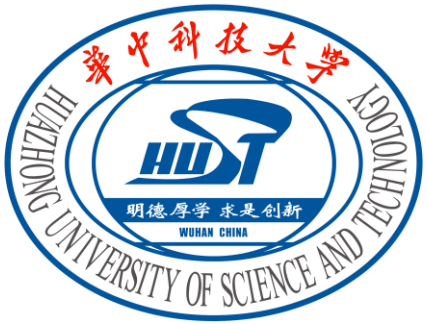
基于Java的面向对象程序设计

陈维亚

weiya_chen@hust.edu.cn

华中科技大学软件学院

第31-32讲：设计模式



目录

1. OOD 概述

❑ 面向对象设计 Object Oriented Design

OOD是
- **遵守**面向对象的概念和原则
- **运用**面向对象的思想和方法
- **结合**所用编程语言的特点

进行系统设计。

OOD的五个部分：

- 问题域部分的设计
- 人机交互部分的设计
- 控制流管理部分的设计
- 数据管理部分的设计
- 构件部署设计

1. OOD 概述

❑ Object Oriented Design (OOD)

“How I explained OOD to my wife”

http://www.codeproject.com/Articles/93369/How-I-explained-OOD-to-my-wife

走在结冰的河边不会湿鞋，开发需求不变的项目畅通无阻

(Walking on water and developing software from a specification are easy if both are frozen)——Edward V. Berard

1. OOD 概述

❑ 敏捷开发 Agile Software Development

- 一个设计敏捷的软件能以最小的代价**满足变化**，不用改变现有代码**满足扩展**，并且能被**复用**。
- 应用好OOD是做到敏捷设计的关键。

❑ SOLID 原则

- **S** 单一职责原则 Single Responsibility Principle
- **O** 开放闭合原则 Open Closed Principle
- **L** Liskov替换原则 Liskov Substitution Principle
- **I** 接口隔离原则 Interface Segregation Principle
- **D** 依赖倒置原则 Dependency Inversion Principle

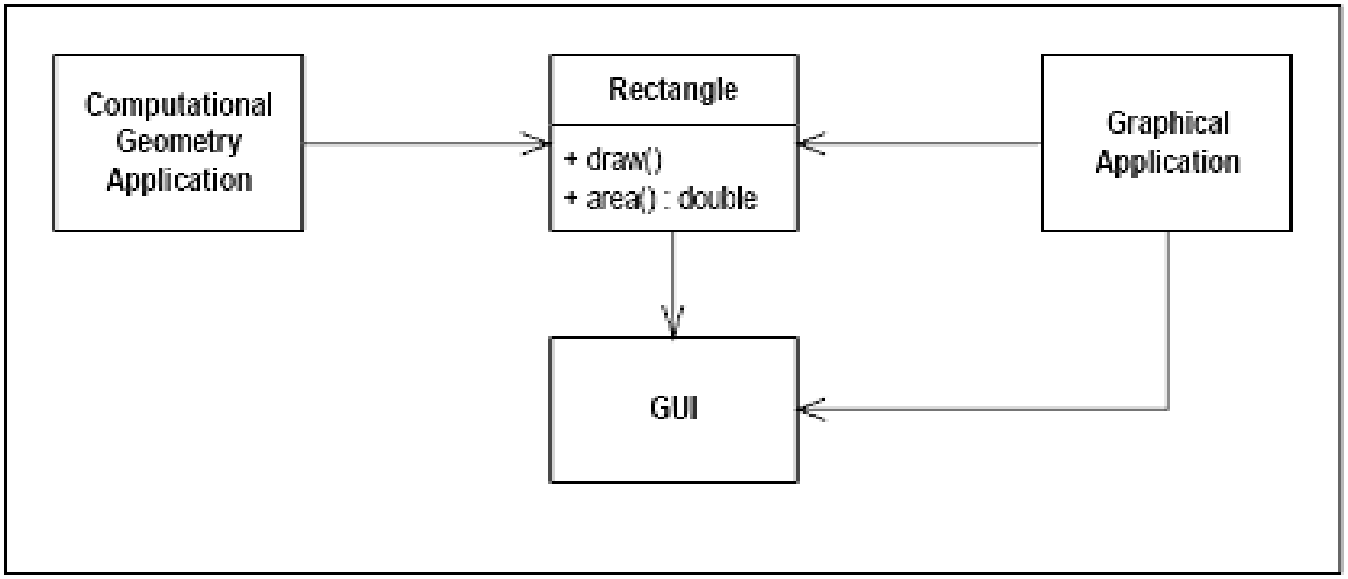
Robert Cecil Martin (Uncle Bob)

❑ SOLID - 单一职责原则

引起类变化的因素永远不要多于一个

一个类有且仅有一个职责，否则代码会变得耦合

❑ SOLID - 单一职责原则



Rectangle的职责：

- 计算矩形面积；
- 在界面上绘制矩形；

切分：

- Rectangle：这个类应该定义Area()方法；
- RectangleUI：这个类应继承Rectangle类，并定义Draw()方法。

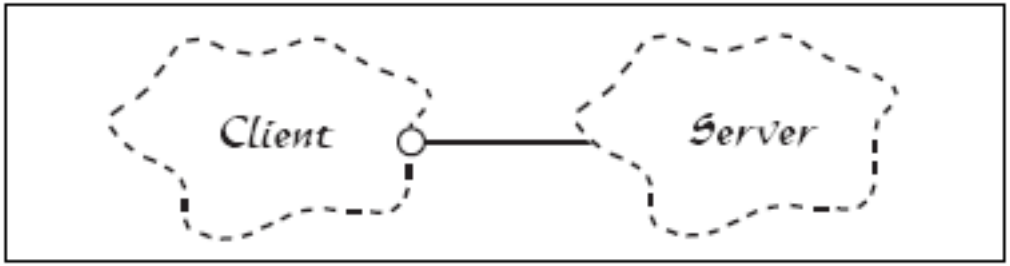
❑ SOLID - 开放闭合原则

软件实体(类,模块,函数等等)应当对扩展开放，对修改闭合。

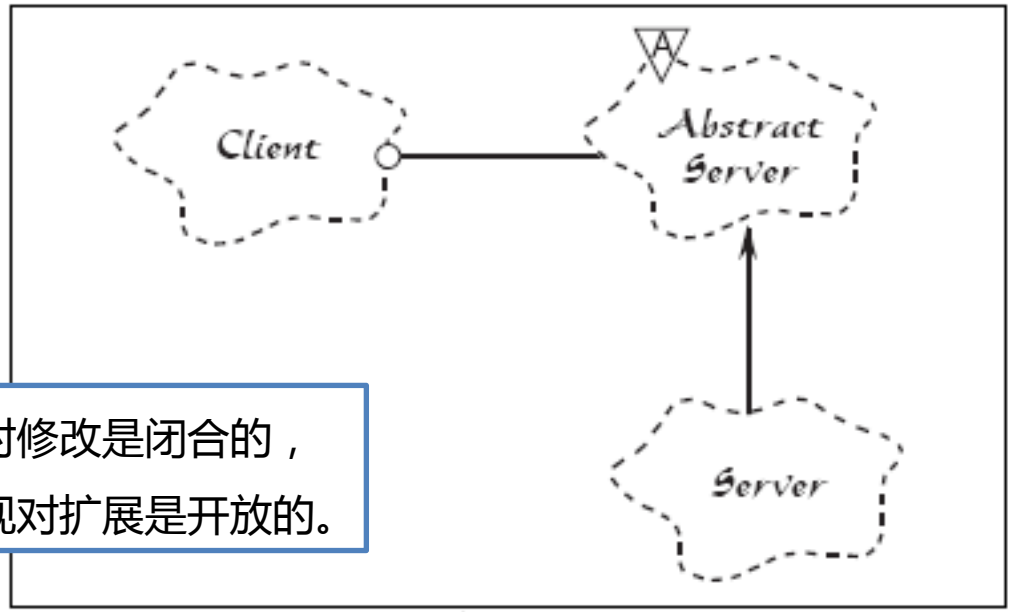
它意味着你应当能在不修改类的前提下扩展一个类的行为。

1988 “Object Oriented Software Construction” by Bertrand Meyer

❑ SOLID - 开放闭合原则



不支持"开放闭合"原则

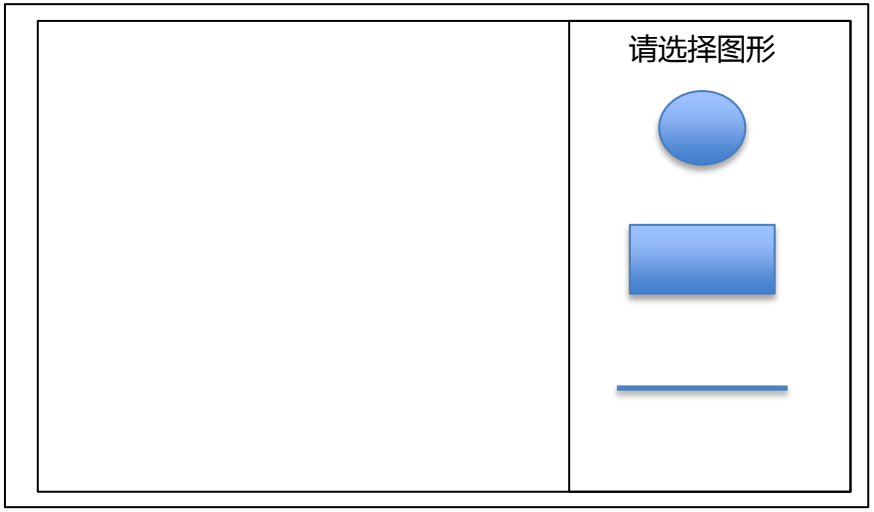


抽象服务类对修改是闭合的，
实体类的实现对扩展是开放的。

❑ SOLID - 开放闭合原则

实现方法：合理运用抽象类或接口

Shape类和Canvas类



Fruit和购物车类

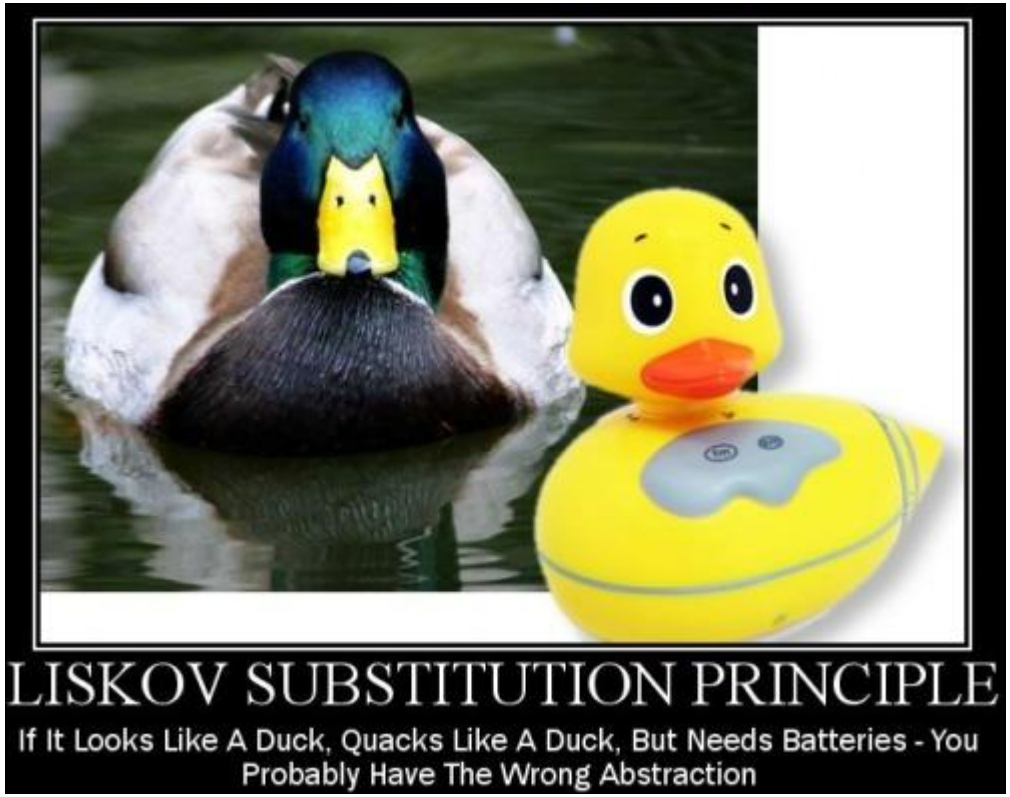
```
public class ShopCart {

    List<Fruit> items = new ArrayList<Fruit>();
    public void addFruit(Fruit f){
        items.add(f);
    }

    public float calculateTotalPrice(){
        float total = 0.0f;
        for(Fruit f : items){
            total += f.getPrice();
        }
        return total;
    }
}
```

1. OOD 概述

❑ SOLID - Liskov替换原则



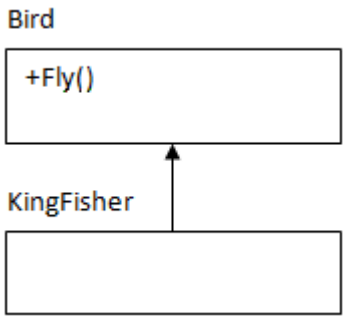
由芭芭拉·利斯科夫（Barbara Liskov）在1987年在一次会议上名为“数据的抽象与层次”的演说中首先提出。

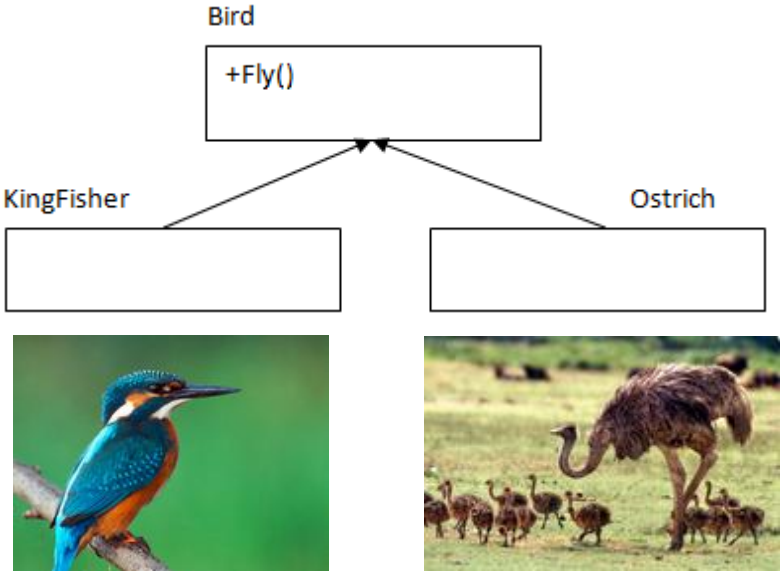
子类型必须能够替换它们基类型。

使用基类引用的函数必须能使用继承类的对象而不必知道它。

= “向上兼容”

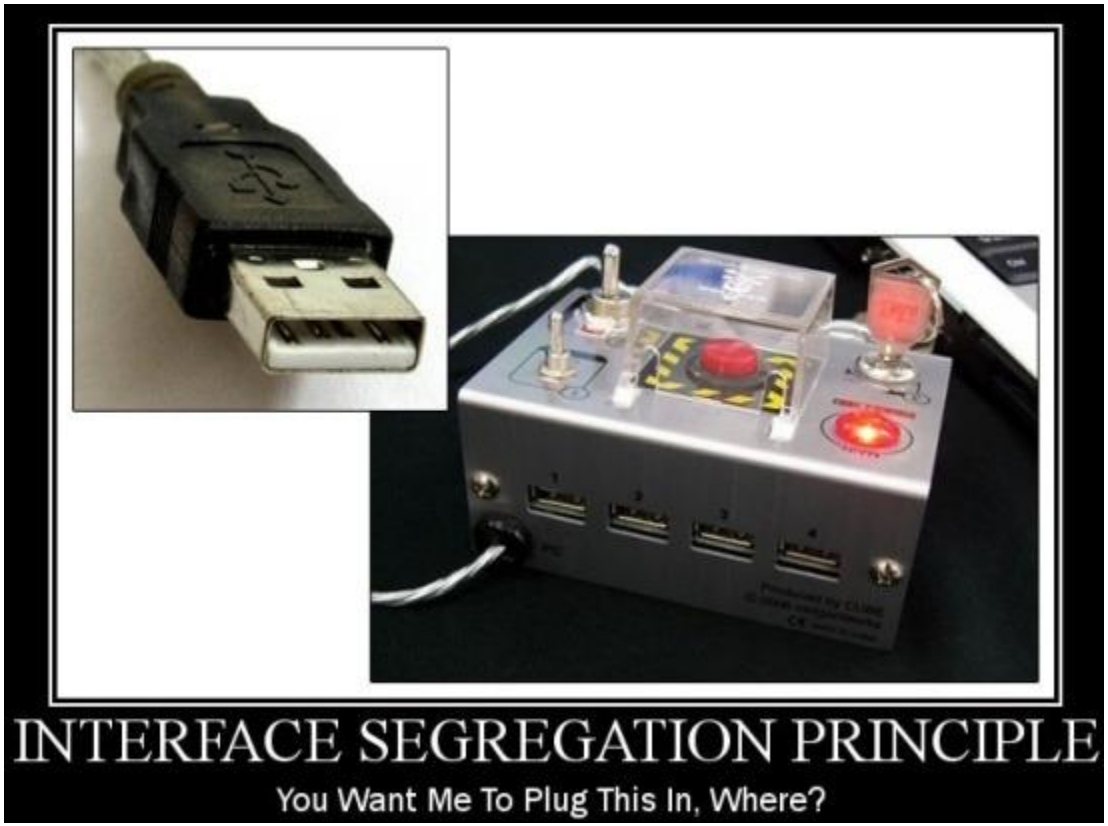
❑ SOLID - Liskov替换原则





- 如果没有LSP，类继承就会混乱：如果子类作为一个参数传递给方法，将会出现未知行为；
- 如果没有LSP，适用于基类的单元测试将不能成功用于测试子类；

❑ SOLID - 接口隔离原则



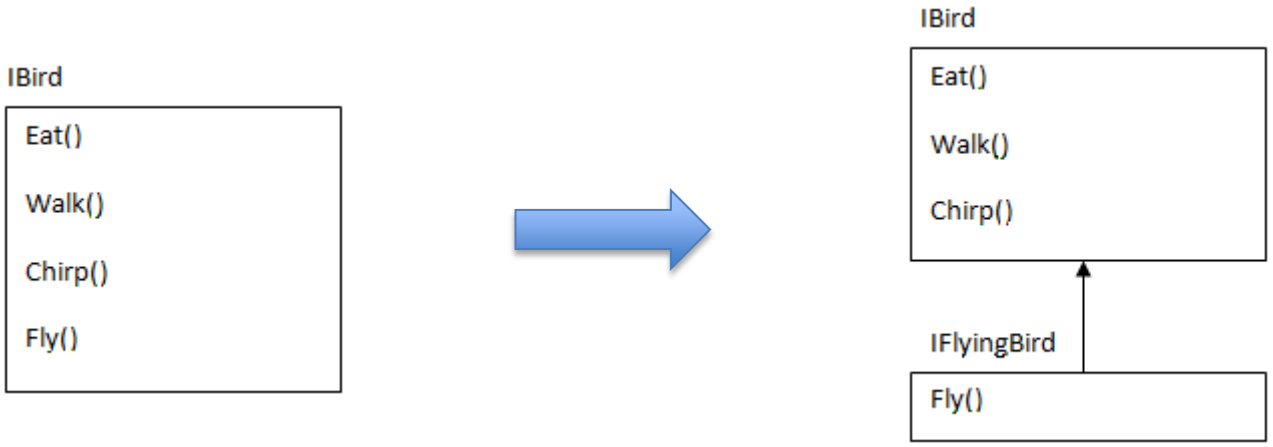
客户端不应该被迫依赖于它们不用的接口。

❑ SOLID - 接口隔离原则

假设你想买个电视机，你有两个选择。一个有很多开关和按钮，它们看起来很混乱，且好像对你来说没必要。另一个只有几个开关和按钮，它们很友好，且适合你使用。假定两个电视机提供同样的功能，你会选哪一个？

- 如果接口太大（胖接口），包含很多方法，在外界看来会很混乱，使其可用性降低；
- 如果一个类想实现该接口，那么它需要实现其所有的方法，尽管有些对它来说可能完全没用，这么做会在系统中引入不必要的复杂度；
- 接口隔离原则确保实现的接口有他们共同的职责，它们是明确的，易理解的，可复用的。

❑ SOLID - 接口隔离原则

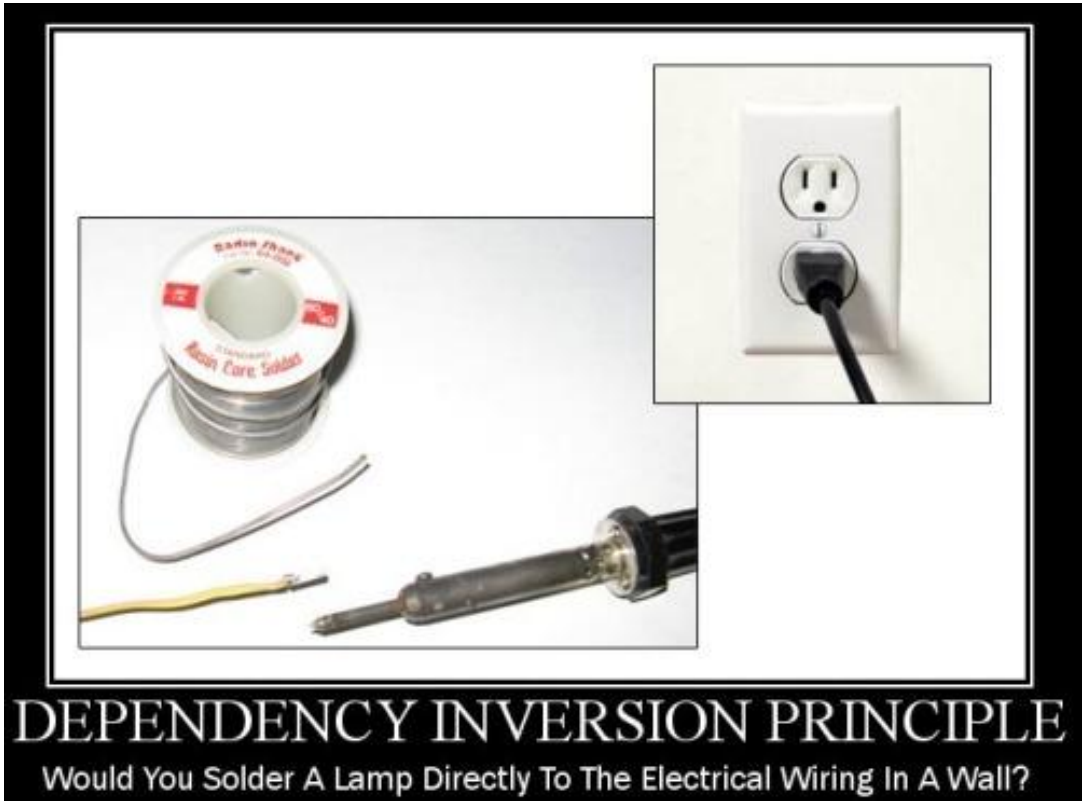


奥卡姆剃刀（Occam's Razor, Ockham's Razor）

❑ SOLID - 依赖倒置原则

A. 高层模块不应该依赖底层模块，两者都应该依赖其抽象。

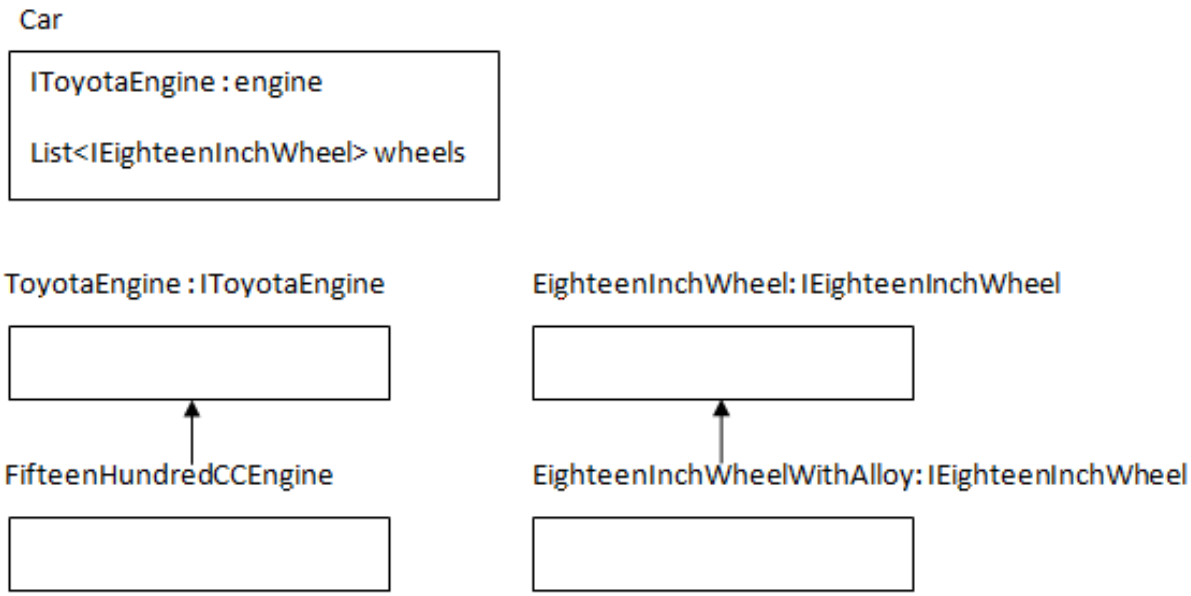
B. 抽象不应该依赖实现，实现应该依赖抽象。



合理设计接口（系统功能的抽象）

❑ SOLID - 依赖倒置原则

如何理解“inversion”？



高层模块：汽车
低层模块：引擎、车轮。

相比直接依赖于引擎或车轮，汽车应依赖于某些抽象的有规格的引擎或车轮。

❑ 其它原则

➢ 类设计原则（SOLID）

- 单一职责原则 SRP
- 开放闭合原则 OCP
- Liskov替换原则 LSP
- 接口隔离原则 ISP
- 依赖倒置原则 DIP

- 类要瘦
- 用多态
- 善用继承
- 接口要瘦
- 善用接口

➢ 其它设计原则

- "组合替代继承":这是说相对于继承，要更倾向于使用组合；
- "笛米特法则"：这是说"你的类对其它类知道的越少越好"；
- "共同封闭原则"：这是说"相关类应该打包在一起"；
- "稳定抽象原则"：这是说"类越稳定，越应该由抽象类组成";

其它原则

DRY

- Don't repeat yourself
- once and only once , 简称OAOO
- one rule, one place

KISS

- Keep It Simple, Stupid
- Keep It Sweet & Simple
- Keep it Simple, Sweetheart
- Keep it Simple, Sherlock

Uncle Bob

http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

2. OOD 模式实例

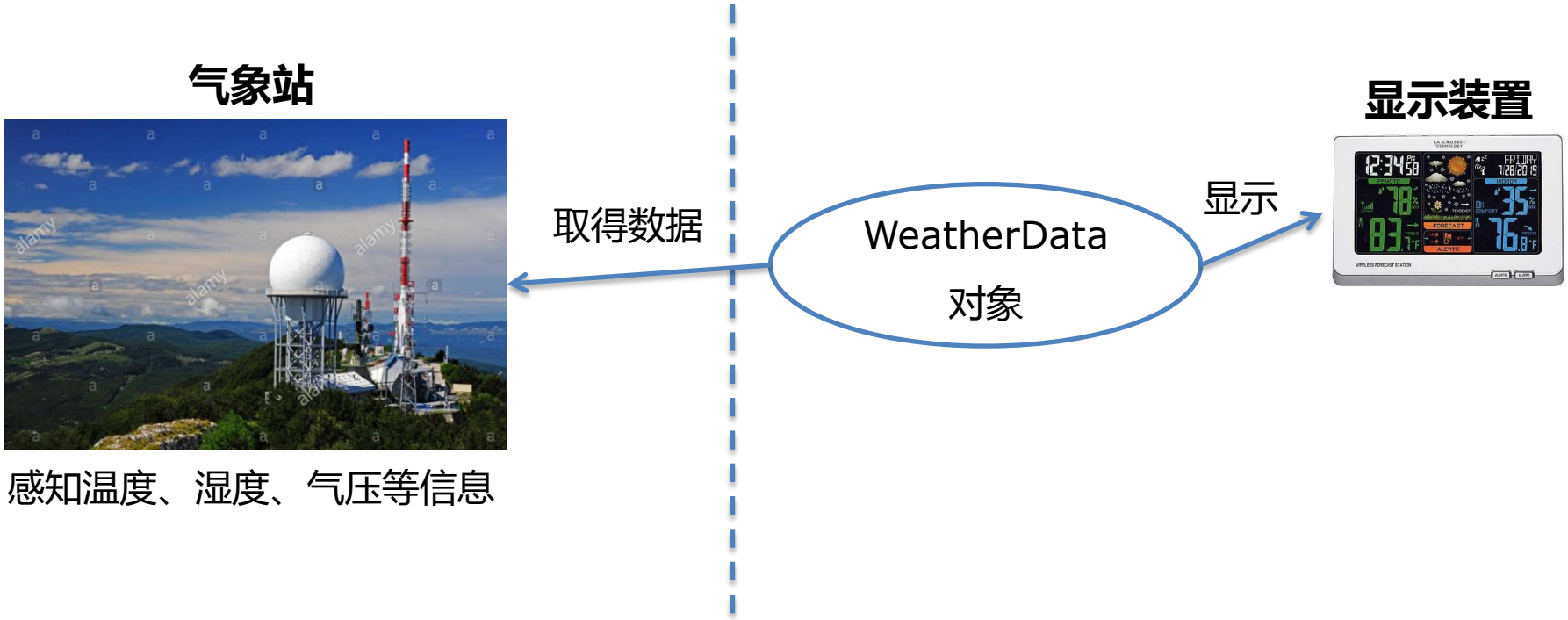
1) 观察者模式

2) 装饰者模式

3) 其它模式

2. OOD 模式实例

❑ 观察者模式 - 气象站



目标：建立一个应用，利用WeatherData对象取得数据，并更新3个布告板：

目前状况、气象统计和天气预报

2. OOD 模式实例

❑ 观察者模式 - 气象站

WeatherData
getTemperature() getHumidity() getPressure() measurementChanged() …

```
/**
 * 一旦气象测量更新，此方法会被调用
 */
public void measurementChanged() {
   // 你的代码放在这里
}
```

温度：25C
湿度：60
气压：low

1号布告板

平均温度
最高温度
最低温度

2号布告板

天气预报

晴天

3号布告板

?

2. OOD 模式实例

❑ 观察者模式 - 气象站

- 我们是针对具体实现编程，而非针对接口
- 每增加新的布告板，我们都得修改代码
- 我们在运行时动态地增加（删除）布告板
- 我们侵犯了WeatherData类的封装

```
/**
 * 一旦气象测量更新，此方法会被调用
 */
public void measurementChanged() {
  float temp = getTemperature();
  float humidity = getHumidity();
  float pressure = getPressure();

  currentConditionDisplay.update(temp, humodity, pressure);
  statisticsDisplay.update(temp, humodity, pressure);
  forecastDisplay.update(temp, humodity, pressure);
}
```

2. OOD 模式实例

❑ 观察者模式 - 订报纸

出版者 + 订阅者 = 订报纸

- 报社的业务是出版报纸
- 客户可以向报社订阅报纸，只要有新报纸出版，客户就会自动收到新报纸
- 当你不想再收到报纸的时候，只要取消订阅就可以了
- 只要报社还在运营，就会一直有人订阅或者取消订阅报纸

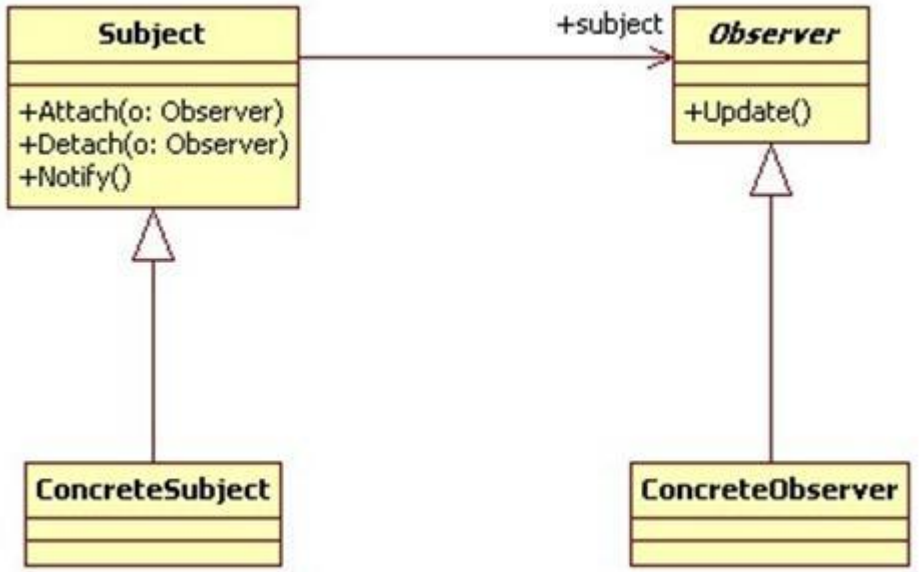
主题 + 观察者 = 观察者模式

Subject + Observer

❑ 观察者模式 - 定义

观察者模式定义了对象之间的一对多依赖，当一个对象改变状态时，它的所有依赖者都会收到通知并自动更新。

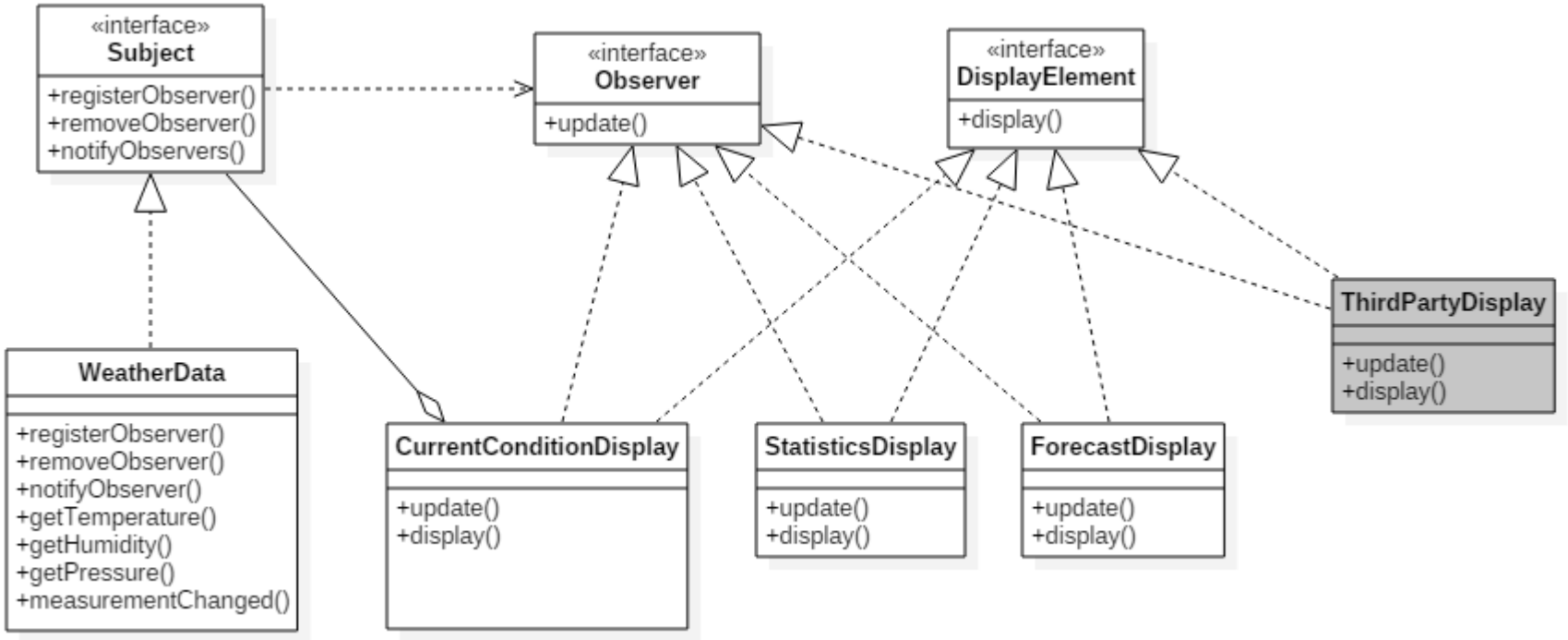
registerObserver()
removeObserver()
notifyObserver()



观察者模式提供了一种对象设计，让主题和观察者之间松耦合。

2. OOD 模式实例

❑ 观察者模式 - 气象站



2. OOD 模式实例

❑ 观察者模式 - 气象站

```
public interface Subject {
  public void registerObserver(Observer o);
  public void removeObserver(Observer o);
  public void notifyObservers();
}
```

```
public interface Observer {
  public void update(float temp, float humidity, float pressure);
}
```

```
public interface DisplayElement {
  public void display();
}
```

2. OOD 模式实例

❑ 观察者模式 - 气象站

```
public class WeatherData implements Subject {
  private ArrayList<Observer> observers;
  …
  public void notifyObservers(){
    for(Observer o : observers){
        o.update(temperature, humidity, pressure);
    }
  }

  public void measurementChanged(){
    notifyObservers();
  }

  public void setMeasurements(float t, float h, float p){
    this.temperature = t;
    this.humidity = h;
    this.pressure = p;
    measurementChanged();
  }
}
```

2. OOD 模式实例

❑ 观察者模式 - 气象站

```
public class CurrentConditionDisplay implements Observer, DisplayElement {
  private float temperature;
  private float humidity;
  private float pressure;
  private Subject weatherData;

  public CurrentConditionDisplay(Subject wd){
    this.weatherData = wd;
    this.weatherData.registerObserver(this);
  }

  public void update(float temp, float humidity, float pressure ){
    this.temperature = temperature;
    this.humidity = humidity;
    display();
  }

  public void display(){
    System.out.println(…);
  }
}
```

2. OOD 模式实例

❑ 观察者模式 - 气象站

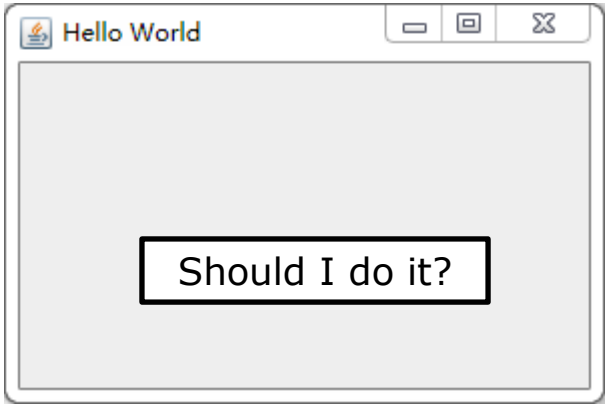
```
// 测试类
public class WeatherStation {
    public static void main(String[] args){
        WeatherData wd = new WeatherData();

        CurrentConditionDisplay cd = new CurrentConditionDisplay(wd);

        wd.setMeasurements(80, 65, 30.4f);
        wd.setMeasurements(82, 30, 23.2f);
        wd.setMeasurements(70, 55, 27.5f);
    }
}
```

2. OOD 模式实例

- 观察者模式 - JDK

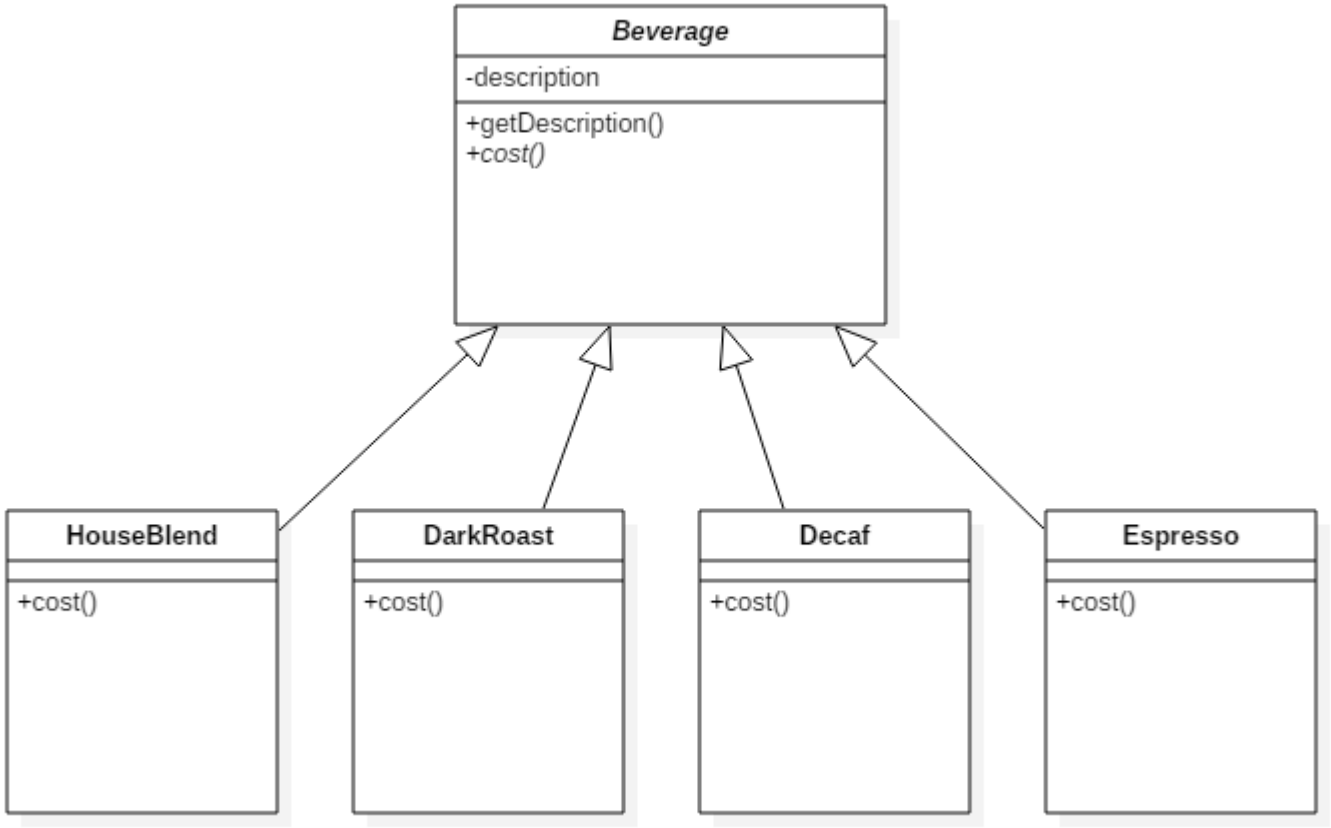


JButton

ActionListener

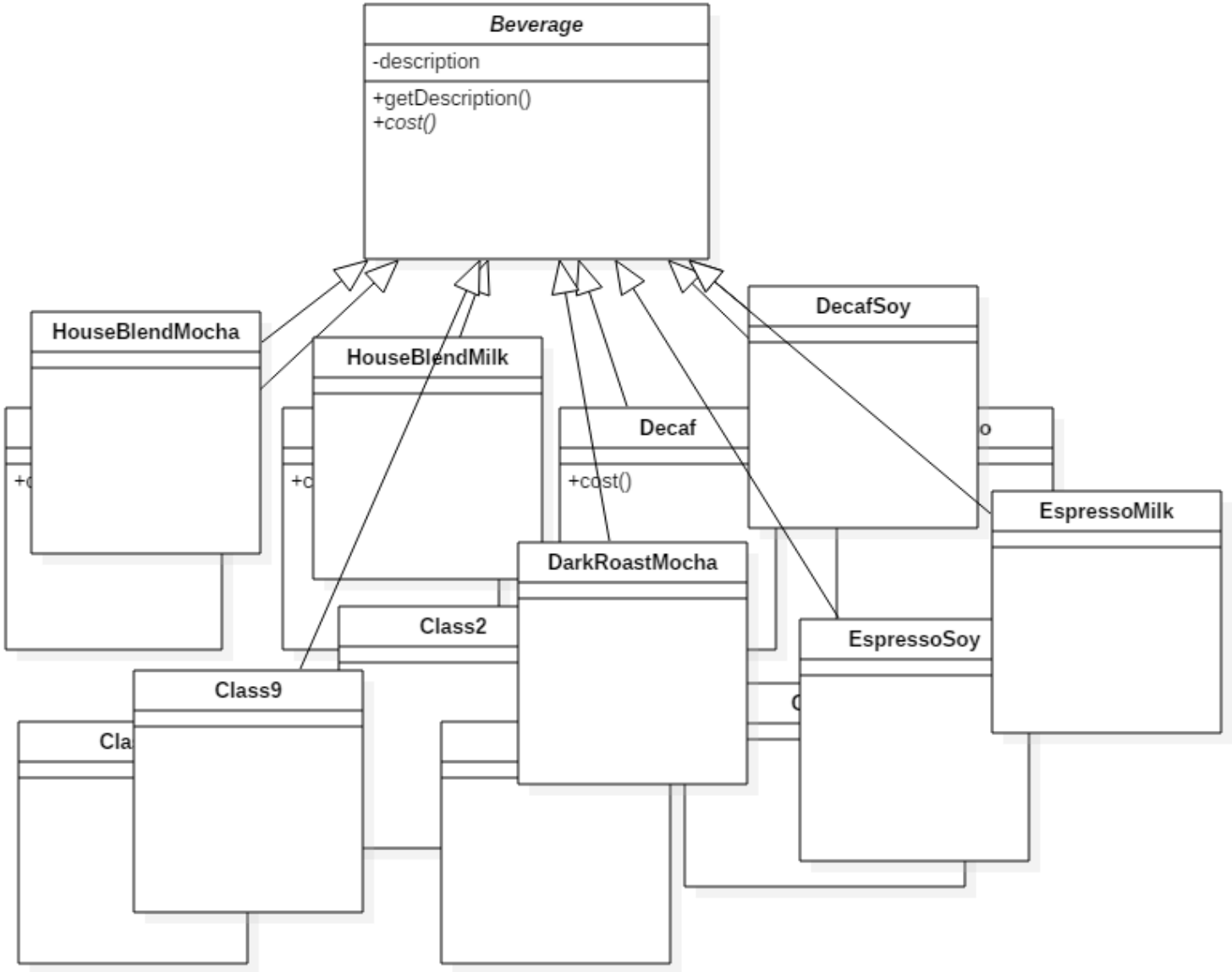
2. OOD 模式实例

❑ 装饰者模式 - 咖啡店



顾客在购买咖啡时可以选择加调料（费用会有不同），比如加蒸奶（Steamed Milk），豆浆（Soy）和摩卡（Mocha）

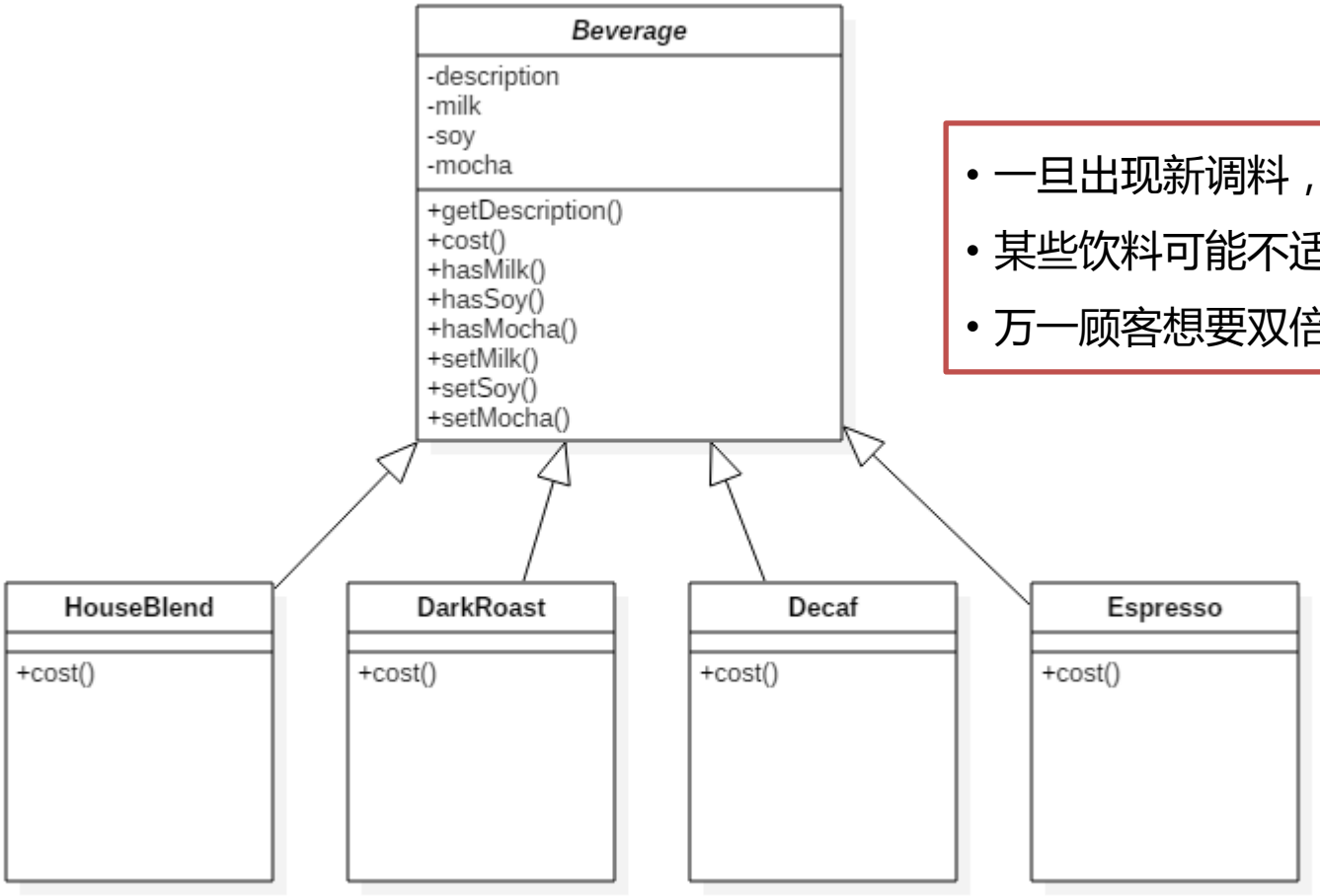
2. OOD 模式实例


Beverage
-description
+getDescription()
+cost()
HouseBlendMocha
HouseBlendMilk
Decaf
+cost()
DecafSoy
EspressoMilk
DarkRoastMocha
Class2
EspressoSoy
Class9

2. OOD 模式实例

❑ 装饰者模式 - 咖啡店

cost() 不再是抽象方法



- 一旦出现新调料，Beverage类就会大改
- 某些饮料可能不适合与某些调料组合
- 万一顾客想要双倍的摩卡？

2. OOD 模式实例

❑ 装饰者模式 - 咖啡店

问题

类的数量爆炸、设计死板、加入的新功能不一定适合所有子类

解决办法

以饮料为主体，在运行时以调料来“装饰”（decorate）饮料。

比如：

1）拿一个深焙咖啡（DarkRoast）对象

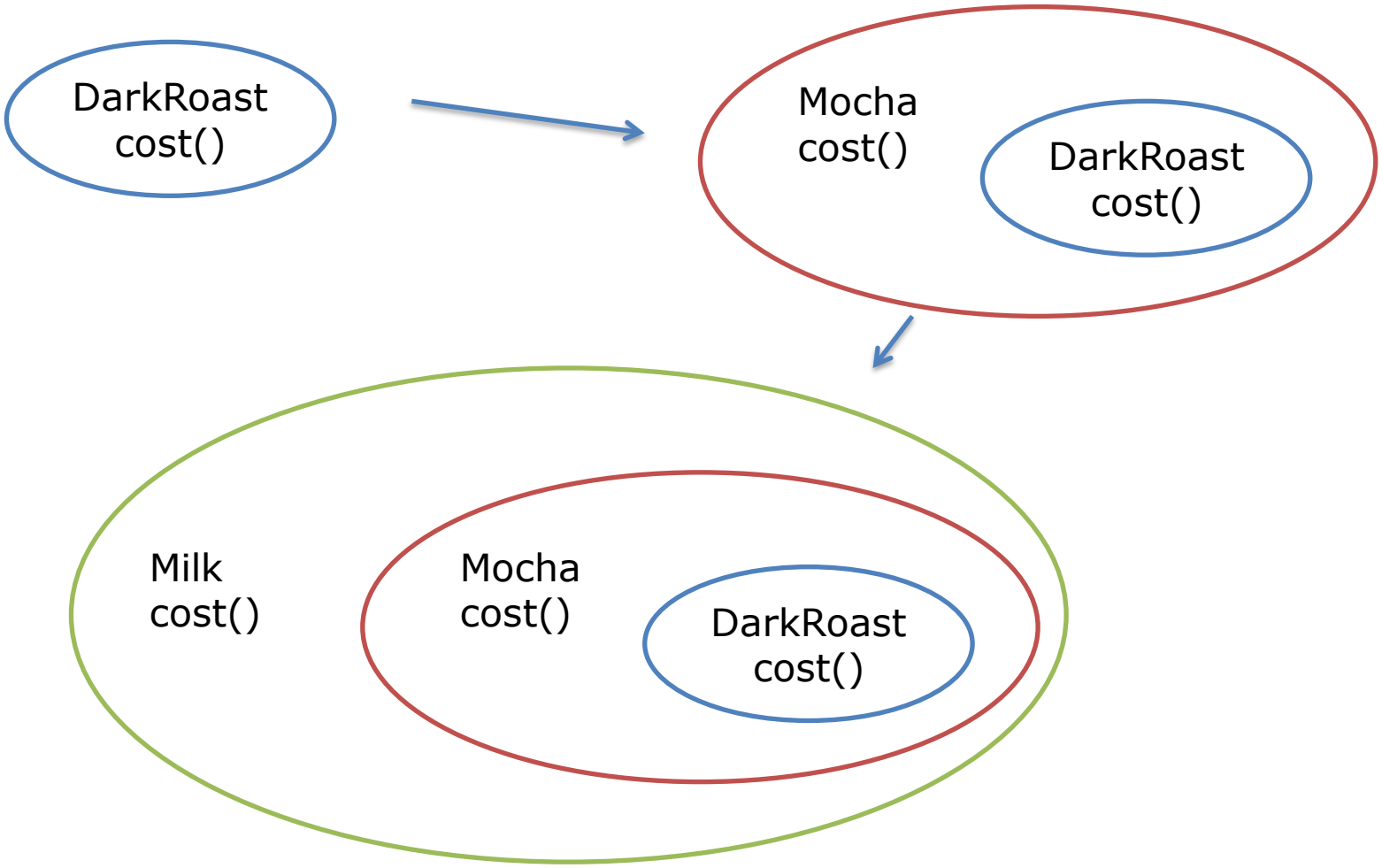
2）以摩卡（Mocha）对象装饰它

3）以牛奶（Milk）对象装饰它

4）调用cost() 方法，将调料的价格加上去

2. OOD 模式实例

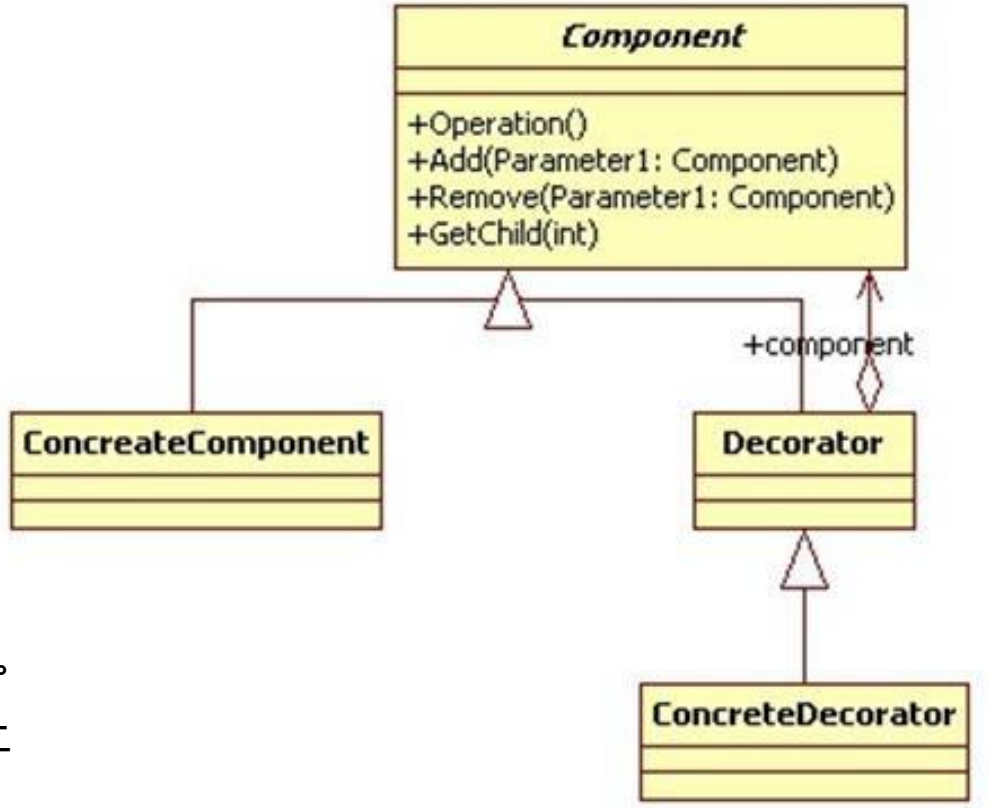
- 装饰者模式 - 咖啡店



2. OOD 模式实例

❑ 装饰者模式 - 定义

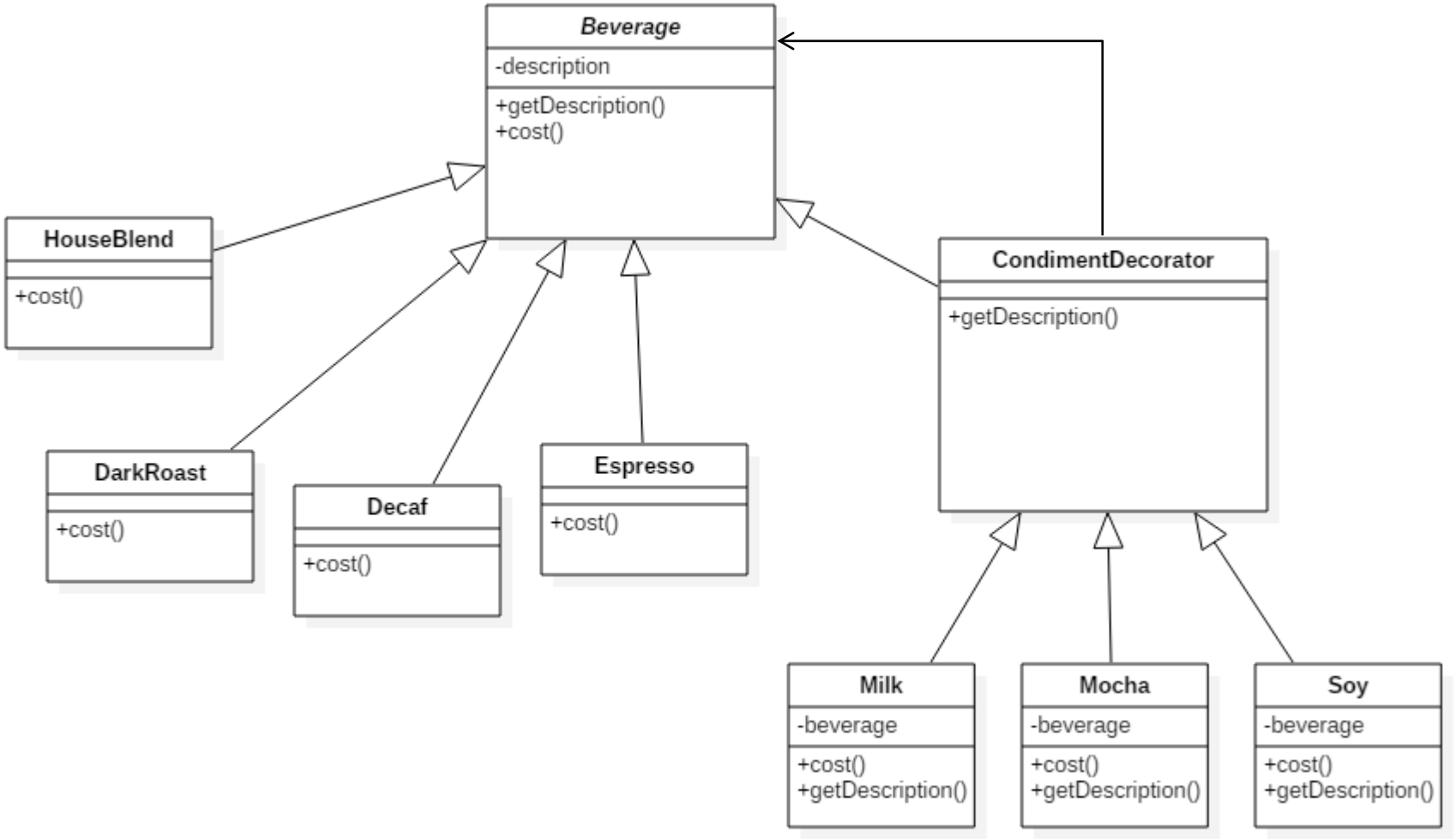
装饰者模式动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更有弹性的替代方案。



- 装饰者和被装饰对象拥有相同的超类型。
- 可以用一个或者多个装饰者包装一个对象。
- 装饰者可以在被装饰对象的行为前/后加上自己的行为，以达到特定的目的。
- 装饰过程可以在运行时动态地完成。

2. OOD 模式实例

❑ 装饰者模式 - 咖啡店



2. OOD 模式实例

❑ 装饰者模式 - 咖啡店

```
public abstract class Beverage {
    String description = "unknown";

    public String getDescription(){
        return this.description;
    }

    public double cost();
}
```

```
public class HouseBlend extends Beverage {
    public HouseBlend(){
        this.description = "House Blend Coffee";
    }

    public double cost(){
        return 1.99;
    }
}
```

2. OOD 模式实例

- 装饰者模式 - 咖啡店

```
public class Mocha extends CondimentDecorator{
    Beverage beverage;

    public Mocha(Beverage b){
        this.beverage = b;
    }

    public String getDescription(){
        return this.beverage.getDescription() + ", Mocha";
    }

    public String cost(){
        return 0.2 + this.beverage.cost();
    }
}
```

2. OOD 模式实例

❑ 装饰者模式 - 咖啡店

```
// 测试类
public class CoffeeShop {
     public static void main(String[] args){
        Beverage b1 = new Espresso();
        System.out.println(b1.getDescription() + ": " +
    b1.cost());

        Beverage b2 = new DarkRoast();
        b2 = new Mocha(b2);
        b2 = new Mocha(b2);
        b2 = new Milk(b2);
        System.out.println(b2.getDescription() + ": " +
    b2.cost());
    }

}
```

2. OOD 模式实例

❑ 装饰者模式 - JDK

Java IO 类即采用了装饰者模式

```
BufferedReader inputStream = new BufferedReader(new FileReader("xanadu.txt"));
BufferedWriter outputStream = new BufferedWriter(new FileWriter("output.txt"));

String s;
while ((s = inputStream.readLine()) != null) {
    outputStream.write(s);
}
…
outputStream.flush();
```

2. OOD 模式实例

❑ 其它模式

工厂模式

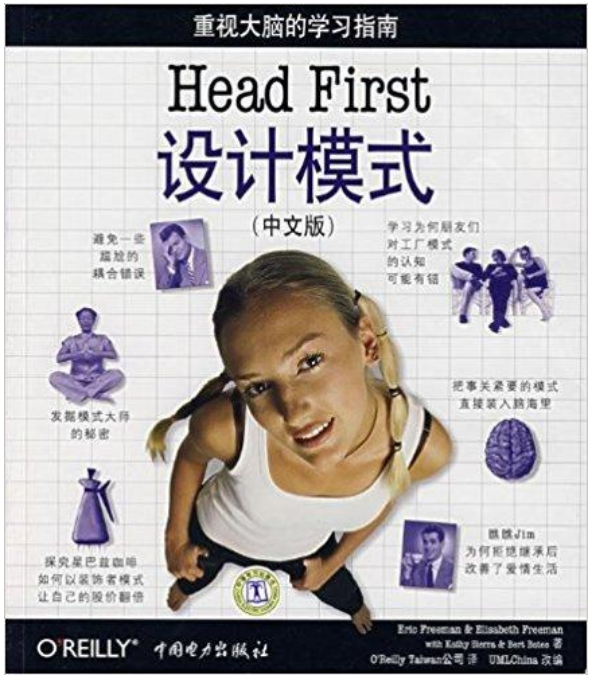
单例模式

适配器模式

迭代器模式

代理模式

等等...



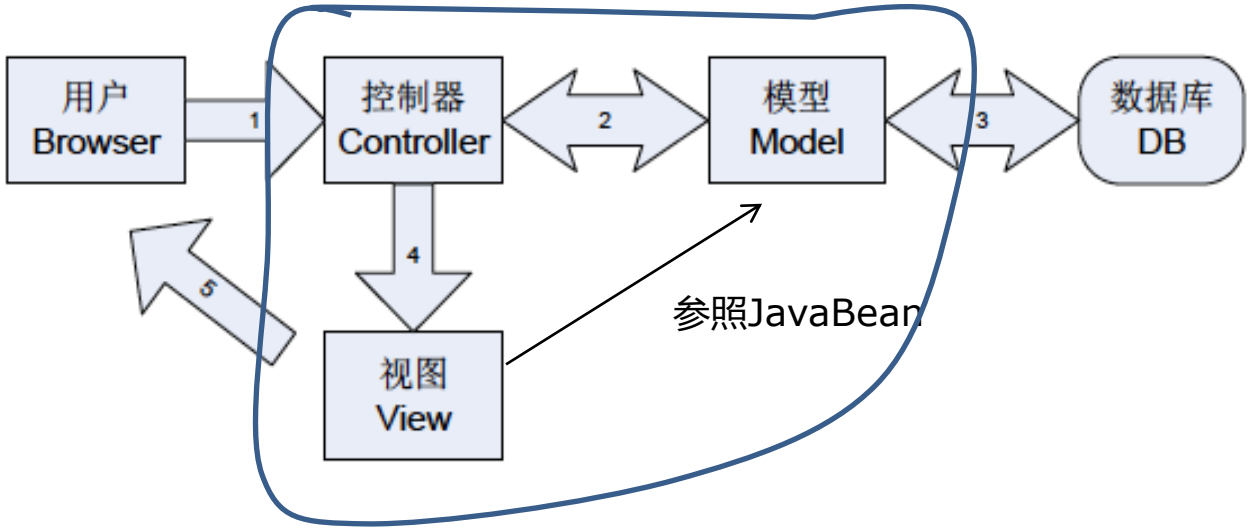
3. OOD 构架

- 模型-视图-控制器 Model-View-Controller (MVC)

游戏设计

3. OOD 构架

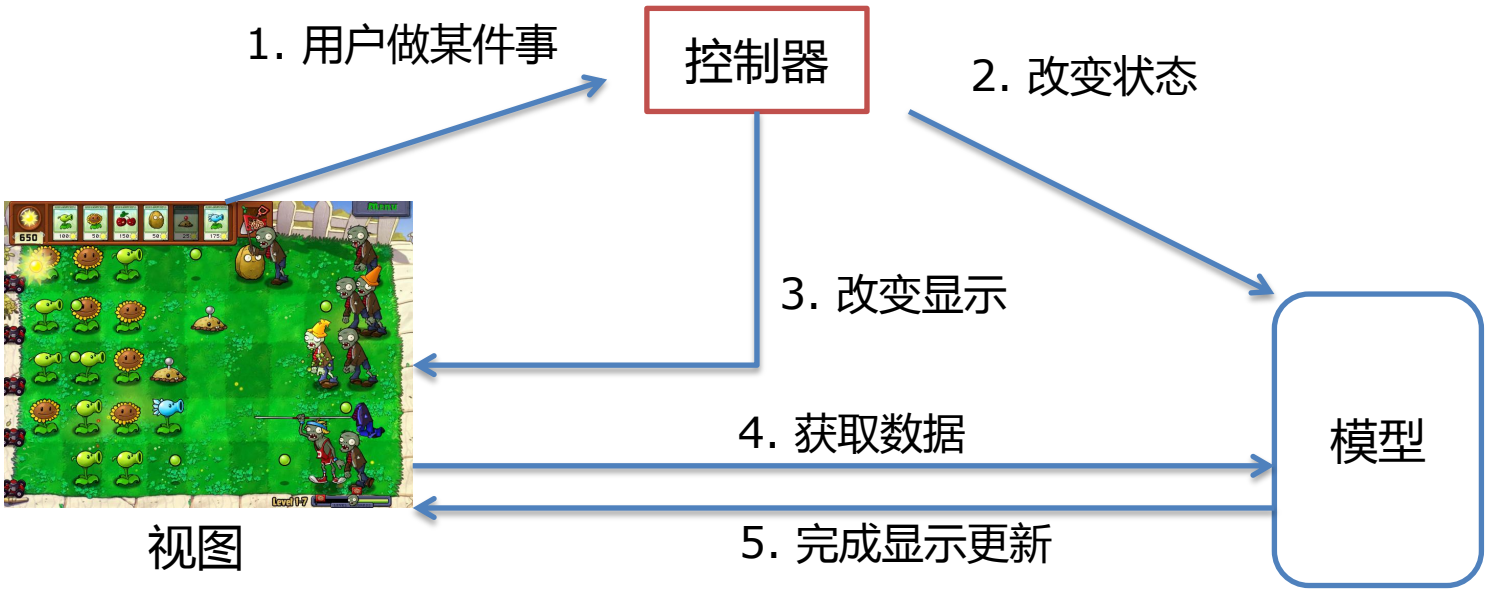
- 模型-视图-控制器 Model-View-Controller (MVC)



Web程序开发

3. OOD 构架

- 模型-视图-控制器 Model-View-Controller (MVC)



3. OOD 构架

❑ 三层程序设计模型 Three-tier design model

➢ 表示层（用户界面）

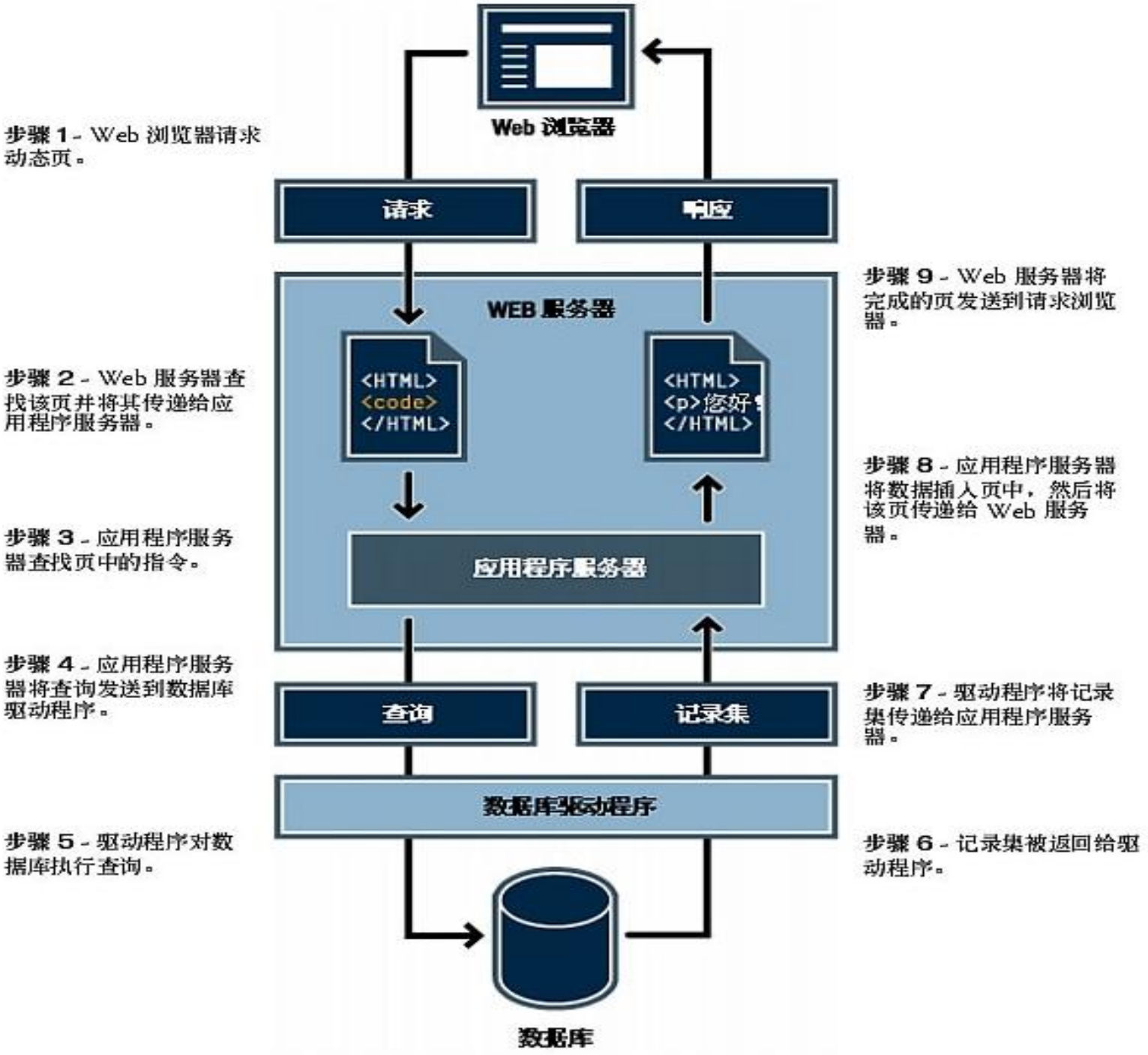
- GUI class
- 显示数据，接收操作

➢ 业务逻辑层（问题域层）

- PD class
- 从用户界面接收请求，根据业务逻辑处理请求，从DA类获得数据/向DA类发送数据，将处理结果送回表示层

➢ 数据访问层

- DA class
- 建立与数据库的连接，完成增删改查，关闭连接。


步骤 1 - Web 浏览器请求动态页。
Web 浏览器
请求
响应
步骤 9 - Web 服务器将完成的页发送到请求浏览器。
WEB 服务器
步骤 2 - Web 服务器查找该页并将其传递给应用程序服务器。
<HTML>
<code>
</HTML>
<HTML>
<p>您好
</HTML>
步骤 8 - 应用程序服务器将数据插入页中，然后将该页传递给 Web 服务器。
步骤 3 - 应用程序服务器查找页中的指令。
应用程序服务器
步骤 4 - 应用程序服务器将查询发送到数据库驱动程序。
查询
记录集
步骤 7 - 驱动程序将记录集传递给应用程序服务器。
数据库驱动程序
步骤 5 - 驱动程序对数据库执行查询。
步骤 6 - 记录集被返回给驱动程序。
数据库

3. OOD 构架

❑ 分层的好处

- 使程序在逻辑上保持相对独立
- 允许更灵活地选用相关软硬件系统
- 允许并行开发
- 便于保护业务逻辑以及数据

总结

- 理解OOP概念
- 记住OOD原则
- 实践是检验真理的唯一标准

下节预告

课程设计引导课 3