CSCI 4050/6050
Software Engineering

Software Testing

Dr. Eman Saleh eman.saleh@uga.edu

Verification and Validation

IEEE definitions:

Verification. The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process. Contrast with *verification*.

Validation. The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers. Contrast with *validation*.

Verification and Validation

Or, in other words:

Verification: Are we building the system correctly?

Validation: Are we building the correct system?

In this lecture, we will focus on verification.

IEEE Terminology

- Failure: Any deviation of the observed behavior from the specified behavior
- Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure
- Fault: The mechanical or algorithmic cause of an error (aka, a "bug")
- Verification: Activity of checking for deviations between the observed behavior of a system and its specification.

Examples of Faults and Errors

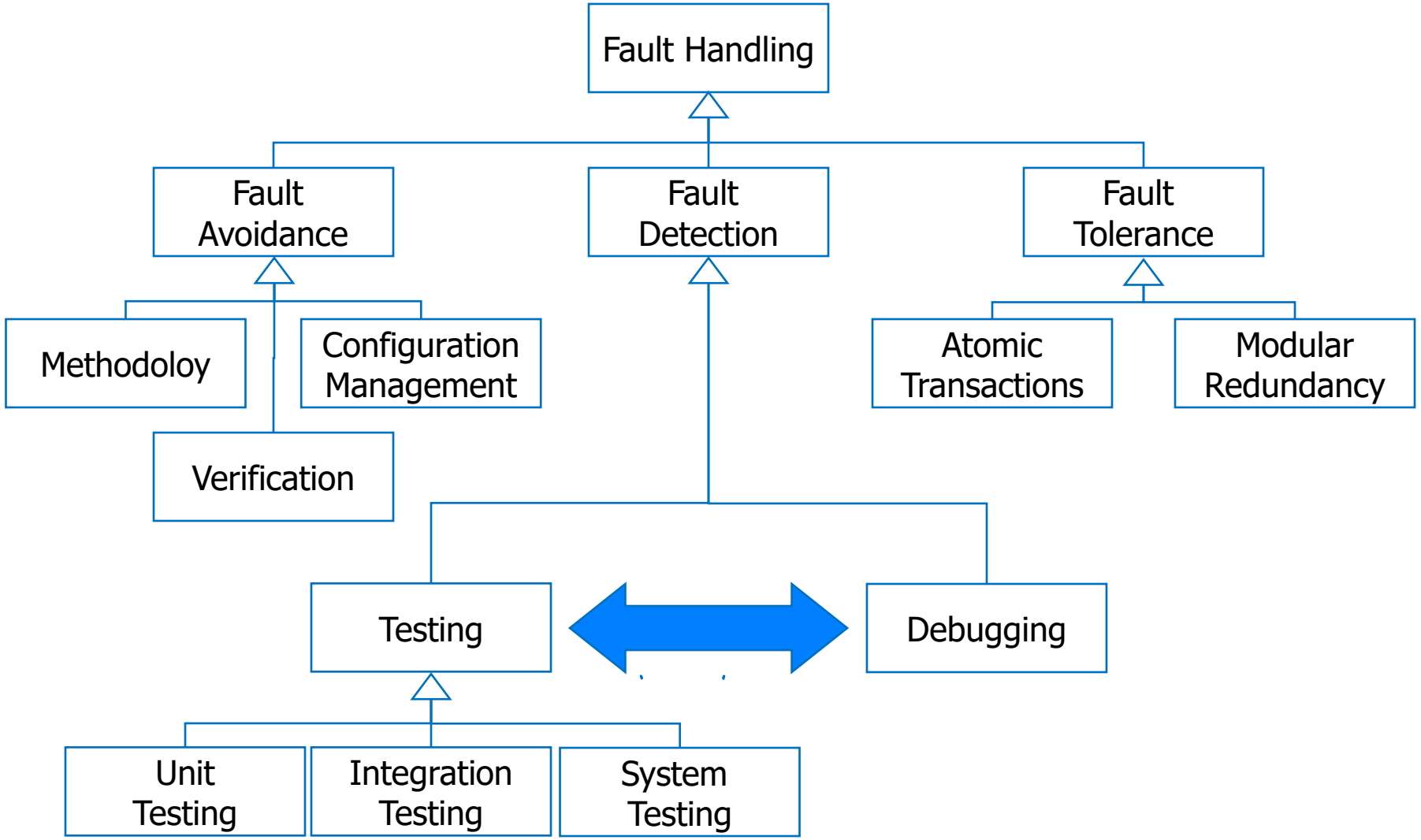
- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Null reference errors
  - Concurrency errors
  - Exceptions.

“How do we deal with Errors, Failures and Faults?”

How to Deal with Faults

- Fault avoidance
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Code Reviews and walkthroughs
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and eliminate the cause (Faults) of an observed failure
  - Monitoring: Deliver information about state => Used during debugging
- Fault tolerance
  - Exception handling
  - Modular redundancy.

Fault Handling Techniques



Observations

- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g., The Halting Problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra).
- Testing is not for free

=> Define your goals and priorities

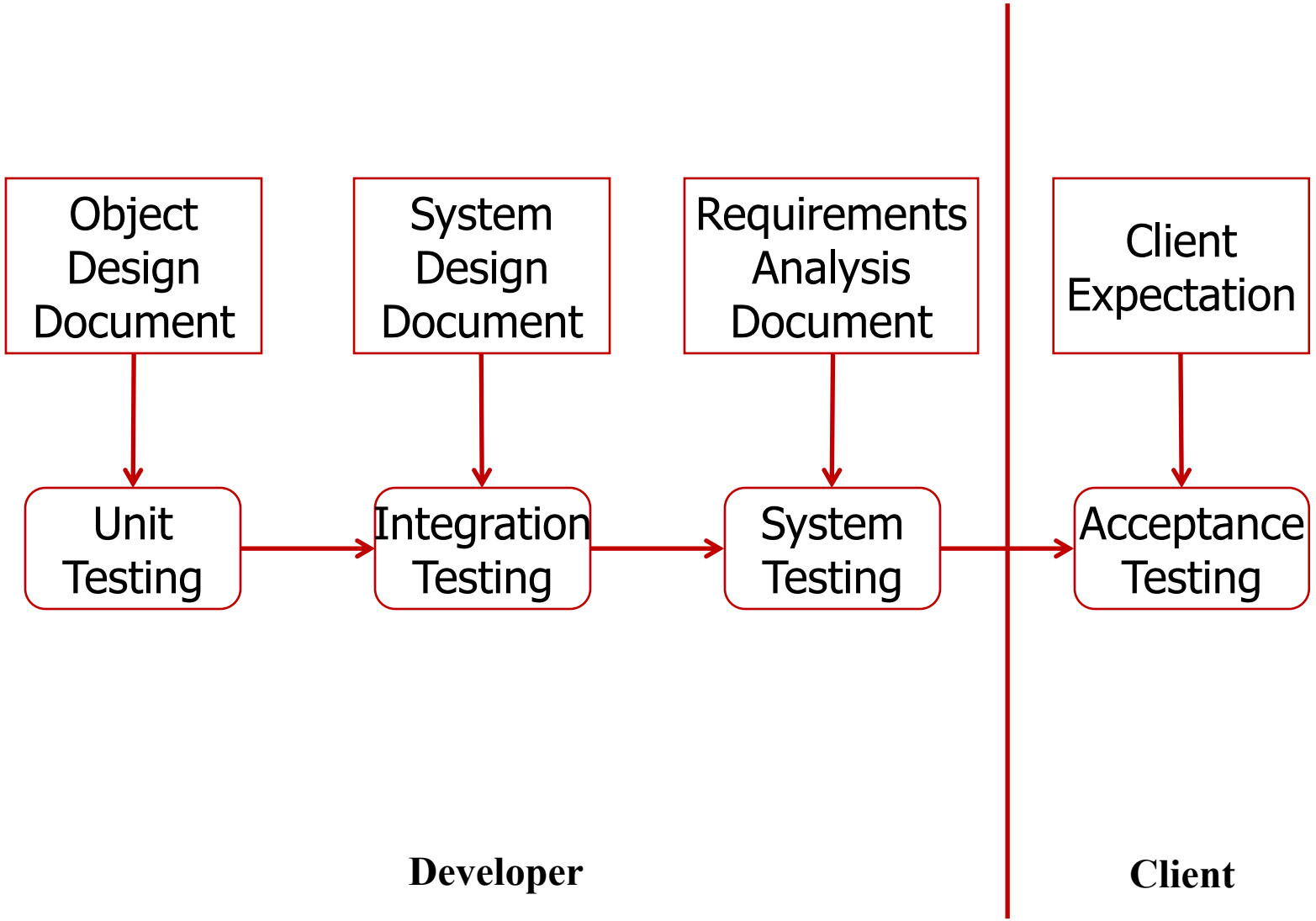
Testing Concepts

- Testing – the process of examining a component, subsystem, or system to determine its operational characteristics and whether it contains any defects
- Test case – a formal description of a starting state, one or more events to which the software must respond, and the expected response or ending state
  - Defined based on well understood functional and non-functional requirements
  - Must test all normal and exception situations
- Test data – a set of starting states and events used to test a module, group of modules, or entire system
  - The data that will be used for a test case

Testing takes creativity

- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

Testing Activities



Types of Testing

- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.

Types of Testing

- System Testing
  - The entire system
  - Carried out by developers
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

When should you write a test?

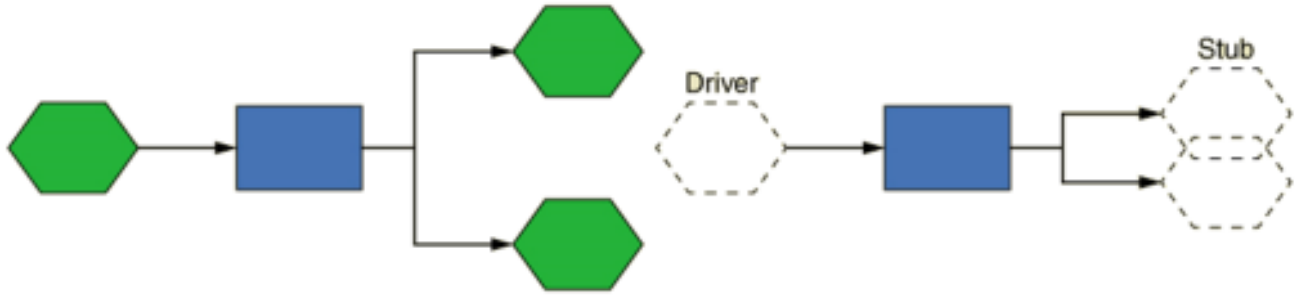
- Traditionally after the source code is written
- In Agile methodologies, even before the source code is written
  - Test-Driven Development Cycle
    - Add a test
    - Run the automated tests
      => see the new one fail
    - Write some code
    - Run the automated tests
      => see them succeed
    - Refactor code.

Most common types of tests

Test type	Core process	Need and purpose
Unit testing	Implementation	Software components must perform to the defined requirements and specifications when tested in isolation—for example, a component that incorrectly calculates sales tax amounts in different locations is unacceptable.
Integration testing	Implementation	Software components that perform correctly in isolation must also perform correctly when executed in combination with other components. They must communicate correctly with other components in the system. For example a sales tax component that calculates incorrectly when receiving money amounts in foreign currencies is unacceptable .
System and stress testing	Deployment	A system or subsystem must meet both functional and non-functional requirements. For example an item lookup function in a Sales subsystems retrieves data within 2 seconds when running in isolation, but requires 30 seconds when running within the complete system with a live database.
User acceptance testing	Deployment	Software must not only operate correctly, but must also satisfy the business need and meet all user "ease of use" and "completeness" requirements—for example, a commission system that fails to handle special promotions or a data-entry function with a poorly designed sequence of forms is unacceptable.

Unit Testing

- Unit test – tests of an individual method, class, or component before it is integrated with other software
- Driver – a method or class developed for unit testing that simulates the behavior of a method that sends a message to the method being tested
- Stub – a method or class developed for unit testing that simulates the behavior of a method invoked that hasn't yet been written



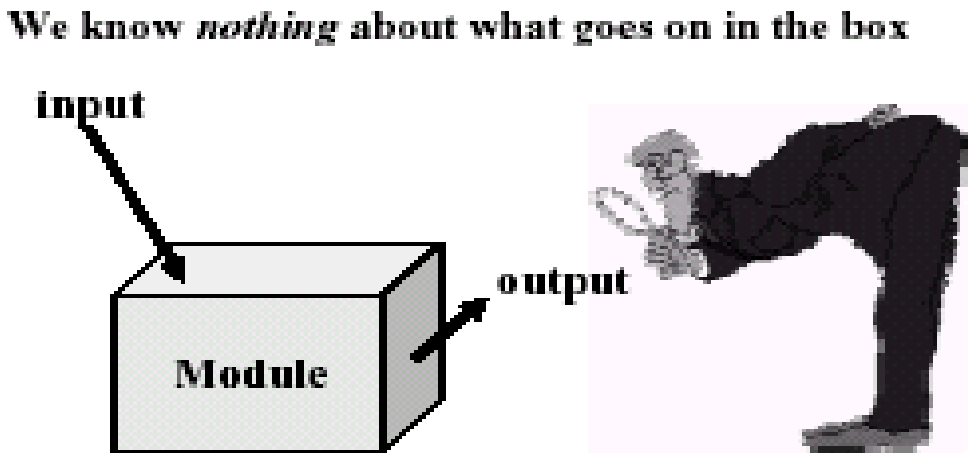
Unit Testing

- Static Testing (at compile time)
  - Static Analysis
  - Review
    - Walk-through (informal)
    - Code inspection (formal)
- Dynamic Testing (at run time)
  - Black-box testing
  - White-box testing.

Black-Box Testing

- Black-box testing is testing from a functional or behavioural perspective to ensure a program meets its specification
- Testing usually conducted without knowledge of software implementation (Code)
- system treated as a “black box”. Only know system inputs/outputs.

Black box testing



derives test cases from specifications

Complement to White-box test(detects different class of errors), Not alternative

Black-box testing: Test cases

a) Input is valid across range of values
- Developer selects test cases from 3 equivalence classes:
  - Below the range
  - Within the range
  - Above the range

b) Input is only valid, if it is a member of a discrete set
- Developer selects test cases from 2 equivalence classes:
  - Valid discrete values
  - Invalid discrete values

- No rules, only guidelines.

Black-box testing

- Focus: I/O behavior
  - If for any given input, we can predict the output, then the component passes the test
  - Requires test oracle
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide input conditions into equivalence classes
  - Choose test cases for each equivalence class.

Black Box Testing: Example

- A program should read two integer values and display the sum of the two integer values:

- Test cases:

1. Input two positive numbers: 90 8
2. Input two negative numbers: -4 -200
3. Input one negative input and one positive: 456 -30

White-box testing overview

- Code coverage
- Branch coverage
- Condition coverage
- Path coverage

Unit Testing Heuristics

1. Create unit tests when object design is completed
   - Black-box test: Test the functional model
   - White-box test: Test the dynamic model
2. Develop the test cases
   - Goal: Find effective number of test cases
3. Cross-check the test cases to eliminate duplicates
   - Don't waste your time!

4. Desk check your source code
   - Sometimes reduces testing time
5. Create a test harness
   - Test drivers and test stubs are needed for integration testing
6. Describe the test oracle
   - Often the result of the first successfully executed test
7. Execute the test cases
   - Re-execute test whenever a change is made ("regression testing")
8. Compare the results of the test with the test oracle
   - Automate this if possible.

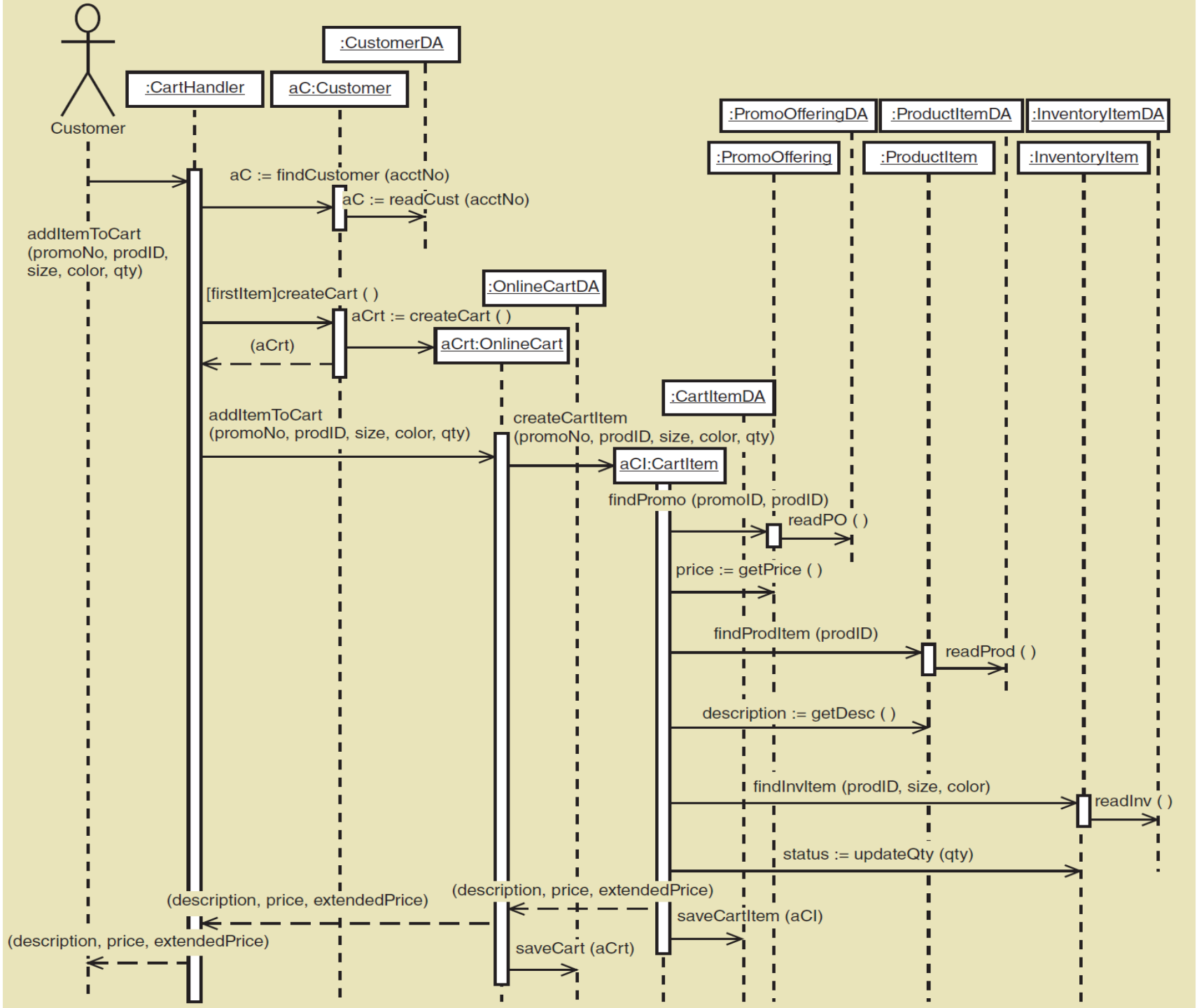
White Box test: Example

```
read(x);
read(y) ;
if x > 0 then
   write("1") ;
else
   write("2") ;
end if;
if y > 0 then
   write("3") ;
else
   write("4") ;
end if ;
```

Test Criterion: Cover all conditions.

We might choose the following test set, which cover the criterion:

{<x = - 13, y= 51>,
<x = 2, y = -3>}


Customer
:CustomerDA
:CartHandler
aC:Customer
:PromoOfferingDA
:ProductItemDA
:InventoryItemDA
:PromoOffering
:ProductItem
:InventoryItem
aC := findCustomer (acctNo)
aC := readCust (acctNo)
addItemToCart
(promoNo, prodID,
size, color, qty)
[firstItem]createCart ( )
:OnlineCartDA
aCrt := createCart ( )
(aCrt)
aCrt:OnlineCart
:CartItemDA
addItemToCart
(promoNo, prodID, size, color, qty)
createCartItem
(promoNo, prodID, size, color, qty)
aCI:CartItem
findPromo (promoID, prodID)
readPO ( )
price := getPrice ( )
findProdItem (prodID)
readProd ( )
description := getDesc ( )
findInvItem (prodID, size, color)
readInv ( )
status := updateQty (qty)
(description, price, extendedPrice)
(description, price, extendedPrice)
saveCartItem (aCI)
saveCart (aCrt)

Simple driver to test *createCartItem()*

```
main()
{
 // driver method to test CartItem::createCartItem()

 // declare input parameters and values

 int promoID = 23;
 int prodID = 1244;
 String size = "large";
 String color = "red";
 int quantity = 1;

 // perform test

 cartItem cartItem = new cartItem();
 cartItem.createcartItem(promoID,prodID,size,color,quantity);

 // display results

 System.out.println("price=" + cartItem.getPrice());
 System.out.println("description=" + cartItem.getDescription();
 System.out.println("status=" + cartItem.getStatus());
} // end main()
```

JUnit: A recap

- **JUnit is covered in earlier courses (e.g., CSCI 1302)**
- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Written by Kent Beck and Erich Gamma
- Written with “test first” and pattern-based development in mind
  - Tests written along with the code, or even before
  - Allows for regression testing
  - Facilitates refactoring
- JUnit is Open Source
  - www.junit.org

Integration Testing

Integration Testing

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design

- Goal: Test all interfaces between subsystems and the interaction of subsystems

Why integration testing?

- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration testing will be discovered after the system is deployed and can be very expensive.

Integration Testing

- Integration test – tests of the behavior of a group of methods, classes, or components
  - Interface incompatibility—For example, one method passes a parameter of the wrong data type to another method
  - Parameter values—A method passes or returns a value that was unexpected, such as a negative number for a price.
  - Run-time exceptions—A method generates an error, such as "out of memory" or "file already in use," due to conflicting resource needs
  - Unexpected state interactions—The states of two or more objects interact to cause complex failures, as when an OnlineCart class method operates correctly for all possible Customer object states except one

Integration Testing

- Integration testing of object-oriented software is very complex because an object-oriented program consists of a set of interacting objects
  - Methods can be (and usually are) called by many other methods, and the calling methods may be distributed across many classes.
  - Classes may inherit methods and state variables from other classes.
  - The specific method to be called is dynamically determined at run time based on the number and type of message parameters.
  - Objects can retain internal variable values (i.e., the object state) between calls. The response to two identical calls may be different due to state changes that result from the first call or occur between calls.

Integration Testing

- Required Procedures
  - Build and unit test the components to be integrated
  - Create test data – comprehensive test data, must be coordinated between developers
  - Conduct the integration test – Assign resources and responsibilities. Plan frequency and procedures
  - Evaluate the test results – Identify valid and invalid responses
  - Log the test results – Log valid test runs. Also log errors
  - Correct the code and retest

	A	B	C	D	E	F	G	H	I	J
1	Error #	Component	Date test run	Error description	Severity	Priority	Assigned to	Target date	Fixed date	Resolution comment
2	1001	Order entry	11/15/2015	Wrong dollar result	4-Important	4-High	Mary Ann Holmes	11/20/2015	11/20/2105	Equation error
3	1002	Order entry	11/15/2015	System crashed	5-Serious	5-Urgent	Jack Holdaway	11/20/2015	11/19/2015	Passing wrong data type
4										
5										

Integration Testing Strategies

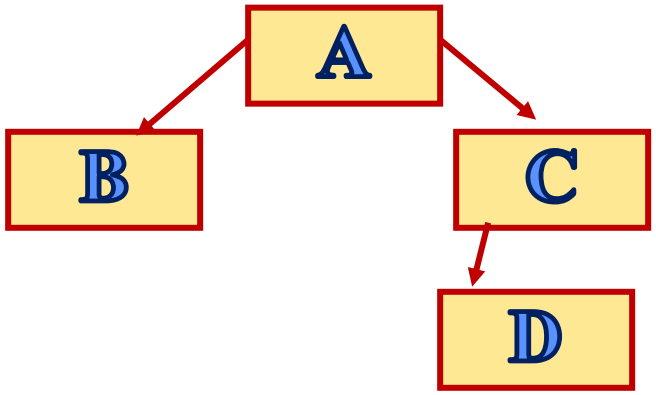
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Bottom-up Testing Strategy

- 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 repeated until all subsystems are included
- Drivers are needed.

Bottom up integration testing example

- Example: A calls routines in B and C, routines in C call routines in D
- Write drivers for B and D, test separately
- Write driver for C, compile D + C + C's drivers, test
- Compile A + B + C + D, test



Bottom-Up Integration Testing

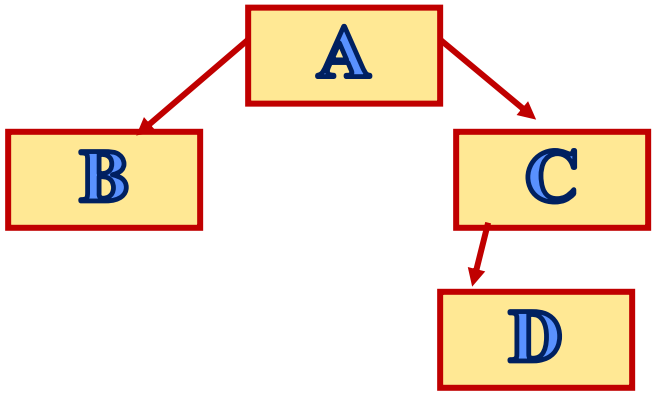
- Cons:
  - Tests the most important subsystem (user interface) last
  - Drivers needed
  - As subtrees are combined, a large number of elements may be integrated at one time.
- Pros
  - Allows early verification of low-level behavior.
  - No stubs are required.
  - Easier to formulate input data for some subtrees;
  - Easier to interpret output data for others.
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements.

Top-down Testing Strategy

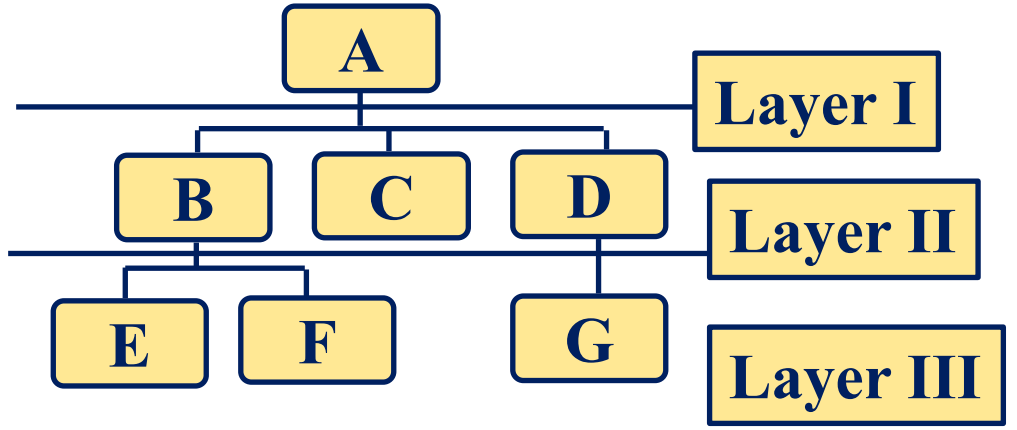
- Test the top layer or the controlling subsystem first
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Stubs are needed to do the testing.

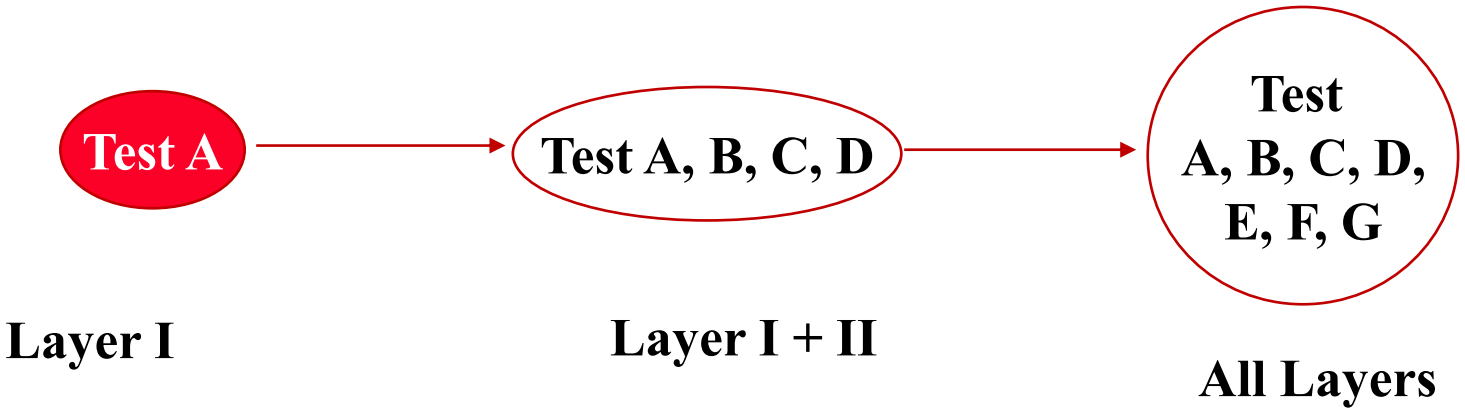
Top down integration testing example

- Have stubs for the routines in B and C called by A
- Compile A + stubs for B + stubs for C, test
- Replace B stubs by actual B, compile, test
- Have stubs for the routines in D called by C
- Compile A + B + C + stubs for D, test
- Replace D stubs by actual D, compile, test



Top-down Integration Testing





Top-down Integration Testing

Pros

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

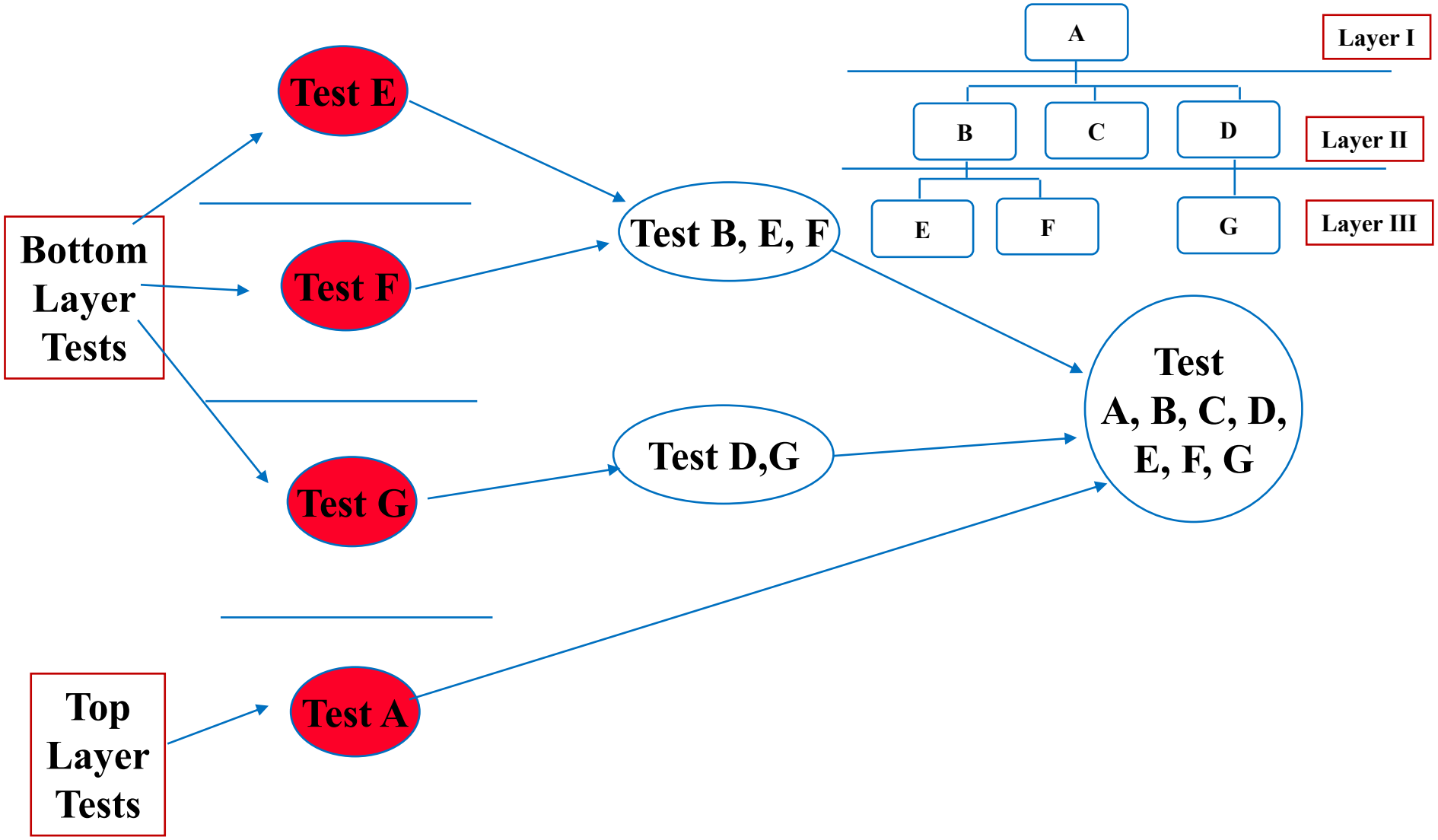
Cons

- Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
- Large number of stubs may be required, especially if the lowest level of the system contains many methods.
- Some interfaces are not tested separately.

Sandwich Testing Strategy

- Combines top-down strategy with bottom-up strategy
- The system is viewed 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.

Sandwich Testing Strategy



Sandwich Testing

- Top and Bottom Layer Tests can be done in parallel
- Problem: Does not test the individual subsystems and their interfaces thoroughly before integration

Continuous Testing

- Continuous build:
  - Build from day one
  - Test from day one
  - Integrate from day one

  $\Rightarrow$ System is always runnable

- Requires integrated tool support:
  - Continuous build server
  - Automated tests with high coverage
  - Tool supported refactoring
  - Software configuration management
  - Issue tracking.

Steps in Integration Testing

1. Based on the integration strategy, *select a component* to be tested. Unit test all the classes in the component.
2. Put selected component together; do any *preliminary fix-up* necessary to make the integration test operational (drivers, stubs)
3. Test functional requirements*:* Define test cases that exercise all uses cases with the selected component
4. Test subsystem decomposition*:* Define test cases that exercise all dependencies
5. Test non-functional requirements: Execute *performance tests*
6. *Keep records* of the test cases and testing activities.
7. Repeat steps 1 to 7 until the full system is tested.

The primary *goal of integration testing is to identify failures* with the (current) component *configuration*.

Remember!

- The Final Exam is on 7/30/2020 03:30- 06:30pm

System Testing

- Functional Testing
  - Verifies functional requirements
- Performance Testing
  - Verifies non-functional requirements
- Acceptance Testing
  - Verifies clients expectations

Functional Testing

Goal: Test functionality of system

- Test cases are designed from the requirements analysis document (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as black box
- Unit test cases can be reused, but new test cases have to be developed as well.

Performance Testing

Goal: Try to violate non-functional requirements

- Test how the system behaves when overloaded.
  - Can bottlenecks be identified? (First candidates for redesign in the next iteration)
- 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.
- What is the amount of time spent in different use cases?
  - Are typical cases executed in a timely fashion?

Types of Performance/ stress Testing

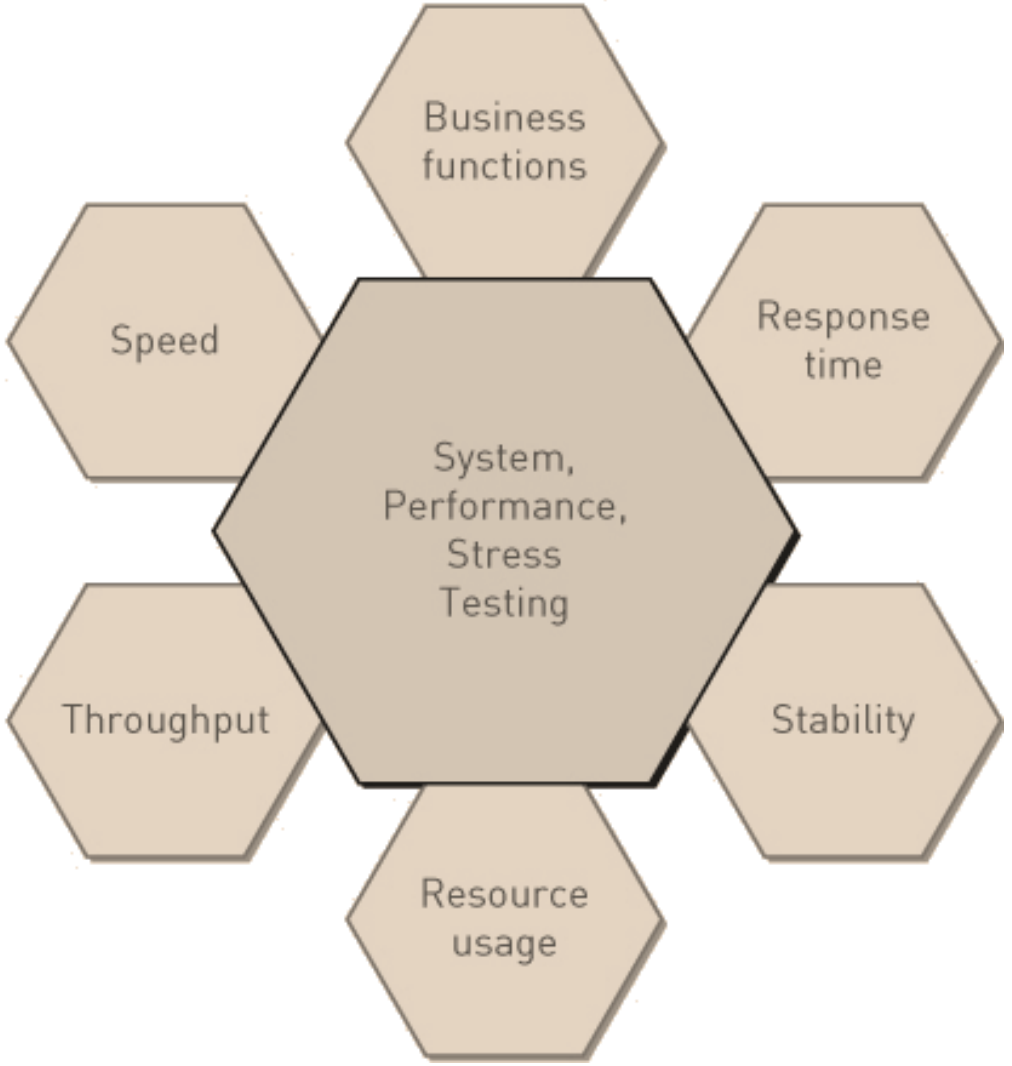
Stress Testing	**Stress limits of system**
Volume testing	**Test what happens if large amounts of data are handled**
Configuration testing	**Test the various software and hardware configurations**
Compatibility test	**Test backward compatibility with existing systems**
Timing testing	**Evaluate response times and time to perform a function**

- Security testing
  - Try to violate security requirements
- Environmental test
  - Test tolerances for heat, humidity, motion
- Quality testing
  - Test reliability, maintain- ability & availability
- Recovery testing
  - Test system's response to presence of errors or loss of data
- Human factors testing
  - Test with end users.

Performance Testing

- Performance test or stress test – an integration and usability test that determines whether a system or subsystem can meet time-based performance criteria
  - Response time – the desired or maximum allowable time limit for software response to a query or update
  - Throughput – the desired or minimum number of queries and transactions that must be processed per minute or hour

System, Performance, and Stress Testing



Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.

- Alpha test:
  - Client uses the software at the developer's environment.
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- Beta test:
  - Conducted at client's environment (developer is not present)
  - Software gets a realistic workout in target environ- ment

Testing has many activities

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

User Acceptance Testing (UAT)

- Plan the UAT
  - Should be done early in the project
  - Test cases for every use case and user stories
  - Identify conditions to verify that the system supports the use case accurately and completely
- Sample test case list

	A	B	C	D	E	F
1	Spec ID	Cross refer to use case	Short description	Test conditions	Expected outcomes	Comments
2	10	101	Maintain customer Info	Add customer, update customer, delete not allowed	New customer with all fields, updated customer with selected fields	
3	11	201	Maintain sale info	Create sale, update sale, finalize sale, pay for sale	New sale in DB, update selected fields, payment creates transaction	
4	12	202	Ship items	Display items, update status	Sale update, sale items updated, shipment created	

User Acceptance Testing (UAT)

- Preparation and Pre-UAT Activities
  - Develop test data – data entry and database records
  - Plan and schedule specific tests
  - Set up test environment

User Acceptance Testing (UAT)

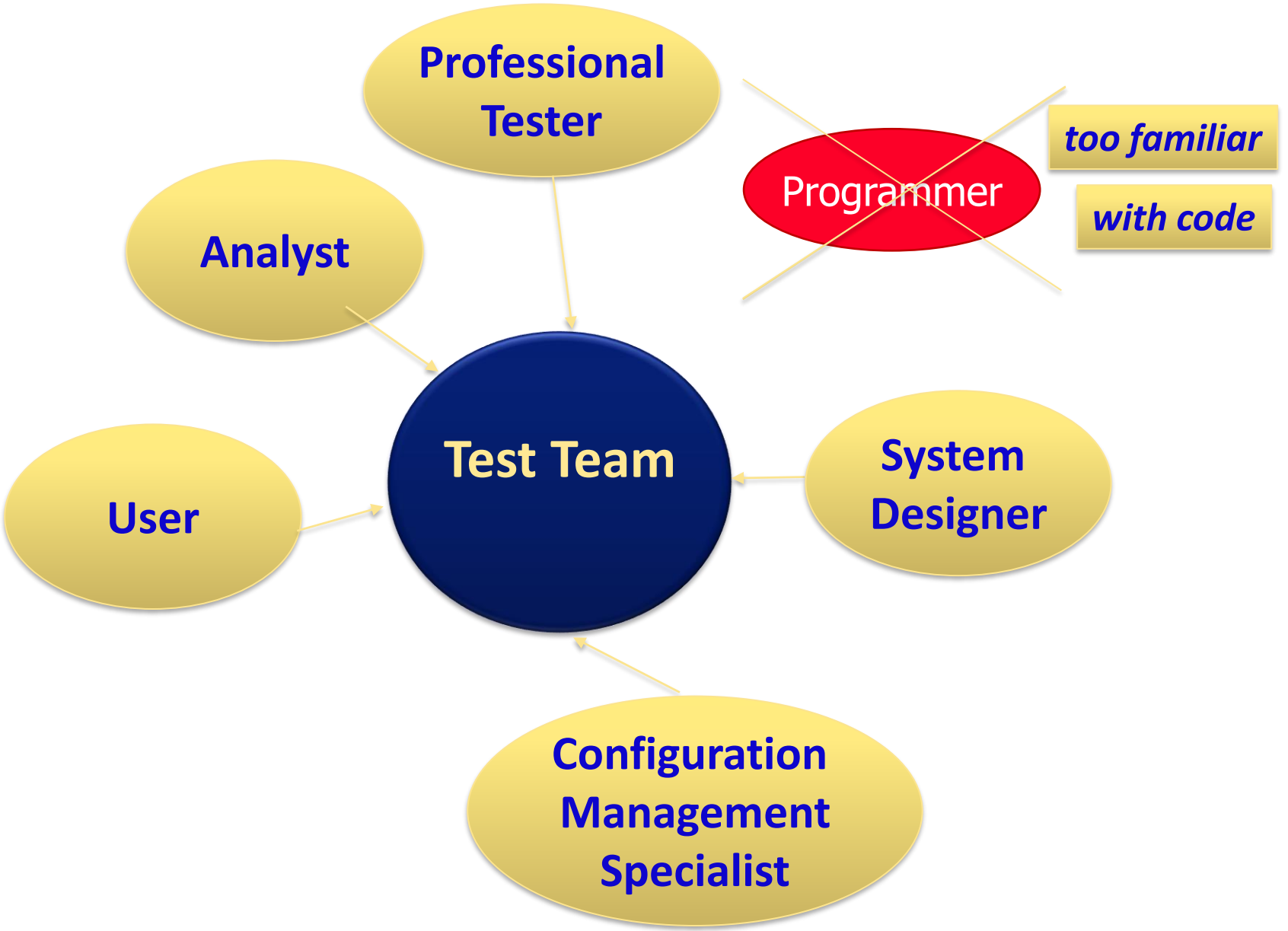
- Manage and execute the UAT
  - Much like a mini-project
  - Assign responsibilities
  - Document and track results (especially errors and fixes)
  - Rework the plan for re-testing as required

User Acceptance Testing (UAT)

- Log and Results tracking list

	A	B	C	D	E	F	G	H	I	J
1	Spec ID	Cross refer to use case	Short description	Test condition	Expected outcomes	Name of tester	Date executed	Acceptance criteria	Status	Outstanding issues
2	10	101	Maintain customer info	Add customer, update customer, delete not allowed	New customer with all fields, updated customer with selected fields	Mary Helper	7/15/2015	All expected outcomes, DB updated successfully	Accepted	None
3	11	201	Maintain sale info	Create sale, upate sale, finalize sale, pay for sale	New sale in DB, update selected fields, payment creates transaction	Mary Helper	7/15/2015	All expected outcomes, DB updated successfully	Pending	1005, 1006
4	12	202	Ship items	Display items, update status	Sale update, sale items updated, shipment created				Not started	

Test Team


Professional Tester
Programmer
too familiar
with code
Analyst
Test Team
User
System Designer
Configuration Management Specialist

The 4 Testing Steps

1. Select what has to be tested
   - Analysis: Completeness of requirements
   - Design: Cohesion
   - Implementation: Source code
2. Decide how the testing is done
   - Review or code inspection
   - Proofs (Design by Contract)
   - Black-box, white box,
   - Select integration testing strategy (big bang, bottom up, top down, sandwich)

3. Develop test cases
   - A test case is a set of test data or situations that will be used to exercise the unit (class, subsystem, system) being tested or about the attribute being measured
4. Create the test oracle
   - An oracle contains the predicted results for a set of test cases
   - The test oracle has to be written down before the actual testing takes place.

Guidance for Test Case Selection

- Use *analysis knowledge* about functional requirements (black-box testing):
  - Use cases
  - Expected input data
  - Invalid input data
- Use *design knowledge* about system structure, algorithms, data structures (white-box testing):
  - Control structures
    - Test branches, loops, ...
  - Data structures
    - Test records fields, arrays, ...

- Use *implementation knowledge* about algorithms and datastructures:
  - Force a division by zero
  - If the upper bound of an array is 10, then use 11 as index.

Summary

- Testing is still a black art, but many rules and heuristics are available
- Testing consists of
  - Unit testing
  - Integration testing
  - System testing
    - Acceptance testing
- Design patterns can be used for integration testing
- Testing has its own lifecycle