Testing

Slides prepared by Brend Bruegge & Allen Dutoit

Famous Problems

- F-16 : crossing equator using autopilot
  - Result: plane flipped over
  - Reason?
    - Reuse of autopilot software from a rocket

- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem
- The Therac-25 accidents (1985-1987), quite possibly the most serious non-military computer-related failure ever in terms of human life (at least five died)
  - Reason: Bad event handling in the GUI,

The Therac-25

- The Therac-25 was a medical linear accelerator
- Linear accelerators create energy beams to destroy tumors
  - Used to give radiation treatments to cancer patients
  - Most of the patients had undergone surgery to remove a tumor and were receiving radiation to remove any leftover growth
- For shallow tissue penetration, electron beams are used
- To reach deeper tissue, the beam is converted into x-rays
- The Therac-25 had two main types of operation, a low energy mode and a high energy mode:
  - In low energy mode, an electronic beam of low radiation (200 rads) is generated
  - In high energy mode the machine generates 25000 rads with 25 million electron volts
- Therac-25 was developed by two companies, AECL from Canada and CGR from France
  - Newest version(reusing code from Therac-6 and Therac-20).

A Therac-25 Accident

- In 1986, a patient went into the clinic to receive his usual low radiation treatment for his shoulder
- The technician typed X (x-ray beam), realizing the error, quickly changed X into E (electron beam), and hit "enter":
  - `X <Delete char>  E <enter>`
  - This input sequence in a short time frame (about 8 sec) was never tested
- Therac-25 signaled "beam ready" and it also showed the technician that it was in low energy mode
- The technician typed B to deliver the beam to the patient
  - The beam that actually came from the machine was a blast of 25 000 rads with 25 million electron volts, more than 125 times the regular dose
  - The machine responded with error message “Malfunction 54”, which was not explained in the user manual. Machine showed under dosage.
  - Operator hit “P” to continue for more treatment. Again, the same error message
- The patient felt sharp pains in his back, much different from his usual treatment. He died 3 months later.

Reasons for the Therac-25 Failure

- Failure to properly reuse the old software from Therac-6 and Therac-20 when using it for new machine
- Cryptic warning messages
- End users did not understand the recurring problem (5 patients died)
- Lack of communication between hospital and manufacturer
- The manufacturer did not believe that the machine could fail
- No proper hardware to catch safety glitches.

How the Problem was solved

- On February 10, 1987, the Health Protection Branch of the Canadian government along with the FDA (United States Food and Drug Administration) announced the Therac-25 dangerous to use
- On July 21, 1987 recommendations were given by the AECL company on how to repair the Therac-25. Some of these recommendations were
  - Operators cannot restart the machine without re-entering the input command
  - The dose administered to the patient must be clearly shown to the operator
  - Limiting the input modalities to prevent any accidental typos
  - Error messages must be made clearer
  - All manuals must be rewritten to reflect new changes.

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.

Examples of Faults and Errors

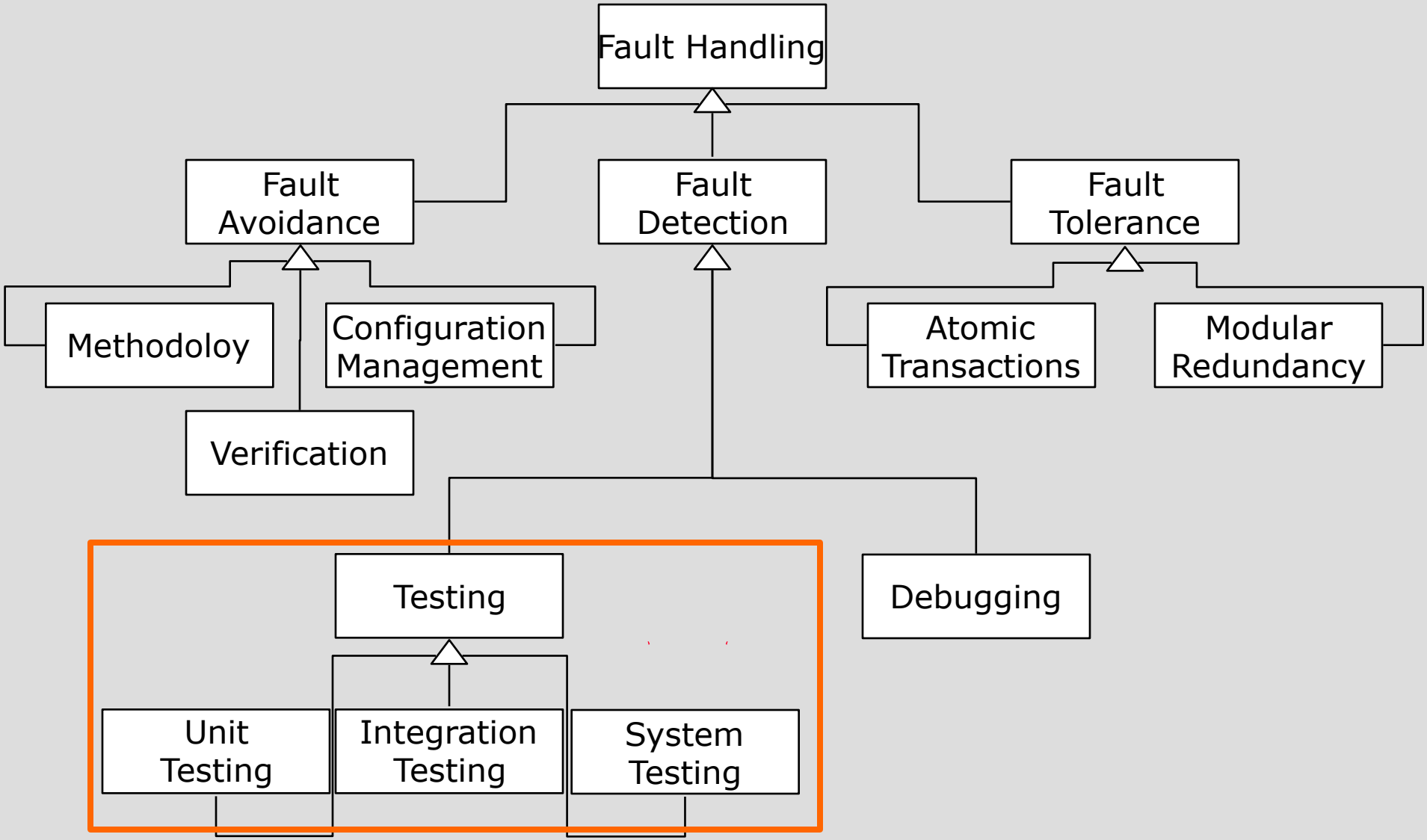
- Faults in the Interface specification
  - Mismatch between what the client needs and what the server offers
  - Mismatch between requirements and implementation
- Algorithmic Faults
  - Missing initialization
  - Incorrect branching condition
  - Missing test for null

- Mechanical Faults (very hard to find)
  - Operating temperature outside of equipment specification
- Errors
  - Wrong user input
  - 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 to identify faults already in the design
- Fault detection
  - Testing: Activity to provoke failures in a planned way
  - Debugging: Find and remove the cause (fault) of an observed failure
  - Monitoring: Deliver information about state and behavior => 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

Edsger W. Dijkstra (1930-2002)
- First Algol 60 Compiler
- 1968:
  - T.H.E.
  - Go To considered Harmful, C
- Since 1970 Focus on Verificatio and Foundations of Computer S
- 1972 A. M. Turing Award

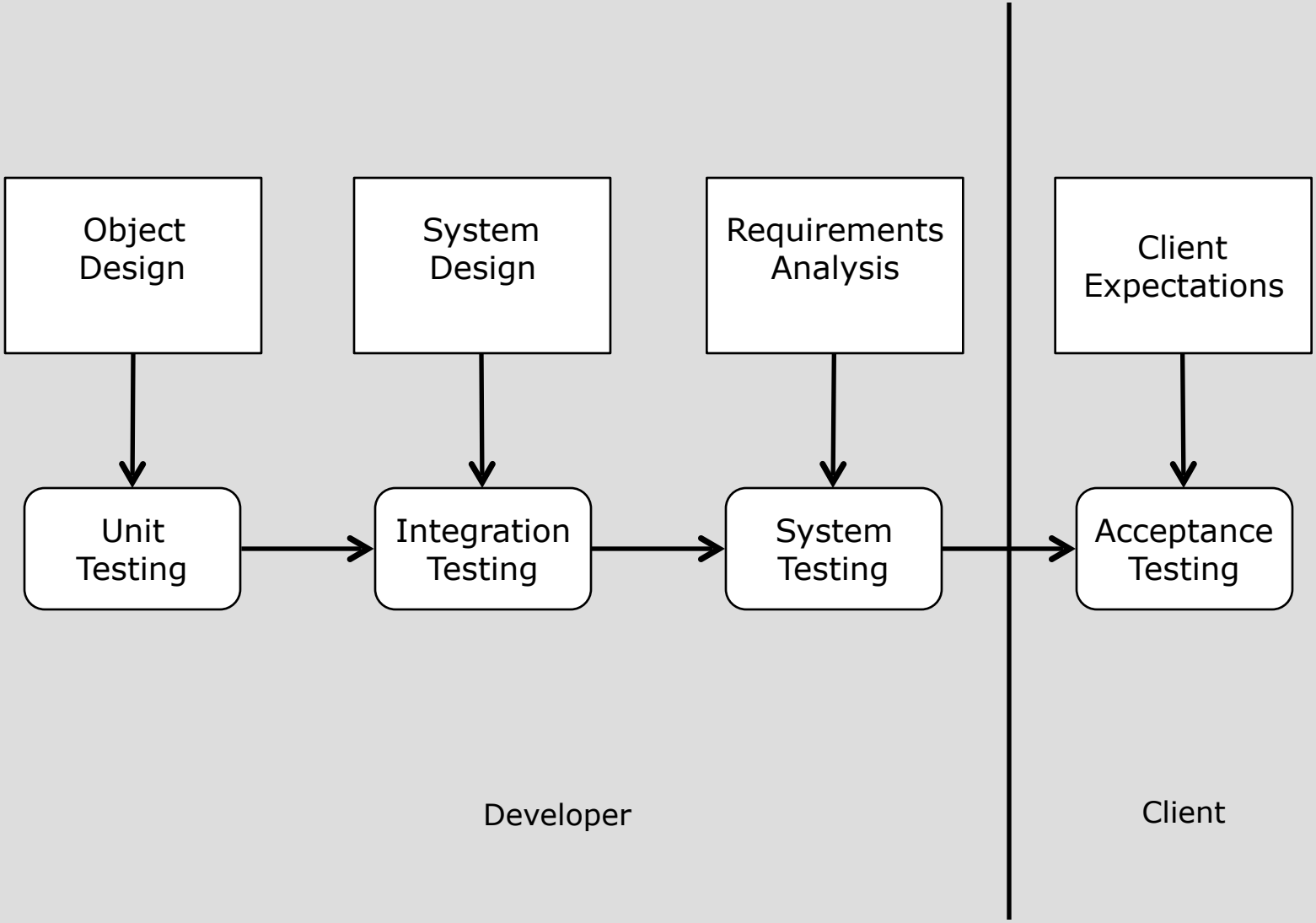
Test Model

- The Test Model consolidates all test related decisions and components into one package (sometimes also test package or test requirements)
- The test model contains tests, test driver, input data, oracle and the test harness
  - A test driver (the program executing the test)
  - The input data needed for the tests
  - The oracle comparing the expected output with the actual test output obtained from the test
  - The test harness
    - A framework or software components that allow to run the tests under varying conditions and monitor the behavior and outputs of the system under test (SUT)
    - Test harnesses are necessary for automated testing.

Automated Testing

- There are two ways to generate the test model
  - Manually: The developers set up the test data, run the test and examine the results themselves. Success and/or failure of the test is determined through observation by the developers
  - Automatically: *Automated generation* of test data and test cases. Running the test is also done automatically, and finally the comparison of the result with the oracle is also done automatically
- Definition Automated Testing
  - All the test cases are *automatically executed* with a test harness
- Advantage of automated testing:
  - Less boring for the developer
  - Better test thoroughness
  - Reduces the cost of test execution
  - Indispensable for regression testing.

Testing Activities and Models



Types of Testing

- Unit Testing
  - Individual components (class or subsystem) are tested
  - 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 are tested
  - Carried out by developers
  - Goal: Test the interfaces among the subsystems.
- System Testing
  - The entire system is tested
  - Carried out by developers
  - 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.

Unit Testing: Equivalence Testing (Black-Box Testing)

- Is a Black Box Testing.
- Focus: I/O behaviour. If for any given input, we can predict the output, then the unit passes the test.
  - Almost always impossible to generate all possible inputs ("test cases")
- Goal: Reduce number of test cases by equivalence partitioning:
  - Divide inputs into equivalence classes
  - Choose test cases for each equivalence class
    - Example: If an object is supposed to accept a negative number, testing one negative number is enough.

Equivalence Testing: An example

```
public class MyCalendar {

   public int getNumDaysInMonth(int month, int year)
            throws InvalidMonthException
   { … }
}
```

Assume the following representations:

`Month: (1,2,3,4,5,6,7,8,9,10,11,12)`

where 1= Jan, 2 = Feb, …, 12 = Dec

`Year: (1904,…,1999,2000,…,2010)`

How many test cases do we need to do a full black box unit test of `getNumDaysInMonth()`?

Equivalence Testing: An example

- Depends on calendar. We assume the Gregorian calendar
- Equivalence classes for the `month` parameter
  - Months with 30 days, Months with 31 days, February (3 classes)
  - Equivalence classes for the `Year` parameter (2 classes)
    - A normal year
    - Leap years
      - Divisible by /4
      - Divisible by /100 (also have to be divisible by 400 to be leap year)
      - Divisible by /400

How many equivalence classes do we need to do a full black box unit test of **`getNumDaysInMonth()`**? 6 test cases

Equivalence Testing (Continued)

- Selection of inputs:
  - Input is valid across range of values:
    - Below the range
    - Within the range
    - Above the range
  - Input is valid if it is from a discrete set:
    - Valid discrete value
    - Invalid discrete value
- Another solution to select only a limited amount of test cases:
  - Get knowledge about the inner workings of the unit being tested => white-box testing

White-box Testing

- Focus: Thoroughness (Coverage). Every statement in the component is executed at least once
- Four types of white-box testing
  - Statement Testing
  - Loop Testing
  - Path Testing
  - Branch Testing.

White-box Testing (Continued)

- Statement Testing (Algebraic Testing)
  - Tests each statement (Choice of operators in polynomials, etc)
- Loop Testing
  - Loop to be executed exactly once
  - Loop to be executed more than once
  - Cause the execution of the loop to be skipped completely
- Path testing:
  - Makes sure all paths in the program are executed
- Branch Testing (Conditional Testing)
  - Ensure that each outcome in a condition is tested at least once
  - Example:

```
if ( i =TRUE) printf(”Yes");else  printf(”No");
```

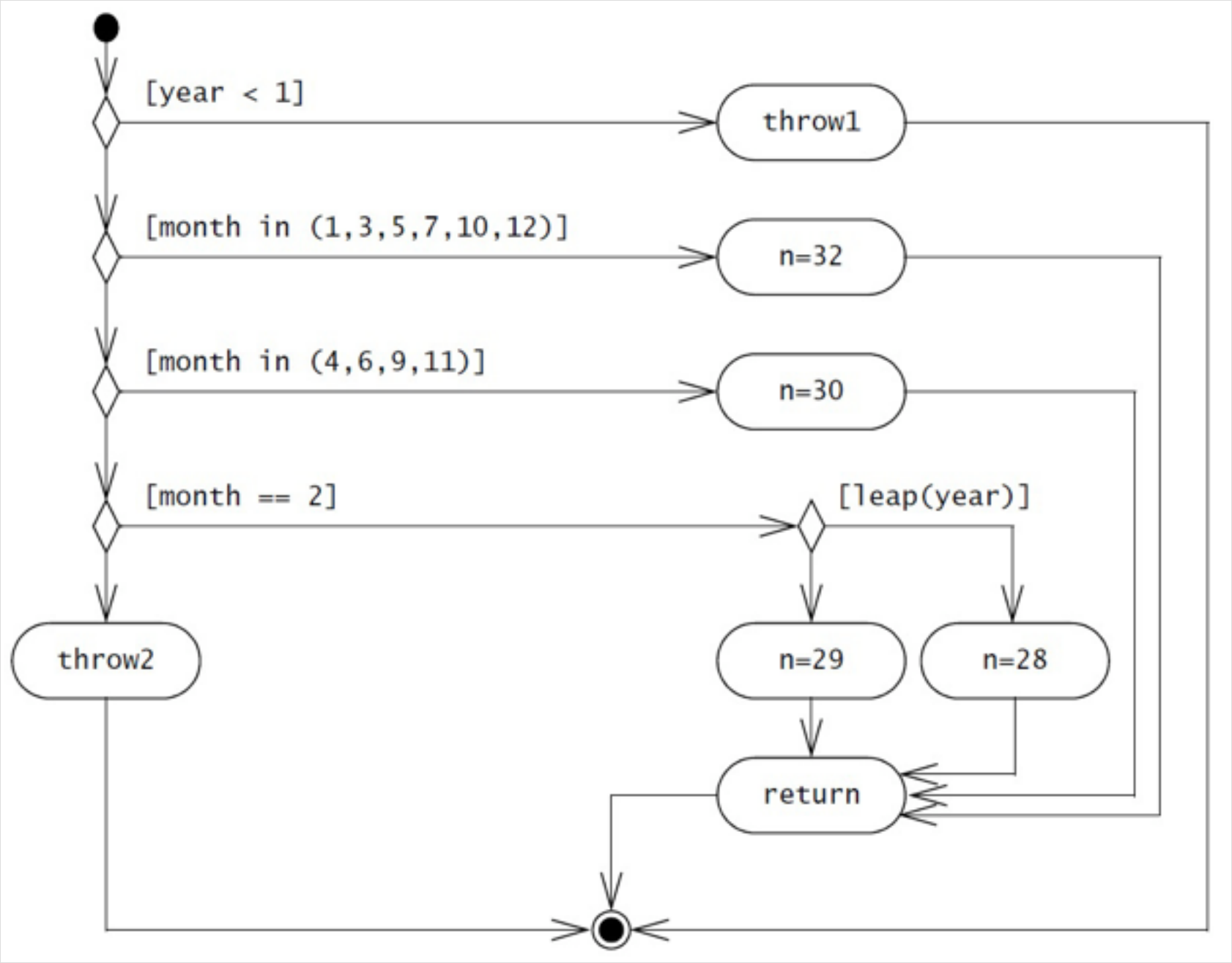
How many test cases do we need to unit test this statement?

Path Testin

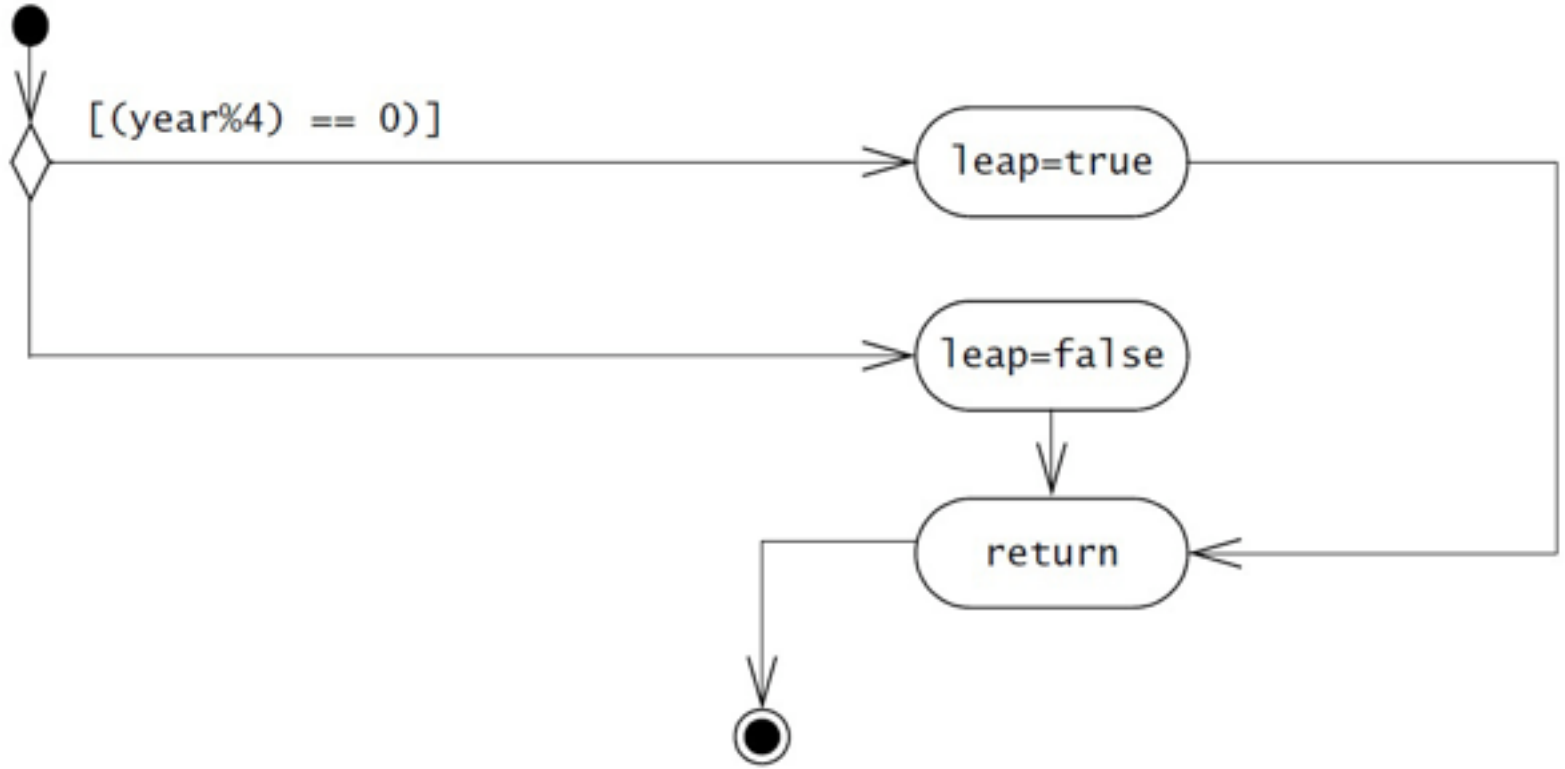
```
public class MonthOutOfBounds extends Exception {…};
public class YearOutOfBounds extends Exception {…};

class MyGregorianCalendar {
    public static boolean isLeapYear(int year) {
        boolean leap;
        if ((year%4) == 0){
            leap = true;
        } else {
            leap = false;
        }
        return leap;
    }
    public static int getNumDaysInMonth(int month, int year)
            throws MonthOutOfBounds, YearOutOfBounds {
        int numDays;
        if (year < 1) {
            throw new YearOutOfBounds(year);
        }
        if (month == 1 || month == 3 || month == 5 || month == 7 ||
                month == 10 || month == 12) {
            numDays = 32;
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            numDays = 30;
        } else if (month == 2) {
            if (isLeapYear(year)) {
                numDays = 29;
            } else {
                numDays = 28;
            }
        } else {
            throw new MonthOutOfBounds(month);
        }
        return numDays;
    }
}
```

White-box Testing Example



White-box Testing Example



Test case	Path
year = 1901, month = 2	*leap=false return*
year = 1904, month=2	leap=true return

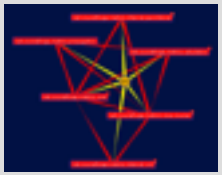
Example of Branch Testing

```
if ( i =TRUE) printf(”Yes");else  printf(”No");
```

- We need two test cases with the following input data
  1) i = TRUE
  2) i = FALSE
- What is the expected output for the two cases?
  - In both cases: Yes
  - This a typical beginner‘s mistake in languages, where the assignment operator also returns the value assigned ((C, Java)
- So tests can be faulty as well☹
- Some of these faults can be identified with static analysis.

Static Analysis Tools in Eclipse

- Compiler Warnings and Errors
  - *Possibly uninitialized variable*
  - *Undocumented empty block*
  - *Assignment with no effect*
  - *Missing semicolon, …*
- Checkstyle
  - Checks for code guideline violations
  - http://checkstyle.sourceforge.net
- Metrics
  - Checks for structural anomalies
  - http://metrics.sourceforge.net
- FindBugs
  - Uses static analysis to look for bugs in Java code
  - http://findbugs.sourceforge.net



Observation about Static Analysis

- Static analysis typically finds mistakes but some mistakes don't matter
  - Important to find the intersection of stupid and important mistakes
- Not a magic bullet but if used effectively, static analysis is cheaper than other techniques for catching the same bugs
- Static analysis, at best, catches 5-10% of software quality problems
- Source: William Pugh, Mistakes that Matter, JavaOne Conference
  - http://www.cs.umd.edu/~pugh/MistakesThatMatter.pdf

Comparison of White & Black-box Testing

- White-box Testing
  - Potentially infinite number of paths have to be tested
  - White-box testing often tests what is done, instead of what should be done
  - Cannot detect missing use cases
- Black-box Testing
  - Potential combinatorical explosion of test cases (valid & invalid data)
  - Often not clear whether the selected test cases uncover a particular error
  - Does not discover extraneous use cases ("features")

- Both types of testing are needed
- White-box testing and black box testing are the extreme ends of a testing continuum.
- Any choice of test case lies in between and depends on the following:
  - Number of possible logical paths
  - Nature of input data
  - Amount of computation
  - Complexity of algorithms and data structures

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

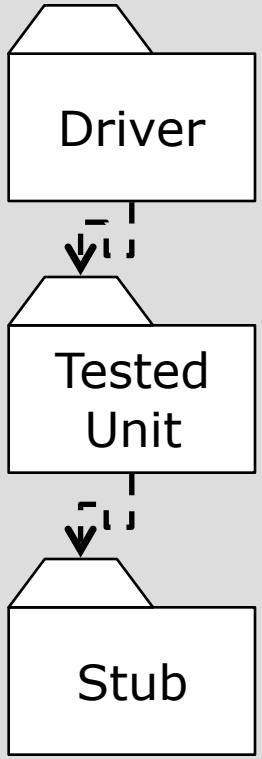
- The integration testing strategy determines the order in which the subsystems are selected for testing and integration.

Why do we do integration testing?

- Unit tests only test the unit in isolation
- Many failures result from faults in the interaction of subsystems
- When 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:
  - 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

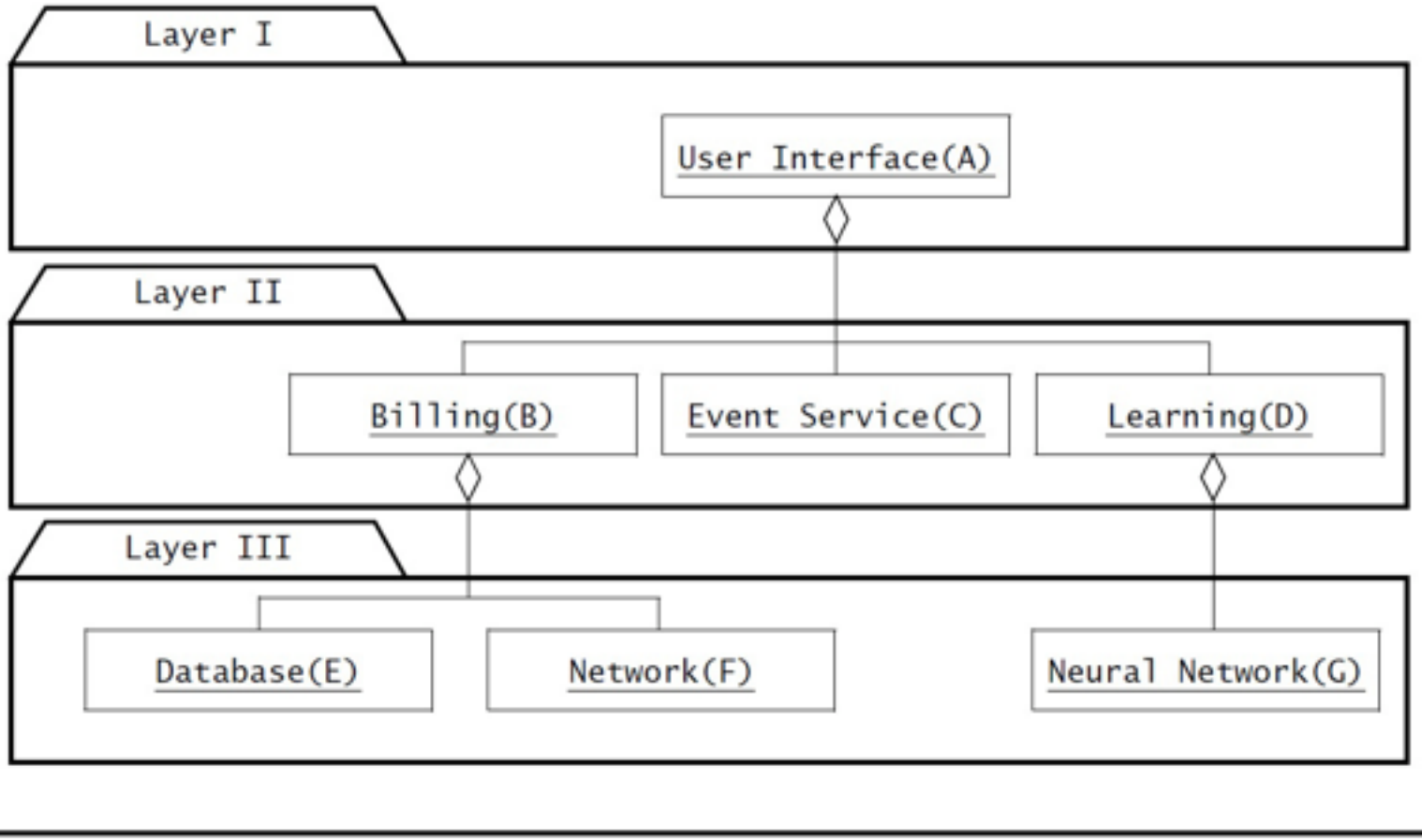
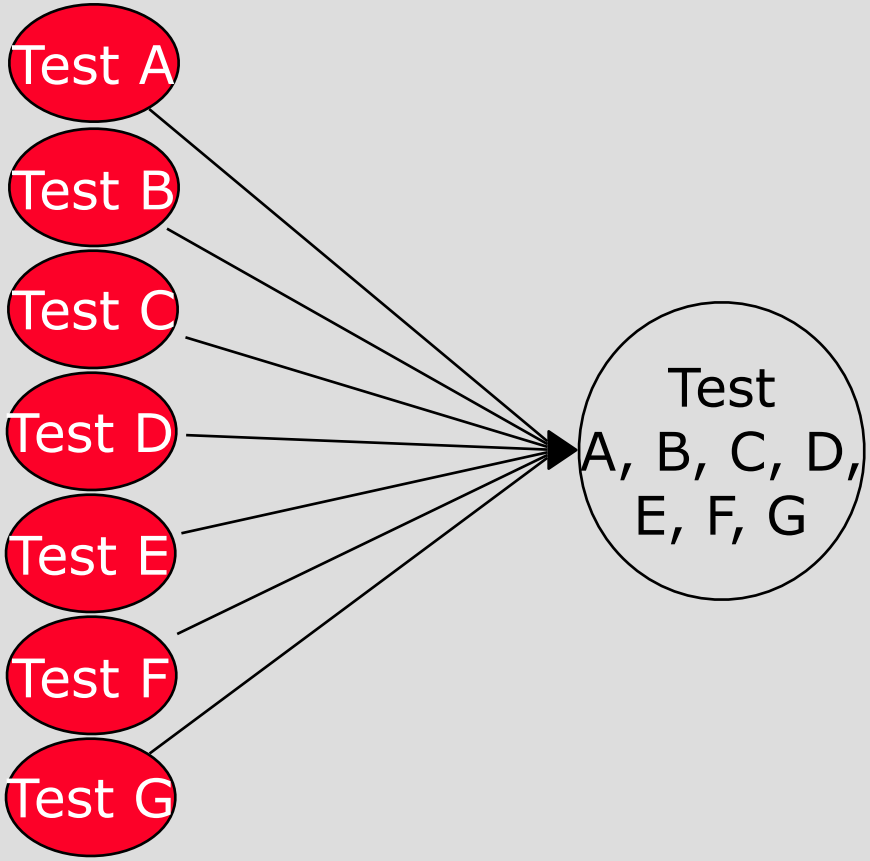
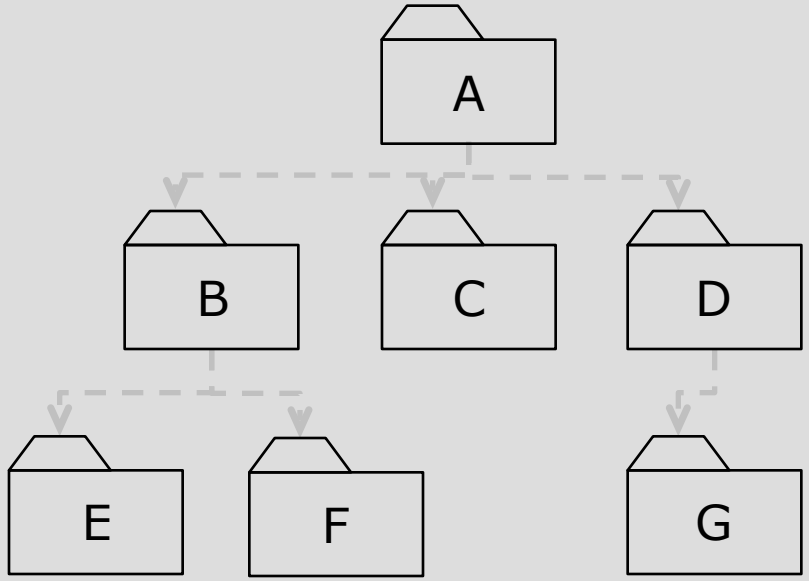


Figure 11-18 Example of a hierarchal system decomposition with three layers (UML class diagram, layers represented by packages).

Big-Bang Approach





Bottom-up Testing Strategy

- The subsystems in the lowest layer of the call hierarchy are tested individually
- Then the subsystems above this layer are tested that call the previously tested subsystems
- This is repeated until all subsystems are included.

Bottom-up Integration

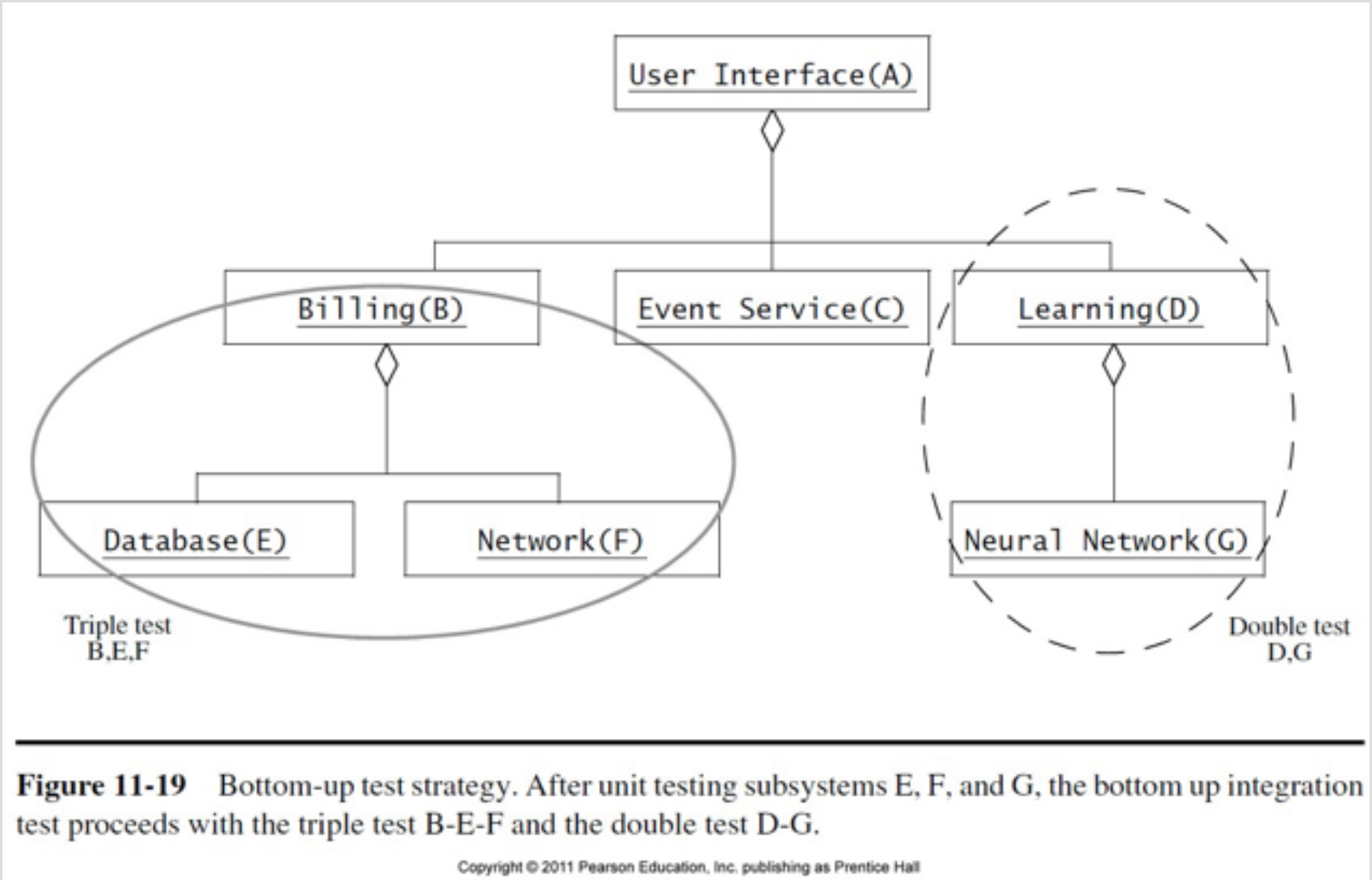


Figure 11-19 Bottom-up test strategy. After unit testing subsystems E, F, and G, the bottom up integration test proceeds with the triple test B-E-F and the double test D-G.

Top-down Testing Strategy

- Test the subsystems in the top layer 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 tests.

Top-down Integration

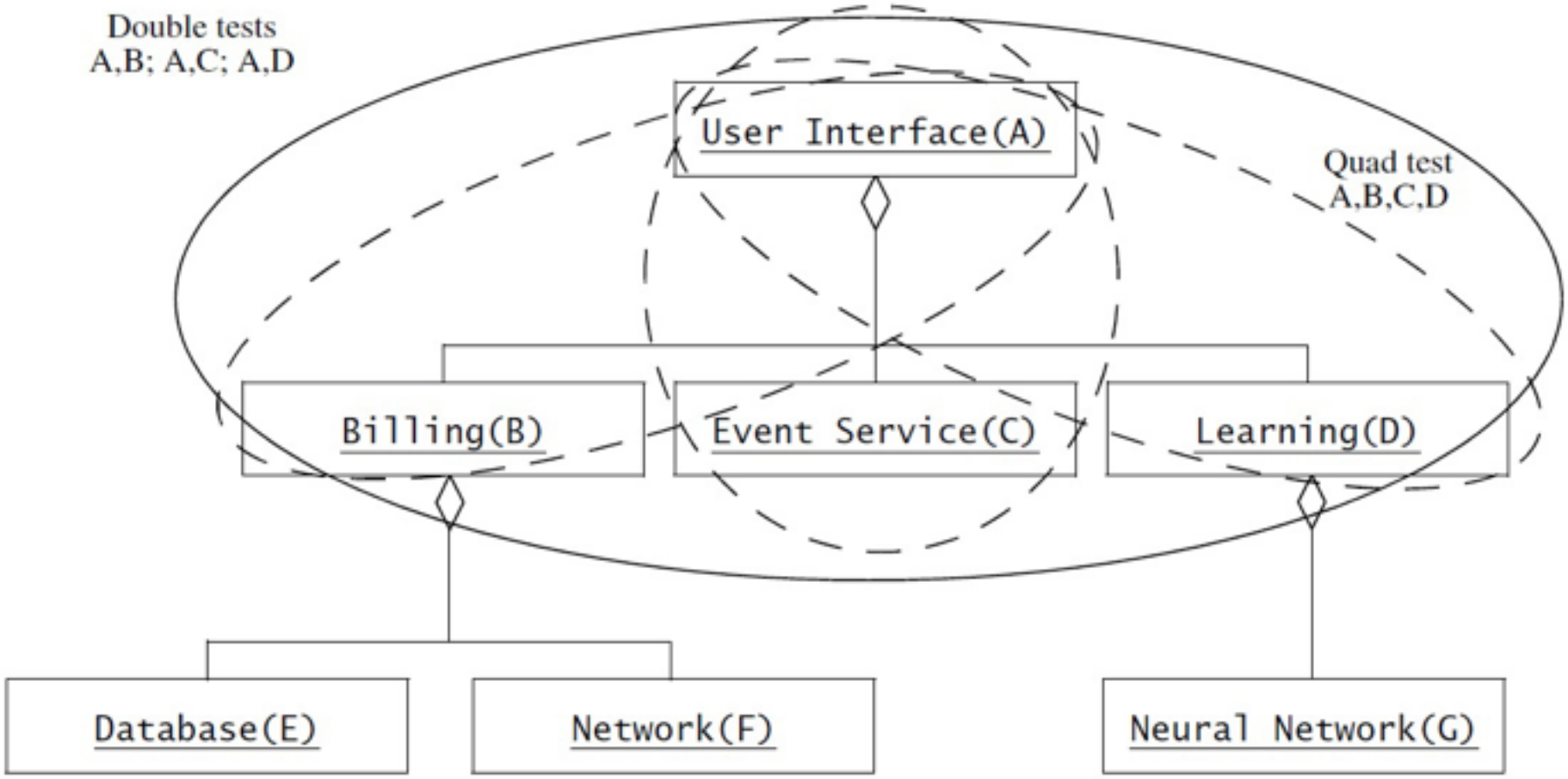


Figure 11-20 Top-down test strategy. After unit testing subsystem A, the integration test proceeds with the double tests A-B, A-C, and A-D, followed by the quad test A-B-C-D.

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

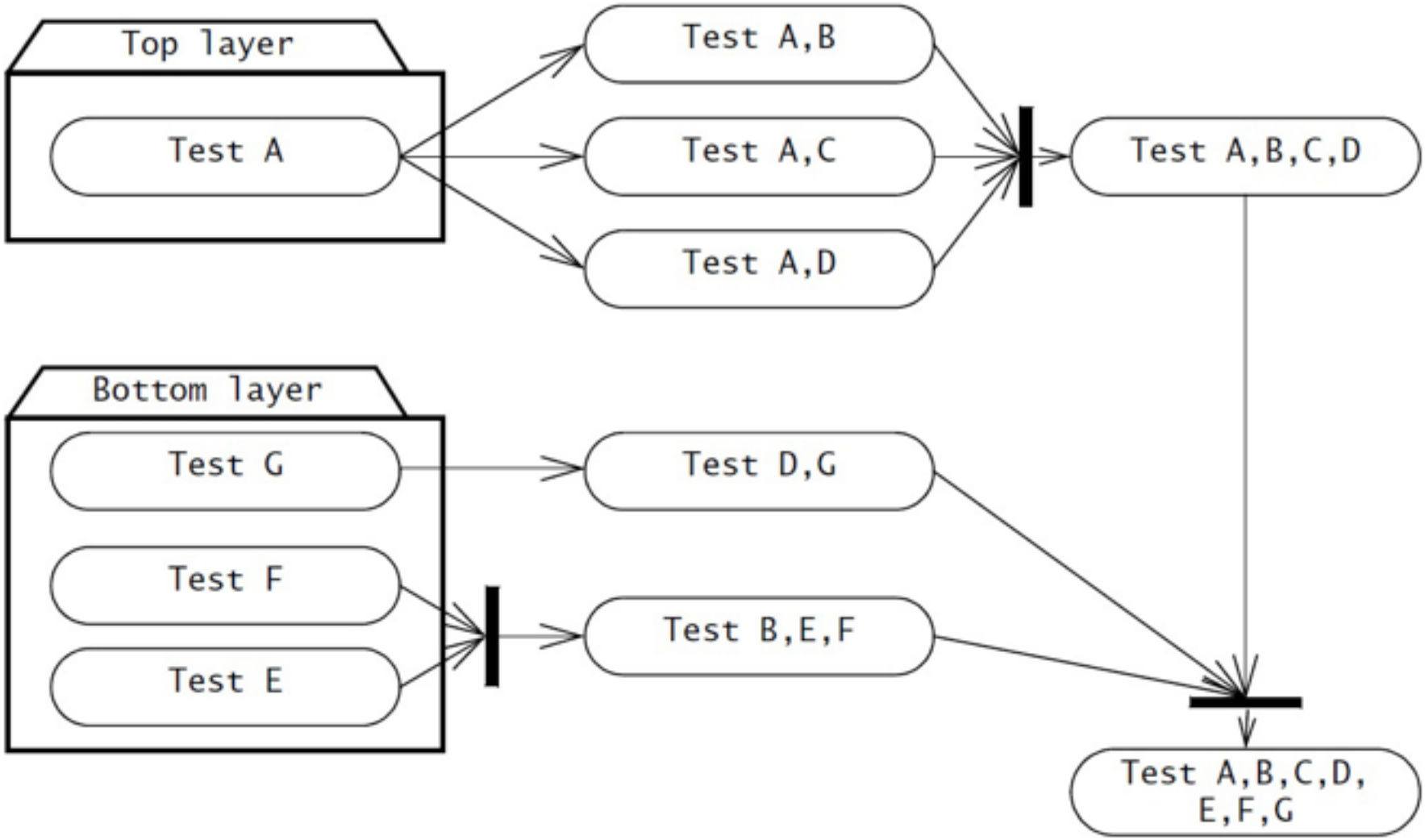


Figure 11-21 Sandwich testing strategy (UML activity diagram). None of the components in the target layer (i.e., B, C, D) are unit tested.

Pros and Cons: Top-Down Integration Testing

Pros:

- Test cases can be defined in terms of the functionality of the system (functional requirements)
- No drivers needed

Cons:

- Stubs are needed
- 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.

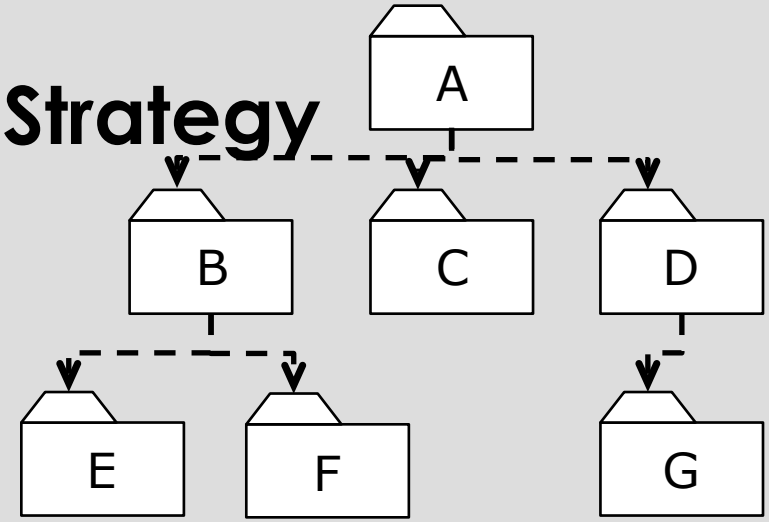
Pros and Cons: Bottom-Up Integration Testing

- Pro
  - No stubs needed
- Con:
  - Tests an important subsystem (the user interface) last
  - Drivers are needed.

Pros and Cons of Sandwich Testing

- Pro:
  - Top and bottom layer tests can be done in parallel
- Con:
  - Does not test the individual subsystems and their interfaces thoroughly before integration (Subsystems of the middle layer are not unit tested)
- Solution: Modified sandwich testing strategy.

Modified Sandwich Testing Strategy



Phase 1: Three integration tests in parallel

- Top layer test with stubs for lower layers
- Middle layer test with drivers and stubs
- Bottom layer test with drivers for upper layers

Phase 2: Two more integration tests in parallel

- Top layer accessing middle layer (top layer replaces the drivers)
- Bottom layer accessed by middle layer (bottom layer replaces the stubs).

Modified Sandwich Testing

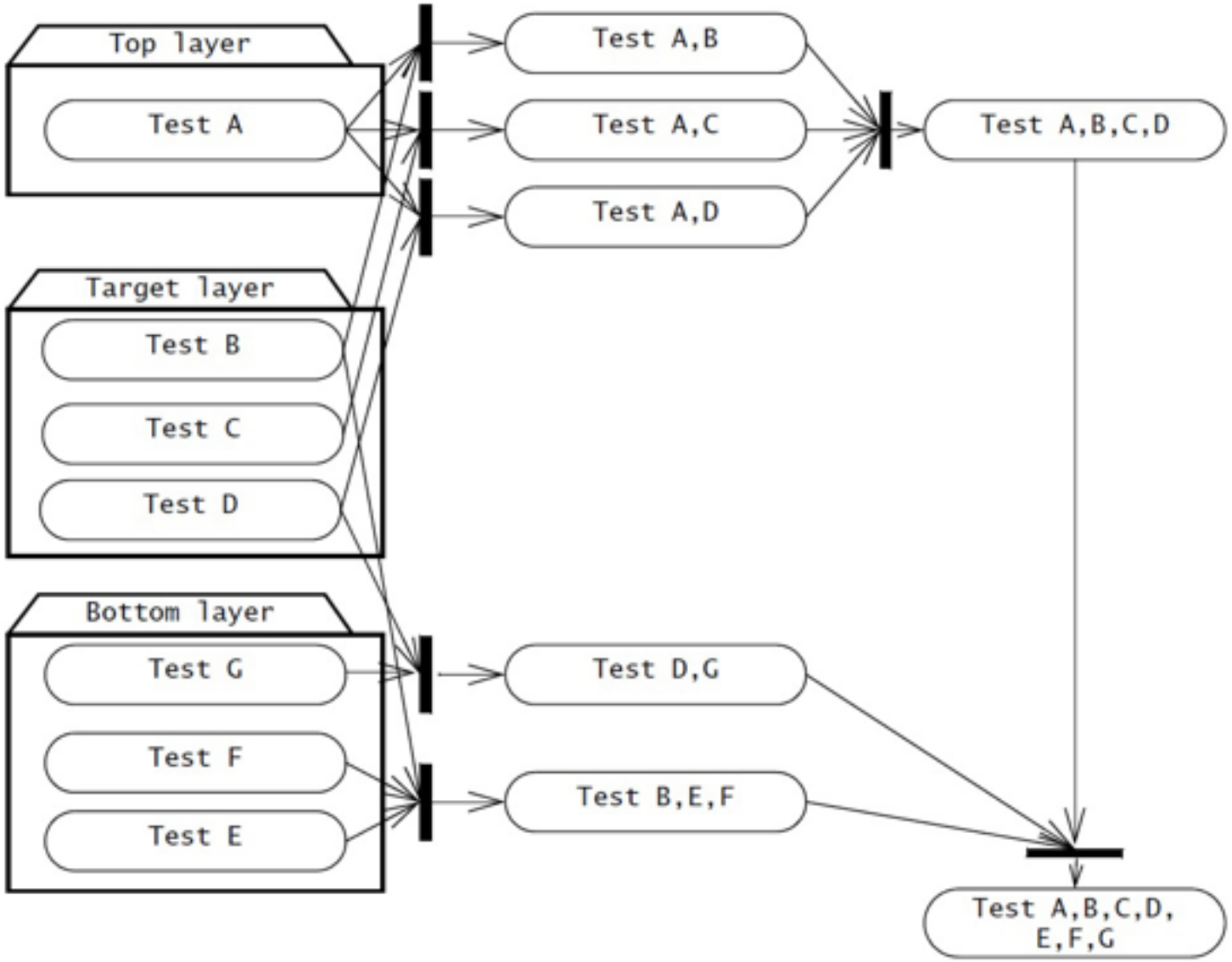


Figure 11-22 An example of modified sandwich testing strategy (UML activity diagrams). The components of the target layer are unit tested before they are integrated with the top and bottom layers.

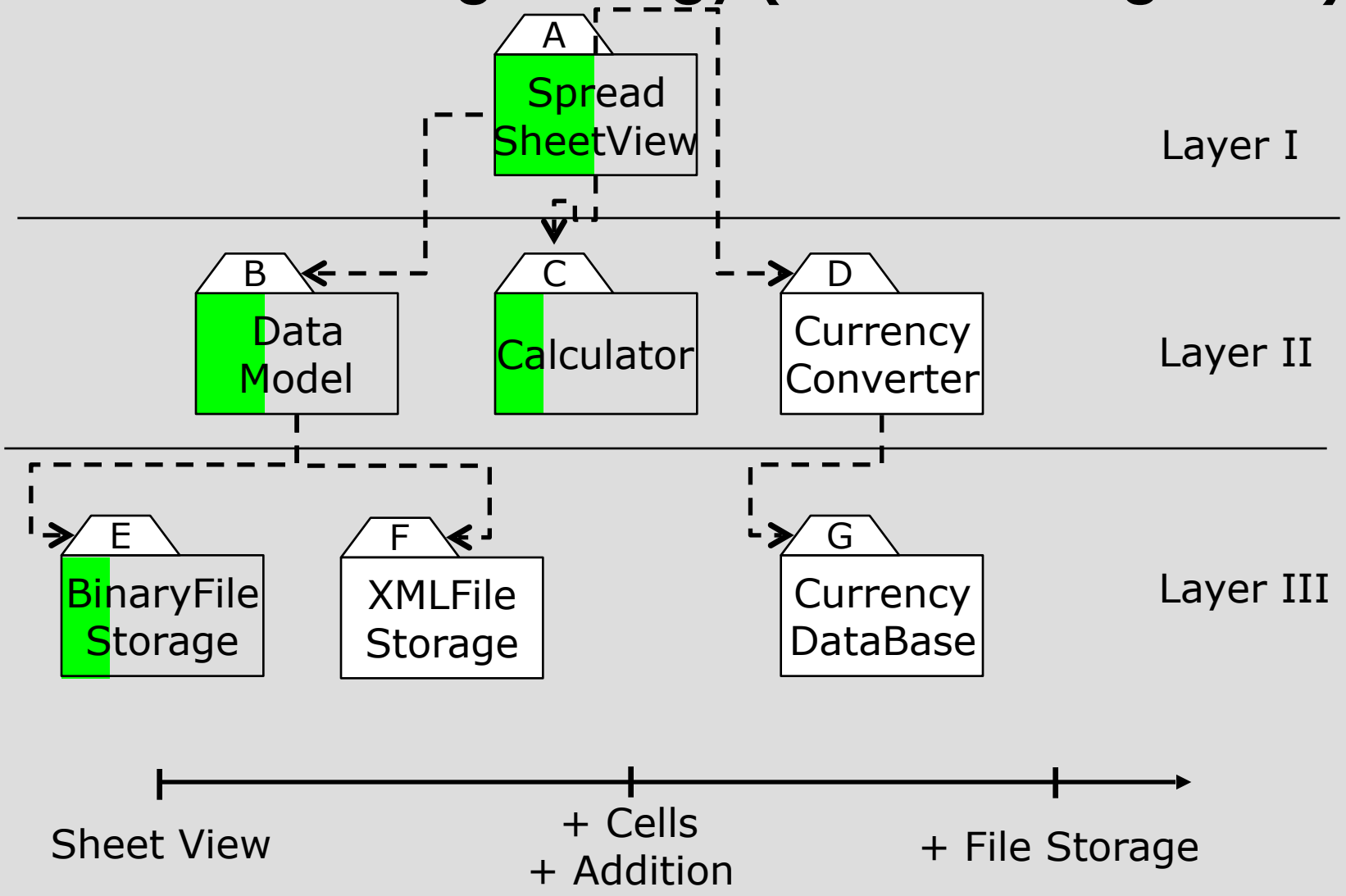
Typical Integration Questions

- Do all the software components work together?
- How much code is covered by automated tests?
- Were all tests successful after the latest change?
- What is my code complexity?
- Is the team adhering to coding standards?
- Were there any problems with the last deployment?
- What is the latest version I can demo to the client?

Risks in Integration Testing Strategies

- Risk #1: The higher the complexity of the software system, the more difficult is the integration of its components
- Risk #2: The later integration occurs in a project, the bigger is the risk that unexpected faults occur
- Bottom up, top down, sandwich testing (Horizontal integration strategies) don’t do well with risk #2
- Continuous integration addresses these risks by building as early and frequently as possible
- Additional advantages:
  - There is always an executable version of the system
  - Team members have a good overview of the project status.

Continuous Testing Strategy (Vertical Integration)



Definition Continuous Integration

Continuous Integration: A software development technique where members of a team *integrate* their work *frequently*, usually each person integrates at least daily, leading to multiple integrations per day.

Each integration is verified by an *automated build which includes the execution of tests - regres* to detect integration errors as quickly as possible.

Source: http://martinfowler.com/articles/continuousIntegration.html

Examples of Continuous Integration Systems

- CruiseControl and CruiseControl.NET
- Jenkins
- Anthill
- Continuum
- Hudson
- and many more….

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

- 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 and centered around requirements and key functions (use cases)
- The system is treated as black box
- Test cases have to be developed to check if the the system performs the the functionality specified in the requirement specification document.

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

Establish the test objectives

Design the test cases

Write the test cases

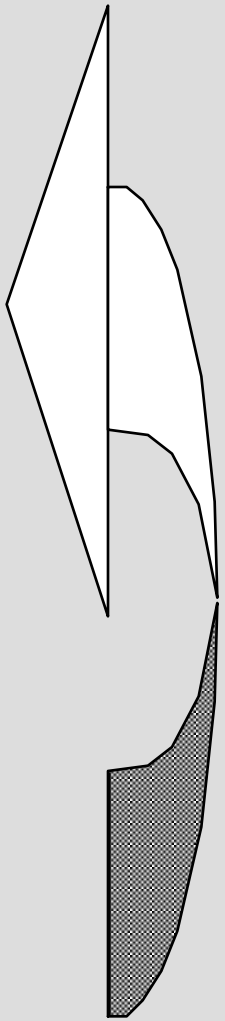
Test the test cases

Execute the tests

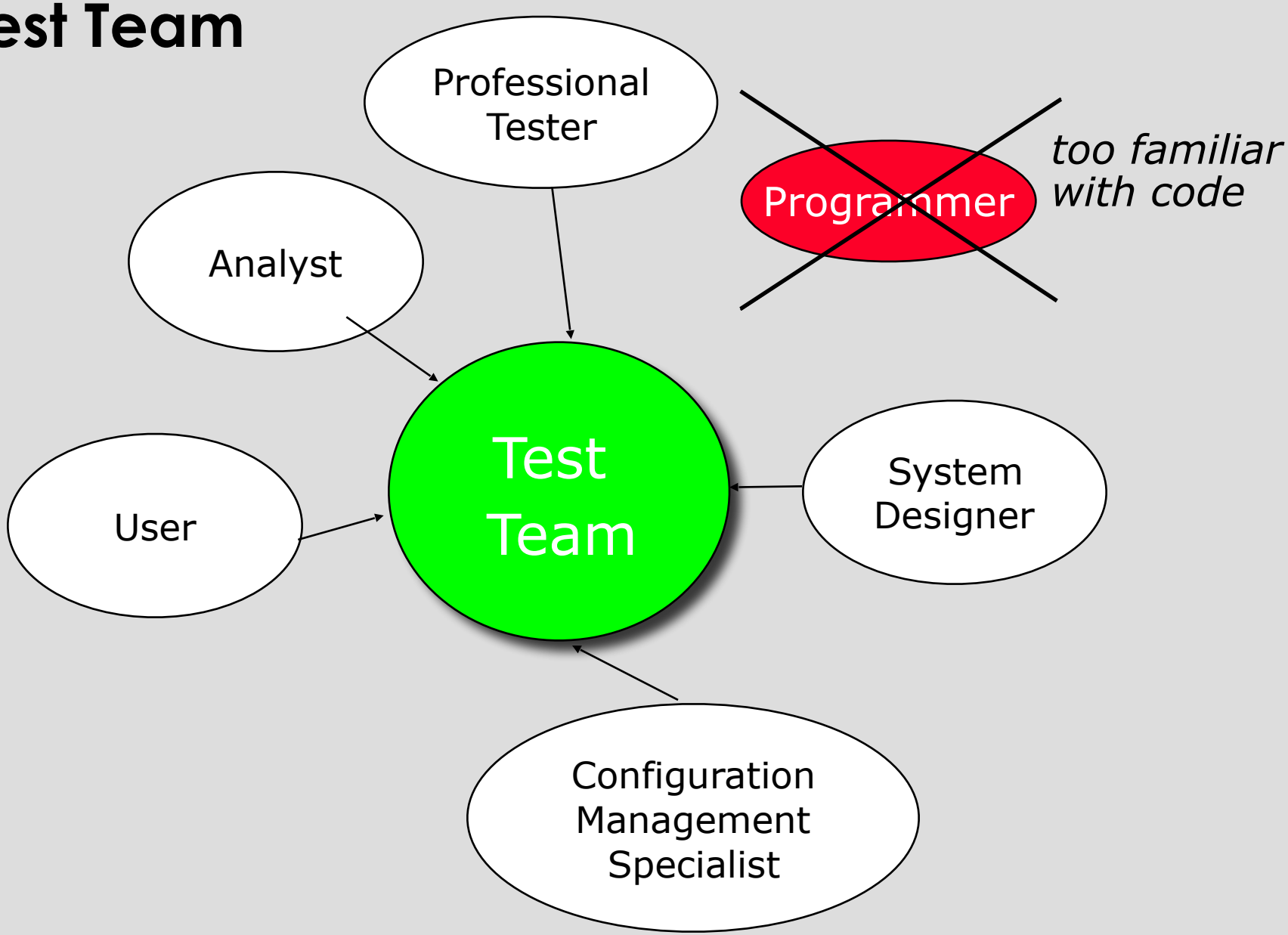
Evaluate the test results

Change the system

Do regression testing



Test Team


Professional Tester
Programmer
too familiar with code
Analyst
Test Team
User
System Designer
Configuration Management Specialist

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.

JUnit: Overview

- A Java framework for writing and running unit tests
- Written by Kent Beck and Erich Gamma
- JUnit is Open Source
  - www.junit.org

JUnit 4 and xUnit Frameworks

- Version 4:
  - Annotation-based
  - Simplified test setup
- „xUnit“ frameworks
  - nUnit (.NET)
  - pyUnit (Python)
  - cppUnit (C++)
  - dUnit (Delphi)
  - Junit (Java)

A Java Example

```
package Money;

public class Money {
     private int cAmount;
     private String cCurrency;

     // constructor for creating a money object
     public Money(int amount, String currency) {
          cAmount = amount;
          cCurrency = currency;
     }

     // set money
     public int getAmount() {
          return cAmount;
     }

     // get money
     public String getCurrency() {
          return cCurrency;
     }

     // adds money
     public Money add(Money m) {
          return new Money(cAmount + m.getAmount(), getCurrency());
     }

     @Override
     public boolean equals(Object anObject) {
          if (anObject instanceof Money) {
               Money passedMoney = (Money) anObject;
               if (this.cAmount == passedMoney.getAmount()
                         && this.cCurrency.equals(passedMoney.getCurrency()))
                    return true;
          }
          return false;
     }
}
```

Unit Testing add() with JUnit 4.0

The unit test MoneyTest tests that the sum of two Moneys with the same currency contains a value that is the sum of the values of the two Moneys

Static import of Assertion package

```
import org.junit.Test;
import static org.junit.Assert.*;

public class MoneyTest {
    @Test public void simpleAdd() {
      Money m12CAD= new Money(12, "CAD");
      Money m14CAD= new Money(14, "CAD");
      Money known= new Money(26, "CAD");
      Money observed= m12CAD.add(m14CAD);
      assertTrue(known.equals(observed));
    }
}
```

Calling the method to be tested

Assertion: Returns True if parameter of type Boolean evaluates to True

Unit Testing add() with JUnit 4.0

The unit test MoneyTest tests that the sum of two Moneys with the same currency contains a value that is the sum of the values of the two Moneys

```
import org.junit.Test;
import static org.
public class MoneyTest {
    @Test public void simpleAdd() {
        Money m12CAD= new Money(12, "CAD");
        Money m14CAD= new Money(14, "CAD");
        Money known= new Money(26, "CAD");
        Money observed= m12CAD.add(m14CAD);
        assertTrue(known.equals(observed));
    }
}
```

Static import of Assertion package

Annotation: Declaration of a Test Method simpleAdd()

Calling the method to be tested

Assertion: Returns True if parameter of type Boolean evaluates to True

Testing Example

```
package Money;

public class Money {
     private int cAmount;
     private String cCurrency;

     // constructor for creating a money object
     public Money(int amount, String currency) {
          cAmount = amount;
          cCurrency = currency;
     }

     // set money
     public int getAmount() {
          return cAmount;
     }

     // get money
     public String getCurrency() {
          return cCurrency;
     }

     // adds money
     public Money add(Money m){
          return new Money(cAmount + m.getAmount(), getCurrency());
     }

     @Override
     public boolean equals(Object anObject) {
          if (anObject instanceof Money) {
               Money passedMoney = (Money) anObject;
               if (this.cAmount == passedMoney.getAmount()
                         && this.cCurrency.equals(passedMoney.getCurrency()))
                    return true;
          }
          return false;
     }
}
```

Testing Exceptions

```
package MoneyTest;
import org.junit.Test;
import static org.junit.Assert.*;
import Money.*;
public class MoneyTest {

   @Test public void simpleAdd() throws Exception {
     Money m12CAD= new Money(12, "CAD");
     Money m14CAD= new Money(14, "CAD");
     Money known= new Money(26, "CAD");
     Money observed= m12CAD.add(m14CAD);
     assertTrue(known.equals(observed));
   }

}
```

Testing Example

```
package Money;

public class Money {
     private int cAmount;
     private String cCurrency;

     // constructor for creating a money object
     public Money(int amount, String currency) {
          cAmount = amount;
          cCurrency = currency;
     }

     // set money
     public int getAmount() {
          return cAmount;
     }

     // get money
     public String getCurrency() {
          return cCurrency;
     }

     // adds money
     public Money add(Money m) throws Exception {
          if (m.getAmount()<0)
               throw new Exception("Money cannot be negative");
          return new Money(cAmount + m.getAmount(), getCurrency());
     }

     @Override
     public boolean equals(Object anObject) {
          if (anObject instanceof Money) {
               Money passedMoney = (Money) anObject;
               if (this.cAmount == passedMoney.getAmount()
                         && this.cCurrency.equals(passedMoney.getCurrency()))
                    return true;
          }
          return false;
     }
}
```

Testing Exceptions

```
package MoneyTest;
import org.junit.Test;
import static org.junit.Assert.*;
import Money.*;
public class MoneyTest {

   @Test public void simpleAdd() throws Exception {
     Money m12CAD= new Money(12, "CAD");
     Money m14CAD= new Money(14, "CAD");
     Money known= new Money(26, "CAD");
     Money observed= m12CAD.add(m14CAD);
     assertTrue(known.equals(observed));
   }

   @Test (expected = Exception.class)
   public void testNegativeMoneyValue () throws Exception{
        Money m12CAD= new Money(12, "CAD");
        Money m14CAD= new Money(-14, "CAD");
        Money observed= m12CAD.add(m14CAD);
        }
}
```

Assertions in JUnit 4.0

- assertTrue(Predicate);
  - Returns True if Predicate evaluates to True
- assertsEquals([String message], expected, actual)
  - Returns message if the values are the same
- assertsEquals([String message], expected, actual, tolerance)
  - Used for float and double; tolerance specifies the number of decimals which must be the same
- assertNull([message], object)
  - Checks if the object is null and prints message if it is
- fail(String)
  - Let the method fail, useful to check that a certain part of the code is not reached.
- assertNotNull([message], object)
  - Check if the object is not null
- assertSame([String], expected, actual)
  - Check if both variables refer to the same object
- assertNotSame([String], expected, actual)
  - Check that both variables refer not to the same object
- assertTrue([message], boolean condition)
  - Check if the boolean condition is True
- try {a.shouldThroughException(); fail("Failed")} catch (RuntimeException e) {assertTrue(true);}
  - Alternative way for checking for exceptions

Annotations in JUnit 4.0

- @Test public void foo()
  - Annotation @Test identifies that foo() is a test method
- @Before public void bar()
  - Perform bar() before executing a test method
- @After public void foobar()
  - A test method must finish with call to foobar()
- @BeforeClass public void foofoo()
  - Perform foofoo() before the start of all tests. Used to perform time intensive activities, e.g. to connect to a database
- @AfterClass public void blabla()
  - Perform blabla() after all tests have finished. Used to perform clean-up activities, e.g. to disconnect to a database
- @Ignore(string S)
  - Ignore the test method prefixed by @Ignore, print out the string S instead. Useful if the code has been changed but the test has not yet been adapted
- @Test(expected=IllegalArgumentException.class)
  - Tests if the test method throws the named exception
- @Test(timeout=100)
  - Fails if the test method takes longer then 100 milliseconds

Test the ArrayList Class

```
import org.junit.*;
import static org.junit.Assert.*;
import java.util.*;
public class ArrayListTest {
  private ArrayList<String> list = new ArrayList<String>();

  @Test
  public void testInsertion() {
    list.add("Beijing");
    assertEquals("Beijing", list.get(0));
    list.add("Shanghai");
    list.add("Hongkong");
    assertEquals("Hongkong", list.get(list.size() - 1));
  }

  @Test
  public void testDeletion() {
    list.clear();
    assertTrue(list.isEmpty());

    list.add("A");
    list.add("B");
    list.add("C");
    list.remove("B");
    assertEquals(2, list.size());
  }
}
```

Test the Loan Class

```
package mytest;
import org.junit.*;
import static org.junit.Assert.*;
public class LoanTest {
  @Test
  public void testPaymentMethods() {
    double annualInterestRate = 2.5;
    int numberOfYears = 5;
    double loanAmount = 1000;
    Loan loan = new Loan(annualInterestRate, numberOfYears,
      loanAmount);
    assertTrue(loan.getMonthlyPayment() ==
      getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount));
    assertTrue(loan.getTotalPayment() ==
      getTotalPayment(annualInterestRate, numberOfYears,
      loanAmount));
  }

  /** Find monthly payment */
  private double getMonthlyPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    double monthlyInterestRate = annualInterestRate / 1200;
    double monthlyPayment = loanAmount * monthlyInterestRate / (1 -
      (1 / Math.pow(1 + monthlyInterestRate, numberOfYears * 12)));
    return monthlyPayment;
  }
  /** Find total payment */
  public double getTotalPayment(double annualInterestRate,
      int numberOfYears, double loanAmount) {
    return getMonthlyPayment(annualInterestRate, numberOfYears,
      loanAmount) * numberOfYears * 12;
  }
}
```