CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 14 Patterns, Refactoring & Testing

Contents

Patterns and Coding .................................................................................................................... 2
Refactoring ................................................................................................................................. 3
Testing ........................................................................................................................................ 4
  Johnson’s Law .......................................................................................................................... 4
  Why Unit Testing .................................................................................................................... 5
  Testing First ............................................................................................................................... 7
JUnit ........................................................................................................................................ 9
  Using JUnit .............................................................................................................................. 10
What to Test ............................................................................................................................... 20
Back To Refactoring .................................................................................................................. 31
Code Smells ............................................................................................................................... 32
  Most Common Refactoring: Extract Method ................................................................. 33
    Motivation ............................................................................................................................. 33
    Mechanics ............................................................................................................................ 34
    Example ............................................................................................................................... 36

References

JUnit Cookbook Local copy at: http://www.eli.sdsu.edu/java-SDSU/junit/cookbook/cookbook.htm

JUnit Test Infected: Programmers Love Writing Tests Local copy at: http://www.eli.sdsu.edu/java-SDSU/junit/testinfected/testing.htm

JUnit on-line documentation Local copy at: http://www.eli.sdsu.edu/java-SDSU/docs/

Originals of the above can be found at: http://www.junit.org/

Refactoring: Improving the Design of Existing Code, Fowler, 1999,


  Used here with permission from Brian Marick


  Used here with permission from Brian Marick

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.
Patterns and Coding

Ralph Johnson recommends using patterns with you know you really need it

You have a program in development

You discover that using a pattern would improve the existing code

But adding the pattern requires modifying existing code!
Refactoring

Refactoring is the modifying existing code without adding functionality.

Changing existing code is dangerous:

- Changes can break existing code.

To avoid breaking code while refactoring:

- Need tests for the code.
  - Tests need to be automated.
- Proceed in small steps.
Testing

Johnson's Law

If it is not tested it does not work

Types of tests

• Unit Tests
  Tests individual code segments

• Functional Tests
  Test functionality of an application

• Regression Tests
  Determine if new code produces results as old code

• White box testing
  Testing with knowledge of the code

• Black box testing
  Testing with no knowledge of the code
Why Unit Testing

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

Without unit tests

- Code integration is a nightmare
- Changing code is a nightmare
Who Writes Unit Tests?

Programmers

Waiting for QA or testing wastes time

• Long delay for feedback on code
• Programmers forget code in this delay
• Programmers use untested code in new code
Testing First

First write the tests

Then write the code to be tested

Writing tests first:

- Removes temptation to skip tests
- Makes you define of the interface & functionality of the code before
Why Automated Tests

Why bother with automated test?

Just print results to standard out

Automated tests

• Allow you to run many tests at once
• Allow others to run the tests
• Allows you to keep the tests for later
JUnit

Framework for unit testing Java code

Available at: [http://www.junit.org/](http://www.junit.org/)

Already installed in JDK 1.2 & 1.3 on rohan and moria

Ports of JUnit are available in

- C++
- Forte 4GL
- PowerBuilder
- Smalltalk
- Delphi
- Objective-C
- Python
- Visual Basic
- Eiffel
- Perl
- Ruby
- .net

See [http://www.xprogramming.com/software.htm](http://www.xprogramming.com/software.htm) to download ports of JUnit
Using JUnit
Example

Goal: Implement a Stack containing integers.

Tests:

Subclass junit.framework.TestCase

Methods starting with 'test" are run by TestRunner

First tests for the constructors:

package example;

Import junit.framework.TestCase;

public class StackTest extends TestCase {

    //required constructor
    public StackTest(String name) {
        super(name);
    }

    public void testDefaultConstructor() {
        Stack test = new Stack();
        assert( test.isEmpty() );
    }

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

}
First part of the Stack

package example;

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;
    }
}

Running JUnit

JUnit has three interfaces

- Text  (junit.textui.*)
- AWT   (junit.ui.*)
- Swing (junit.swingui.*)

  Shows list of previously run test classes

JUnit has two class loaders

- Normal java class loader (TestRunner)
- junit.util.TestCaseClassLoader  (LoadingTestRunner)

  Reloads classes without having to restart program
Starting TestRunner

Make sure your classpath includes the code to tested

On Rohan use:

java junit.ui.LoadingTestRunner

You get a window like:
Enter the full name of the test class

Click on the Run button

If there are errors/failures select one and click on Show

You will see a stack trace of the error

With LoadingTestRunner you can recompile the Stack & StackTest classes without exiting LoadingTestRunner
Testing the Tests

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

package example;

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

    etc.

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

One company had an automatic build and test cycle that ran at night. The daily build was created and all the tests were run at night. The test results were available first thing in the morning. One night the build process crashed, so the daily build was not made. Hence there was no code to test. Still 70% of the tests passed. If they had tested their tests, they would have discovered immediately that their tests were broken.
Test Fixtures

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 StackTest(String name) {
        super(name);
    }

    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--)
            assert("Popping element " + k, test.pop() == k);
    }
}

Suites – Multiple Test Classes

Multiple test classes can be run at the same time

Running AllTests in TestRunner runs the test in

StackTest
QueueTest

package example;
import junit.framework.TestSuite;

public class AllTests
{
    static public TestSuite suite()
    {
        TestSuite suite= new TestSuite();
        try
        {
            suite.addTest(new TestSuite(StackTest.class));
            suite.addTest(new TestSuite(QueueTest.class));
        }
        catch (Exception e)
        {
        }
        return suite;
    }
}
Using Main

We can use main to run the test via textui.TestRunner

The command:

    java example.AllTests

will run all the tests in StackTest & QueueTest

```java
package example;

import junit.framework.TestSuite;
import junit.textui.TestRunner;

public class AllTests
{
    static public void main(String[] args)
    {
        TestRunner.main(args);
    }

    static public TestSuite suite()
    {
        same as last page
    }
}
```
What to Test

**Fowler on Testing**

"It is better to write and run incomplete tests than not to run complete tests"

"Don't let the fear that testing can't catch all bugs stop you from writing the tests that will catch most bugs"

"Trying to write too many tests usually leads to not writing enough"

"Run your tests frequently"

"When you get a bug report, start by writing a unit test that exposes the bug"

Think of the boundary conditions and concentrate your tests there

---

1 Fowler Chapter 4, pp. 89-102
Programming Errors

Programmers tend to make the same errors many times

Keep a list or catalog of your errors
A Short Catalog of Test Ideas

Tests develop catalogs of commonly found errors in programs.

Since errors are often repeated, this helps testers find common errors.

As programmers such a catalog:

- Suggests tests to uncover errors
- Help avoid errors when writing code

If we know these are common errors, we can keep them in mind while coding.

The following catalog is from Brian Marick:


The catalog is used here with permission.
Any Object

Test nil(null) references and pointers to objects

In Java/Smalltalk

Does the code handle correctly variables & parameters that are null(nil)

Java

String firstName = null

Smalltalk

| firstName |
firstName := nil.
Strings

Test the empty string

Does the code do the correct thing when string variables/parameters are the empty string

In Java/Smalltalk an empty string is not the same as a null(nil) reference to a string

Java

    String firstName = "";
    String secondName = new String();

Smalltalk

    | firstName secondName |
    firstName := ".
    secondName := String new
Numbers

Test the code using:

- 0
- The smallest number

Often numbers are used in a context with a valid range

The smallest number refers to the smallest valid number in the range

- Just below the smallest number
- The largest number
- Just above the largest number

Example

```java
int planetIndex; // Represents the I'th planet from the Sun
```

<table>
<thead>
<tr>
<th>Numbers to test</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Below the smallest</td>
</tr>
<tr>
<td>1</td>
<td>Smallest</td>
</tr>
<tr>
<td>9</td>
<td>Largest (Pluto is still considered a planet)</td>
</tr>
<tr>
<td>10</td>
<td>Above the largest</td>
</tr>
</tbody>
</table>
Collections

Test the code using:
• An empty collection
• A collection with one element
• The largest possible collection

Not the largest possible collection allowed by the language/hardware

The largest possible collection the system will encounter

If this is not possible use a collection with more than one element

• A collection with duplicate elements
Linked Structures (trees, graphs, etc.)

Test the code using:
- An empty structure
- Minimal non-empty structure
- A circular structure
- A structure with depth greater than one

The test must make the code reach the lowest depth

If the structure in the context has a maximally deep use that level
Equality Testing of Objects

Objects have two meanings of equality

• Pointer Identical
  Two object references point to the same memory location

• Equal
  The fields of the two objects have the same value

Java

• ==
  Tests if two object references are pointer identical

• equals()
  Tests if two objects are equal

  If this method is not implemented in a class it defaults to ==

Smalltalk

• ==
  Tests if two object references are pointer identical

• =
  Tests if two objects are equal

  If this method is not implemented in a class it defaults to ==
Test the code with objects equal but not identical

Lack of pointer identity should extend as far down as is meaningful to the code

Test the code with objects different at the lowest level
Don't Forget

"Trying to write too many tests usually leads to not writing enough ... You get many benefits from testing even if you do a little testing ..."

Fowler
Back To Refactoring

Refactoring requires:
• Testing

• Knowing when code could be improved

• Knowing how to improve the code

• Knowing the benefits of the improvement
**Code Smells**

If it stinks, change it

-- Grandma Beck on child-rearing

## Some Smells

<table>
<thead>
<tr>
<th>Code Smell</th>
<th>Code Smell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicate Code</td>
<td>Long Method</td>
</tr>
<tr>
<td>Large Class</td>
<td>Long Parameter List</td>
</tr>
<tr>
<td>Divergent Change</td>
<td>Shotgun Surgery</td>
</tr>
<tr>
<td>Feature Envy</td>
<td>Data Clumps</td>
</tr>
<tr>
<td>Primitive Obsession</td>
<td>Switch Statements</td>
</tr>
<tr>
<td>Parallel Inheritance Hierarchies</td>
<td>Lazy Class</td>
</tr>
<tr>
<td>Speculative Generality</td>
<td>Temporary Field</td>
</tr>
<tr>
<td>Message Chains</td>
<td>Middle Man</td>
</tr>
<tr>
<td>Inappropriate Intimacy</td>
<td>Alternative Classes with Different Interfaces</td>
</tr>
<tr>
<td>Incomplete Library Class</td>
<td>Data Class</td>
</tr>
<tr>
<td>Refused Bequest</td>
<td>Comments</td>
</tr>
</tbody>
</table>
Most Common Refactoring: Extract Method$^2$

You have a code fragment that can be grouped together.

*Turn the fragment into a method whose name explains the purpose of the method*

**Motivation**

Short methods:

- Increase possible reuse
- Makes high level methods easier to read
- Makes easier to override methods

$^2$ Refactoring Text, pp. 110-116
Mechanics

• Create a new method - the target method

  Name the target method after the intention of the method

  With short code only extract if the new method name is better than the code at revealing the code's intention

• Copy the extracted code from the source method into the target method

• Scan extracted code for references to local variables (temporary variables or parameters) of the source method

• If a temporary variable is used only in the extracted code declare it local in the target method

• If a parameter of the source method is used in the extracted code, pass the parameter to the target method
Mechanics - Continued

• See if the extracted code modifies any of the local variables of the source method

If only one variable is modified, then try to return the modified value

If more than one variable is modified, then the extracted code must be modified before it can be extracted

Split Temporary Variables or Replace Temp with Query may help

• Compile when you have dealt with all the local variables

• Replace the extracted code in source code with a call to the target method

• Compile and test
Example

No Local Variables

Note I will use Fowler’s convention of starting instance variables with "_" even though one can not do this is Squeak.

```java
void printOwing() {

    //print banner
    System.out.println("*************************");
    System.out.println("****Customer Owes*********");
    System.out.println("*************************");

    Iterator orders = _orders.iterator();
    double outstanding = 0.0;

    // Calculate outstanding
    while (orders.hasNext() ) {
        Order each = (Order) orders.next();
        outstanding = outstanding + each.getAmount();
    }

    //Print Details
    System.out.println("name: " + _name);
    System.out.println("amout: " + outstanding);
}
```

3 Example code is Squeak version of Fowler's Java example
Extracting the banner code we get:

```java
void printOwing() {

    printBanner();

    Iterator orders = _orders.iterator();
    double outstanding = 0.0;

    // Calculate outstanding
    while (orders.hasNext() ) {
        Order each = (Order) orders.next();
        outstanding = outstanding + each.getAmount();
    }

    //Print Details
    System.out.println("name: " + _name);
    System.out.println("amout: " + outstanding);
}

void printBanner() {
    System.out.println("*************************");
    System.out.println("****Customer Owes***********");
    System.out.println("*************************");
    System.out.println("" + outstanding);
}
```
Examples: Using Local Variables

We can extract printDetails() to get

```java
void printOwing() {

    printBanner();

    Iterator orders = _orders.iterator();
    double outstanding = 0.0;

    // Calculate outstanding
    while (orders.hasNext() ) {
        Order each = (Order) orders.next();
        outstanding = outstanding + each.getAmount();
    }

    printDetails(outstanding);
}

void printDetails( double amountOwed) {
    System.out.println("name: " + _name);
    System.out.println("amout: " + outstanding);
}
```
Then we can extract outstanding to get:

```java
void printOwing() {
    printBanner();
    double outstanding = outStanding();
    printDetails(outstanding);
}

double outStanding() {
    Iterator orders = _orders.iterator();
    double outstanding = 0.0;

    while (orders.hasNext() ) {
        Order each = (Order) orders.next();
        outstanding = outstanding + each.getAmount();
    }
    return outstanding;
}
```
Using Replace Parameter with Method\textsuperscript{4} we can change this to:

```java
void printOwing() {
    printBanner();
    printDetails();
}

void printDetails() {
    System.out.println("name: "+_name);
    System.out.println("amout: "+outstanding());
}
```

\textsuperscript{4} Fowler pp. 292-294
Reducing Coupling

The printing is still coupled to System.out

Using Add Parameter we get:

```java
void printOwing(PrintString out) {
    printBanner(out);
    printDetails(out);
}
```

where

```java
void printDetails(PrintString out) {
    out.println("name: " + _name);
    out.println("amout: " + outstanding());
}
```

```java
void printBanner(PrintString out) {
    out.println("*************************");
    out.println("****Customer Owes*********");
    out.println("*************************");
}
```

If you really do print to the screen a lot, you might add:

```java
void printOwing() {
    printOwing(System.out);
}
```