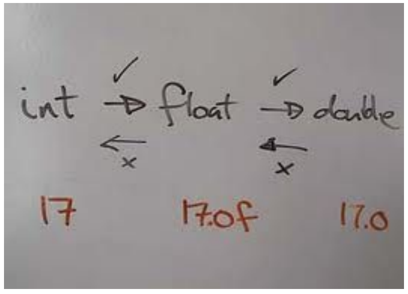
Object-Oriented Software Engineering
Using UML, Patterns, and Java

Chapter 11, Testing

ARIANE Flight 501

- Disintegration after 39 sec
- Caused by wrong data being sent to On Board Computer
- Large correction for attitude deviation
- Software exception in Inertial Reference System after 36 sec.
  - Overflow in conversion of a variable from 64-bit floating point to 16-bit signed integer
  - Of 7 risky conversions, 4 were protected
  - Reasoning: physically limited, or large margin of safety
  - In case of exception: report failure and shut down

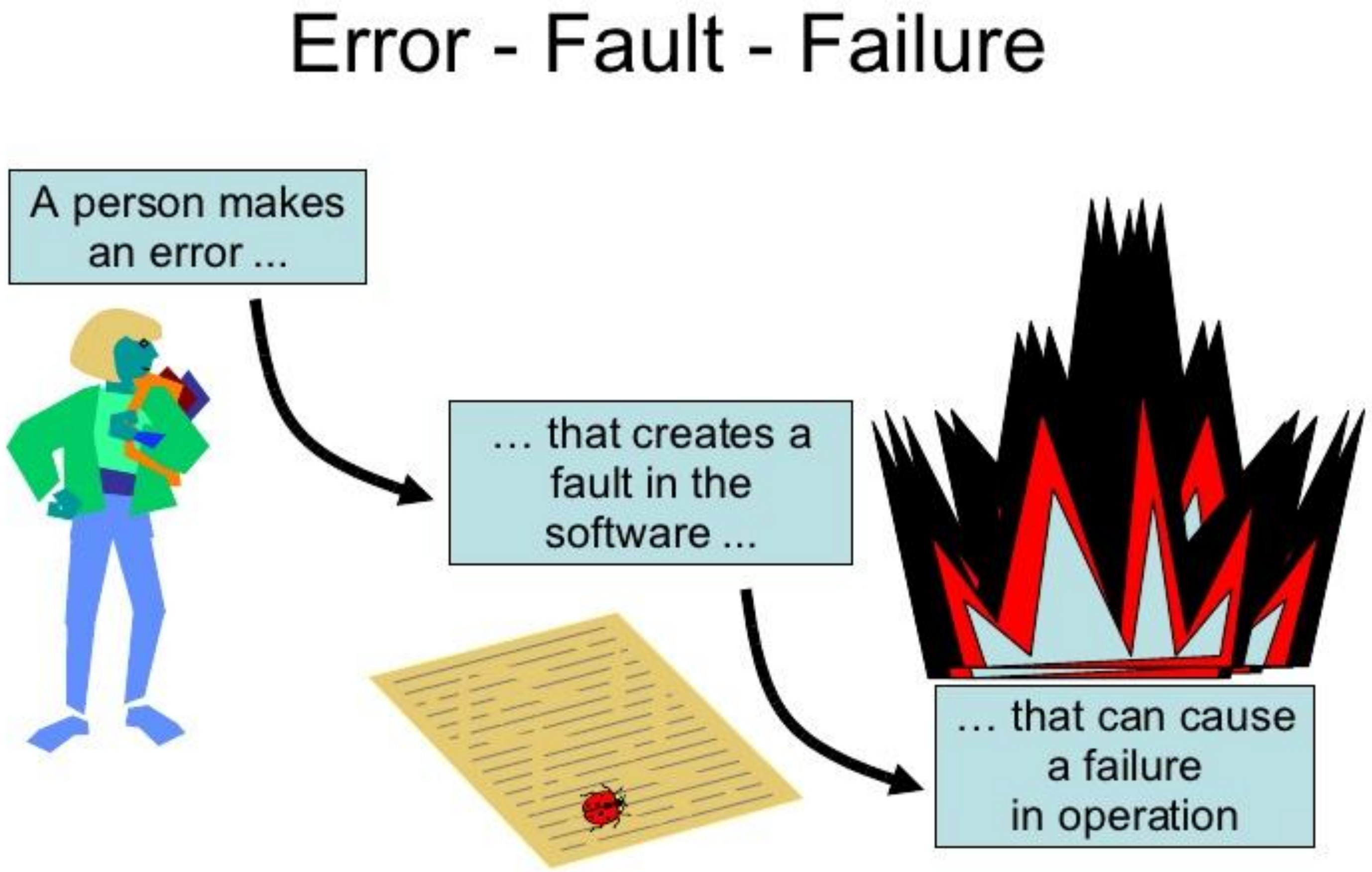


http://www.devtopics.com/20-famous-software-disasters-part-4/
http://en.wikipedia.org/wiki/List_of_software_bugs

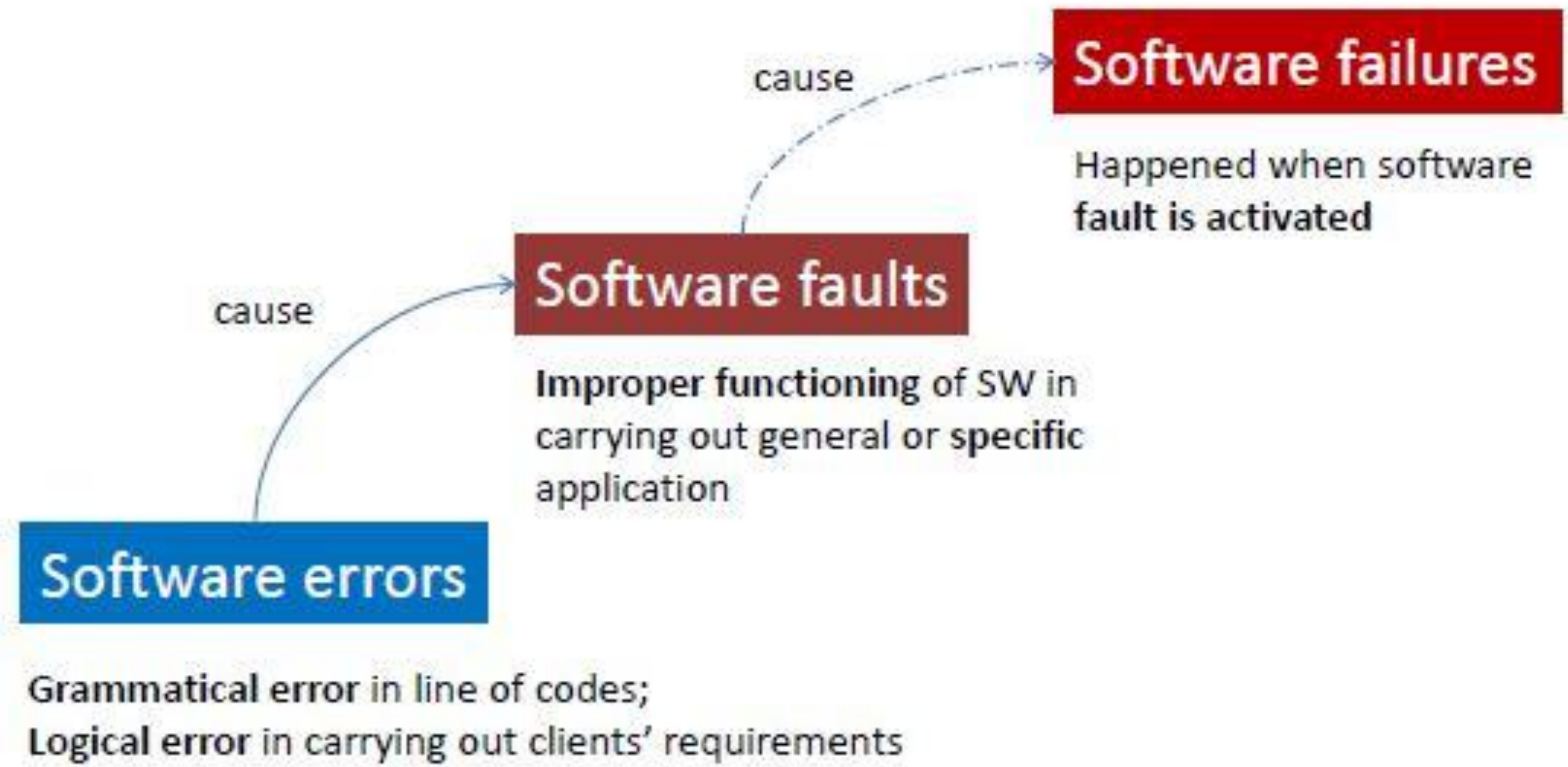
Terminology

- Failure: Any deviation of the observed behavior from the specified behavior
- Erroneous state (error): The system is in a state such that further processing by the system can lead to a failure
- Fault: The mechanical or algorithmic cause of an error ("bug")
- Validation: Activity of checking for deviations between the observed behavior of a system and its specification.

Error – Fault - Failure



Error – Fault - Failure



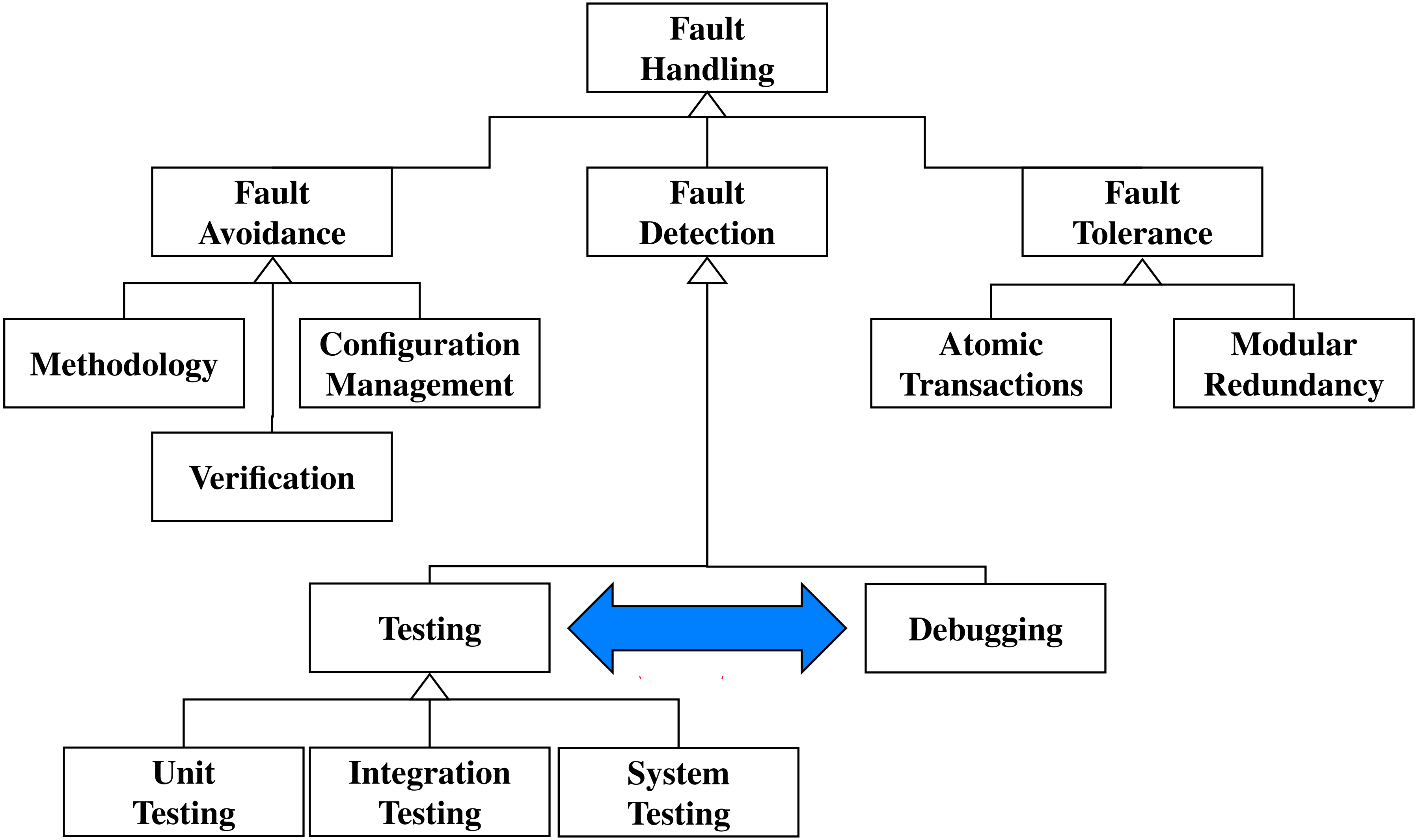
Examples of Faults and Errors

- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null
- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Null reference errors
  - Concurrency errors
  - Exceptions.

Another View on How to Deal with Faults

- Fault avoidance
  - Use methodology to reduce complexity
  - Use configuration management to prevent inconsistency
  - Apply verification to prevent algorithmic faults
  - Use Reviews
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (Faults) of an observed failure
  - Monitoring: Deliver information about state => Used during debugging
- Fault tolerance
  - Exception handling
  - Modular redundancy.

Taxonomy for Fault Handling Techniques



Observations

- It is impossible to completely test any nontrivial module or system
  - Practical limitations: Complete testing is prohibitive in time and cost
  - Theoretical limitations: e.g. Halting problem
- “Testing can only show the presence of bugs, not their absence” (Dijkstra).
- Testing is not for free

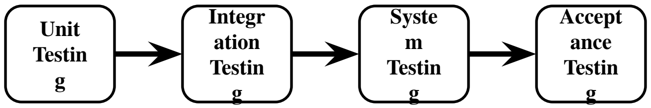
=> Define your goals and priorities

Testing takes creativity

- Development vs Testing mentality
- To develop an effective test, one must have:
  - Detailed understanding of the system
  - Application and solution domain knowledge
  - Knowledge of the testing techniques
  - Skill to apply these techniques
- Testing is done best by independent testers
  - We often develop a certain mental attitude that the program should behave in a certain way when in fact it does not
  - Programmers often stick to the data set that makes the program work
  - A program often does not work when tried by somebody else.

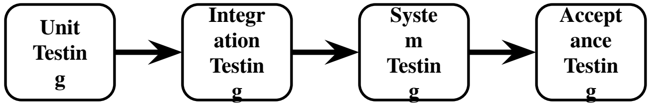
Unit test vs Integration Test

Types of Testing



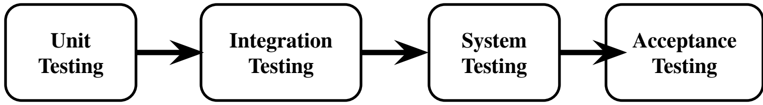
- Unit Testing
  - Individual component (class or subsystem)
  - Carried out by developers
  - Goal: Confirm that the component or subsystem is correctly coded and carries out the intended functionality
- Integration Testing
  - Groups of subsystems (collection of subsystems) and eventually the entire system
  - Carried out by development organization
  - Goal: Test the interfaces among the subsystems.

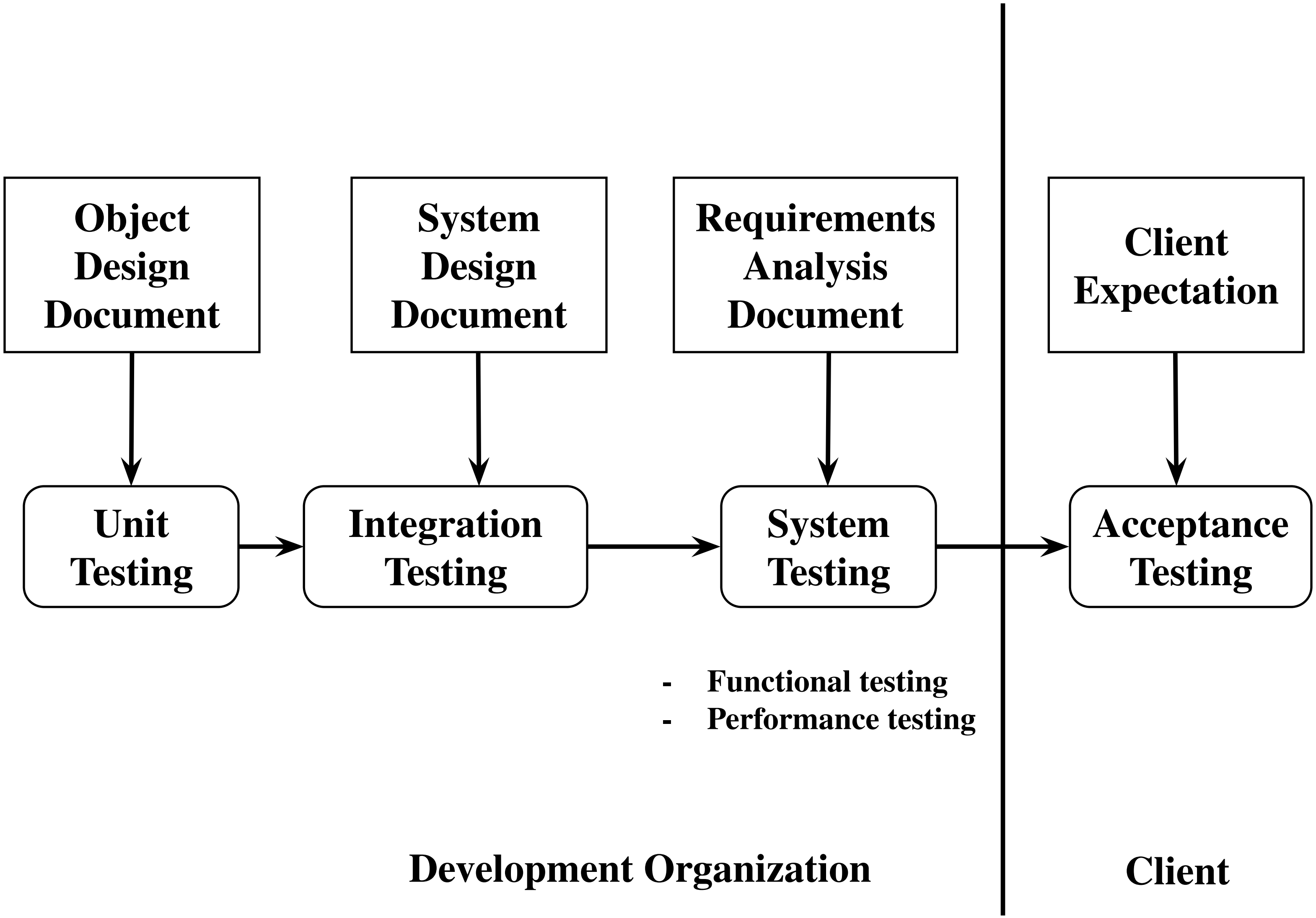
Types of Testing continued...



- System Testing
  - The entire system
  - Carried out by development organization
  - Goal: Determine if the system meets the requirements (functional and nonfunctional)
- Acceptance Testing
  - Evaluates the system delivered by developers
  - Carried out by the client. May involve executing typical transactions on site on a trial basis
  - Goal: Demonstrate that the system meets the requirements and is ready to use.

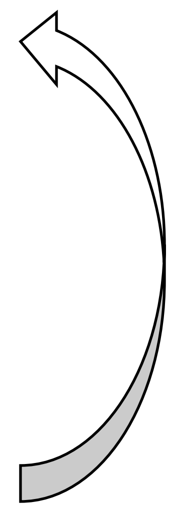
Testing Activities





When should you write a test?

- Traditionally after the source code is written
- In XP before the source code written
  - Test-Driven Development Cycle
    - Add a test
    - Run the automated tests
      => see the new one fail
    - Write some code
    - Run the automated tests
      => see them succeed
    - Refactor code.



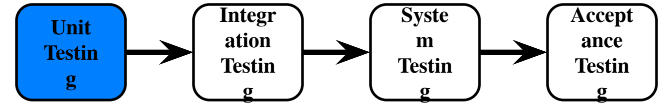
TDD Example

```java
public class TestMyClass extends TestCase {
    public void testSumUp() {
        MyClass myClass = new MyClass();
        assertEquals(5, myClass.sumUp(2, 3));
        assertEquals(9, myClass.sumUp(1, 2, 6));
    }

    public void testSumUpOverSize() {
        MyClass myClass = new MyClass();
        try {
            myClass.sumUp(Integer.MAX_VALUE, Integer.MAX_VALUE);
            myClass.sumUp(Integer.MIN_VALUE, Integer.MIN_VALUE);
            fail("Exception should be thrown here");
        } catch (ArithmeticException e) {
            // do nothing
        }

    }
}
```

JUnit: Overview



- A Java framework for writing and running unit tests
  - Test cases and fixtures
  - Test suites
  - Test runner
- Written by Kent Beck and Erich Gamma
- Written with “test first” and pattern-based development in mind
  - Tests written before code
  - Allows for regression testing
  - Facilitates refactoring
- JUnit is Open Source
  - www.junit.org
  - JUnit Version 4, released Mar 2006

Junit Example

```java
package junitTutorial;

public class Airthematic {
    public int sum(int a,int b){
        return a+b;
    }

}
```

```java
    @Test
    public void testAirthematicTest() {
        // assert statements
        assertEquals("10 +10 must be 20", 20, airthematic.sum(10, 10))
        assertEquals("20 +20 must be 40", 40, airthematic.sum(20, 20))
        assertEquals("30 +10 must be 40", 40, airthematic.sum(30, 10))

    }
}

```

An example: Testing MyList

- Unit to be tested
  - MyList
- Methods under test
  - add()
  - remove()
  - contains()
  - size()
- Concrete Test case
  - MyListTestCase

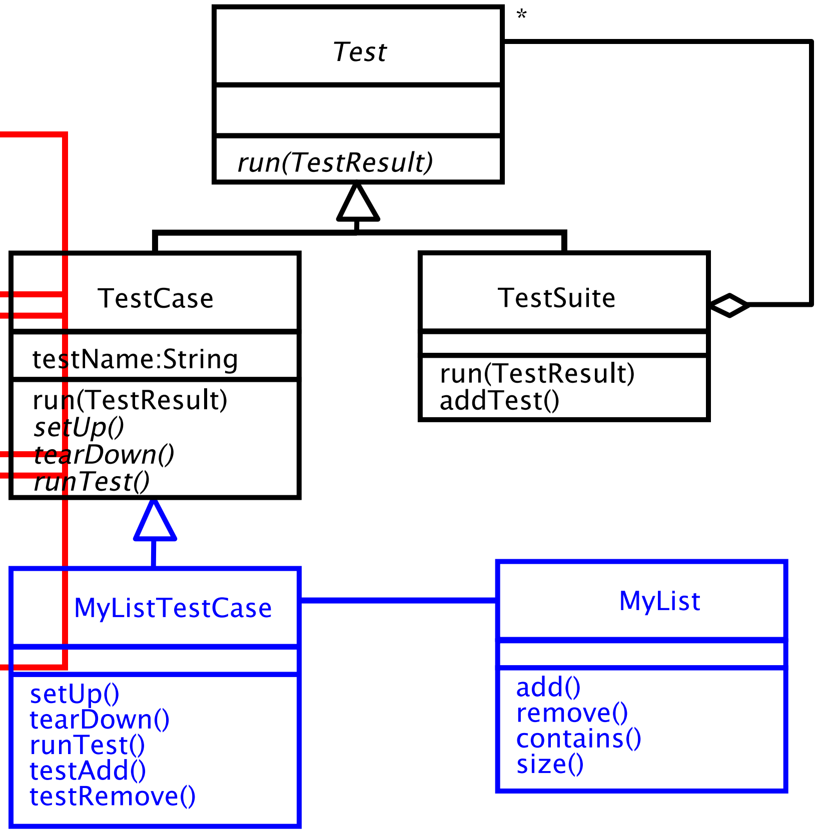
Writing TestCases in JUnit

```java
public class MyListTestCase extends TestCase {

public MyListTestCase(String name) {
    super(name);
}
public void testAdd() {
    // Set up the test
    List aList = new MyList();
    String anElement = "a string";

    // Perform the test
    aList.add(anElement);

    // Check if test succeeded
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}
protected void runTest() {
    testAdd();
}
}
```



Writing Fixtures and Test Cases

```
public class MyListTestCase extends TestCase {
// …
private MyList aList;
private String anElement;
public void setUp() {
    aList = new MyList();
    anElement = "a string";
}

public void testAdd() {
    aList.add(anElement);
    assertTrue(aList.size() == 1);
    assertTrue(aList.contains(anElement));
}

public void testRemove() {
    aList.add(anElement);
    aList.remove(anElement);
    assertTrue(aList.size() == 0);
    assertFalse(aList.contains(anElement));
}
```

Test Fixture

Test Case

Test Case

Integration Testing

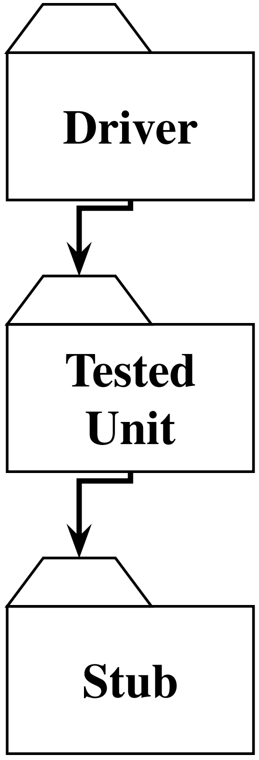
- The entire system is viewed as a collection of subsystems (sets of classes) determined during the system and object design
- Goal: Test all interfaces between subsystems and the interaction of subsystems
- The Integration testing strategy determines the order in which the subsystems are selected for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

Why do we do integration testing?

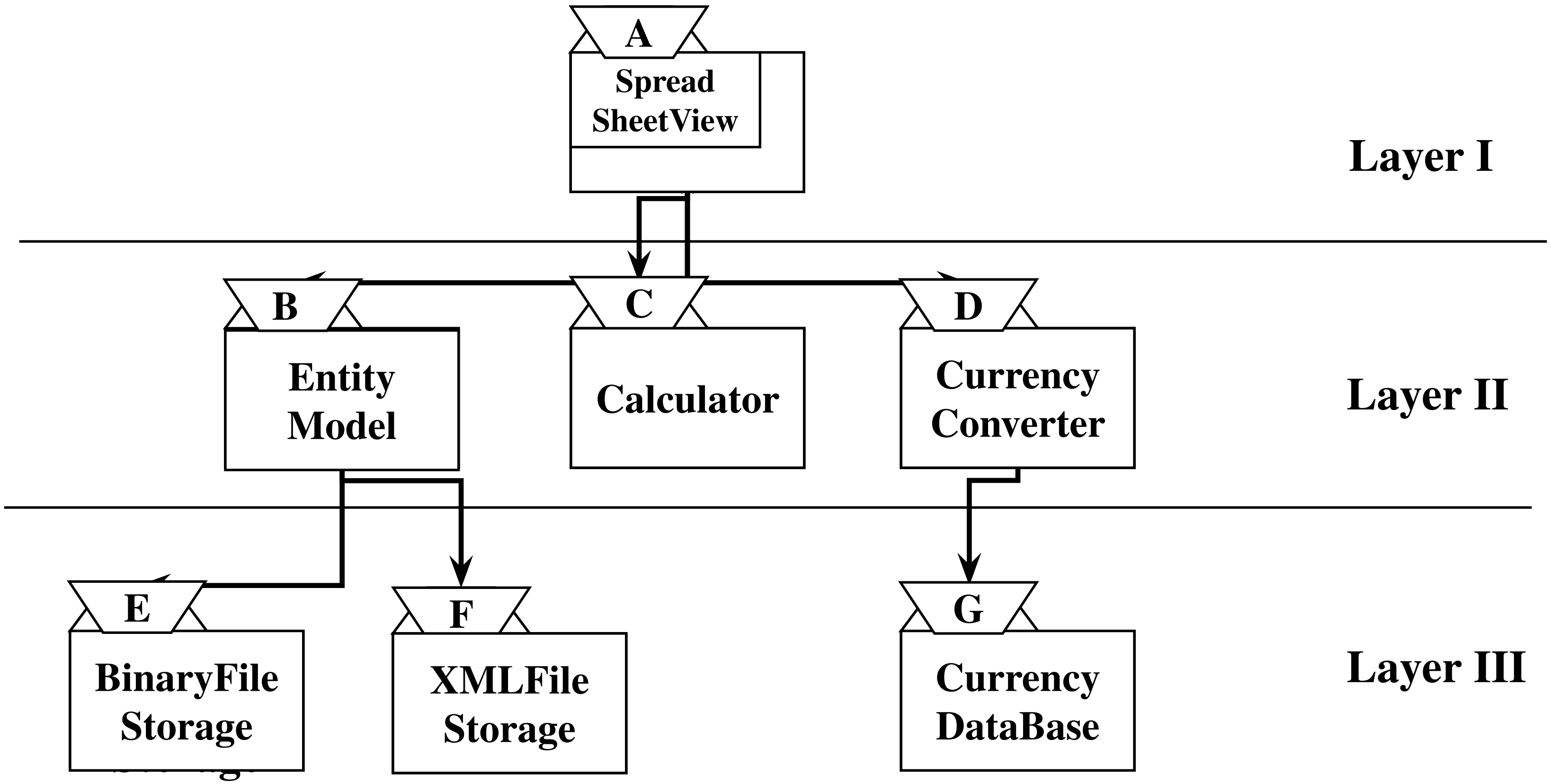
- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- Often many Off-the-shelf components are used that cannot be unit tested
- Without integration testing the system test will be very time consuming
- Failures that are not discovered in integration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

Stubs and drivers

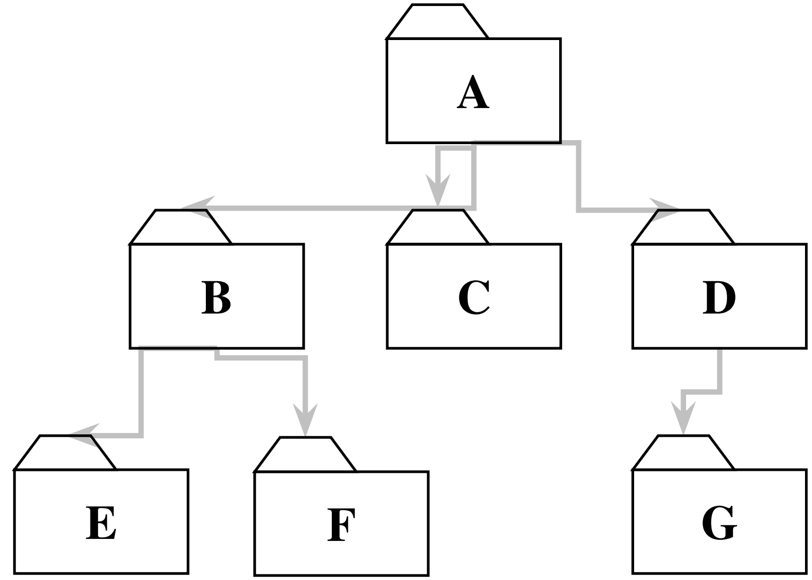
- Driver:
  - Partial implementation of a component, that calls the `TestedUnit`
  - Controls the test cases
- Stub:
  - A component, the `TestedUnit` depends on
  - Partial implementation of components on which the tested component depends
  - Returns fake values.

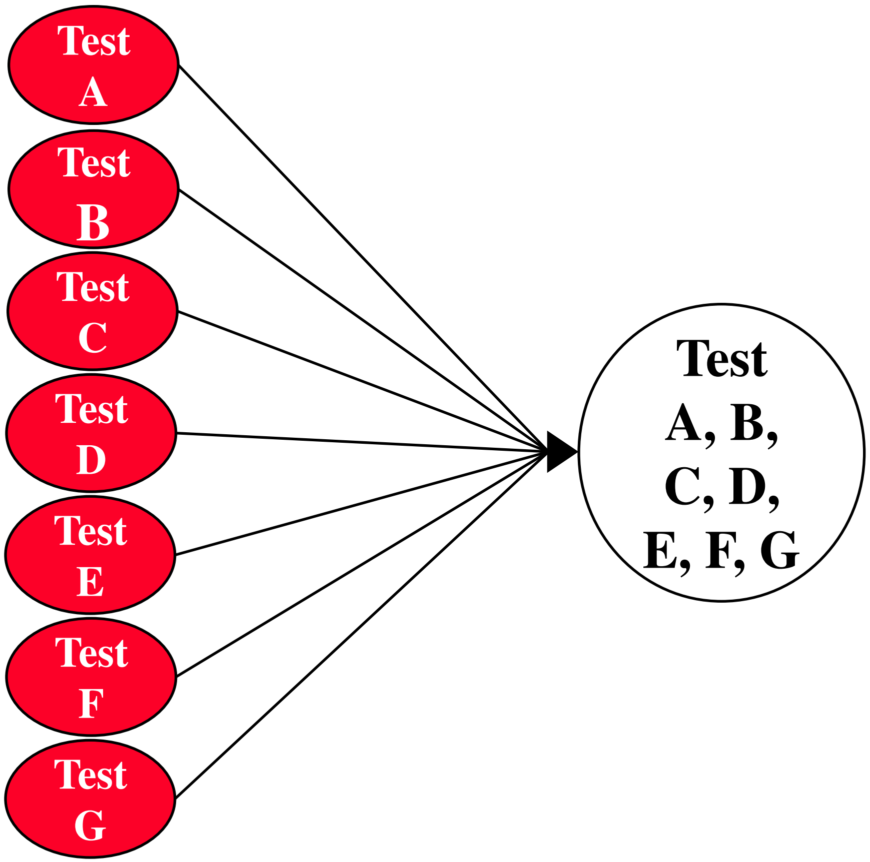


Example: A 3-Layer-Design (Spreadsheet)



Big-Bang Approach

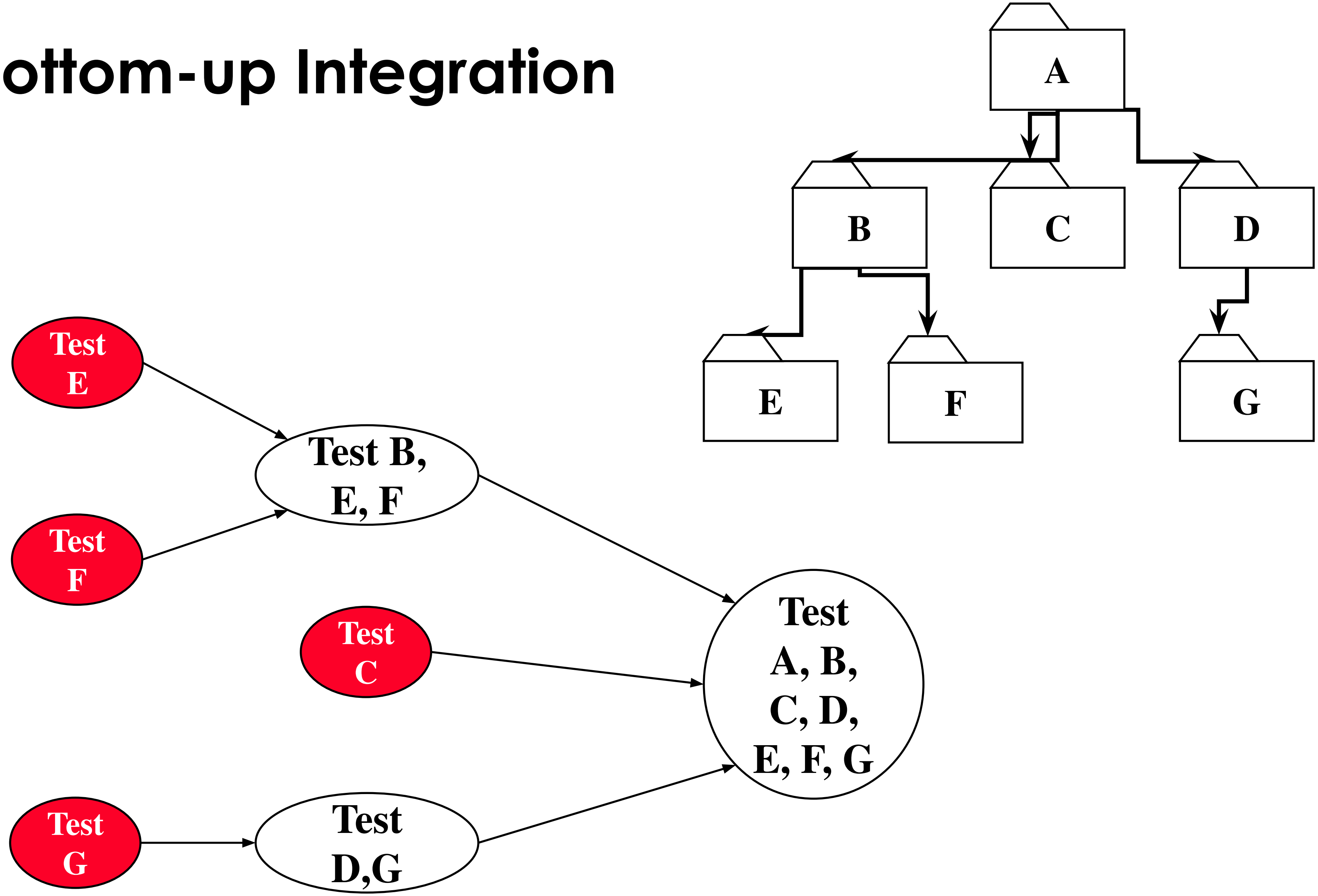




Bottom-up Testing Strategy

- 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



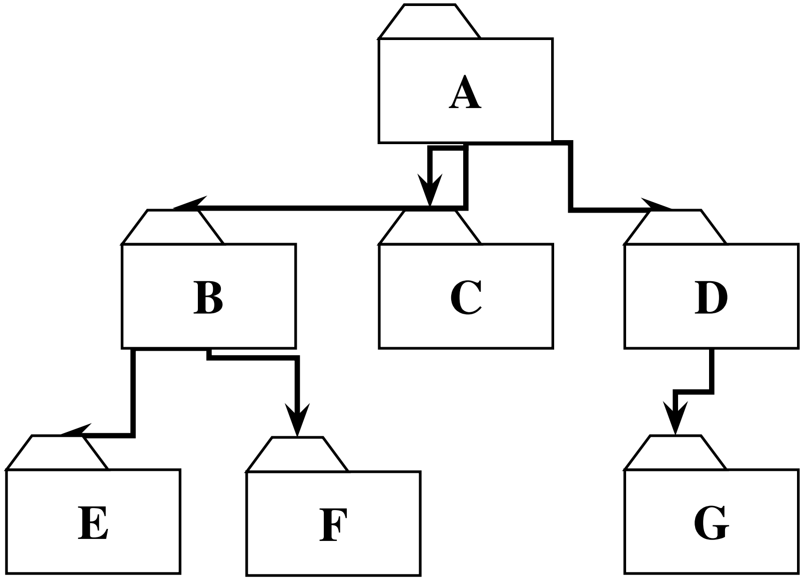
Pros and Cons of Bottom-Up Integration Testing

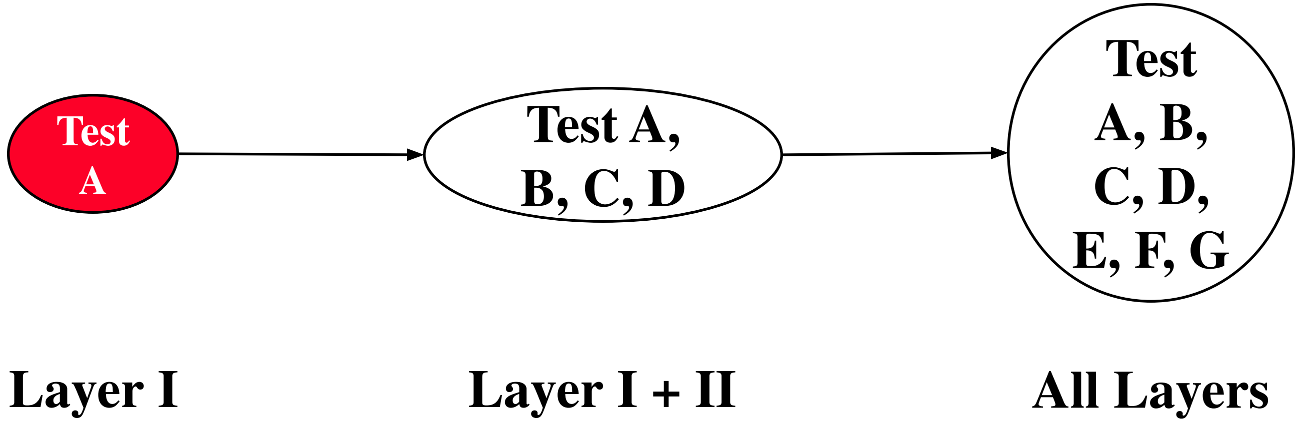
- Con:
  - Tests the most important subsystem (user interface) last
  - Drivers needed
- Pro
  - No stubs needed
  - Useful for integration testing of the following systems
    - Object-oriented systems
    - Real-time systems
    - Systems with strict performance requirements.

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.

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.

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

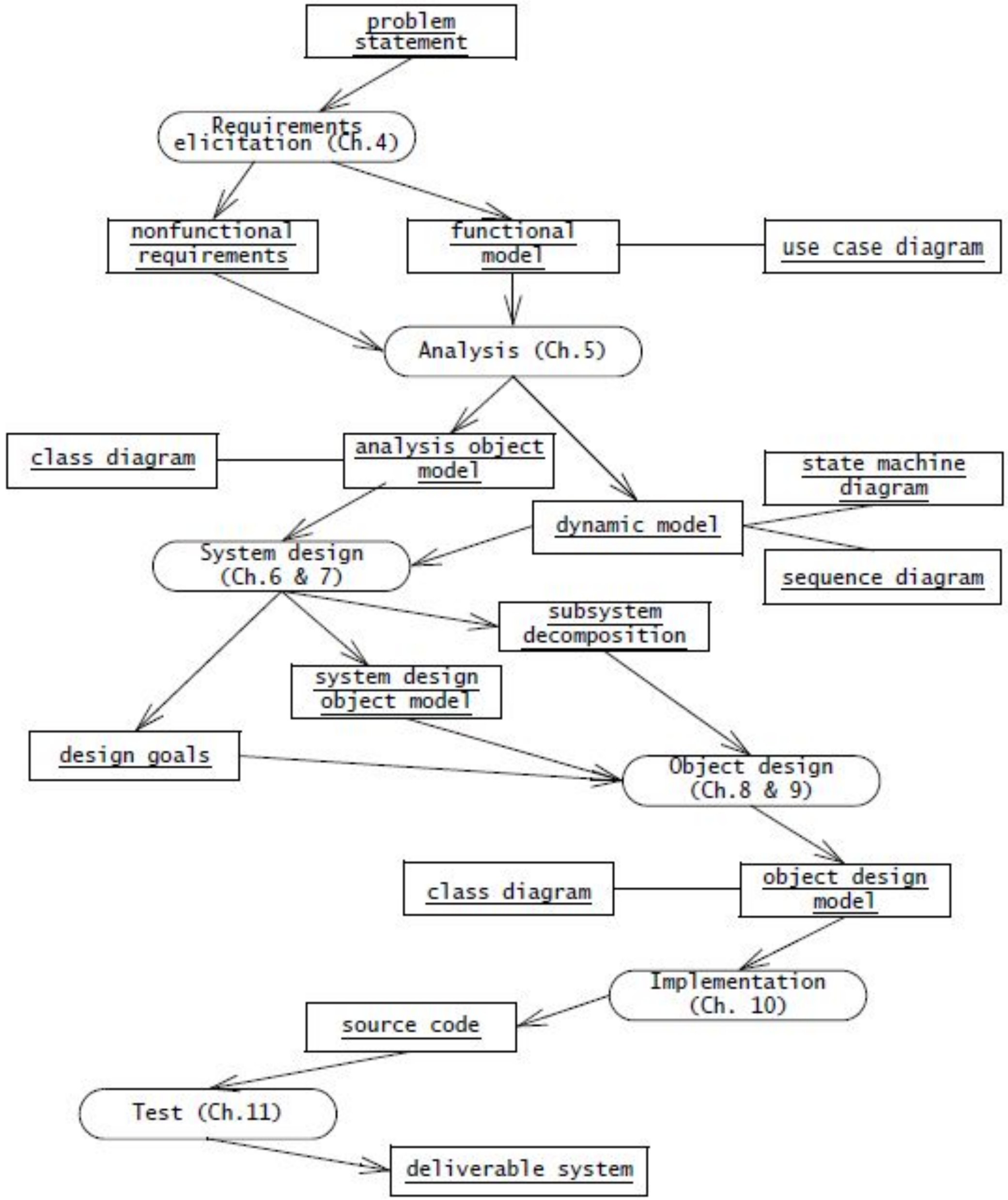
- Stress Testing
  - Stress limits of system
- Volume testing
  - Test what happens if large amounts of data are handled
- Configuration testing
  - Test the various software and hardware configurations
- Compatibility test
  - Test backward compatibility with existing systems
- Timing testing
  - Evaluate 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 environ-ment

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


problem statement
Requirements elicitation (Ch.4)
nonfunctional requirements
functional model
use case diagram
Analysis (Ch.5)
class diagram
analysis object model
state machine diagram
dynamic model
sequence diagram
System design (Ch.6 & 7)
subsystem decomposition
system design object model
design goals
Object design (Ch.8 & 9)
class diagram
object design model
Implementation (Ch. 10)
source code
Test (Ch.11)
deliverable system

Final Exam

Chapter 1 - Introduction
Chapter 2 - Modeling with UML
Chapter 3 - Project Organization and Communication
Chapter 4 - Requirements Elicitation
Chapter 5 - Analysis – Object / Dynamic Model
Chapter 6 - System Design: Decomposing The System
Chapter 7 - System Design: Addressing Design Goals
Chapter 8 - Object Design: Reusing Pattern Solutions
Chapter 8 & Appendix A - Object Design: Design Patterns I
Chapter 9 - Object Design: Specifying Interfaces / OCL
Chapter 10 - Mapping Models to Code
Chapter 11 – Testing / Integration & System Testing

- Closed Book

Final Exam tips

- Go through sample final/midterm exam
- Go through book first /read through slides
- UML Basics
- Go through few UML exercises on your own (in paper) e.g : Use case, state, activity, sequence diagrams etc
- Go through design patterns on your own
- Go through model transformations on your own

Term Project Tips

- Make sure to go through the comments that your TA provided in the first iteration.
- Make sure to implement my comments I delivered in class related to the reports in general (Especially related to the choose of correct and informative UML diagrams)
- Make sure to implement my comments in the private demos
- Make sure all the slides / diagrams in the reports are legible
- Make sure you have the necessary cabling before the demo and be on time.
- Make sure you carefully read the instructions related to the report and presentation formats
- For majority of the projects, the current form of implementations is very basic which is normal for the first iteration. For the second iteration, I have much higher expectations.
- For the second iteration, you are expected to update/enrich requirements/design/implementation.

Term Project

Requirements (25 points)

- Use case (3 points)
- NFR (2 points)
- Activity (4 points)
- State (4 points)
- Sequence (4 points)
- Class (4 points)
- UI mockups (4 points)

Design (25 points)

- High-level architecture (8 points)
- Design goals (2 points)
- Class Diagram (5 points)
- Design patterns (10 points)

Term Project

- Implementation (35 points)
  - Criteria: Number and complexity of features, quality of implementation decisions, code comments, final report (user guide and build instructions), code style, naming conventions,, etc.
- Demo and presentation (15 points)
  - Criteria: Flow and quality of the presentation, Demo performance, creativity of demo videos etc.
- Individual performance factors
  - Criteria: Peer review grades, GitHub records, individual presentation and QA performance etc.

- Perform 2 full rehearses for final demo
- Emphasize your strong attributes