COM6050
Java and UML for Programmers

Lecture 19: (J)Unit Testing

Steve Renals
http://www.dcs.shef.ac.uk/∼sjr/com6050

05.12.2002

Objectives

An overview of JUnit, a simple framework for repeatable tests.

“Never in the field of software development was so much owed by so many to so few lines of code” – Martin Fowler

Reading:
See the JUnit website, http://www.junit.org, to download JUnit, and to get these documents:

- *Test Infected: Programmers Love Writing Tests*
- *JUnit: A cook’s tour*
- *JUnit cookbook*

Why Test

- Working code is tested code — you be confident that a method performs as advertised if you encode its contract in a test — with enough test cases
- Failure is an opportunity to learn — a failed test provides important information
- Writing test cases is faster then debugging...
- You are never too busy to write tests — tests improve your productivity

JUnit

- JUnit is a regression testing framework — repeated testing of a software system, to ensure that any bugs have been fixed, no previously working methods have been broken as a result of changes, and that newly added features have not created problems with the existing software
- JUnit is closely linked to extreme programming
- Download JUnit from http://www.junit.org
- Documentation (on which this lecture is based) from the same website

The Example

- We would like a system to deal with money in multiple currencies
- In particular we would like to do arithmetic in multiple currencies
- Since there is no single exchange rate, and exchange rates change, we cannot just convert currencies
- Basic classes will be Money (single currency) and MoneyBag (a collection of money in different currencies).

Money (part 1)

```
public class Money {
    public Money(int amount, String currency) {
        this.amount = amount;
        this.currency = currency;
    }
    public int getAmount() { return amount; }
    public String getCurrency() { return currency; }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;

        Money m = (Money)obj;
        return amount == m.amount && currency.equals(m.currency);
    }
    public int hashCode() {
        return currency.hashCode() + 13*amount;
    }
    private int amount;
    private String currency;
}
```


MoneyTest

```
import junit.framework.*;

public class MoneyTest extends TestCase {
    public void testEquals() {
        Assert.assertTrue(!tenPounds.equals(null));
        Assert.assertEquals(tenPounds, tenPounds);
        Assert.assertEquals(tenPounds, new Money(10, "UKP"));
        Assert.assertTrue(!tenPounds.equals(eightPounds));
    }
    protected void setUp() {
        tenPounds = new Money(10, "UKP");
        eightPounds = new Money(8, "UKP");
    }
    public static Test suite() {
        return new TestSuite(MoneyTest.class);
    }
    private Money tenPounds;
    private Money eightPounds;

    public static void main(String args[]) {
        junit.textui.TestRunner.run(suite());
    }
}
```


Running the JUnit tests

1. Make sure junit.jar is in your classpath
2. Define a test class that extends junit.framework.TestCase
3. Override setUp() and tearDown() methods
4. Define the test suite — this can be done dynamically by a static suite method returning:
   new TestSuite(MoneyTest.**class**);
5. Can run the test suite in two ways:
   - Command line: main calls junit.textui.TestRunner.run(suite());
   - GUI: run `java junit.swingui.TestRunner`

Money.add

We would like a simple add method for Money, that doesn't deal different currencies, for now. We can write a test for it in MoneyTest:

```
public class MoneyTest extends TestCase {

    public void testSimpleAdd() {
        Money expected = new Money(18, "UKP");
        Money result = tenPounds.add(eightPounds);
        Assert.assertTrue(expected.equals(result));
    }
    ...
}
public class Money {
    ...
    public Money add(Money m) {
        return new Money(getAmount() + m.getAmount(), getCurrency());
    }
    ...
}
```


MoneyBag

It is time to move to multiple currencies. We define MoneyBag as being composed of a list of Money objects:

```
public class MoneyBag {
    public MoneyBag() { }
    public MoneyBag(Money m1, Money m2) {
        appendMoney(m1);
        appendMoney(m2);
    }
    public MoneyBag(Money mm[]) {
        for(int i = 0; i < mm.length; ++i)
            appendMoney(mm[i]);
    }
    private void appendMoney(Money m) {
        // append m to the set if a new Currency, otherwise add to the
        //      appropriate item
    }
    private SortedSet monies = new TreeSet();
    // Money should implement Comparable
    // needs to write a test for Money.compareTo()
}
```


MoneyBag.equals

```
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass()) return false;

    MoneyBag m = (MoneyBag)obj;
    if (monies.size() != m.monies.size())
        return false;
    Iterator i = monies.iterator();
    Iterator j = m.monies.iterator();
    boolean ret = true;
    while(ret && i.hasNext()) {
        Money m1 = (Money)i.next();
        Money m2 = (Money)j.next();
        ret &= m1.equals(m2);
    }
    return ret;
}
```


More Tests

```
public void testBagEquals() {
    Assert.assertFalse(bag5e10p.equals(null));
    Assert.assertEquals(bag5e10p, bag5e10p);
    Assert.assertEquals(bag5e10p, new MoneyBag(fiveEuros, tenPounds));
    Assert.assertEquals(bag5e10p, new MoneyBag(tenPounds, fiveEuros));
    Assert.assertFalse(bag5e10p.equals(bag12e8p));
    Assert.assertFalse(tenPounds.equals(bag5e10p));
    Assert.assertFalse(bag5e10p.equals(tenPounds));
}

protected void setUp() {
    ...
    bag5e10p = new MoneyBag(tenPounds, fiveEuros);
    bag12e8p = new MoneyBag(eightPounds, twelveEuros);
}
}
```

IMoney Interface

- We would like to have a generalized Money.add method, that gives us a MoneyBag if the argument is a different currency:

```
public Money add(Money m) {
    if (m.getCurrency().equals(currency))
        return new Money(getAmount() + m.getAmount(), getCurrency());
    return new MoneyBag(this, m);
}
```

But this will not compile.

- The solution is a general interface:

```
public interface IMoney {
  public IMoney add(IMoney);
  ...
}
```

which is implemented by Money and MoneyBag

IMoney

- Can immediately write a test case:

```
public void testSimpleMixedAdd() {
    Assert.assertEquals(bag5e10p, tenPounds.add(fiveEuros));
}
```

- And we also write test cases for:
  - Adding Money to a MoneyBag
  - Adding a MoneyBag to Money
  - Adding a MoneyBag to a MoneyBag
- Now we have the test cases we can write the code to pass the tests...

Generalized adding

```
public class Money implements IMoney, Comparable {
  ...
    public IMoney add(IMoney m) {
        return m.addMoney(this);
    }

    public IMoney addMoney(Money m) {
        if(m.getCurrency().equals(currency))
            return new Money(getAmount() + m.getAmount(), getCurrency());
        return new MoneyBag(this, m);
    }

    public IMoney addMoneyBag(MoneyBag b) {
        return b.addMoney(this);
    }
...
```



```
public class MoneyBag implements IMoney {
...
    public IMoney add(IMoney m) {
        return m.addMoneyBag(this);
    }

    public IMoney addMoney(Money m) {
        return new MoneyBag(m, this);
    }

    public IMoney addMoneyBag(MoneyBag b) {
        return new MoneyBag(b, this);
    }
...
```

More features

- We would like the following test to work:
  Assert.assertEquals(fiveEuros, bag5e10p.add(**new** Money(−10,"UKP")));
- Do this with a simplify() method used when addMoney or addMoneyBag returns a MoneyBag:

```
public class MoneyBag implements IMoney {
    public IMoney addMoney(Money m) {
        return new MoneyBag(m, this).simplify();
    }

    public IMoney addMoneyBag(MoneyBag b) {
        return new MoneyBag(b, this).simplify();
    }
...
public class Money implements IMoney {
  public IMoney addMoney(Money m) {
        if(m.getCurrency().equals(currency))
            return new Money(getAmount() + m.getAmount(), getCurrency());
        return new MoneyBag(this, m).simplify();
    }
...
```


Simplify

```
public IMoney simplify() {
    for(Iterator i = monies.iterator(); i.hasNext();){
        Money m = (Money)i.next();
        if (m.getAmount() == 0)
            i.remove();
    }

    if(monies.size() == 1)
        return (IMoney)monies.first();
    return this;
}
```


Test and Develop

- We retain all the tests that are written, so that as more code is added (or existing code refactored) everything is still being tested
- Can specify additional behaviour as a test, then write the code so the test passes
- Write tests for all methods (except maybe accessors, etc.) including equals, compareTo
- If you are tempted to write a print statement or a debugging expression.... write a test instead — it will just add to your library of tests
- Write tests that will give you information (eg unexpected failure... or success)
- Keep your tests running!

Summary

- JUnit is a clean and simple framework for unit testing Java code
- It is good to get into a testing habit as it makes it easier to refactor code
- JUnit is closely linked to the Extreme Programming (XP) approach
- How does JUnit work — look at the “Cook’s Tour” at www.junit.org/