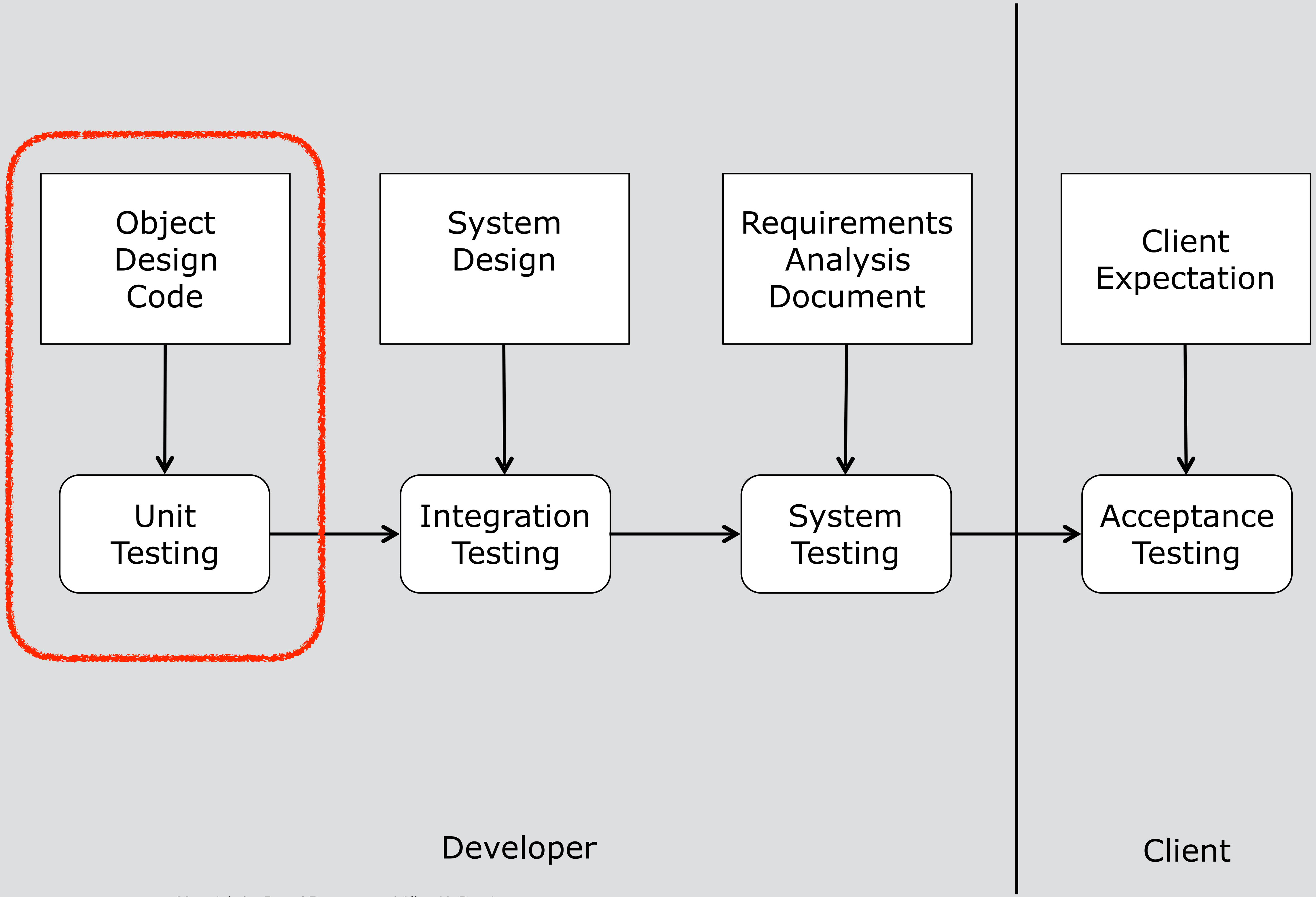
COIS2240 Lecture 9

Famous Problems

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

- 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
- NASA Mars Climate Orbiter destroyed due to incorrect orbit insertion (September 23, 1999)
  - Reason: Unit conversion problem.

Testing Activities



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 developers
  - Goal: Test the interfaces among the subsystems.
- System Testing
  - The entire system
  - 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.

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

```java
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

Static import of Assertion package

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

Annotation: Declaration of a Test Method simpleAdd()

```
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

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

```java
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;
  }
}
```