Lecture 28: OO Design Heuristics (Part 2)

Kenneth M. Anderson
Object-Oriented Analysis and Design
CSCI 6448 - Spring Semester, 2002

Credit is where Credit is Due

- Some material for this lecture is taken from
  - Effective Java
    - by Joshua Bloch
    - ISBN: 0-201-31005-8
      - © Addison Wesley, 2001
  - Bitter Java
    - by Bruce A. Tate
    - ISBN: 1-930110-43-X
      - © Manning Publications, 2002
- While both address Java, I try to generalize…

Last Lecture: OO Heuristics

- Rules of Thumb
  - covered heuristics on
    - classes and objects
    - OO topologies
      - Beware God classes
    - Inheritance

This Lecture

- Cover OO Design Heuristics
  - Specific "best practices"
  - Antipatterns
    - how to avoid and/or solve them
- Quiz
  - A large class or method is considered a "bad smell" in OO code
    - Why? There is more than one correct answer, list as many as you can
  - In the refactoring tutorial, why did Extract Method always seem to lead to Move Method?

Effective Java by Bloch

- An example of a "design heuristics" book that advocates "best practices"
  - presents 57 best practices or rules of thumb and shows how they were (or were not!) applied in the design of the Java class libraries
    - I'll try to keep Java-specific information to a minimum, because many of the heuristics generalize to other languages

Examples from Bloch

- Consider static factory methods in place of constructors
- Favor immutability
- Favor composition over inheritance
- Design and document for inheritance or else prohibit it

Static Factory Methods

- Typically an object is obtained from a class constructor

```
public Boolean(boolean value) {
    _value = value;
}
```

- but an alternative is to use a static factory method

```
public static Boolean valueOf(boolean b) {
    return (b ? Boolean.TRUE : Boolean.FALSE);
}
```


Static Factory Methods, cont.

- Benefits
  - static factory methods have names
    - two constructors with same signatures not allowed
      - factory methods can simply use different names
  - static factory methods do not have to create a new object each time they are invoked
    - consider my implementation of the flyweight pattern, no matter how many times the factory method was invoked, only a maximum of 26 objects could be created
  - static factory methods can return an object of any subtype of their return type; constuctors cannot
    - this method allows frameworks to return objects whose classes are private; like Java's ListIterator

Static Factory Methods, cont.

- Disadvantages
  - If a class provides only static factory methods, and makes its constructor private, then the class cannot be subclassed
  - static factory methods are not readily distinguishable from other static methods
    - whereas constructors "stand out" since their name matches the name of the class

 © Kenneth M. Anderson, 2002

Immutability

- Consider creating immutable classes
  - An immutable class is simply a class whose instances cannot be modified
- Why?
  - Immutable objects are simple
    - they can be in only one state
  - Immutable objects are thread-safe
    - since they never change value, multiple threads can access them without synchronization!
  - Immutable objects can be shared freely
    - one object can't change the value of an immutable object unexpectedly, because the value cannot change at all!

 © Kenneth M. Anderson, 2002

Immutability, continued

- To make a class immutable
  - Do not provide mutator methods
    - e.g. setName(String name)
  - Ensure that methods cannot be overridden
    - make constructor private, add factory method
  - Make all fields final (e.g. const)
  - Make all fields private
    - you should do this anyway
  - Ensure exclusive access to any mutable components
    - Do not store a pointer to a mutable object provided from the outside world, and do not return a pointer to any of your mutable objects; perform defensive copies instead

 © Kenneth M. Anderson, 2002

Immutability Example

```
public final class Complex {
   private final float realPart;
   private final float imaginaryPart;
   public Complex(float re, float im) {
      this.realPart = re;
      this.imaginaryPart = im;
   }
   public Complex add(Complex c) {
      return new Complex(realPart + c.getReal(),
        imaginaryPart + c.getImaginary());
   }
   …
```

 © Kenneth M. Anderson, 2002

Immutability

- Disadvantage
  - Immutable objects require a separate object for each distinct value
- Turning lemons into lemonade
  - use a static factory method to manage a pool of objects for your immutable class
  - if two objects with the same value are requested, return the same object each time
    - this lets the class ensure that a==b is equivalent to a.equals(b)
      - » e.g. a logically equals b, only when a and b point to the same object)
      - » the latter is typically much faster at run-time

Favor Composition

- Inappropriate use of inheritance leads to fragile software
  - Why? Because inheritance breaks encapsulation; subclasses can see and access the internals of their parents (unless protected by “private”)
    - thus, a subclass may “break” due to a change in one of its parents, even when the source code of the subclass was not modified
      - mainly because a subclass may have depended on the implementation details of its parents

A bad example

- Extend a Set class to keep track of how many elements have been added to the set since it was created
- We start by creating a subclass of HashSet called IHashSet (for instrumented hash set)
  - see next slide

IHashSet

```
public class IHashSet extends HashSet {
    private int addCount = 0;
    // constructors are not shown
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll©;
    }
    public int getAddCount() {
        return addCount;
    }
}
```

addCount = ???

```
IHashSet s = new IHashSet()
s. addAll(
    Arrays.asList(
        new String[] {"Snap", "Crackle", "Pop"}));
```

- We would hope addCount = 3, but it doesn't, it equals 6! Why?
  - because HashSet's addAll makes use of the add routine to add elements to the set; since we provided a method body for add, our method was invoked via polymorphism and we incremented addCount twice for each element added
- We depended on our parent's implementation not to "self invoke" its other methods and we guessed wrong!
  - To correct this problem, while still using inheritance, involves a lot of bad design choices
    - if we just override add, and not addAll, our class will break when Sun changes the implementation of addAll to no longer call add!
    - if we change our method to invoke add directly, then we ignore the method for addAll in our parent (and we might make a mistake in our implementation)

Composition to the Rescue!

- Rather than subclass HashSet, lets wrap it, furthermore, since HashSet implements the Set interface, lets create a Wrapper that can handle any class that implements the Set interface
  - Our new class ISet will take a reference to an existing Set, forward all Set related calls to it, and add our instrumentation to the add and addAll operations
    - now since we are delegating to a composed object, rather than overriding inherited methods, polymorphism no longer applies; HashSet's addAll method is free to use HashSet's add method (or not) and our code still works
  - See code next slide

ISet

```
public class ISet extends Set {
    private final Set s; private int addCount = 0;
    public Iset(Set s) {
        this.s = s;
    }
    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() {
        return addCount;
    }
```


Favor Composition, cont.

- Now ISet can handle any class that implements the Set interface, not just HashSet

```
Set s1 = new ISet(new TreeSet(list));
Set s2 = new ISet(new HashSet(20));
```

- ISet is an instance of the Decorator pattern, it wraps other Set classes
- Disadvantages
  - Hard to use in callback frameworks
    - because wrapped objects do not know they are wrapped
  - Writing forwarding methods is tedious

Prohibit Inheritance

- How do you design and document a class for inheritance
  - if not done properly, problems can arise, as we saw in the last example
    - we did not know that HashSet.addAll invoked HashSet.add
- First, a class must document precisely the effects of overriding any method
  - including documenting its self-use of its own methods; which methods and in what order
- Now, wait a minute! Does this violate the dictum that good documentation should describe "what" a method does not "how" it does it? Yes! Why? Because inheritance breaks encapsulation!

Inheritance, continued

- Second, to provide hooks to subclasses, protected methods must be provided
  - choose with care if your class is used by external clients
    - you are committing to these methods (and your self-use documentation from the previous slide) forever!
- Third, constructors must not invoke overridable methods
  - since superclass constructors are invoked before subclass constructors, use of an overridable method may cause a subclass method to be invoked before the subclass constructor has been invoked!
- Bloch highlights some other issues, but they are Java-specific and so I will skip them

Inheritance, continued

- By now it should be apparent that designing a class for inheritance places substantial limitations on the class
  - hence this is not a decision to be taken lightly
- The best solution to this problem is to prohibit subclassing in classes that are not designed and documented to be safely subclassed
  - Easiest way to do this is to make the constructors private and provide only public static factory methods
- If this is not an option, consider designing your class to eliminate self-use

Bitter Java by Bruce Tate

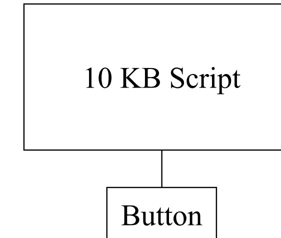
- Bitter Java's approach is to present antipatterns from "real-world" contexts and then document ways to fix and/or avoid them
  - So, what is an antipattern?
    - An antipattern is a "literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences"
  - Note, it's the **solution** to the problem that generates the negative consequences
    - the pattern is essentially saying "if you have this problem and then use this common solution, you will get into trouble in the following ways, so DO NOT USE THIS SOLUTION!"

Tate’s approach

- In Tate’s book, he describes an antipattern, and then describes a set of refactorings that can be used to go from the bad state of the common solution to another solution that has better consequences
  - I don’t have the space to document some of Tate’s more interesting examples; so I will show a very simple example instead
    - I highly recommend reading Bitter Java however; it has a lot of valuable information for avoiding software development problems!

The Magic Pushbutton

- The magic pushbutton looks like this



- a button in a user-interface invokes a huge complicated script to perform an operation

Background

- When Visual Basic came out (and toolkits like it), it encouraged developers to “go ahead and start building your house (application), you don’t need a foundation (software design)”
  - So, the user interface was designed first and code was chunked into event handlers that duplicated code and intertwined user interface code with application logic
- The problem, of course, is a lack of decomposition
  - This problem has resurfaced with the use of Servlets in Web Applications
    - I have heard of one case where an “electronic shopping cart” web application was written as a Servlet with a single method that was 27 pages of source code long!

The desired state?

- Model-View-Controller!
  - One of the earliest design patterns, and still ignored (often through ignorance!)
- We need to separate the user interface from the application objects via a controller
  - On the next slide is source code that does not follow this design pattern
    - It is an attempt to write a simple ATM application

The Magic Pushbutton, illustrated

```
public class BitterAccount {
  public String balance; // View
  public String fieldValue; // View
  public int radioButtons; // View
  public float balanceNumber; // Model
public BitterAccount() {
  balance = “0”;
}
// continued on next slide
```


BitterAccount, continued

```
public String buttonPressed() {
    radioButtons = userInterface.getRadioState(); // view
    if (radioButtons == 1) { // view
        fieldValue = userInterface.getEntryField(); // view
        balanceNumber = balanceNumber +
          Float.valueOf(fieldValue.trim()).floatValue(); // model
        balance = Float.toString(balanceNumber); // view
        return balance;
    } else if (radioButtons == 2) {
        fieldValue = userInterface.getEntryField();
        balanceNumber = balanceNumber -
          Float.valueOf(fieldValue.trim()).floatValue();
        balance = Float.toString(balanceNumber);
        return balance;
    } else if (radioButtons == 3) {
        return balance;
    }
}
}
```


How to start fixing this problem? Create a model!

- public class Account {
  - private float balance;
  - public Account(float openingBalance) {
    - setBalance(openingBalance);
  - }
  - public float debit(float amount) { …
  - public float credit(float amount) { …
  - public void setBalance(float amount) { …
  - public float getBalance(float amount) { …

Then, simplify the controller

```
public String buttonPressed() {
    radioButtons = userInterface.getRadioState();
    if (radioButtons == 1) {
        amount = Float.valueOf(
            userInterface.getEntryField()).floatValue();
        balanceNumber = account.credit(amount);
        balance = Float.toString(balanceNumber);
        return balance;
    } else { …
    …
```

- And now we can iterate and continue to separate model from view; the new account class will make it easier to add new features to our ATM