JUnit Testing Framework Architecture

- **Unit under test** (usually a class or a small number of classes)
- **Test environment** (fixture)
  - Program state (e.g., some collection of variables/objects that will be used in several test cases)
  - JUnit provides facilities to maintain the test environment for many test cases
- **Tests**
  - Test case: A single test that basically checks preconditions/postconditions
  - Test suite: Collection of multiple test cases
  - JUnit provides facilities to create individual test cases and to combine test cases into test suites.
- **Test execution**
  - JUnit automatically runs a set of test cases or test suites
  - JUnit checks the correctness of the test cases and produces a test report, indicating test cases that passed, test cases that failed and some summary statistics (e.g., total number of failed test cases)

Example: Account.java

```java
package banking;
import banking.exceptions.InsufficientFundsException;
public class Account {
    private int bal;
    public Account(int amt) {
        bal = amt;
    }
    public Account(Account acc) {
        bal = acc.balance();
    }
    public void transfer(int amt, Account acc) throws InsufficientFundsException {
        acc.withdraw(amt);
        deposit(amt);
    }
    public void withdraw(int amt) throws InsufficientFundsException {
        if (checkSufficientFunds(amt))
            bal -= amt;
        else
            throw new InsufficientFundsException("Insufficient funds.");
    }
    public void deposit(int amt) {
        bal += amt;
    }
    public int balance() {
        return bal;
    }
    private boolean checkSufficientFunds(int amt) {
        if (bal >= amt)
            return true;
        else
            return false;
    }
}
```

Writing Tests with JUnit4: Initial Preparation

- Create a new Java class that will contain individual test cases.
  - Suppose the name of the class you want to test is “Foo”. By convention, the name of the class that contains the test cases for Foo should be FooTest.
- Include the following imports:
  - import org.junit.*
  - import static org.junit.Assert.*

```java
package banking.tests;
import org.junit.*;
import static org.junit.Assert.*;
import banking.SavingsAccount;
public class SavingsAccountTest {
    ...
}
```

Example: SavingsAccount.java

```java
package banking;
public class SavingsAccount extends Account{
    double interestRate;
    public SavingsAccount(double amt) {
        super(amt);
        interestRate = 0.01;
    }
    public void setInterestRate(double r) {
        interestRate = r;
    }
    public double getInterestRate() {
        return interestRate;
    }
    public void accrueInterest() {
        bal = bal + bal * interestRate;
    }
}
```

Writing Tests with JUnit4: Preparing the Test Environment (Test Fixture)

- Before/After annotation designates a method that deals with the test fixture:
  - @org.junit.Before – Sets up the objects in the test fixture (usually allocates the objects and sets their initial values)
  - @org.junit.After – Tears down the objects in the test fixture (usually “deallocates” the objects by setting their references to null)
- Important to execute both methods for each test case so that the test cases are isolated from each other; thus can execute the test cases in any order
Example : Preparing the Test Environment

```
package banking.tests;
import org.junit.*;
import static org.junit.Assert.*;
import banking.SavingsAccount;
public class SavingsAccountTest {
    private SavingsAccount acc;
    @Before
    public void setUp() {
        acc = new SavingsAccount(100);
    }
    @After
    public void tearDown() {
        acc = null;
    }
    ……
```

Example: Writing Test Cases, Nominal Behavior

```
@Test
public void testTransfer() throws InsufficientFundsException {
    Account acc2 = new Account(100);
    acc.transfer(100, acc2);
    assertTrue(acc.balance()==100);
    assertTrue(acc2.balance()==0);
}
//Note, this method assumes that the test fixture has created an account object “acc” with 100 dollars.
```

Example: Writing Test Cases, Exceptional Behavior

```
@Test(expected=InsufficientFundsException.class)
public void testTransfer_InsufficientFundsException() throws InsufficientFundsException {
    Account acc2 = new Account(100);
    acc.transfer(101, acc2);
}
//We expect that an exception will be thrown as we try to transfer 101 dollars from acc2, which has only 100 dollars in it.
```

General Tips for Writing Test Cases

- Let m be a method under test
- A method that is a test case for m is usually defined as follows:
  - Checks any preconditions of m (sometimes ignored)
  - Invokes m
  - Checks any postconditions of m

JUnit4 Pre/Postconditions: Assert class

- org.junit.Assert provides the assertX methods where X may be:
  - False/True
  - Null/NotNull
  - Same/NotSame
  - Equals
  - ...
- It also provides the fail method that is usually used to signal that an exception should have been thrown.

```
public void testIndexOutOfBoundsException() {
    Vector v = new Vector(10);
    try {
        Object o = v.elementAt(v.size());
        fail("Should raise an ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {
    }
```

Writing Tests with JUnit4: Writing Test Cases

- "@Test" annotation designates a method that is a test case
  - @org.junit.Test: Nominal behavior expected (i.e. an exception is NOT expected to be thrown)
  - @org.junit.Test(expected=MyException.class): Exceptional behavior expected (i.e. an exception is expected to be thrown)
- Suppose you want to test method foo. By convention, the method that will test foo should be named testFoo.
Writing Tests with JUnit4: Test Suite

- A test suite may be composed of:
  - Tests cases
  - Other test suites

- A test suite is defined as a class or a set of classes
  - Single class test suite
  - Multiple class test suite

Criteria for grouping test cases

- Single class test suite usually contains tests for the methods of a single class from the software system
  - e.g., the AccountTest class contains tests for the methods of the Account.java class.
  - If class under test is too large, additional test grouping strategies can be used -- e.g., exceptional vs. normal behavior, based on fixture reuse.
- Multiple class test suites can contain all tests related to a given package from the software system
  - e.g., OverallBankingTest class contains tests for the methods of all classes in the banking package.

Example: Single Class Test Suite

```
package banking.tests;
import org.junit.*;
import static org.junit.Assert.*;
import banking.Account;
import banking.exceptions.InsufficientFundsException;
public class AccountTest {
    private Account acc;
    @Before
    public void setUp() {
        ...
    }
    @After
    public void tearDown() {
        ...
    }
    @Test
    public void testAccountConstructor() {
        ...
    }
    @Test
    public void testDeposit() {
        ...
    }
    ...
}
```

Example: Multiple Class Test Suite

```
package banking.tests;
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
/**
 * This class serves as a "master" test suite, i.e.
 * it runs the test cases from several other test suites (classes).
 */
@RunWith(Suite.class)
@SuiteClasses(value={AccountTest.class, SavingsAccountTest.class})
public class OverallBankingTester {
    ...
}
```

Running Tests: Test Case Pass/Fail Semantics

- For a given test suite, all methods whose annotation starts with @Test will be run
  - @Test: Nominal behavior
    - When all assertX method invocations succeed and no exception is thrown: Succeeds
    - Otherwise: Fails
  - @Test(expected=MyException.class): Exceptional behavior
    - When all assertX method invocations succeed and an exception of class MyException.class is thrown: Succeeds
    - Otherwise: Fails

Running Tests: Test Results

- Test results include:
  - Number of test cases that were run
  - Number of test cases that failed
  - For each test case that failed, details about how it failed

Running Tests: JUnit tools

- JUnit4 comes with standard installations of Eclipse
  - Make sure JUnit4 is on the Java Build Path of your Eclipse project
  - Right click on the class that contains your test cases (either single class test suite or multiple class test suite) and run JUnit4.
- Command line
  - `java org.junit.runner.JUnitCore TestClass1 [...other test classes...]`

**Demo**

- Unit testing simple banking system
- Unit testing the finite-state automaton system

**Software used**
- Eclipse 3.4.1
- JUnit 4.3.1 (Eclipse plugin)
- dJUnit 0.8.5 (Eclipse plugin)

**References**

- JUnit
  - JUnit web site: [http://www.junit.org/](http://www.junit.org/)
- Coverage Tool
  - dJUnit: [http://works.deic.co.jp/djunit/](http://works.deic.co.jp/djunit/)
  - supports statement and branch coverage