References


Refactoring: Improving the Design of Existing Code, Fowler, Addison-Wesley, 1999
Unit Testing
Testing

Johnson's Law

If it is not tested it does not work

The more time between coding and testing

More effort is needed to write tests
More effort is needed to find bugs
Fewer bugs are found
Time is wasted working with buggy code
Development time increases
Quality decreases
Unit Testing

Tests individual code segments

Automated tests
What wrong with:

Using print statements

Writing driver program in main

Writing small sample programs to run code

Running program and testing it be using it
We have a QA Team, so why should I write tests?
When to Write Tests

First write the tests

Then write the code to be tested

Writing tests first saves time

Makes you clear of the interface & functionality of the code

Removes temptation to skip tests
What to Test

Everything that could possibly break

Test values
- Inside valid range
- Outside valid range
- On the boundary between valid/invalid

GUIs are very hard to test
- Keep GUI layer very thin
- Unit test program behind the GUI, not the GUI
Common Things Programs Handle Incorrectly

Adapted with permission from “A Short Catalog of Test Ideas” by Brian Marick,
http://www.testing.com/writings.html

Strings
Empty String

Collections
Empty Collection
Collection with one element
Collection with duplicate elements
Collections with maximum possible size

Numbers
Zero
The smallest number
Just below the smallest number
The largest number
Just above the largest number
XUnit

Free frameworks for Unit testing

SUnit originally written by Kent Beck 1994

JUnit written by Kent Beck & Erich Gamma

Available at: http://www.junit.org/

Ports to many languages at:
http://www.xprogramming.com/software.htm
import static org.junit.Assert.*;
import java.util.ArrayList;
import org.junit.Before;
import org.junit.Test;

public class HelloWorldTest {
    int testValue;

    @Test
    public void testMe() {
        assertEquals(1, testValue);
    }

    @Test
    public void foo() {
        assertTrue(2 == testValue);
    }

    @Before
    public void initialize(){
        testValue = 1;
    }
}

@Test(expected=IndexOutOfBoundsException.class)
public void testIndexOutOfBoundsException() {
    ArrayList emptyList = new ArrayList();
    Object notValid = emptyList.get(0);
}

@Before
public void initialize(){
    testValue = 1;
}
JUnit Example - JUnit 3.x

Goal: Implement a Stack containing integers.

Tests:
    Subclass junit.framework.TestCase
    Methods starting with 'test' are run by TestRunner

import junit.framework.*;
public class TestStack extends TestCase {

    public void testDefaultConstructor() {
        Stack test = new Stack();
        assertTrue("Default constructor", test.isEmpty() );
    }

    public void testSizeConstructor() {
        Stack test = new Stack(5);
        assertTrue( test.isEmpty() );
    }

}
public class Stack {
    int[] elements;
    int topElement = -1;

    public Stack() {
        this(10);
    }

    public Stack(int size) {
        elements = new int[size];
    }

    public boolean isEmpty() {
        return topElement == -1;
    }
}

Start of Stack Class
Running JUnit Using Eclipse

Creating Test Case
Running JUnit Using Eclipse

Fill in dialog window & create the test cases

Select Junit test case from the "Run as..." menu
Assert Methods

assertTrue()
assertFalse()
assertEquals()
assertNotEquals()
assertSame()
assertNotSame()
assertNull()
assertNotNull()
assertNotNull()
fail()

For a complete list see

Testing the Tests
If can be useful to modify the code to break the tests

```java
package example;

public class Stack {
    int[] elements;
    int topElement = -1;
    etc.

    public boolean isEmpty() {
        return topElement == -1;
    }
}
```
Before each test setUp() is run
After each test tearDown() is run

package example;
import junit.framework.TestCase;

public class StackTest extends TestCase {
    Stack test;

    public void setUp() {
        test = new Stack(5);
        for (int k = 1; k <= 5; k++)
            test.push(k);
    }

    public void testPushPop() {
        for (int k = 5; k >= 1; k--)
            assertEquals("Pop fail on element " + k, test.pop(), k);
    }
}

Test Fixtures - JUnit 3.x
Testing Exceptions - JUnit 3.x

public void testIndexOutOfBoundsException() {

    ArrayList list = new ArrayList(10);
    try {
        Object o = list.get(11);
        fail("Should raise an IndexOutOfBoundsException");
    } catch (IndexOutOfBoundsException success) {}
}

Example is from the JUnit FAQ
Refactoring

Changing the internal structure of software without changing its observable behavior

Done to make the software easier to understand and cheaper to modify
When to Refactor

Rule of three

Three strikes and you refactor
When to Refactor

When you add a new function
When you need to fix a bug
When you do a code review
When Refactoring is Hard

Databases

Changing published interfaces

Major design issues
public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }
}

Example directly from Refactoring: Improving the Design of Existing Code, Martin Fowler, chapter 1.
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + ":n;
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() -2) *1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) *1.5;
                break;
        }
        frequentRenterPoints ++;
        if (((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1)
            frequentRenterPoints ++;
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += You earned " + String.valueOf(frequentRenterPoints) + "frequent renter points";
    return result;
}
Comments?
When you add a feature to a program

If needed, refactor the program to make it easy to add the feature.

Then add the feature.
Before you start refactoring

Make sure that you have a solid suite of tests

Test should be self-checking
Do I need tests when I use my IDEs refactoring tools?

Are your IDE refactoring tools bug free?
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        }
        frequentRenterPoints ++;
        if (((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1))
            frequentRenterPoints ++;
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;
}
How Not to Refactor

public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    etc.
}

private void initializeStatement() {
    _totalAmount = 0;
    _frequentRenterPoints = 0;
    _rentals = _rentals.elements();
    _result = "Rental Record for " + getName() + "\n";
}

Bad idea – just copying parts of code to other methods and making the local variables fields so the methods can communicate.