Design Patterns 2

Michael Li
email: jwl@cs.nott.ac.uk
http://www.cs.nott.ac.uk/~jwl/G52APR

Interface

• An Interface is a collection of abstract methods that an object implements
• Usage of interfaces:

```java
interface AnInterface {
    void MethodOne(String a);
}
class Aclass implements AnInterface {
    void MethodOne(String a) {
        System.println("The string is:" + a);
    }
}
```
• Interfaces can be extended:

```java
interface InterfaceA {
    void MethodOne(String a);
}
interface InterfaceB extends InterfaceA {
    void MethodTwo(String b);
}
```

Why use Interface?

• In solving complex problems.
  – Assume that you have a lot of screwdrivers…
  – Slotted, Phillips, square, hex, jeweller's…
  – Use a big box or many small boxes?

• Interfaces provides another way of programming.
• A class can implement multiple interfaces.
• A class has all the types of the interfaces that it implements.

Adapter

• The Adapter pattern lets you use an existing class to meet new requirements.
• Class adapter: create new class that implements new interface and subclasses an existing class.
• Object adapter: create a new client subclass that uses an instance of the existing class.

Design Principles

• “Identify the aspects of your application that vary and separate them from what stays the same”
• “Program to an interface, not an implementation”
**Strategy Pattern**

- Strategy Pattern defines a family of algorithms (methods), encapsulates each one, and makes them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.
- Loosely coupled
- When you modify an algorithm, the classes that use it will not be affected.

```java
interface TaxCalculator { float CalcTax(float sum); }
public class DollarTax implements Tax {
    public float CalcTax(float sum) { return sum*1.6073; }
}
public class EuroTax implements Tax {
    public float CalcTax(float sum) { return sum*1.1605; }
}
public class CadTax implements Tax {
    public float CalcTax(float sum) { return sum*1.6334; }
}
public class Main {
    TaxCalculator tc;
    float rate_of_tax = 0.20;
    float sales1 = 100.0;
    tc = new DollarTax();
    float tax1 = tc.CalcTax(sales * rate_of_tax);
    float sales2 = 150.0;
    tc = new EuroTax();
    float tax2 = tc.CalcTax(sales * rate_of_tax);
}
```

**Strategy**

- Our class doesn't implement the algorithm.
- Algorithms implemented in their own class.
- Our class is then composed with an object that implements the algorithm.
- Algorithm can be changed easily.

**Applicability**

Use the Strategy pattern whenever:
- Many related classes differ only in their behavior.
- You need different variants of an algorithm.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

```java
interface Foldable { void SetFold(); }
interface HasMoter { void SetMoter(); }
public class FoldableBehaviour implements Foldable {
    public FoldableBehaviour (return this);
    public void SetFold() { // Do something here}
}
public class NonFoldableBehaviour implements Foldable {
    public NonFoldableBehaviour NonFoldableBehaviour() {return this;
    public void SetFold() { // Do nothing}
}
public class Bicycle {
    protected Foldable fd;
    protected HasMoter hm;
}
public class myBike extends Bicycle {
    public void myBike() {
        fd = new FoldableBehaviour();
        fd.SetFold();
    }
}
```

**Change in Strategy**

- A new subclass of Bicycle: Bicycle_without_pedals
  - Non-foldable
  - Without motor
  - Without pedals
- Can easily implement
- Need a new behaviour
  - WithoutPedalsBehaviour
// create a WithoutPedals interface
// create a set of WithoutPedalsBehaviour classes
// declare a WithoutPedals type variable wp

public class Bicycle_without_pedals extends Bicycle
{
    public void Bicycle_without_pedals()
    {
        fd = new NonfoldableBehaviour();
        hm = new NoMotorBehaviour();
        wp = new WithoutPedalsBehaviour();
    }
    public void display() { ... }
};

Design principle

• “Favour composition over inheritance”

Composition

• Inheritance is fixed at compile time
• Composed objects can be changed
• We could change the implementation of the ‘foldable’ behaviour of a bicycle subclass
• Just by changing the reference to the FoldableBehaviour object

Decorator Pattern

• Used to deal with the situation when you’d end up with a large number of similar classes
• Think about if we were to model the software system in a cafe
• Each beverage would have a different class

Drinks

• Coffees
  – House Blend
  – Dark Roast
  – Decaf
  – Espresso

• Toppings
  – Mocha
  – Steamed Milk
  – Soy
  – Whip
public class Beverage
{
    protected String m_description;

    public string GetDescription()
    {
        return m_description;
    }

    public abstract double cost();
};

public class HouseBlend extends Beverage
{
    public HouseBlend()
    {
        m_description = "House Blend";
    }

    public double cost()
    {
        return 0.89;
    }
}

---

**Toppings**

- What about the Toppings?
- Inherit the various types?
  - EspressoWithMocha
  - DarkRoastWithSteamedMilk
  - ...
- Implementation as before

---

**Class Explosion**

- Where did all these classes come from?
  - 16 combination of toppings
  - 4 drinks
  - 64 different classes to implement
- Maintenance nightmare!
  - What if the cost of Mocha topping goes up?
- Expanding difficulty.
  - A new coffee means 16 new classes
  - A new topping means ≥64 new classes

---

**Favour Composition**

- Inheritance 64 — Composition 0
- Nothing is encapsulated
  - DarkRoastWithMocha
  - HouseBlendWithMocha
  - HouseBlendWithSteamedMilk
- No Code Reuse…
Inheritance

- Inheritance is powerful
- Doesn’t always lead to flexible designs
- Can ‘inherit’ behaviour at runtime via composition

Inherit or Compose

- Behaviour is fixed statically at compile time
- Composition can extend at runtime
- Composition allows us to add new responsibilities to objects without touching the superclass
- New functionality by writing new code, not editing old (and working) code

Decorator Pattern

- Attach additional responsibilities or functions to an object dynamically or statically. Also known as Wrapper.
- Can use the Decorator Pattern to solve this design
- Uses composition rather than inheritance

Decorator

- Start with a DarkRoast object
- Customer wants Mocha, wrap a Mocha object around DarkRoast
- Also want Whip, wrap a Whip object around Mocha
- Both decorators and concrete classes share same type, Drink

Coffee Decorators

- How it works
  - Call cost() on the decorated object
  - Decorator calls cost() on the object it decorates and adjusts the price
- Decorators can decorate Decorators
public class Beverage{
    protected String m_description;
    public string GetDescription(){
        return m_description;
    }
    public abstract double cost();
};

public class HouseBlend extends Beverage{
    public HouseBlend()
    {
        m_description = "House Blend";
    }
    public double cost(){
        return 0.89;
    }
};

public class CondimentDecorator extends Beverage{
    private Beverage m_beverage;
    public abstract String GetDescription();
    public abstract double Cost();
};

public class Mocha extends CondimentDecorator{
    public Mocha(Beverage beverage)
    {
        m_beverage = beverage;
    }
    public String GetDescription()
    {
        return m_beverage.GetDescription() + "Mocha";
    }
    public double Cost()
    {
        return m_beverage.Cost() + 0.20;
    }
};

Decorator Pattern

- The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality
- Black-box reuse
- Classes implementation unchanged
Design Principle

• “Classes should be open for extension, but closed for modification”

Open-Closed Principle

• *Classes should be open for extension*
• Feel free to extend our classes with any new behaviour you like, but…
• *closed for modification*
• Sorry, but our code is fixed and bug free you can’t change it

Observer Pattern

• Used when an object updates several other objects about changes in its state
• Push model — the *subject* pushes updates to the *observers*
• *Subject* keeps a list of *observers*
• List can change over time
• *Subject* provides methods to add/remove *Observers*

Observer Pattern

• *Observer defined*
  — *The Observer Pattern* defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically

Observer defined

• One-to-Many
  — One object holds the data ——> the Subject
  — Many recipients ——> the Observers

• Observers dependent on the Subject to inform them of changes in the data
Observer pattern

• Loose Coupled objects interact, but have little knowledge of each other
• Observer Pattern is an example of a loosely coupled OO design

Loose Observation

• Subject only knows that Observers implement an interface
• Can add new observers at any point
• New observer types don’t need modification of the Subject
• Changes to subject or observer don’t affect the other

Design Principle

• “Strive for loosely coupled designs between objects that interact.”

Design Patterns

• Patterns of commonly occurring object relationships
• Not code libraries
• Not a solution for every problem
• Vocabulary

<table>
<thead>
<tr>
<th>Behavioural</th>
<th>Structural</th>
<th>Creational</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strategy</td>
<td>Adapter</td>
<td>Factory Method</td>
</tr>
<tr>
<td>Observer</td>
<td>Decorator</td>
<td>Abstract Factory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Builder</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Singleton</td>
</tr>
</tbody>
</table>