### 简单工厂模式：

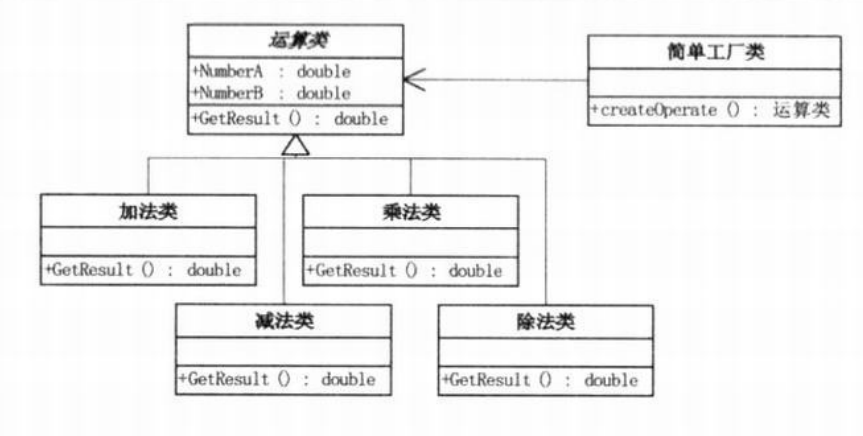
**简介：**

又叫做静态工厂方法（Static Factory Method）模式。由一个工厂对象决定创建出哪一种产品类的实例。

**实现:**

定义产品的公共接口以及具体的产品实现类和一个工厂类。工厂类通过传入的条件不同，生成相应的产品类对象。

**UML图：**



### 策略模式：

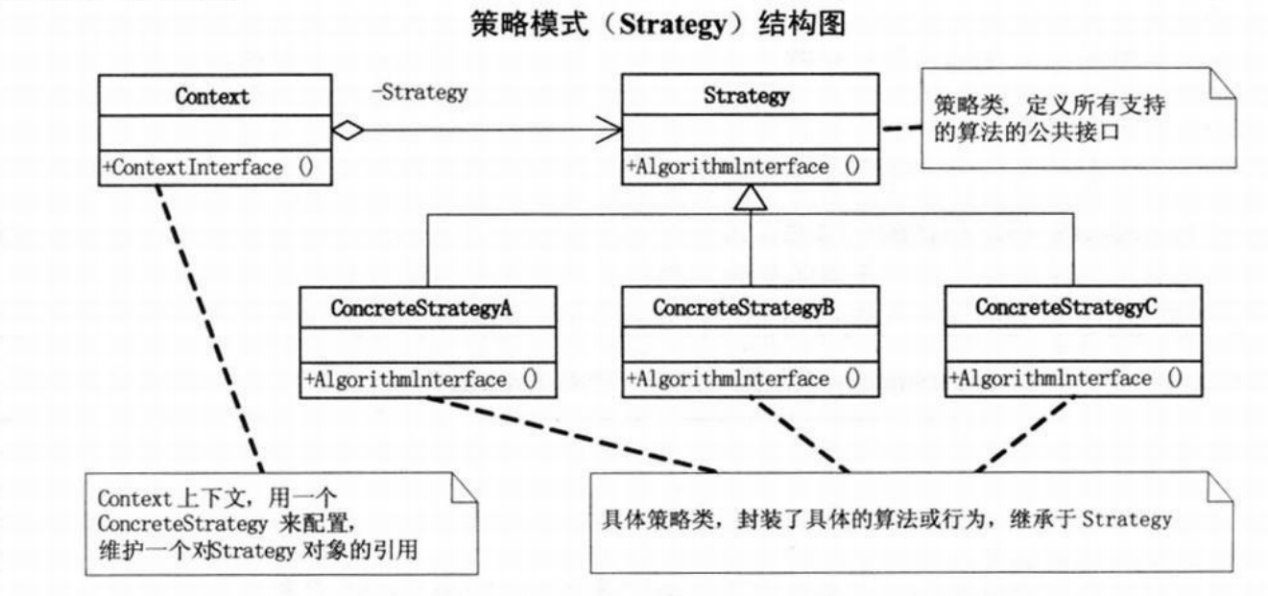
**简介：**

定义了算法家族，分别封装起来，让他们之间相互可以替换，让算法的变化，不会影响到使用者。

**实现：**

定义一个支持所有算法的接口或者抽象类，以及一个上下文context对象和具体的算法类。context对象持有算法接口，通过构造方法传入具体的算法类，通过调用context对象的方法来调用具体算法类的算法方法。

**UML图：**



### 装饰模式：

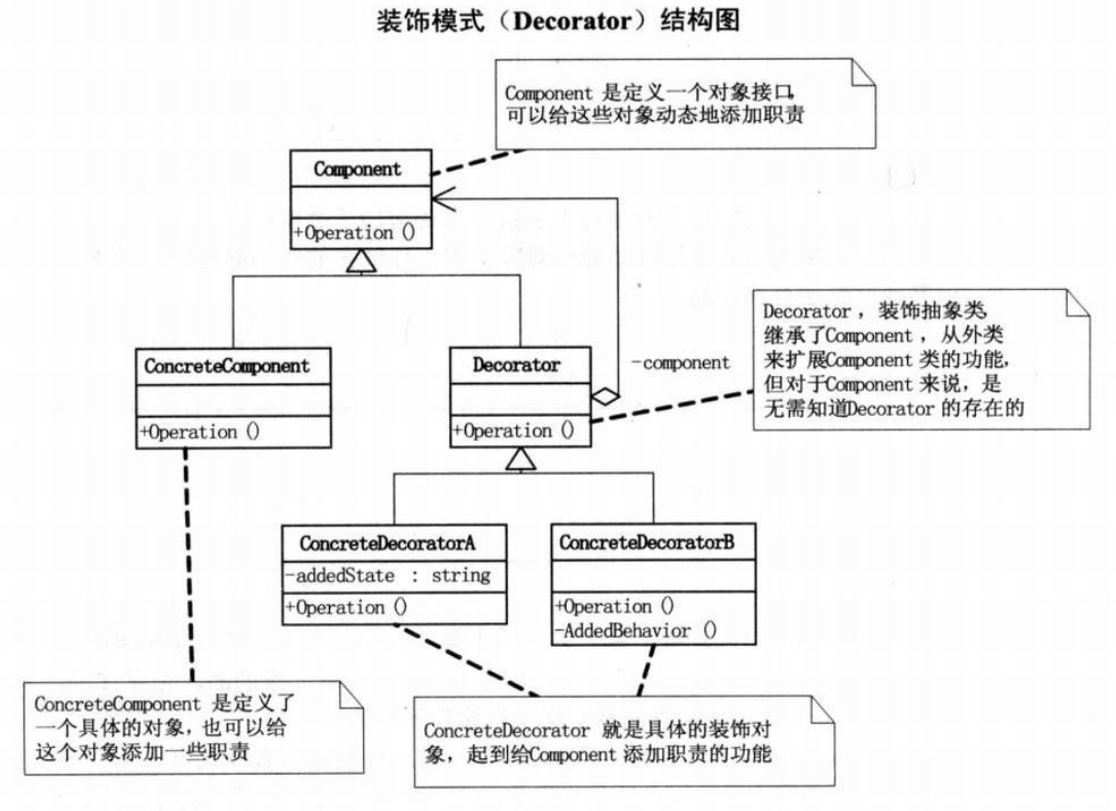
**简介：**

动态的给一个对象添加一些额外的职责，就增加功能来说比生成子类更灵活。

**实现：**

首先定义一个对象接口component，然后定义具体对象类和装饰抽象类，实现（继承）component。然后再定义具体的装饰类，继承装饰抽象类，这些类就是用来给component添加职责的。

**UML图：**



### 代理模式：

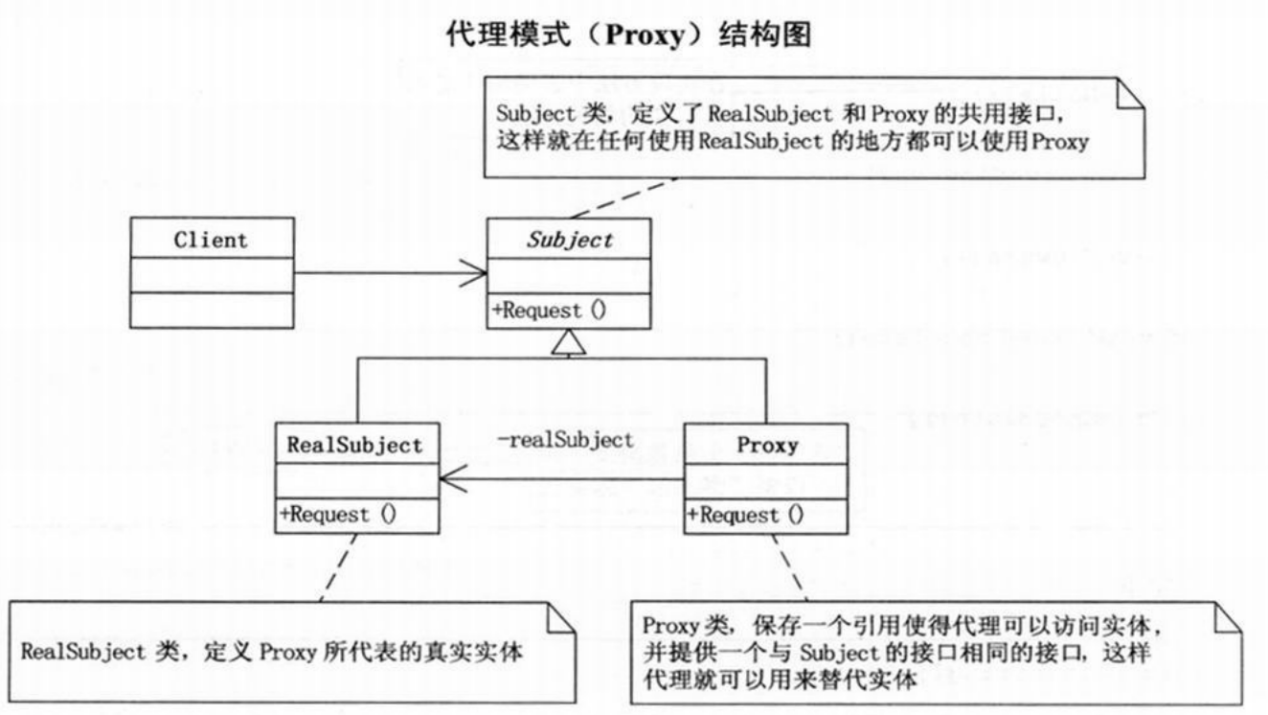
**简介：**

为其他对象提供一种代理以控制对这个对象的访问。

**实现：**

定义一个公共接口，代理类和被代理类都实现这个接口。代理类持有被代理类的对象，再代理类的方法中调用被代理类对应的方法，用户通过调用代理类的方法来实现被代理类的调用。

**UML图:**



### 工厂方法模式：

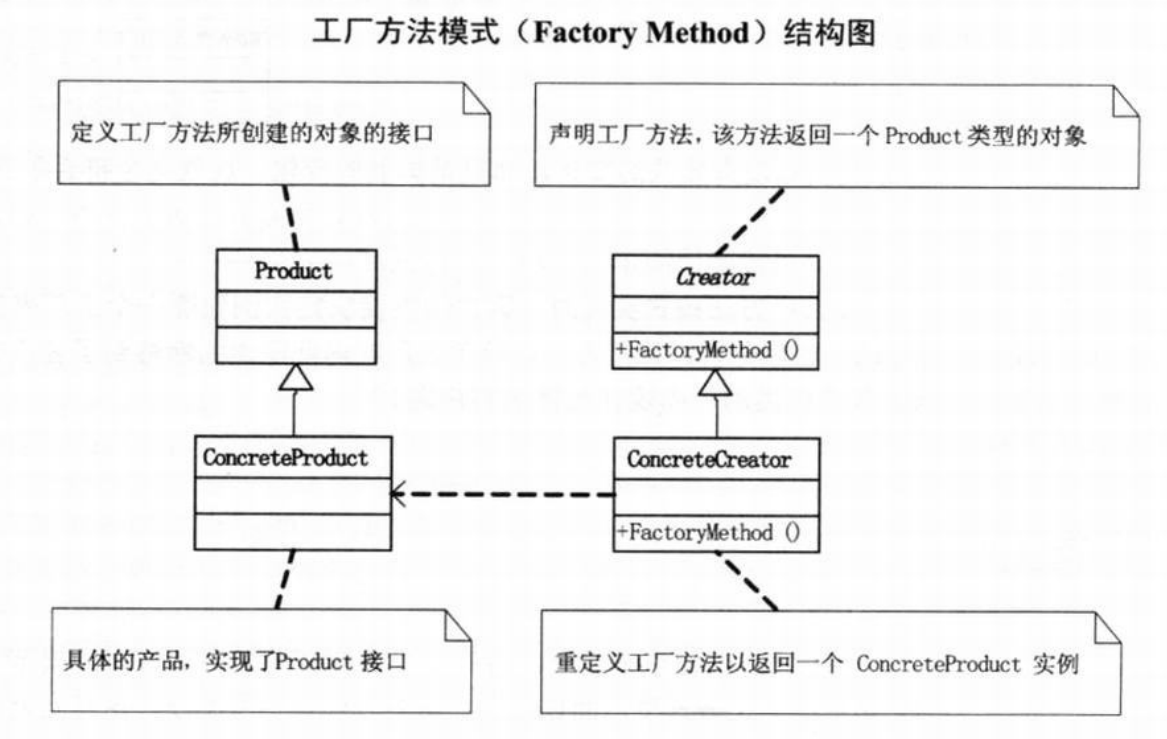
**简介：**

同简单工厂方法相似，通过工厂类来创建产品类对象。但是在工厂方法每个产品类都对应一个工厂类，所有工厂类都实现同一个接口。

**实现：**

为每一个具体产品类创建一个工厂类，这些工厂类实现同一个工厂接口。调用时先创建对应工厂类，然后再用工厂类创建产品类对象。

**UML图：**



### 原型模式：

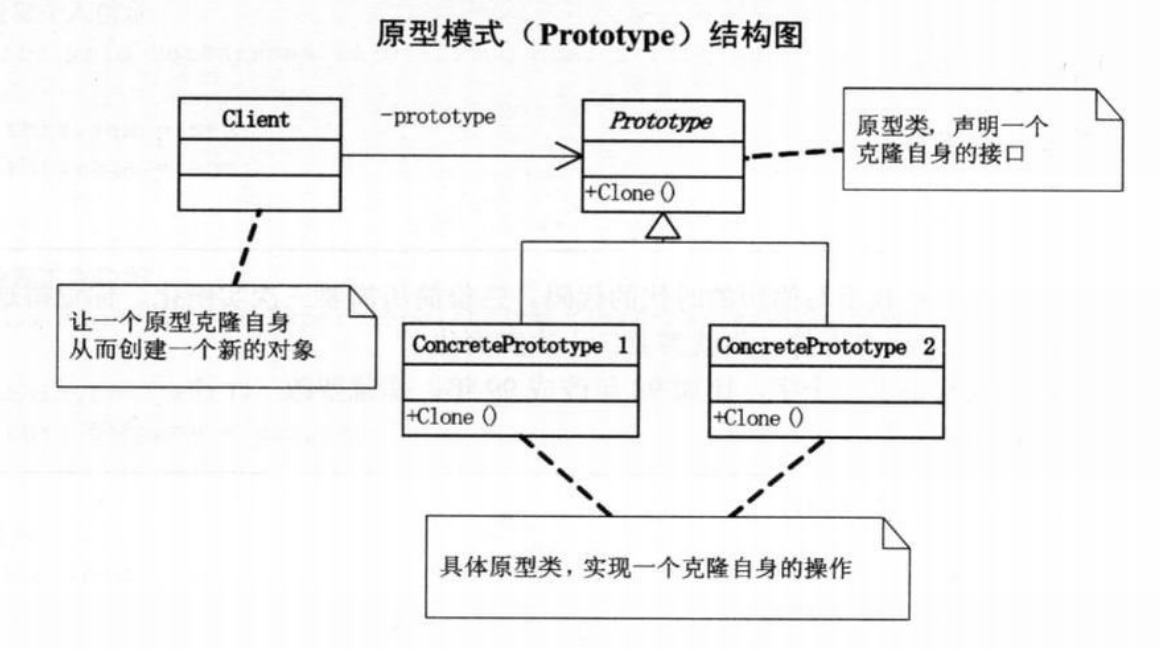
**简介：**

通过原型类对象拷贝（克隆）得到新的对象。

**实现：**

在类中定义克隆方法，注意浅复制和深复制。

**UML图：**



### 模板方法模式：

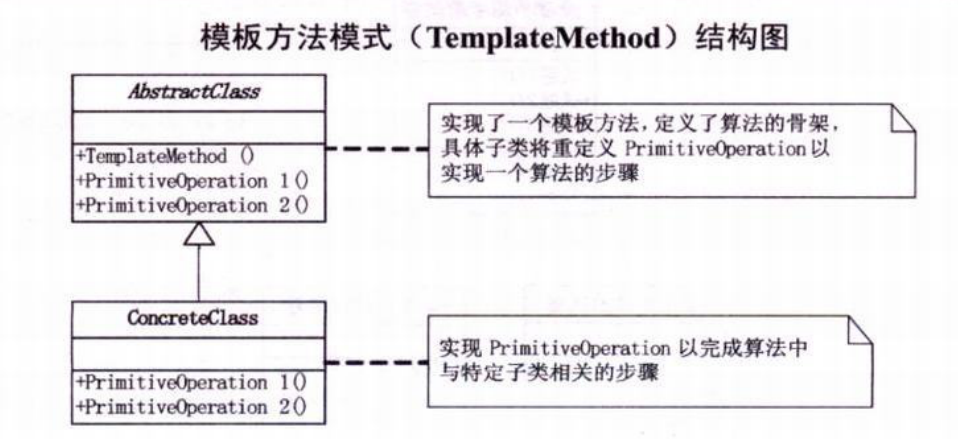
**简介：**

定义一个操作中算法的骨架（不变的部分），而将一些步骤（需要改变的部分）延迟到子类中，使子类不需改变算法的结构即可重新定义算法的某些特定的步骤。

**实现：**

父类实现算法的方法A调用了其他方法B，子类继承A方法，重写了B方法。当子类对象调用A时，它会调用子类重写的B方法，而不是父类的B方法。

**UML图：**



### 外观模式：

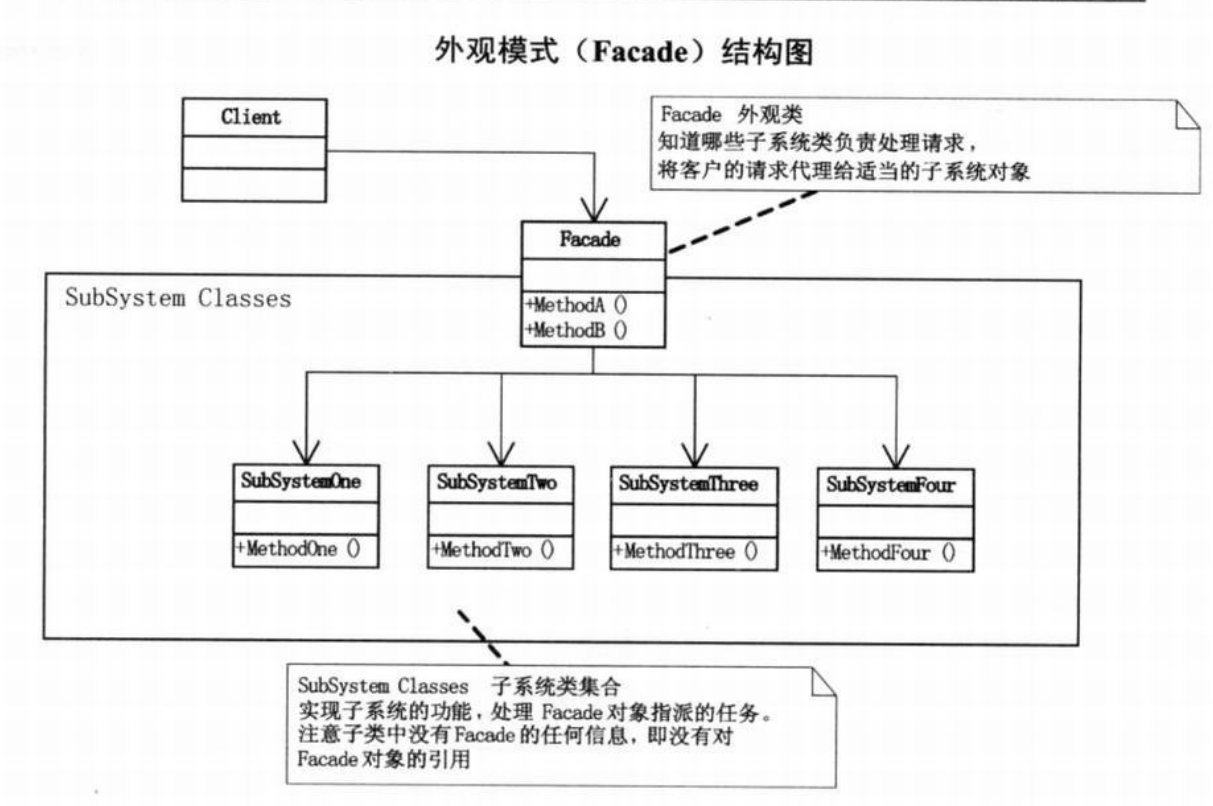
**简介：**

为子系统的一组接口提供一个一致的界面。定义一个高层接口，使得这一子系统更加容易使用。

**实现：**

为子系实现一个外观类，通过这个外观类来调用子系统的功能实现代码。适用于维护遗留的大型系统，当系统难以修改和扩展时，可以通过外观类来调用该系统的功能代码。而新系统只需调用外观类即可。

**UML图：**



### 建造者模式：

**简介:**

将一个复杂对象的构建与他的表示分离，使得同样的构建过程可以构建不同的表示。

**实现：**

创建指挥者类Director和建造者公共接口以及具体的建造者类和产品类，建造者类接口定义建造产品对象的公共行为，具体建造者类的建造细节不同。指挥者类执行建造过程，通过传入的建造者对象不同，建造出来的产品类也会有所不同。

**UML图：**



### 观察者模式：

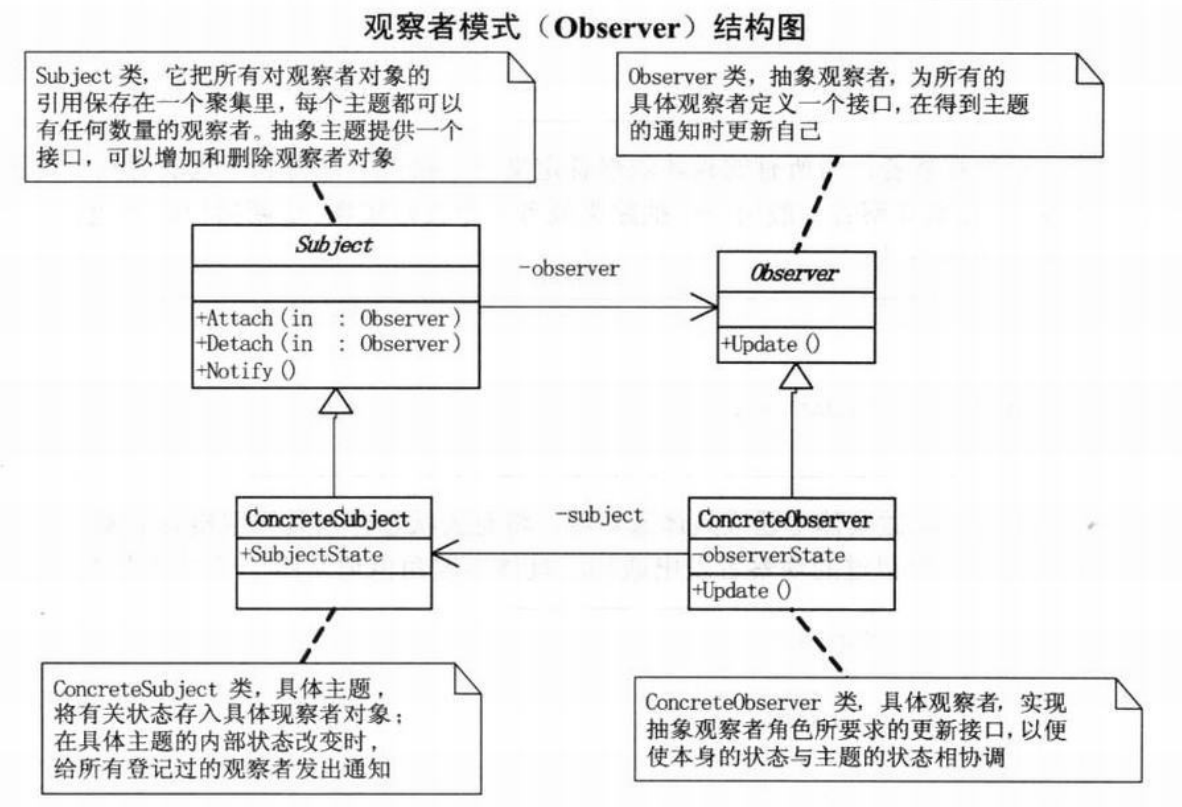
**简介：**

定义了一种一对多的依赖关系，让多个观察者对象同时监听某一个主题对象，使这个主题在状态发生变化时能通知所有的观察者对象，让他们自己更新状态。

**实现：**

定义主题的抽象接口和抽象观察者，创建具体主题类和具体观察者，主题类中包含观察者集合。当状态发生变化时，遍历通知观察者。

**UML图：**



### 抽象工厂模式：

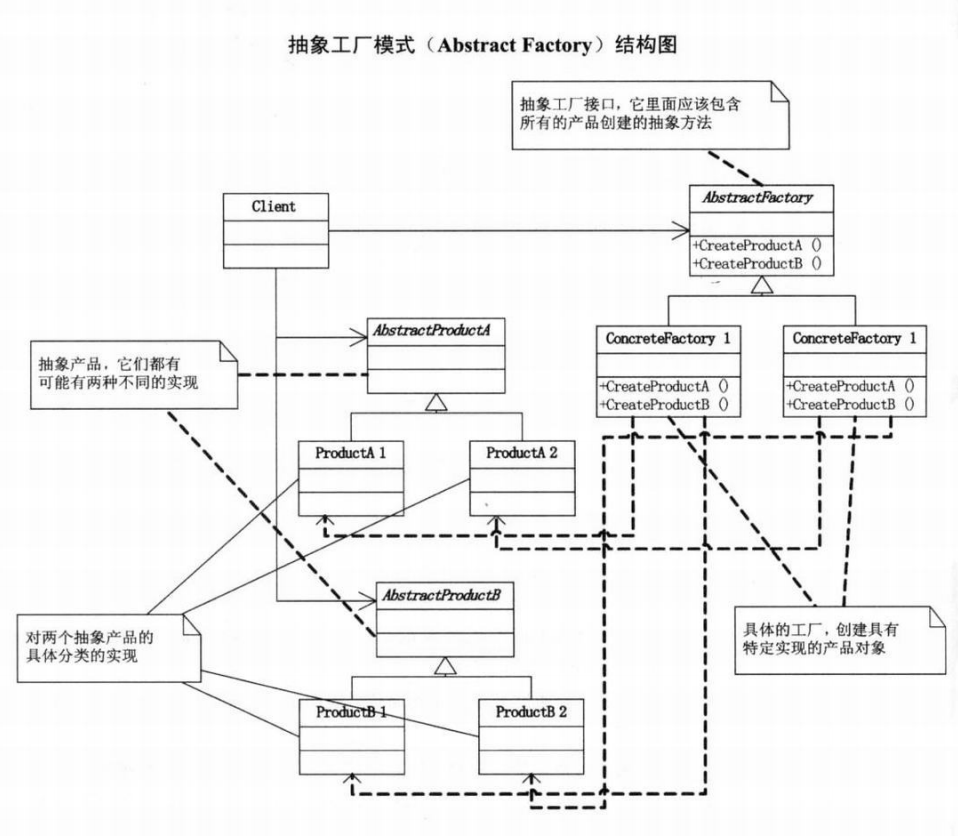
**简介：**

和工厂方法类似，工厂方法模式中每个工厂对应的一个具体的产品类。而抽象工厂模式中的工厂对应的是一个产品的抽象，根据条件不同再生成具体的产品类，它对应的是一个产品系列。

**实现：**

和工厂方法类似，只需要把生成具体产品类换成系列产品的抽象即可。

**UML图：**



### 状态模式：

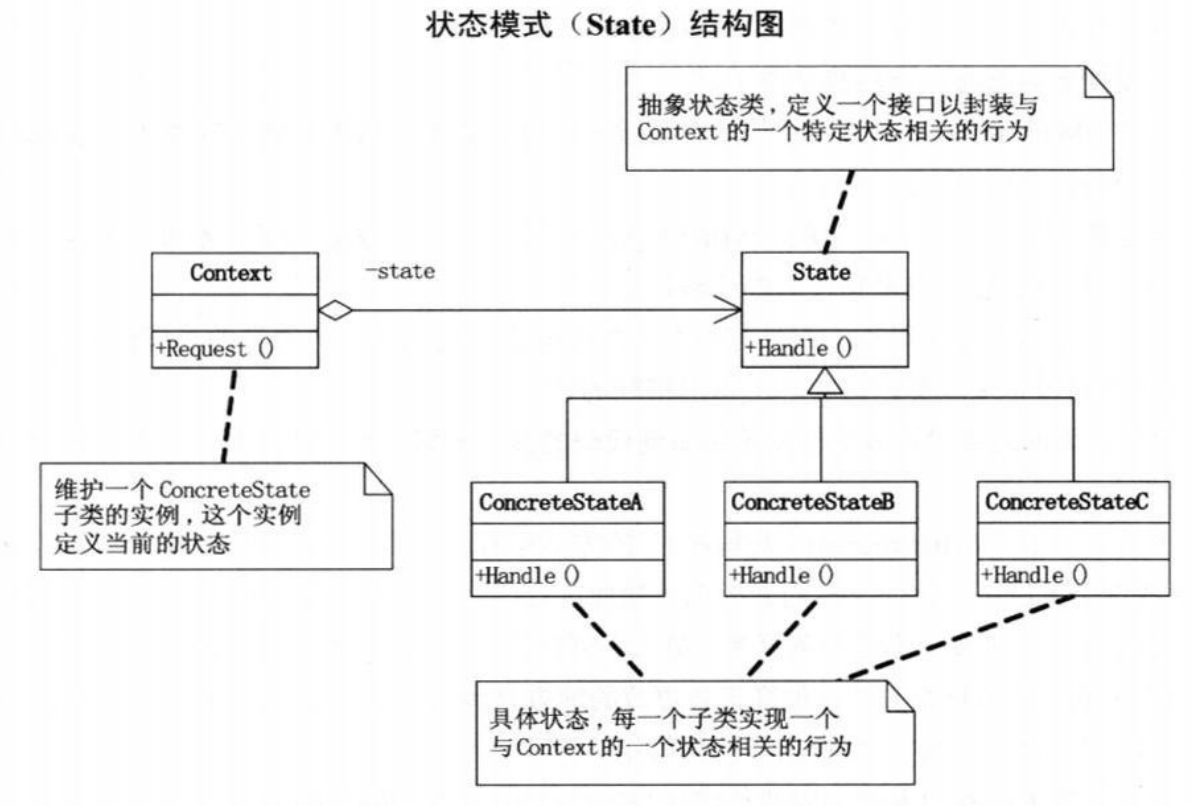
**简介：**

当一个对象的内在状态改变时，允许改变其行为，这个对象看起来像是改变了其类。具体表现为当控制一个对象状态转换的条件表达式过于复杂时，把状态的判断逻辑转移到表示不同状态的一系列类中，可以简化复杂的判断逻辑。

**实现：**

将对象的一系列状态抽象为一系列的状态类，并持有一个状态类对象。当状态发生变化时，将这个状态类对象改为变化后的状态类类型。各个状态子类必须知道下一个状态是什么。状态模式的UML图和策略模式的UML图基本相同。但是状态模式的状态类是有关联的，一个状态必须要知道它下一个状态是谁。而策略模式的算法是独立的，用户根据自己的需要选择合适的算法类。

**UML图：**



### 适配器模式：

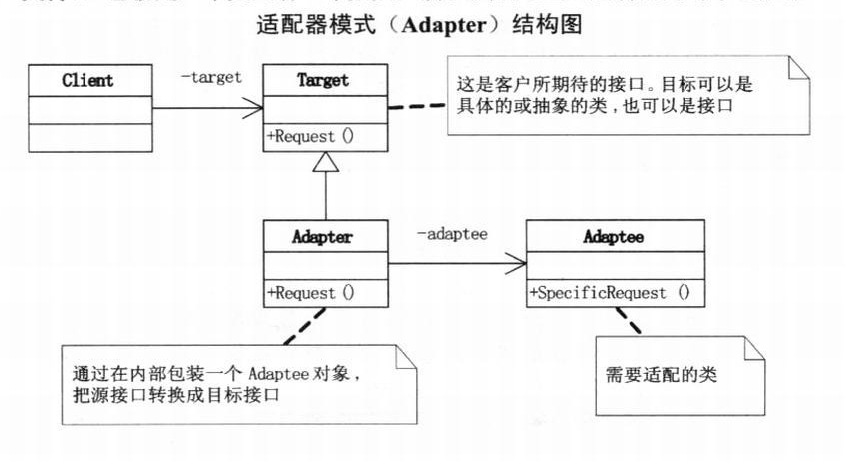
**简介：**

将一个类的接口转换成客户希望的另外一个接口，Adapter模式使得原本由于接口不兼容而不能工作的那些类可以一起工作。

**实现：**

创建一个适配器类Adpter继承（实现）客户需要的接口Target（抽象类或具体类），该类持有一个Adptee类对象（需要适配的类），适配器类通过重写Target类的方法，在方法中调用Adptee的方法来实现Adptee类型向Target类型的转化。

**UML图：**



### 备忘录模式：

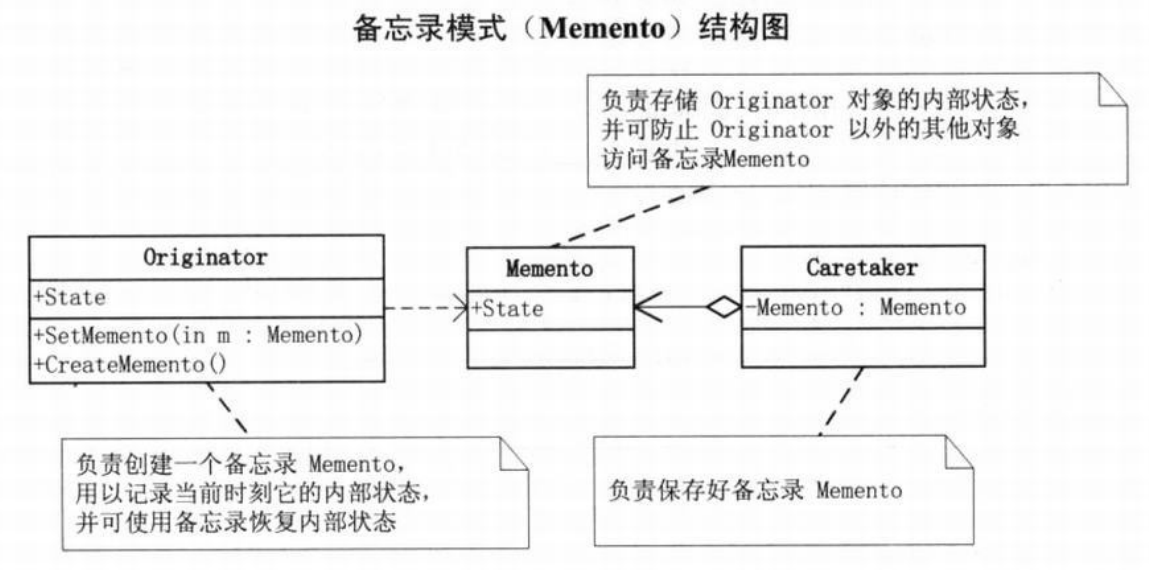
**简介：**

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该状态。这样以后就可以将该对象恢复到以前的状态。

**实现：**

创建一个备忘录类（Memento），保存发起类(Originator)的状态。再创建一个管理者类(Caretake)，用来保存备忘录类。Caretake类不能操作备忘录内容。

**UML图：**



### 组合模式：

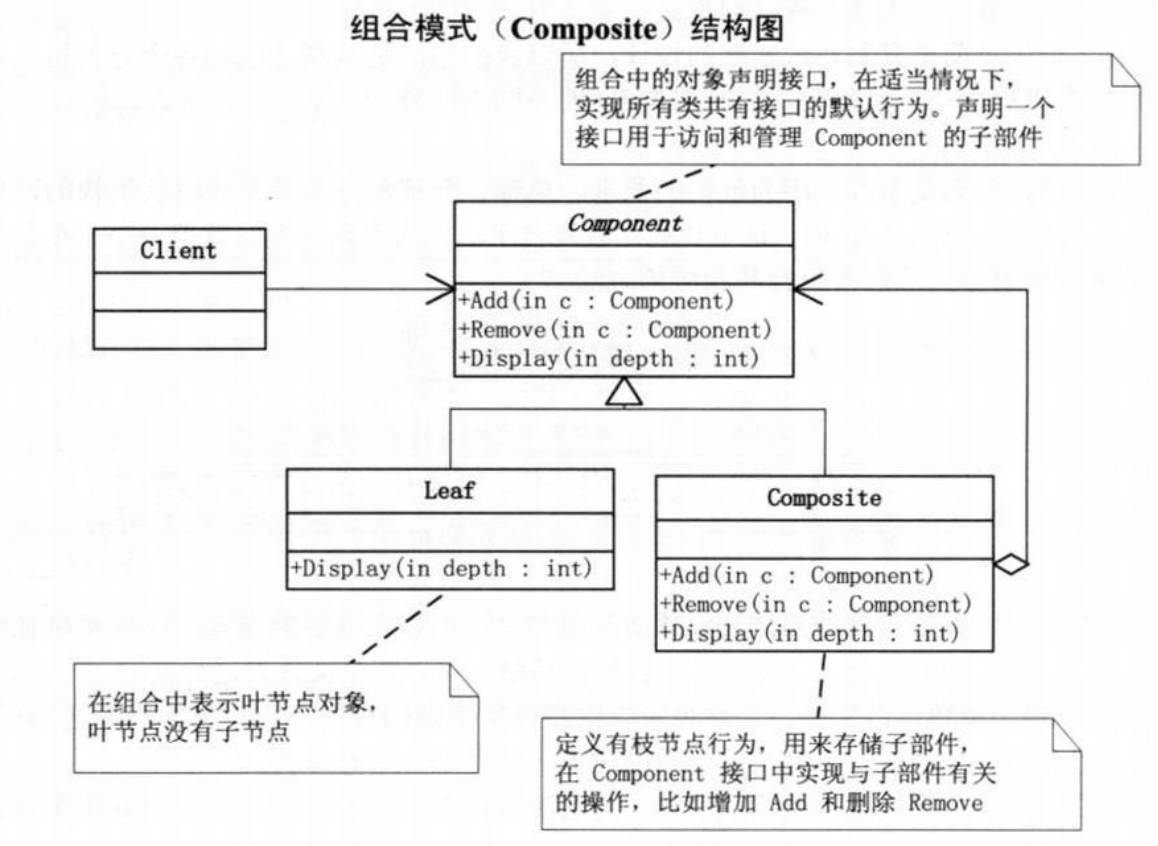
**简介：**

将对象组合成树形结构以表示“部分-整体”的层次结构，组合模式使得用户对单个对象和组合对象的使用性一致。

**实现：**

先设计一个公共接口（抽象类）component，然后设计叶子节点类Leaf和枝节点类composite实现（继承）这个公共接口（抽象类），叶子类不包含子节点，枝类包含子节点（子节点类型就是这些叶子类或者枝类）集合。

**UML图：**



### 迭代器模式：

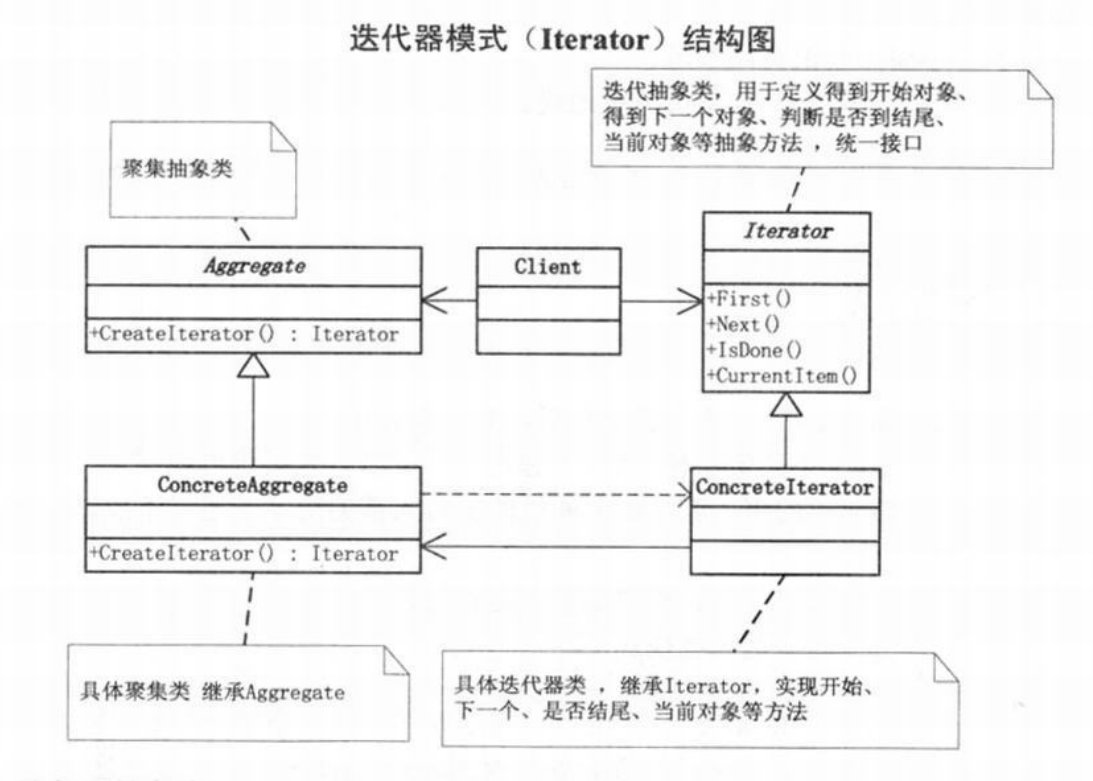
**简介：**

提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露该对象的内部表示。

**实现：**

Java中的foreach遍历以及java.Lang.Iterator类就是迭代器模式的一种使用。

**UML图：**



### 单例模式：

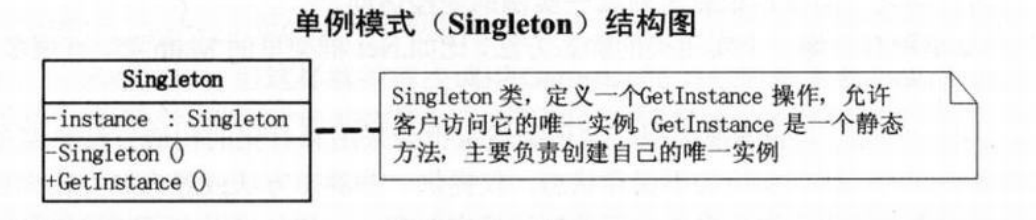
**简介：**

保证一个类只有一个实例，并提供一个访问它的全局访问点。

**实现：**

私有化类构造器，然后持有一个本类类型的静态属性，然后提供一个public方法获得这个静态属性。

**UML图:**



### 桥接模式：

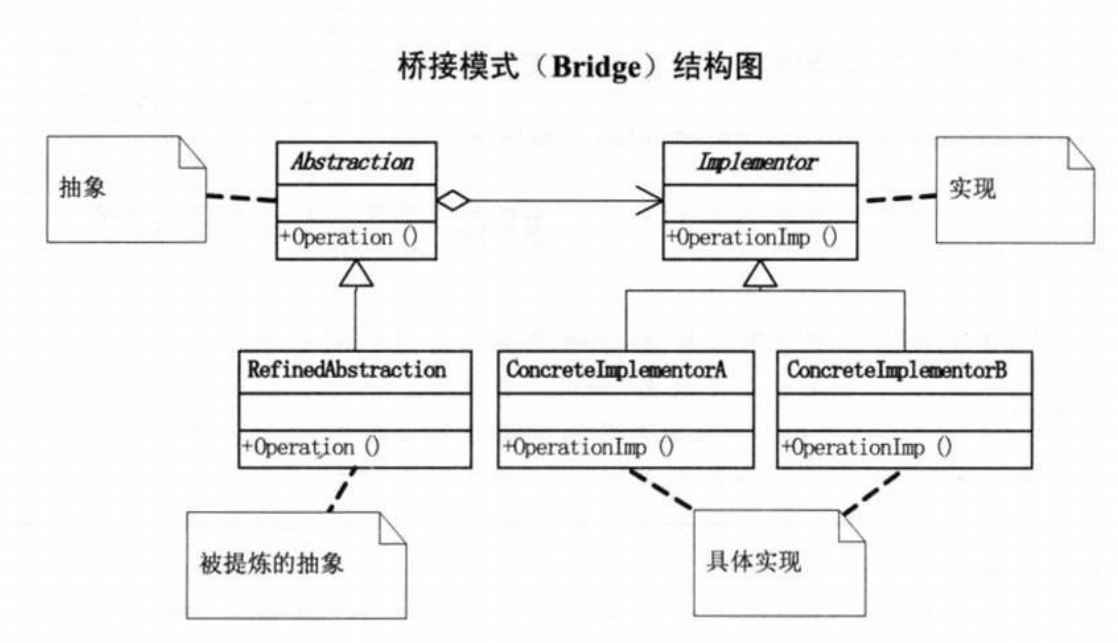
**简介：**

将抽象部分与它的实现部分分离，使他们都可以独立变化。通俗的讲就是实现系统可能有多角度分类，而且每个分类都有可能变化，那么我们应该把这种多角度分离出来让他们独立变化。减少他们之间的耦合

**实现：**

以形状和颜色为例，形状是一种变化，如：圆形，三角形等。颜色也是一种变化，如：白色，蓝色等。按照简单继承的做法，就是各种形状继承形状接口，然后各种颜色的形状分别继承他们各种的形状类，这时候如果要加一种形状，则需要再加对应的各种颜色形状子类，加一种颜色，则需要加各种改颜色形状类，这样做显然不太合理，类之间耦合过多。而按照桥接模式来实现的话，可以将形状和颜色两个变化都独立出来，然后在形状类中只有颜色对象，这样不管是新增颜色还是形状都只需要新增对应的类即可，耦合减少。

UML图：



### 命令模式：

**简介：**

将一个请求封装为一个对象，从而可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。

**实现:**

将系统中的相关操作抽象成命令接口，具体命令实现这个命令接口，命令类持有一个执行者（receiver）对象，执行命令实际上是调用receiver对象的具体方法。定义一个命令的调用者（invoke），它持有一个命令对象或者对象集合，它可以通知命令去执行。

**UML图：**



### 责任链模式：

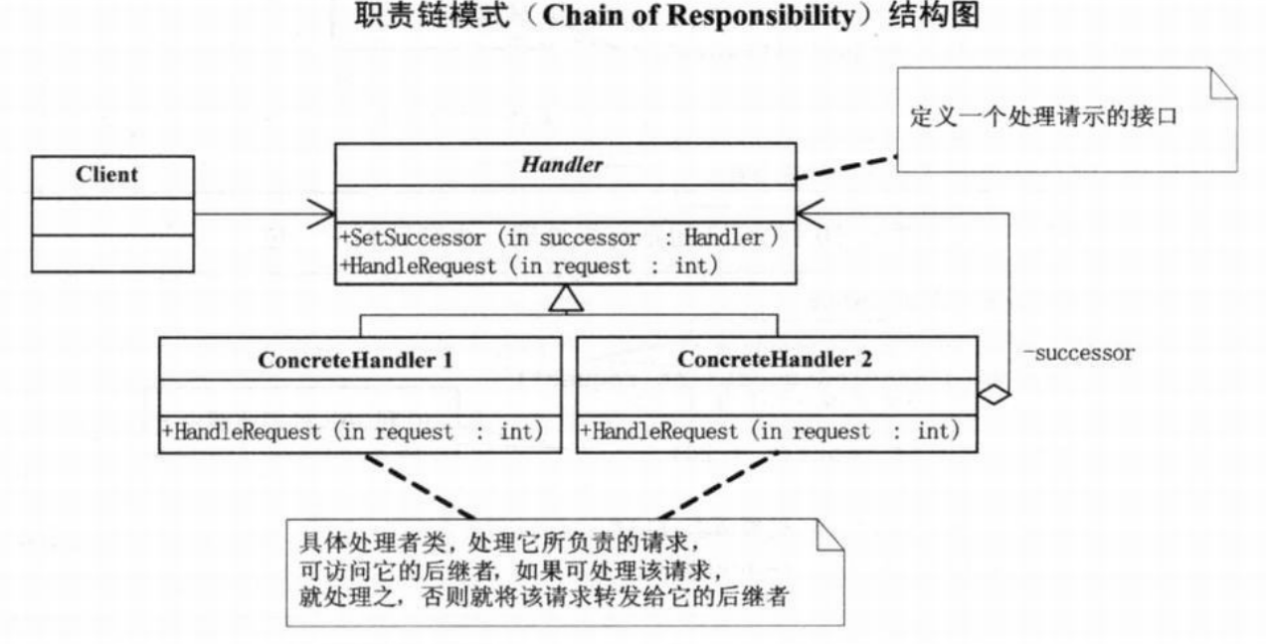
**简介：**

使多个对象都有机会处理请求，从而避免请求的发送者与接收者之间的耦合关系。将这些对象连成一条链，并沿着该链传递请求，直到有对象处理请求。

**实现：**

责任链模式和状态模式类似都可以处理判断分支过多的情况。状态模式在编写时就已经确定了下一个处理对象是谁。而责任链模式则需要在使用时指定下一个处理对象，比如，通过set方法设置。

UML图：



### 中介者模式：

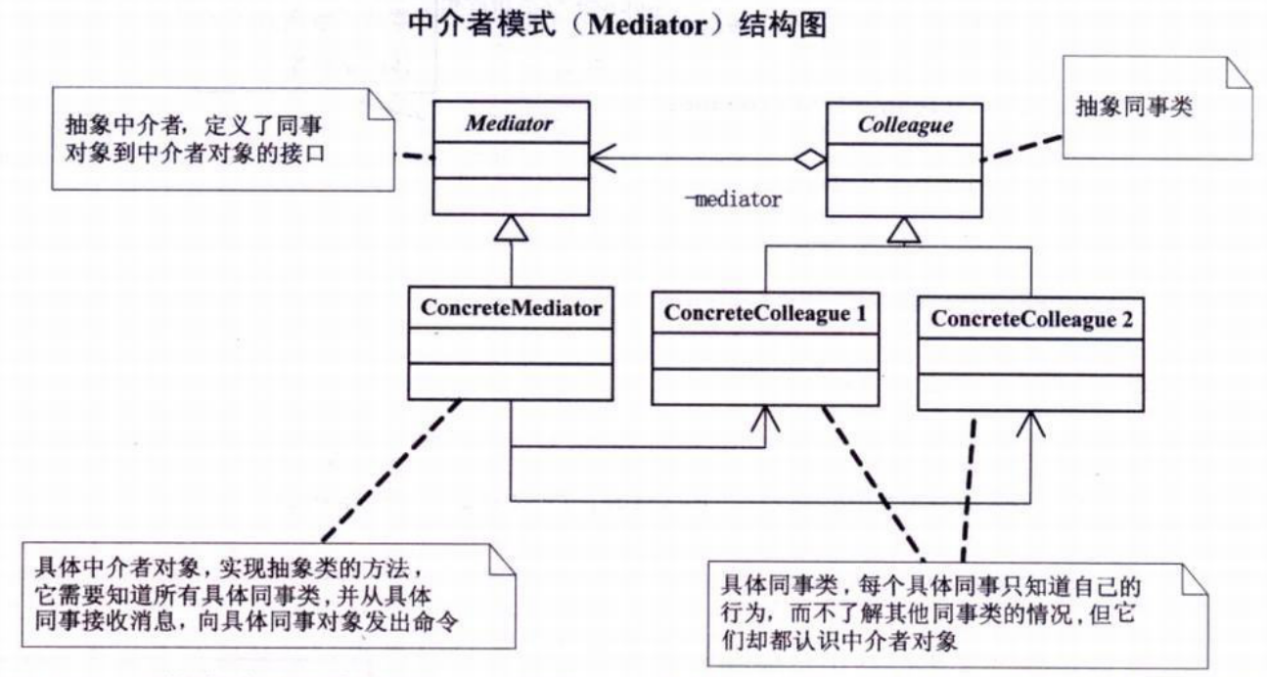
**简介：**

用一个中介对象来封装一系列的对象交互，中介者使各对象之间不需要显式的相互引用，从而使耦合松散。而且可以独立地改变他们之间的交互。

**实现：**

编写一个抽象中介者接口和一个抽象同事类接口。具体的中介者类以及具体的同事类分别实现这两接口。抽象同事类持有一个中介者对象，具体抽象类内需要持有所有需要交互的具体同事类对象，并从具体同事类接收信息和向具体同事类发送消息。

**UML图:**



### 享元模式：

**简介：**

运用共享技术有效的支持大量细粒度的对象。

**实现：**

在java中常见的享元模式有String的常量池和数据库连接池技术。享元模式包括一个享元类的接口，以及具体的享元类，和一个享元工厂，用来创建并管理享元类。

UML图：

