设计模式

设计模式的目的:

- 代码具有更好的重用性
- 可读性
- 可扩展性
- 可靠性
- 高内聚,低耦合的特性

设计模式常用的七大原则有

1. 单一职责原则
2. 接口隔离原则
3. 依赖倒转原则
4. 里氏替换原则
5. 开闭原则
6. 迪米特法则
7. 合成复用原则

单一职责

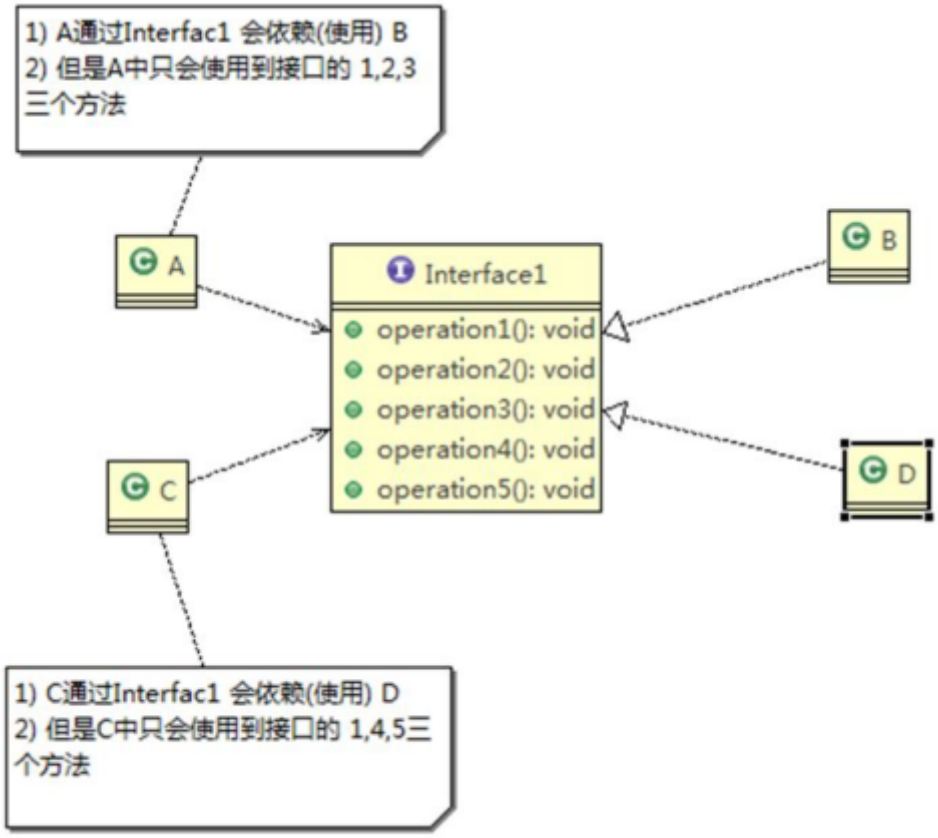
基本介绍:一一个类只负责完成一个职责或者功能。不要设计大而全的类，要设计粒度小、功能单一的类。单一职责原则是为了实现代码高内聚、低耦合，提高代码的复用性、可读性、可维护性

注意事项:

- 降低类的复杂度,一个类只负责一项职责
- 提高类的可读性,可维护性
- 降低变更引起的风险
- 通常情况下,应该遵守单一职责原则,除非是逻辑足够简单,才可以违背单一职责原则,只有类中方法足够少,可以在方法级别保持单一职责原则

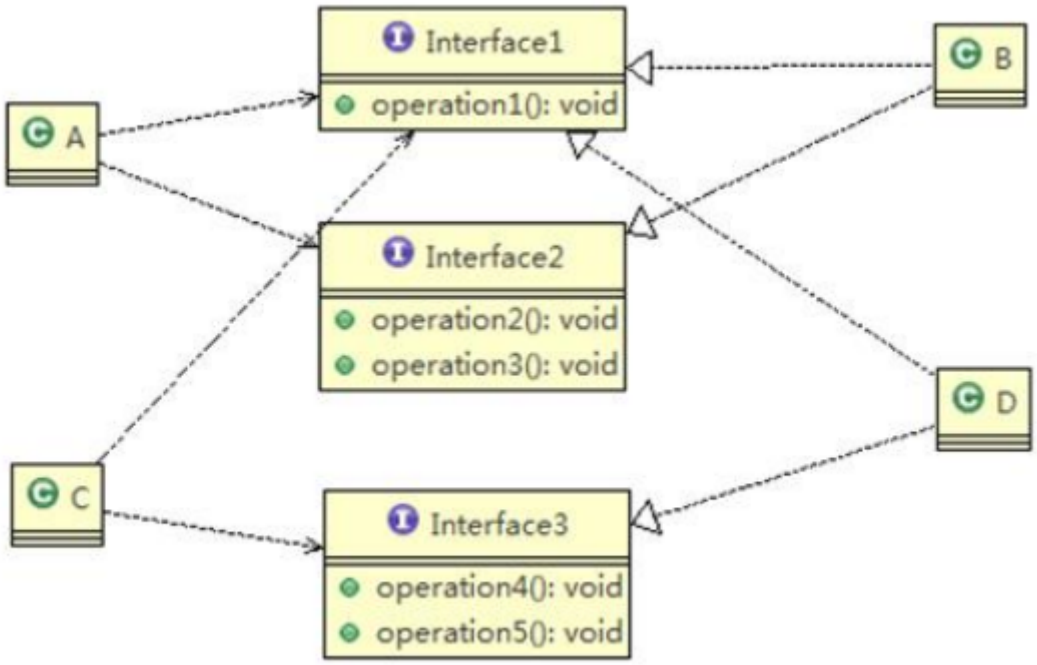
接口隔离原则

基本介绍:客户端不应该依赖它不需要的接口,即一个类中对另一个类的依赖应该建立在最小的接口上



如图:这样A和C都会分别多实现自身并不需要使用的方法,不满足接口隔离原则,应该需要将Interface拆分为多个接口,A和C分别与只需要满足自身需要的接口建立关系,以满足接口隔离原则.

拆解之后UML图:



依赖倒转原则

基本介绍:

- 高层模块不应该依赖底层模块,二者都应该依赖其抽象
- 抽象不应该依赖细节,细节应该依赖抽象
- 依赖倒转的中心思想是面向接口编程

- 依赖倒转原则是基于:相对于细节的多变性,抽象的东西要稳定的多,以抽象为基础搭建的架构比细节为基础的架构要稳定的多.
- 使用接口或抽象类的目的是制定好规范,而不涉及任何具体的操作,把展现细节的任务交给他们的实现类去完成

注意事项:

- 低层模块尽量要有抽象类或者接口
- 变量的声明类型尽量是抽象类或接口,这样我们的变量引用和实际对象间就存在一个缓冲层,利于程序扩展和优化
- 继承时遵循里氏替换原则

里氏替换原则

面向对象中一定会用到继承:

- 如果对每个类型为T1的对象o1,都有类型为T2的对象o2,使得以T1定义的所有程序P在所有的对象o1都代换到o2时,程序P的行为没有发生变化,那么类型T2是类型T1的子类型,**所有引用基类的地方必须能透明地使用其子类的对象**
- 在子类中尽量不要重写父类的方法
- 继承实际上让两个类耦合性增强了,在适当的情况下,可以通过聚合,组合,依赖来解决问题

这样会导致如下问题:

```
public class LiskovDemoA {
    public static void main(String[] args) {
        A a = new A();
        System.out.println(a.func1(2,1));
        A b = new B();
        System.out.println(b.func1(2,1));
    }
}
class A{
    protected int func1(int num1,int num2){
        return num1 - num2;
    }
}
class B extends A{
    @Override
    public int func1(int num1,int num2){
        return num1 + num2;
    }
}
```

```
1
3
```

B破坏了父类A的方法,导致不符合预期目标

解决办法: 需要将更加基础的方法和成员重新声明一个Base类,A和B都继承Base类,这样就去除了原有的继承,可以使用聚合,组合,依赖来解决具体问题.在B中声明一个A的实例,可以达到组合的效果,可以用到A的相关方法,并且降低了原先代码的耦合度.

```
class Base{

}
```

```
class A extends Base{
    protected int func1(int num1,int num2){
        return num1 - num2;
    }
}
class B extends Base{
    private A a = new A();
    public int func1(int num1,int num2){
        return num1 + num2;
    }
}
```

开闭原则

基本介绍:

- 开闭原则是编程中最基础,最重要的设计原则
- 一个软件实体如类,模块和函数应该对**扩展开放(对提供方),对修改关闭(使用方)**,用抽象创建框架,用实现扩展细节
- 当软件需要变化时,尽量通过扩展软件实体的行为来实现变化,而不是通过修改以后的代码来实现变化
- 编程中遵循其他原则,以及使用设计模式的目的就是遵循开闭原则

迪米特法则

基本介绍:

- 一个对象应该对其他对象保持最少的了解
- 类与类关系越密切,耦合度越大
- 迪米特法则又称为**最少知道原则**,即一个类对自己依赖的类知道的越少越好,被依赖的类尽量将逻辑封装在类的内部,对外除了提供public方法,不对外泄露任何信息
- 迪米特法则还有个更简单的定义:只与直接的朋友通信
- **直接的朋友**:每个对象都会与其他对象有耦合关系,只要两个对象之间有耦合关系,我们就称为这两个对象之间是朋友关系,耦合的方式很多,依赖,关联,组合,聚合等.我们称出现在成员变量,方法参数,方法返回值中的类为直接的朋友,**而出现在局部变量中的类不是直接的朋友**.陌生的类最好不要以局部变量的形式出现在类的内部

注意细节:

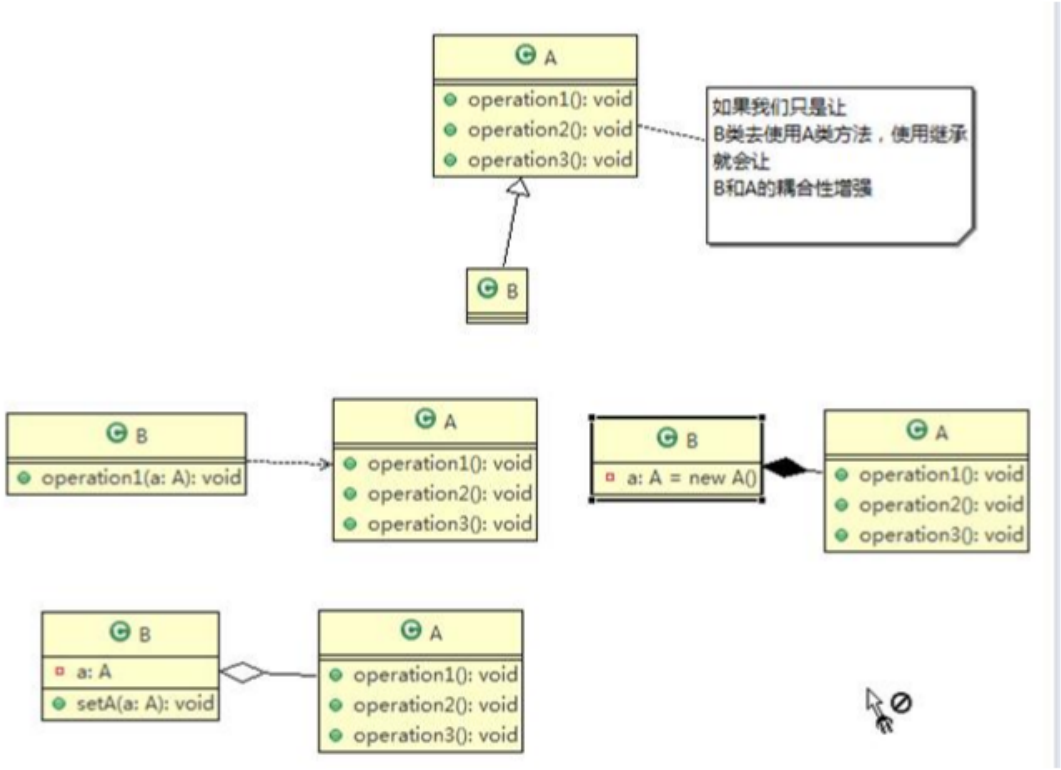
- 迪米特法则的核心是**降低**类之间的耦合,并不是要求完全没有依赖关系

合成复用原则

基本介绍:

- 尽量使用合成/聚合的方式,而不是使用继承

原则是尽量使用合成/聚合的方式，而不是使用继承



总结

设计原则的核心思想:

- 找出应用中可能需要变化的地方,把它们独立出来,不要和那些不需要变化的代码混在一起
- 针对接口编程,而不是针对实现编程
- 为了交互对象之间的松耦合设计而努力

UML介绍

UML 统一建模语言中定义了六种类之间的关系。它们分别是：泛化、实现、关联、聚合、组合、依赖

泛化:泛化（Generalization）可以简单理解为继承关系。**泛化用空心三角形 + 实线来表示**.具体到 Java 代码就是下面这样：

```
public class A { ... }
public class B extends A { ... }
```

实现:实现（Realization）一般是指接口和实现类之间的关系。**实现接口用空心三角形 + 虚线**.具体到 Java 代码就是下面这样：

```
public interface A {...}
public class B implements A { ... }
```

聚合:聚合（Aggregation）是一种包含关系，A 类对象包含 B 类对象，B 类对象的生命周期可以不依赖 A 类对象的生命周期，也就是说可以单独销毁 A 类对象而不影响 B 对象，比如课程与学生之间的关系。**聚合关系用空心的菱形 + 实线箭头来表示**.具体到 Java 代码就是下面这样：

```
public class A {
private B b;
public A(B b) {
this.b = b;
}
}
```

组合:组合（Composition）也是一种包含关系。A 类对象包含 B 类对象，B 类对象的生命周期跟依赖 A 类对象的生命周期，B 类对象不可单独存在，比如鸟与翅膀之间的关系。**组合关系用实心的菱形 + 实线箭头**.具体到ava 代码就是下面这样：

```
public class A {
private B b;
public A() {
this.b = new B();
}
}
```

关联:关联（Association）是一种非常弱的关系，包含聚合、组合两种关系。具体到代码层面，如果 B 类对象是 A 类的成员变量，那 B 类和 A 类就是关联关系。**关联关系用实线箭头来表示。**具体到 Java 代码就是下面这样：

```
public class A {
private B b;
public A(B b) {
this.b = b;
}
}
或者
public class A {
private B b;
public A() {
this.b = new B();
}
}
```

依赖:依赖（Dependency）是一种比关联关系更加弱的关系，包含关联关系。不管是 B 类对象是 A 类对象的成员变量，还是 A 类的方法使用 B 类对象作为参数或者返回值、局部变量，只要 B 类对象和 A 类对象有任何使用关系，我们都称它们有依赖关系。**依赖用虚线箭头来表示**.具体到 Java 代码就是下面这样：

```
public class A {
private B b;
public A(B b) {
this.b = b;
}
}
或者
public class A {
private B b;
public A() {
this.b = new B();
}
}
```

或者

```
public class A {
public void func(B b) { ... }
}
```