1. 从招式与内功谈起——设计模式概述

模式是在特定环境下人们解决某类重复出现问题的一套成功或有效的解决方案。

软件模式是指在软件开发过程中某些可重现问题的有效解决方法，包含架构模式、分析模式、过程模式和设计模式等。

设计模式就是代码设计经验总结，使用设计模式的原因包括编写可重用的代码和提高代码的可靠性等。

设计模式可以分为三种：

1. 创建型
2. 结构型
3. 行为型

滥用模式还不如不用模式

1. 预备知识——UML类图与面向对象设计原则

UML类图用于描述每一个设计模式的结构

常用的面向对象设计原则包括7个，分别是单一职责原则、开闭原则、里氏替换原则、依赖倒转原则、借口隔离原则、合成复用原则和迪米特法则。

UML是一种由图形符号表达的建模语言，其结构主要包括以下几个部分：

1. 视图
   1. UML视图包括用户视图、结构视图、行为视图、实现视图和环境视图
2. 图
   1. 最新的UML2.0提供了13中图，分别是用例图、类图、对象图、包图、组合结构图、状态图、活动图、顺序图、通信图、定时图、交互概览图、组件图和部署图。
3. 模型元素
4. 通用机制

类之间的关系：

1. 关联关系
   1. 在UML类图中，用实线连接有关联关系对象所对应的类。

面向对象的设计原则之一在于支持可维护性复用：一方面需要实现设计方案或者源代码的重用；另一方面需要确保系统能够易于扩展和修改，具有较好的灵活性。

最常见的7种面向对象设计原则：

|  |  |
| --- | --- |
| 原则名称 | 描述 |
| 单一职责原则 | 一个类只负责和一个功能领域中的相应职责 |
| 开闭原则 | 软件实体应对扩展开放，而对修改关闭 |
| 里氏替换原则 | 所有引用基类对象的地方能够透明的使用其子类对象 |
| 依赖倒置原则 | 抽象不应该依赖于细节，细节应该以抽象 |
| 接口隔离原则 | 使用多个专门的接口而不使用单一的总接口 |
| 合成复用原则 | 尽量使用对象组合而不是继承来达到复用的目的 |
| 迪米特法则 | 一个软件实体应当尽可能少的与其他实体发生相互作用 |

1. 单一职责原则：

用于控制类的粒度的大小

实现高内聚低耦合的直到方针

1. 开闭原则

一个软件对象的实体应该对扩展开放对修改关闭，即软件实体应该在不修改原有代码的情况下进行扩展。

1. 里氏替换原则

通俗的说就是所有引用基类（父类）的地方必须能透明的使用其子类的对象。

里氏替换原则表明，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何的错误和异常（这里不讨论功能）。

这一原则要求尽量使用基类类型对对象进行定义（me：实际就是尽量使用多态，也即面向接口编程），将父类设计为抽象类或者接口。

1. 依赖倒置原则

该原则指的是抽象不应该依赖于细节，细节应该依赖于抽象，即要针对接口编程，而不是针对实现编程。

该原则要求尽量使用层次较高的抽象层类，进行变量声明、参数类型声明、方法返回类型声明和数据类型的转换等。为了能够使用父类或者接口调用实现类的方法，具体的实现类中不要定义其他的非实现自接口或者继承自基类的公开方法。

常用的恶依赖注入方式有三种：

构造注入、设置注入（setter方法）和接口注入（指的是通过实现在接口中声明的业务方法来传入具体的对象）。这些方法在使用时传入的都是抽象类型，实际调用时传递具体的类型。

5． 接口隔离原则

客户端不应该依赖那些它不需要的接口

使用多个专门的接口而不是一个总的大的接口。

接口仅仅通过给客户端（client）它需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

接口应该尽量细化，每个接口中的方法应该尽量少。

1. 合成复用原则

尽量使用对象服务而不是继承来达到复用的目的。

继承复用会破坏系统的封装性

1. 迪米特法则

又称为最小只是原则，一个软件实体应该可能少的与其他实体发生作用

第2 部分 创建的艺术——创建型模式

单例模式 用于创建那些在软件系统中独一无二的对象。

浪费资源，包括CPU资源和内存资源

实现步骤：

1. 私有化构造器

可见性只对类外有效，在类的内部任何都会可见的

1. 私有的类实例字段/域
2. 公有的静态方法返回该字段/域

*/\*\*  
 \* 设计步骤：  
 \* 1. 私有化构造器  
 \* 2. 私有的该类的静态成员变量  
 \* 3. 公共的外部访问该成员变量的静态方法  
 \*  
 \* 两种构造方式：  
 \* 1. 懒汉式  
 \* 2. 饿汉式  
 \*/  
  
/\*\*  
 \* 懒汉式  
 \*/*public class Singleton {  
 private static Singleton *instance*;  
  
 private Singleton(){}  
  
 public static Singleton getInstance(){  
 if(null==*instance*){  
 synchronized (Singleton.class){  
 if(null==*instance*){  
 *instance*=new Singleton();  
 }  
 }  
 }  
  
 return *instance*;  
 }  
}  
  
*/\*\*  
 \* 饿汉式  
 \*/*class Singleton2{  
 private static Singleton2 *instance*=new Singleton2();  
 private Singleton2(){}  
  
 public static Singleton2 getInstance(){  
 return *instance*;  
 }  
}