Software Engineering
Testing

Prof. Dr. Peter Müller

Software Component Technology

The slides in this section are partly based on the courses
“Software Engineering I” by Prof. Bernd Brügge, TU München and
“Software Engineering” by Prof. Jan Vitek, Purdue University

Summer Semester 06

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Why Does Software Contain Bugs?

- Our ability to **predict the behavior** of our creations is imperfect
  - Software is extremely **complex**
  - No developer can understand the whole system

- We make **mistakes**
  - **Unclear requirements**, miscommunication
  - Wrong **assumptions** (e.g., behavior of operating system)
  - Design **errors** (e.g., capacity of data structure too small)
  - Coding **errors** (e.g., wrong loop condition)

“First actual case of bug being found.”

9/9

0800 Arctan started

1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 795 correct

13"UC (032) MP - MC ~~1.9826[illegible]000~~ ~~2.13047641[illegible]~~) 4.615925059(-2)

(033) PRO 2 2.13047615

correct 2.13067615

Relays 6-2 in 033 failed special speed test

In relay " 11.000 test.

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Multy Adder Test.

1545 Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

~~1455~~ 1630 Arctangent started.

1700 closed down.

Increasing Software Reliability

Fault Avoidance

- Detect faults statically without executing the program
- Includes development methodologies, reviews, and program verification

Fault Detection

- Detect faults by executing the program
- Includes testing

Fault Tolerance

- Recover from faults at runtime (e.g., transactions)
- Includes adding redundancy (e.g., n-version programming)

Goal of Testing

- An error is a deviation of the observed behavior from the required (desired) behavior
  - Functional requirements (e.g., user-acceptance testing)
  - Nonfunctional requirements (e.g., performance testing)

- Testing is a process of executing a program with the intent of finding an error

- **A successful test is a test that finds errors**

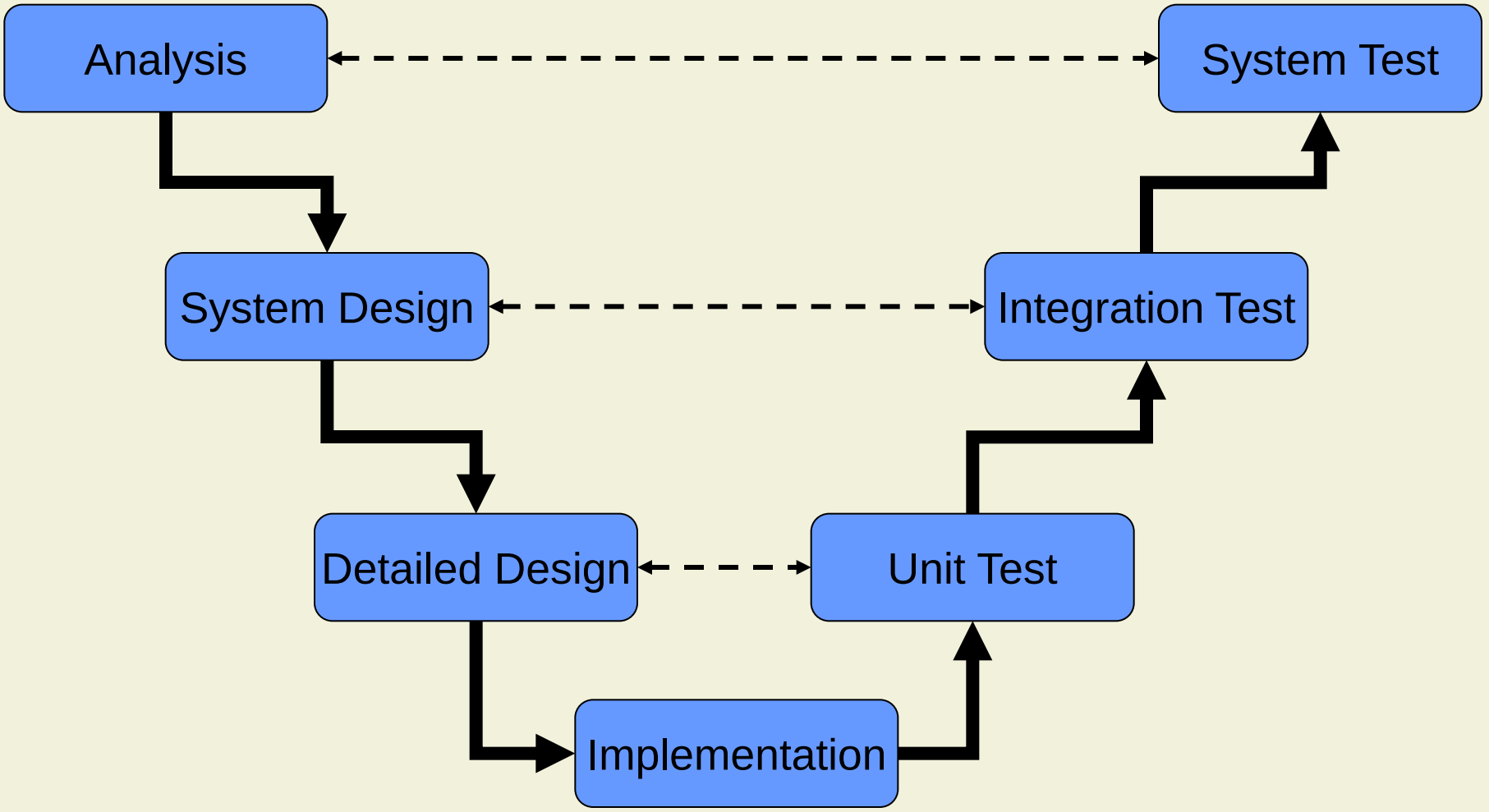
Limitations of Testing

"Testing can only show the presence of bugs, not their absence."

[E. W. Dijkstra]

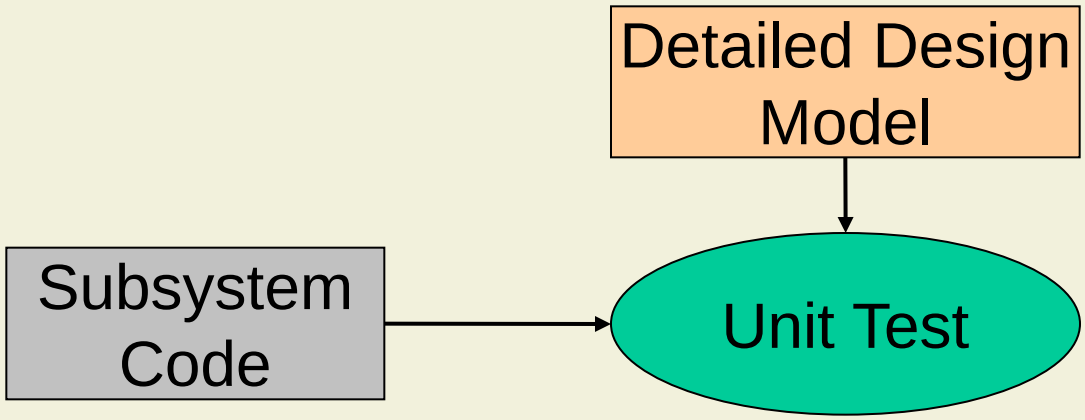
- It is impossible to completely test any nontrivial module or any system
  - Theoretical limitations: termination
  - Practical limitations: prohibitive in time and cost

Test Stages


Analysis
System Test
System Design
Integration Test
Detailed Design
Unit Test
Implementation

Unit Testing

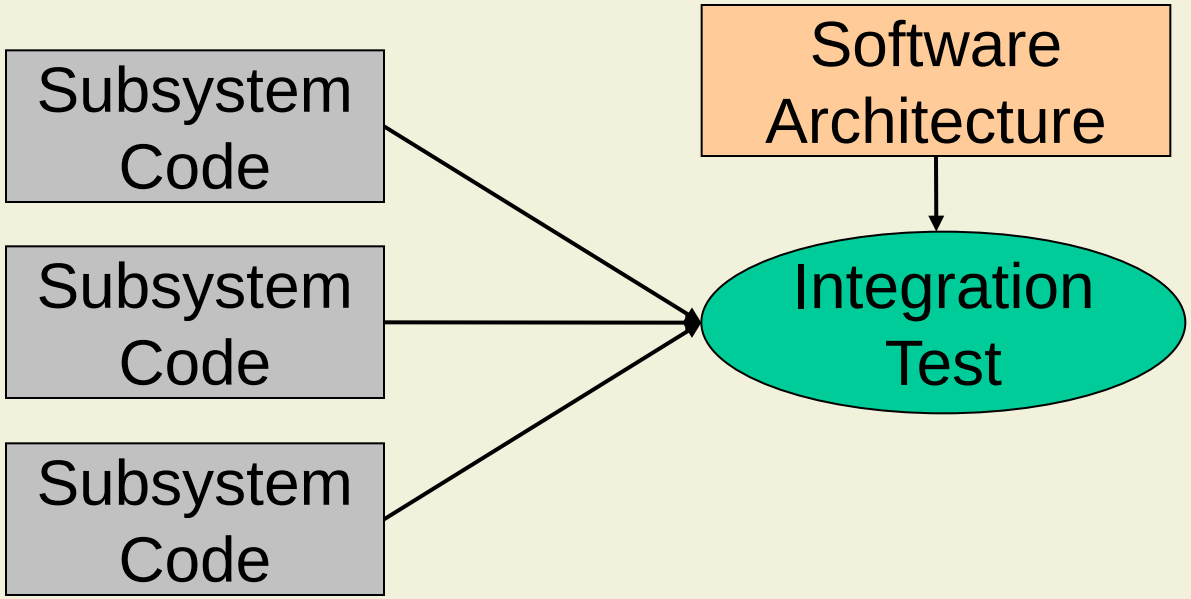
- Testing individual subsystems (collection of classes)



- Goal: Confirm that subsystem is correctly coded and carries out the intended functionality

Integration Testing

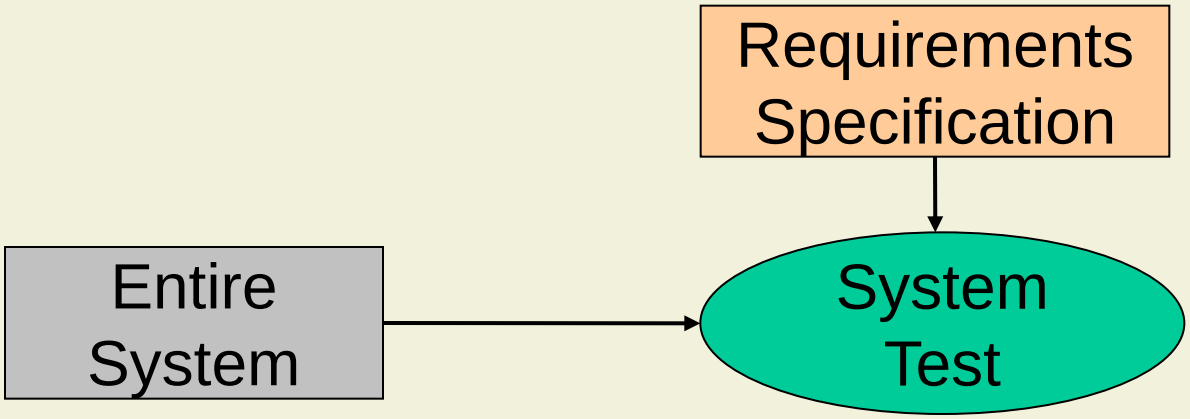
- Testing groups of subsystems and eventually the entire system



- Goal: Test interfaces between subsystems

System Testing

- Testing the entire system



- Goal: Determine if the system meets the requirements (functional and non-functional)

7. Testing

Test Case Design

Black-box testing

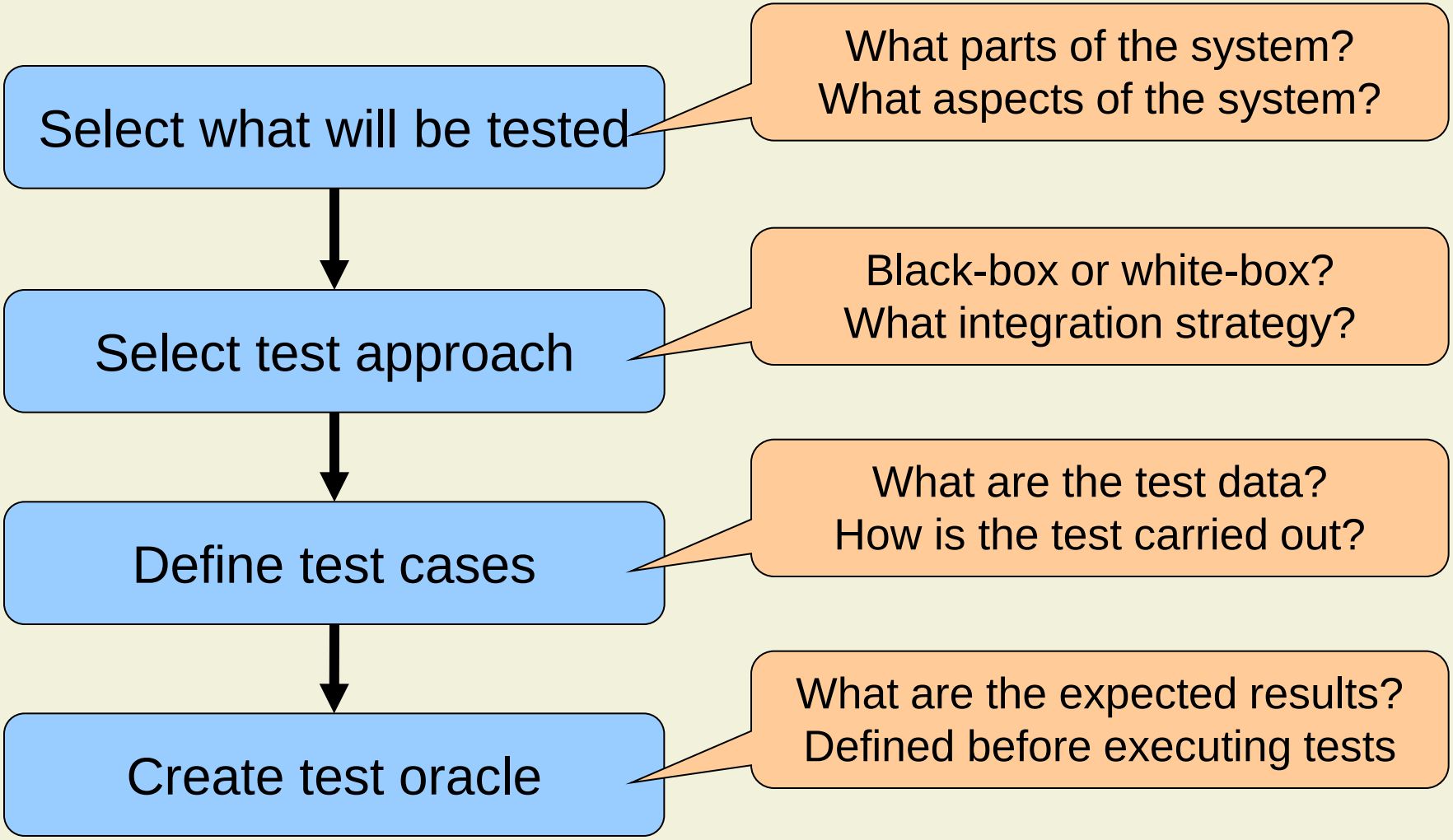
- Testing that UUT satisfies **requirements**
- Focus: I/O behavior
- No knowledge of the internals of the UUT
- Goal: Cover all the requirements

White-box testing

- Testing control structures
- Focus: Thoroughness (coverage)
- Knowledge of the internal structure of the UUT
- Goal: Cover all the code

- UUT = “Unit under test”

Testing Steps



Black-Box Testing

- Attempts to find
  - Incorrect or missing functions
  - Interface errors
  - Performance errors
  - Initialization and termination errors
- Use **analysis knowledge** about requirements
  - Use cases
  - Expected input data
  - Invalid input data
- Impossible to generate all possible inputs

Black-Box Testing: Equivalence Testing

- Divide input conditions into **equivalence classes**
  - Choose test cases for each equivalence class
- **Coverage**
  - Each possible input belongs to one of the equivalence classes
- **Disjointness**
  - No input belongs to more than one equivalence class
- **Representation**
  - If one test case of an equivalence class produces an error then the same error can be detected by using any other test case of the same equivalence class

Black-Box Testing: Valid and Invalid Input

- Equivalence classes have to cover **valid and invalid values**
- Input from a range of valid values
  - Below the range
  - Within the range
  - Above the range
- Input from a discrete set of valid values
  - Valid discrete value
  - Invalid discrete value

Black-Box Testing: Example

```
static int getDaysPerMonth( int month, int year )
  requires 1 <= month && month <= 12;
{ … }
```

Equivalence classes for month

1. Months with 30 days
2. Months with 31 days
3. February with 28 or 29 days

Equivalence classes for year

1. Non-leap years
2. Leap years

Partitioning ignores special rules for leap years

- Requires six test cases to cover all equivalence classes of valid input values

Black-Box Testing: Boundary Testing

- Large number of errors tend to occur at **boundaries of the input domain**
- Rather than select any element in an equivalence class, select those at the “edge” of the class
- Examples for boundary values
  - Leap years: 1900, 2000
  - Invalid months: 0, 13
  - Numbers in general: zero, a very small number, a very large number

White-Box Testing

- Why do white-box testing when black-box testing is used to test conformance to requirements?
  - "Bugs lurk in corners and congregate at boundaries" [B. Beizer]
- Use **design knowledge** about system structure, algorithms, data structures
  - Control structures (branches, loops, etc.)
  - Data structures (fields, arrays, etc.)
- Use **implementation knowledge** about algorithms
  - Force division by zero

White-Box Testing: Coverage

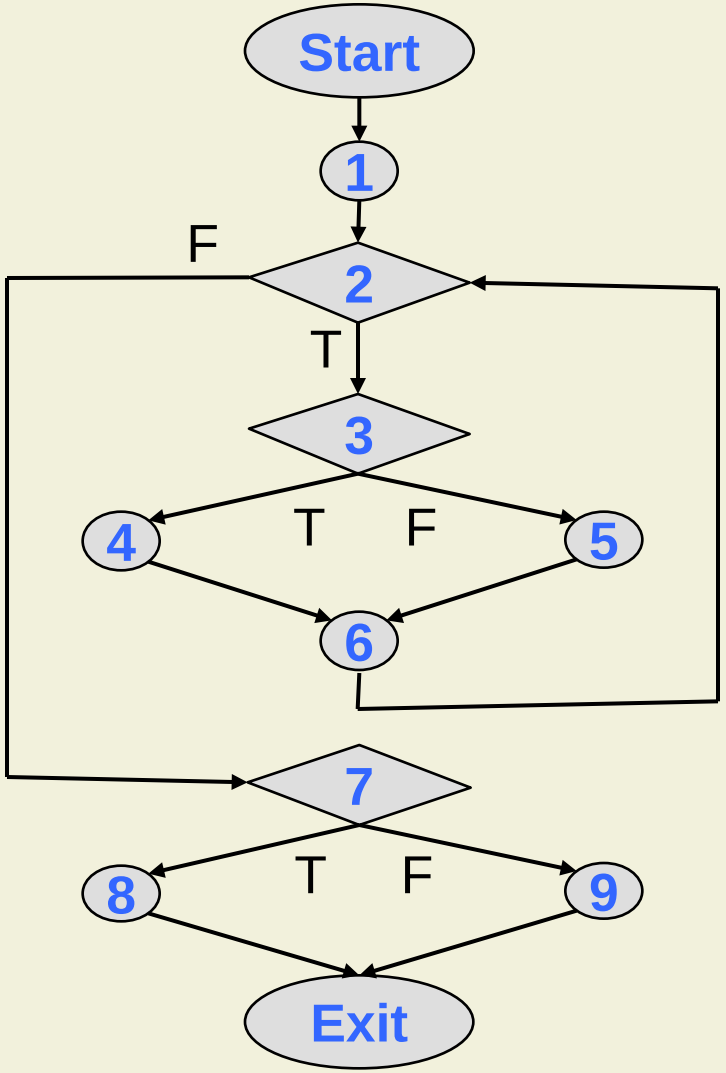
- Path Testing
  - Execute each possible path
  - Not practical with many nested conditionals
  - Impossible for most loops
- Branch Testing
  - Test each possible outcome from a condition
- Loop Testing
  - Cause execution of the loop to be skipped completely
  - Execute loop exactly once
  - Execute loop more than once

White-Box Testing: Example

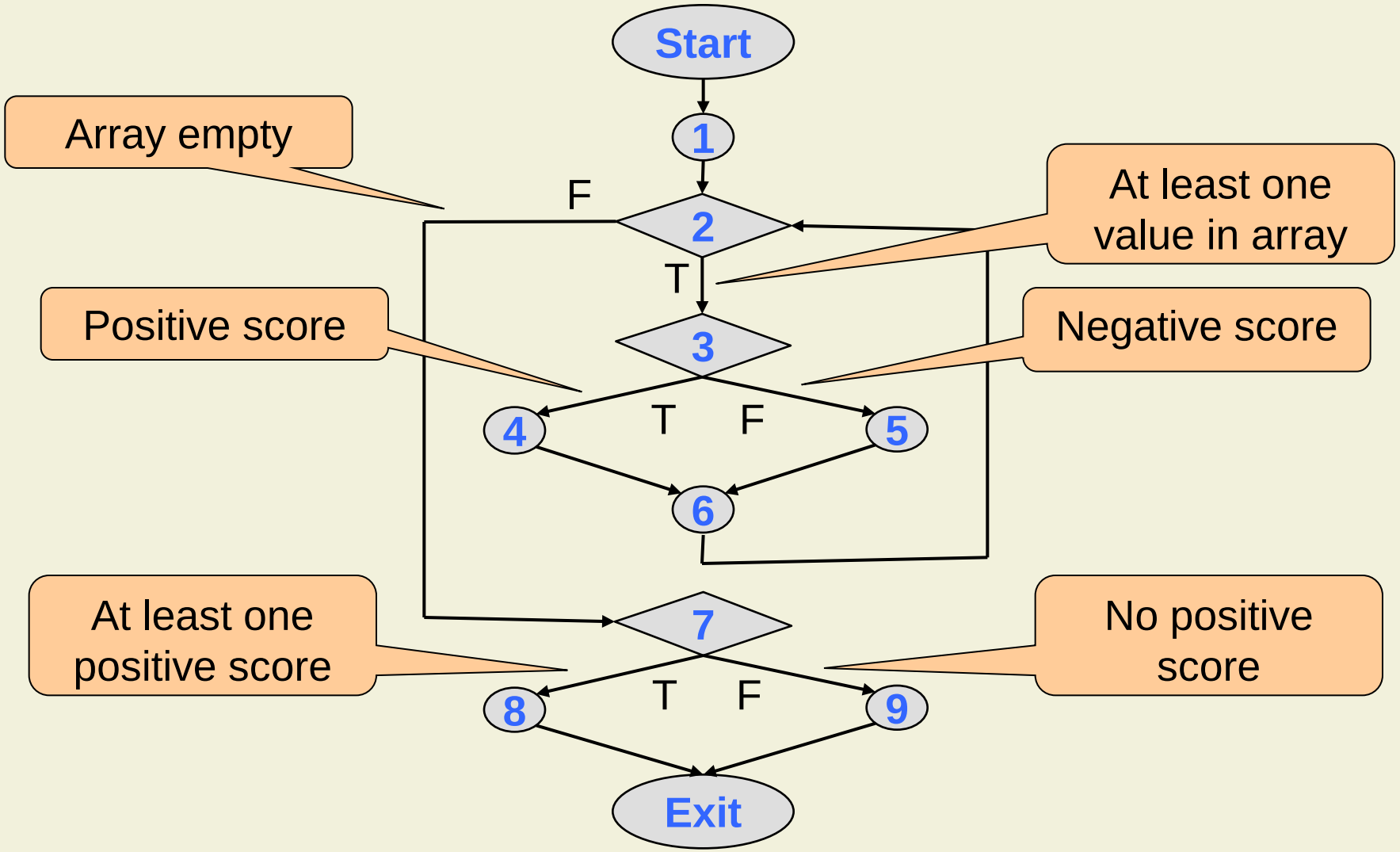
```
void printMean( float[ ] scores ) {
  float sum = 0.0;  int number = 0;
  for( int i = 0; i < scores.length; i++ ) {
    if( scores[ i ] > 0.0 ) {
      sum += scores[ i ];  number++;
    }
  }
  if( number > 0 )
    System.out.println( “The mean is: “ + sum / number );
  else
    System.out.println( “No scores found” );
}
```

White-Box Testing: Logic Flow Diagram

```
void printMean( float[ ] scores ) {
  float sum = 0.0;  int number = 0;                    1
2 for( int i = 0; i < scores.length; i++ ) {           6
3   if( scores[ i ] > 0.0 ) {
      sum += scores[ i ];  number++;                   4
    } 5
  }
7 if( number > 0 )
    System.out.println( … );                           8
  else
    System.out.println( … );                           9
}
```

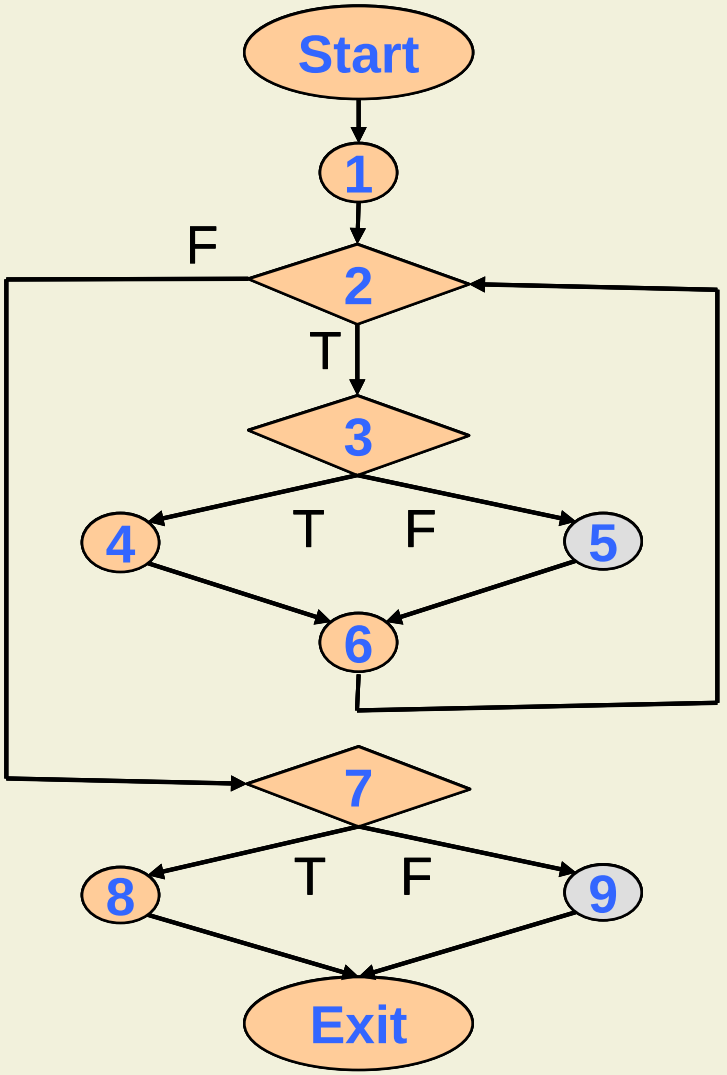


White-Box Testing: Finding the Test Cases



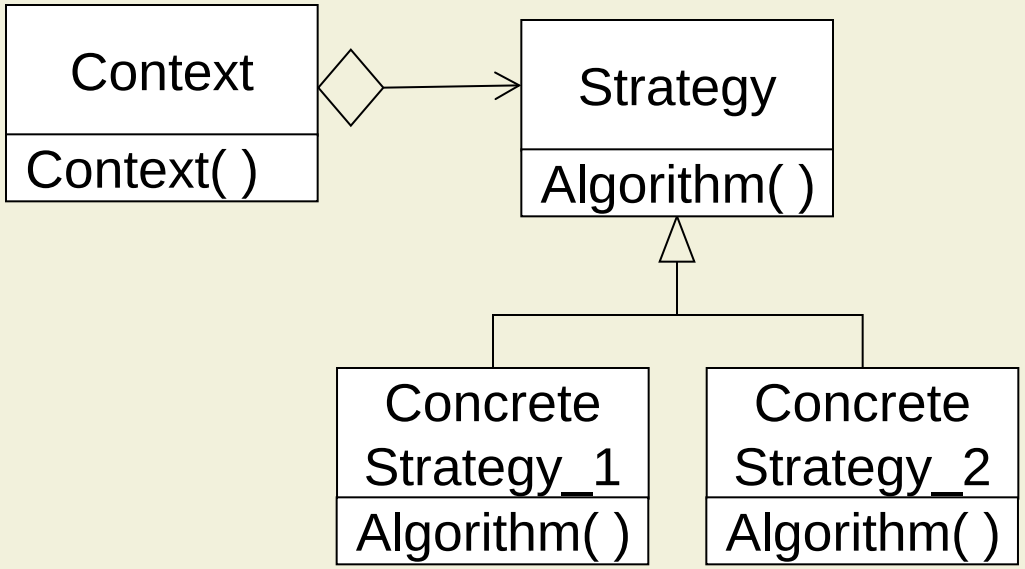
White-Box Testing: Test Cases

- Test case 1: skip loop body
  - Test data: [ ]
- Test case 2: execute loop exactly once
  - Test data: [-1]
- Test case 3: execute loop more than once
  - Test data: [3, 2]



Dynamic Method Binding in Path Testing

- Dynamic method binding requires more test cases
- Analogous to conditional over type information
- Execute each method implementation



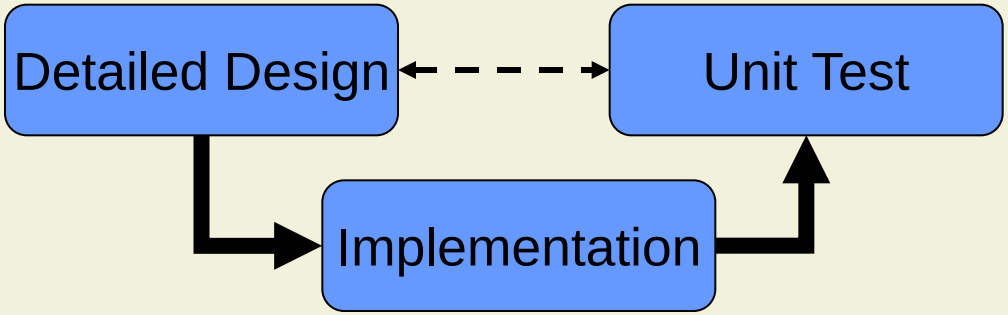
White-Box Versus Black-Box Testing

- Both types of testing are needed

- Black-box testing
  - Potential combinatorial explosion of test cases (valid and invalid data)
  - Cannot detect extraneous use cases ("features")
- White-box testing
  - Potentially infinite number of paths
  - Often tests what is done, instead of what should be done
  - Cannot detect missing use cases

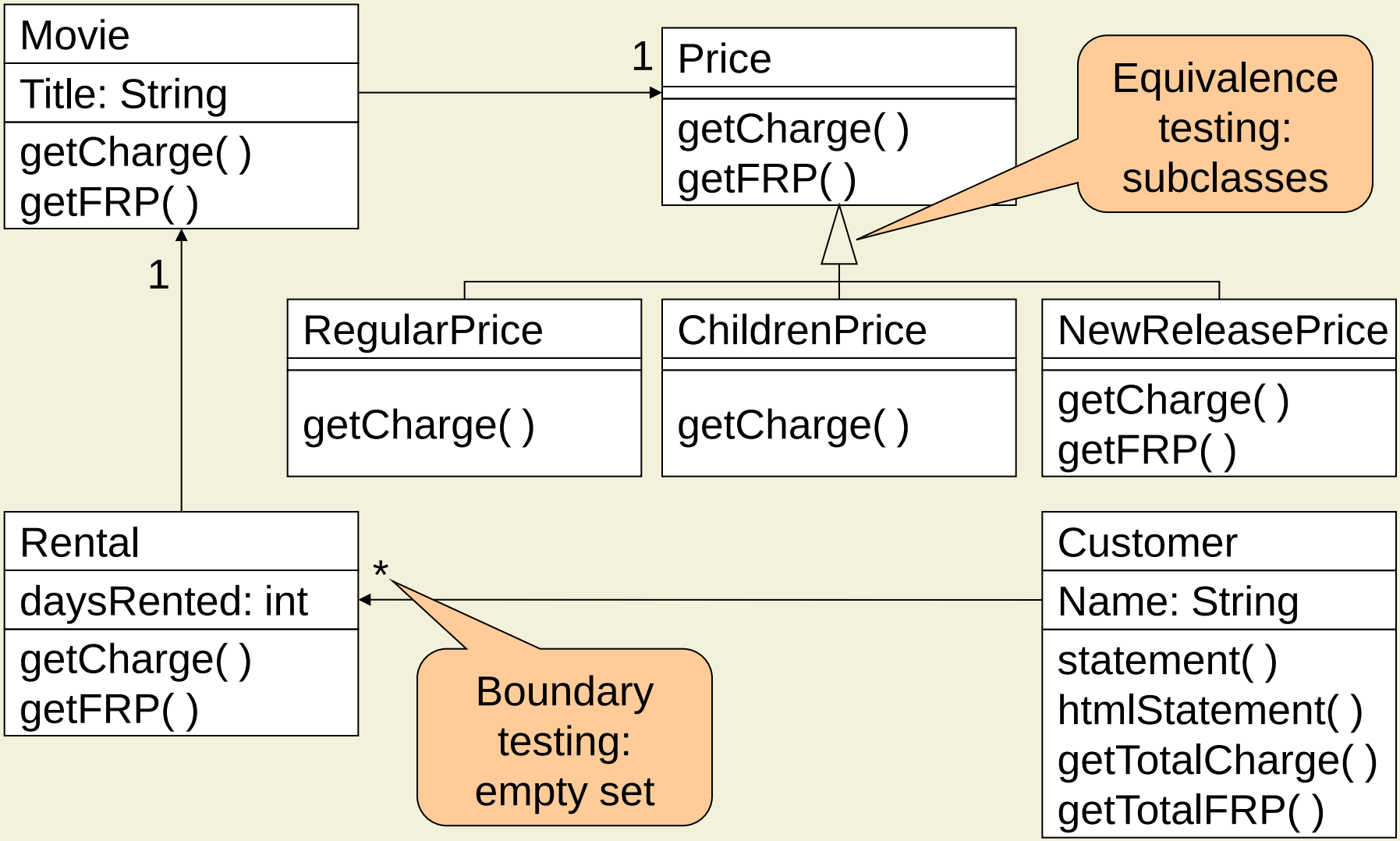
7. Testing

Creation of Unit Tests

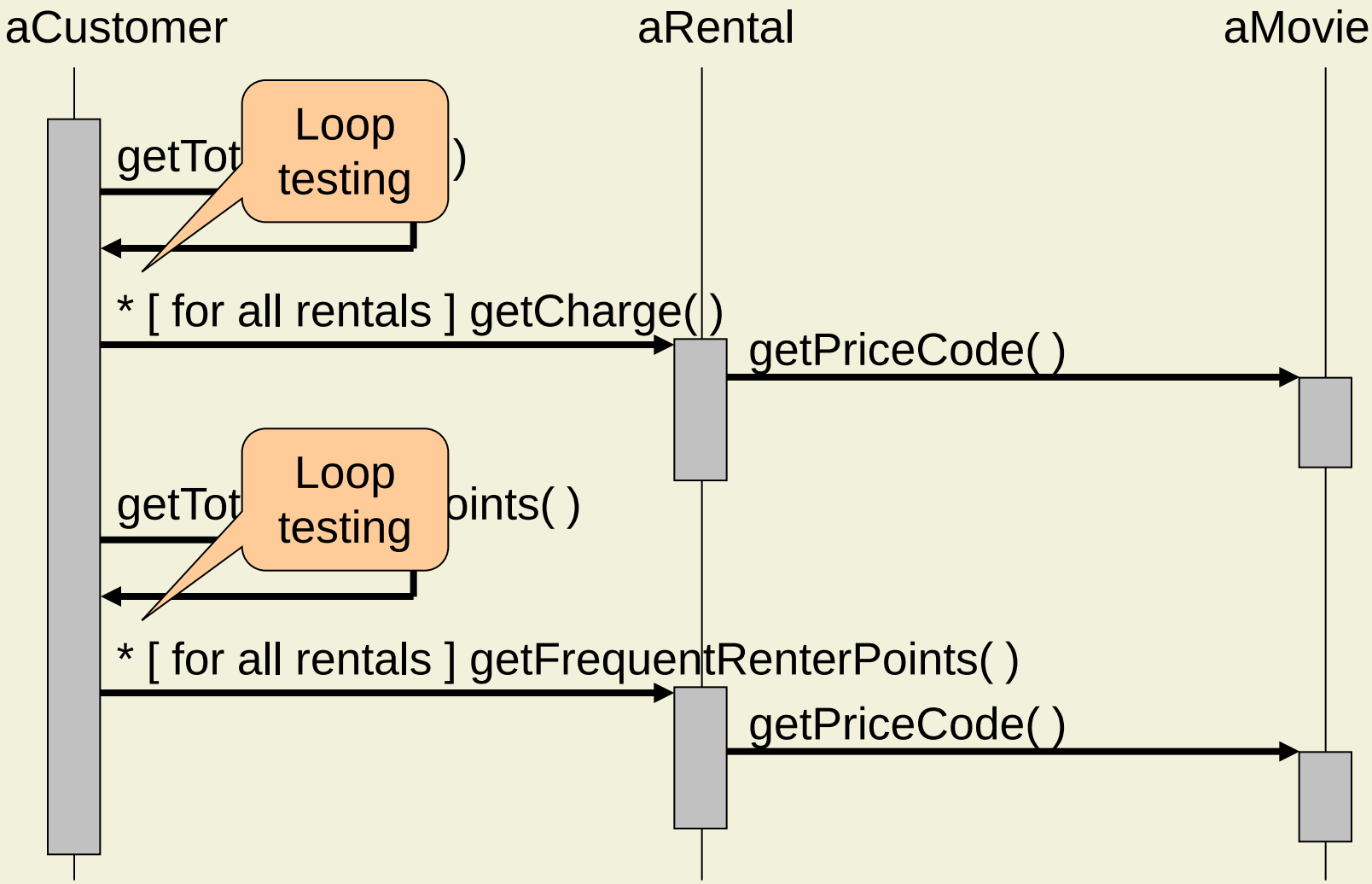


- Create tests as soon as detailed design is completed
  - Black-box test: Test functional requirements
  - White-box test: Test the dynamic model
  - Data-structure test: Test the object model
- Find the minimal number of test cases to cover as many paths as possible
  - Cross-check the test cases to eliminate duplicates

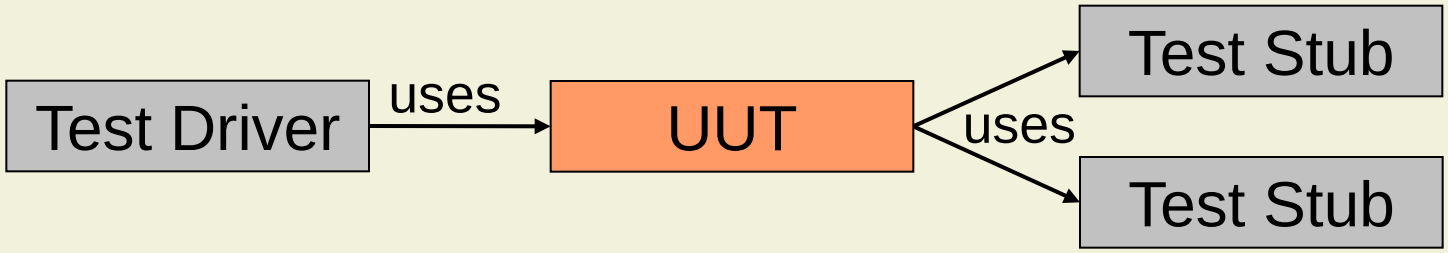
Creation of Unit Tests: Movie Rental Example



Creation of Unit Tests: Example (cont'd)



Creation of Test Harness



- Test driver
  - Class that applies test cases to UUT including setup and clean-up
  - Created automatically by JUnit using reflection (classes TestRunner and TestSuite)
- Test stub
  - Partial, temporary implementation of a component used by UUT

Creation of Test Oracle

- Test Oracle
  - Produces the results expected for a test case
- JUnit
  - Compares actual result and expected result using assert methods
  - Expected result is produced manually
- Specification-based testing
  - Uses contracts (postconditions) as test oracles
  - Limited by expressiveness of contract language

Test Execution

- Execute the test cases
- Re-execute test cases after every change
  - **Automate** as much as possible
  - For instance, after each refactoring

- Regression testing
  - Testing that everything that used to work **still works** after changes are made to the system
  - Also important for system testing

Eight Rules of Testing

1. Make sure all tests are **fully automatic** and check their own results
2. A **test suite** is a powerful bug detector that reduces the time it takes to find bugs
3. **Run** your tests **frequently**– every test at least once a day
4. When you get a bug report, start by writing a **unit test that exposes the bug**
5. Better to write and run incomplete tests than not run complete tests
6. Concentrate your tests on **boundary conditions**
7. Do not forget to test **exceptions** raised when things are expected to go wrong
8. Do not let the fear that testing can't catch all bugs stop you from writing tests that will catch most bugs

[M. Fowler]

Complement: Code Reviews

- Form team of technical experts including the programmer
- Finds 70%-90% of bugs in studies
- Finds bugs earlier than testing
- Dramatically reduces cost of finding bugs
- Teaches everyone the code
- Variations
  - Walk-through (informal)
  - Code inspection (formal: records, metrics)

Complement: Static Analyses

- Code checkers (PMD, lint, PreFIX)
  - Find certain classes of errors
  - Easy to apply

- Program verifiers (ESC/Java, Boogie, SDV)
  - Typically cause significant overhead for programmers
  - First successful industrial applications in very specific areas (e.g., device drivers)

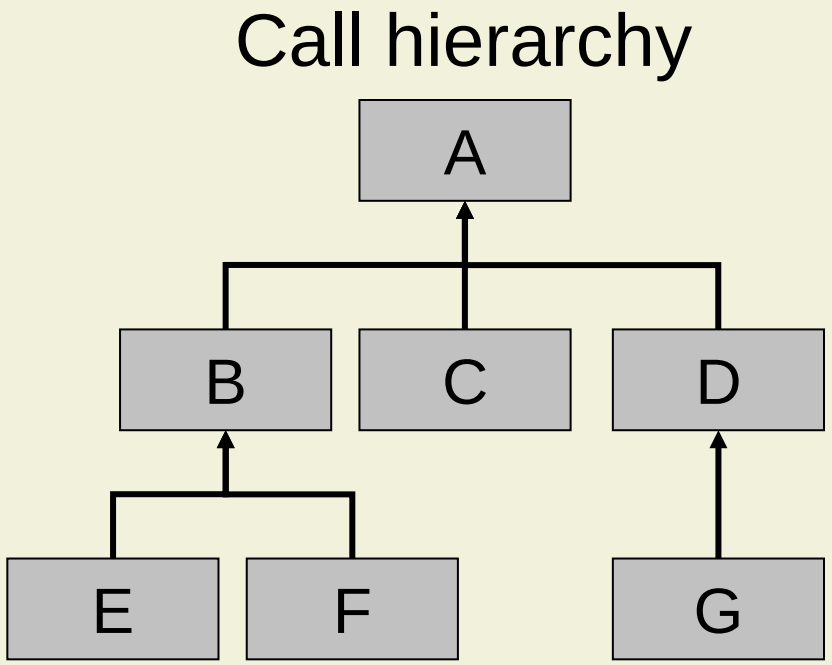
7. Testing

Steps in Integration-Testing

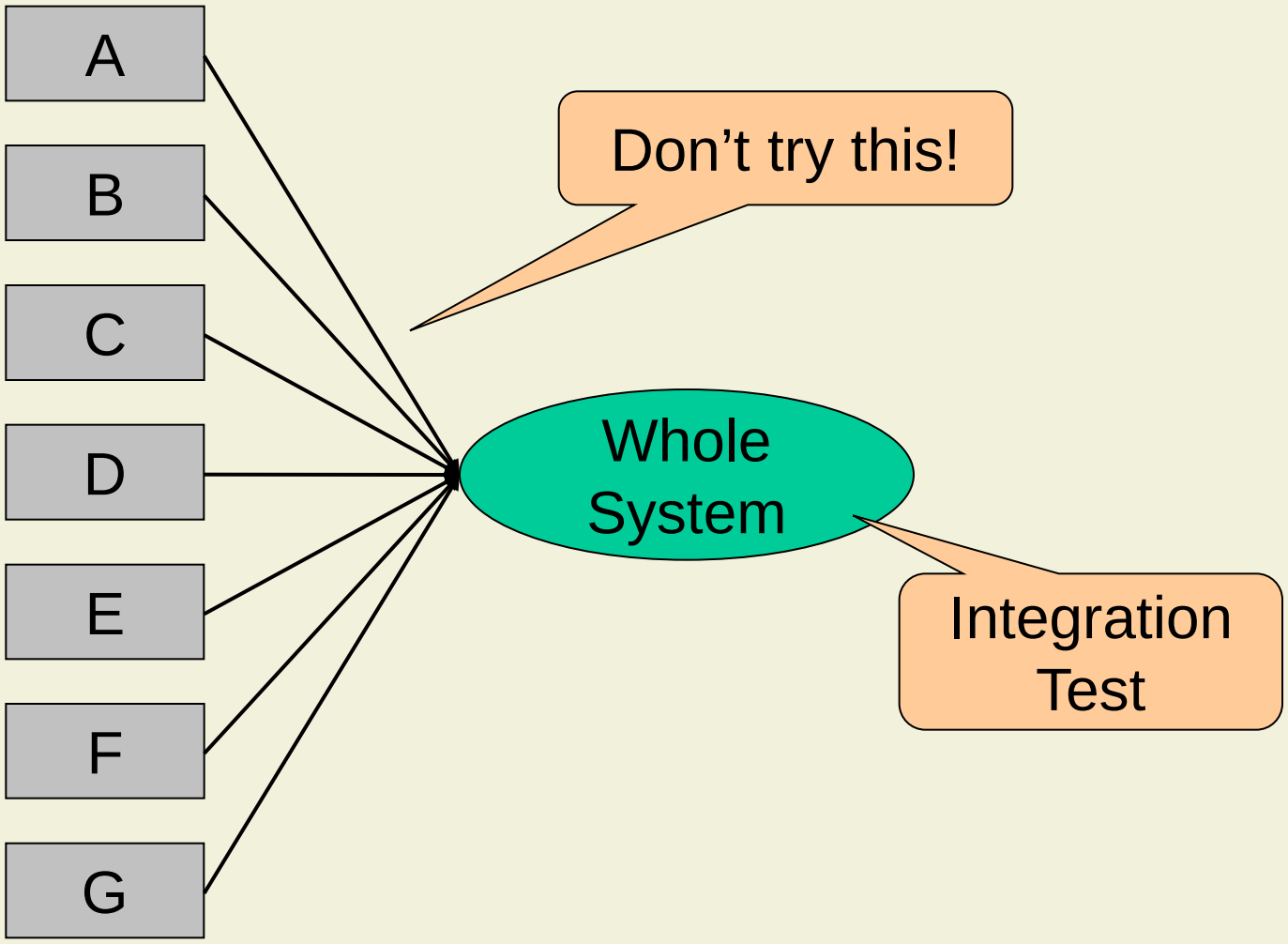
1. Select a component to be tested
   - Unit test all the classes in the component
2. Put selected components together
   - Make the integration test operational (drivers, stubs)
3. Do the testing
   - Functional testing, structural testing, performance testing
4. Keep records of the test cases and testing activities
5. Repeat steps 1 to 4 until the full system is tested

Integration Testing Strategy

- The order in which the subsystems are selected for testing and integration
- Typical strategies
  - Big-bang integration (Nonincremental)
  - Bottom-up integration
  - Top-down integration
  - Sandwich testing
  - Variations of the above



Big-Bang Strategy: Example



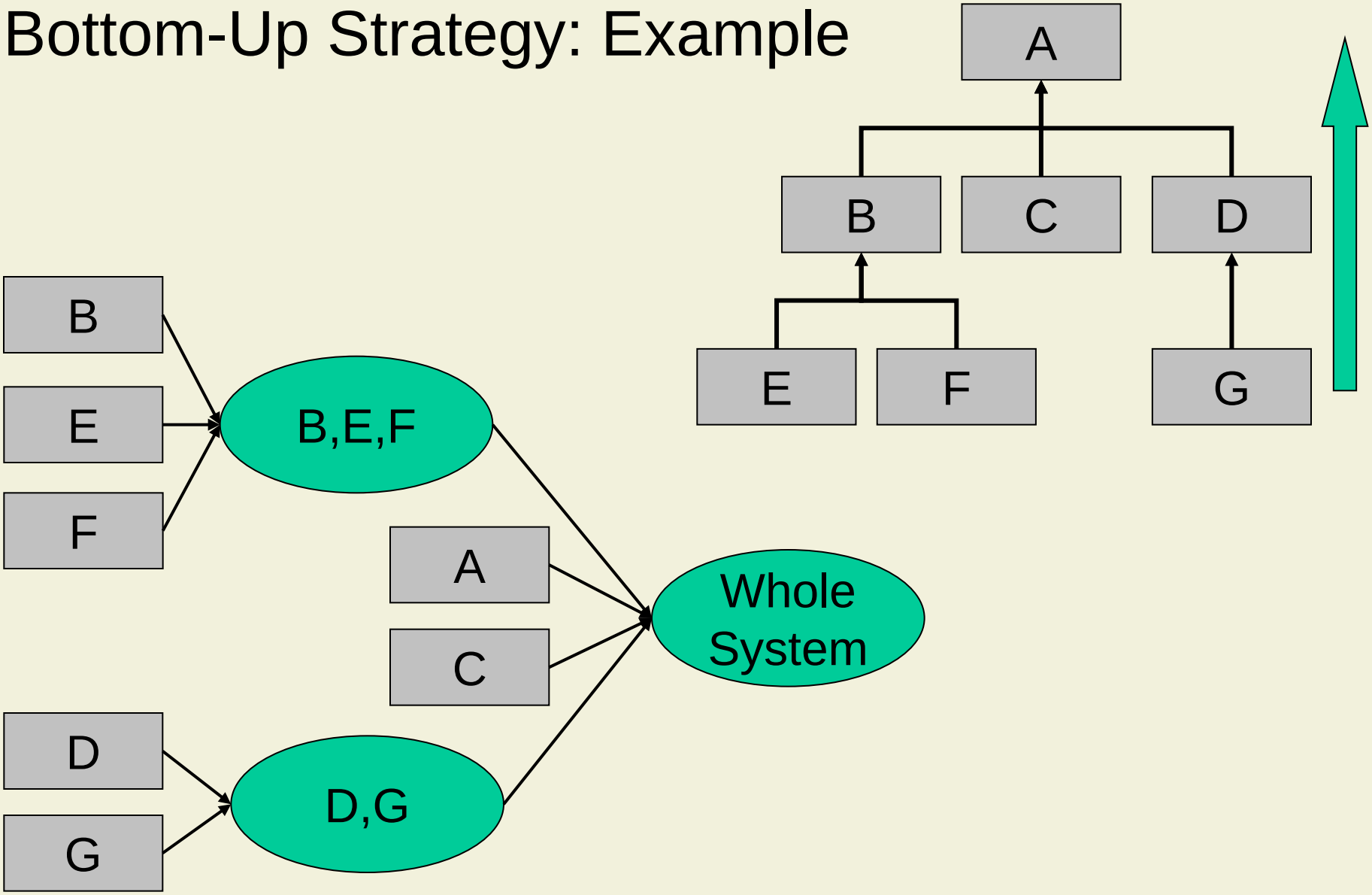
Bottom-Up Strategy

- Strategy
  1. Start with subsystems in lowest layer of call hierarchy
  2. Test subsystems that call the previously tested subsystems
  3. Repeat until all subsystems are included

- **Pros**
  - Useful for integrating object-oriented systems and systems with strict performance requirements

- **Cons**
  - Tests the most important subsystem (UI) last

Bottom-Up Strategy: Example

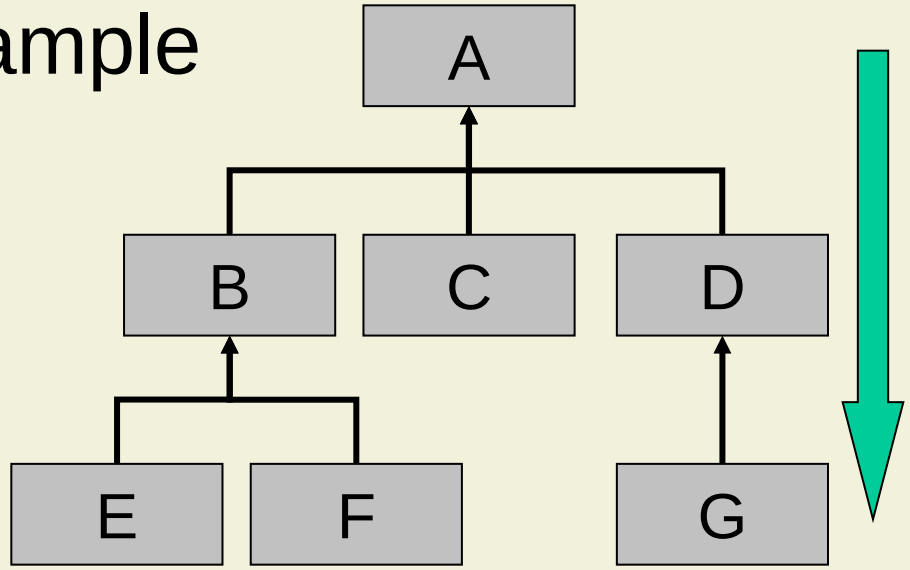


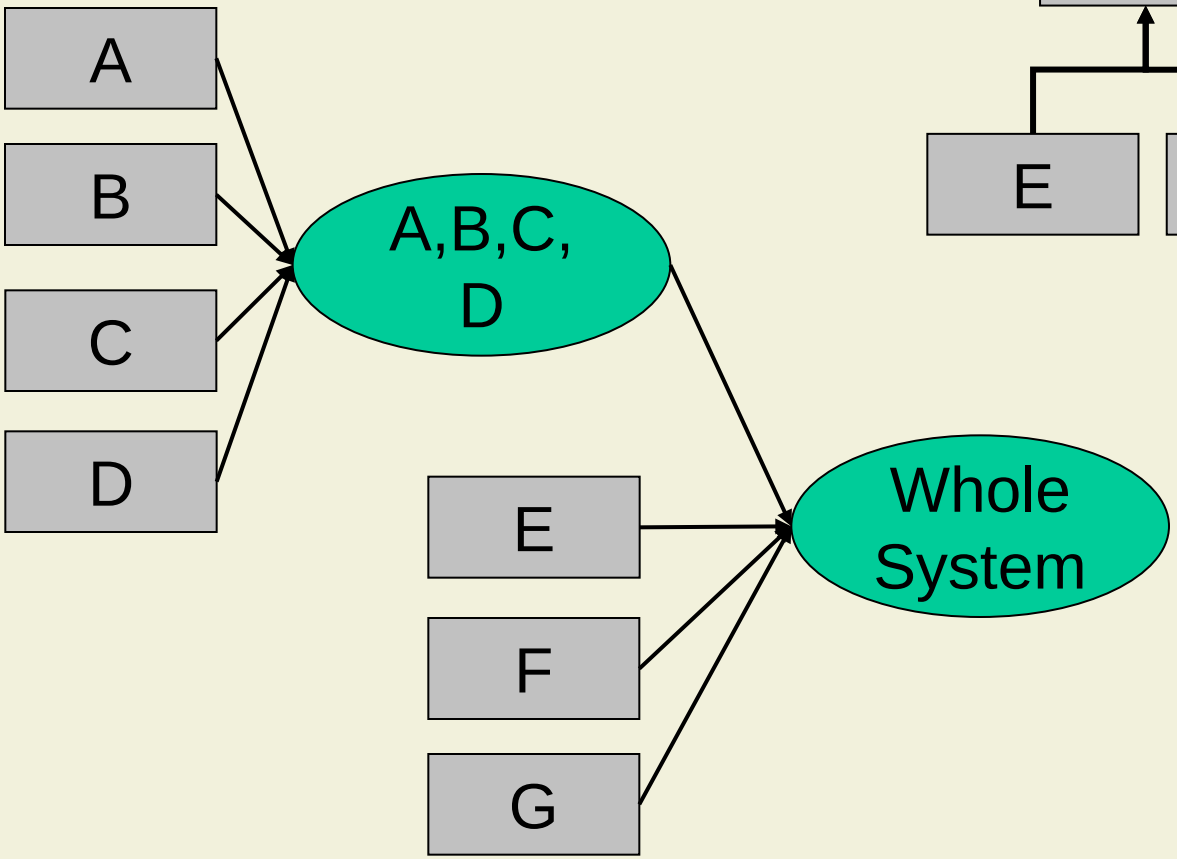
Top-Down Strategy

- Strategy
  1. Start with subsystems in top layer of call hierarchy
  2. Include subsystems that are called by the previously tested subsystems
  3. Repeat until all subsystems are included

- Requires test stubs
  - Simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data

Top-Down Strategy: Example





Top-Down Strategy: Discussion

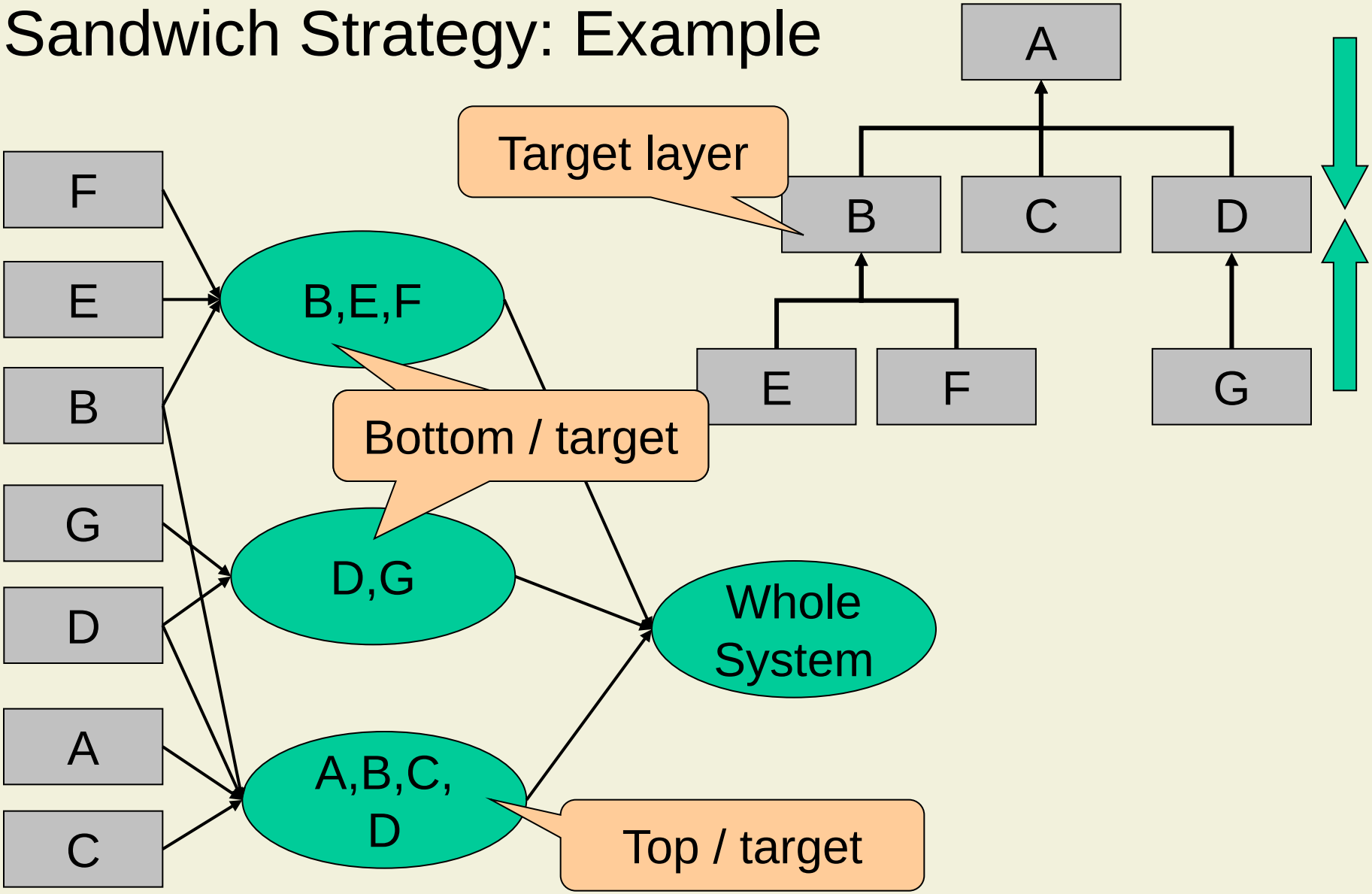
- **Pros**
  - Supports test cases for the functionality of the system

- **Cons**
  - Writing stubs can be difficult: Stubs must allow all possible conditions to be tested
  - Possibly very large number of stubs required

Sandwich Strategy

- Combines top-down with bottom-up strategy
- The system is view 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
- How do you select the target layer if there are more than 3 layers?
  - Try to minimize the number of stubs and drivers

Sandwich Strategy: Example



Sandwich Strategy: Discussion

- **Pros**
  - Top and bottom layer can be tested in parallel
  - Fewer drivers and stubs needed (target layer instead of driver for bottom layer and stub for top layer)

- **Cons**
  - Does not test the individual subsystems thoroughly before integration

Modified Sandwich Strategy

- Test in parallel
  - Middle layer with drivers and stubs
  - Top layer with stubs
  - Bottom layer with drivers

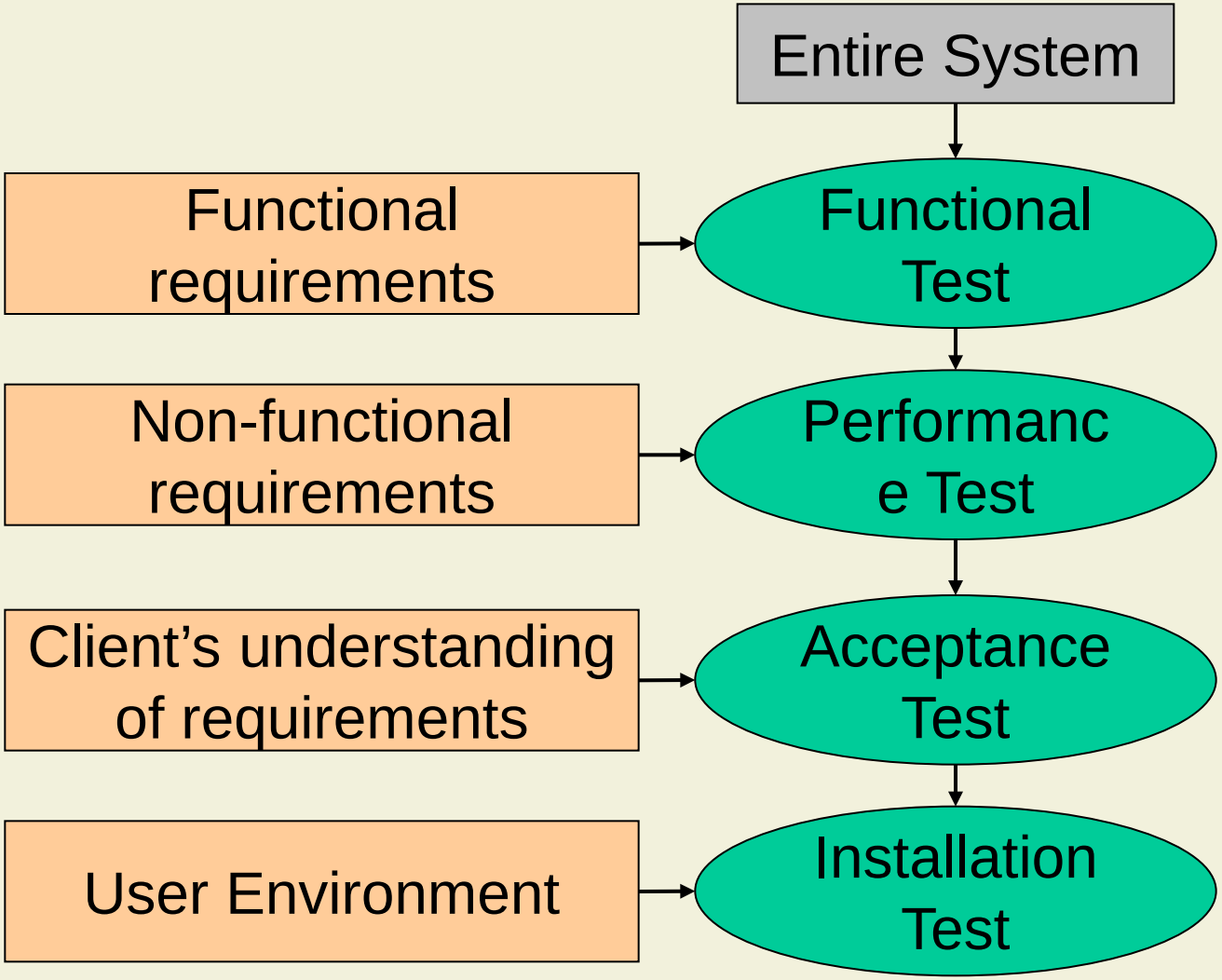
- Test in parallel
  - Top layer accessing middle layer (top layer replaces drivers)
  - Bottom accessed by middle layer (bottom layer replaces stubs)

Choosing an Integration Strategy

- Amount of test harness (stubs and drivers)
- Availability of hardware (e.g., parallelization)
- Scheduling concerns
  - Availability of components
  - Location of critical parts in the system

7. Testing

System Testing Stages



Functional Testing

- Goal: Test functionality of system
  - System is treated as black box
- Test cases are designed from requirements analysis document
  - Based on use cases
  - Alternative source: user manual
- Test cases describe
  - Input data
  - Flow of events
  - Results to check

Performance Testing

- Stress Testing
  - Stress limits of system (maximum number of users, peak demands)
- Volume testing
  - Large amounts of data
- Configuration testing
  - Various software and hardware configurations
- Compatibility testing
  - Backward compatibility with existing systems
- Security testing
  - Try to violate security requirements (“red team”)

Performance Testing (cont'd)

- Timing testing
  - Response times and time to perform a function
- Environmental test
  - Tolerances for heat, humidity, motion
- Quality testing
  - Reliability, maintainability, and availability
- Recovery testing
  - System's response to presence of errors or loss of data
- Usability testing
  - Tests user interface with user

Acceptance Testing

- Goal: Demonstrate that the system meets customer requirements and is ready to use
- Choice of tests is made by client
  - Many tests can be taken from integration testing
- Performed by the client, not by the developer

Acceptance Testing (cont'd)

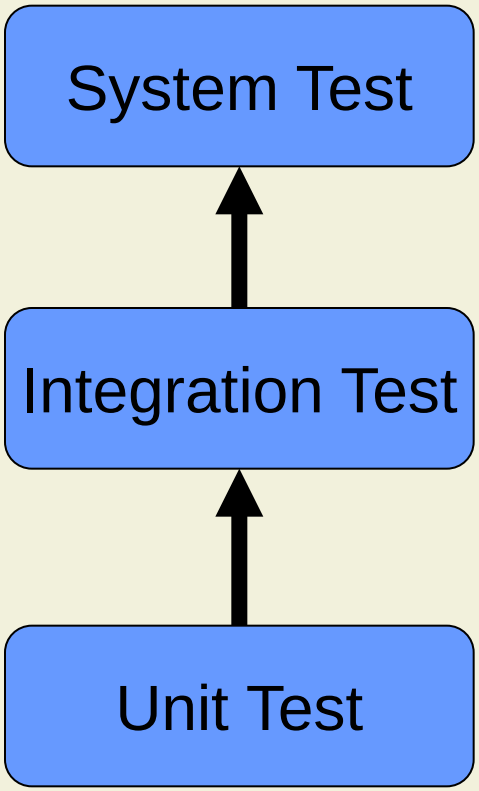
- Majority of bugs is typically found by the client, not by the developers or testers
- Alpha test
  - Client uses the software at the developer's site
  - Software used in a controlled setting, with the developer always ready to fix bugs
- Beta test
  - Conducted at client's site (developer is not present)
  - Software gets a realistic workout in target environment
  - Potential client might get discouraged

7. Testing

Independent Testing

- Programmers have a hard time believing they made a mistake
  - Plus a vested interest in not finding mistakes
  - Often stick to the data that makes the program work
- Designing and programming are constructive tasks
  - Testers must seek to break the software
- Testing is done best by independent testers

Independent Testing: Responsibilities



- Performed by independent test team
  - Exception: Acceptance test performed by client
- Performed by independent test team
- Performed by programmer
  - Requires detailed knowledge of the code
  - Immediate bug fixing

Independent Testing: Wrong Conclusions

- The developer should not be testing at all
  - “Test before you code”
- Testers get only involved once software is done
- Toss the software over the wall for testing
  - Testers and developers collaborate in developing the test suite
- Testing team is responsible for assuring quality
  - Quality is assured by a good software process

When to Stop Testing?

- In practice, typically determined by budget and schedule constraints

- White-box testing
  - Achieved coverage

- Black-box testing
  - High coverage difficult to achieve
  - Always perform at least boundary and regression testing

Fault Seeding

- Test team 1 inserts faults (errors) into the program
- Test team 2 performs the test
- Assumption

$$\frac{\text{detected seeded faults}}{\text{total seeded faults}} = \frac{\text{detected non-seeded faults}}{\text{total non-seeded faults}}$$

- Problem: Difficult to make seeded faults representative of the real ones
- Conclusion: most useful for testing systems that are similar to ones we have built before

Summary

- Main objective
  - Design tests that systematically uncover different classes of errors with a minimum amount of time and effort
  - A good test has a high probability of finding an error
  - A successful test uncovers an error
- Secondary benefits
  - Demonstrate that software appears to be working according to specification (functional and non-functional)
  - Data collected during testing provides indication of software reliability and software quality
  - Good testers clarify the specification (creative work)