# 设计模式

主讲人：章程

2019.1.10 & 2019.1.15

## 软件开发面临的问题

### 问题

#### 终端用户的视角

* 前提：满足功能性需求
* UI设计良好和功能需求的冲突
* 良好特性：安全性
* 终端用户可能是客户，但客户可能是需求方，不一定就是终端用户

#### 客户的视角

* 市场效应
* 成本和市场的关系
* 控制成本
* 开发周期

#### 开发者的视角

* 人员流动大
* 生产力问题

### 产生问题的原因

#### 质量问题

* 系统与业务战略冲突 —— 战略系统规划
* 组织文化被忽略 —— 用户参与、需求分析
* 系统分析不争气 —— 系统性方法
* 项目基于错误的原因执行 —— 复用（复用市场上已有的模块）

#### 生产率问题

* 用户改变想法 —— 管理
* 外部事件改变环境 —— 管理
* 实现不可行（直到启动项目才发现） —— 复用（复用市场上已有的模块）
* 糟糕的项目控制（项目经理缺乏经验） —— 培训

## 面向对象的技术、复用

#### 面向对象的技术

* 更高的抽象级别：抽象范围越来越广
* 模块化的软件：缩短开发周期、降低成本；方便维护，降低维护成本
* 软件复用：提交软件开发质量；降低开发成本（费用+开发周期）

#### 设计复用

* 偶然复用
  + 设计的偶然复用是在某些特定的应用领域开发软件的组织相当普遍。
  + 后期不可控，只是偶然性
* 必然复用
  + 从设计阶段开始有意识的使用复用技术
* 复用技术：
  + 库和工具包
    - 一组相关的可重用的程序集合
    - e.g 科学软件、GUI类库货工具包
    - 设计者负责整个产品的流程逻辑
  + 应用框架
    - 结合了设计的控制逻辑
    - 比重用工具包更能加快产品的开发
  + 设计模式
    - 通常的设计问题的解决方案：一组交互类的形式出现
    - 需要用户根据需要定制交互类
    - 设计模式不包含代码
    - 基于文档的形式体现，相当于是使用说明书
  + 体系结构
    - 包含多种复用技术：工具包、框架、设计模式

#### 面向对象的本质

* 数据和函数进行分离
* **封装稳定的抽象**
* **设计原则为指导**
* **设计模式为手段**

## 设计模式的发展历史、概念

#### 诞生与发展

* 起源于建筑业

#### 概念

* 每个模式描述环境中不断出现的问题，根据问题给出其解决方案
* 模式是经验的积累，随着时间也会出现相应问题，需要针对环境进行相应的调整
* << Design Patterns：Elements of Reusable Objrft-Oriented Software >>
* 软件开发的总体指导思路或者参照模版
* **successful solutions for recurring problems with high-level abstractions**：一套被反复使用、多数人知晓、经过分类编目、代码设计经验的总结

#### 定义与分类

* 基本要素
  + 模式名称
  + 问题
  + 解决方案
  + 效果
* 学习步骤
  + 问题提出
  + 问题分析
  + 解决方案——设计模式应用
* 分类
  + 根据行为：创建型、结构型、行为型
  + 类模式、对象模式
* 优点
  + 成功经验的总结，并以标准的形式展现
  + 更加简单的设计和复用
  + 提高设计的灵活度，易于修改，节约设计成本
  + 有助于初学者更加深入的理解面向对象的思想，提高软件的设计水平和代码质量

## OO设计原则（6个）+设计模式（23个）

#### 设计原则

##### 单一职责原则

* 有且只有一个原因引起类的变更，不能在一个类中封装过多操作
* e.g 基于角色的访问控制：用户管理、修改信息、增加机构、增加角色 -> 抽象出两个用户接口：收集和反馈用户信息、完成用户信息的维护和变更
* e.g 电话：拨通电话、通话、说话、挂断电话
  + 需要有协议管理和数据传送
* 优点
  + 类的复杂度降低
  + 可读性提高
  + 可维护性提高
  + 变更风险降低

##### 开闭原则

* 对扩展开放，对修改关闭

##### 里氏替换原则

* 主要和继承机制相关，主要让继承机制发挥最大作用
* 子类可以替换父类，但父类不一定能替换子类
* 子类必须实现父类的抽象方法，使用子类的时候使用父类的接口
* *子类不能修改父类实现的方法？*
* 子类可以增加方法
* 子类如果不能完整的实现父类的方法，则建议断开父子继承关系，采用依赖、聚集、组合等其他方式
* 覆盖或实现父类的方法等时候输入参数可以被放大
* 复写或实现的父类的方法时输出结果可以被缩小

##### 迪米特法则

* 不和陌生人说话
* 最少知识原则

##### 接口隔离原则

* 接口细化，接口中方法尽量少

##### 依赖倒置原则

* 高层不依赖底层
* 抽象不依赖细节
* 细节依抽象
* 总之：针对抽象进行编程
* 降低耦合，提高稳定性

#### 设计模式

##### 概述

* 高层次的定义

##### 访问者模式

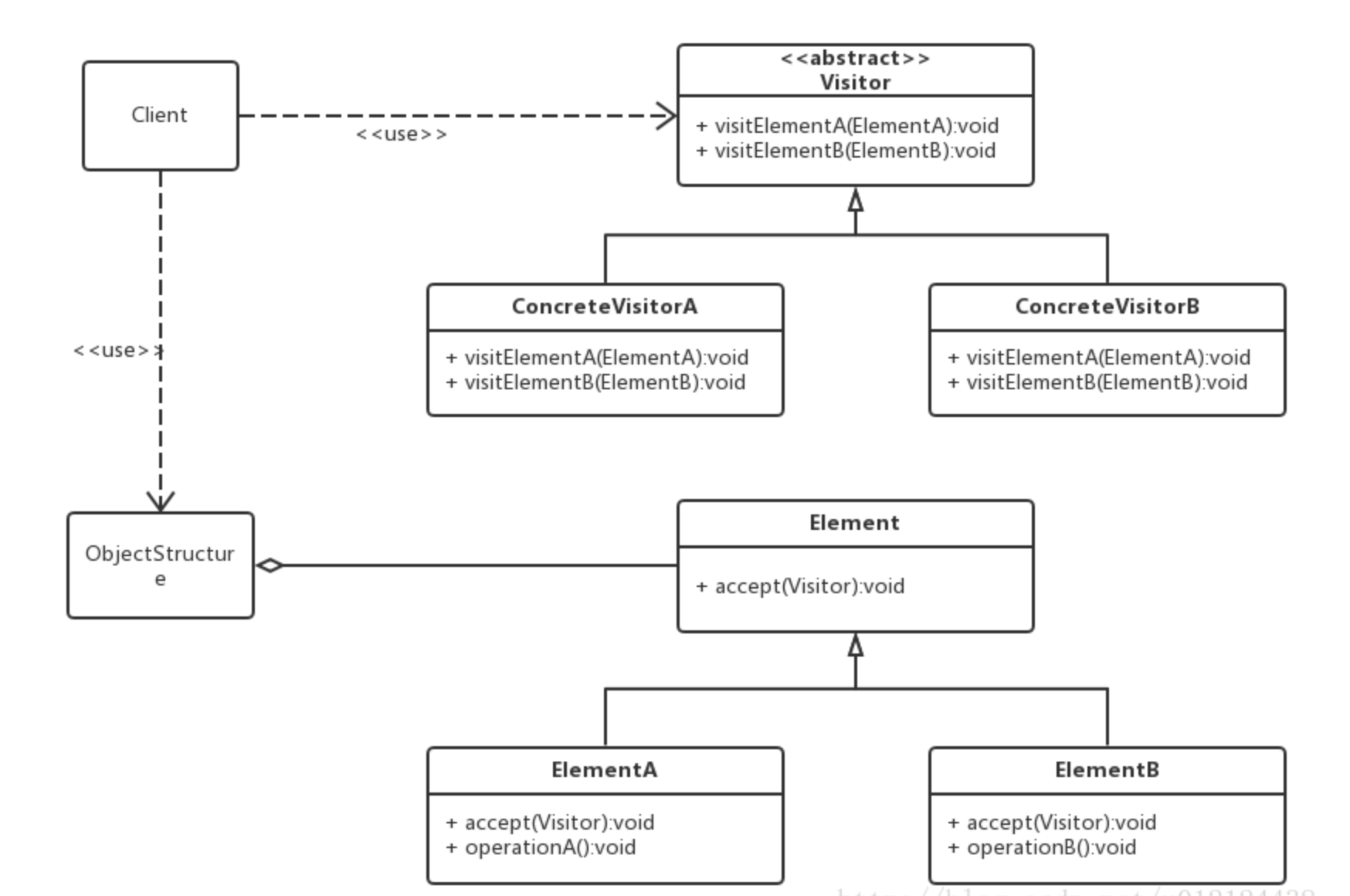
定义：表示一个作用与某对象结构中的各个元素的操作。不改变元素的前提下定义新的操作

* 对象结构中存储了多种不同类型的对象信息
* 对同一对象结构中元素的操作方式并不唯一，可能要提供给多种不同的处理方式
* 可能要增加新的处理方式
* 为每种类型对象提供一个解决方案

角色

* 抽象访问者：接口或者抽象类，它定义了对每一个元素（Element）访问的行为，它的参数就是可以访问的元素，它的方法数理论上来讲与元素个数是一样的，因此，访问者模式要求元素的类族要稳定，如果经常添加、移除元素类，必然会导致频繁地修改Visitor接口，如果这样则不适合使用访问者模式。
* 具体访问者：需要给出对每一个元素类访问时所产生的具体行为。
* 抽象元素：元素接口或者抽象类，它定义了一个接受访问者的方法（Accept），其意义是指每一个元素都要可以被访问者访问。
* 具体元素：具体的元素类，它提供接受访问方法的具体实现，而这个具体的实现，通常情况下是使用访问者提供的访问该元素类的方法。
* 对象结构：定义当中所说的对象结构，对象结构是一个抽象表述，它内部管理了元素集合，并且可以迭代这些元素供访问者访问。

UML图



实现

* 双重分派机制

体现原则

* 开闭原则
* **单一职责原则**

优点

e.g

* 部门为抽象接口

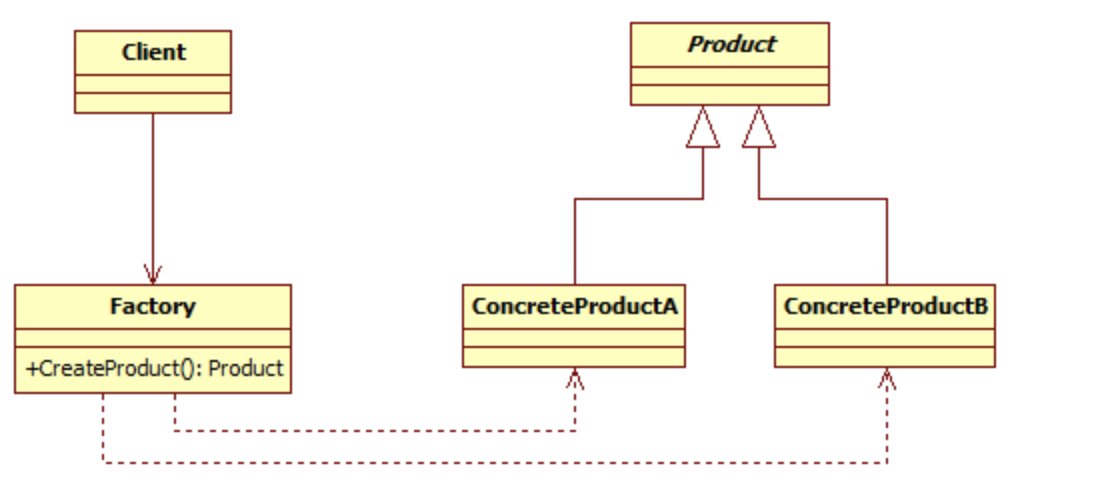
##### 简单工厂模式

* 不属于23个设计模式

角色

* 抽象产品
* 具体产品
* 构造者

UML图



优点

* 隔离类和对象之间的耦合

缺点

* 违反高内聚责任分配原则

##### 工厂方法模式

* 定义了用于创建对象接口，让子类决定用什么

角色

* 抽象产品
* 具体产品
* 构造者
* 具体构造者

优点

* 允许系统不修改具体工厂而引进新产品

##### 策略模式

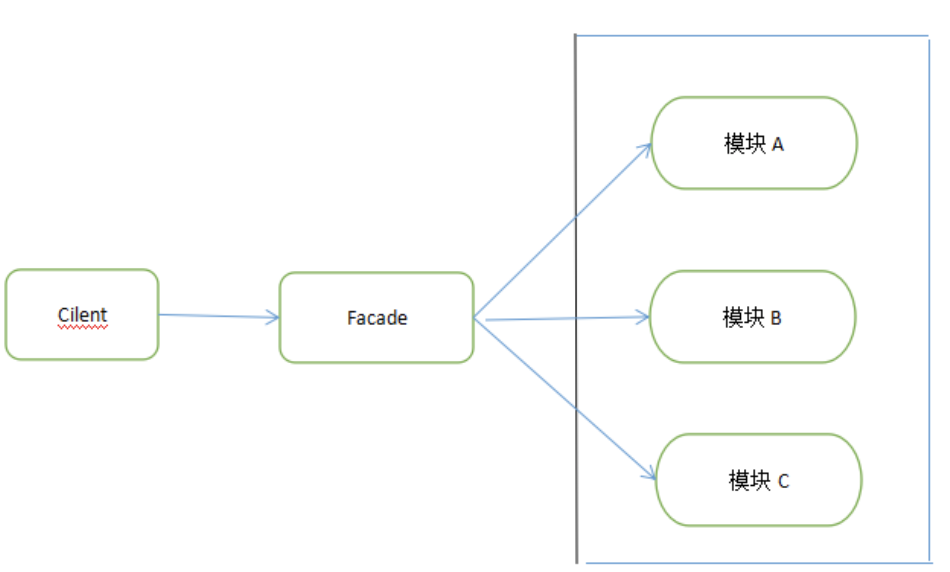
##### 外观模式（门面模式）

* 外观类负责多个子类的交互，解决交互关系
* 将客户端和子系统分隔开

角色

* 门面角色：外观模式的核心。它被客户角色调用，它熟悉子系统的功能。内部根据客户角色的需求预定了几种功能的组合。
* 子系统角色:实现了子系统的功能。它对客户角色和Facade时未知的。它内部可以有系统内的相互交互，也可以由供外界调用的接口。
* 客户角色:通过调用Facede来完成要实现的功能。

UML图



优点

* 降低耦合

## 注意事项

* 软件开发过程中维护占据了很大比例
* 面向对象基本原则：高内聚 低耦合
* 继承机制的优缺点
  + 优点
    - 代码共享，减少创建类的工作量
    - 提高代码可重用性
    - 子类形似父类，但又异于父类
    - 提高代码可扩展性
    - 提高产品或项目的开放性
  + 缺点
    - 继承是侵入性的，子类必须拥有父类的属性和方法
    - 降低代码灵活性，多了很多约束
    - 增强耦合，父类修改必须考虑子类的修改，大量代码重构