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

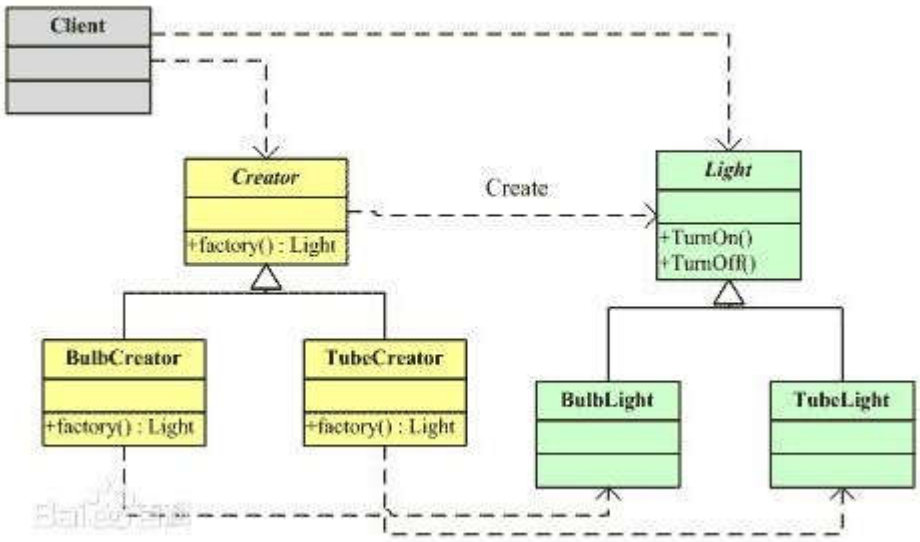
1 单例模式

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。
https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Singleton-%E5%8D%95%E4%BE%8B%E6%A8%A1%E5%BC%8F.java

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：

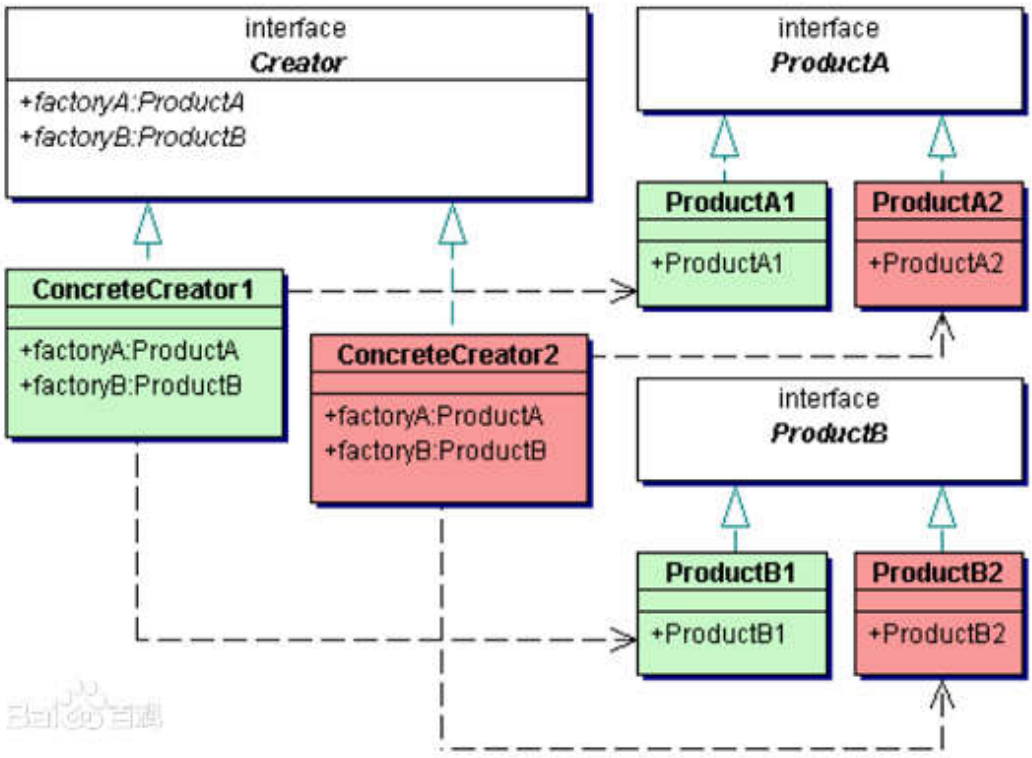


https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/FactoryMethod-%E5%B7%A5%E5%8E%82%E6%96%B9%E6%B3%95.java

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：

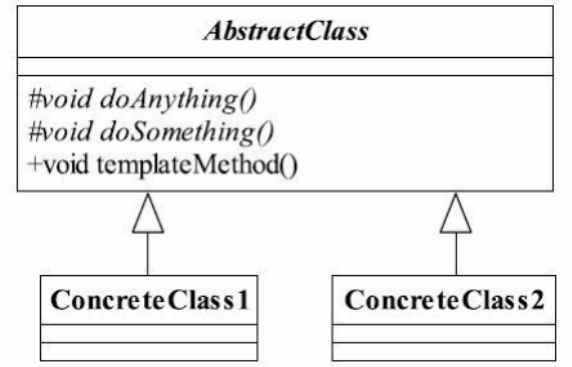


https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/AbstructFactory-%E6%8A%BD%E8%B1%A1%E5%B7%A5%E5%8E%82.java

4 模板模式

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

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Template-%E6%A8%A1%E6%9D%BF%E6%A8%A1%E5%BC%8F.java

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：

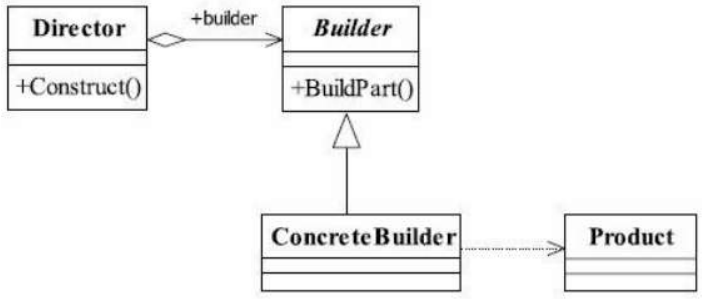


图11-4 建造者模式通用类图

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Builder-%E5%BB%BA%E9%80%A0%E8%80%85%E6%A8%A1%E5%BC%8F.java

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：

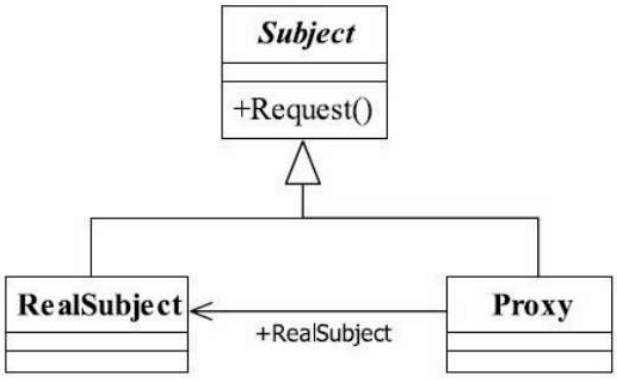


图12-3 代理模式的通用类图

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：

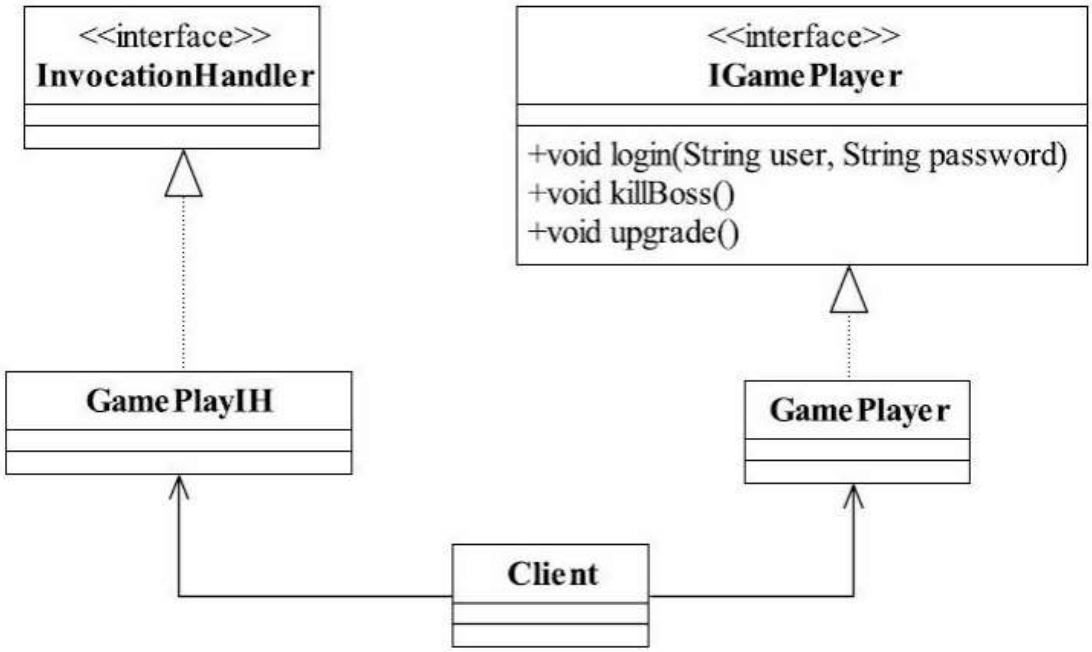


图12-7 动态代理

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Agent-%E4%BB%A3%E7%90%86%E6%A8%A1%E5%BC%8F.java

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**。

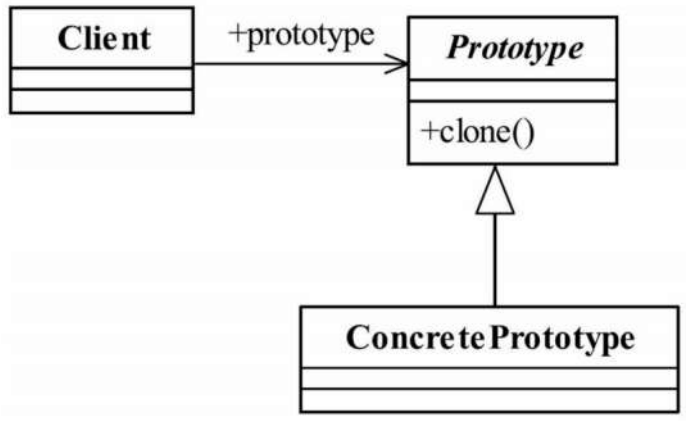


图13-3 原型模式的通用类图

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Prototype-%E5%8E%9F%E5%9E%8B%E6%A8%A1%E5%BC%8F.java

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：

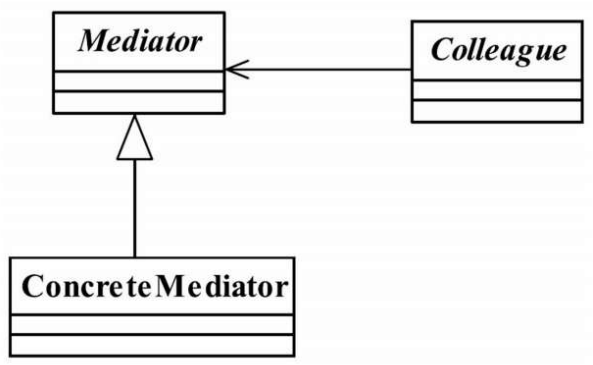


图14-7 中介者模式通用类图

- **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。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Mediator-%E4%B8%AD%E4%BB%8B%E8%80%85%E6%A8%A1%E5%BC%8F.java

9 命令模式

命令模式是一个高内聚的模式，其定义为：将一个请求封装成一个对象，从而让你使用功能不同的请求把客户端参数化，对请求排列或记录请求日志，可以提供命令的撤销和恢复功能。

命令模式类图：

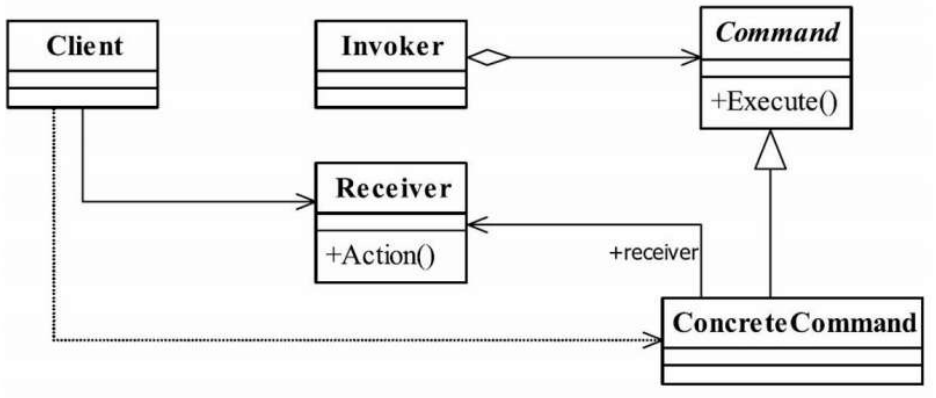


图15-4 命令模式的通用类图

- **Receive 接收者角色**：这是干活的角色，命令传到这里是应该被执行的。

- **Command 命令角色**：需要执行的所有命令都在这里声明。
- **Invoker 调用者角色**：接收到命令，并执行命令。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Command-%E5%91%BD%E4%BB%A4%E6%A8%A1%E5%BC%8F.java

10 责任链模式

责任链模式定义：使多个对象都有机会处理请求，从而避免了请求的发送者和接收者之间的耦合关系。将这些对象组成一条链，并沿着这条链来传递请求，直到有对象处理它为止。

UML 类图：

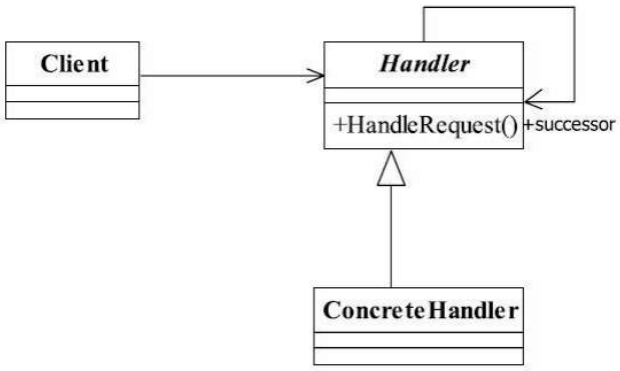


图16-4 责任链模式通用类图

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Chain-%E8%B4%A3%E4%BB%BB%E9%93%BE%E6%A8%A1%E5%BC%8F.java

11 装饰模式

装饰模式定义：动态地给一个对象添加额外的职责，就增加功能来说，装饰模式相比于生成子类更加灵活。

装饰模式类图：

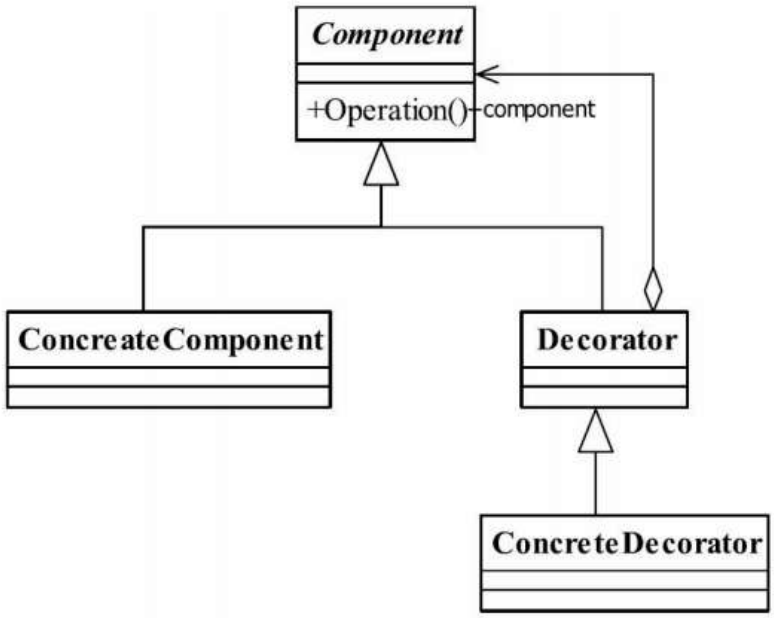


图17-5 装饰模式的通用类图

- **Component 抽象构件**：是一个接口或者抽象类，定义我们最核心的对象，即最原始的对象。
- **ConcreteComponent 具体构件**：是最核心、最原始、最基本的接口或抽象类的实现，你要装饰的就是它。
- **Decorator 装饰角色**：一般是一个抽象类，作用为是实现接口或者方法，它里面可不一定有抽象的方法，在它的属性里必然有一个 private 变量指向 Component 抽象构件。
- **具体装饰角色**：ConcreteDecorator 是具体的装饰类，你要把最核心、最原始、最基本的东西装饰成其他东西。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Decorator-%E8%A3%85%E9%A5%B0%E6%A8%A1%E5%BC%8F.java

12 策略模式

策略模式是比较简单的模式，定义一组算法，将每个算法封装起来，并且使它们之间可以互换。

UML 类图：

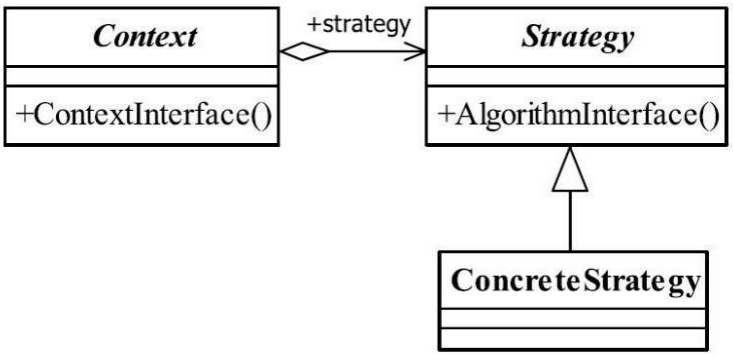


图18-3 策略模式通用类图

- **Context 封装角色**
  它也叫做上下文角色， 起承上启下封装作用， 屏蔽高层模块对策略、 算法的直接访问，封装可能存在的变化。
- **Strategy 抽象策略角色**
  策略、 算法家族的抽象， 通常为接口， 定义每个策略或算法必须具有的方法和属性。
- **ConcreteStrategy 具体策略角色**
  实现抽象策略中的操作， 该类含有具体的算法。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Strategy-%E7%AD%96%E7%95%A5%E6%A8%A1%E5%BC%8F.java

13 适配器模式

适配器模式定义：将一个类的接口变换为客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作。

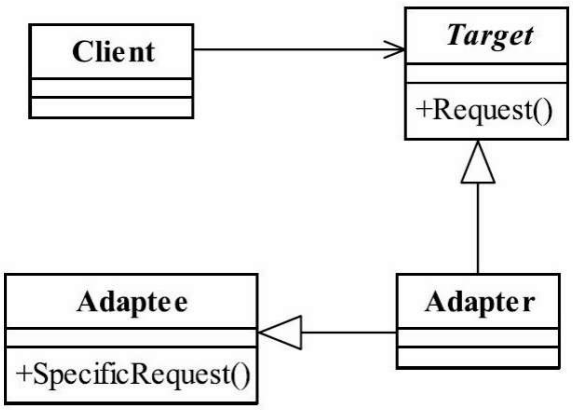


图19-4 适配器模式通用类图

适配器模式的注意事项

适配器模式最好在详细设计阶段不要考虑它，它不是为了解决还处在开发阶段的问题，而是解决正在服役的项目问题，没有一个系统分析师会在做详细设计的时候考虑使用适配器模式，这个模式使用的主要场景是扩展应用中，就像我们上面的那个例子一样，系统扩展了，不符合原有设计的时候才考虑通过适配器模式减少代码修改带来的风险。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Adapter-%E9%80%8

2%E9%85%8D%E5%99%A8%E6%A8%A1%E5%BC%8F.java

14 迭代器模式

迭代器模式提供了一种方法访问一个容器对象中各个元素，而又不暴露该对象的内部细节(目前已很少用到了，容器一般都提供了迭代器)。

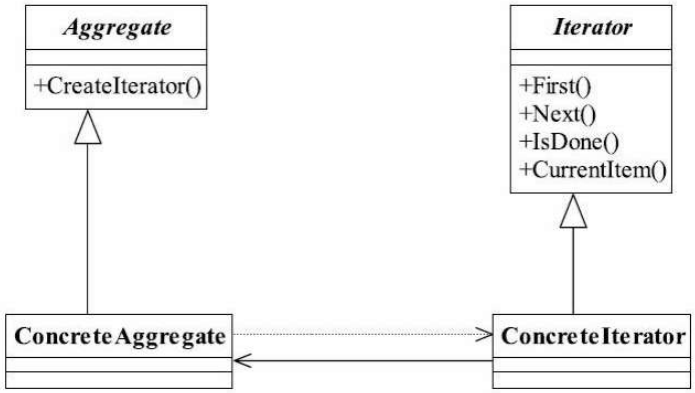


图20-3 迭代器模式的通用类图

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Iterator-%E8%BF%AD%E4%BB%A3%E5%99%A8%E6%A8%A1%E5%BC%8F.java

15 组合模式

组合模式将对象组合成树形结构以表示“整体-部分”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

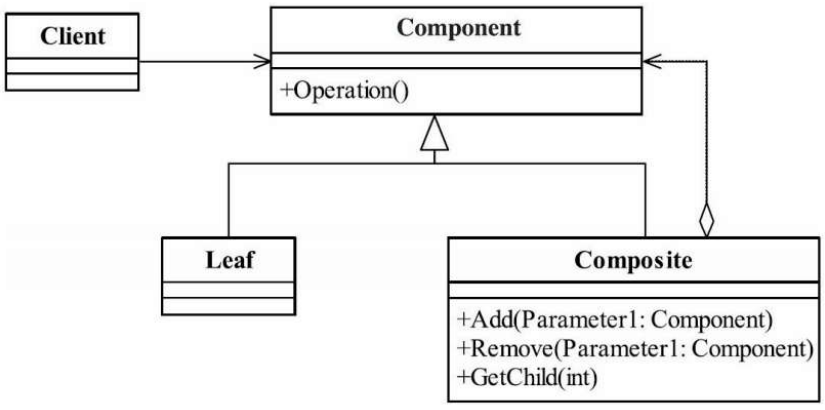


图21-6 组合模式通用类图

- Component 抽象构件角色
  定义参加组合对象的共有方法和属性， 可以定义一些默认的行为或属性， 比如我们例子中的 getInfo 就封装到了抽象类中。
- Leaf 叶子构件

叶子对象， 其下再也没有其他的分支， 也就是遍历的最小单位。

- Composite 树枝构件

  树枝对象， 它的作用是组合树枝节点和叶子节点形成一个树形结构。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Component-%E7%BB%84%E5%90%88%E6%A8%A1%E5%BC%8F.java

16 观察者模式

观察者模式(Observer Pattern)也叫作发布订阅模式，其定义如下：定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并自动更新。

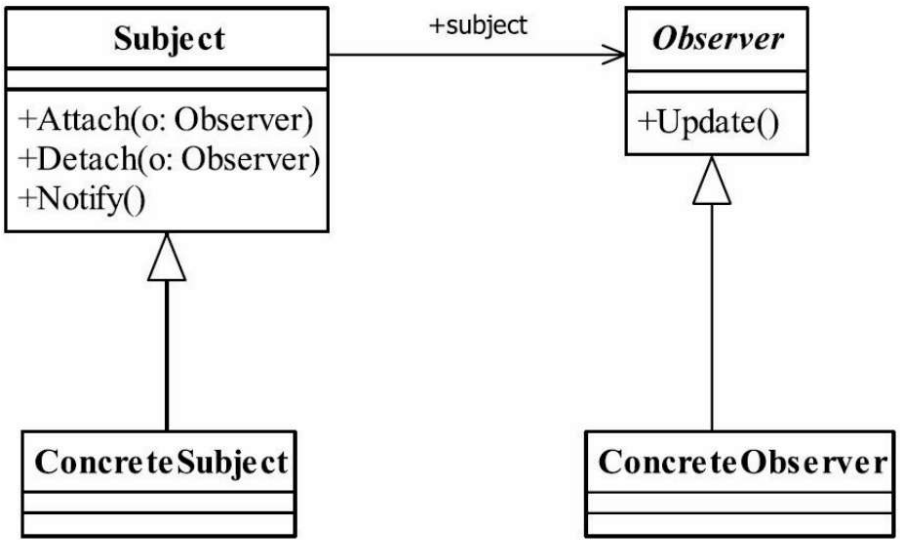


图22-5 观察者模式通用类图

Subject：被观察者，Observer：观察者

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Observer-%E8%A7%82%E5%AF%9F%E8%80%85%E6%A8%A1%E5%BC%8F.java

17 门面模式

门面模式要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。

门面模式类图：

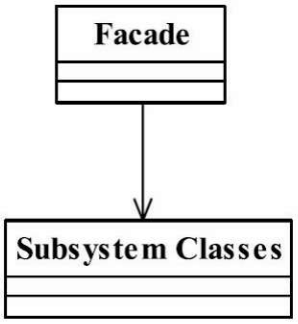


图23-4 扩展后的系统类图

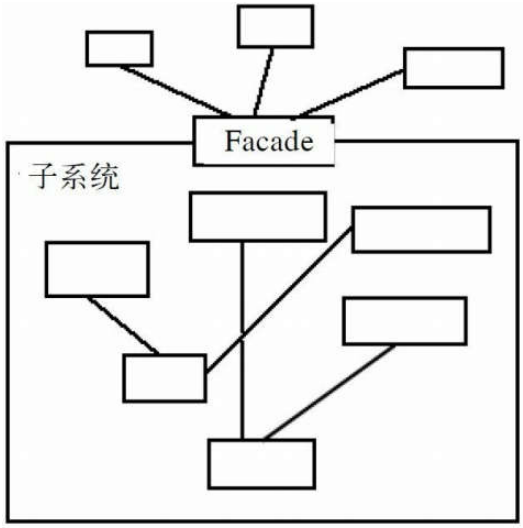


图23-5 门面模式示意图

- Facade 门面角色：客户端可以调用这个角色的方法。 此角色知晓子系统的所有功能和责任。 一般情况下，本角色会将所有从客户端发来的请求委派到相应的子系统去， 也就说该角色没有实际的业务逻辑， 只是一个委托类。
- subsystem 子系统角色：可以同时有一个或者多个子系统。 每一个子系统都不是一个单独的类， 而是一个类的集合。 子系统并不知道门面的存在。 对于子系统而言， 门面仅仅是另外一个客户端而已。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Facade-%E9%97%A8%E9%9D%A2%E6%A8%A1%E5%BC%8F.java

18 备忘录模式

备忘录模式定义：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

备忘录模式通用类图

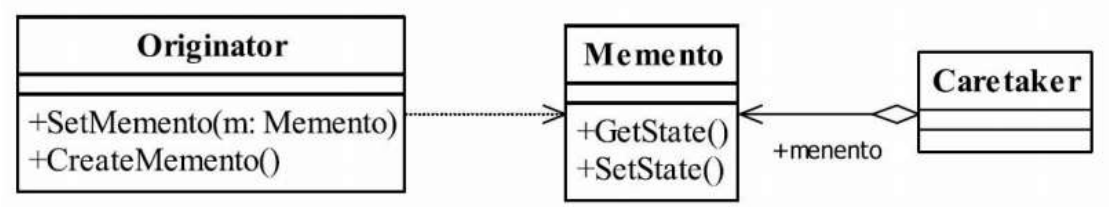


图24-4 备忘录模式的通用类图

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Memento-%E5%A4%87%E5%BF%98%E5%BD%95%E6%A8%A1%E5%BC%8F.java

19 访问者模式

访问者模式定义：封装一些作用于某种数据结构中的个元素的操作，它可以在不改变数据结构的前提下定义作用于这些元素的新的操作。

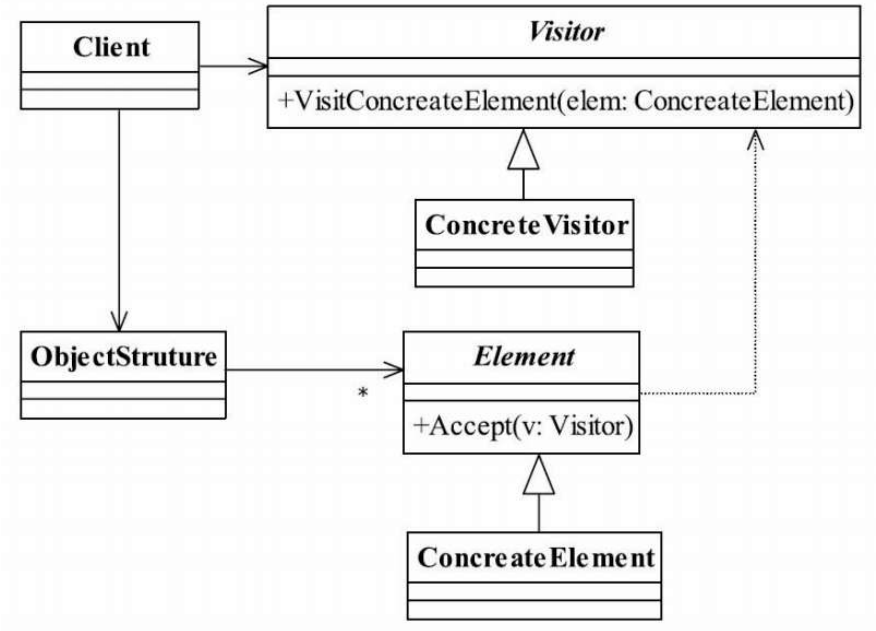


图25-5 访问者模式的通用类图

- **Visitor——抽象访问者**
  抽象类或者接口， 声明访问者可以访问哪些元素， 具体到程序中就是 visit 方法的参数定义哪些对象是可以被访问的。
- **ConcreteVisitor——具体访问者**
  它影响访问者访问到一个类后该怎么干， 要做什么事情。
- **Element——抽象元素**
  接口或者抽象类， 声明接受哪一类访问者访问， 程序上是通过 accept 方法中的参数来定义的。
- **ConcreteElement——具体元素**
  实现 accept 方法， 通常是 visitor.visit(this)， 基本上都形成了一种模式了。
- **ObjectStruture——结构对象**
  元素产生者， 一般容纳在多个不同类、 不同接口的容器， 如 List、 Set、 Map 等，在项目中， 一般很少抽象出这个角色。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Visitor-%E8%AE%BF%E9%97%AE%E8%80%85%E6%A8%A1%E5%BC%8F.java

20 状态模式

状态模式定义：当一个对象内在状态改变时允许其改变行为，这个对象看起来像改变了

其类。状态模式的核心是封装，状态的变更引起了行为的变更，从外部看起来就好像这个对象对应的类发生了改变一样。 状态模式的通用类图：

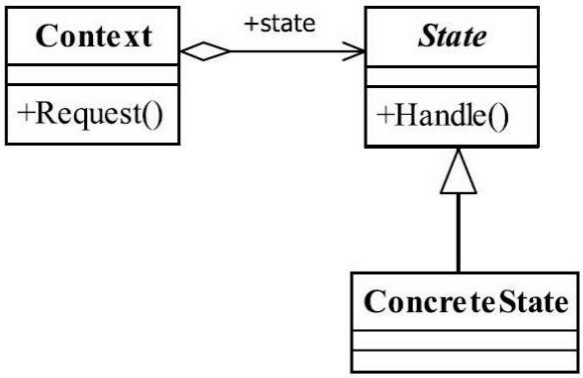


图26-5 状态模式通用类图

● State——抽象状态角色

接口或抽象类， 负责对象状态定义， 并且封装环境角色以实现状态切换。

● ConcreteState——具体状态角色

每一个具体状态必须完成两个职责：本状态的行为管理以及趋向状态处理，通俗地说，就是本状态下要做的事情， 以及本状态如何过渡到其他状态。

● Context——环境角色

定义客户端需要的接口， 并且负责具体状态的切换。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/State-%E7%8A%B6%E6%80%81%E6%A8%A1%E5%BC%8F.java

21 解释器模式

解释器模式定义：给定一门语言， 定义它的文法的一种表示， 并定义一个解释器， 该解释器使用该表示来解释语言中的句子。

22 享元模式

享元模式是池技术的重要实现，使用共享对象可有效地支持大量的细粒度的对象。

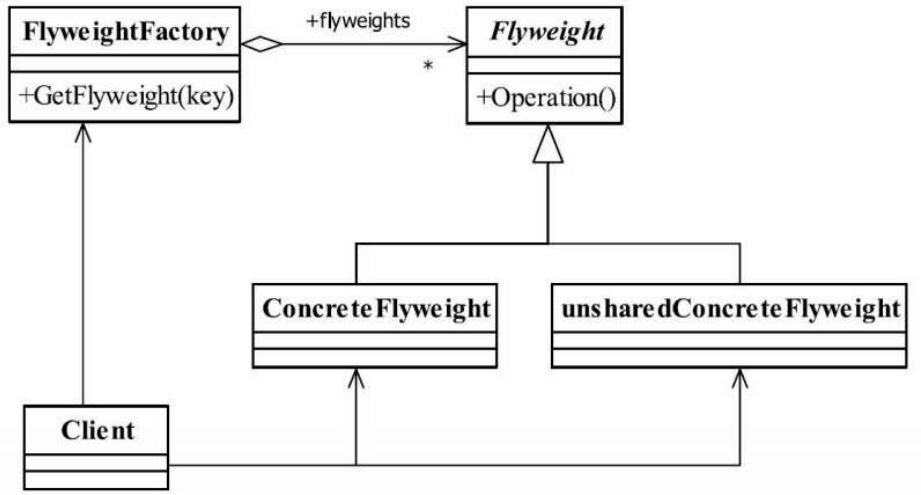


图28-3 享元模式的通用类图

Flyweight——抽象享元角色

它简单地说就是一个产品的抽象类，同时定义出对象的外部状态和内部状态的接口或实现。

● ConcreteFlyweight——具体享元角色

具体的一个产品类，实现抽象角色定义的业务。该角色中需要注意的是内部状态处理应该与环境无关，不应该出现一个操作改变了内部状态，同时修改了外部状态，这是绝对不允许的。

● unsharedConcreteFlyweight——不可共享的享元角色

不存在外部状态或者安全要求（如线程安全）不能够使用共享技术的对象，该对象一般不会出现在享元工厂中。

● FlyweightFactory——享元工厂

职责非常简单， 就是构造一个池容器， 同时提供从池中获得对象的方法。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Flyweight-%E4%BA%AB%E5%85%83%E6%A8%A1%E5%BC%8F.java

23 桥梁模式

桥梁模式定义：将抽象和实现解耦，使得两者可以独立的变化。

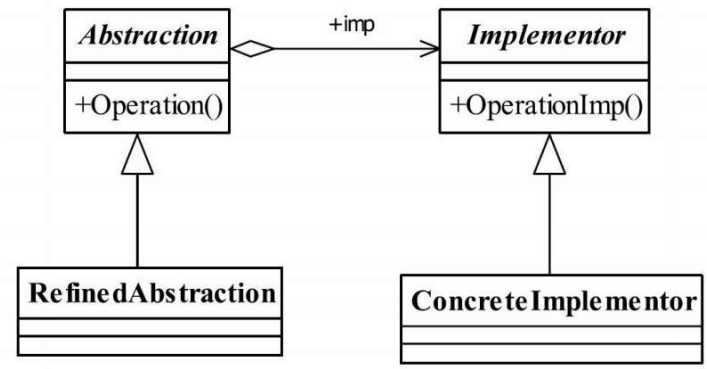


图29-4 桥梁模式通用类图

- **Abstraction——抽象化角色**：它的主要职责是定义出该角色的行为， 同时保存一个对实现化角色的引用， 该角色一般是抽象类。
- **Implementor——实现化角色**：它是接口或者抽象类， 定义角色必需的行为和属性。
- **RefinedAbstraction——修正抽象化角色**：它引用实现化角色对抽象化角色进行修正。
- **ConcreteImplementor——具体实现化角色**：它实现接口或抽象类定义的方法和属性。

https://github.com/luoxn28/ThinkInTechnology/blob/master/DesignPattern/Bridge-%E6%A1%A5%E6%A2%81%E6%A8%A1%E5%BC%8F.java