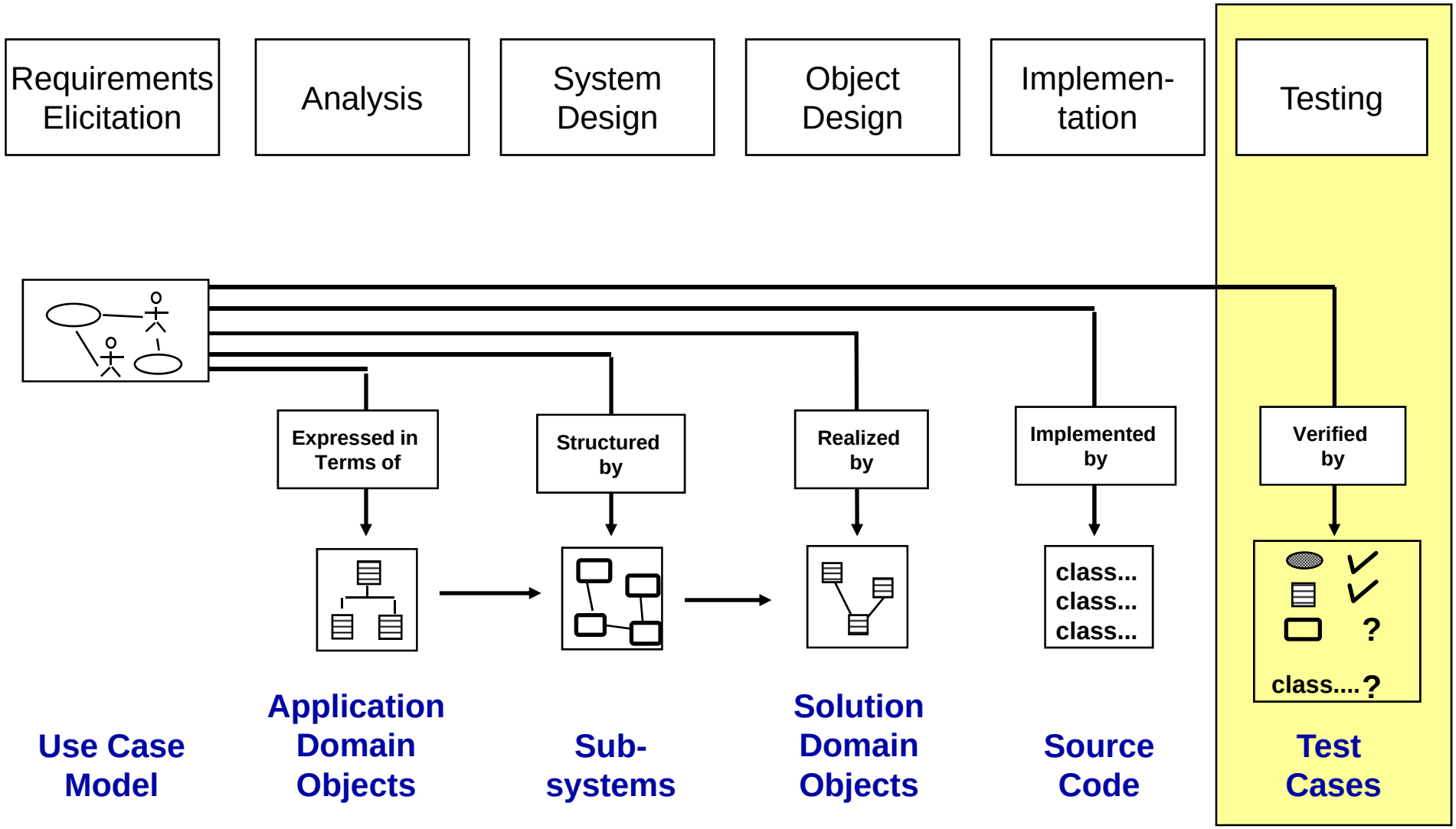
Chapter 15: Testing - 2

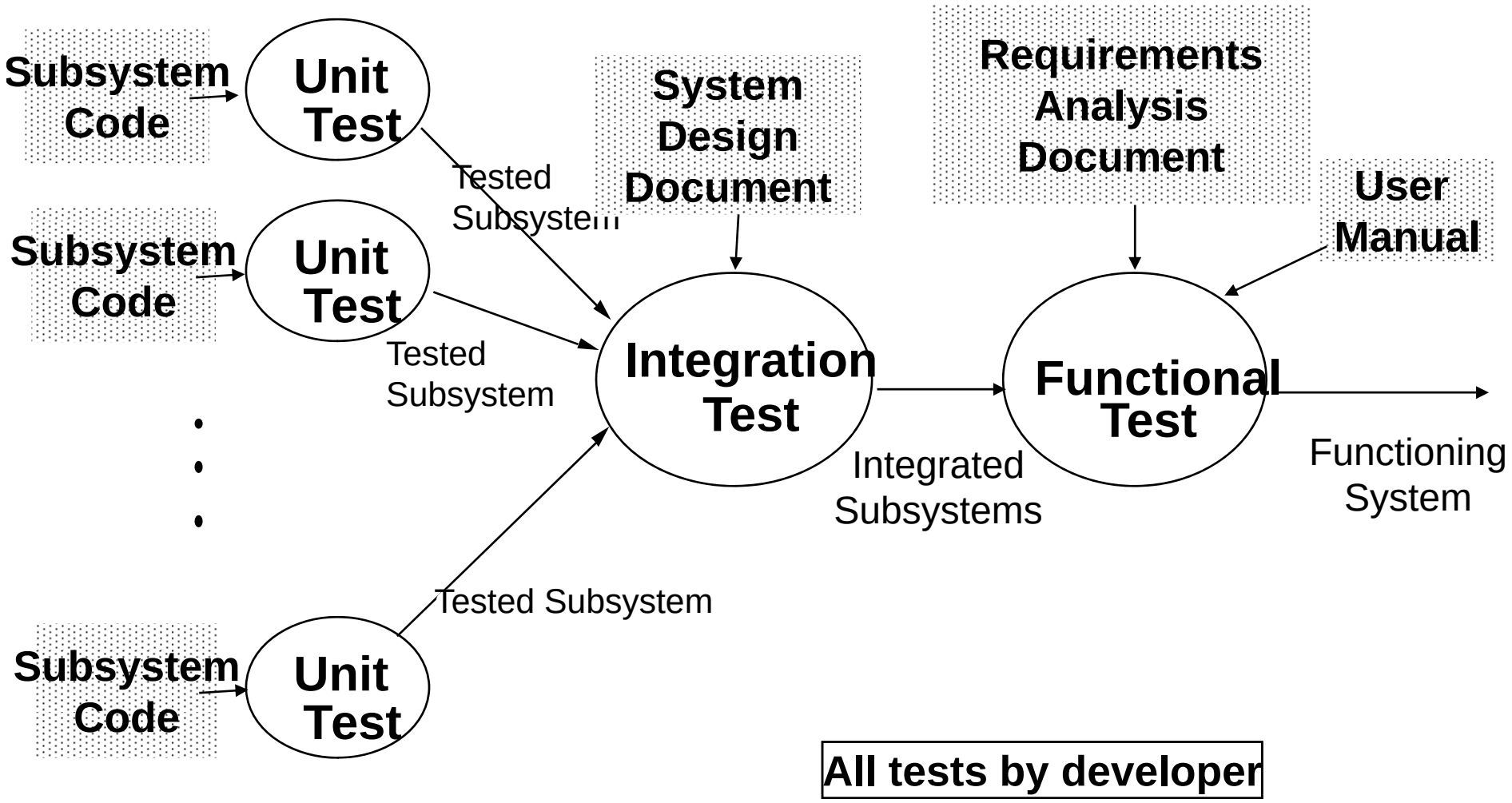
Object-**O**riented
Software**C**onstruction

Armin B. Cremers, Tobias Rho, Daniel Speicher, Holger Mügge
(based on Bruegge & Dutoit)

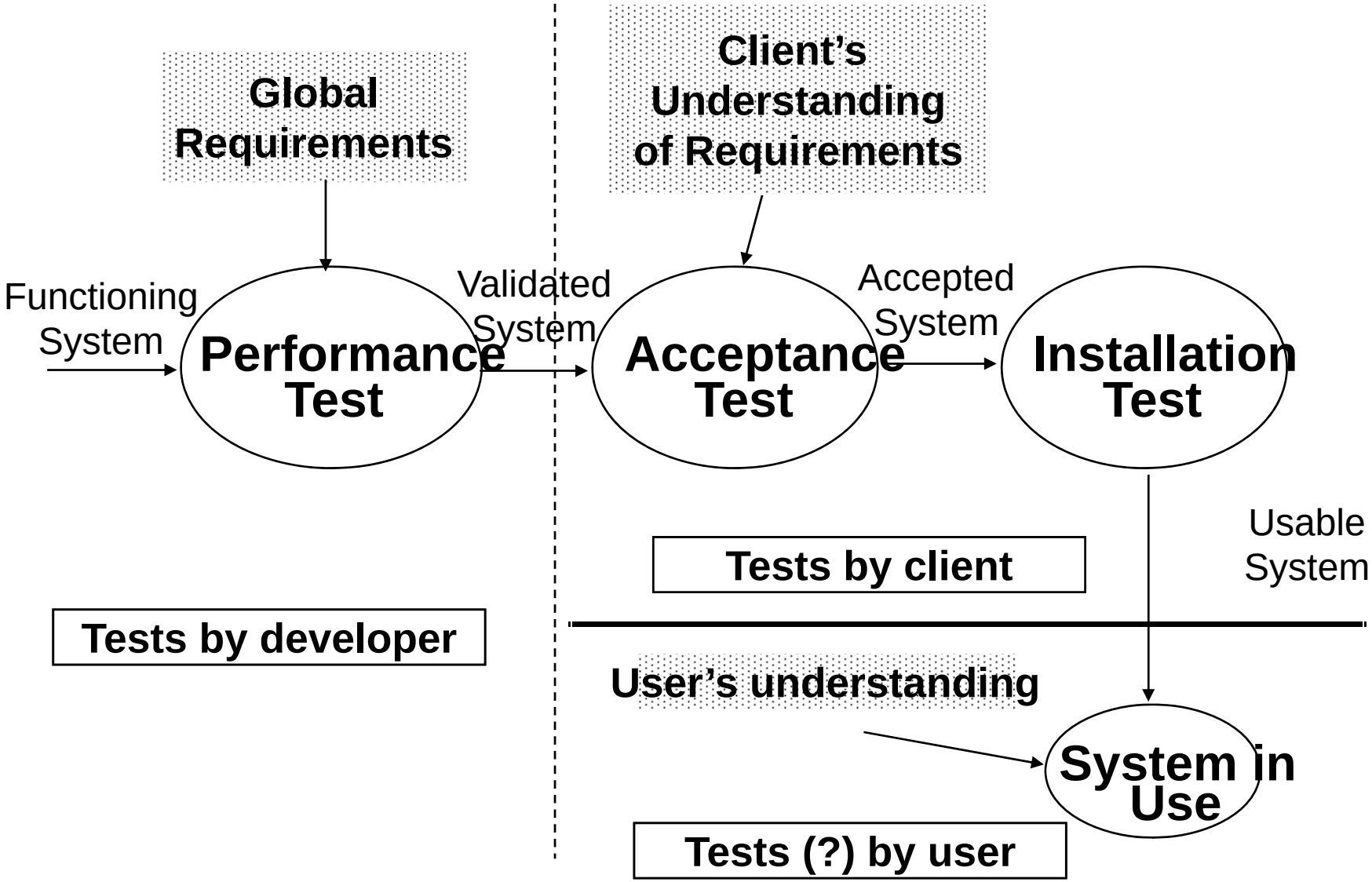
Software Lifecycle Activities ...and their models



Testing Activities



Testing Activities continued



The 4 Testing Steps

- 1. Select what has to be measured
  - Analysis: Completeness of requirements
  - Design: tested for cohesion
  - Implementation: Code tests
- 2. Decide how the testing is done
  - 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 (code, module, system) being tested or about the attribute being measured
- 4. Create the test oracle
  - An oracle contains of the predicted results for a set of test cases
  - The test oracle has to be specified 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:
  - Examples:
    - Force division by zero
    - Use sequence of test cases for interrupt handler

Unit-Testing Heuristics

- 1. Create unit tests as soon as object design is completed:
  - Black-box test: Test the use cases & functional model
  - White-box test: Test the dynamic model
- 2. Develop the test cases
  - Goal: Find the minimal number of test cases to cover as many paths as possible
- 3. Create a test harness
  - Test drivers and test stubs are needed for integration testing
- 4. Specify the test oracle
  - Often the result of the first successfully executed test
- 5. Execute the test cases
  - Don't forget regression testing
  - Re-execute test cases whenever a change is made.
- 6. Compare the results of the test with the test oracle
  - Automate as much as possible

Integration Testing Strategy

- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design.
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (non-incremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection, use the system decomposition from the system 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. Do functional testing: Define test cases that exercise all use cases with the selected component
- 4. Do structural testing: Define test cases that exercise the selected component
- 5. 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 errors in the (current) component configuration.

Which integration strategy should you use?

- Factors to consider
  - Amount of test harness (stubs &drivers)
  - Location of critical parts in the system
  - Availability of hardware
  - Availability of components
  - Scheduling concerns
- Bottom up approach
  - Good for object oriented design methodologies
  - Test driver interfaces must match component interfaces
  - Top-level components are usually important and cannot be neglected up to the end of testing
  - Detection of design errors postponed until end of testing
- Top down approach
  - Test cases can be defined in terms of functions examined
  - Need to maintain correctness of test stubs
  - Writing stubs can be difficult

System Testing

- Functional Testing
- Structure Testing
- Performance Testing
- Acceptance Testing
- Installation Testing
- Impact of requirements on system testing:
  - The more explicit the requirements, the easier they are to test.
  - Quality of use cases determines the ease of functional testing
  - Quality of subsystem decomposition determines the ease of structure testing
  - Quality of non-functional requirements and constraints determines the ease of performance tests

Structure Testing

- Essentially the same as white-box testing.
- Goal: Cover all paths in the system design
  - Exercise all input and output parameters of each component.
  - Exercise all components and all calls (each component is called at least once and every component is called by all possible callers.)
  - Use conditional and iteration testing as in unit testing.

Functional Testing

- Essentially the same as black-box testing
- Goal: Test the functionality of system
- Test cases are designed based on the requirements specification (better: user manual) and centered around requirements and key functions (use cases)
- The system is treated as a black box.
- Unit test cases can be reused but in the end, new user-oriented test cases have to be developed as well.

Performance Testing

- Stress Testing
  - Stress limits of system (peak demands, maximum # of users, long-term operation)
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility tests
  - Test backward compatibility with existing systems
- Security testing
  - Try to violate security requirements
- Timing tests
  - Evaluate response times and time to perform a function
- Environmental tests
  - Test tolerances for heat, humidity, motion, portability
- Quality testing
  - Test reliability, maintainability and availability of the system
- Recovery testing
  - Tests the system's response to presence of errors or loss of data.
- Human factors testing
  - Tests the user interface

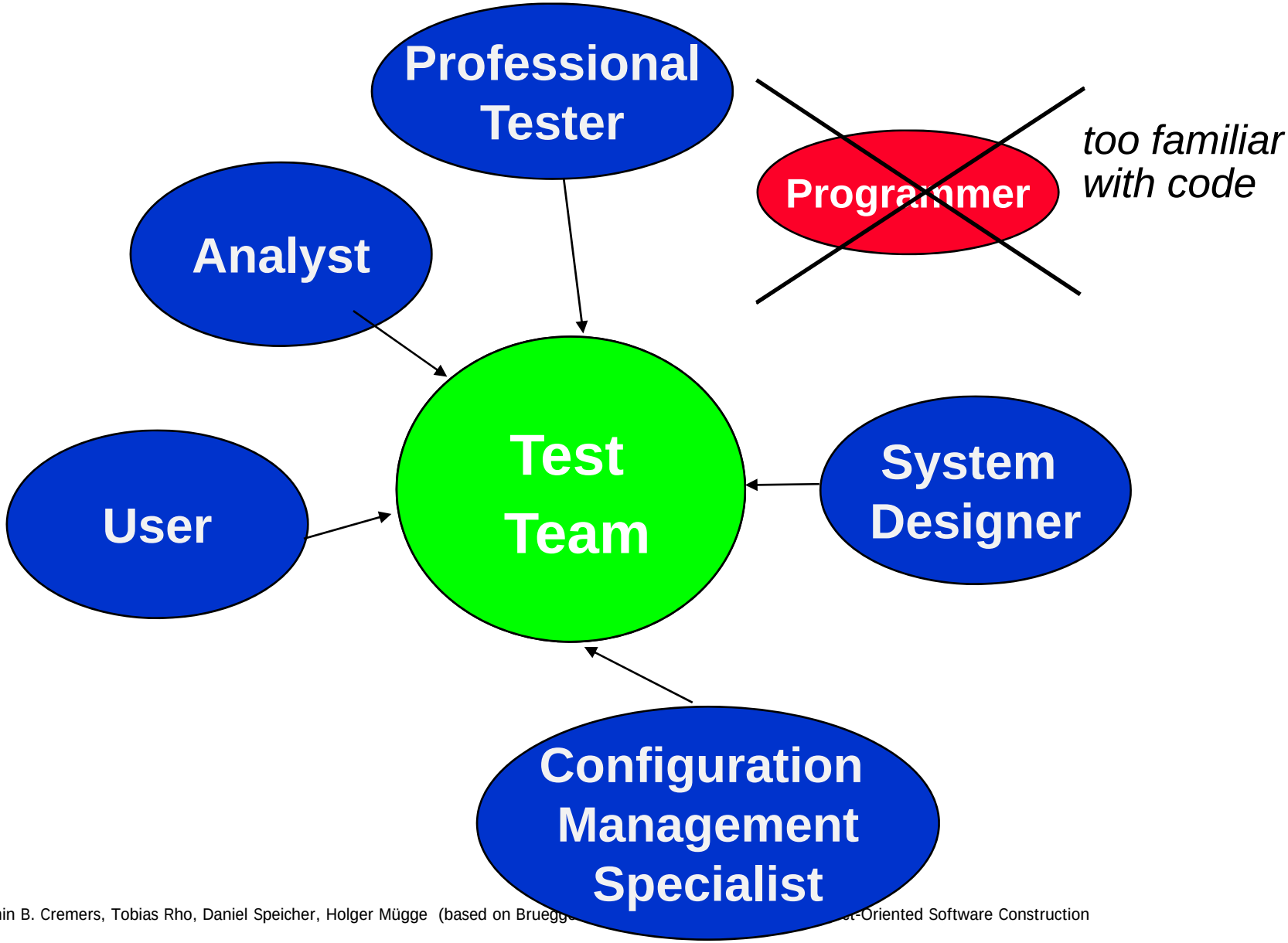
Test Cases for Performance Testing

- Push the (integrated) system to its limits.
- Goal: Try to break the subsystem
- Test how the system behaves when overloaded.
  - Can bottlenecks be identified? (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, 2000, 5000, 10000 items.
- What is the amount of time spent in different use cases?
  - Are typical cases executed in a timely fashion?

Acceptance Testing

- Goal: Demonstrate system is ready for operational use
  - Choice of tests is made by client/sponsor
  - Many tests can be taken from integration testing
  - Acceptance test is performed by the client, not by the developer.
- Majority of all bugs in software is typically found by the client after the system is in use, not by the developers or testers. Therefore two kinds of additional tests:
- Alpha tests:
  - Sponsor uses the software at the developer's site.
  - Software used in a controlled setting, with the developer always ready to fix bugs.
- Beta tests:
  - Conducted at sponsor's site (developer is not present)
  - Software gets a realistic workout in target environment
  - Potential customer might get discouraged

Test Team
Organizational issues



Types of Testing

- Unit Testing (last lecture):
  - Individual subsystem
  - Carried out by developers (of components)
  - Goal: Confirm that subsystems is correctly coded and carries out the intended functionality

- Integration Testing (mainly this lecture):
  - Groups of subsystems (collection of classes) and eventually the entire system
  - Carried out by developers
  - Goal: Test the interface and the interplay among the subsystem

Types of Testing

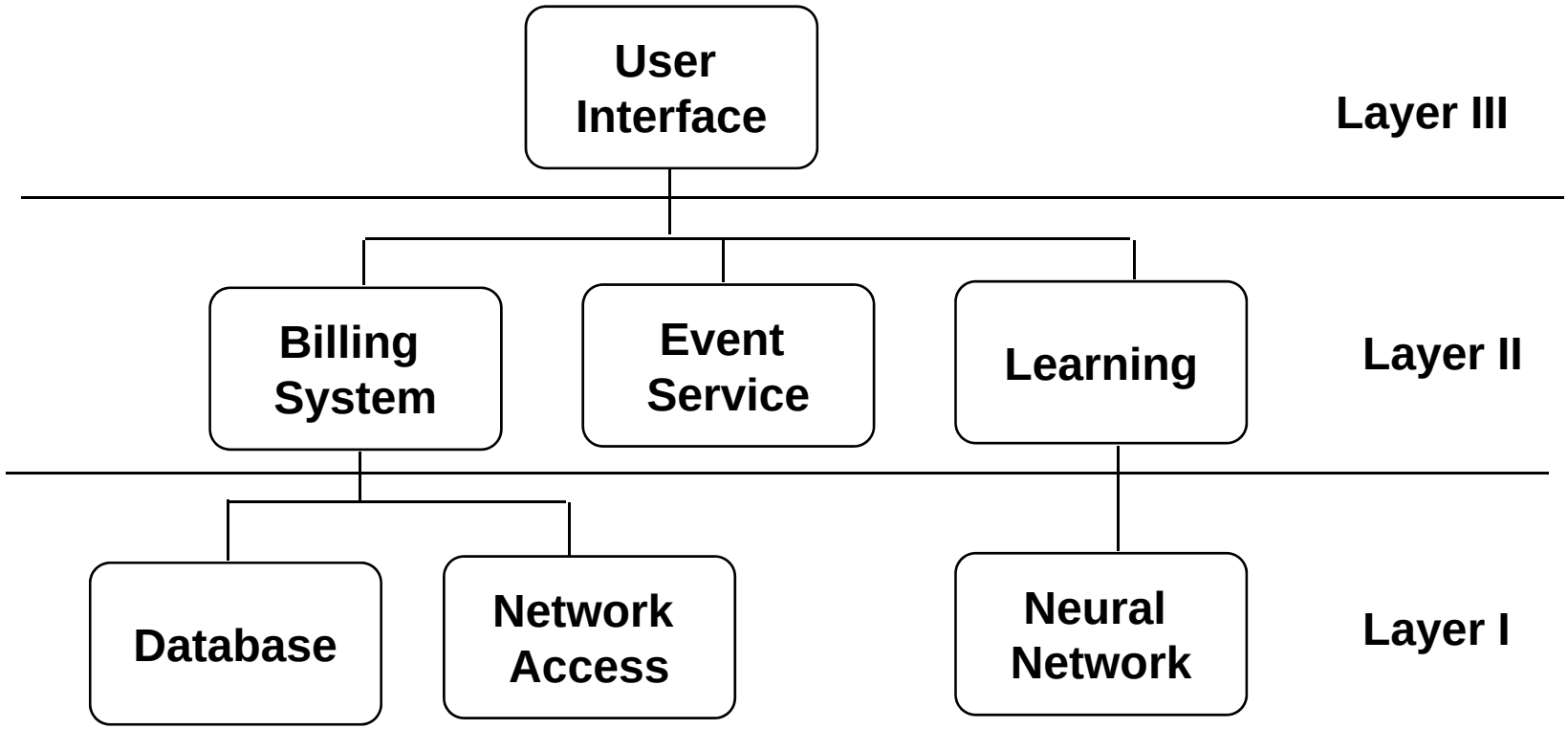
- System Testing:
  - The entire system
  - Carried out by developers (testers!)
  - Goal: Determine if the system meets the requirements (functional and global)
  - Functional Testing: Test of functional requirements
  - Performance Testing: Test of non-functional requirements

- Acceptance and Installation Testing:
  - Evaluates the system delivered by developers
  - Carried out by the client.
  - Goal: Demonstrate that the system meets customer requirements and is ready to use

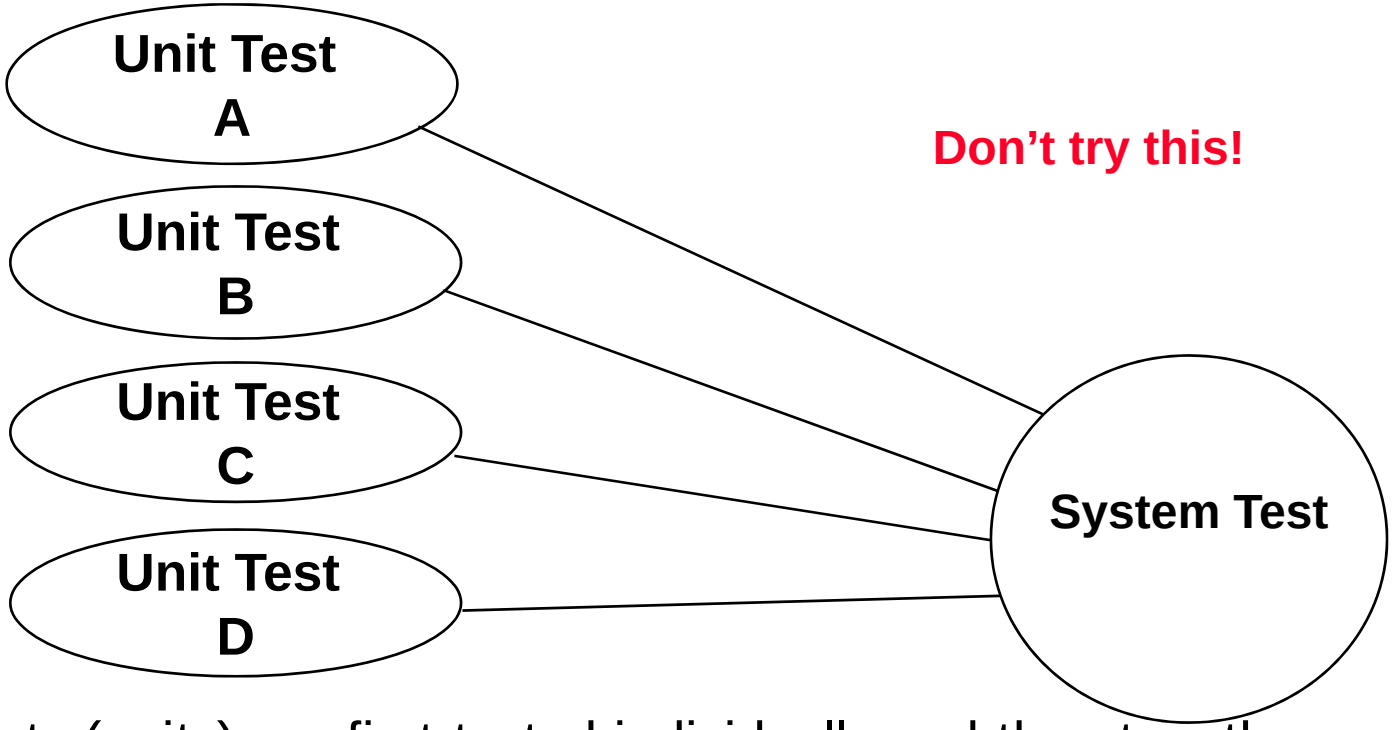
Integration Testing Strategy

- Test (sub-)systems (cluster) for problems that arise from subsystem interactions
- Assumption:
  - The entire system is viewed as a collection of subsystems determined during the system and object design.
  - System Decomposition is hierarchical
- The order in which the subsystems are selected for testing and integration determines the testing strategy
  - Big bang integration (Nonincremental)
  - Bottom up integration
  - Top down integration
  - Sandwich testing
  - Variations of the above
- For the selection use the system decomposition from the System Design

Example: Three Layer Call Hierarchy



Integration Testing: Big-Bang Approach



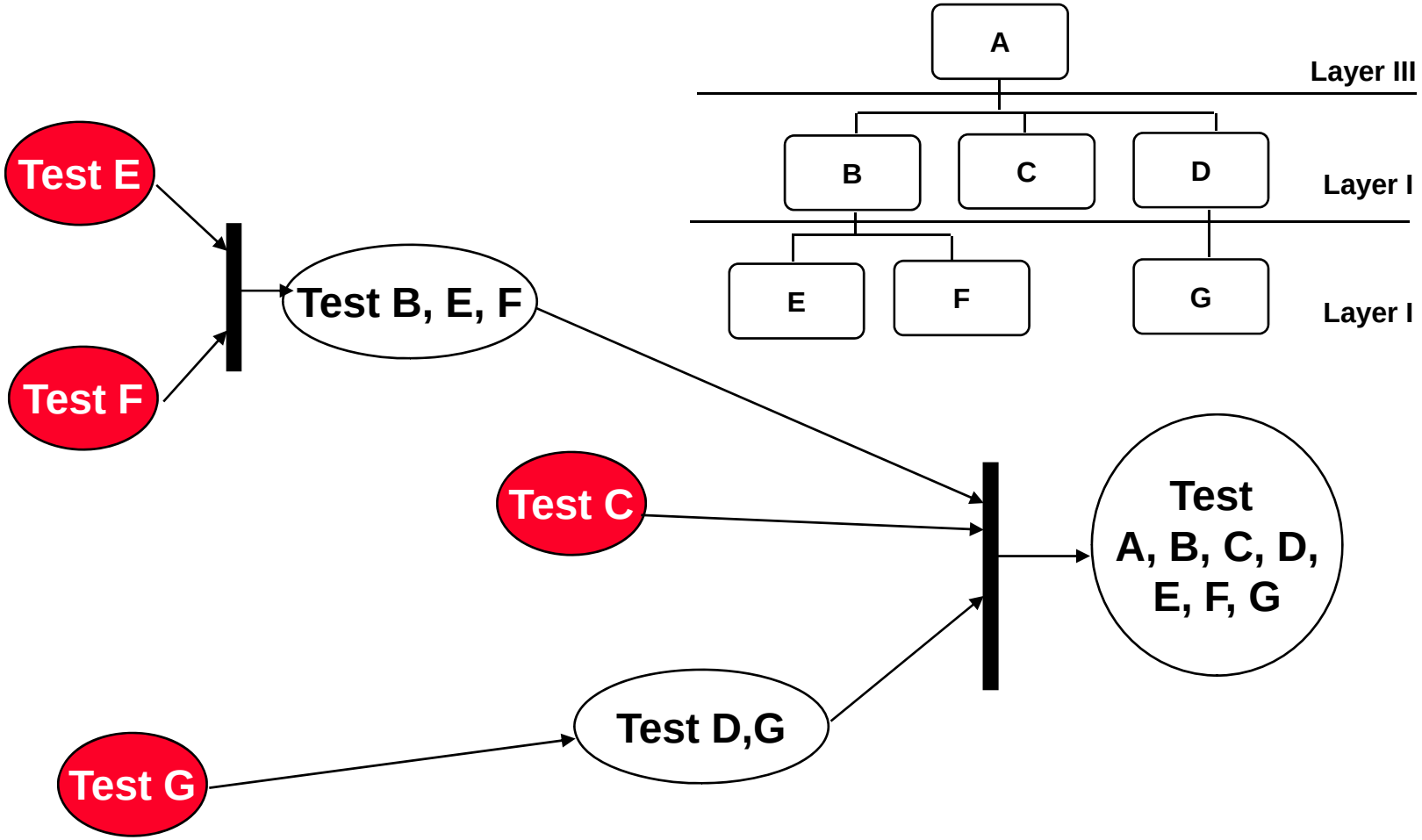
- All components (units) are first tested individually and then together as a single and entire system:
- Pros:
  - No test stubs (mocks) and drivers are needed
- Cons:
  - Difficult to pinpoint the specific component responsible for the failure

➔ Results in Strategies that integrate only a few components at the time

Bottom-up Testing Strategy

- The subsystem in the lowest layer of the call hierarchy are tested individually
  - Infrastructure is tested first
- Then the next subsystems are integrated and tested from the next layer up that call the previously tested subsystems
  - Increment one subsystem at a time
  - Order of integration depends on importance of subsystem etc.
- This is done repeatedly until all subsystems are included in the testing
  - Regression Tests: Rerun previous tests
- Only Test Drivers are used to simulate the components of higher layers
- No Test Stubs!

Bottom-up Integration



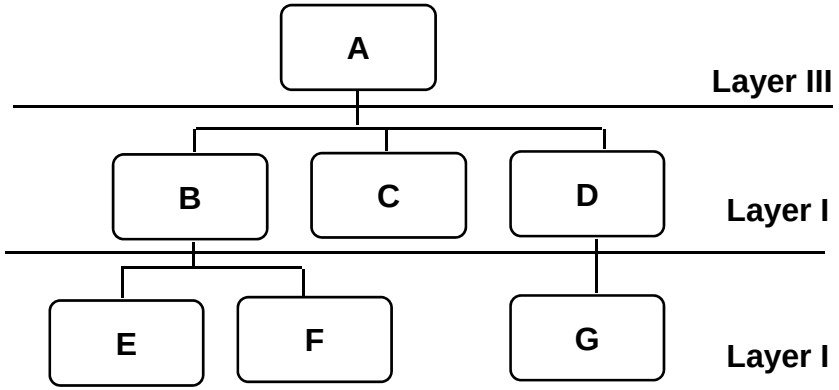
Pros and Cons of bottom up integration testing

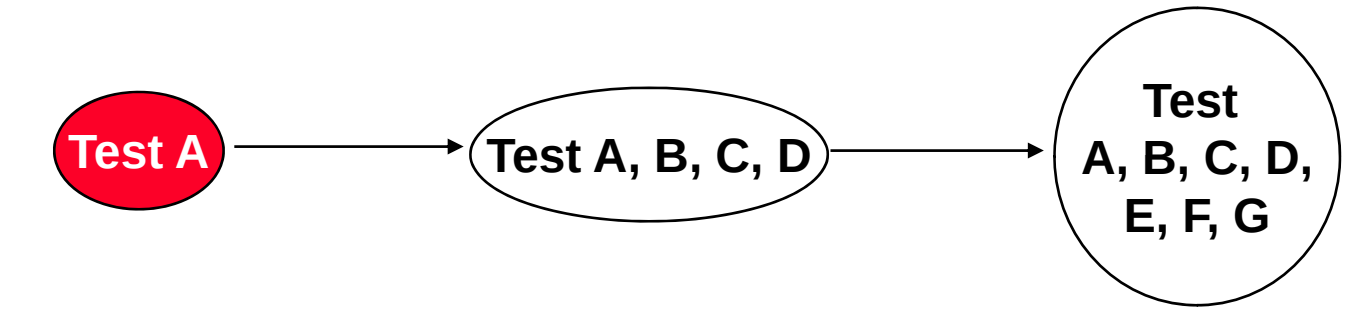
- Pros:
  - Interface faults can be more easily found (the usage of test drivers accomplishes a clear intention of the underlying interfaces of the lower layer)
  - No Stubs are necessary
- Cons:
  - Components of the User Interface are tested last
  - Test cases often hard to derive
  - Faults found in the top layer may lead to changes in the subsystems of lower layers, invalidating previous tests.

Top-down Testing Strategy

- Test the top layer of the controlling subsystem first
  - The skeleton of the program is tested
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
  - Increment one subsystem at a time
- Do this until all subsystems are incorporated into the test
- Test Stubs are used to simulate the components of lower layers that have not yet been integrated.
- No drivers are needed

Top-down Integration Testing





Layer III

Layer III + II

All Layers

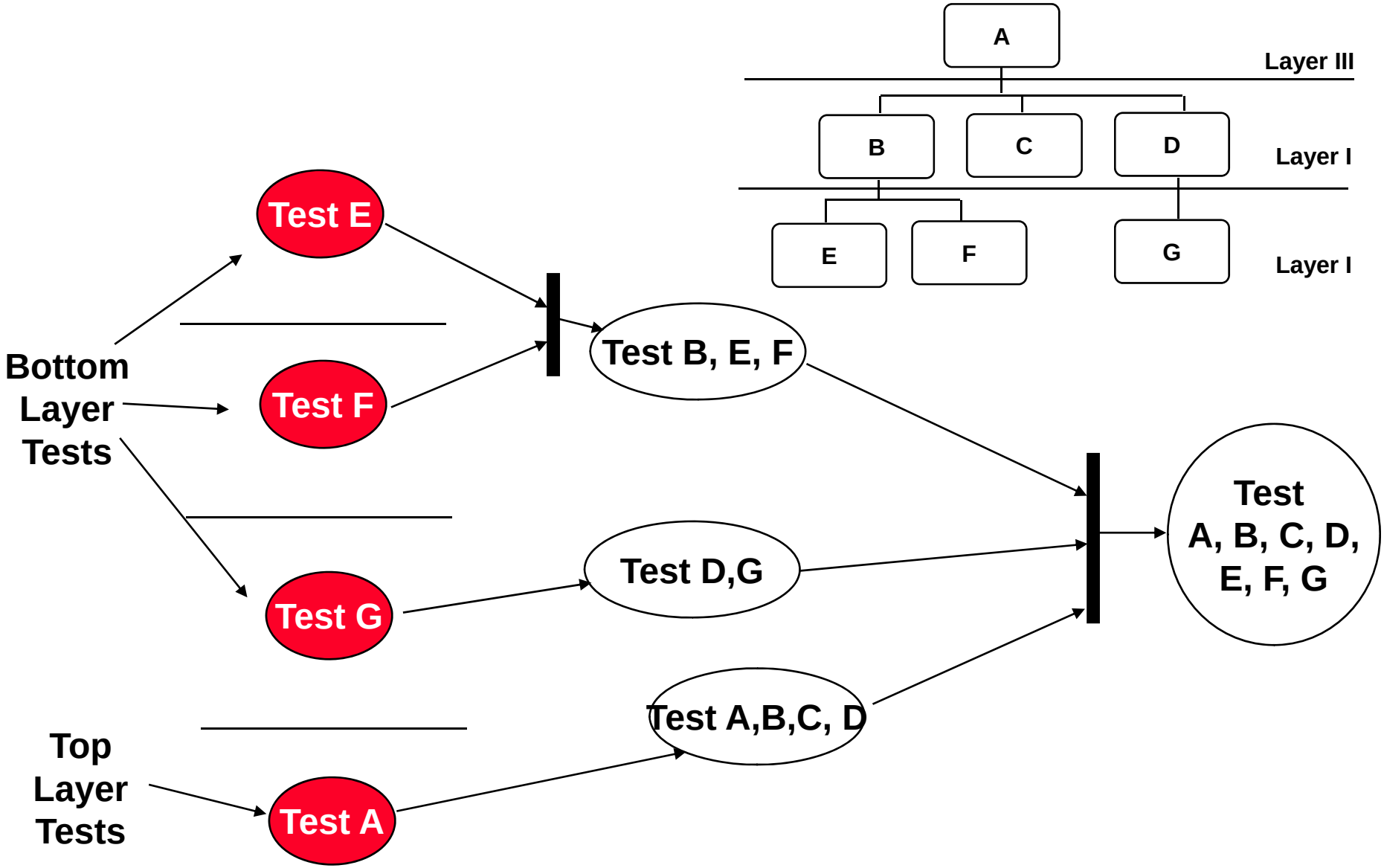
Pros and Cons of top-down integration testing

- Pros:
  - Test cases can be defined in terms of the functionality of the system (functional requirements)
  - More effective for finding faults that are visible to the user
- Cons:
  - 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 Strategy

- Combines top-down strategy with bottom-up strategy (parallel testing is possible)
- The system is view as having three layers
  - A target layer in the middle
  - A layer above the target (top layer)
  - A layer below the target (bottom layer)
  - Testing converges towards the target layer
- No Test Stubs and Drivers are necessary for bottom and top layer

Sandwich Testing Strategy



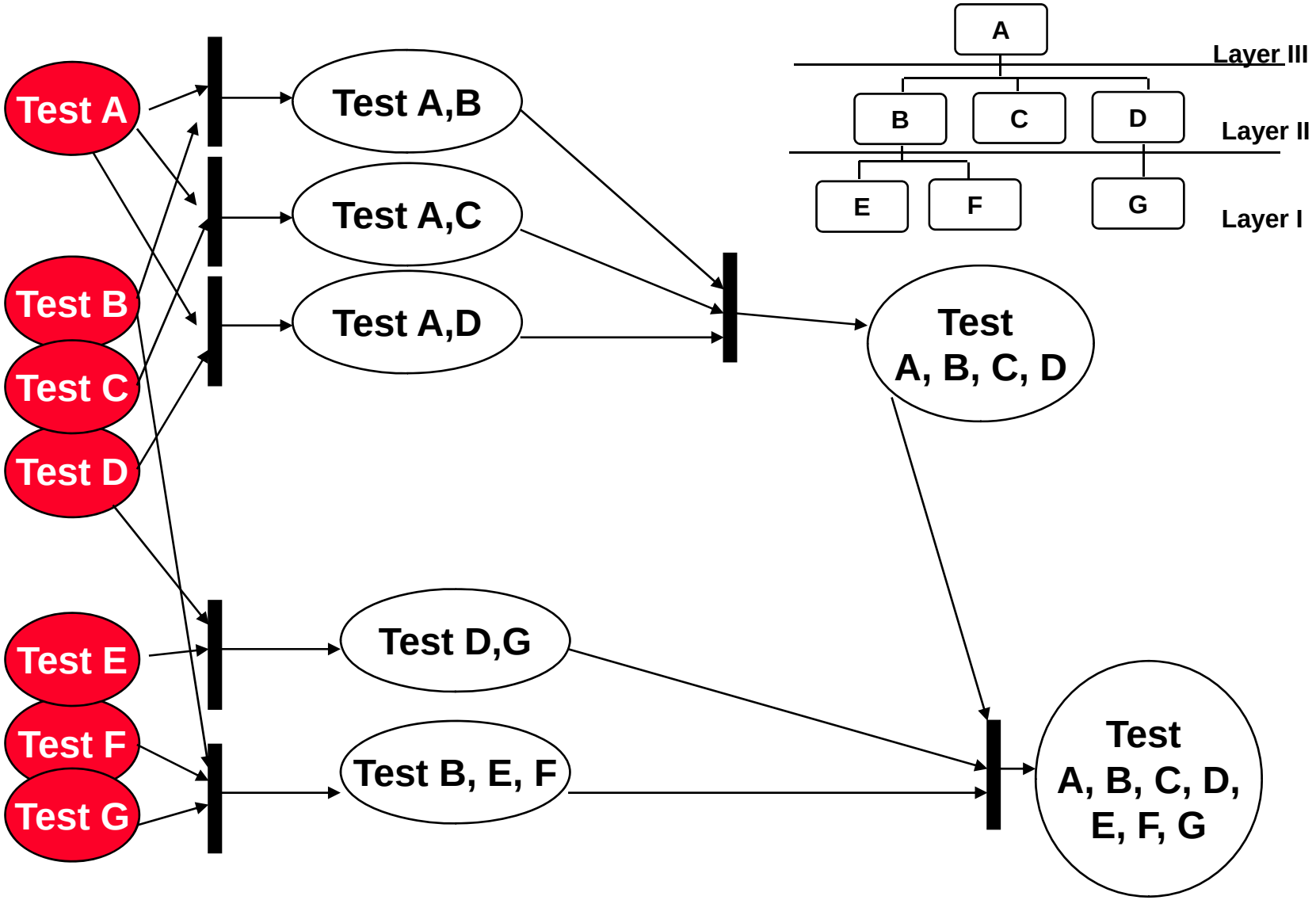
Pros and Cons of Sandwich Testing

- Pros:
  - Top and Bottom Layer Tests can be done in parallel
  - No Stubs and Drivers (saves development time)
- Cons:
  - System implementation needs to be finished
  - Does not test the individual subsystems on the target layer thoroughly before integration (C in the example)
- Solution: Modified sandwich testing strategy

Modified Sandwich Testing Strategy

- Tests the three layers individually before combining them in incremental tests with one another
- The individual layer tests consists of three tests:
  - Target layer test with drivers and stubs
  - Top layer test with stubs
  - Bottom layer test with drivers
- The combined Layer Tests consist of two tests:
  - Top layer accessing target layer (top layer replaces drivers)
  - Bottom accessed by target layer (bottom layer replaces stubs)

Modified Sandwich Testing Strategy



Using the Bridge Pattern to enable early Integration Testing

- Usage of Design Patterns supports testing
- Use the bridge pattern to provide multiple implementations under the same interface.