设计模式

- 5种构建型模式
  - 工厂模式
    - 为每个类对象建立工厂，由工厂创建对象，客户端只和工厂打交道
  - 抽象工厂模式
    - 为每个类工厂提取抽象接口，使得新增工厂、替换工厂变得容易
  - 单例模式
    - 全局使用一个对象，分为懒汉式和饿汉式，懒汉式有双重校验锁和内部类两种实现方式
  - 建造者模式
    - 创建构造过程稳定的对象，可以定义不同的配置（属性）
  - 原型模式
    - 为类定义clone()方法，使得创建相同对象更方便
- 7种结构型模式
  - 适配器模式
    - 由于有相关性但不兼容的接口
    - 将一个类的接口转换成客户希望的另一个接口，使得原本由于接口不兼容的类能一起工作。A、B类本来不可以一起使用，通过C进行适配，使得A、B可以一起工作
  - 代理模式
    - 静态代理
      - 给一个对象提供一个代理，由代理对象控制原对象的引用
      - 作用：打印日志，权限管理
      - 装饰器模式为了增强功能，代理模式则为了加以控制
    - 动态代理
      - 将多个方法统一为一个，根据调用方法名判断是否加以控制，相比于静态代理更省代码量
  - 装饰器模式
    - 增强一个类原有的功能（仅在原方法上改变，不增加新方法），透明装饰模式，可无限修饰
    - 为一个类添加新功能（不修改原方法，新增方法），半透明模式，无法多次装饰
    - 容易造成程序中有大量相似的类
  - 桥接模式
    - 用于同等级的接口相互结合
    - 将抽象部分与它的实现部分分离，使得他们可以独立变化。就是一个对象有多种分类方式，并且各个方式容易变化，将各个分类方式分类出来，在组合即可。（类型与颜色，应用于多个同等级的接口）
  - 组合模式
    - 用于整体和部分的结构，（他们有相似的结构）不同于桥接模式（文件夹与子文件夹）
    - 有安全方式（接口方法少，仅有实现类中公有的方法）和透明模式（接口方法多，部分子类需要实现空方法）
  - 外观模式
    - 将多个子系统封装起来，提供简洁的接口供外部使用（中间件思想）
  - 享元模式
    - 共享对象（对象本身不同，通过一点点变化后即可复用），提高复用性，轻量级模式
- 11种行为型模式
  - 责任链模式
    - 用于处理职责相同，程度不同的类，从程度最低的类开始处理，处理不了则传递给上层梳理（低级程序员到高级程序员处理bug）
    - 好处：降低对象间耦合度、扩展性强、灵活性强、简化了队形之间的连接、责任分层
  - 命令模式
    - 将一个请求封装成一个对象，可以用不同的请求对客户端进行参数化，对请求排队或记录请求日志，以及可撤销。可以使用宏命令，一次实现多步控制。
    - 降低系统耦合度，扩展性强，封装"方法调用"，灵活性强，缺点是会产生大量命令类
  - 解释器模式
    - 给定一门语言及其文法表示，同时定义一个解释器使用该表示来解释语言。例如：将依据中文公式解释给计算机，计算机运行出正确的结果
  - 迭代器模式
    - 提供一种方法访问一个容器对象中的各个元素，而又不暴露该对象的内部细节（for-each是迭代器的一种应用）
  - 中介者模式
    - 对于多个类呈现网状关系时，引入中介者可以使他们都与中介者交互，变成星型。（微信群转账）
    - 将类与类之间的多对多关系简化成一对多，多对一关系
  - 备忘录模式
    - 在不破坏封装的条件下，通过备忘录对象存储另一个对象内部状态的快照，在合适的时候将这个对象还原到存储起来的状态，读档、存档功能
    - 提供给用户可以恢复状态的机制，实现了信息的封装，缺点：消耗资源
  - 观察者模式
    - 处理一对多的依赖关系，一个被观察者有多个观察者，当一个对象的状态发生变化时，所有依赖于它的对象都得到通知，并被自动更新（一个小偷被多个警察观察）
  - 状态模式
    - 一个对象的内在状态改变时，其行为也改变了，像是变成了另一个类（普通用户与会员）
  - 策略模式
    - 定义一系列算法，并将每个算法封装，他们可以互相替换，让算法独立于使用他的用户独立变化（最好与工厂模式结合，工厂封装各种策略）
  - 模板方法模式
    - 定义操作中的算法骨架，将部分步骤延迟到子类，子类可以不改变算法的结构即可以重定义该算法的特定步骤，即为继承关系
  - 访问者模式
    - 顾客和餐厅，在顾客中实现是否需要某食物的方法，而不由餐厅实现
- 设计原则
  - 开闭原则（OCP）
    - 对扩展开放，对修改关闭，实现热插拔效果（接口和抽象类）
  - 里氏替换原则（LSP）
    - 任何父类可以出现的地方，子类一定可以出现，只扩展新功能，而不破坏父类原有功能
    - 开闭原则的补充，继承复用的基石
  - 依赖倒转原则（DIP）
    - 依赖于抽象而不是具体（针对接口编程）
  - 接口隔离原则（ISP）
    - 使用多个隔离的接口，而不用单个接口。降低依赖，降低耦合
  - 迪米特法则（最少知道原则，DP）
    - 减少相互作用，使系统功能模块相对独立
  - 合成复用原则（CRP）
    - 尽量使用合成、聚合，而不是继承