Software Architecture and Engineering

Testing Basics

Peter Müller

Chair of Programming Methodology

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

Spring Semester 12

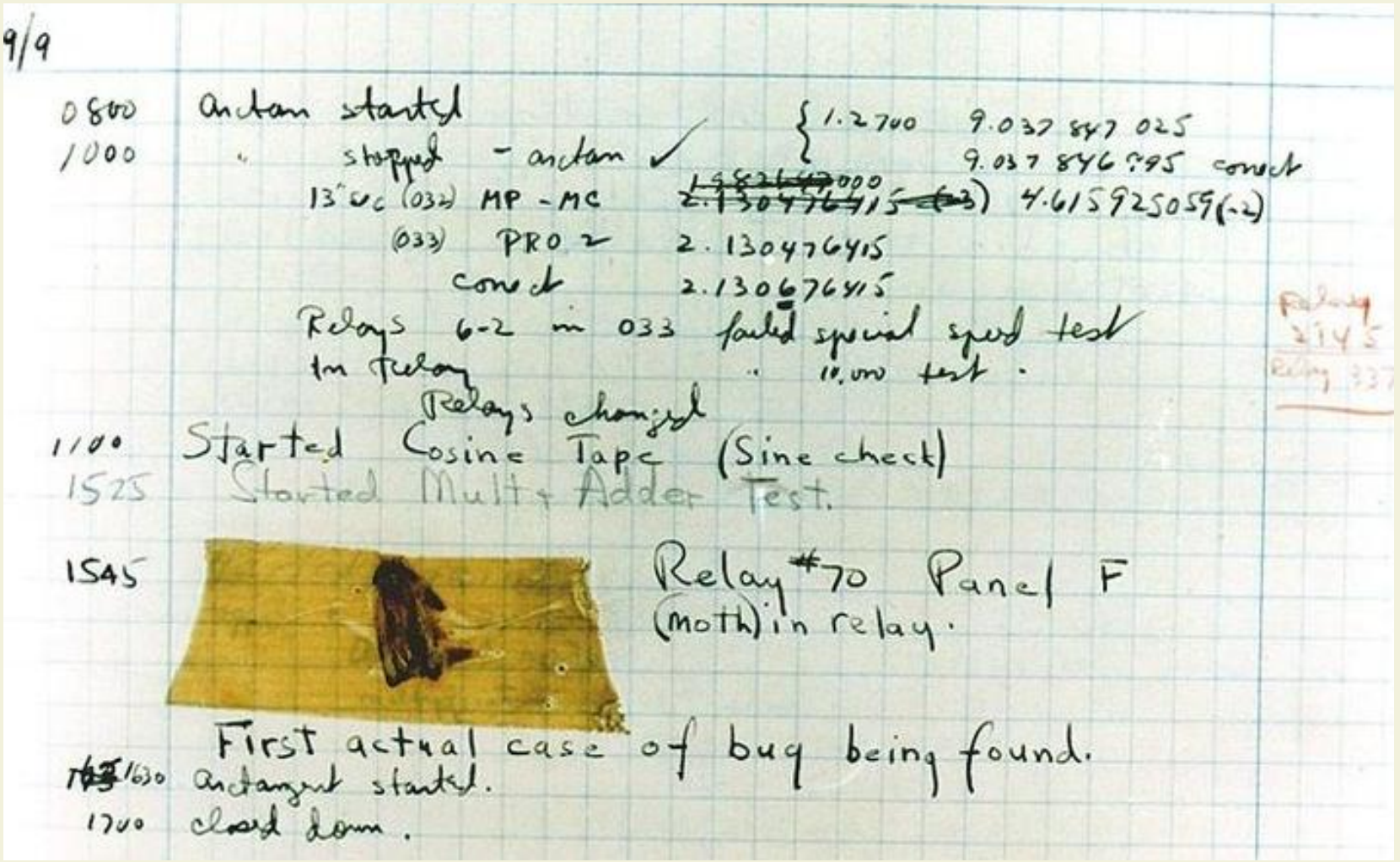
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

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.”



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]

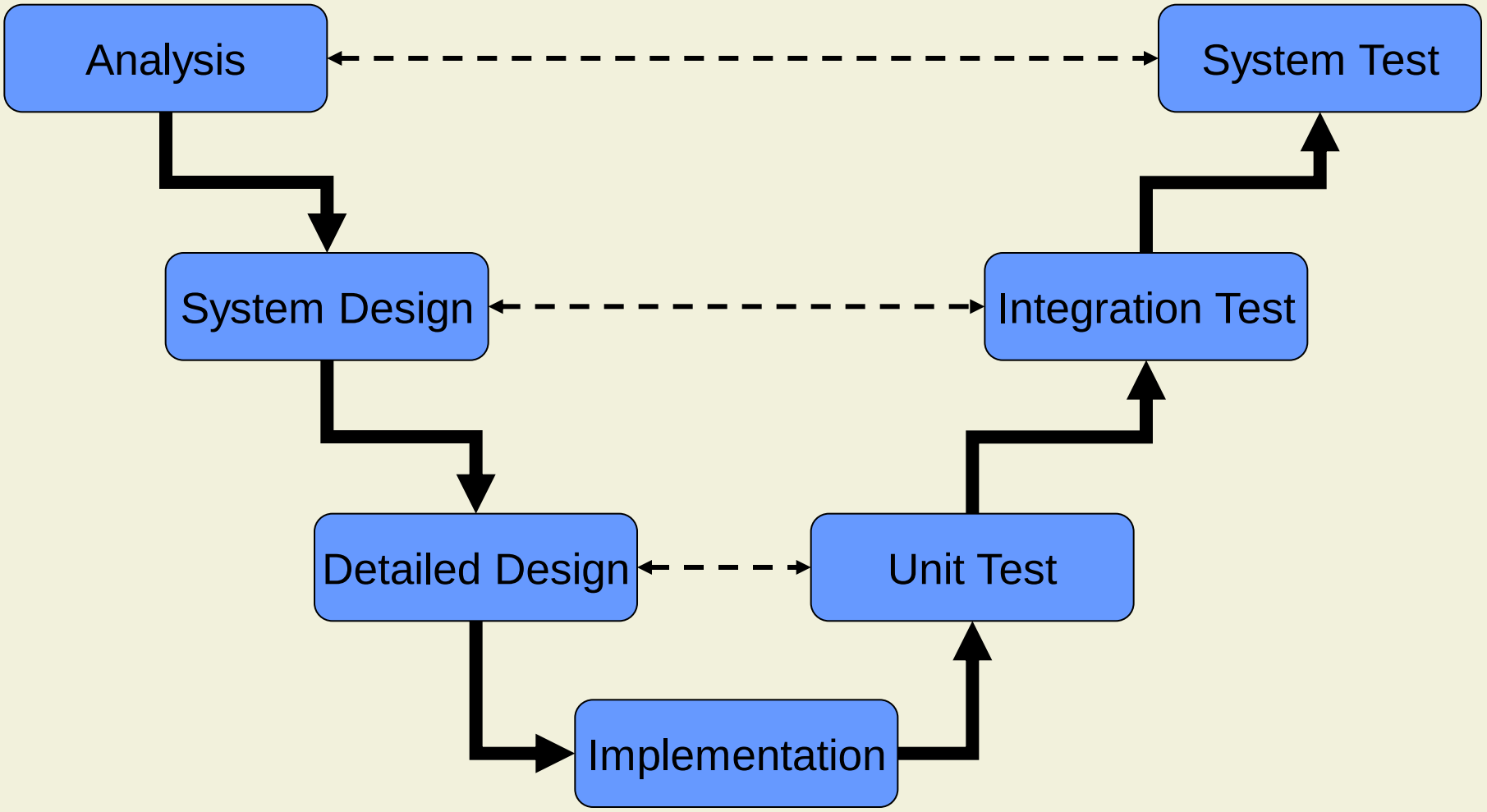
- It is impossible to completely test any nontrivial module or any system
  - Theoretical limitations: termination
  - Practical limitations: prohibitive in time and cost

7. Testing Basics Basics

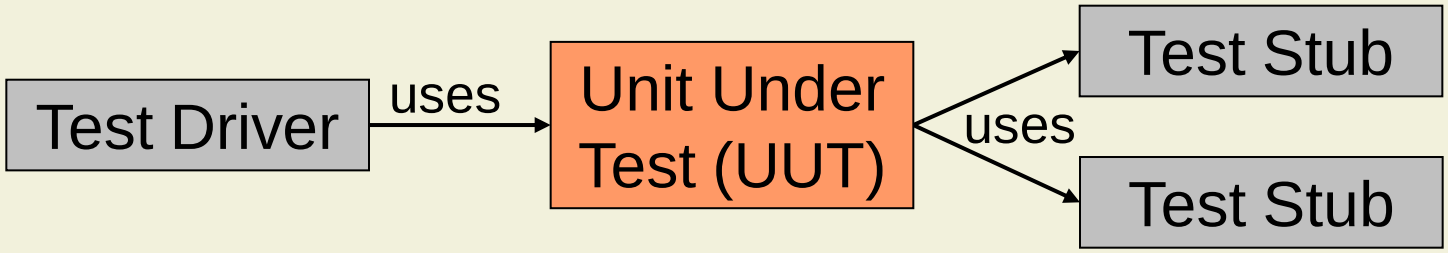
7.1 Test Stages

7.2 Test Strategies

Test Stages


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

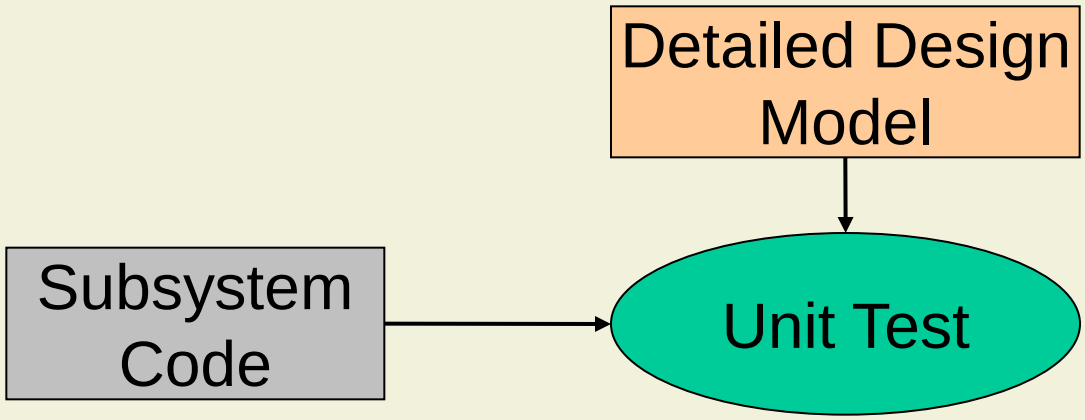
Creation of Test Harness



- Test driver
  - Applies test cases to UUT including setup and clean-up
- Test stub
  - Partial, temporary implementation of a component used by UUT
  - Simulates the activity of a missing component by answering to the calling sequence of the UUT and returning back fake data

Unit Testing

- Testing individual subsystems (collection of classes)



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

Unit Test Example (JUnit)

```
class SavingsAccount {
  …
  public void deposit( int amount ) { … }
  public void withdraw( int amount ) { … }
  public int getBalance( ) { … }
}
```

```
@Test
public void withdrawTest( ) {
  SavingsAccount target = new SavingsAccount();
  target.deposit( 300 );
  int amount = 100;
  target.withdraw( amount );
  Assert.assertTrue( target.getBalance( ) == 200 );
}
```

Implement test driver

Create test data

Create test oracle

Unit Testing: Discussion

- To achieve a reasonable test coverage, one has to test each method with several inputs
  - To cover valid and invalid inputs
  - To cover different paths through the method

```
@Test
public void withdrawTest( ) {
  SavingsAccount target = new SavingsAccount();
  target.deposit( 500 );
  int amount = 0;
  target.withdraw( amount );
  Assert.assertTrue( target.getBalance( ) == 500 );
}
```

Boiler-plate code for creating test data and writing test oracles

Parameterized Unit Tests (NUnit)

- Parameterized test methods take arguments for test data
  - Decouple test driver (logic) from test data

```
[ Test ]
public void withdrawTest( int balance, int amount ) {
  SavingsAccount target = new SavingsAccount();
  target.deposit( balance );
  target.withdraw( amount );
  Assert.IsTrue( target.getBalance( ) == balance – amount );
}
```

- Test data can be specified as values, ranges, or random values
- Requires generic test oracles

Generic Test Oracles: Example

```
public static void bubbleSort( int[ ] a ) {
  for( int i = 0; i < a.Length - 1; i++ ) {
    for( int j = i + 1; j < a.Length; j++ ) {
      if( a[ i ] > a[ j ] )
      { int tmp = a[ i ]; a[ i ] = a[ j ]; a[ j ] = tmp; }
    }
  }
}
```

```
[ Test ]
public void bubbleSortTest( ) {
  int[ ] a = { 7, 2, 5, 2 };

  bubbleSort( a );

  int[ ] expected = { 2, 2, 5, 7 };
  Assert.AreEqual( expected, a );
}
```

Create test data

Create test oracle

Generic Test Oracles: Example

```
[ Test ]
public void bubbleSortTest( int[ ] a ) {
  int[ ] original = ( int[ ] ) a.Clone();

  bubbleSort( a );

  for( int i = 0; i < a.Length - 1; i++ )
    Assert.IsTrue( a[ i ] <= a[ i+1 ] );

  bool[ ] visited = new bool[ a.Length ];
  for( int i = 0; i < a.Length; i++ ) {
    int j;
    for ( j = 0; j < a.Length; j++ ) {
      if( !visited[ j ] && a[ i ] == original[ j ] )
      { visited[ j ] = true; break; }
    }
    Assert.IsFalse( j == a.Length );
  }
}
```

Save test data for later comparison

Check that array is sorted

Check that array is a permutation of original array

Value a[i] is not in the original array

Parameterized Unit Tests: Discussion

- Parameterized unit tests **avoid boiler-plate** code
- Writing generic test oracles is sometimes difficult
  - Analogous to writing strong postconditions
- Still several test methods are needed, for instance, for valid and invalid input
- Parameterized unit tests are especially useful **when test data is generated automatically** (see later)

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

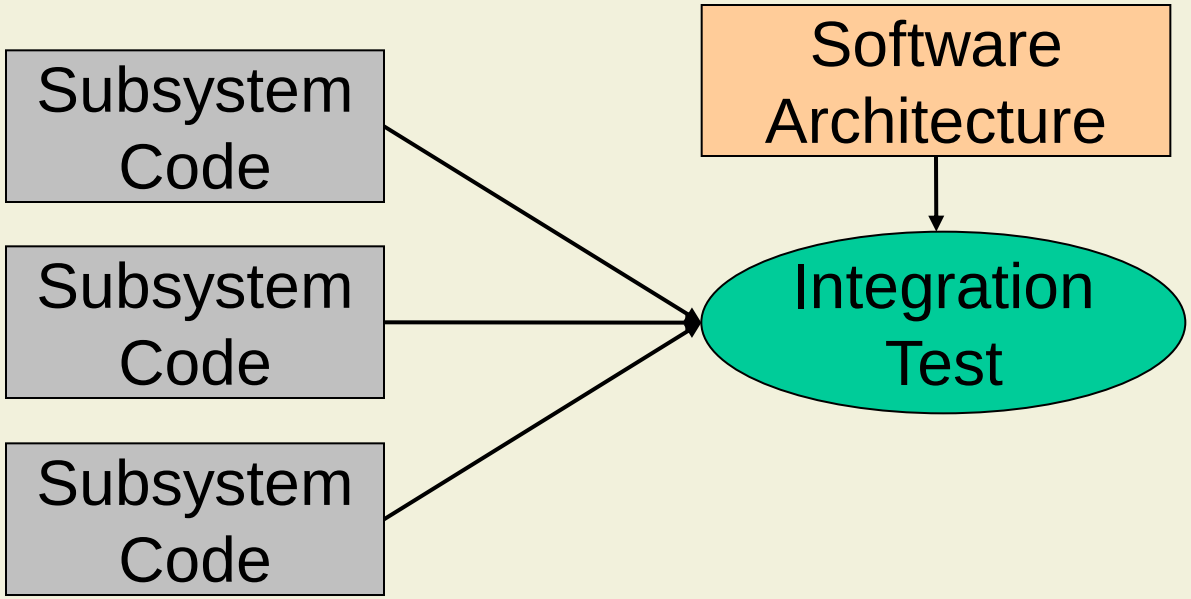
Eight Rules of Testing

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]

Integration Testing

- Testing groups of subsystems and eventually the entire system



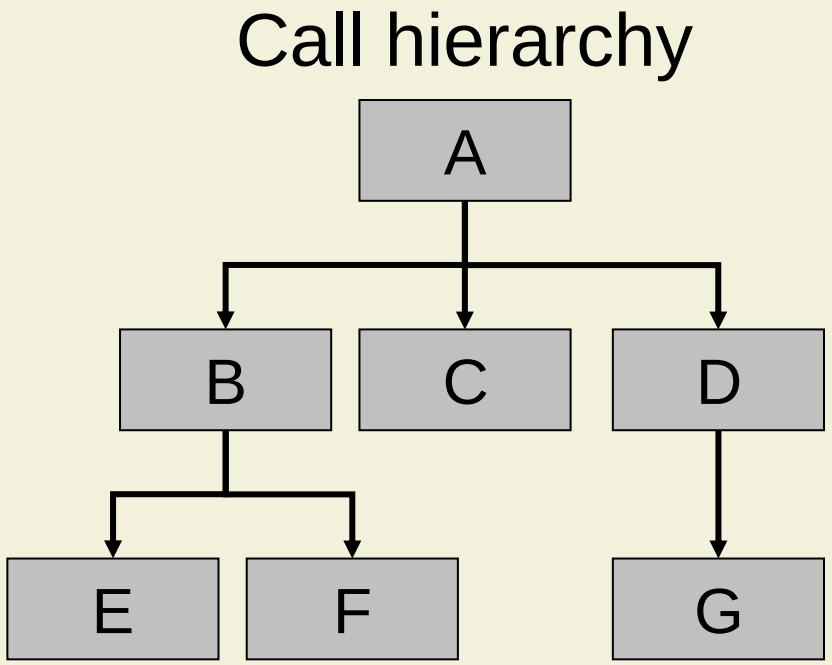
- Goal: Test interfaces between subsystems

Steps in Integration-Testing

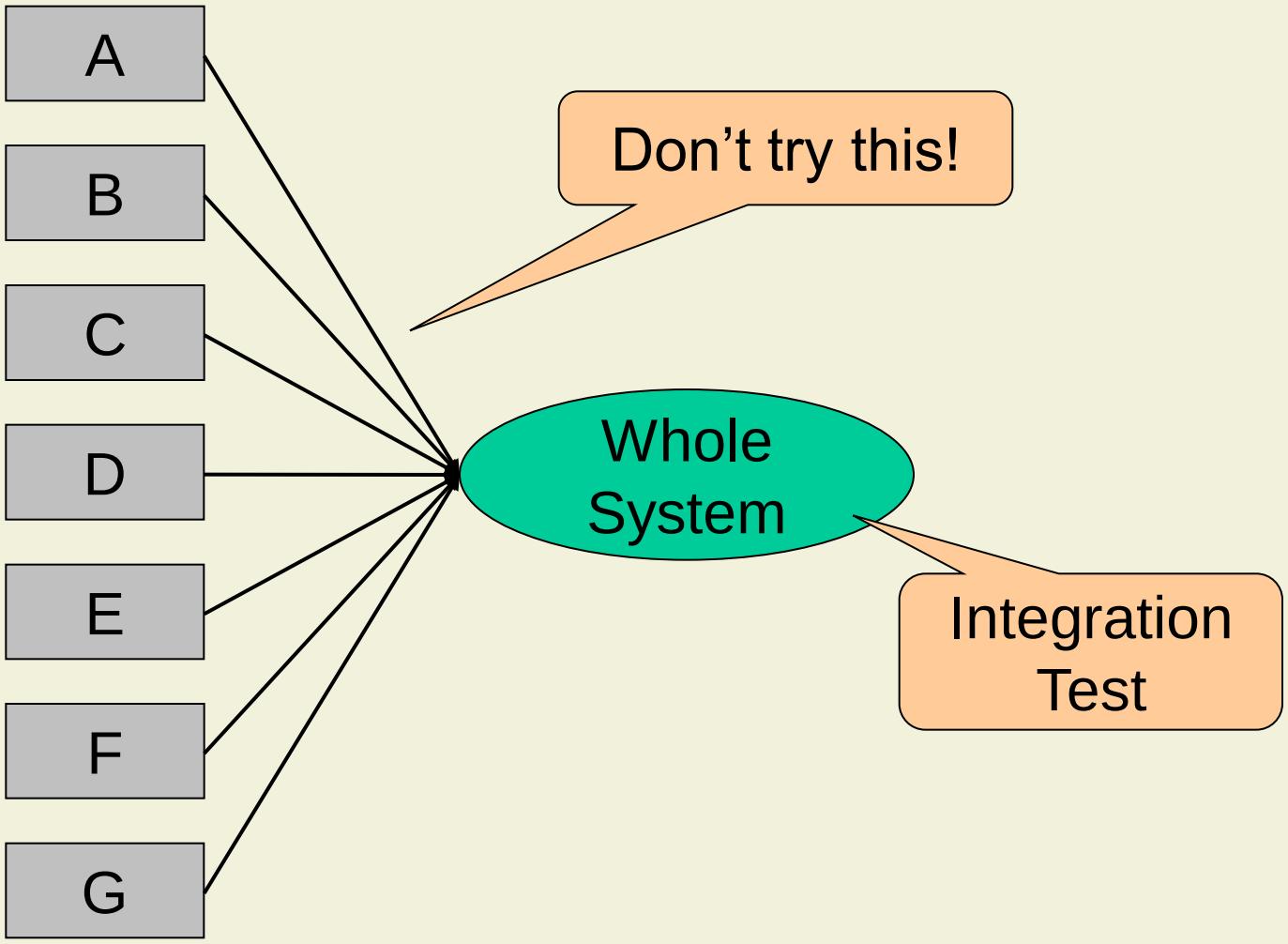
1. Select a component to be tested
   - Unit test all the classes in the component
2. Put selected components together
   - Make the integration test operational (drivers, stubs)
3. Do the testing
   - Functional testing, structural testing, performance testing
4. Keep records of the test cases and testing 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 (non-incremental)
  - Bottom-up integration
  - Top-down integration
  - Sandwich testing
  - Variations of the above



Big-Bang Strategy: Example



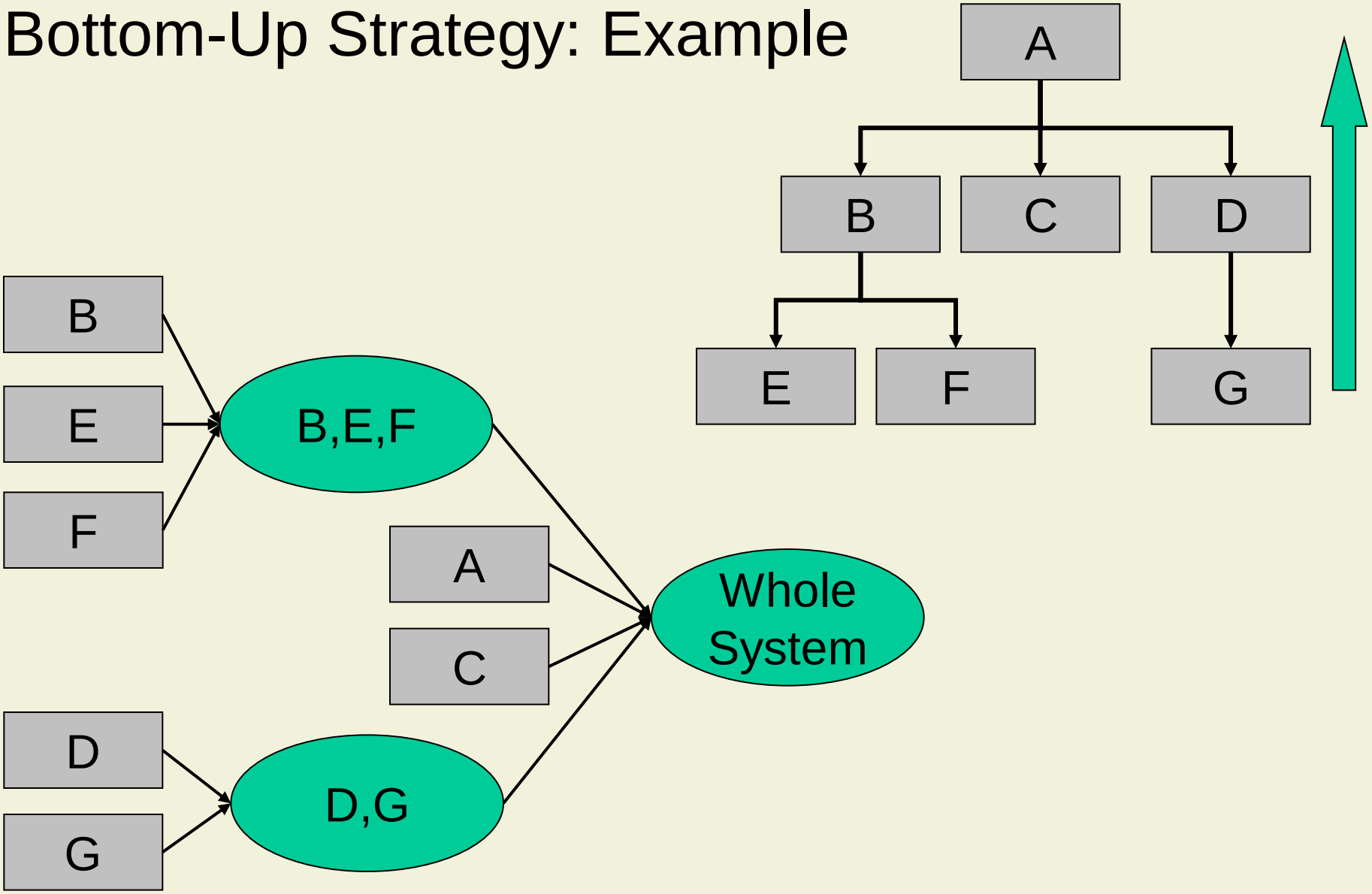
Bottom-Up Strategy

- 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



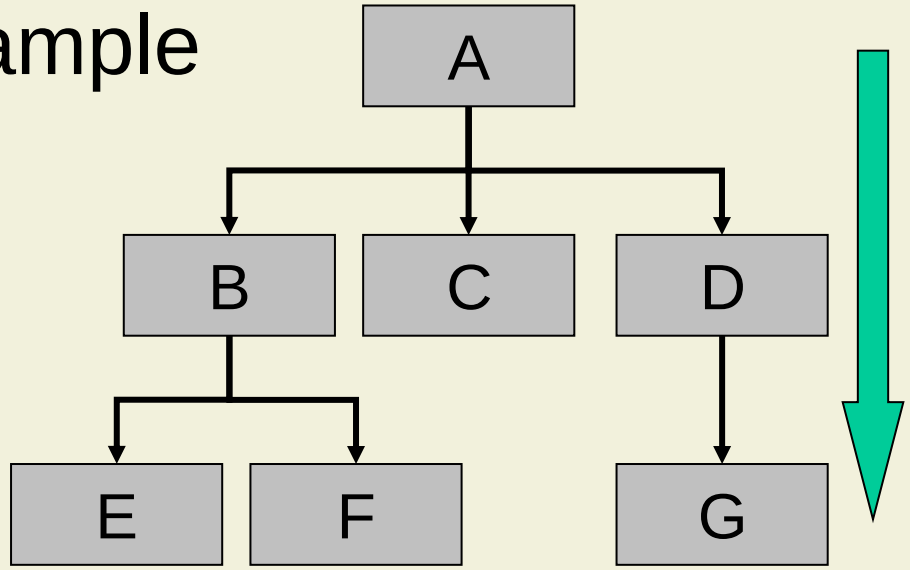
Top-Down Strategy

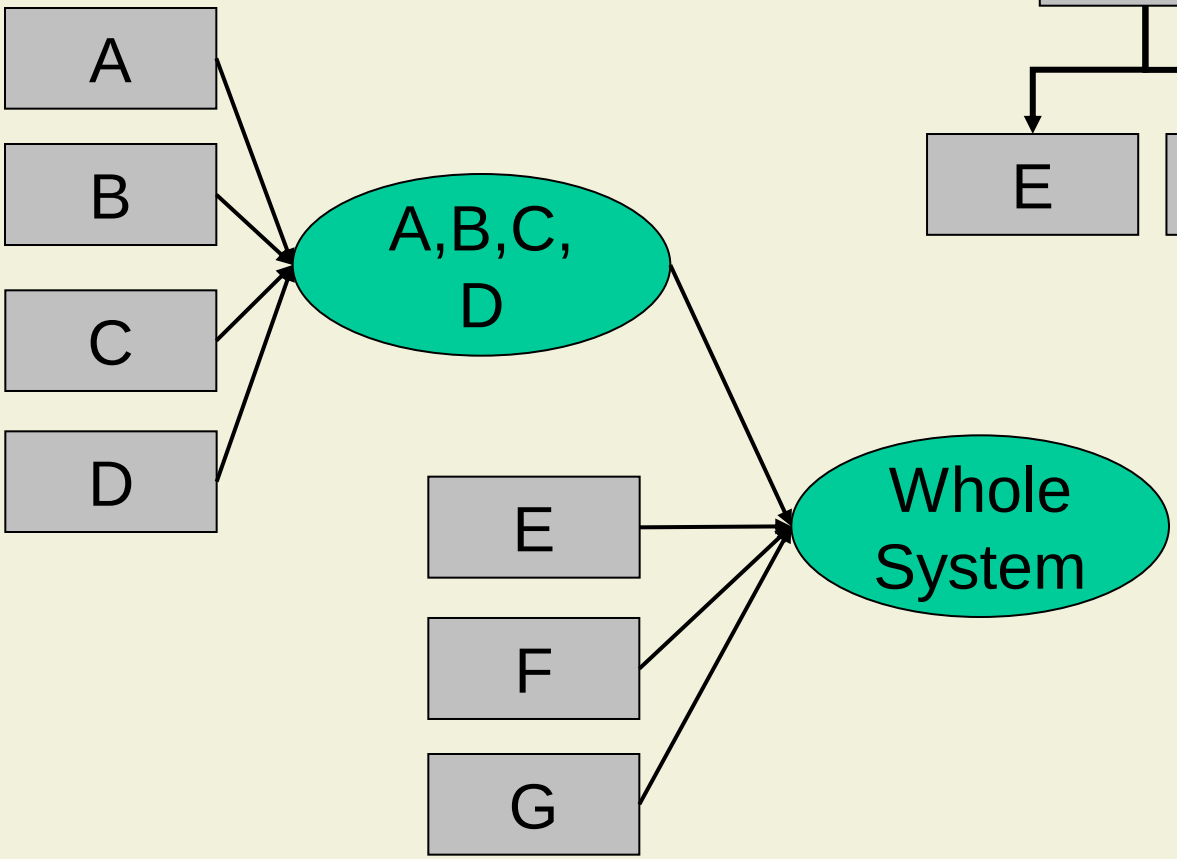
- Strategy
  1. Start with subsystems in top layer of call hierarchy
  2. Include subsystems that are called by the previously tested subsystems
  3. Repeat until all subsystems are included

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

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

Top-Down Strategy: Example

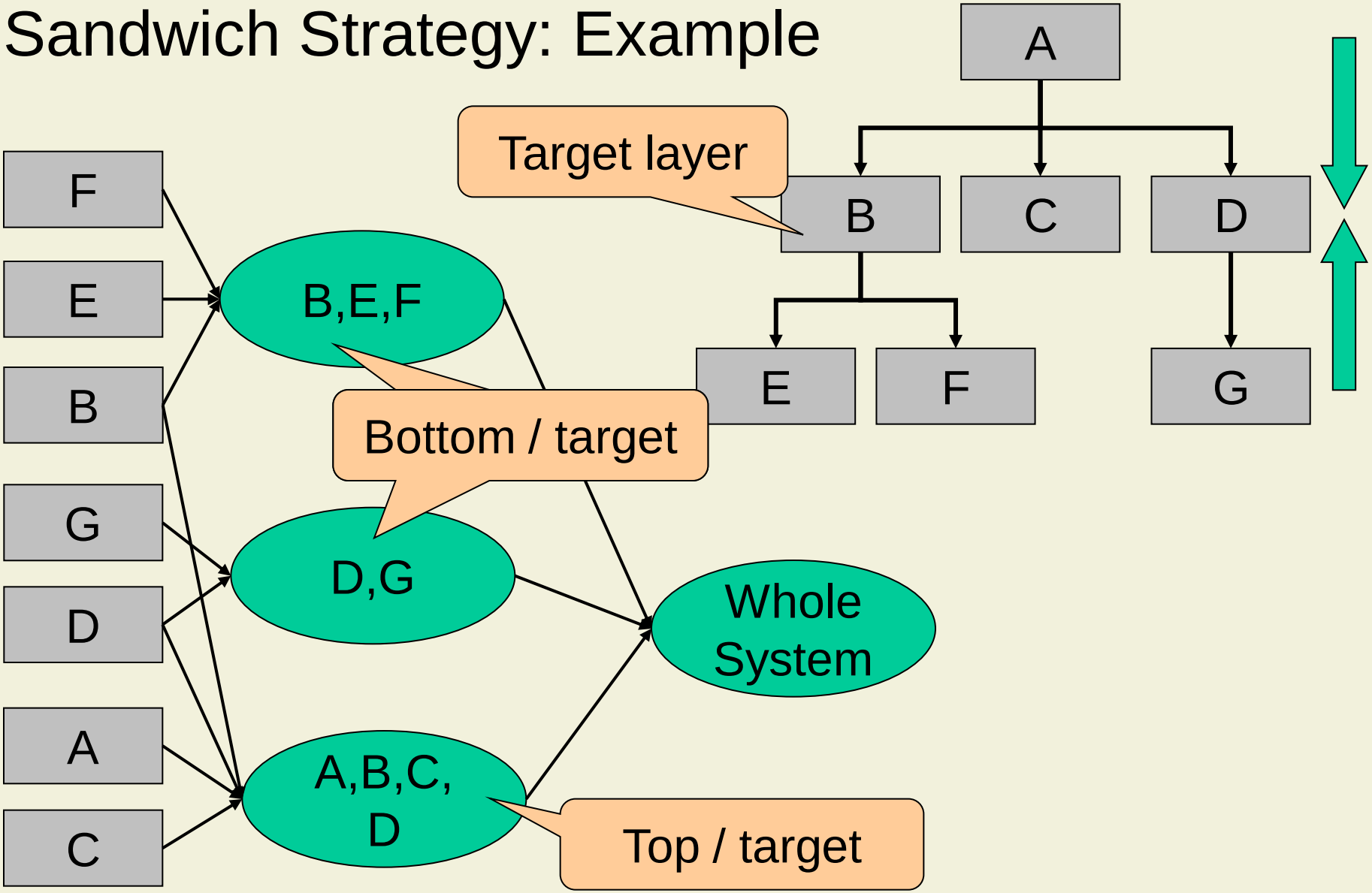




Sandwich Strategy

- Combines top-down 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
- How do you select the target layer if there are more than three 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)

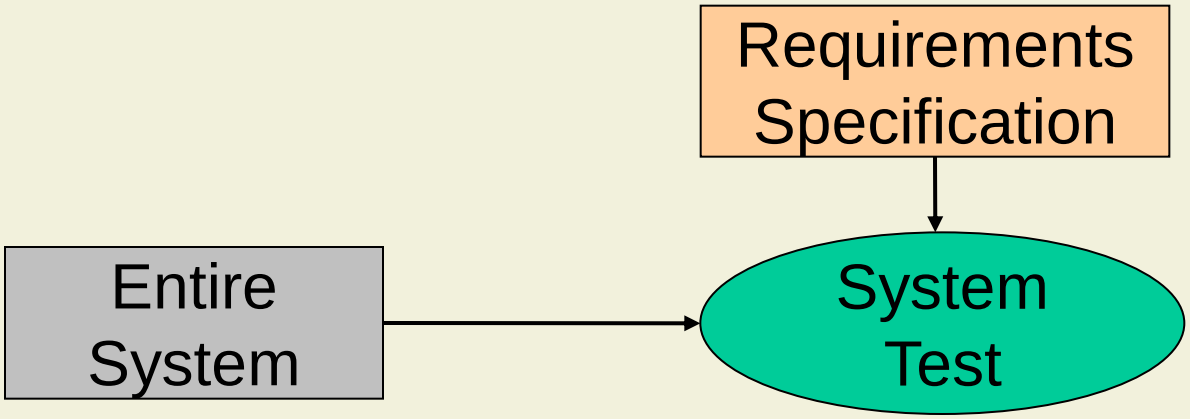
- **Cons**
  - Does not test the individual subsystems thoroughly before integration

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

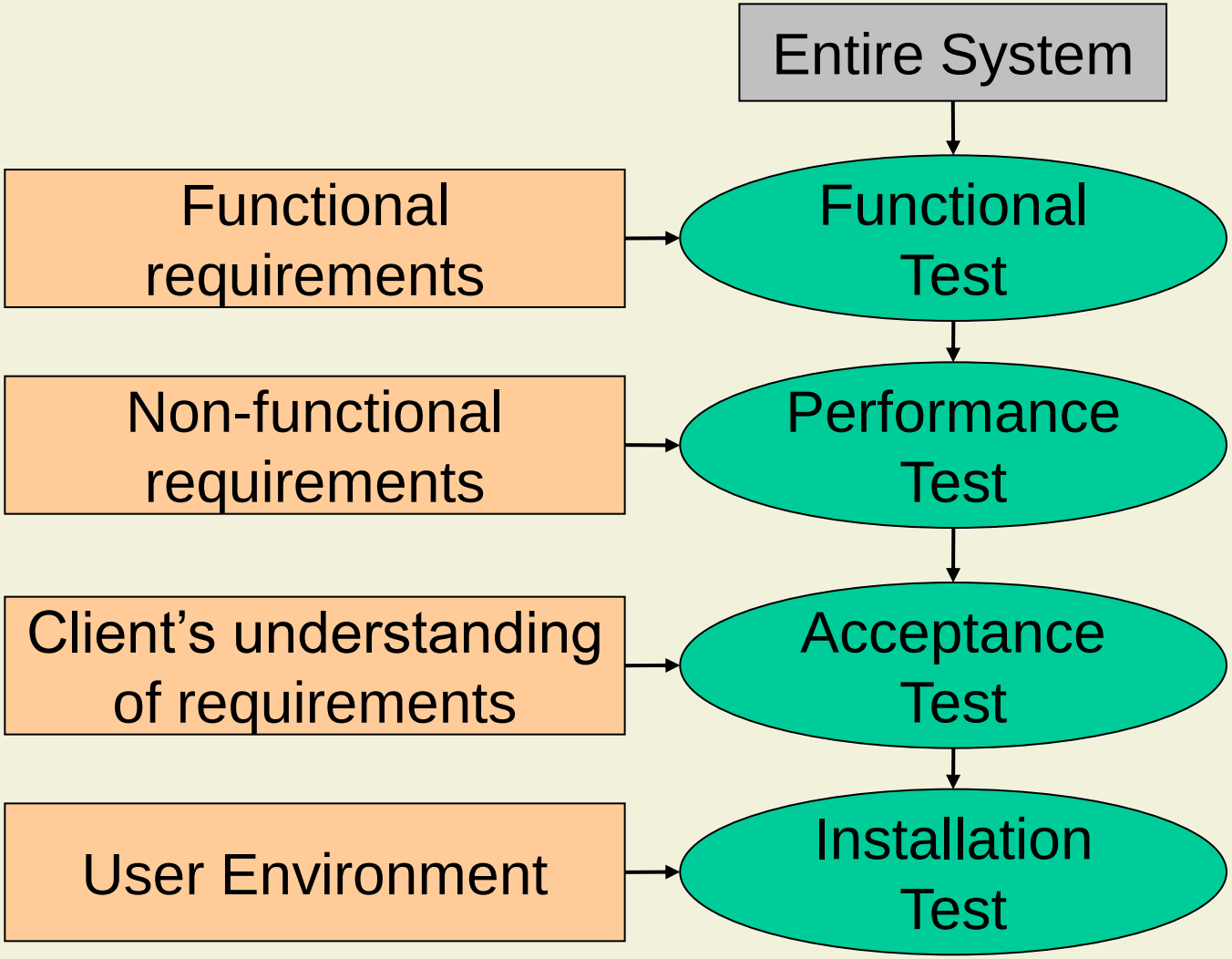
System Testing

- Testing the entire system



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

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 testing
  - 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

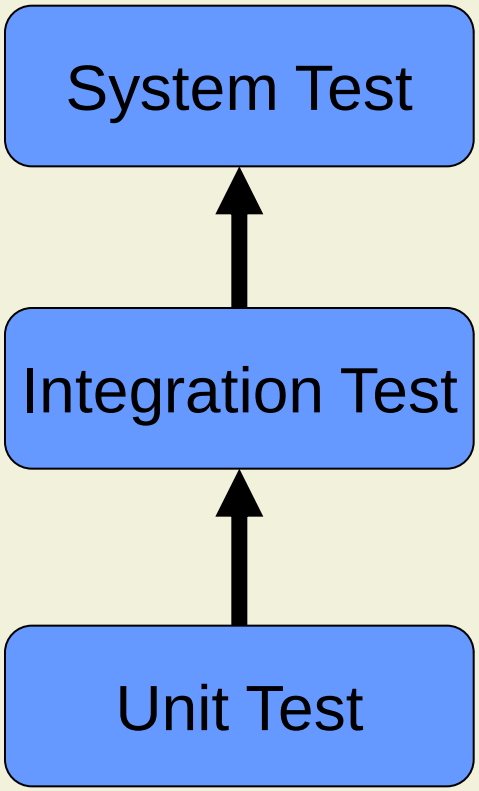
Acceptance Testing (cont'd)

- 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

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

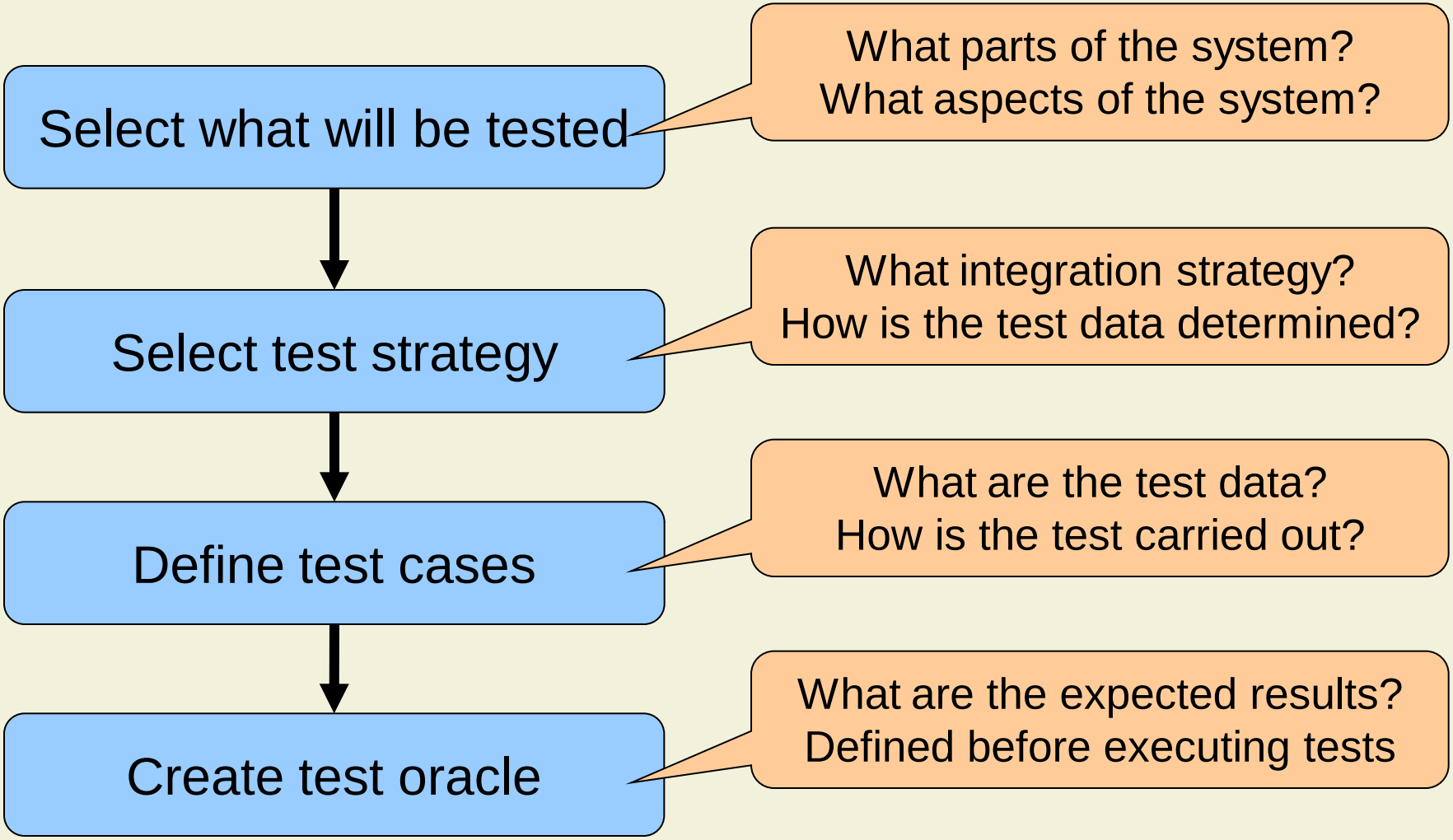
- The developer should not be testing at all
  - “Test before you code”
- Testers get only involved once software is done
- Toss the software over the wall for testing
  - Testers and developers collaborate in developing the test suite
- Testing team is responsible for assuring quality
  - Quality is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

7. Testing Basics Basics

7.1 Test Stages

7.2 Test Strategies

Testing Steps



Example: Solve Quadratic Equation

```
void roots( double a, double b, double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    numRoots = 2;
    double r = Math.sqrt( q );
    x1 = (-b + r) / (2 * a);
    x2 = (-b - r) / (2 * a);
  } else if( q == 0 ) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else {
    numRoots = 0;
  }
}
```

Fails if a==0 and b*b–4*a*c == 0

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Strategy 1: Exhaustive Testing

- Check UUT for all possible inputs
  - Not feasible, even for trivial programs

```
void roots( double a, double b, double c ) {
  …
}
```

- Assuming that **double** represents 64-bit values, we get $(2^{64})^3 \approx 10^{58}$ possible values for a, b, c
- Programs with heap data structures have a much larger state space!

Strategy 2: Random Testing

- Select test data uniformly

```
void roots( double a, double b, double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    …
  } else if( q == 0 ) {
    numRoots = 1;
    x1 = -b / (2 * a);
  } else { … }
}
```

Fails if a==0 and b*b–4*a*c == 0

The likelihood of selecting a==0 and b==0 randomly is $1/10^{38}$

Random Testing: Observations

- Random testing focuses on generating test data **fully automatically**

- Advantages
  - Avoids designer/tester bias
  - Tests robustness, especially handling of invalid input and unusual actions

- Disadvantages
  - Treats all inputs as equally valuable

Strategy 3: Functional Testing

- Use **analysis knowledge** to determine test cases that check requirements

Given three values, a, b, c, compute all solutions of the equation $ax^2 + bx + c = 0$

Two solutions	One solution	No solution
$a \neq 0$ and $b^2-4ac > 0$	$a = 0$ and $b \neq 0$ or $a \neq 0$ and $b^2-4ac = 0$	$a = 0$, $b = 0$, and $c \neq 0$ or $a \neq 0$ and $b^2-4ac < 0$

Test each case of the specification

Functional Testing: Observations

- Functional testing focuses on input/output behavior
  - Goal: **Cover all the requirements**
- Attempts to find
  - Incorrect or missing functions
  - Interface errors
  - Performance errors
- Limitations
  - Does not effectively detect design and coding errors (e.g., buffer overflow, memory management)
  - Does not reveal errors in the specification (e.g., missing cases)

Strategy 4: Structural Testing

- Use **design knowledge** about system structure, algorithms, data structures to determine test cases that exercise a large portion of the code

```
void roots( double a, double b, double c ) {
  double q = b*b – 4*a*c;
  if( q > 0 && a != 0 ) {
    …
  } else if( q == 0 ) {
    …
  } else {
    …
  }
}
```

Test this case

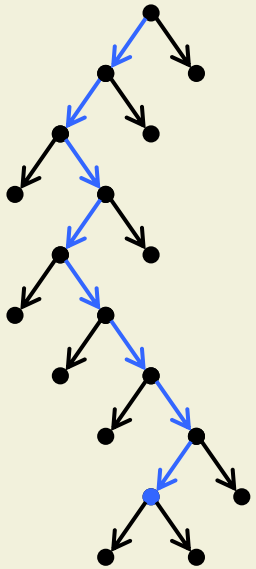
and this case

and this case

Error might still be missed, for instance, when case is tested with a==1, b==2, c==1

Structural Testing: Observations

- Structural testing focuses on thoroughness
  - Goal: **Cover all the code**
- Not well suited for system test
  - **Focuses on code** rather than on requirements, for instance, does not detect missing logic
  - **Requires design knowledge**, which testers and clients do not have (and do not care about)
  - Thoroughness would lead to **highly-redundant** tests



Testing Strategies: Summary

Functional testing

- Goal: Cover all the requirements
- Black-box test
- Suitable for all test stages

Structural testing

- Goal: Cover all the code
- White-box test
- Suitable for unit testing

Random testing

- Goal: Cover corner cases
- Black-box test
- Suitable for all test stages

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)