting potential bugs that might have remained hidden or bugs that might be nearly impossible to re
identifies areas in your code that don't follow recommended API usage, such as Foundation, UIKit, and AppKit idioms

To perform static code analysis, choose Product > Analyze. The Xcode static analyzer parses the project source code
problems:

- Logic flaws, such as accessing uninitialized variables and dereferencing null pointers
- Memory management flaws, such as leaking allocated memory
- Dead store (unused variable) flaws
- API usage flaws that result from not following the policies required by the frameworks and libraries the project is u

The static analyzer reports problems in the issue navigator, available by clicking the Issue Navigator button in the prc
analyzer message in the issue navigator to display the associated code in the source editor. Click the corresponding r
Use the pop-up menu in the analysis results bar above the source code editor to study the flow path of the flaw. Ther

## Performing Static Code Analysis

Find flaws—potential bugs—in the source code of a project with the static analyzer built into Xcode. Source code may
the compiler and manifest themselves only at runtime, when they could be difficult to identify and fix.

To find flaws in your source code using the static analyzer:

1. Choose Product > Analyze.
2. In the issue navigator, select an analyzer message.

# White-box Testing

- Focus: coverage
  - Every statement in the component is executed at least once.
- Types of white-box testing
  - Statement testing
  - Loop testing
    - Cause execution of the loop to be skipped completely (exception: Repeat loops)
    - Loop to be executed exactly once
    - Loop to be executed more than once
  - Path testing
  - Branch testing

**Exercise #11 Q1-Q3**

```
FindMean(float Mean, FILE ScoreFile)
{ SumOfScores = 0.0; NumberOfScores = 0; Mean = 0;
 Read(ScoreFile, Score);  /*Read in and sum the scores*/
 while (! EOF(ScoreFile) {
         if ( Score > 0.0 ) {
               SumOfScores = SumOfScores + Score;
              NumberOfScores++;
          }
          Read(ScoreFile, Score);
 }
 /* Compute the mean and print the result */
 if (NumberOfScores > 0 ) {
             Mean = SumOfScores/NumberOfScores;
             printf("The mean score is %f \n",  Mean);
 } else
             printf("No scores found in file\n");
}
```

```
public class GeneratePrimeNumbersExample {

   public static void main(String[] args) {

        int limit = 100;

        System.out.println("Prime numbers between 1 and " + limit);

        for(int i=1; i < 100; i++){

           boolean isPrime = true;

           for(int j=2; j < i ; j++){

        if(i % j == 0){
           isPrime = false;
           break;
        }
           }
           // print the number
           if(isPrime)
        System.out.print(i + " ");
        }
   }
}
```

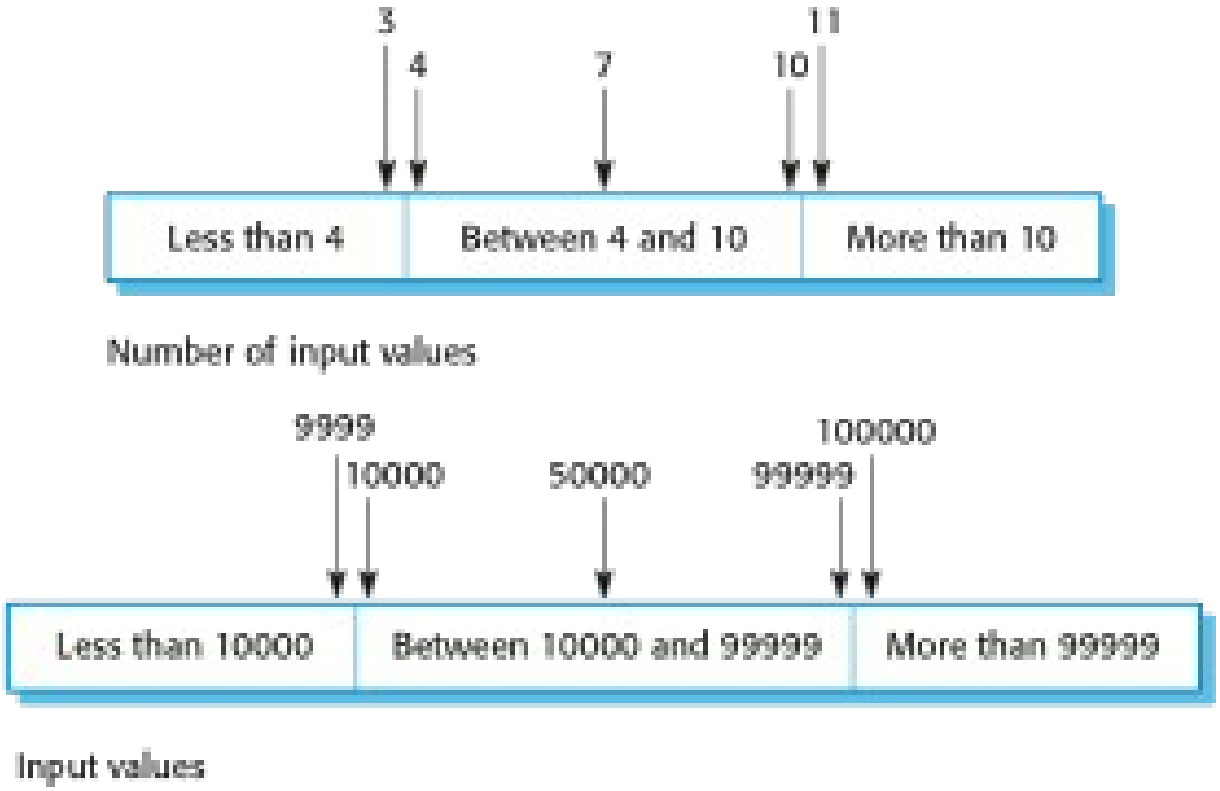
# Complexity of Control Structure

- McCabe complexity metric
  - Between 1 and 10: simple and easy to understand
  - Between 11 and 20: more complex, but still be comprehensible
  - More than 20: very difficult to test
  - Over 50: unmaintainable

# Black-box Testing

- Focus: I/O behavior
  - Reduce number of test cases by equivalence partitioning.
  - Divide input conditions into equivalence partitions.
  - Choose test cases for each partition.
- For example, input is valid across range of values.
  - Select test cases from 3 equivalence partitions:
    - Below the range
    - Within the range
    - Above the range

# Examples of Equivalence Partitioning



```
/**
 * Insert a new item at the right place in an (assumed sorted) list.
 * @param item The item to insert in the right place.
 */
public void insertSorted(int item) {
```

**Exercise #11 – Q4**

# Test Cases

- Name
  - Name of test case
- Location
  - Full path name of executable
- Input
  - The set of input data or commands to be entered
- Oracle
  - Expected test results against which the output of the test is compared
- Log
  - Output produced by the test

# Comparison of White and Black-box Testing

- Both types of testing are needed.
  - These are the extreme ends of a testing continuum.
- White-box testing
  - It often tests what is done, instead of what should be done.
    - Cannot detect missing use cases.
- Black-box testing
  - It does not discover extraneous use cases ("features").

# Integration Testing

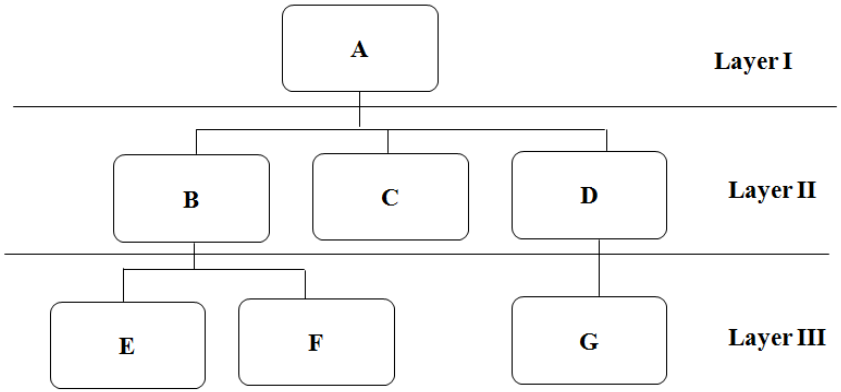
- Focuses on small groups of components.
  - Includes interface testing
- Two or more components are integrated and tested, and when no new faults are revealed, additional components are added to the group.

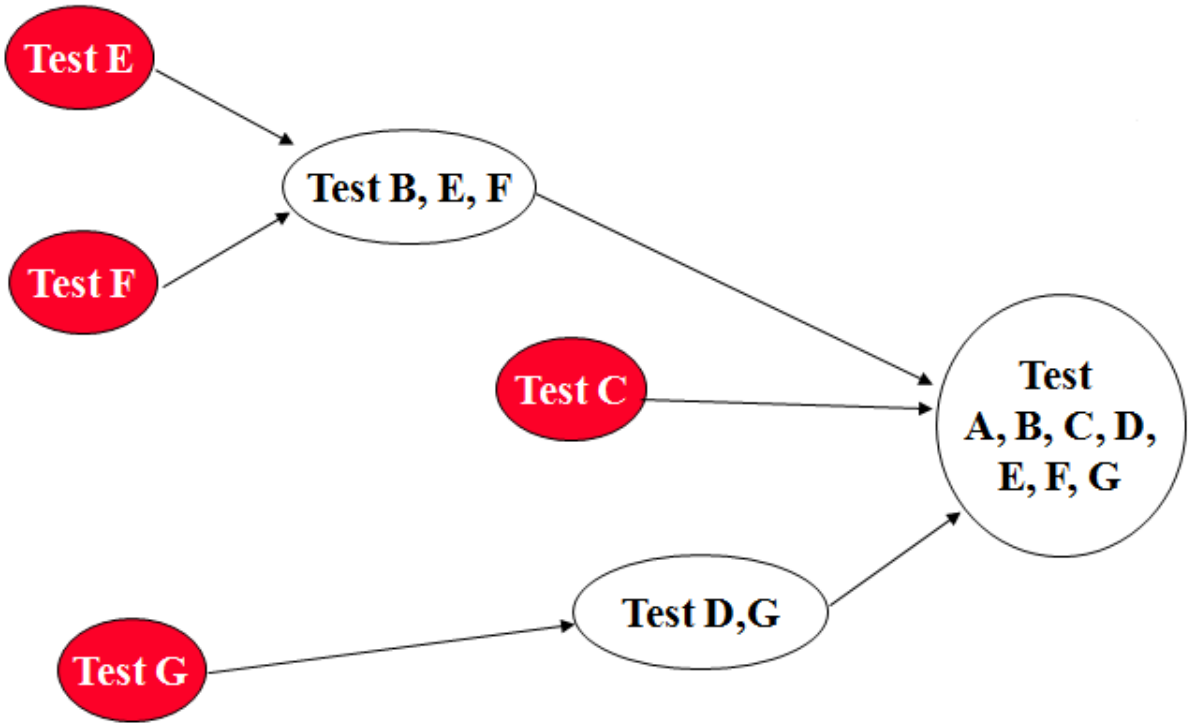
# Integration Testing Strategy

- Big bang integration
  - Non-incremental
- Bottom-up integration
- Top-down integration
- Sandwich testing
- Variations of the above

# Bottom-up Integration

- The subsystems in the lowest layer of the call hierarchy are tested individually.
- Then the next subsystems are tested that call the previously tested subsystems.
- This is done repeatedly until all subsystems are included in the testing.
- A special program needed to do the testing, Test driver:
  - A routine that calls a subsystem and passes a test case to it


A
Layer I
B
C
D
Layer II
E
F
G
Layer III

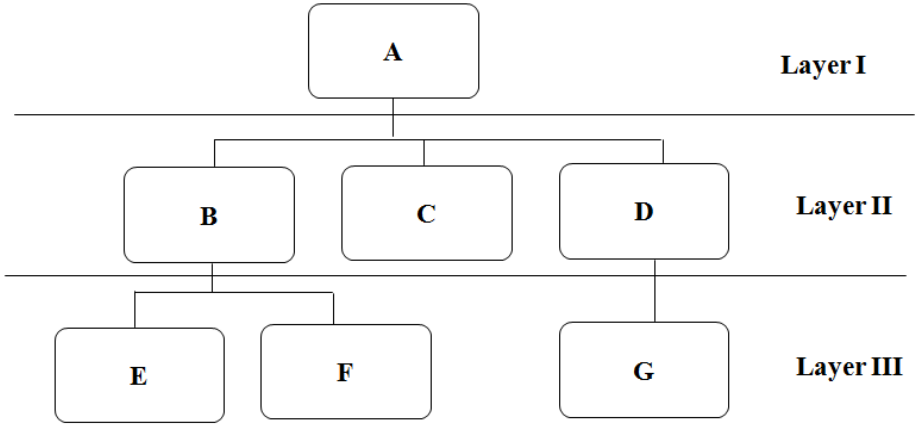

Test E
Test B, E, F
Test F
Test C
Test
A, B, C, D,
E, F, G
Test D,G
Test G

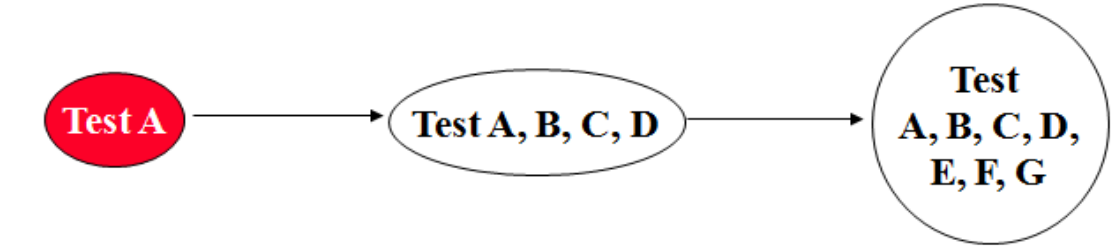
# Bottom-up Integration (cont.)

- Interface faults can be more easily found.
  - The developers have a clear model of how the lower-level component works and of the assumptions embedded in its interface.
- It tests the most important subsystems, namely the components of the UI, last.
  - Faults found in the top layer may often lead to changes in the subsystem decomposition invalidating previous tests.

# Top-down Integration

- The top layer or the controlling subsystem is tested first.
- Then all the subsystems that are called by the tested subsystems are combined and tested.
- This is done repeatedly until all subsystems are included in the testing.
- A special program is needed to do the testing, Test stub:
  - A program or a method that simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data.




Layer I

Layer I + II

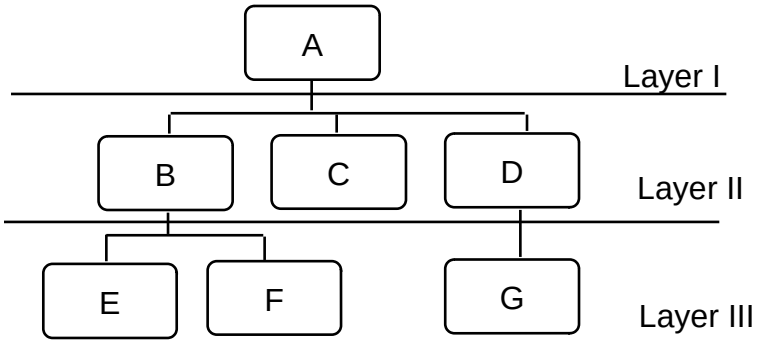
All Layers

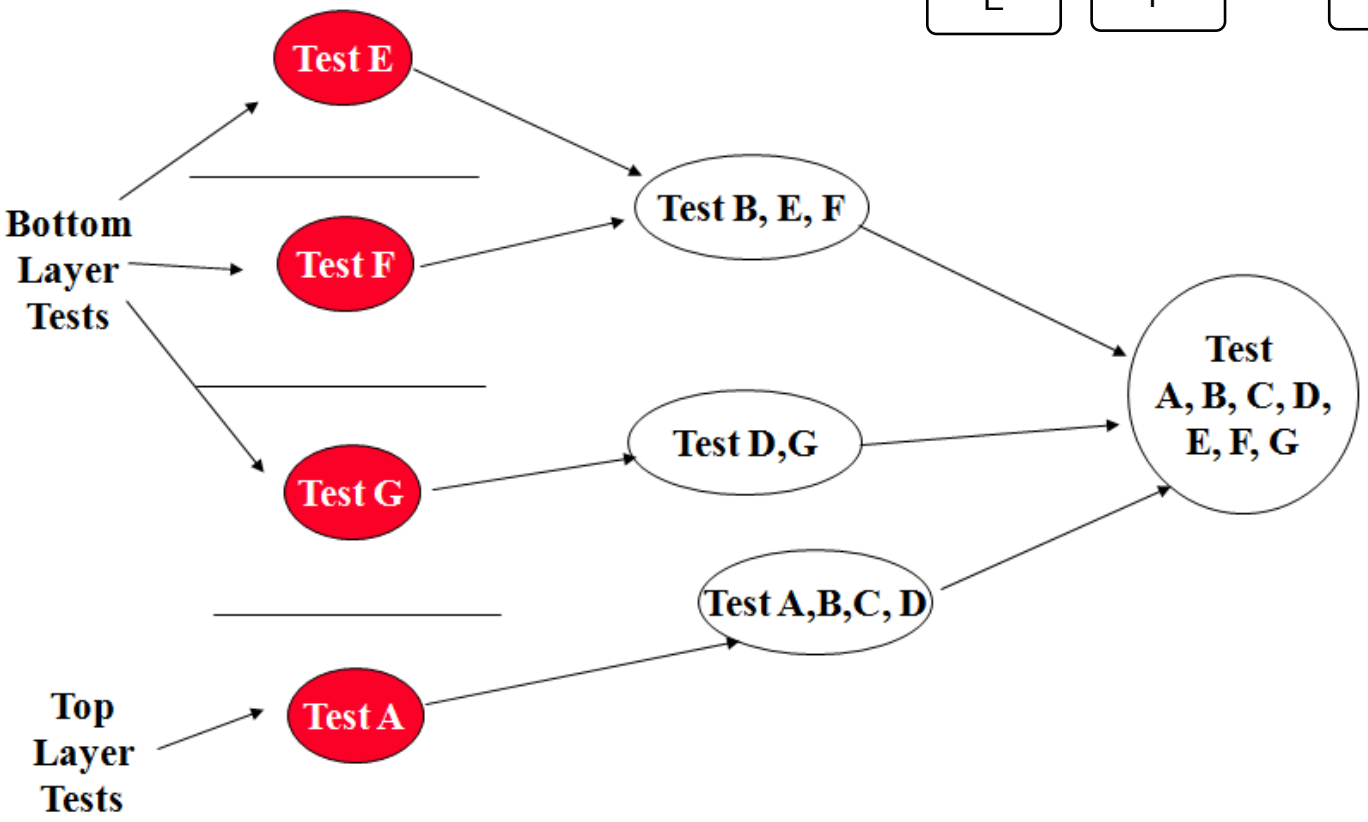
# Top-down Integration (cont.)

- Test cases can be defined in terms of the functionality of the system (functional requirements).
- Writing stubs can be difficult.
  - Stubs must allow all possible conditions to be tested.
  - Possibly a very large number of stubs may be required.

# Sandwich Testing

- Combines top-down with bottom-up strategy.
- The system is considered as having three layers.
  - A target layer in the middle
  - A layer above the target
  - A layer below the target
- Testing converges at the target layer.


A
Layer I
B
C
D
Layer II
E
F
G
Layer III


Test E
Bottom Layer Tests
Test F
Test G
Test B, E, F
Test D,G
Test A,B,C, D
Test A, B, C, D, E, F, G
Top Layer Tests
Test A

# Which Integration?

- Factors to consider
  - Amount of test harness (stubs and drivers)
  - Location of critical parts
  - Availability of hardware and components
- Bottom-up approach
  - Good for OO approach
  - Top-level components are important and cannot be neglected up to the end of testing.
- Top-down approach
  - Test cases are defined in terms of functions.
  - Need to maintain correctness of test stubs

# System Testing

- Structure testing
  - Essentially the same as white-box testing
- Functional testing
  - Essentially the same as black-box testing
- Performance testing
- Acceptance testing
- Installation testing

# System Testing (cont.)

- Impact of requirements on system testing
  - The more explicit the requirements, the easier they are to test.
  - Quality of written specifications determines the ease of functional testing.
  - Quality of nonfunctional requirements and constraints determines the ease of performance testing.

# Structure Testing

- Goal: cover all paths in the system design
  - Exercise all input and output parameters of each component.
  - Exercise all components and all calls.

# Functional Testing

- Goal: test functionality of system
- Test cases are designed from the requirements analysis document and centered around requirements and key functions (use cases)

**Exercise #12**

# Performance Testing

- Stress testing
  - Stress limits of system (maximum # of users, peak demands, extended operation).
- Configuration testing
  - Test the various software and hardware configurations.
- Compatibility testing
  - Test backward compatibility with existing systems.
- Timing testing
  - Evaluate response times and time to perform a function.
- Recovery testing
  - Tests system's response to presence of errors or loss of data.

# Test Cases for Performance Testing

- Push the system to its limits.
  - Try to break the system.
- Try unusual orders of execution.
  - Call a receive()  before send()
- Check the system's response to large volumes of data.
  - If the system is supposed to handle 1000 items, try it with 1001 items.

# General Testing Guidelines

- Choose inputs that force the system to generate all error messages.
- Design inputs that cause input buffers to overflow.
- Repeat the same input or series of inputs numerous times.
- Force invalid outputs to be generated.
- Force computation results to be too large or too small.

# Acceptance Testing

- In order to demonstrate system is ready for operational use
- Alpha testing
  - Users of the software work with the development team to test the software at the developer's site.
- Beta testing
  - A release of the software is made available to users to allow them to experiment and to raise problems that they discover.

# Software Testing Life Cycle

- Establish the test objectives
- Design the test cases
- Write the test cases
- Test the test cases
- Execute the tests
- Evaluate the test results
- Change the system
- Do regression testing
  - Checks that changes have not “broken” previously working code.

Withdraw Cash:

1. The use case begins when Bank Customer inserts his/her Bank Card.
2. ValidateCard use case is performed.
    a. ValidateCard checks if the card is acceptable to the ATM or not.
3. The ATM asks for PIN.
4. The Bank Customer provides a PIN.
5. The ATM displays the different alternatives that are available on this unit. The Bank Customer selects "Withdraw Cash".
6. The ATM prompts for an account.
7. The Bank Customer selects an account.
8. The ATM prompts for an amount.
9. The Bank Customer enters an amount.
10. Card ID, PIN, amount, and account are sent to Bank as a transaction. The Bank Consortium replies with a go/no. The go reply indicates that the transaction is ok.
11. Then money is dispensed.
12. The Bank Card is returned.
13. The receipt is printed.

Develop equivalent partitions and the corresponding test cases for the above use case.

```java
/**
 * Insert a new item at the right place in an (assumed sorted) list.
 * @param item The item to insert in the right place.
 */
public void insertSorted(int item) {
```

Test partitions for `insertSorted(int item)`:

T1: to the empty list

T2: to the non-empty list

T2-1: when item <= the first int

T2-2: when item > the first int

(a) item > the last int

(b) item == the last int

(c) item > an int in the middle (but item < the last int)

(d) item == an int in the middle (but item < the last int)

**Exercise #11 – Q4**