**Java设计模式整理汇总**

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可**重用代码**、**让代码更容易被他人理解、保证代码可靠性**。 毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。简单说：**模式：在某些场景下，针对某类问题的某种通用的解决方案。**

场景：项目所在的环境问题：约束条件，项目目标等

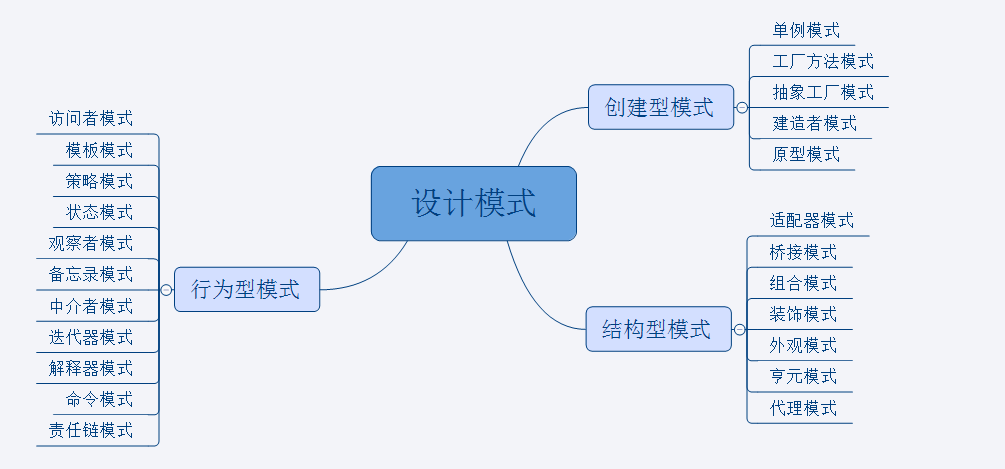
解决方案：**通用、可复用的设计**，解决约束达到目标。

**设计模式的三个分类 ：**

**创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。**

**结构型模式：把类或对象结合在一起形成一个更大的结构。**

**行为型模式：类和对象如何交互，及划分责任和算法。**



**各分类中模式的关键点 ：**

**单例模式：**某个类只能有一个实例，提供一个全局的访问点。

**工厂模式：**创建相关或依赖对象的家族，而无需明确指定具体类。

**适配器模式：**将一个类的方法接口转换成客户希望的另外一个接口。

**装饰模式：**动态的给对象添加新的功能。

**代理模式：**为其他对象提供一个代理以便控制这个对象的访问。

**策略模式**：定义一系列算法，把他们封装起来，并且使它们可以相互替换。

**责任链模式**：将请求的发送者和接收者解耦，使的多个对象都有处理这个请求的机会。

**模板模式**：定义一个算法结构，而将一些步骤延迟到子类实现。

概说23种设计模式 **：**

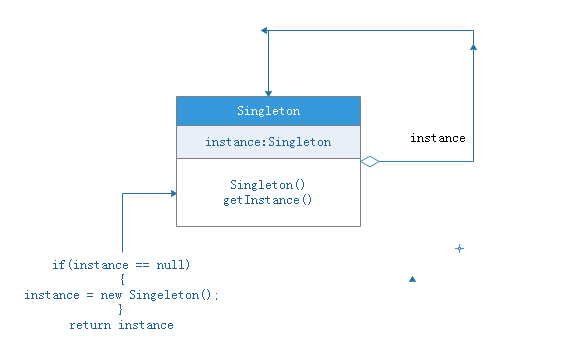
**单例模式**

单例模式，**它的定义就是确保某一个类只有一个实例，并且提供一个全局访问点**。

单例模式具备典型的3个特点：1、只有一个实例。 2、自我实例化。 3、提供全局访问点。

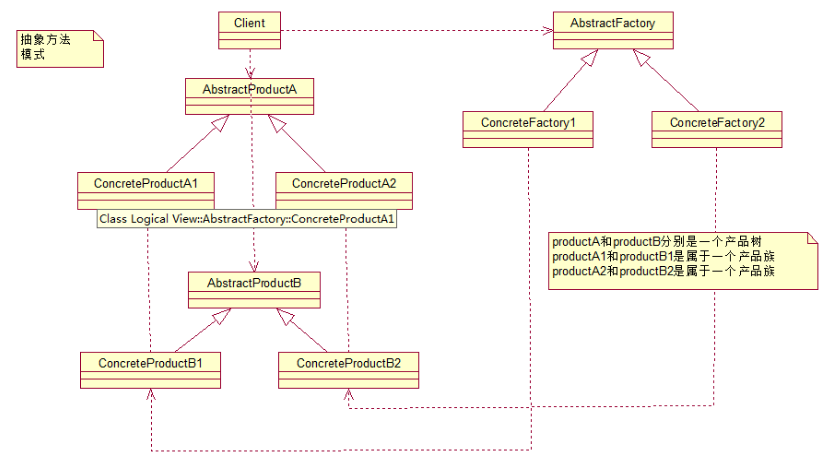
  因此**当系统中只需要一个实例对象或者系统中只允许一个公共访问点，除了这个公共访问点外，不能通过其他访问点访问该实例时，可以使用单例模式。**

单例模式的主要优点就是**节约系统资源、提高了系统效率**，同时也能够**严格控制客户对它的访问**。也许就是因为系统中只有一个实例，这样就导致了单例类的职责过重，违背了“单一职责原则”，同时也没有抽象类，**所以扩展起来有一定的困难**。如下图：



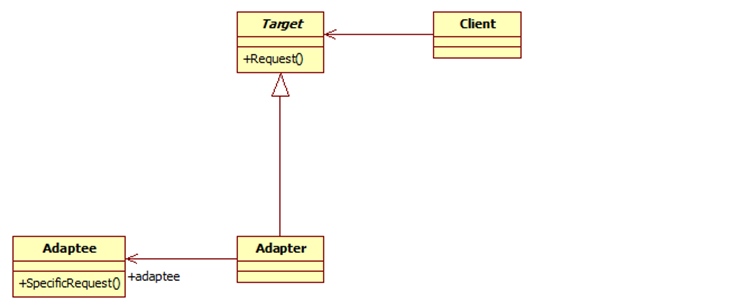
**抽象工厂模式**

所谓抽象工厂模式就是**提供一个接口，用于创建相关或者依赖对象的家族，而不需要明确指定具体类**。他允许客户端使用抽象的接口来创建一组相关的产品，而不需要关系实际产出的具体产品是什么。这样一来，客户就可以从具体的产品中被解耦。**它的优点是隔离了具体类的生成，使得客户端不需要知道什么被创建了**，而缺点就在于新增新的行为会比较麻烦，因为当添加一个新的产品对象时，需要更加需要更改接口及其下所有子类。



适配器模式

在我们的应用程序中我们可能**需要将两个不同接口的类来进行通信**，在不修改这两个的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。**所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口**。它可以让原本两个不兼容的接口能够无缝完成对接。作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

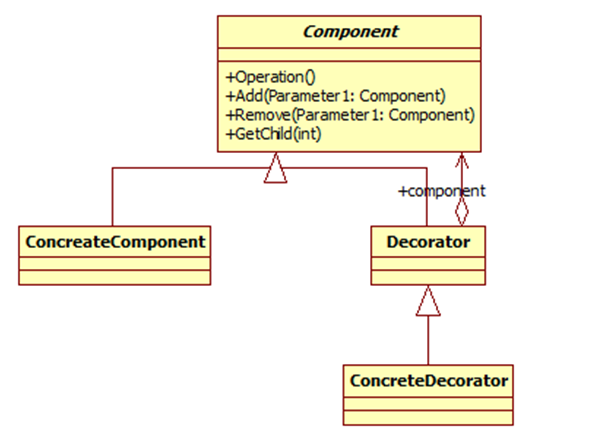


适配器模式包含如下角色：  
**Target：目标抽象类  
Adapter：适配器类  
Adaptee：适配者类  
Client：客户类**

装饰模式

我们可以通过**继承和组合的方式来给一个对象添加行为**，虽然使用继承能够很好拥有父类的行为，但是它存在几个缺陷：一、**对象之间的关系复杂的话**，系统变得复杂不利于维护。二、**容易产生“类爆炸”**现象。三、是静态的。在这里我们可以通过使用装饰者模式来解决这个问题。

**装饰者模式**，动态地将责任附加到对象上。若要扩展功能，装饰者提供了比继承更加有弹性的替代方案。虽然装饰者模式能够动态将责任附加到对象上，但是他会产生许多的细小对象，增加了系统的复杂度。

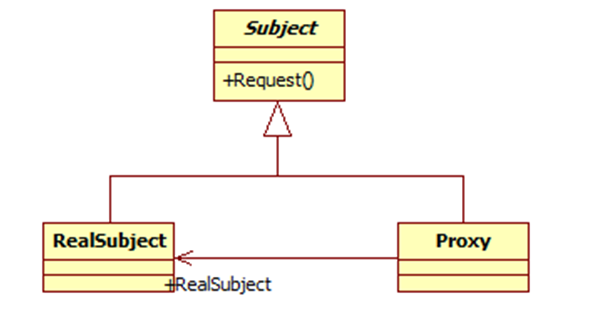


**模式结构**  
装饰模式包含如下角色：  
Component: 抽象构件  
ConcreteComponent: 具体构件  
Decorator: 抽象装饰类  
ConcreteDecorator: 具体装饰类

代理模式

**代理模式就是给一个对象提供一个代理，并由代理对象控制对原对象的引用**。它使得客户**不能直接与真正的目标对象通信**。代理对象是目标对象的代表，其他需要与这个目标对象打交道的操作都是和这个代理对象在交涉。

代理对象可以在客户端和目标对象之间起到**中介的**作用，这样起到了的作用和保护了目标对象的，同时也在一定程度上面减少了系统的耦合度。



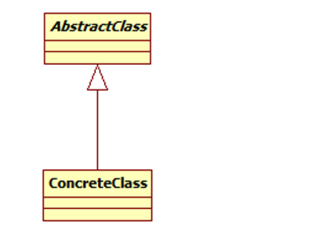
代理模式包含如下角色：  
Subject: 抽象主题角色  
Proxy: 代理主题角色  
RealSubject: 真实主题角色

模板模式

有些时候我们做某几件**事情的步骤都差不多**，仅有那么一小点的不同，在软件开发的世界里同样如此，如果我们都将这些步骤都一一做的话，费时费力不讨好。所以我们可以将这些步骤分解、封装起来，然后利用继承的方式来继承即可，当然不同的可以自己重写实现嘛！这就是模板方法模式提供的解决方案。

所谓模板方法模式就是**在一个方法中定义一个算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。**

模板方法模式就是**基于继承的代码复用技术的**。在模板方法模式中，我们可以将相同部分的代码放在父类中，而将不同的代码放入不同的子类中。也就是说我们需要声明一个抽象的父类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法让子类来实现剩余的逻辑，不同的子类可以以不同的方式来实现这些逻辑。所以模板方法的模板其实就是一个普通的方法，只不过这个方法是将算法实现的步骤封装起来的。



模板方法模式包含如下角色：  
AbstractClass: 抽象类   
ConcreteClass: 具体子类

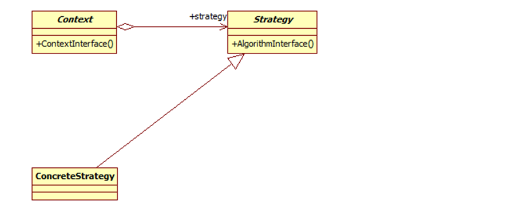
策略模式

我们知道一件事可能会有很多种方式来实现它，但是其中总有一种最高效的方式，在软件开发的世界里面同样如此，我们也有很多中方法来实现一个功能，但是我们需要一种简单、高效的方式来实现它，使得系统能够非常灵活，这就是策略模式。

所以策略模式就是定义了算法族，分别封装起来，让他们之前可以互相转换，此模式然该算法的变化独立于使用算法的客户。

在策略模式中它将这些解决问题的方法定义成一个算法群，每一个方法都对应着一个具体的算法，这里的一个算法我就称之为一个策略。虽然策略模式定义了算法，但是它并不提供算法的选择，即什么算法对于什么问题最合适这是策略模式所不关心的，所以对于策略的选择还是要客户端来做。客户必须要清楚的知道每个算法之间的区别和在什么时候什么地方使用什么策略是最合适的，这样就增加客户端的负担。

同时策略模式也非常完美的符合了“开闭原则”，**用户可以在不修改原有系统的基础上选择算法或行为，也可以灵活地增加新的算法或行为。但是一个策略对应一个类将会是系统产生很多的策略类。**

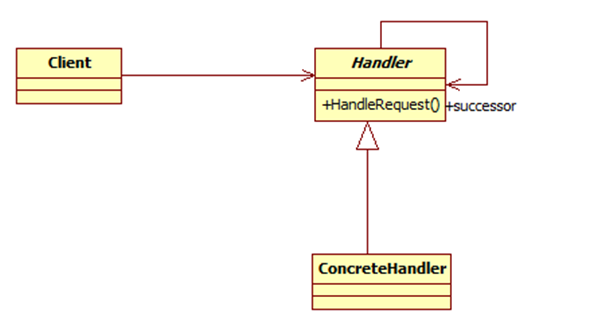


策略模式包含如下角色：  
Context: 环境类  
Strategy: 抽象策略类  
ConcreteStrategy: 具体策略类

责任链模式

职责链模式描述的请求如何**沿着对象所组成的链来传递的**。它**将对象组成一条链，发送者将请求发给链的第一个接收者，并且沿着这条链传递，直到有一个对象来处理它或者直到最后也没有对象处理而留在链末尾端。**

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止，这就是职责链模式。在职责链模式中，使得每一个对象都有可能来处理请求，从而实现了请求的发送者和接收者之间的解耦。同时职责链模式简化了对象的结构，它使得每个对象都只需要引用它的后继者即可，而不必了解整条链，这样既提高了系统的灵活性也使得增加新的请求处理类也比较方便。但是在职责链中我们不能保证所有的请求都能够被处理，而且不利于观察运行时特征。



职责链模式包含如下角色：  
Handler: 抽象处理者  
ConcreteHandler: 具体处理者  
Client: 客户类