《面向对象程序先导》Lec7-设计模式入门

北京航空航天大学计算机学院

吴 际

2023.10.27

目录

- 面向对象设计的核心机制
- 设计模式的概念分析
- 策略模式及其应用场景
- 观察者模式及其应用场景
- 工厂模式及其应用场景
- 作业内容介绍

面向对象设计的核心机制

- **封装**：内部复杂性外部不可见
  - 状态和行为封装在类中，隐藏内部实现细节
  - 对象之间通过方法进行交互
- **抽象**：通过抽象层次来协同降低复杂性
  - 利用继承或接口实现机制，建立数据或行为抽象层次
  - 形成代码重用和层次化设计，避免冗余
- **多态**：通过多种形态来解藕处理行为的内在复杂性
  - 按照处理行为的内在差异，设计为多个名字相同但处理逻辑不同的方法
  - 重载和重写

多态

- Polymorphism: poly (many) + morphism (forms)
- 动物都会运动，不同动物运动方式不同：
  - 鸟会飞
  - 狗会跑
  - 鱼会游
- 假设开发一个动物运动模拟程序，如何设计？

多态

```
//client code

// 创建对象
Bird bird = new Bird();
Dog dog = new Dog();
Fish fish = new Fish();
// 使用
bird.fly();
dog.run();
fish.swim();
```

- 当动物种类增多时?
  - 新增动物类型时，client code需要知道新增的动物**对象类型及其运动方法**，维护变得困难
- 能否为所有的动物提取共性行为，然后由各个动物“自己”来细化具体的行为?
  - 即便将来新增动物类型也不必改写client代码!
- **父类定义的共性行为可由子类根据自身的具体情况来实现**

基于接口的多态实现

- 将鸟、狗、鱼抽象为动物，飞、跑、游都是动物运动方式的一种

```
interface Animal {
    void move();
}
public class Bird implements Animal {
    public void move() {
        System.out.println("Bird fly");
    }
}
public class Dog implements Animal {
    public void move() {
        System.out.println("Dog run");
    }
}
public class Fish implements Animal {
    public void move() {
        System.out.println("Fish swim");
    }
}
```

```
//client code
// 创建对象
List<Animal> animals = {new Bird(), new Dog(),  new Fish()};
// 使用
for (Animal animal : animals) {
    animal.move();
}
```

Java的对象方法调用是基于运行时的实际类型的动态调用，而非变量的声明类型

针对抽象编程，而不是针对实现编程

多态

- Client代码**不需要**知道传进来的具体动物对象类型
  - 只需抓住Animal接口类型
  - 任何新增的某种具体动物类只需实现Animal接口即可
- 也可以通过继承方式来实现，建立一条继承链
  - 允许添加更多类型的子类实现功能扩展，**不需要**修改父类代码
  - 子类直接重写实现父类所定义的move方法，达到相同的效果

设计模式（design pattern）

- 模式：针对某种具有一定特征的问题所形成的一般性解决方案
  - 可以适用于多种不同的具体应用，关键是识别其中的问题特征
- 模式组成
  - 名称
  - 动机，问题及其特征
  - 解决方案
  - 参与者和协作者
  - 效果
  - 实现

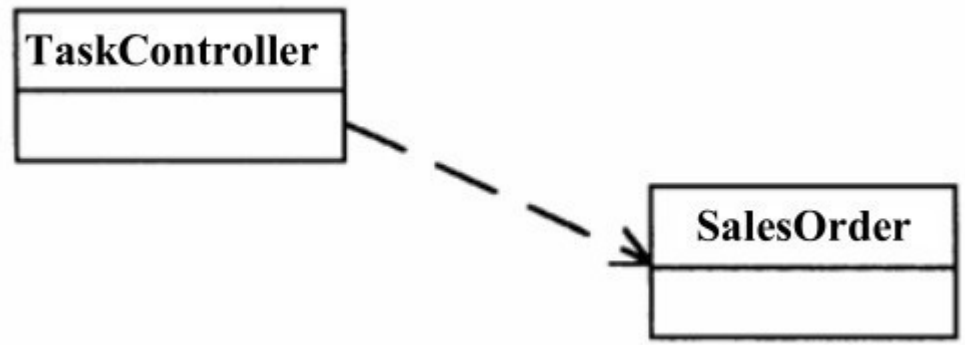
设计模式

- 为什么学习设计模式
  - 复用既有的、高质量的解决方案来应对常见问题
  - 理解和形成初步的设计思维
  - 提升观察分析和设计的层次——既见树木，又见森林
  - 提高代码的可修改性和可维护性

设计模式和面向对象设计是相得益彰的关系

案例分析：电子商务系统

- 某电子商务公司的订单处理系统目前能够处理国内订单
  - 其中有一个控制器类，用于（代理式）处理销售请求
  - 当请求涉及某销售订单时，把请求转发给相应的SalesOrder对象处理
- SalesOrder类的功能包括：
  - 创建订单
  - 处理税额计算
  - 处理订单（如配货和状态跟踪等）
  - 打印订单内容



案例分析：电子商务系统

- 公司业务发展，要求能够处理多个不同国家（地区）的订单
  - 新增缴税规则需求，需要处理中国之外地区的订单税额计算
- 如何处理
  - 业务层没有变化，都是订单处理
  - 但订单的具体处理有变化
- 复制和粘贴?
  - 忽略重用的可能性，增加维护成本

案例分析：电子商务系统

- 在SalesOrder中按照国家（地区）进行特判处理

```
// 打印订单
switch (nation) {
        case China:
        // 汉语
        break;
        case US:
        case Canada:
        // 英语
        break;
        }
```

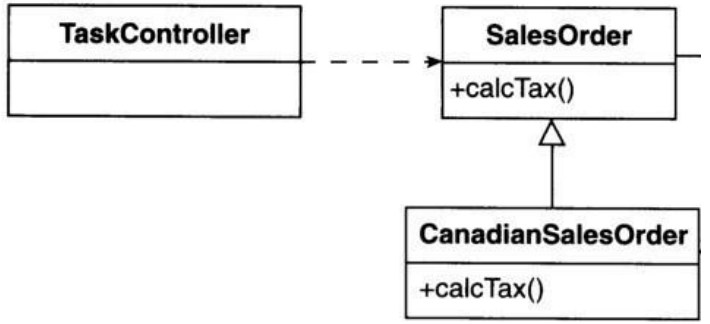
如果选项更多呢？

```
// 在分支内部添加选项，加拿大魁北克说法语
switch (nation) {
        case China:
        // 汉语
        break;
        case US:
        // 英语
        case Canada:
        if (inQuebec)
        // 法语
        else // 英语
        break;
        }
```

分支蔓延：每次增加新的分支时，程序员都必须搜遍各个角落，找出可能涉及的所有分支，增加了耦合度，测试难度急剧上升

案例分析：电子商务系统

- 使用继承的方式进行处理
  - 从缴税规则角度来建立抽象层次
    - 不同的销售订单类有不同的缴税规则
  - 很多国家不同地区的计税规则都有差异，如何处理？
  - 如果多种订单的运费计费规则发生变化呢？

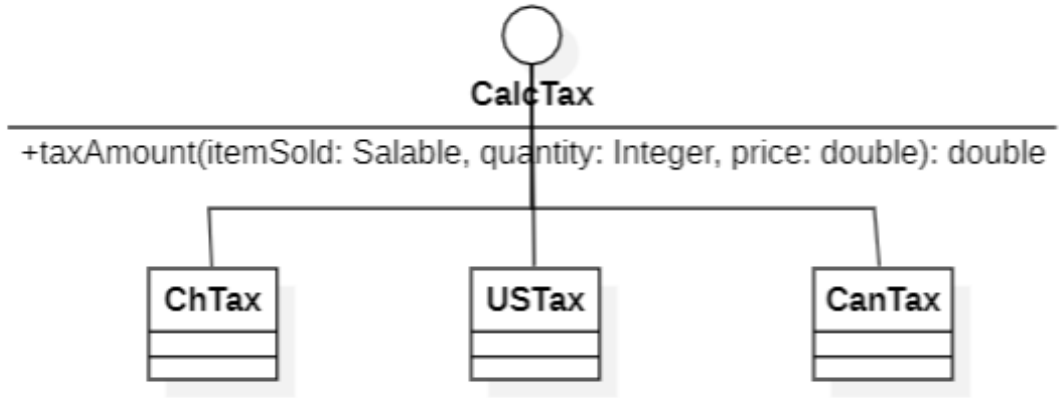


继承层次太深将导致程序难以理解（弱内聚）、存在冗余、难以测试而且多个概念耦合在一起

案例分析：电子商务系统

1. 考虑设计中什么可变
2. 对变化的数据或行为进行封装

1. 发现变化并封装之
   - 缴税规则是**变化**的
   - “封装”意味着创建**接口**（概念上完成税额计算），然后为每种变化创建类实现接口



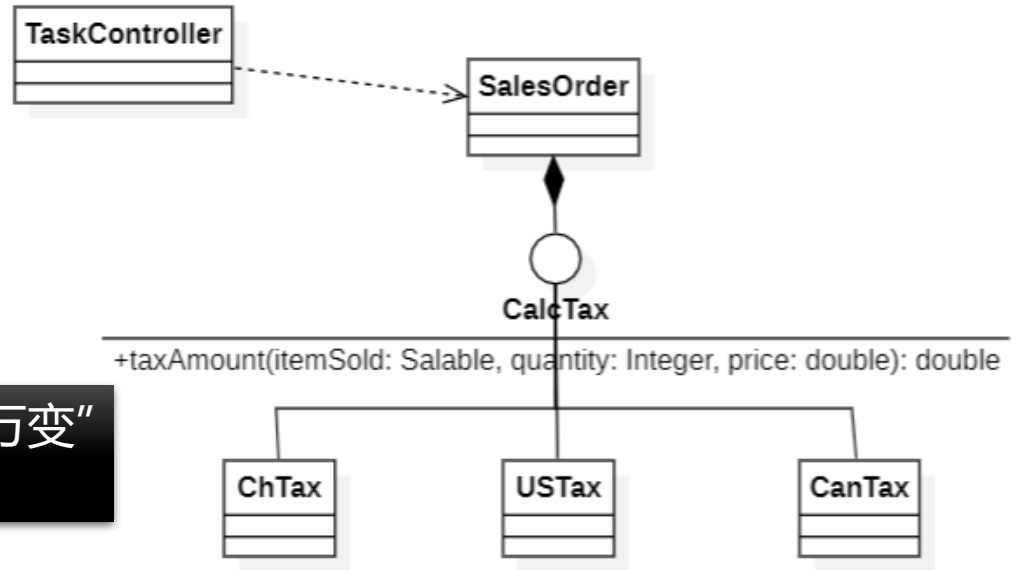
案例分析：电子商务系统

1. 考虑设计中什么可变
2. 对变化的数据或行为进行封装
3. 优先使用对象聚集，而不是类继承

2. 组合优先

组合取代继承，意味着不必创建不同版本的销售订单（使用继承）

利用接口来统一管理各种不同的对象



SalesOrder封装了计税规则的细节，抓住了“不变”以应对“万变”
TaskController对order的处理只关注顶层业务逻辑要求

案例分析：电子商务系统

```
public class TaskController {
    ...
    public void process(int orderID) {
        SalesOrder order = ...;
        order.process();
    }
}
public class SalesOrder {
    private CalcTax tax;
    private String country;
    private Salable itemSold;
    private int quantity;
    private double price;
    ...
    private CalcTax getTaxRulesForCountry() {
        // 获取所在国的计税规则
    }
    public SalesOrder(...){...; tax = getTaxRulesForCountry();...}
    public double process() {
        return tax.taxAmount(itemSold, quantity, price);
    }
}
```

```
public interface CalcTax {
    public double taxAmount(Salable itemSold, Integer quantity, double
price);
}
public class ChTax implements CalcTax {
    @Override
    public double taxAmount(Salable itemSold, Integer quantity, double
price) {
        // 中国计税规则
    }
}
```

提高内聚度，有助于灵活性

在有新的缴税需求时，只需创建一个新类实现 CalcTax 接口

定义一系列具体实现，并把它们封装起来，使得在顶层可忽略其差异：

Strategy模式

策略模式（Strategy Pattern）

- 顶层归纳类的共性行为职责，允许每个类采取不同的实现策略
  - 类有确定的行为职责，其实现与其所管理的数据紧密相关（独立性）
  - 一组类的行为职责在概念上相同或相似，只是实现有差异
  - 把差异实现为策略（独立的类），上层类通过聚合来引用策略
  - Client code看不到策略差异，能够进行统一管理
- 优点
  - 对策略进行封装，且相互独立
  - 可扩展性好，增加新的实现策略不影响client code和已有实现策略

国际电子商务系统案例

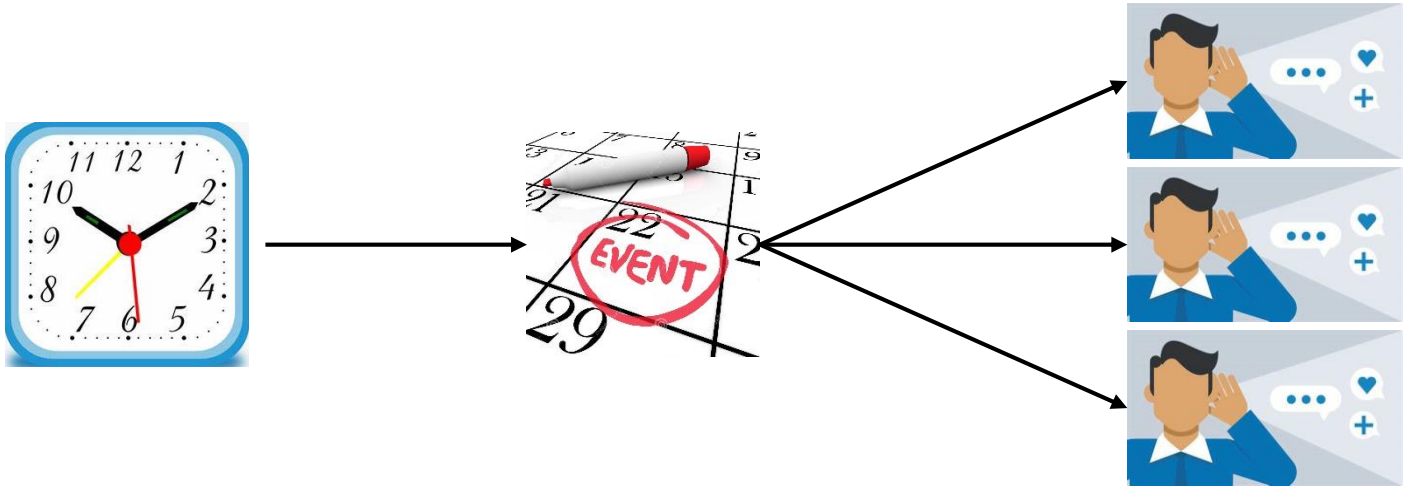
- 增加**订单状态的通知功能**
  - 订单状态会随触发事件而改变（下单，付款，发货，运输，已送达等）
  - 订单用户和客服人员等需要实时获取订单状态的更新信息
- 如果订单用户不止一位呢?
  - A为好朋友B购买了一件礼品
- 通知方式存在多种选择
  - 手机短信订阅、公众号消息推送、邮件通知等

```
public class Order {
    private String status;
    private String productName;
    public Order(String status) {
        this.status = status;
    }
    public void setStatus(String newStatus) {
        this.status = newStatus;
        notifyCustomer();
        notifyCustomerService();
    }
    private void notifyCustomer() {
        …
    }
    private void notifyCustomerService() {
        …
    }
}
```

国际电子商务系统案例

- 什么时候通知?
  - 订单状态变化
  - 通知发起方确定
- 对谁通知?
  - 用户和客服
  - 如何管理被通知对象
- 怎样通知?
  - 通知方式
  - 语义相同，但是实现机制不同

国际电子商务系统案例



- **观察者模式**：当一个对象的状态发生改变时（发生了所关注的事件），所有依赖于它的对象都将得到通知
  - **一对多的依赖关系**
- 观察者Object：希望获得通知的对象
- 目标Subject：触发事件的对象

观察者模式（Observer Pattern）

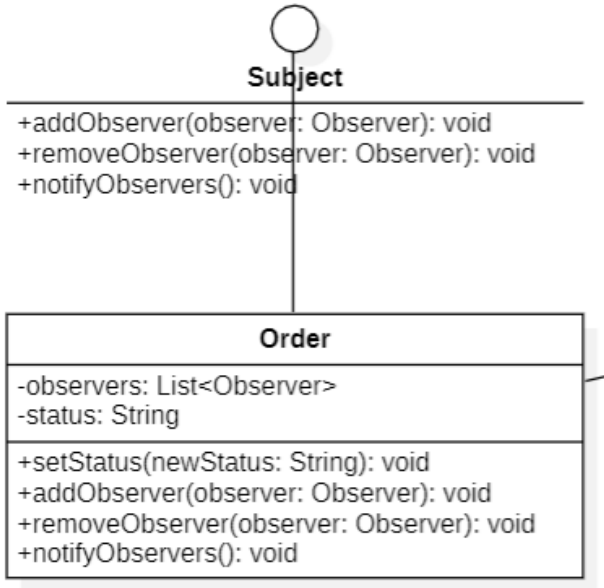
- 寻找变化，封装和抽象化处理
  - **不同类型的观察者对象：**有一系列对象需要在状态发生变化时获得通知
    - 这些对象往往属于不同的类
  - **不同的通知接口：**不同的类有不同的接口/方法来获得通知
  - **不同的通知方式：**谁来确定使用那种通知方式?

观察者模式

1. 让观察者实现统一的**通知接收接口（Observer）**
2. 观察者对象如何确定：主动注册机制
   通知者或者某个控制器提供统一的通知者注册与取消接口
   addObserver(Observer)；deleteObserver(Observer)
3. 事件发生时通知观察者：统一处理机制
   通知者类实现一个 notify方法来遍历其观察者列表，调用其update方法
4. 观察者自行决定通知方式
   表现为通知接口的具体实现
5. 从通知中获取额外信息
   观察者可以根据update接口传入的参数来了解额外信息

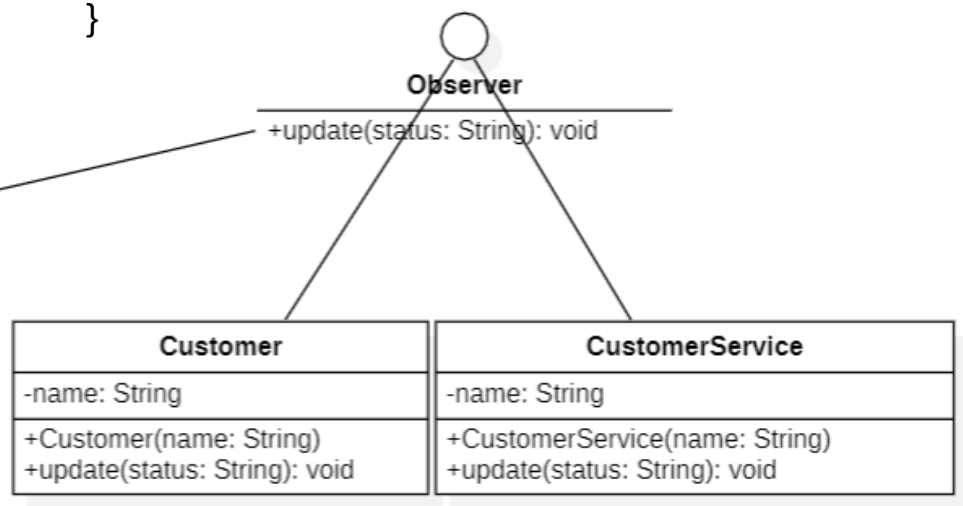
国际电子商务系统案例

```
public class Order implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String status;
    public void setStatus(String newStatus) {
        this.status = newStatus;
        notifyObservers();
    }
    @Override
    public void addObserver(Observer observer) {
        observers.add(observer);
    }
    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }
    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(status);
        }
    }
}
```



update方法甚至可以传入某种封装了的Context对象

```
public class Customer implements Observer {
    private String name;
    public Customer(String name) {
        this.name = name;
    }
    @Override
    public void update(String status) {
        System.out.println("Customer" + name + ":
Order status updated - " + status);
    }
}
```



观察者模式

- 高可扩展性
  - Observer与Subject之间松耦合
  - 添加新的Observer类不影响Subject类的业务逻辑
- 讨论1：如何引入可变的通知方式?
  - 多种通知方式，应由Observer实现类来选择
  - 通知方式需要通信机制的支撑
- 讨论2：如何进一步抽象引入面向事件的通知方式?
  - 进一步把变化抽象为事件

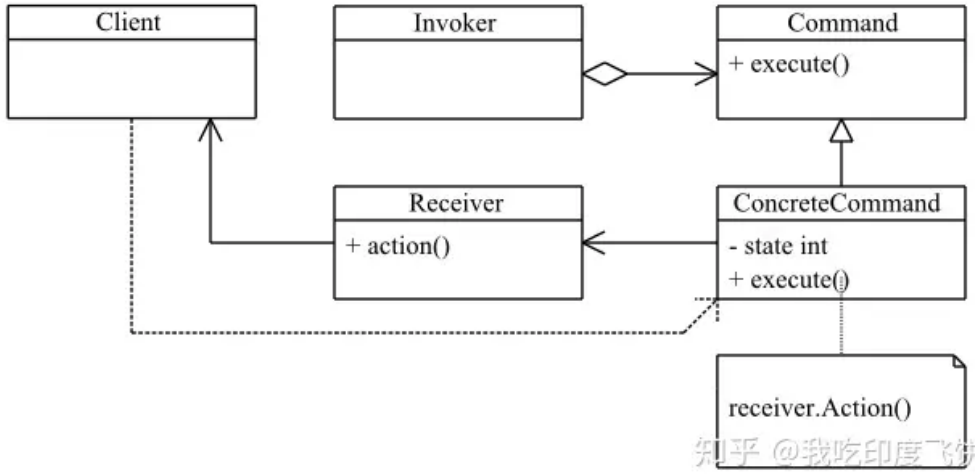
如果通知方式的实现需要物理机制，可以在具体Observer实现类中应用策略模式来应用相应的通知机制

观察者模式

- 面向事件进行通知的业务场景
  - 用户和客服不期望收到快递到达中转站的通知，而配送员需要
  - 根据Event与Observer之间的映射表，来动态确定需要实际被通知的observer对象 (hashmap?)
- 事件处理需要的信息
  - 不同的事件、不同的观察者需要有不同的信息来处理相应的状态变化（即通知）
    - 快递发出事件：承运商、包裹ID、始发地等
    - 快递到达事件：包裹ID、存放地点、取件码等
  - 可用Strategy模式来对这些信息进行封装，通过update方法进行传递

命令模式(Command Pattern)

- 还记得上节课介绍的处理办法吗



- 如果使用接口机制来解决呢?
  - Step 1：设计Command接口（抽象处理能力）
  - Step 2：设计完成具体指令业务的类（具体处理能力）
    - BottleOperator，FoodOperator，…
    - 提供加入、删除和查询等operation
  - Step 3：设计具体指令映射类
    - BottleOperationCommand, FoodOperationCommand, …
    - 实现Command接口 → 交由BottleOperator对象等来完成具体业务操作
    - 分别管理bottle，food等相关的对象
  - Step 4：设计command调度器cmdInvoker
    - 设置ArrayList<Command> cmdList；
    - 从scanner获得具体指令，按照类别创建BottleOperationCommand对象，加入cmdList
    - 按照某种策略执行cmdList中的每个cmd.execute(message);

```
public interface Command {
    void execute(String message);
}
```

工厂模式（Factory Pattern）

- 用于创建对象的模式，提供了灵活而具有扩展性的解决方案
  - 如果对象创建只是简单new XXX(...)，需要这个模式吗?
- 将复杂的对象创建工作隐藏起来，仅暴露接口供客户使用
  - 在不同情况下需要创建不同类型的对象（但client code不关心具体使用哪些类型来创建）
  - 需要为对象创建准备复杂的内容

国际电子商务系统案例

新增需求：为了实现销售多种类型产品，需要创建各种产品，如电脑、手机、上衣、裤子等。

1．直接在客户端创建产品

```
public class ProductControl {
    public void createProduct(Type type) {
        switch (type) {
            case Computer:
                // 创建一款电脑产品
            case Mobile_Phone:
                //创建一款手机产品
            case Pants:
                // 创建一款裤子产品
            }
    }
}
```

优点：

直接追踪到功能要求，直观易理解

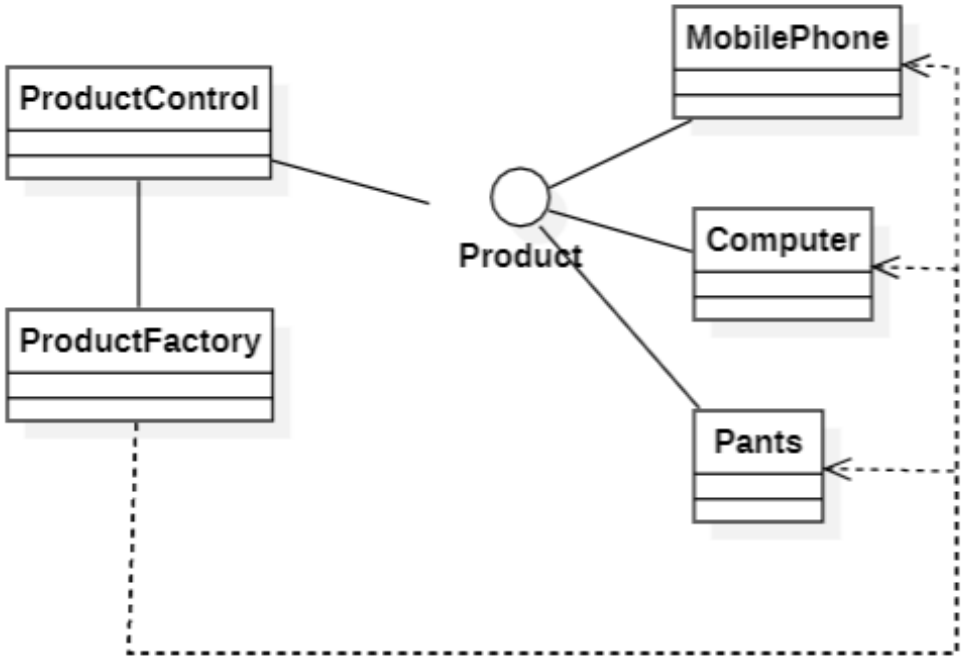
缺点：

商户每新增一款产品，就需要修改客户端代码，繁琐且容易导致大量重复代码

国际电子商务系统案例

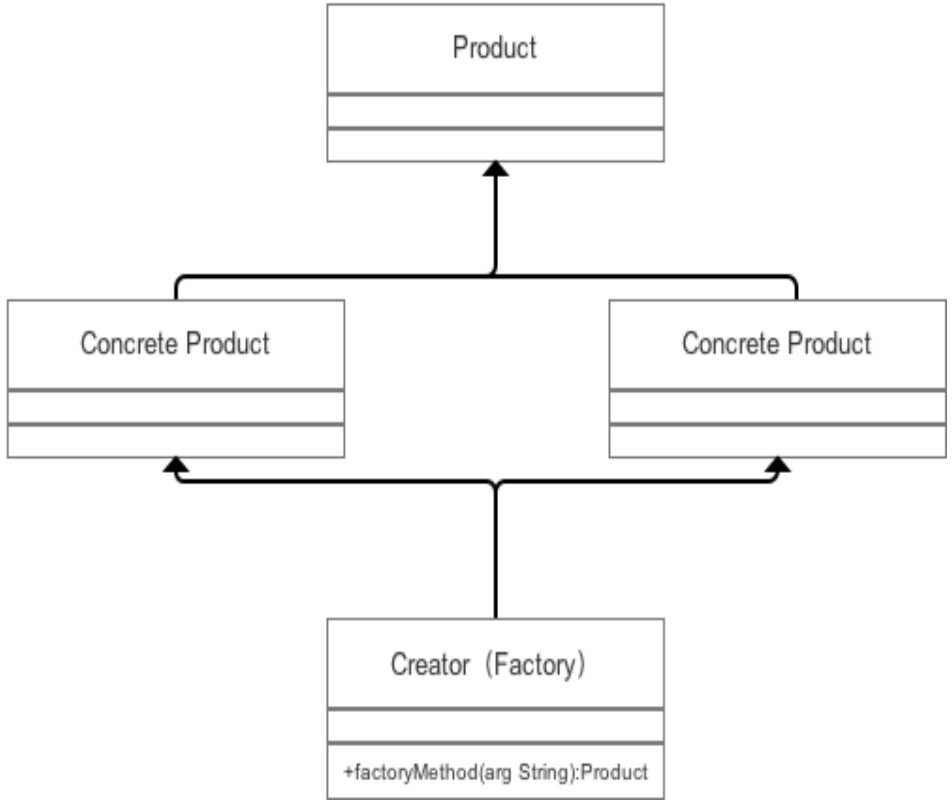
2. 用一个“工厂”对象来“**专职**”创建对象

- 工厂对象ProductFactory类负责产品创建，ProductControl使用工厂负责上层业务
  - 对产品对象进行业务处理，如销售等
- **问题根据职责分解**
  - ProductControl：管理和使用对象（Product）
  - ProductFactory：创建具体对象
- **对变化封闭，对扩展开放**
  - 已列出的具体产品类不必修改
  - 只需增加新的产品类和相应的构造方法
    - 实现Product接口



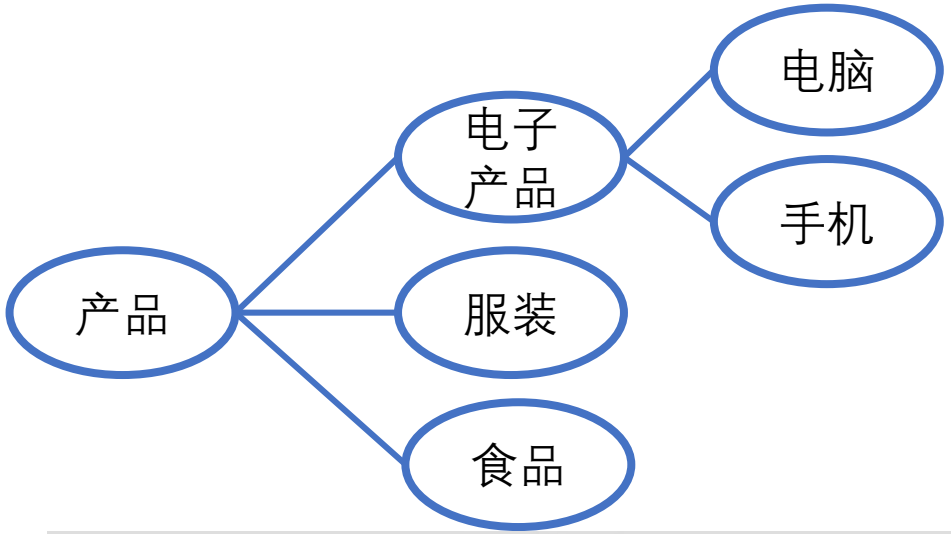
简单工厂模式

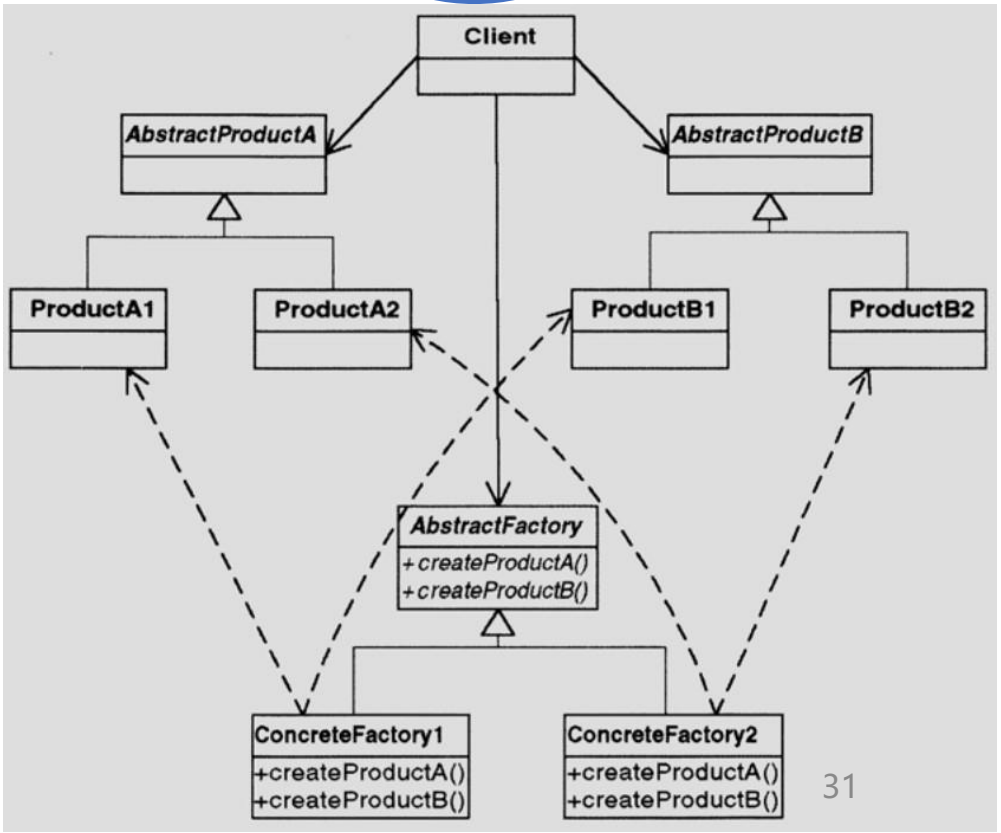
- 定义一个创建对象的接口（Creator）
  - 通过**传入的参数**决定创建哪一种实例对象
- 对Client对象隐藏（即**封装**）了“应该使用哪些类来创建对象”
- **适用情况**：客户仅需知道**需要传入工厂的参数**而**不关心具体的产品类别和实例化过程**



国际电子商务系统

- 产品类型增多或分类层次增加时?
- 简单工厂类难以实现产品族创建逻辑
  - 需要修改工厂类的方法逻辑，违背开闭原则
- 抽象工厂，定义具体工厂的公共接口，让具体工厂决定创建哪种具体产品





工厂模式的讨论

- 大多数情况下，产品可以静态枚举和表示
  - 实现Product的具体类
  - 对象创建还是调用其构造方法来完成
- 有些情况可能涉及一些算法才能完成对象内容的创建
  - 电子商务场景，关心客户下某个订单的商品浏览路径
    - 该商品浏览序列需要在订单创建时就建立
  - 从而针对不同的用户优化商品排列逻辑
  - SaleOrder的创建可以使用工厂模式
    - 可以区分出多种不同的订单：朋友推荐、自己搜索、广告引导等
  - 商品浏览路径的获得可以使用策略模式

工厂模式总结

- 策略模式和观察者模式都是尝试通过接口或基类来屏蔽具体细节的差异，client code可以进行统一处理，极大简化逻辑
  - 然而对象创建必须使用具体的类型
- 工厂模式：负责封装对象的创建逻辑
- 两种工厂模式
  - 简单工厂：通过传入参数决定创建的实例，适合创建少量类型的对象
  - 抽象工厂：适合创建一组相关或相互依赖的对象

作业内容介绍

- 冒险者可在商店购买或售卖物品
  - 商店有且仅有一个 => 建议使用**单例模式**
  - 明确在购买/售卖行为中冒险者类和商店类的职责
- 冒险者之间存在雇佣关系，当雇主冒险者进入战斗模式并退出后，被雇佣的冒险者们需要根据雇主的体力变化来进行援助。
  - 被雇佣者需要观察雇主体力状态 => **观察者模式**
  - 雇主是被观察者，被雇佣的冒险者是观察者，雇主体力满足特定条件是触发事件