Programmieren II

OOP Principles II

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Based on material from Bob Tarr and T. Bögel)

June 18, 2014

Outline

Outline

Summary

OOP Principles from last session

- Minimize Accessibility of Classes and Members
- Favor Composition over Inheritance
- Program to an Interface, not an Implementation

Minimize Accessibility of Classes and Members

Abstraction

- Abstraction deals with complexity
- Subsuming similarities, neglecting special characteristics

Encapsulation

- Separating interface from implementation
- Hiding data, structure and implementation
- Interactions based on public interface

Achieving Encapsulation

Members (instance variables)

- **private** instance variables wherever possible
- Getters and setters if necessary

Methods

- Define interfaces
- Program to interfaces

Favor Composition over Inheritance

Inheritance

- Extending implementation of an existing object
- Superclass: common attributes and methods
- Subclass: extends implementation with additional attributes and methods

Disadvantages of Inheritance

- Breaks encapsulation
- If implementation of superclass changes: subclass might have to be changed
- White-box use of classes

Favor Composition over Inheritance

Inheritance

- Extending implementation of an existing object
- Superclass: common attributes and methods
- Subclass: extends implementation with additional attributes and methods

Disadvantages of Inheritance

- Breaks encapsulation
- If implementation of superclass changes: subclass might have to be changed
- White-box use of classes

Composition

Composition

- Object is composed of other objects
- Objects (components) are embedded into another objects
- Functionality is delegated to sub-components

Advantages

- Encapsulation!
- Each class is focused on one task
- Facilitates black-box reuse of components

Disadvantages

- More objects
- Increased complexity during interface design

Composition

Composition

- Object is composed of other objects
- Objects (components) are embedded into another objects
- Functionality is delegated to sub-components

Advantages

- Encapsulation!
- Each class is focused on one task
- Facilitates black-box reuse of components

Disadvantages

- More objects
- Increased complexity during interface design

When to use Inheritance: Coad's Rules

For Inheritance, all of the following criteria should be satisfied

- Subclass *is a special kind of a superclass* and not *is a role played by a superclass*
- Instance of subclass never needs to become an object of another class
- Subclass *extends* rather than overrides responsibilities of superclass
- Subclass does not extend a utility class
- The subclass specializes a *role, transaction, device* of a superclass

Program to an Interface, not an Implementation

- What we mean by “type” is often an interface, not a concrete class!
- Why program to an interface?
    - Clients do not have to know about specific classes
    - Interfaces specify all methods common to all implementing classes
    - Concrete implementations can be easily replaced by other objects

Outline

Source

The OOP Principles II section is from Bob Tarr

`http://userpages.umbc.edu/~tarr/`

OOP Principles

- 1. Minimize The Accessibility of Classes and Members
- 2. Favor Composition Over Inheritance
- 3. Program To An Interface, Not An Implementation
- 4. The Open-Closed Principle: Software Entities Should Be Open For Extension, Yet Closed For Modification
    - 4a. The Single Choice Principle: Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives
- 5. The Liskov Substitution Principle: Functions That Use References To Base (Super) Classes Must Be Able To Use Objects Of Derived (Sub) Classes Without Knowing It

Outline

Principle #4

The Open-Closed Principle:

Software Entities Should Be Open For Extension, Yet Closed For Modification

The Open-Closed Principle

- The Open-Closed Principle (OCP) says that we should attempt to design modules that never need to be changed
- To extend the behavior of the system, we add new code. We do not modify old code.
- Modules that conform to the OCP meet two criteria:
  - ⇨ Open For Extension - The behavior of the module can be extended to meet new requirements
  - ⇨ Closed For Modification - the source code of the module is not allowed to change
- How can we do this?
  - ⇨ Abstraction
  - ⇨ Polymorphism
  - ⇨ Inheritance
  - ⇨ Interfaces

The Open-Closed Principle

- It is not possible to have all the modules of a software system satisfy the OCP, but we should attempt to minimize the number of modules that do not satisfy it
- The Open-Closed Principle is really the heart of OO design
- Conformance to this principle yields the greatest level of reusability and maintainability

Open-Closed Principle Example

- Consider the following method of some class:

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    total += parts[i].getPrice();
  }
  return total;
}
```

- The job of the above function is to total the price of each part in the specified array of parts
- If Part is a base class or an interface and polymorphism is being used, then this class can easily accommodate new types of parts *without* having to be modified!
- It conforms to the OCP

Open-Closed Principle Example (Continued)

- But what if the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price.
- How about the following code?

```
public double totalPrice(Part[] parts) {
  double total = 0.0;
  for (int i=0; i<parts.length; i++) {
    if (parts[i] instanceof Motherboard)
      total += (1.45 * parts[i].getPrice());
    else if (parts[i] instanceof Memory)
      total += (1.27 * parts[i].getPrice());
    else
      total += parts[i].getPrice();
  }
  return total;
}
```

Open-Closed Principle Example (Continued)

- Does this conform to the OCP? No way!
- Every time the Accounting Department comes out with a new pricing policy, we have to modify the totalPrice() method! It is *not* Closed For Modification. Obviously, policy changes such as that mean that we have to modify code somewhere, so what could we do?
- To use our first version of totalPrice(), we could incorporate pricing policy in the getPrice() method of a Part

Open-Closed Principle Example (Continued)

- Here are example Part and ConcretePart classes:

```
// Class Part is the superclass for all parts.
public class Part {
  private double price;
  public Part(double price) (this.price = price;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return price;}
}

// Class ConcretePart implements a part for sale.
// Pricing policy explicit here!
public class ConcretePart extends Part {
  public double getPrice() {
    // return (1.45 * price);   //Premium
    return (0.90 * price);      //Labor Day Sale
  }
}
```

Open-Closed Principle Example (Continued)

- But now we must modify each subclass of Part whenever the pricing policy changes!
- A better idea is to have a PricePolicy class which can be used to provide different pricing policies:

```
// The Part class now has a contained PricePolicy object.
public class Part {
  private double price;
  private PricePolicy pricePolicy;

  public void setPricePolicy(PricePolicy pricePolicy) {
    this.pricePolicy = pricePolicy;}
  public void setPrice(double price) {this.price = price;}
  public double getPrice() {return pricePolicy.getPrice(price);}
}
```

Open-Closed Principle Example (Continued)

```
/**
 * Class PricePolicy implements a given price policy.
 */
public class PricePolicy {
  private double factor;

  public PricePolicy (double factor) {
    this.factor = factor;
  }

  public double getPrice(double price) {return price * factor;}

}
```

Open-Closed Principle Example (Continued)

- With this solution we can dynamically set pricing policies at run time by changing the PricePolicy object that an existing Part object refers to
- Of course, in an actual application, both the price of a Part and its associated PricePolicy could be contained in a database

The Single Choice Principle

A corollary to the OCP is the Single Choice Principle

The Single Choice Principle:

Whenever a software system must support a set of alternatives, ideally only one class in the system knows the entire set of alternatives

Principle #5

The Liskov Substitution Principle:

Functions That Use References To Base (Super) Classes Must Be

Able To Use Objects Of Derived

(Sub) Classes Without Knowing It

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) seems obvious given all we know about polymorphism
- For example:

```
public void drawShape(Shape s) {
        // Code here.
}
```

- The drawShape method should work with any subclass of the Shape superclass (or, if Shape is a Java interface, it should work with any class that implements the Shape interface)
- But we must be careful when we implement subclasses to insure that we do not unintentionally violate the LSP

The Liskov Substitution Principle

- If a function does not satisfy the LSP, then it probably makes explicit reference to some or all of the subclasses of its superclass. Such a function also violates the Open-Closed Principle, since it may have to be modified whenever a new subclass is created.

LSP Example

- Consider the following Rectangle class:

```
// A very nice Rectangle class.
public class Rectangle {
  private double width;
  private double height;

  public Rectangle(double w, double h) {
    width = w;
    height = h;
  }
  public double getWidth() {return width;}
  public double getHeight() {return height;}
  public void setWidth(double w) {width = w;}
  public void setHeight(double h) {height = h;}
  public double area() {return (width * height);
}
```

LSP Example (Continued)

- Now, had about a Square class? Clearly, a square is a rectangle, so the Square class should be derived from the Rectangle class, right? Let's see!
- Observations:
  - ⇨ A square does not need both a width and a height as attributes, but it will inherit them from Rectangle anyway. So, each Square object wastes a little memory, but this is not a major concern.
  - ⇨ The inherited setWidth() and setHeight() methods are not really appropriate for a Square, since the width and height of a square are identical. So we'll need to override setWidth() and setHeight(). Having to override these simple methods is a clue that this might not be an appropriate use of inheritance!

LSP Example (Continued)

- Here's the Square class:

```
// A Square class.
public class Square extends Rectangle {

  public Square(double s) {super(s, s);}

  public void setWidth(double w) {
    super.setWidth(w);
    super.setHeight(w);
  }

  public void setHeight(double h) {
    super.setHeight(h);
    super.setWidth(h);
  }
}
```

LSP Example (Continued)

- Everything looks good. But check this out!

```
public class TestRectangle {

  // Define a method that takes a Rectangle reference.
  public static void testLSP(Rectangle r) {
    r.setWidth(4.0);
    r.setHeight(5.0);
    System.out.println("Width is 4.0 and Height is 5.0" +
                       ", so Area is " + r.area());
    if (r.area() == 20.0)
      System.out.println("Looking good!\n");
    else
      System.out.println("Huh?? What kind of rectangle is
                                this??\n");
  }
```

LSP Example (Continued)

```
 public static void main(String args[]) {

  //Create a Rectangle and a Square
  Rectangle r = new Rectangle(1.0, 1.0);
  Square s = new Square(1.0);

  // Now call the method above.  According to the
  // LSP, it should work for either Rectangles or
  // Squares.  Does it??
  testLSP(r);
  testLSP(s);
}

}
```

LSP Example (Continued)

- Test program output:

```
Width is 4.0 and Height is 5.0, so Area is 20.0
Looking good!

Width is 4.0 and Height is 5.0, so Area is 25.0
Huh?? What kind of rectangle is this??
```

- Looks like we violated the LSP!

LSP Example (Continued)

- What's the problem here? The programmer of the testLSP() method made the reasonable assumption that changing the width of a Rectangle leaves its height unchanged.
- Passing a Square object to such a method results in problems, exposing a violation of the LSP
- The Square and Rectangle classes look self consistent and valid. Yet a programmer, making reasonable assumptions about the base class, can write a method that causes the design model to break down
- Solutions can not be viewed in isolation, they must also be viewed in terms of reasonable assumptions that might be made by users of the design

LSP Example (Continued)

- A mathematical square might be a rectangle, but a Square object is not a Rectangle object, because the behavior of a Square object is not consistent with the behavior of a Rectangle object!
- Behaviorally, a Square is *not* a Rectangle! A Square object is not polymorphic with a Rectangle object.

The Liskov Substitution Principle

- The Liskov Substitution Principle (LSP) makes it clear that the ISA relationship is all about behavior
- In order for the LSP to hold (and with it the Open-Closed Principle) all subclasses must conform to the behavior that clients expect of the base classes they use
- A subtype must have no more constraints than its base type, since the subtype must be usable anywhere the base type is usable
- If the subtype has more constraints than the base type, there would be uses that would be valid for the base type, but that would violate one of the extra constraints of the subtype and thus violate the LSP!
- The guarantee of the LSP is that a subclass can always be used wherever its base class is used!

Literature

Martin, Robert C.
The Open-Closed Principle
C++ Report
http://www.objectmentor.com/resources/articles/ocp.pdf

Martin, Robert C.
The Liskov Substitution Principle
C++ Report
http://www.objectmentor.com/resources/articles/lsp.pdf

Gamma, E. et al. (Gang of Four)
Design Patterns. Elements of Reusable Object-Oriented Software (Ch. 1).
Addison-Wesley, 1994.

Enum

Enum type

- Special data type
- Enables for a variable to be a set of predefined constants
- Variable must be equal to one of the pre-defined values
- Names of enum's fields are in uppercase letters

Common examples

- compass directions: `NORTH,SOUTH,EAST,WEST`
- days of the week: `MONDAY`, . . .
- . . .

Defining an enum type

Enum for the days of the week

```
public enum Day {
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
        THURSDAY, FRIDAY, SATURDAY;
    }
```

When to use enums?

- Whenever you know a fixed set of constants: use enums
- E.g. natural enum types (planets in our solar system)
- Choices of a menu, command line flags etc.
- Much more efficient than strings

Using an enum type I

```
public class EnumTest {
    Day day;

    public EnumTest(Day day) {
        this.day = day;
    }

    public void tellItLikeItIs() {
        switch (day) {
            case MONDAY:
                System.out.println("Mondays are bad.");
                break;
            case FRIDAY:
                System.out.println("Fridays are better.");
                break;
            case SATURDAY: case SUNDAY:
                System.out.println("Weekends are best.");
                break;
            default:
```

Using an enum type II

```
                System.out.println("Midweek days are so-so.");
                break;
            }
    }

    public static void main(String[] args) {
        EnumTest firstDay = new EnumTest(Day.MONDAY);
        firstDay.tellItLikeItIs();
        EnumTest thirdDay = new EnumTest(Day.WEDNESDAY);
        thirdDay.tellItLikeItIs();
        EnumTest fifthDay = new EnumTest(Day.FRIDAY);
        fifthDay.tellItLikeItIs();
        EnumTest seventhDay = new EnumTest(Day.SUNDAY);
        seventhDay.tellItLikeItIs();
    }
}
```

Enums as classes

Enums as classes

- Enum declaration defines a class
- Class body may contain methods and fields
- All enums have some common methods by default:
  - `values()` returns an array containing all of the values of the enum in declaration order

Example: enum with methods and properties

Modeling planets

- Each planet has a *mass*
- Each planet has a *radius*
- Properties are set via the constructor
- Constructor cannot be called explicitly
- Constants need to be defined first, prior to methods and instance variables

```java
public enum Planet {
/* constant declarations with constructor parameters */
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),

/* declaration of instance variables */
    private final double mass;   // in kilograms
    private final double radius; // in meters

    /* constructor */
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    /* other methods */
    private double getMass() {return mass;}
    private double getRadius() {return radius;}
```

```
// universal gravitational constant  (m3 kg-1 s-2)
public static final double G = 6.67300E-11;

double surfaceGravity() {
    return G * mass / (radius * radius);
}
```

Using the Planet enum

```
...
    public static void main(String[] args) {
        for (Planet p : Planet.values())
            System.out.printf("The surface gravity on %s is %f%n
                ",
                                  p, p.surfaceGravity());
    }

...
```

References

The Java Tutorials
Enum Types
http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Ullenboom, Ch.
Java ist auch eine Insel. (Ch. 9.4)
Galileo Computing, 2012.