CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 3 Refactoring Intro
Jan 28, 2009

Copyright ©, All rights reserved. 2010 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.
JUnit Web site: http://www.junit.org/


Brian Marick's Testing Web Site: http://www.exampler.com/testing-com/

Testing for Programmers, Brian Marick, Available at: http://www.exampler.com/testing-com/writings.html

Refactoring: Improving the Design of Existing Code, Fowler, Addison-Wesley, 1999, chapters 1 & 3
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
XUnit Versions

3.x
Old version
Works with a versions of Java

4.x
Current version 4.8.1
Uses Annotations
Requires Java 5 or later
public class Adder {
    private int base;
    public Adder(int value) {
        base = value;
    }

    public int add(int amount) {
        return base + amount;
    }
}

Creating Test Case in Eclipse
Creating Test Case in Eclipse

Fill in dialog window & create the test cases
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.Test;

public class TestAdder {

    @Test
    public void testAdd() {
        Adder example = new Adder(3);
        assertEquals(4, example.add(1));
    }

    @Test
    public void testAddFail() {
        Adder example = new Adder(3);
        assertTrue(3 == example.add(1));
    }
}
Running the Tests
The result
Assert Methods

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

For a complete list see http://kentbeck.github.com/junit/javadoc/latest/
Annotations

After
AfterClass
Before
BeforeClass
Ignore
Rule
Test
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import org.junit.Before;
import org.junit.Test;

public class TestAdder {
    Adder example;
    @Before
    public void setupExample() {
        example = new Adder(3);
    }

    @Test
    public void testAdd() {
        assertEquals(4, example.add(1));
    }
}

Using Before
Refactoring
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
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?
Code Smells
Duplicate Code

Duplicate Code

Duplicate Code

Duplicate Code

Duplicate Code

Duplicate Code

Duplicate Code

Duplicate Code
The average method size should be less than 8 lines of code (LOC) for Smalltalk and 24 LOC for C++

The average number of methods per class should be less than 20

The average number of fields per class should be less than 6.

The class hierarchy nesting level should be less than 6

The average number of comment lines per method should be greater than 1
a.foo(12, 2, "cat", "<tr>", 19.6, x, y, classList, cutOffPoint)
Divergent Change

One class is changed in different ways for different reasons
ShotGun Surgery

When you have to make a kind of change you have to make a lot of little changes in different locations
Feature Envy

A method seems more interested in a class other than the one it is in.
Data Clumps

Same three or four data items together in lots of places
Primitive Obsession

Using primitive types instead of creating small classes
Switch Statements

How do you program without them?
Lazy Class

Class that is not doing enough to pay for itself
Data Class

Class with just fields and setter/getter methods

Data classes are like children.

They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility
Inappropriate Intimacy

Classes that spend too much time delving into other classes' private parts
location = rat.getRoom().getMaze().getLocation()
Negative Slope

if (foo) {
    if (bar) {
        if (cat = dog) {
            if (rat < 10) {
                ...
            }
        }
    }
}
Temporary Field

Field is only used in certain circumstances

Common case
  field is only used by an algorithm
  Don't want to pass around long parameter list
  Make parameter a field
Refused Bequest

Subclass does not want to support all the methods of parent class

Subclass should support the interface of the parent class