六大原则

单一职责原则

- **核心思想：**应该有且仅有一个原因引起类的变更
- **问题描述：**假如有类 Class1 完成职责 T1，T2，当职责 T1 或 T2 有变更需要修改时，有可能影响到该类的另外一个职责正常工作。
- **好处：**类的复杂度降低、可读性提高、可维护性提高、扩展性提高、降低了变更引起的风险。
- **需注意：** 单一职责原则提出了一个编写程序的标准，用“职责”或“变化原因”来衡量接口或类设计得是否优良，但是“职责”和“变化原因”都是不可以度量的，因项目和环境而异。

里斯替换原则

- **核心思想：**在使用基类的的地方可以任意使用其子类，能保证子类完美替换基类。
- **通俗来讲：**只要父类能出现的地方子类就能出现。反之，父类则未必能胜任。
- **好处：**增强程序的健壮性，即使增加了子类，原有的子类还可以继续运行。
- **需注意：**如果子类不能完整地实现父类的方法，或者父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系 采用依赖、聚合、组合等关系代替继承。

依赖倒置原则

- **核心思想：**高层模块不应该依赖底层模块，二者都该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象；
- **说明：**高层模块就是调用端，低层模块就是具体实现类。抽象就是指接口或抽象类。细节就是实现类。
- **通俗来讲：**依赖倒置原则的本质就是通过抽象（接口或抽象类）使个各类或模块的实现彼此独立，互不影响，实现模块间的松耦合。
- **问题描述：**类 A 直接依赖类 B，假如要将类 A 改为依赖类 C，则必须通过修改类 A 的代码来达成。这种场景下，类 A 一般是高层模块，负责复杂的业务逻辑；类 B 和类 C 是低层模块，负责基本的原子操作；假如修改类 A，会给程序带来不必要的风险。
- **解决方案：**将类 A 修改为依赖接口 interface，类 B 和类 C 各自实现接口 interface，类 A 通过接口 interface 间接与类 B 或者类 C 发生联系，则会大大降低修改类 A 的几率。
- **好处：**依赖倒置的好处在小型项目中很难体现出来。但在大中型项目中可以减少需求变化引起的工作量。使并行开发更友好。

接口隔离原则

- **核心思想**：类间的依赖关系应该建立在最小的接口上
- **通俗来讲**：建立单一接口，不要建立庞大臃肿的接口，尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- **问题描述**：类 A 通过接口 interface 依赖类 B，类 C 通过接口 interface 依赖类 D，如果接口 interface 对于类 A 和类 B 来说不是最小接口，则类 B 和类 D 必须去实现他们不需要的方法。
- **需注意**：

    - **接口尽量小，但是要有限度**。对接口进行细化可以提高程序设计灵活性，但是如果过小，则会造成接口数量过多，使设计复杂化。所以一定要适度
    - **提高内聚，减少对外交互**。使接口用最少的方法去完成最多的事情
    - **为依赖接口的类定制服务**。只暴露给调用的类它需要的方法，它不需要的方法则隐藏起来。只有专注地为一个模块提供定制服务，才能建立最小的依赖关系。

迪米特原则

- **核心思想**：类间解耦。
- **通俗来讲**： 一个类对自己依赖的类知道的越少越好。自从我们接触编程开始，就知道了软件编程的总的原则：低耦合，高内聚。无论是面向过程编程还是面向对象编程，只有使各个模块之间的耦合尽量的低，才能提高代码的复用率。低耦合的优点不言而喻，但是怎么样编程才能做到低耦合呢？那正是迪米特法则要去完成的。

开放闭合原则

- **核心思想**：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化
- **通俗来讲**：一个软件产品在生命周期内，都会发生变化，既然变化是一个既定的事实，我们就应该在设计的时候尽量适应这些变化，以提高项目的稳定性和灵活性。

一句话概括:单一职责原则告诉我们实现类要职责单一；里氏替换原则告诉我们不要破坏继承体系；依赖倒置原则告诉我们要面向接口编程；接口隔离原则告诉我们在设计接口的时候要精简单一；迪米特法则告诉我们要降低耦合。而开闭原则是总纲，他告诉我们要对扩展开

放，对修改关闭。

设计模式

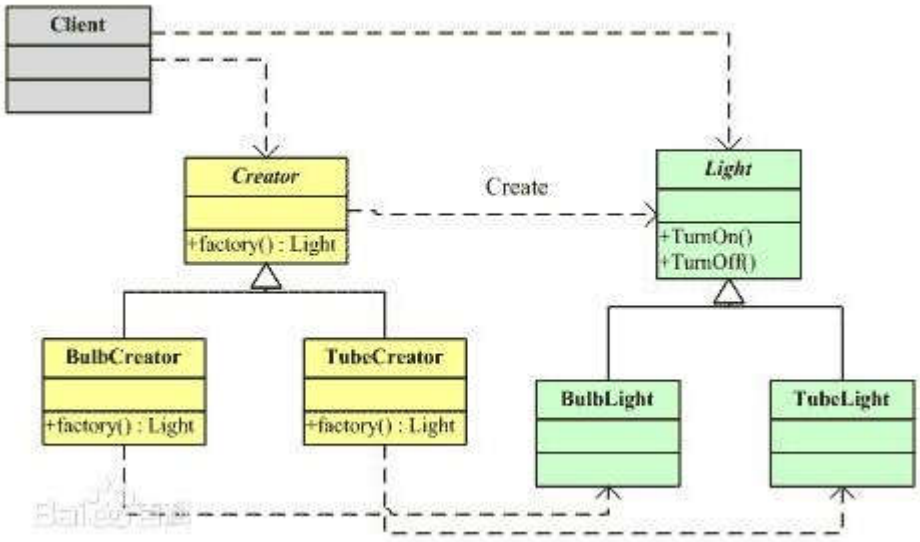
1 单例模式

确保一个类只有一个实例，自行初始化并向整个系统提供这个实例。

2 工厂方法

工厂方法(Factory Method)模式的意义是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类当中。核心工厂类不再负责产品的创建，这样核心类成为一个抽象工厂角色，仅负责具体工厂子类必须实现的接口，这样进一步抽象化的好处是使得工厂方法模式可以使系统在不修改具体工厂角色的情况下引进新的产品。

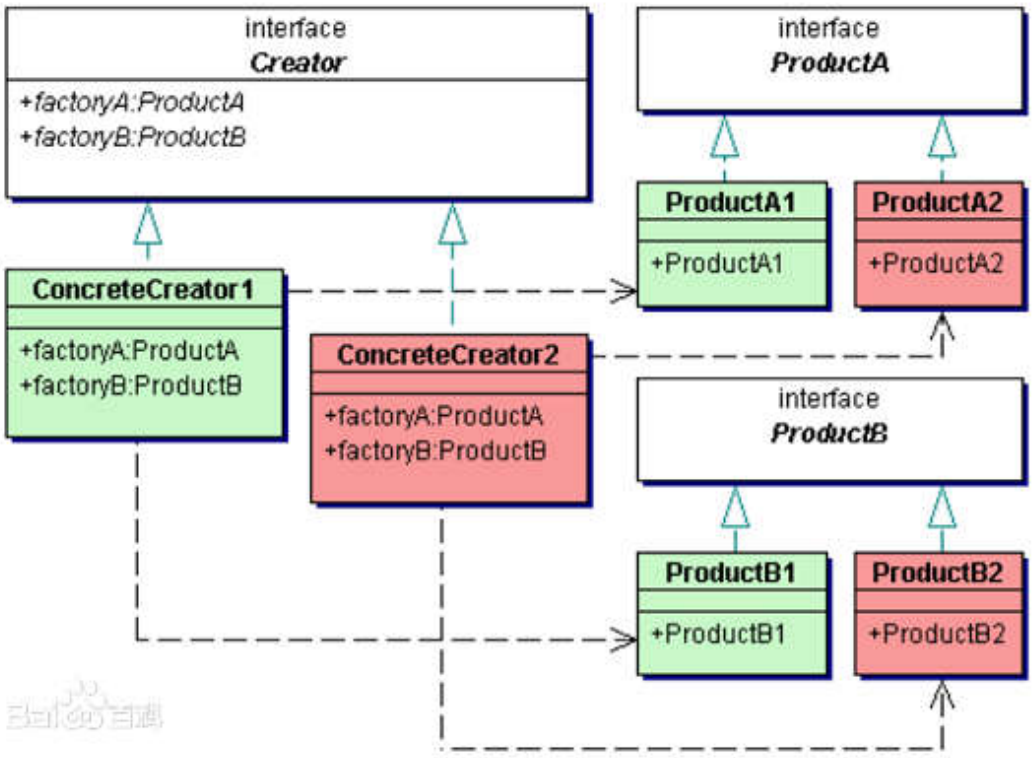
UML 类图：



3 抽象工厂

为创建一组相关或相互依赖的对象提供一个接口，而且无须指定它们的具体类。

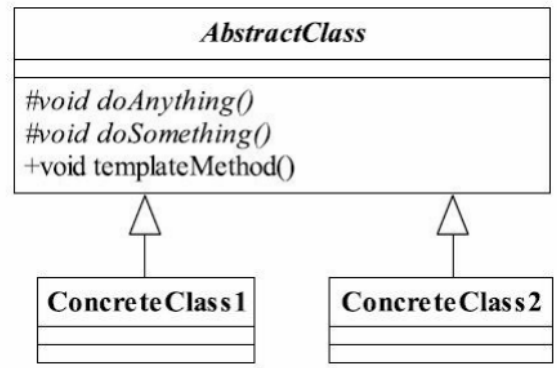
UML 类图：



4 模板方法

定义一个操作中算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法即可重定义该算法的某些特定步骤。

UML 类图：



- **基本方法**：也叫作基本操作，是由子类实现的方法，并且在模板方法被调用
- **模板方法**：可以有一个或几个， 一般是一个具体方法， 也就是一个框架， 实现对基本方法的调度，完成固定的逻辑。一般将模板方法加上 final 关键字，不允许被覆写

5 建造者模式

建造者模式也叫作生成器模式，将一个复杂对象的构建和它的表示分离，使得同样的构建过程可以创建不同的表示。

UML 类图：

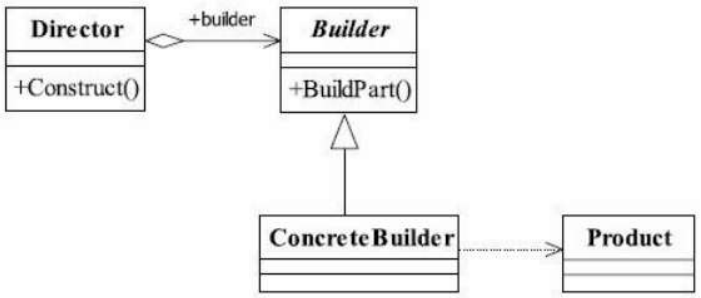


图11-4 建造者模式通用类图

6 代理模式

代理模式是一种使用率非常高的模式，为其他对象提供一种代理以控制对这个对象的访问，

UML 类图：

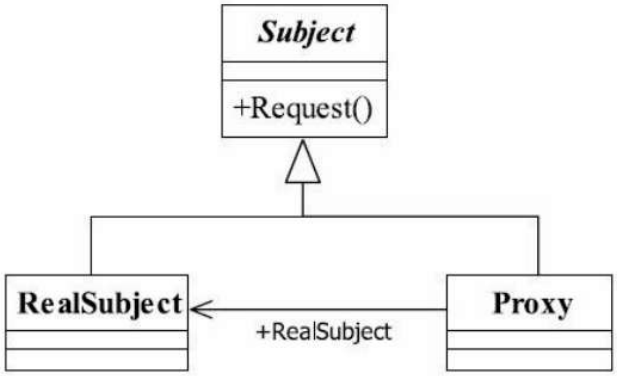


图12-3 代理模式的通用类图

动态代理

动态代理是实现阶段不用关心代理谁，而在运行阶段才指定代理哪一个对象。

UML 类图：

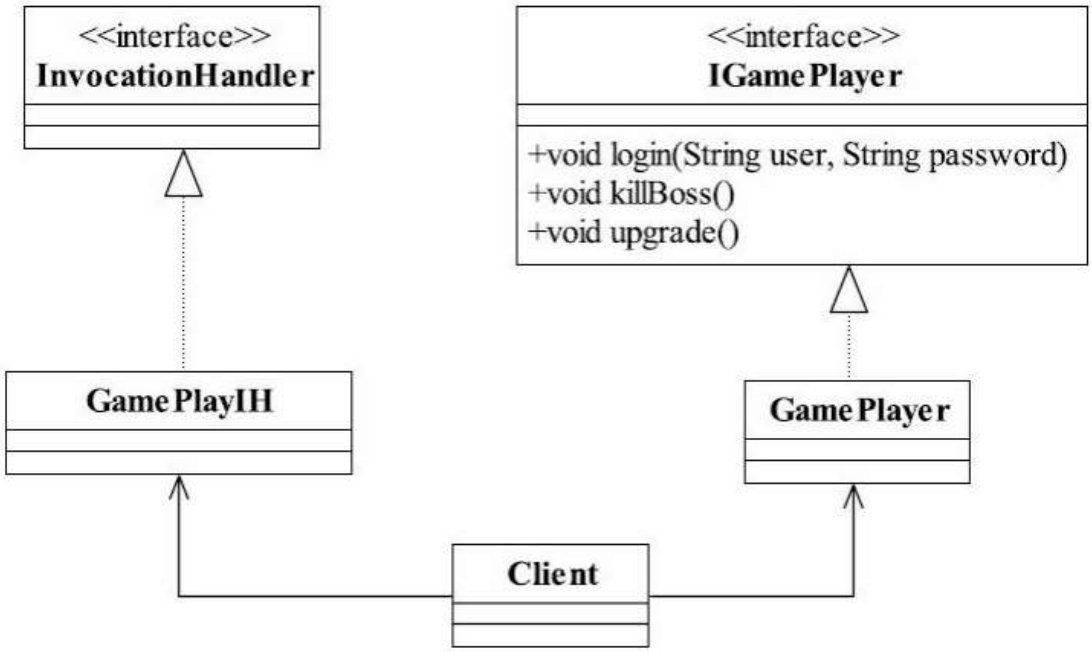


图12-7 动态代理

7 原型模式

原型模式的定义为：用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。原型模式的核心是一个 clone()方法，通过该方法进行对象的复制，Java 提供了一个 Cloneable 来标示这个对象是可拷贝的，为什么说是“标示”呢？这个接口只是一个标记作用，在 JVM 中具有这个标记的对象才有可能被拷贝。那怎么才能从“有可能被拷贝”到“可以被拷贝”呢？方法就是**覆盖 clone()方法**。

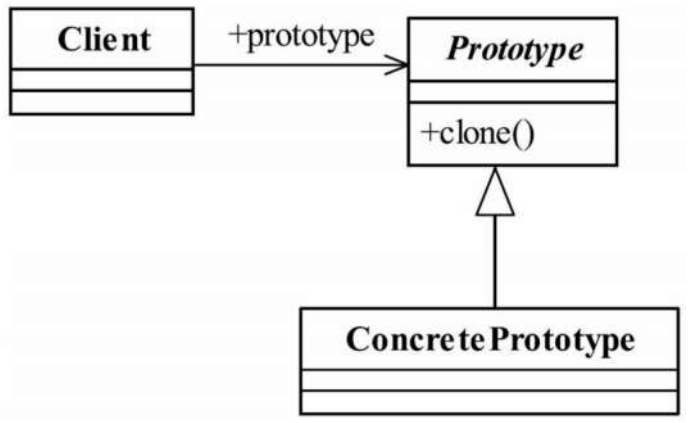


图13-3 原型模式的通用类图

8 中介者模式

用一个中介对象封装一系列的对象交互，中介者使各对象不需要显示地互相作用，从而

使其耦合松散，而且可以独立地改变它们之间的交互。

UML 类图：

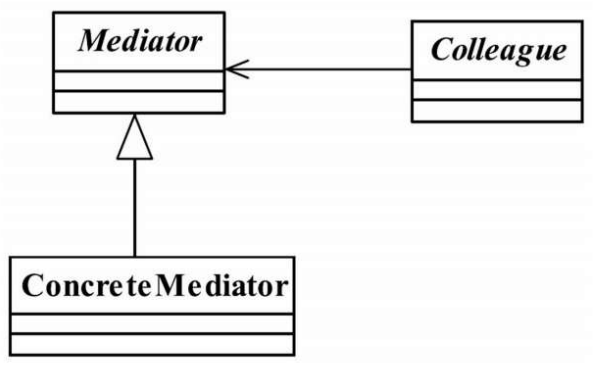


图14-7 中介者模式通用类图

- **终结者模式的优点**：减少类间的依赖，把原来的一对多的依赖变成了一对一的依赖，同事类只依赖中介者，较少了依赖，降低了类间的耦合。
- **终结者模式的缺点**：终结者会膨胀的很大，而且逻辑很复杂，原来 N 个对象直接的相互依赖关系转换为中介者和同事类的依赖关系，同事类越多，中介者的逻辑越复杂。