Software Testing: Integration Testing

Overview – Integration Testing

- Integration testing
  - Big bang
  - Bottom up
  - Top down
  - Sandwich
  - Continuous

Integration Testing

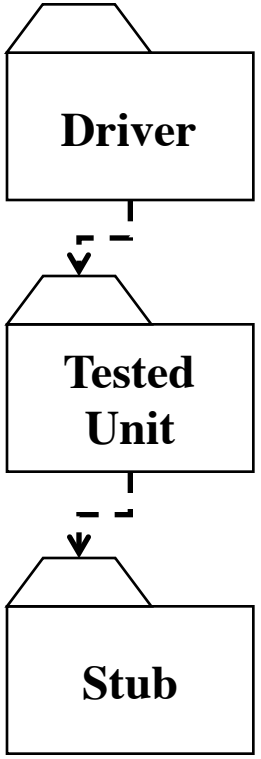
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The Integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

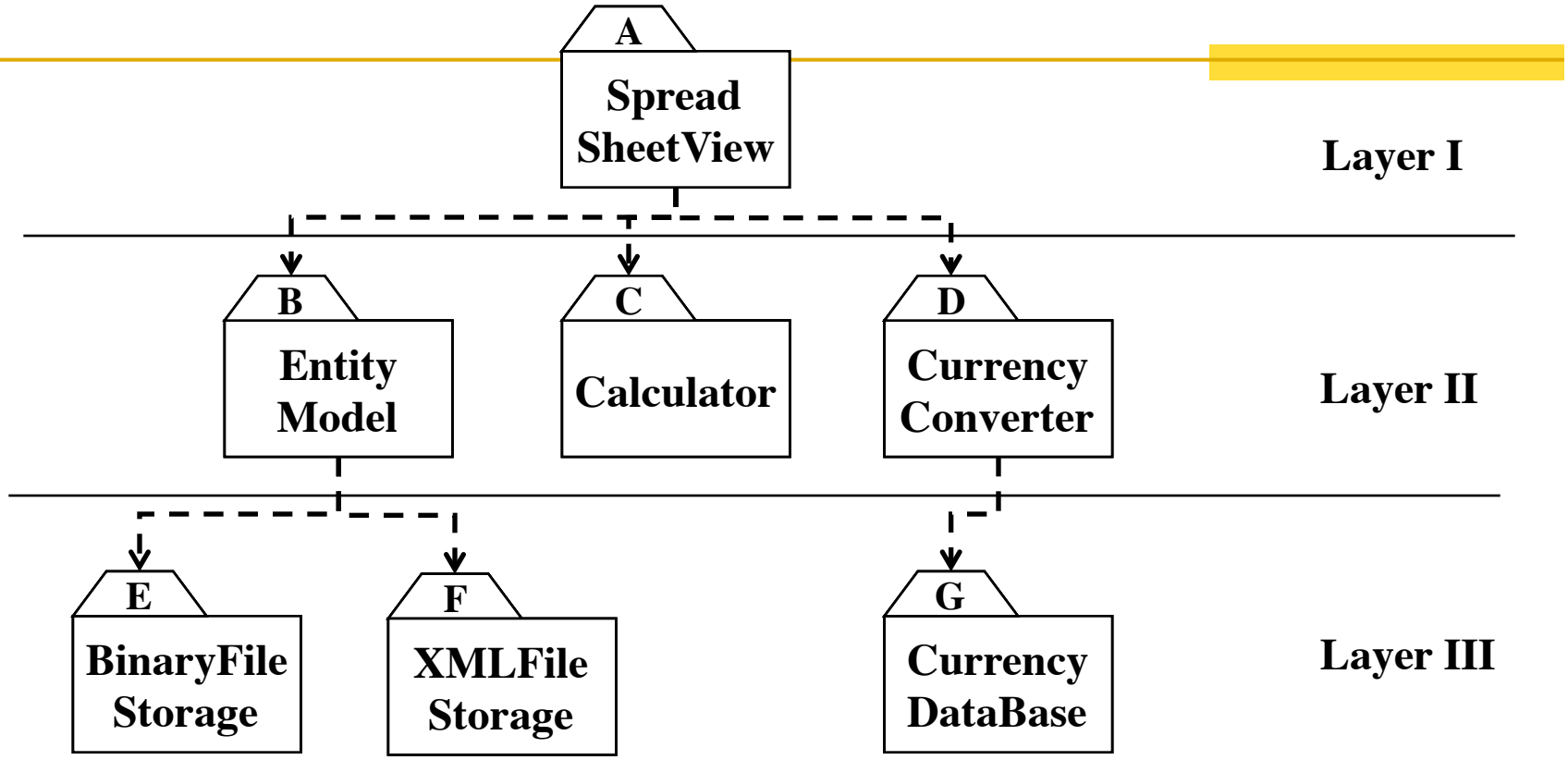
- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration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

Stubs and drivers

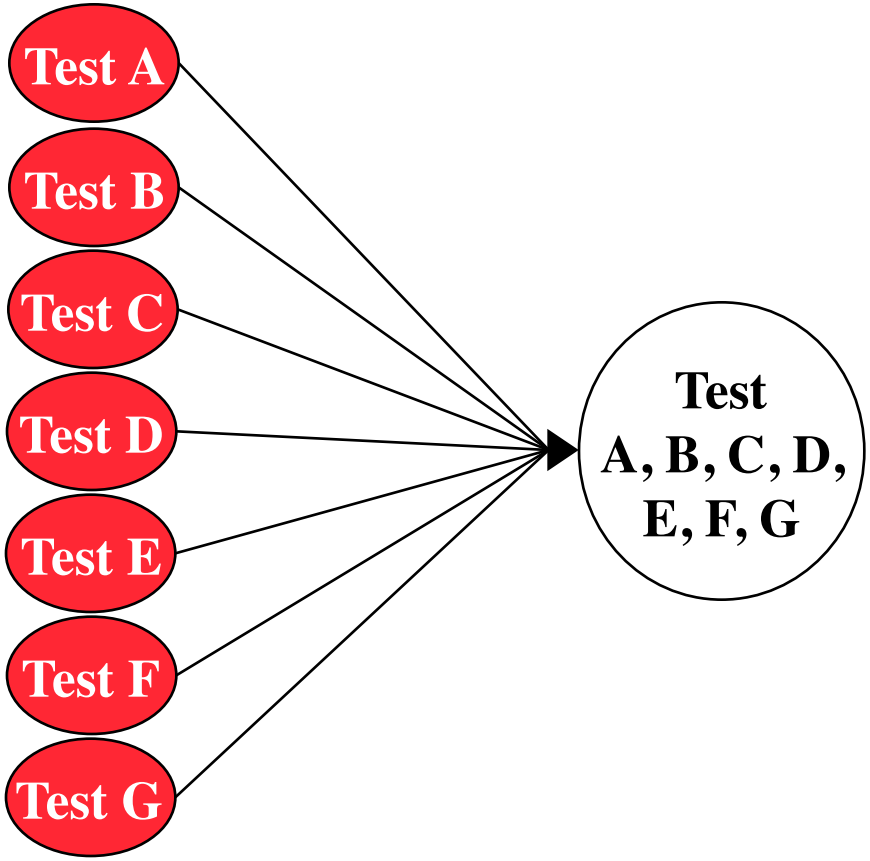
- Driver:
  - A component, that calls the `TestedUnit`
  - Controls the test cases
- Stub:
  - A component, the `TestedUnit` depends on
  - Partial implementation
  - Returns fake values.

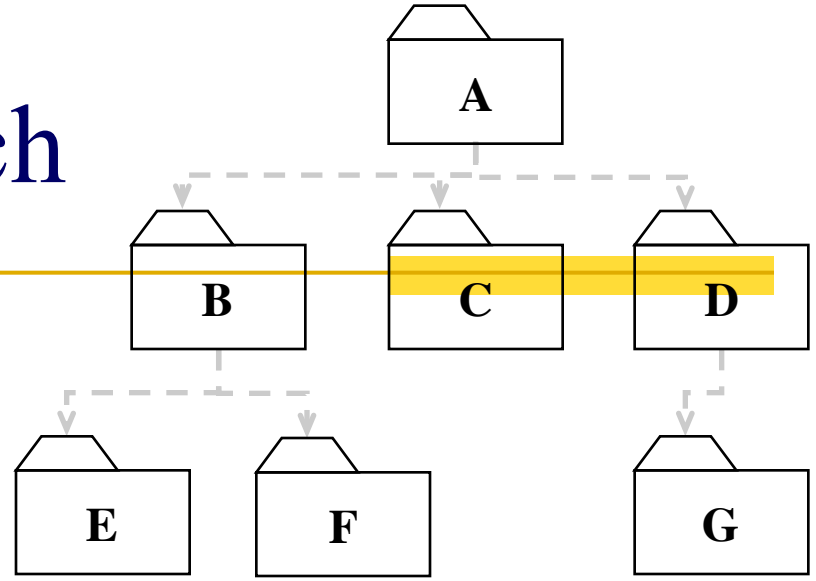


Example: A 3-Layer-Design (Spreadsheet)



Big-Bang Approach

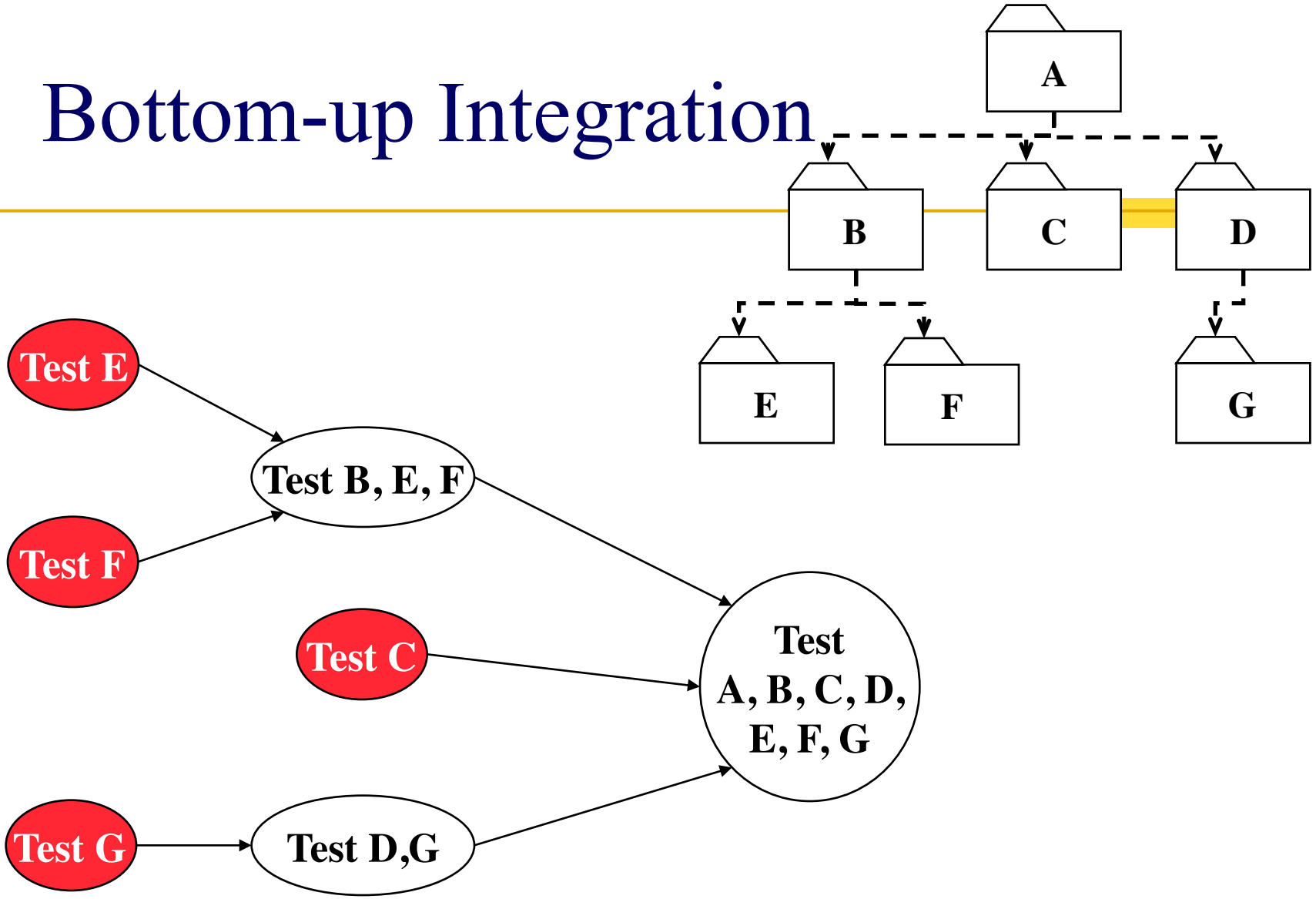




Bottom-up Testing Strategy

- 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



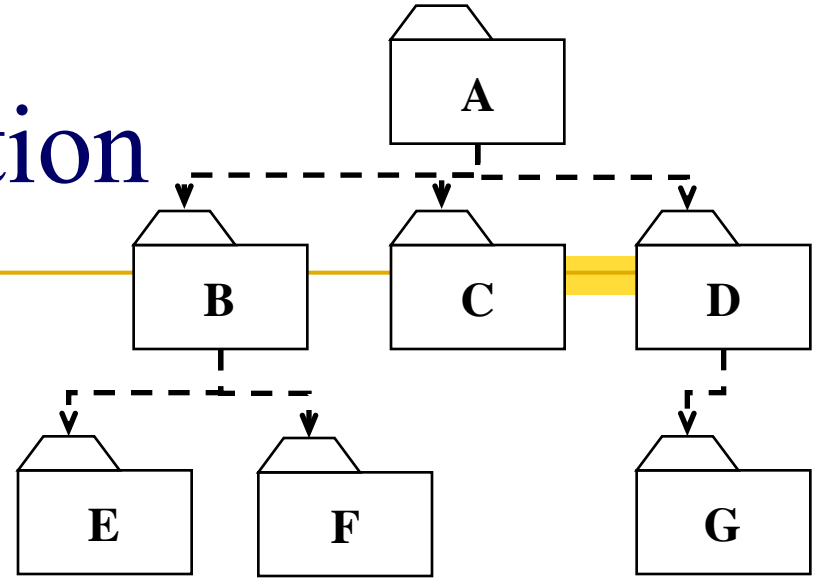
Pros and Cons of Bottom-Up Integration Testing

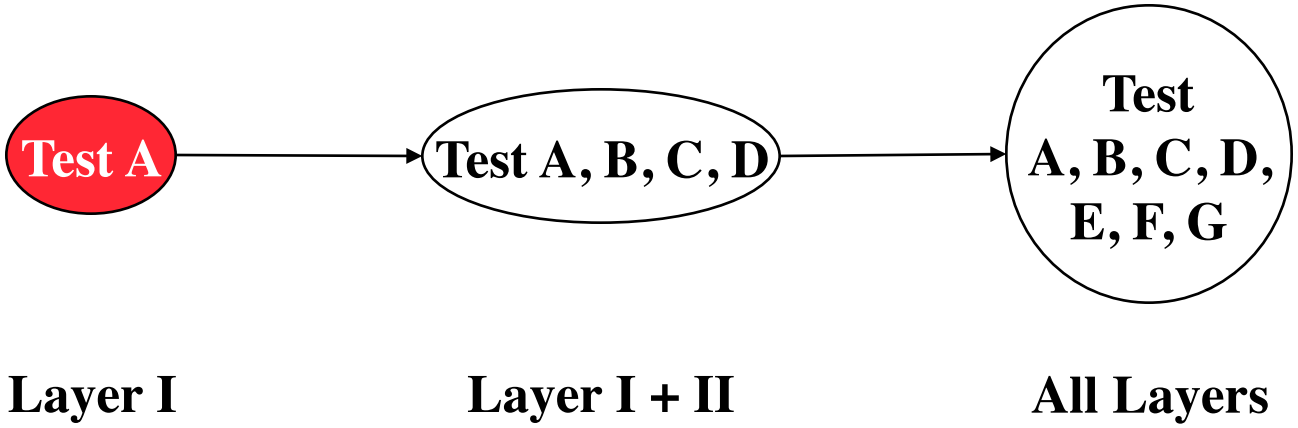
- Con:
  - Tests the most important subsystem (user interface) last
  - Drivers needed
- Pro
  - No stubs needed
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements.

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





Pros and Cons of Top-down Integration Testing

Pro

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

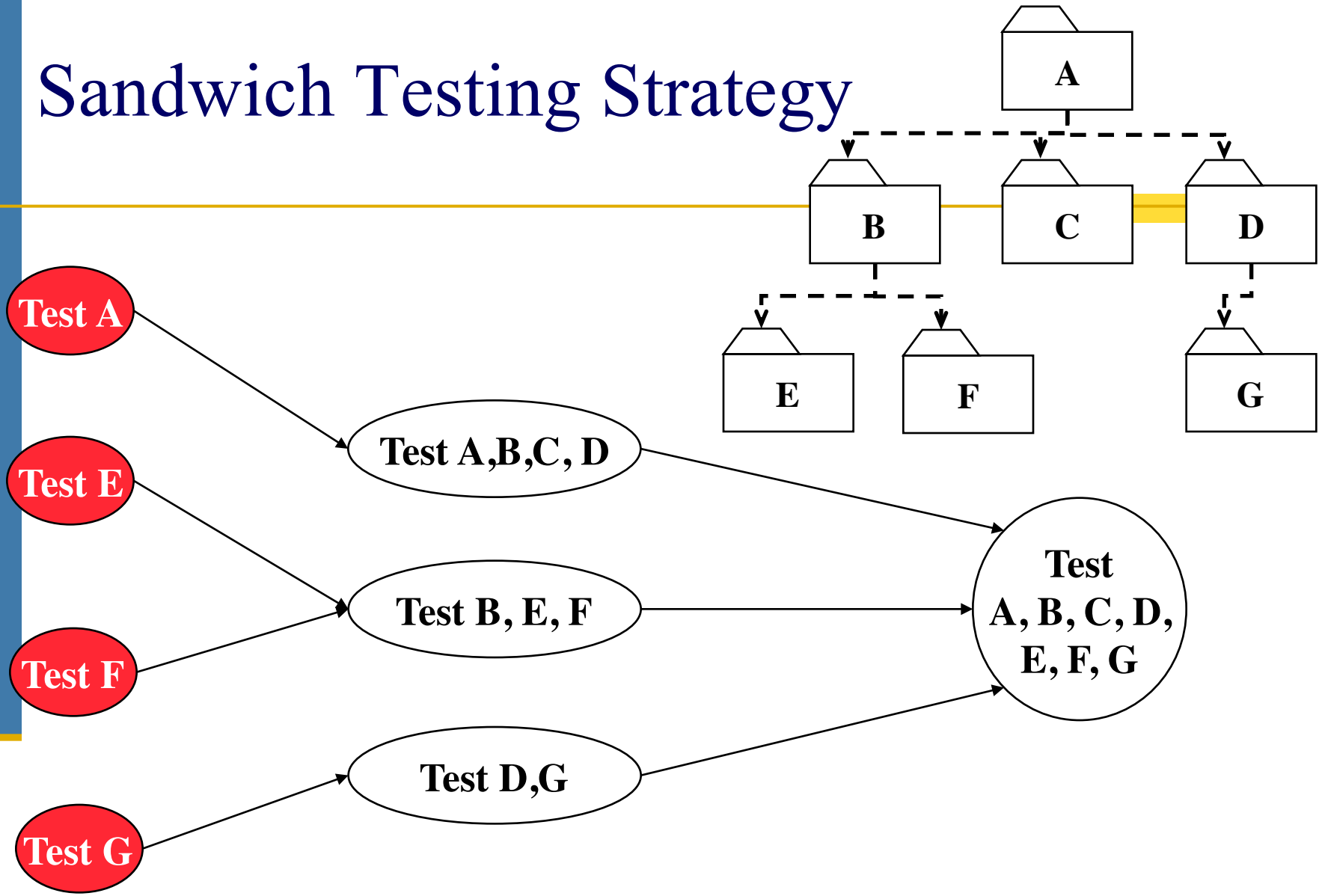
Cons

- 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



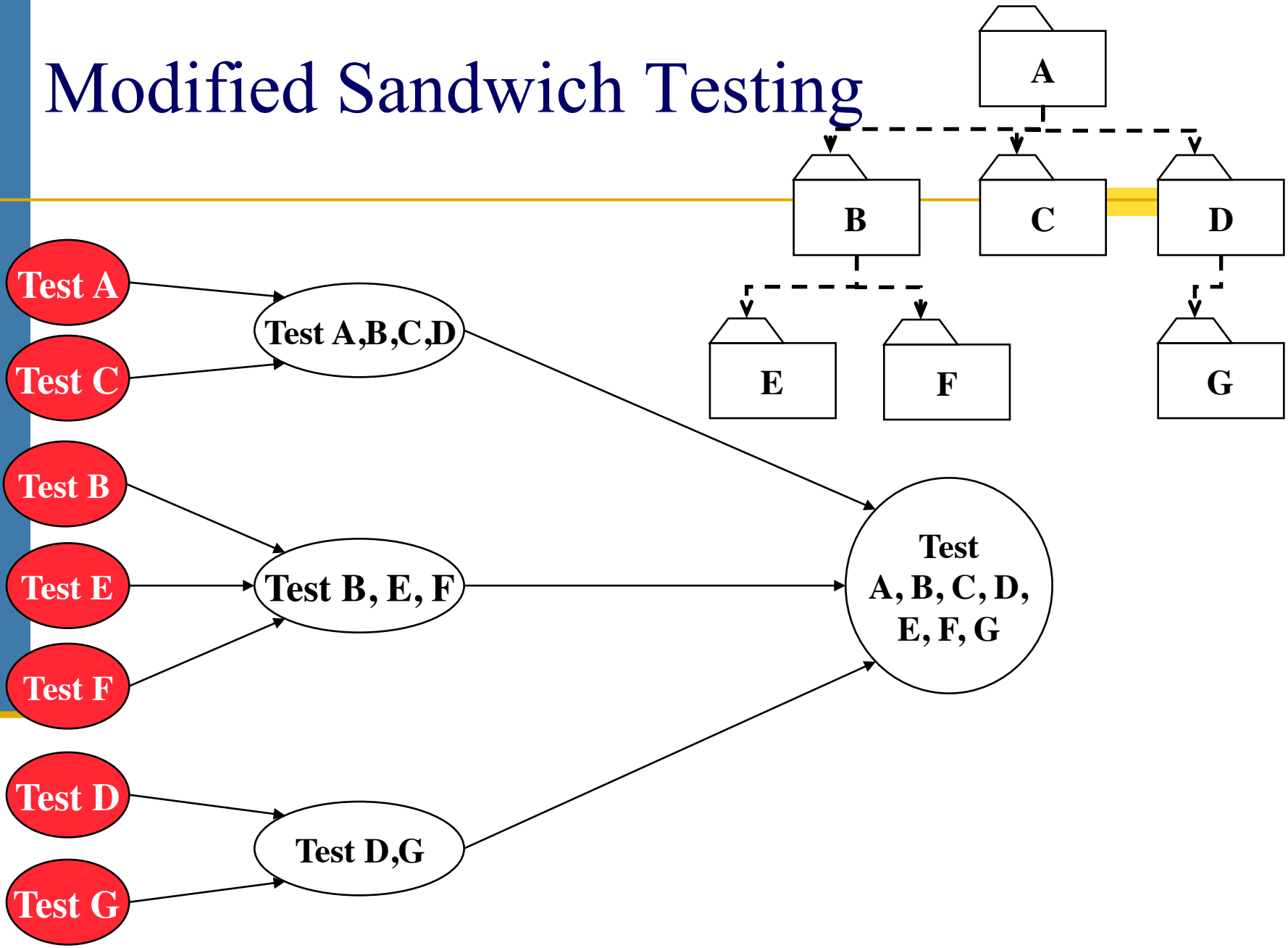
Pros and Cons of Sandwich Testing

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

Modified Sandwich Testing Strategy

- Test in parallel:
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers
- Test in parallel:
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs).

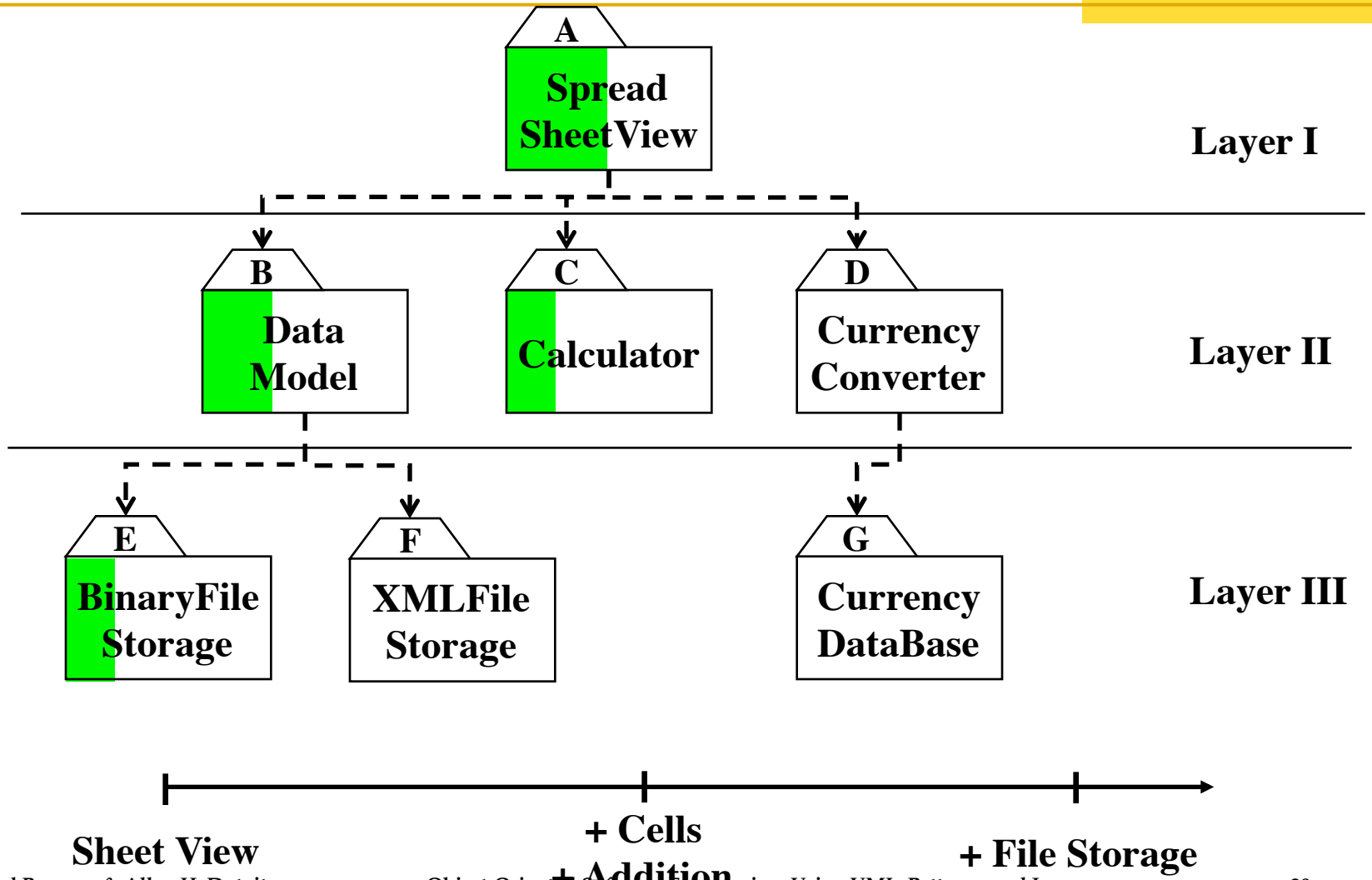
Modified Sandwich Testing



Continuous Testing

- Continuous build:
  - Build from day one
  - Test from day one
  - Integrate from day one
  - ⇒ System is always runnable
- Requires integrated tool support:
  - Continuous build server
  - Automated tests with high coverage
  - Tool supported refactoring
  - Software configuration management
  - Issue tracking

Continuous Testing Strategy



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*.