Design Principles

Outline

• Minimize Access of Member
• Encapsulate what varies
• Favor composition over inheritance
• Program to interface
• Liskov Substitution Principle
• Open-closed Principle
• Program to interface

Abstraction

• “Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.” Tony Hoare

Encapsulation

• “Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.” Grady Booch

Minimize the Accessibility of Classes and Members

• Classes should be opaque
• Classes should not expose their internal implementation details
• Use getters and setters
  – Compare
    public double speed;
  – vs

Advantages of minimizing accessibility

• Centralized checking of constraints
• Add useful side-effects (monitoring)
## Favor Composition over Inheritance

### Composition
- Method of reuse in which new functionality is obtained by creating an object composed of other objects
  - New functionality obtained by delegating to objects being composed
- Sometimes called aggregation or containment
  - Aggregation - when one object owns or is responsible for another object and both objects have identical lifetimes (GoF)
  - Aggregation - when one object has a collection of objects that can exist on their own (UML)

### Advantages of Composition
- Contained objects are accessed by the containing class solely through their interfaces
- "Black-box" reuse, since internal details of contained objects are not visible
- Good encapsulation

### Disadvantages of Composition
- Resulting systems tend to have more objects
- Interfaces must be carefully defined in order to use many different objects as composition blocks

## Inheritance

### Advantages of Inheritance
- New implementation is easy, since most of it is inherited
- Easy to modify or extend the implementation being reused

### Inheritance
- Method of reuse in which new functionality is obtained by extending the implementation of an existing object
- The generalization class (the superclass) explicitly captures the common attributes and methods
- The specialization class (the subclass) extends the implementation with additional attributes and methods
Disadvantages of Inheritance

• Breaks encapsulation, since it exposes a subclass to implementation details of its superclass

• Implementations inherited from superclasses can not be changed at runtime

Example (from Effective Java by Bloch)
Variant of HashSet that tracks number of insertions

```java
public class InstrumentedHashSet extends HashSet {
    private int addCount = 0; // The number of attempted element insertions
    public InstrumentedHashSet() {
        super();
    }
    public InstrumentedHashSet(int initCap, float loadFactor) {
        super(initCap, loadFactor);
    }
    public boolean add(Object o) {
        addCount++; return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size(); return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```

Example: Test it!

```java
public static void main(String[] args) {
    InstrumentedHashSet s = new InstrumentedHashSet();
    s.addAll(Arrays.asList(new String[]{"Snap","Crackle","Pop");
    System.out.println(s.getAddCount());
}
```

Example: Composition

• Let’s write an InstrumentedSet class that is composed of a Set object.
• Our InstrumentedSet class will duplicate the Set interface, but all Set operations will actually be forwarded to the contained Set object.
• The contained Set object can be an object of any class that implements the Set interface (and not just a HashSet)

```java
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;
    public InstrumentedSet(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;}
    public void clear() { s.clear(); }
    public boolean contains(Object o) { return s.contains(o); }
    public boolean isEmpty() { return s.isEmpty(); }
    public int size() { return s.size(); }
    public Iterator iterator() { return s.iterator(); }
    public boolean remove(Object o) { return s.remove(o); }
    public boolean containsAll(Collection c) { return s.containsAll(c); }
    public boolean removeAll(Collection c) { return s.removeAll(c); }
    public boolean retainAll(Collection c) { return s.retainAll(c); }
    public Object[] toArray() { return s.toArray(); }
    public Object[] toArray(Object[] a) { return s.toArray(a); }
    public boolean equals(Object o) { return s.equals(o); }
    public int hashCode() { return s.hashCode(); }
    public String toString() { return s.toString(); }
}
```

Favor Composition over Inheritance

```java
public class InstrumentedSet implements Set {
    private final Set s;
    private int addCount = 0;
    public InstrumentedSet(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;}
}
```

Forwarding methods

```java
public void clear() { s.clear(); }
public boolean contains(Object o) { return s.contains(o); }
public boolean isEmpty() { return s.isEmpty(); }
public int size() { return s.size(); }
public Iterator iterator() { return s.iterator(); }
public boolean remove(Object o) { return s.remove(o); }
public boolean containsAll(Collection c) { return s.containsAll(c); }
public boolean removeAll(Collection c) { return s.removeAll(c); }
public boolean retainAll(Collection c) { return s.retainAll(c); }
public Object[] toArray() { return s.toArray(); }
public Object[] toArray(Object[] a) { return s.toArray(a); }
public boolean equals(Object o) { return s.equals(o); }
public int hashCode() { return s.hashCode(); }
public String toString() { return s.toString(); }
```
Coad's Rules: Use inheritance only when all of the following criteria are satisfied:

- A subclass expresses "is a special kind of" and not "is a role played by a"
- An instance of a subclass never needs to become an object of another class

Program to an Interface, not an Implementation

Advantages:

- Clients are unaware of the specific class of the object they are using

Open/Closed Principle

- Software should be Open for Extension, but Closed for Modification

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.
Open/Closed Principle

• Consider the following method that totals the price of each part in the specified array of parts of some class:

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

• 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

• Suppose the Accounting Department decrees that motherboard parts and memory parts should have a premium applied when figuring the total price:

```java
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;
}
```

Liskov Substitution Principle

• Functions that use References to Super Classes must be able to use Objects of sub-class types

• Barbara Liskov, Turing Award 2008