Review

Object-Oriented Programming is good as it promotes
  Code reuse
  More readable code
  More maintainable code
  Better designs
Basic OO Heuristics

Keep related data and behavior in one place

A class should capture one and only one key abstraction

Beware of classes that have many accessor methods defined in their public interface
OO History

Objects as a formal concept in programming - Simula 67

Smalltalk introduced the term object-oriented programming - 1970s

Became dominant programming methodology
   Early and mid 1990s
So Why is Software Still so Bad?
Title Case

First letter in each word in a sentence is capitalized

This Is In Title Case.

This is not in title case.

NOR IS THIS IN TITLE CASE
Where do you put it in Java

In what class would you put a method that converts a string to title case?
Utility Method (Utility Function)

A method in a class that only uses data passed in as parameters
Code Smell

Hint that something has gone wrong somewhere in your code

Lists of Code Smells

A Taxonomy for "Bad Code Smells"


Coding Horror: Code Smells

http://www.codinghorror.com/blog/2006/05/code-smells.html

Cunningham wiki c2

Code Smell - Utility Method

Helper functions are a sign that related data and operations are not together
Java & OO

In many situations we can not OO in Java

Can not keep data and operations together in many of Java's existing classes

Ruby, Objective-C & Smalltalk allow you to add to existing classes
Result

Can't practice OO in small cases

Develop poor habits

Lose benefits of OO but don't notice
Code Smell - Vague Identifier

meetsCriteria
flag

This generally happens when the One Responsibility Rule has been violated
One Responsibility Rule

"A class has a single responsibility: it does it all, does it well, and does it only"

Bertrand Meyer

Try to describe a class in 25 words or less, and not to use "and" or "or"

If can not do this you may have more than one class
Refactoring
Refactoring

Changing the internal structure of software that changes its observable behavior

Done to make the software easier to understand and easier 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?
Eclipse Refactoring
## Eclipse Refactoring Menu

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename...</td>
<td>⌘ R</td>
</tr>
<tr>
<td>Move...</td>
<td>⌘ V</td>
</tr>
</tbody>
</table>

### Android

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Method Signature...</td>
<td>⌘ C</td>
</tr>
<tr>
<td>Extract Method...</td>
<td>⌘ M</td>
</tr>
<tr>
<td>Extract Local Variable...</td>
<td>⌘ L</td>
</tr>
<tr>
<td>Extract Constant...</td>
<td></td>
</tr>
<tr>
<td>Inline...</td>
<td>⌘ I</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convert Anonymous Class to Nested...</td>
<td></td>
</tr>
<tr>
<td>Convert Member Type to Top Level...</td>
<td></td>
</tr>
<tr>
<td>Convert Local Variable to Field...</td>
<td></td>
</tr>
</tbody>
</table>

### Java

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Superclass...</td>
<td></td>
</tr>
<tr>
<td>Extract Interface...</td>
<td></td>
</tr>
<tr>
<td>Use Supertype Where Possible...</td>
<td></td>
</tr>
<tr>
<td>Push Down...</td>
<td></td>
</tr>
<tr>
<td>Pull Up...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extract Class...</td>
<td></td>
</tr>
<tr>
<td>Introduce Parameter Object...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduce Indirection...</td>
<td></td>
</tr>
<tr>
<td>Introduce Factory...</td>
<td></td>
</tr>
<tr>
<td>Introduce Parameter...</td>
<td></td>
</tr>
<tr>
<td>Encapsulate Field...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generalize Declared Type...</td>
<td></td>
</tr>
<tr>
<td>Infer Generic Type Arguments...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>Key</th>
</tr>
</thead>
<tbody>
<tr>
<td>Migrate JAR File...</td>
<td></td>
</tr>
<tr>
<td>Create Script...</td>
<td></td>
</tr>
<tr>
<td>Apply Script...</td>
<td></td>
</tr>
<tr>
<td>History...</td>
<td></td>
</tr>
</tbody>
</table>
Rename Class

public class Foo {
    public int foo() {
        return 10;
    }
}

public class Bar {
    public int bar() {
        Foo test = new Foo();
        return test.foo() + 99;
    }
}

public class NewFoo {
    public int foo() {
        return 10;
    }
}

public class Bar {
    public int bar() {
        NewFoo test = new NewFoo();
        return test.foo() + 99;
    }
}
public class Bar {
    public int helperMethod(Foo test) {
        return test.foo() + test.fooTwo();
    }

    public int callHelper() {
        Foo data = new Foo();
        return helperMethod(data);
    }
}

public class Foo {
    public int foo() { return 10; }
    public int fooTwo() { return 20; }
    public int sum() {
        return foo() + fooTwo();
    }
}
Eclipse Move
Extract Class

```java
package edu.sdsu.cs;

public class BinaryTree {
    private BinaryTree root;
    private String value;
    private BinaryTree left;
    private BinaryTree right;

    public boolean hasValue(String value) {
        return hasValue(root, value);
    }

    private boolean hasValue(BinaryTree tree, String value) {
        if (0 == tree.value.compareTo(value))
            return true;
        if (-1 == tree.value.compareTo(value))
            return hasValue(tree.right, value);
        if (0 == tree.value.compareTo(value))
            return hasValue(tree.left, value);
    }
}
```
Refactoring Tool Issue

People tend to only use the features they know
Refactoring Tool Issue

Is a tool hard to use because I am unfamiliar with it or is it just hard to use
# Refactoring by 41 Professional Programmers

<table>
<thead>
<tr>
<th>Refactoring</th>
<th>Number of Programmers used Refactoring</th>
<th>Total Times used</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntroduceFactory</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>PushDown</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>UseSupertype</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>EncapsulateField</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>Introduce Parameter</td>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>Convert Local to Field</td>
<td>5</td>
<td>37</td>
</tr>
<tr>
<td>Extract Interface</td>
<td>10</td>
<td>26</td>
</tr>
<tr>
<td>Inline</td>
<td>11</td>
<td>185</td>
</tr>
<tr>
<td>Modify Parameters</td>
<td>11</td>
<td>79</td>
</tr>
<tr>
<td>Pull up</td>
<td>11</td>
<td>37</td>
</tr>
<tr>
<td>Extract Method</td>
<td>20</td>
<td>344</td>
</tr>
<tr>
<td>Move</td>
<td>24</td>
<td>212</td>
</tr>
<tr>
<td>Rename</td>
<td>41</td>
<td>2396</td>
</tr>
</tbody>
</table>
Try In Eclipse

Rename
Move
Encapsulate Field
Extract Method
Extract Class
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
Simple Class to Test

public class Adder {
    private int base;
    public Adder(int value) {
        base = value;
    }

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

Thursday, January 24, 13
Creating Test Case in Eclipse

New
- Show In
- Copy
- Copy Qualified Name
- Paste
- Delete
- Import...
- Export...
- Refresh
- Open Project
- Team

- Java Project
- Android Project
- Project...
- Package
- Class
- Interface
- Enum
- Annotation
- Source Folder
- Java Working Set
- Folder
- File
- Untitled Text File
- JUnit Test Case
- Task
Creating Test Case in Eclipse

Fill in dialog window & create the test cases
Test Class

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()
assertEquals()
assertNotEquals()
assertSame()
assertNotSame()
assertNull()
assertNotNull()
fail()
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));
    }
}
Code Smells
## Classifying Fowler's Code Smells

<table>
<thead>
<tr>
<th>Category</th>
<th>Smells</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bloaters</td>
<td>Long method, Large Class, Primitive Obsession, Long Parameter List, Data Clumps</td>
</tr>
<tr>
<td>Object-Orientation Abusers</td>
<td>Switch Statements, Temporary Field, Refused Bequest, Alternative Classes with Different Interfaces</td>
</tr>
<tr>
<td>Change Preventers</td>
<td>Divergent Change, Shotgun Surgery, Parallel Inheritance Hierarchies</td>
</tr>
</tbody>
</table>
# Classifying Fowler's Code Smells

<table>
<thead>
<tr>
<th>Dispensables</th>
<th>Lazy class</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Data class</td>
</tr>
<tr>
<td></td>
<td>Duplicate Code</td>
</tr>
<tr>
<td></td>
<td>Dead Code,</td>
</tr>
<tr>
<td></td>
<td>Speculative Generality</td>
</tr>
<tr>
<td>Couplers</td>
<td>Feature Envy</td>
</tr>
<tr>
<td></td>
<td>Inappropriate Intimacy</td>
</tr>
<tr>
<td></td>
<td>Message Chains</td>
</tr>
<tr>
<td></td>
<td>Middle Man</td>
</tr>
</tbody>
</table>
Duplicate Code

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
Long Parameter List

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
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
Message Chains

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