Integration Testing

Lecture 22

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.

What is integration testing?
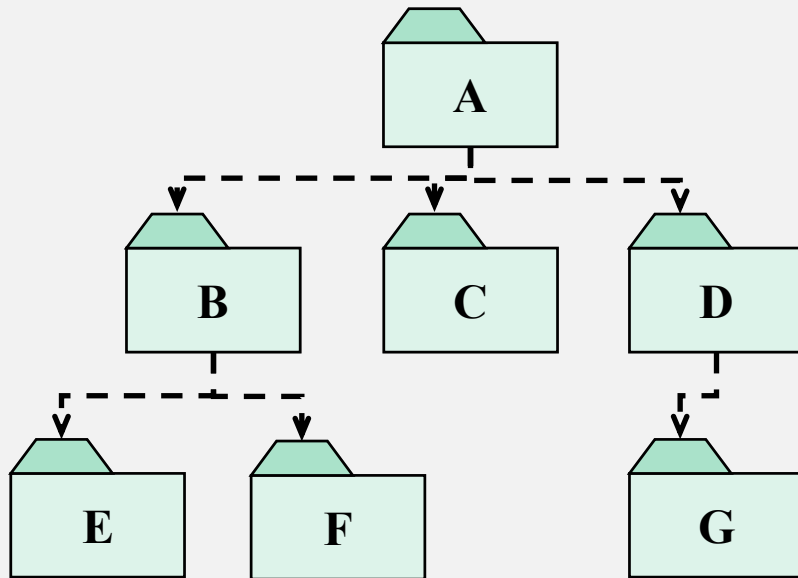
Integration testing is the process of verifying the interactions among software modules.

Classical integration testing strategies, such as top-down and bottom-up, are often used with hierarchically structured software to facilitate **error localization**.

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.
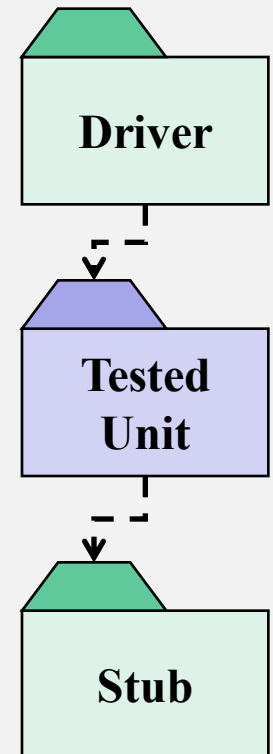
Stubs and drivers

Driver:

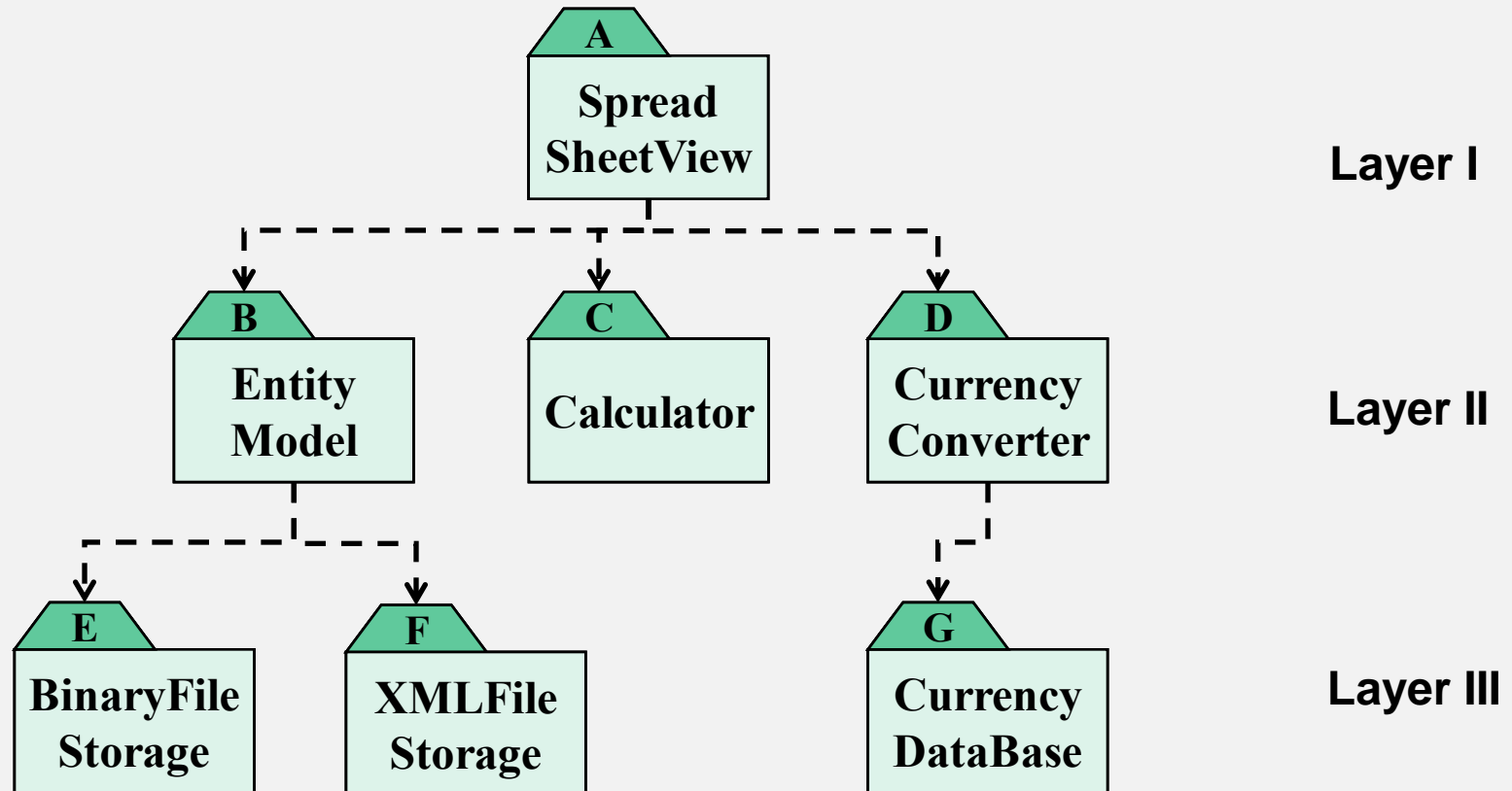- A component, that calls the `TestedUnit`
- Controls the test cases

Stub:

A component, the `TestedUnit`
depends on

Partial implementation

Returns fake values.

Example: A 3-Layer-Design

A — Spread SheetView — Layer I

B — Entity Model — **C** — Calculator — **D** — Currency Converter — Layer II

E — BinaryFile Storage — **F** — XMLFile Storage — **G** — Currency DataBase — Layer III

6

Bottom-up Testing Strategy

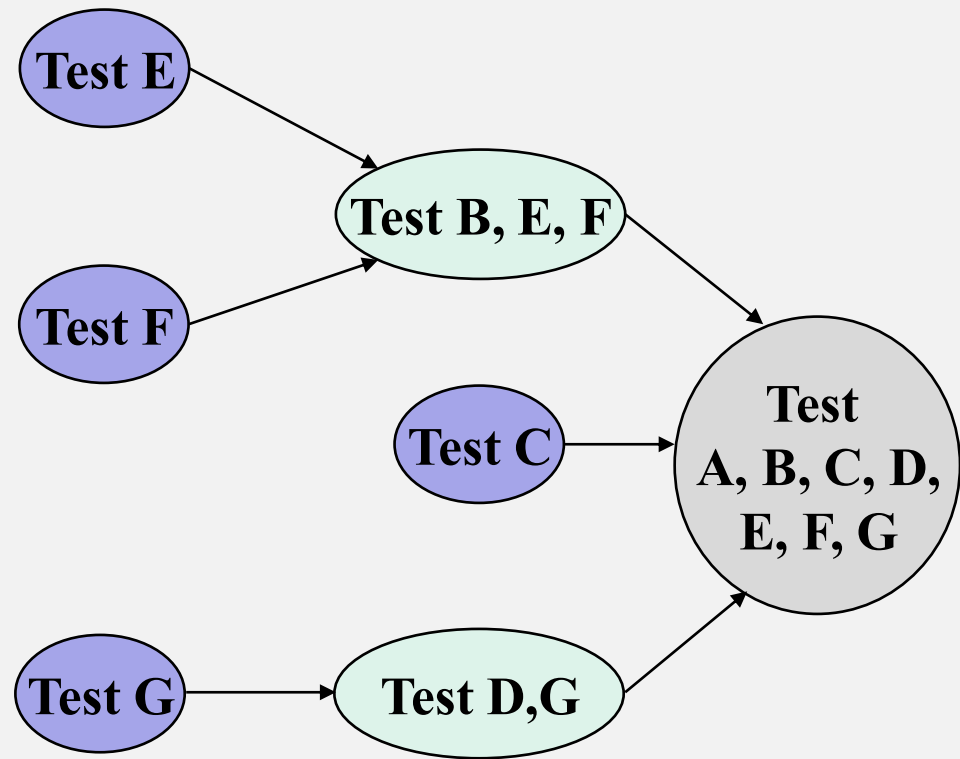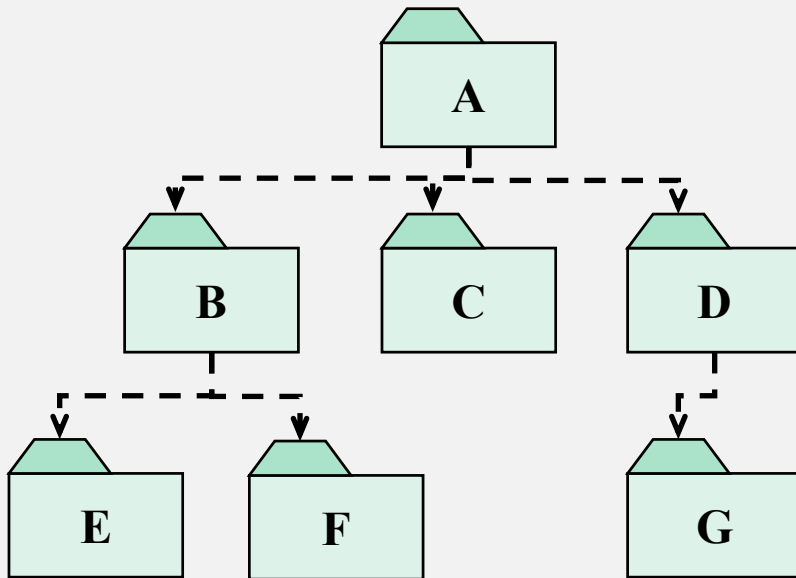The subsystems in the lowest layer of the call hierarchy are tested individually

Then the next subsystems are tested that call the previously tested subsystems

This is repeated until all subsystems are included.

What is integration testing?

Bottom up strategy

Pros and Cons of Bottom-Up Integration Testing

Pro

No stubs needed

Useful for integration testing of the following systems

Object-oriented systems

Real-time systems

Systems with strict performance requirements.

Con:

Tests the most important subsystem (user interface) last

Drivers needed
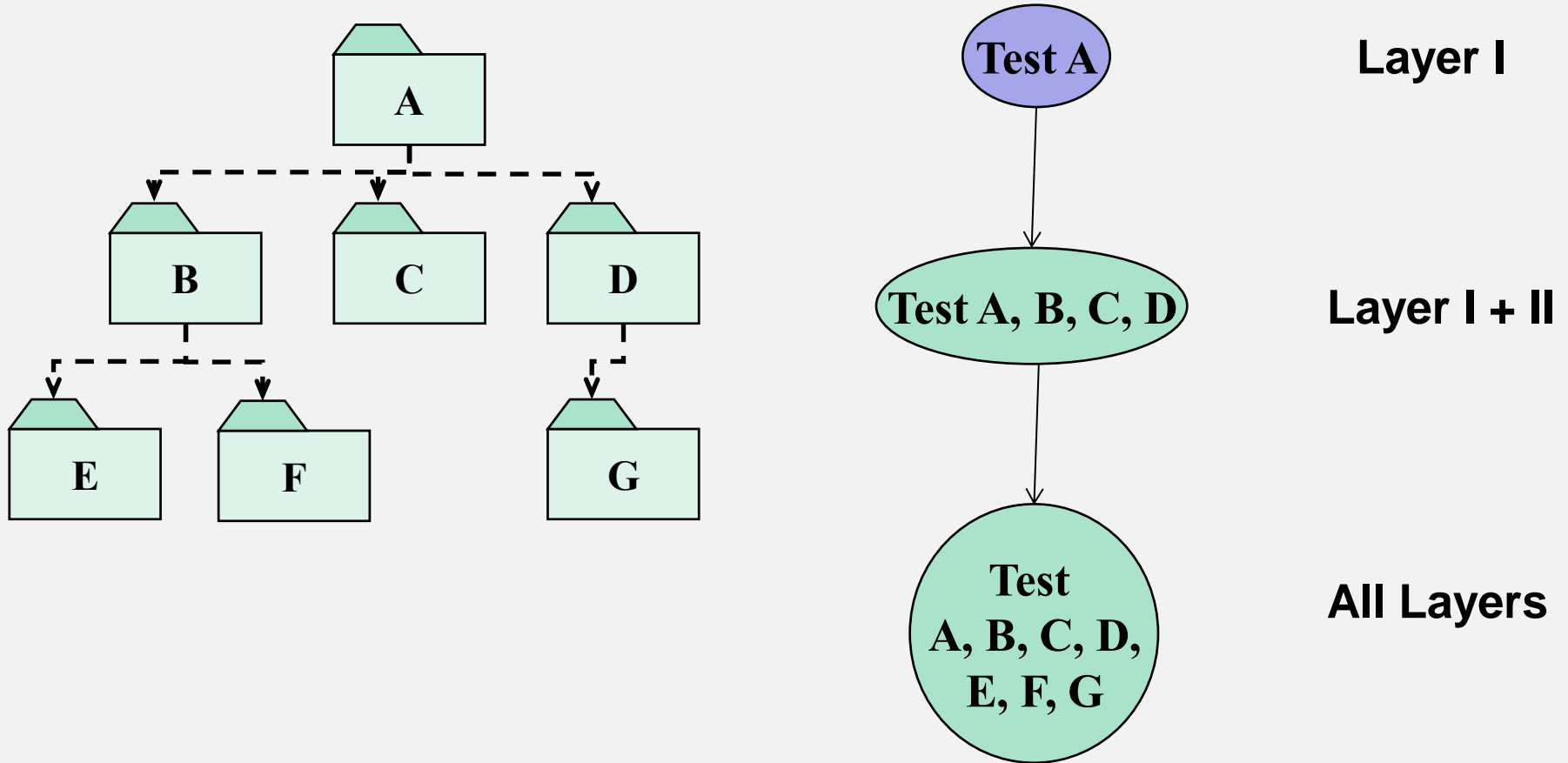
Top-down Testing Strategy

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.

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

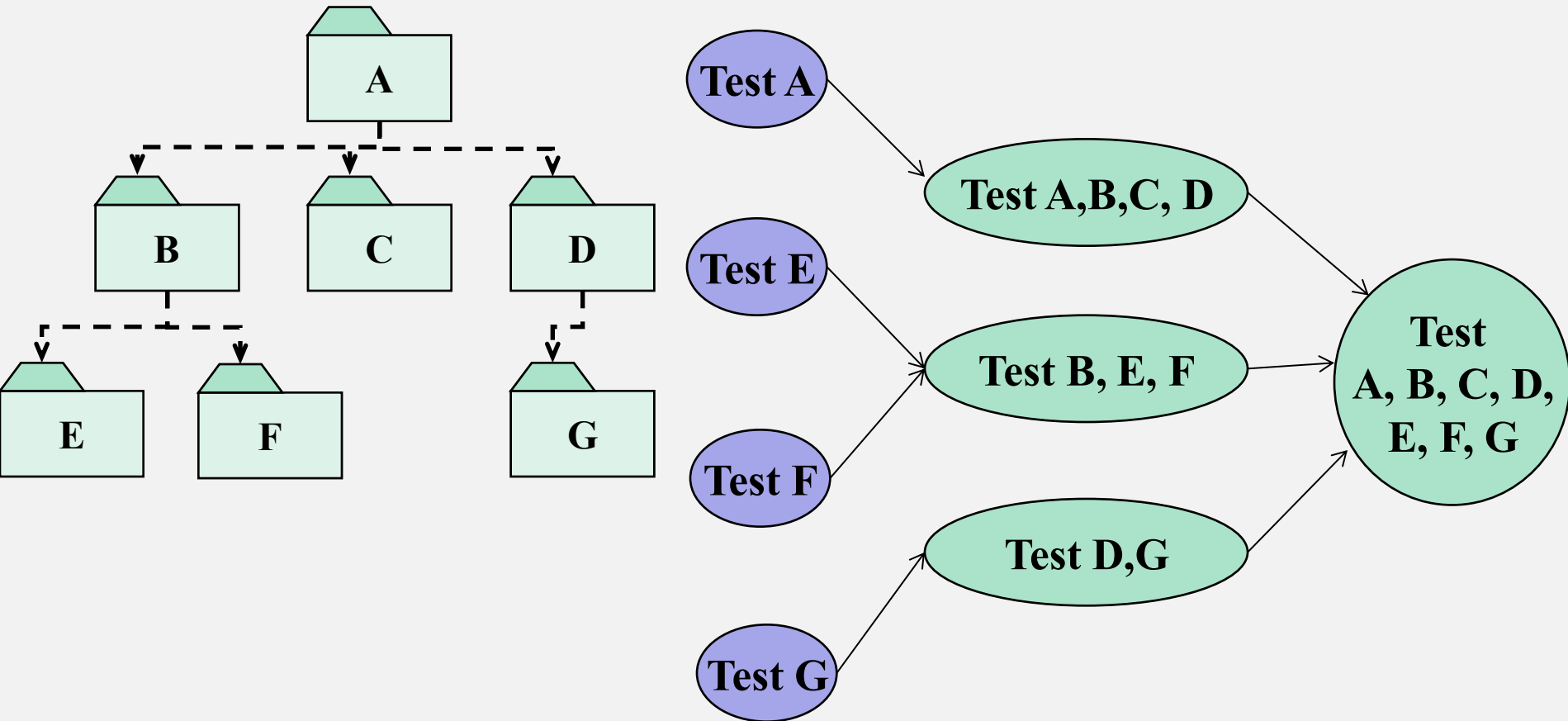The system is viewed as having three layers

  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
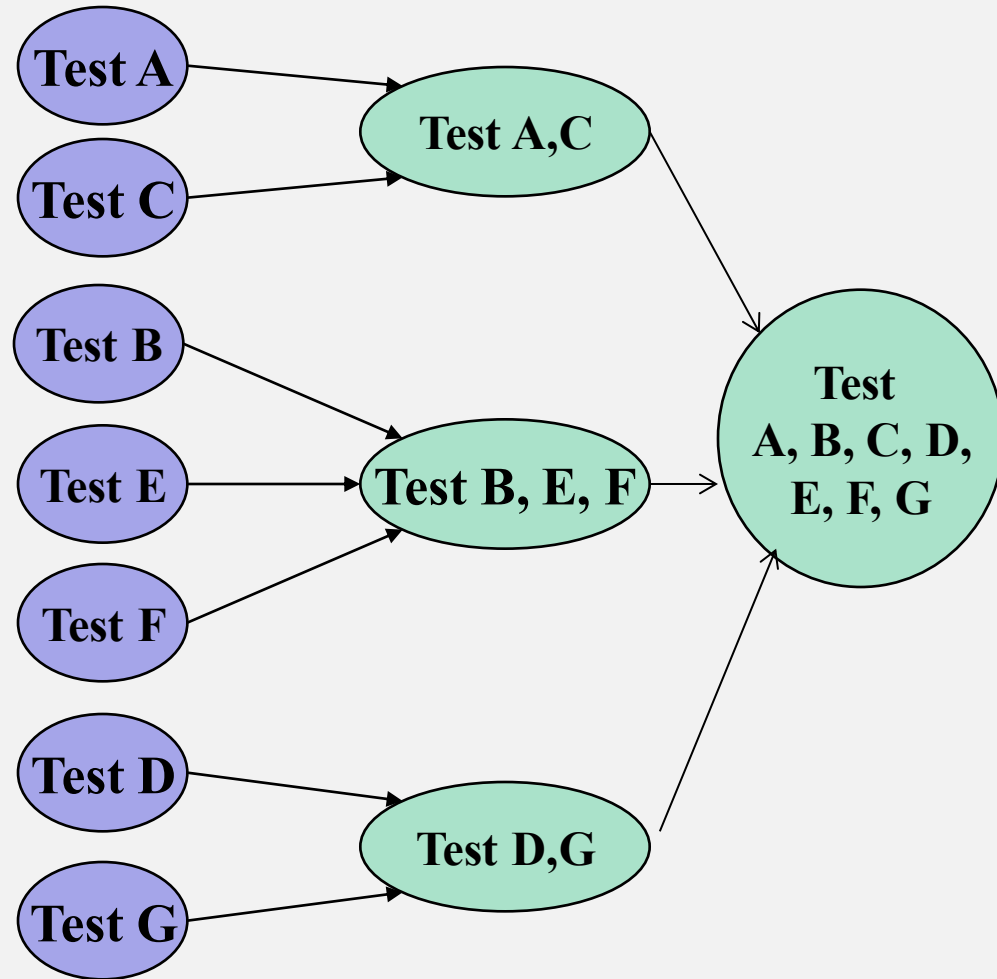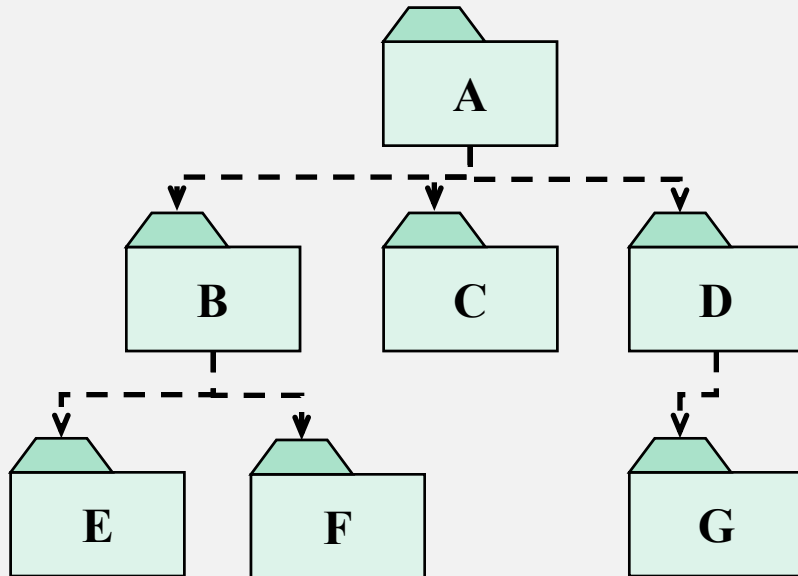
Top layer with stubs

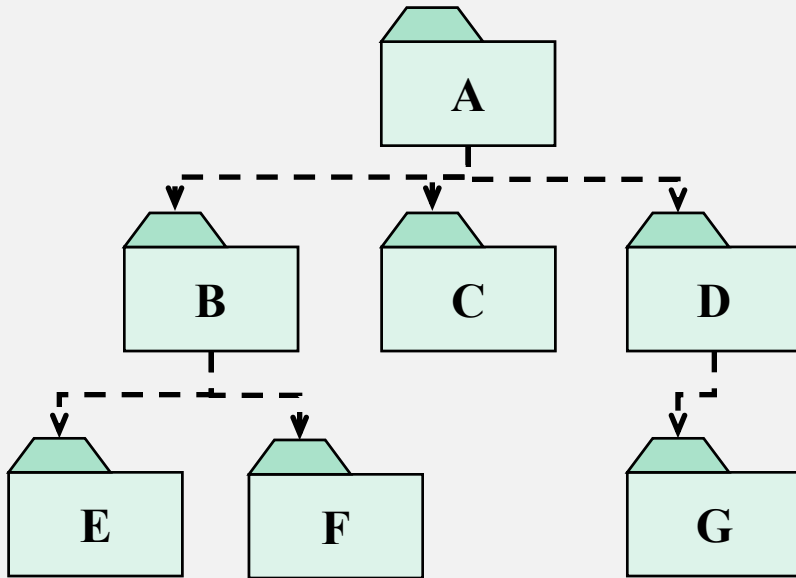Bottom layer with drivers

Test in parallel:

Top layer accessing middle layer (top layer replaces drivers)

Bottom accessed by middle layer (bottom layer replaces stubs).

Modified Sandwich Testing

Big-Bang Approach

This unit tests each of the subsystems, and then does one gigantic integration test, in which all the subsystems are immediately tested together.
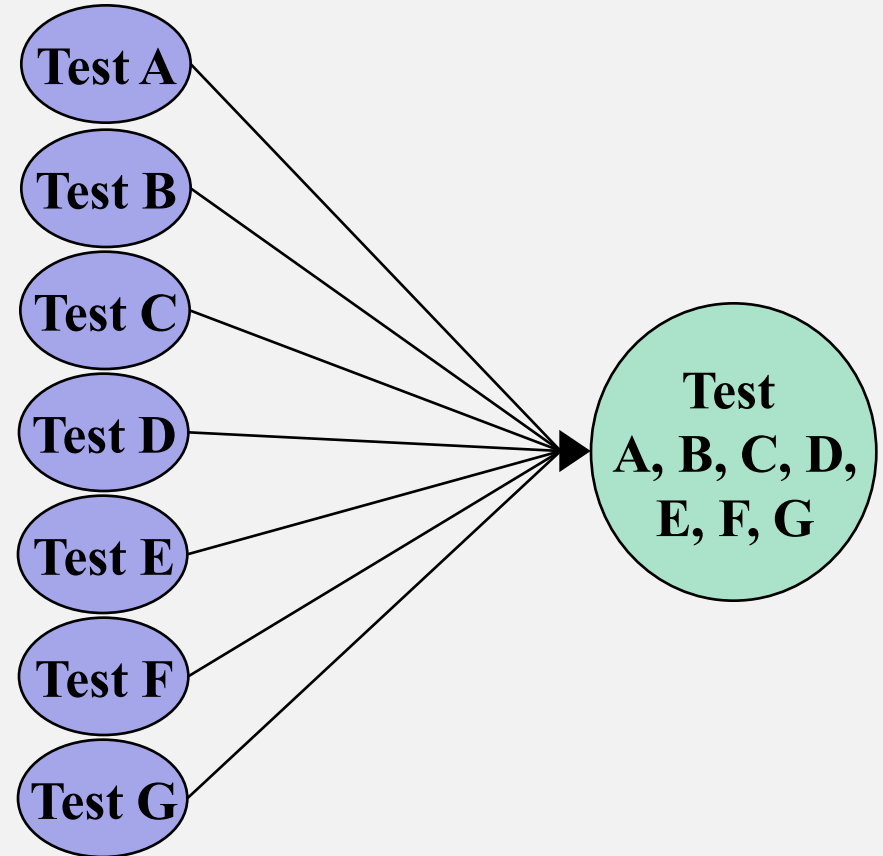
Continuous Testing

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.

Example: A 3-Layer-Design

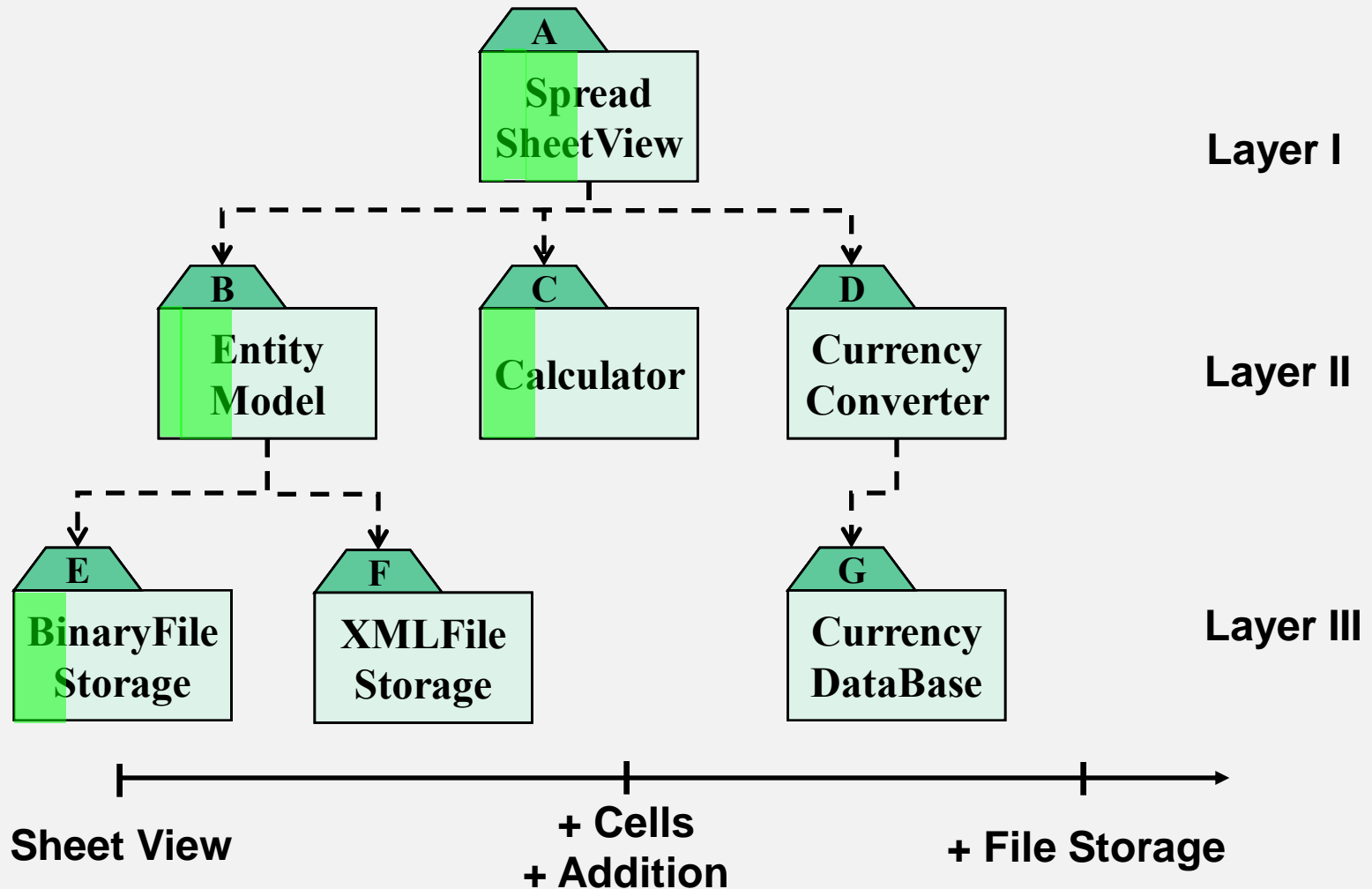# Steps in Integration Testing

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.*

System Testing

System testing is concerned with testing the behavior of an entire system.

System testing is usually considered appropriate for assessing the nonfunctional requirements—such as security, performance, reliability, availability, usability.

System Testing

Functional Testing

Validates functional requirements

Performance Testing

Validates non-functional requirements

Acceptance Testing

Validates 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 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.

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:
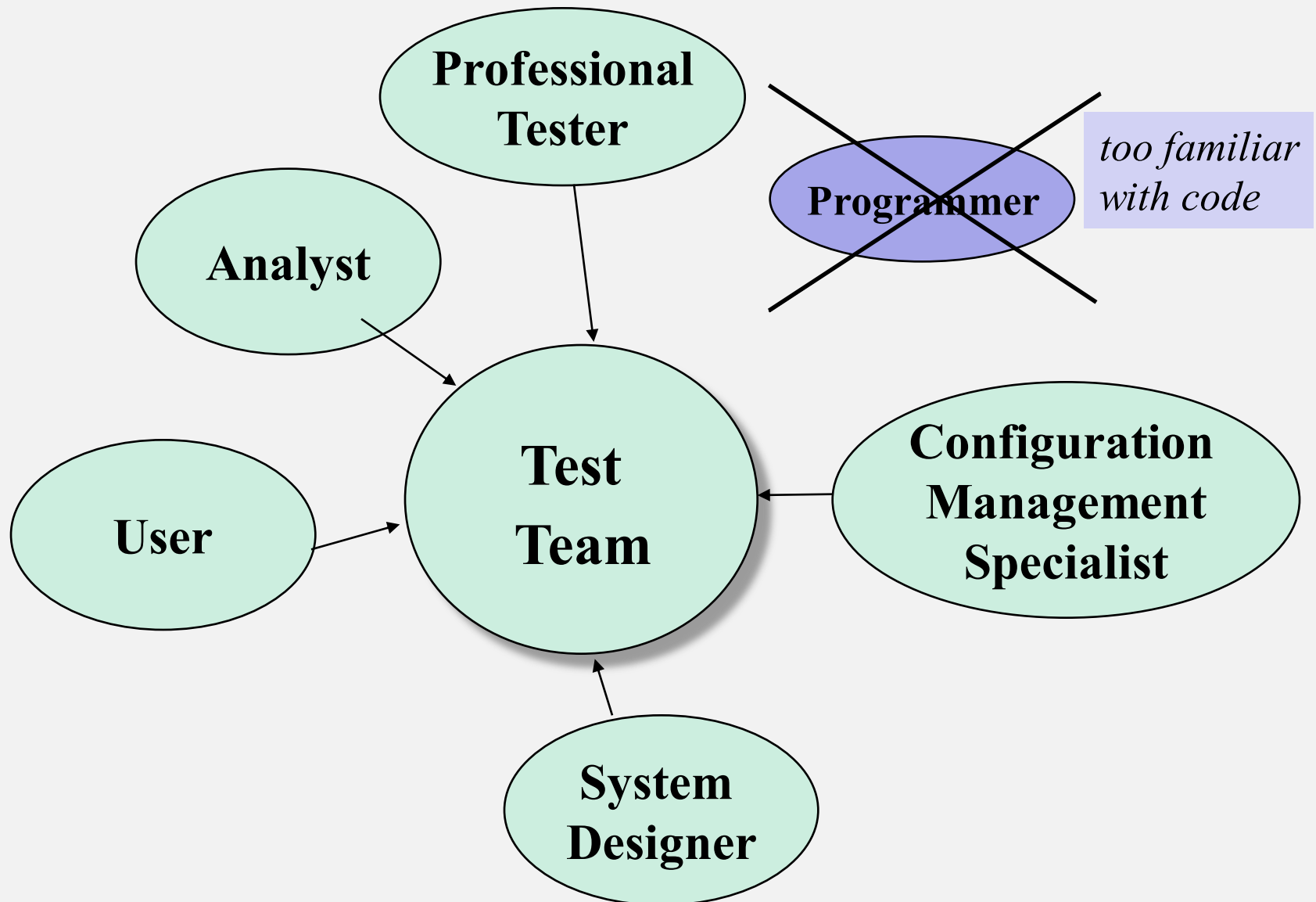
- Conducted at client's environment (developer is not present)
- Software gets a realistic workout in target environment.

Testing has many activities

1. Establish the test objectives
2. Design the test cases
3. Write the test cases
4. Test the test cases
5. Execute the tests
6. Evaluate the test results
7. Change the system
8. Do regression testing

Test Team

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

Testing has its own lifecycle

Thank You