### 简单工厂模式：

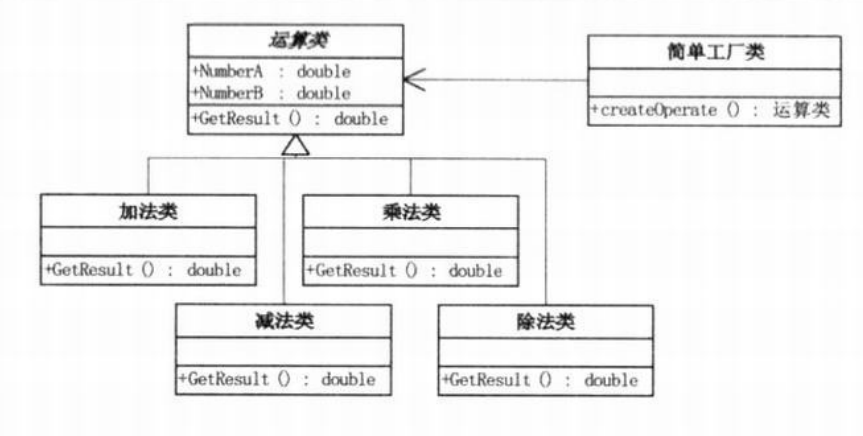
**简介：**

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：

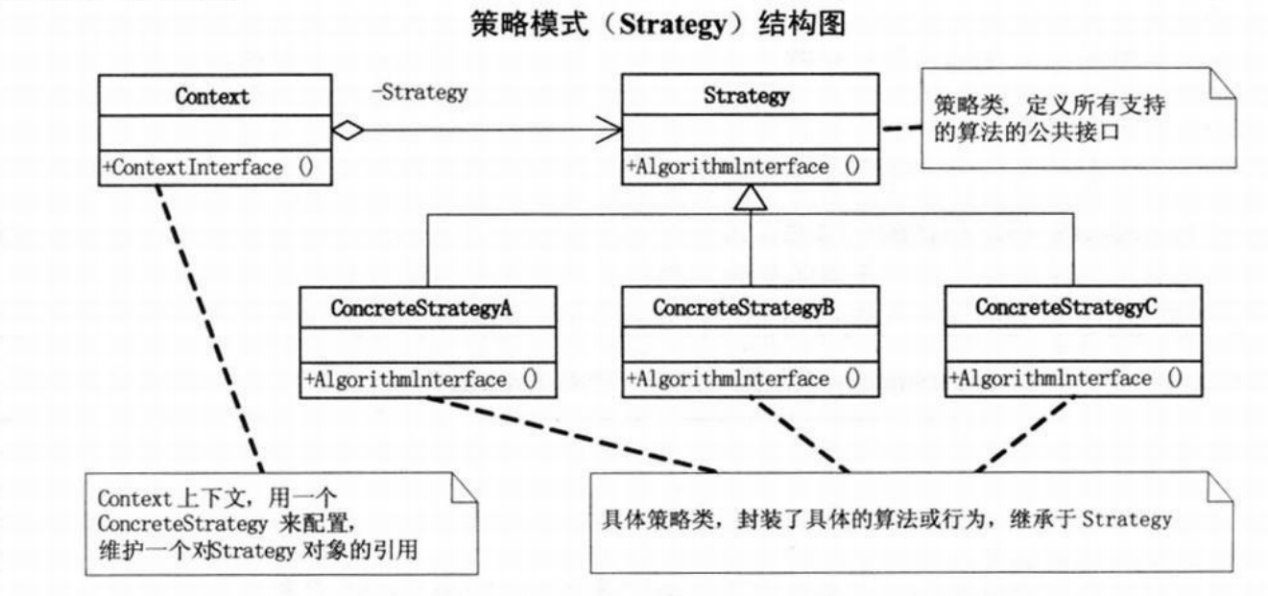
**简介：**

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：

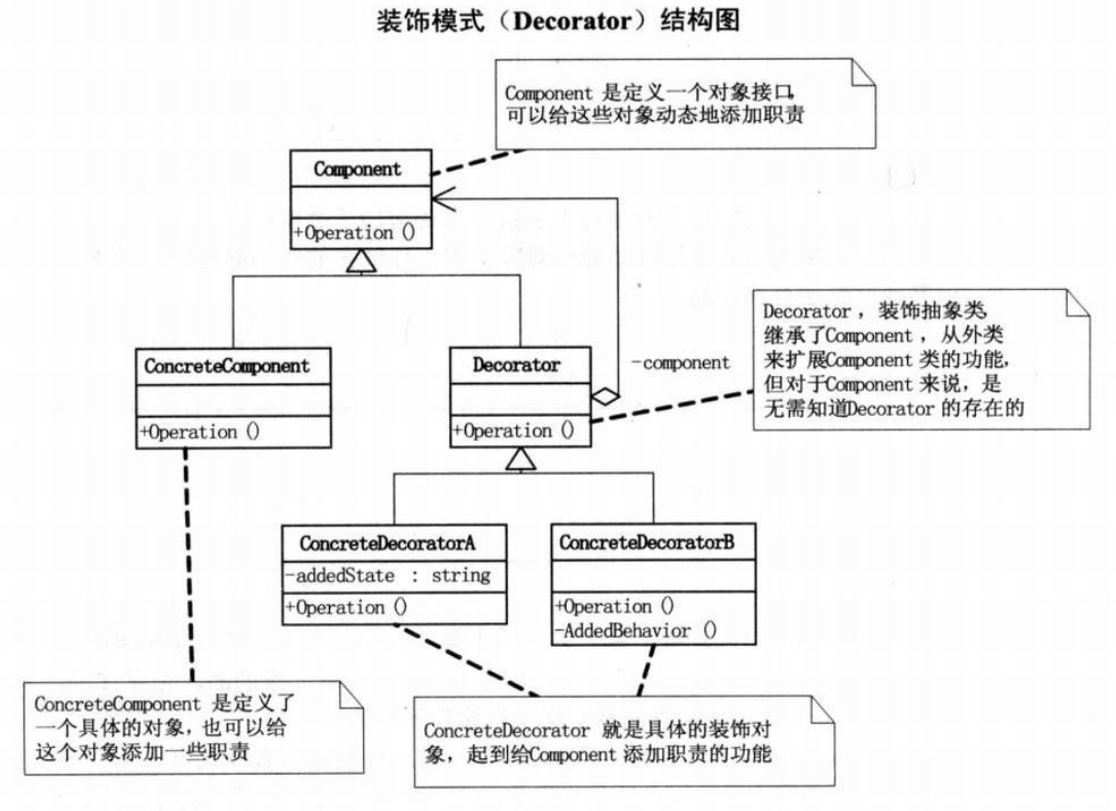
**简介：**

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：

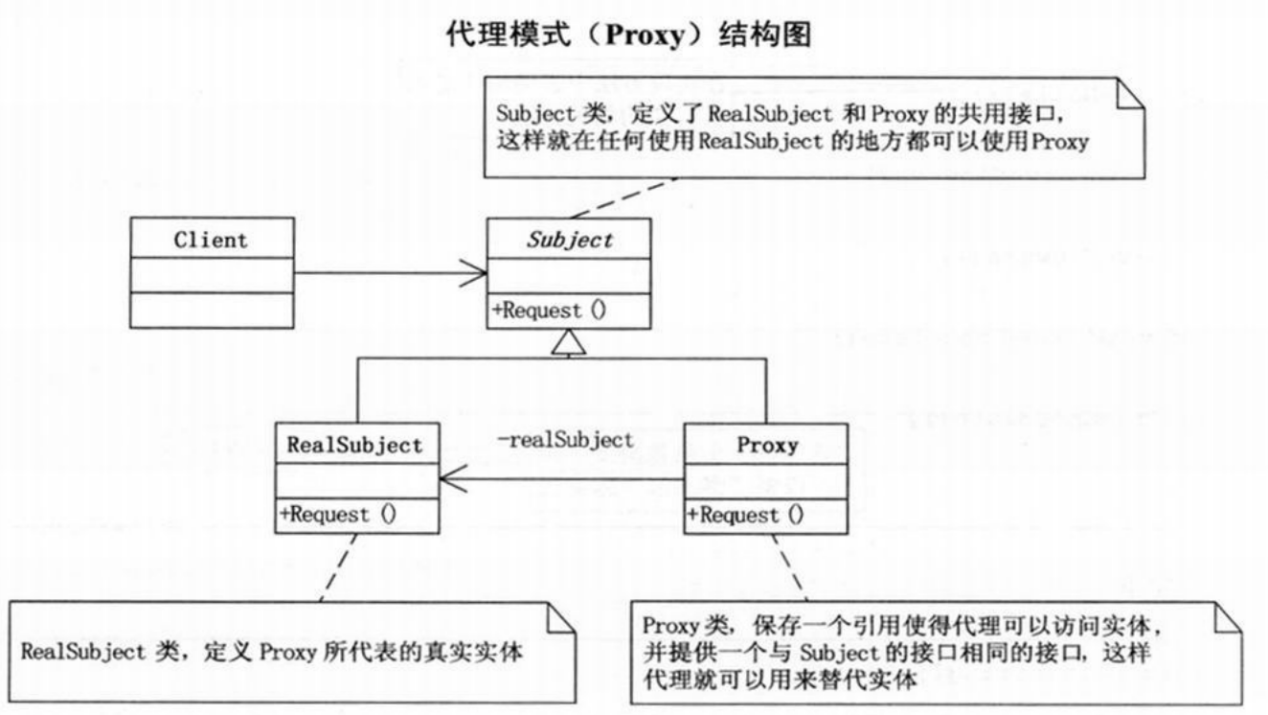
**简介：**

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：

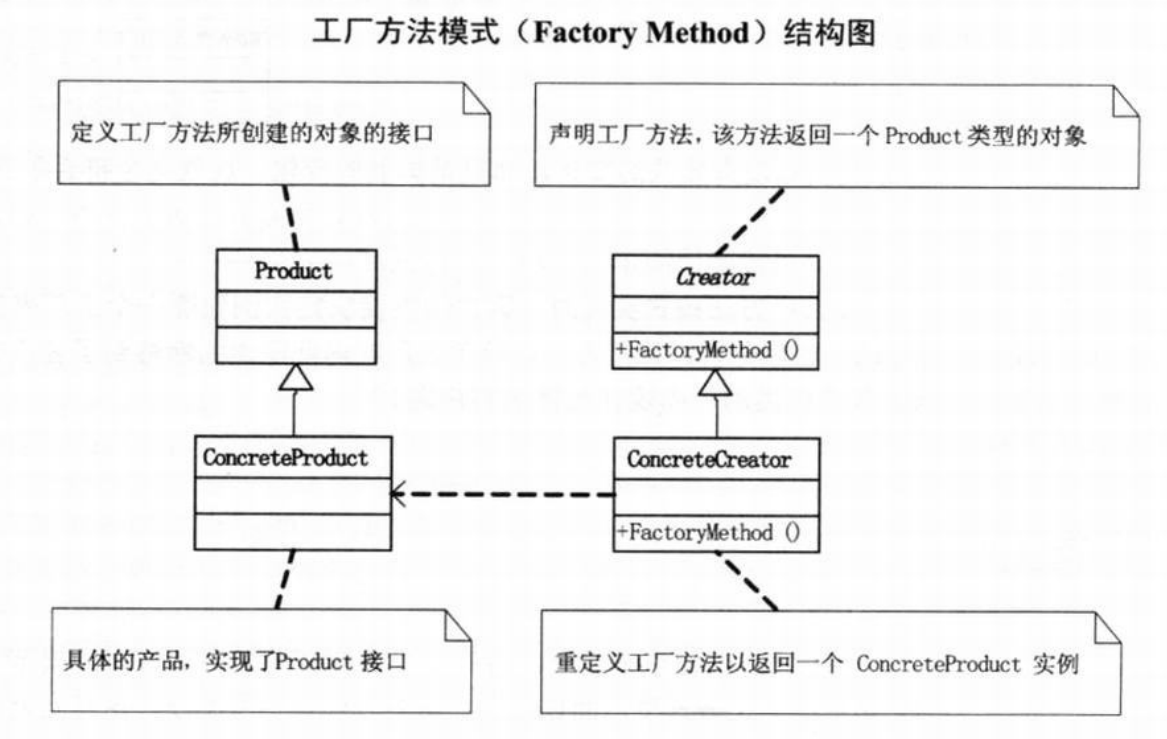
**简介：**

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：

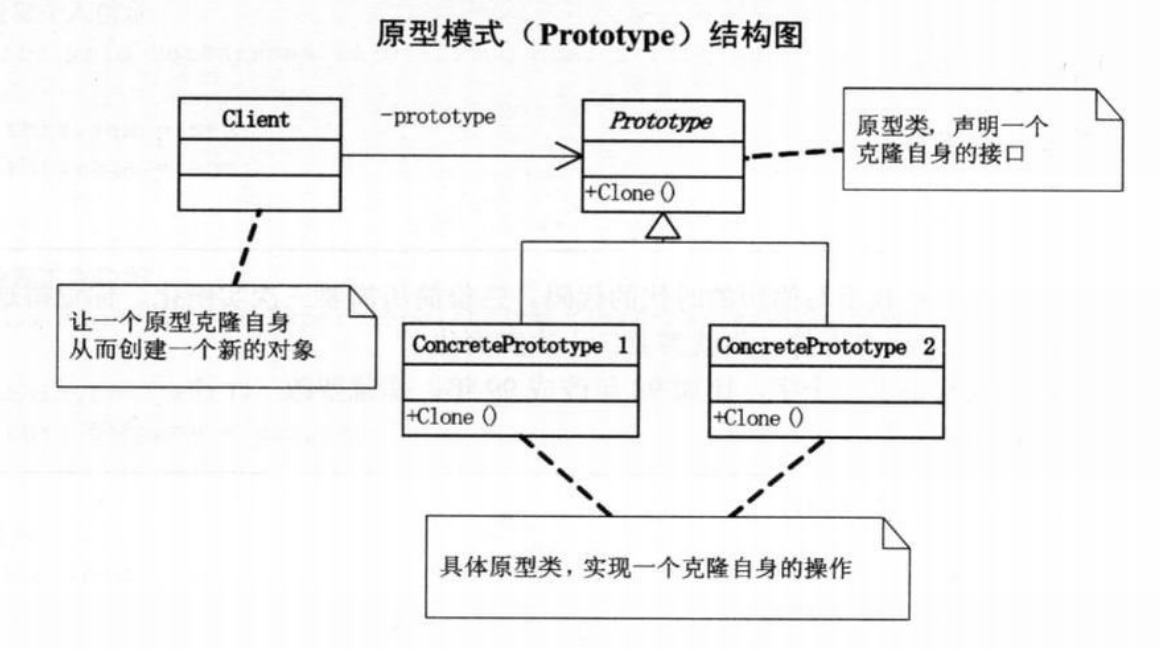
**简介：**

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：

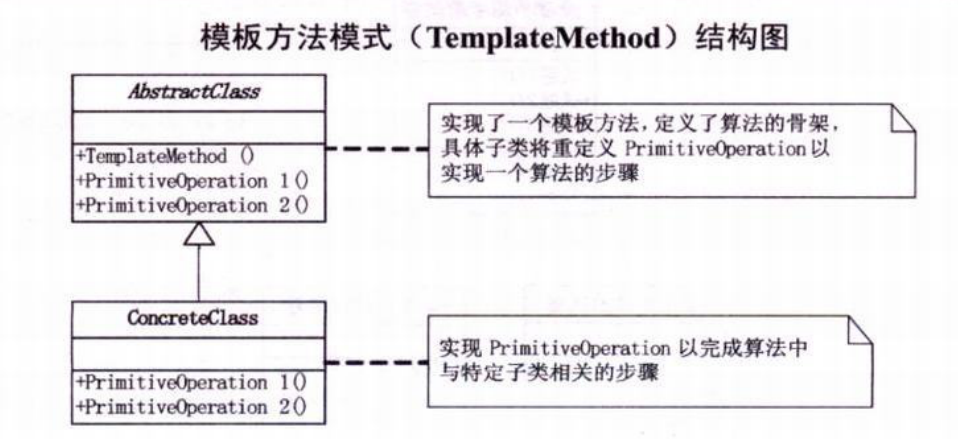
**简介：**

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：

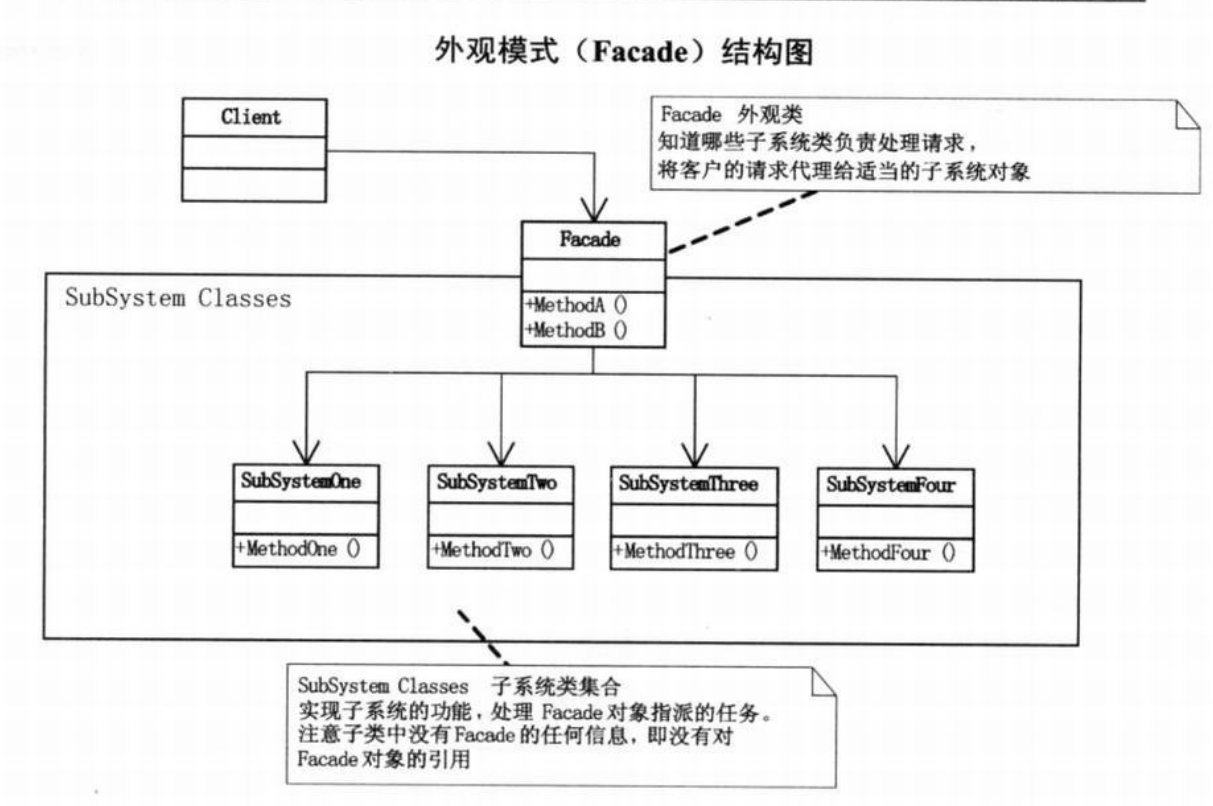
**简介：**

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：

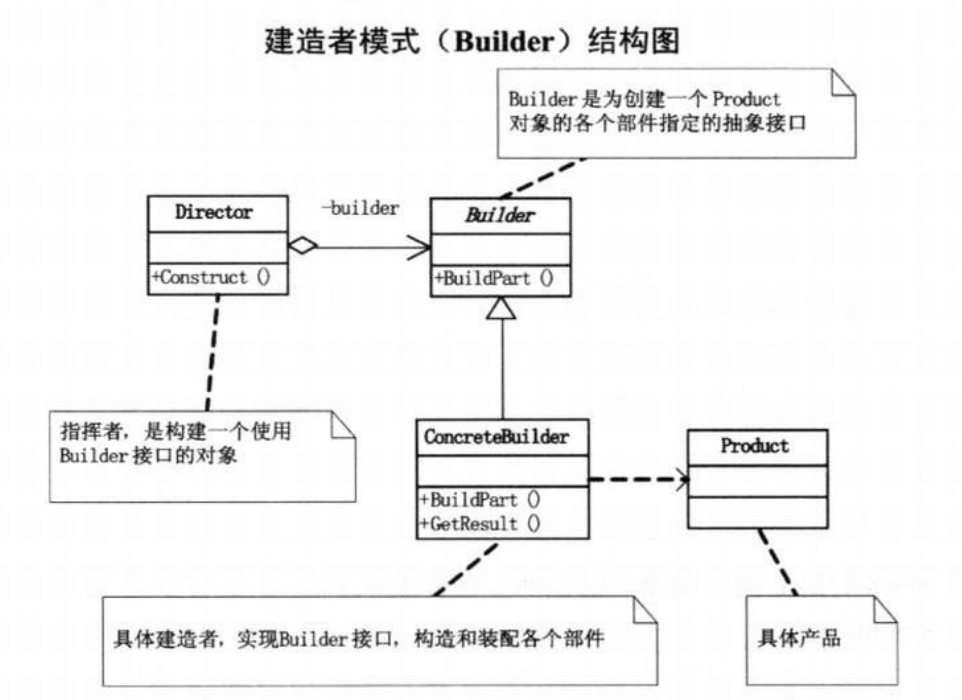
**简介:**

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：

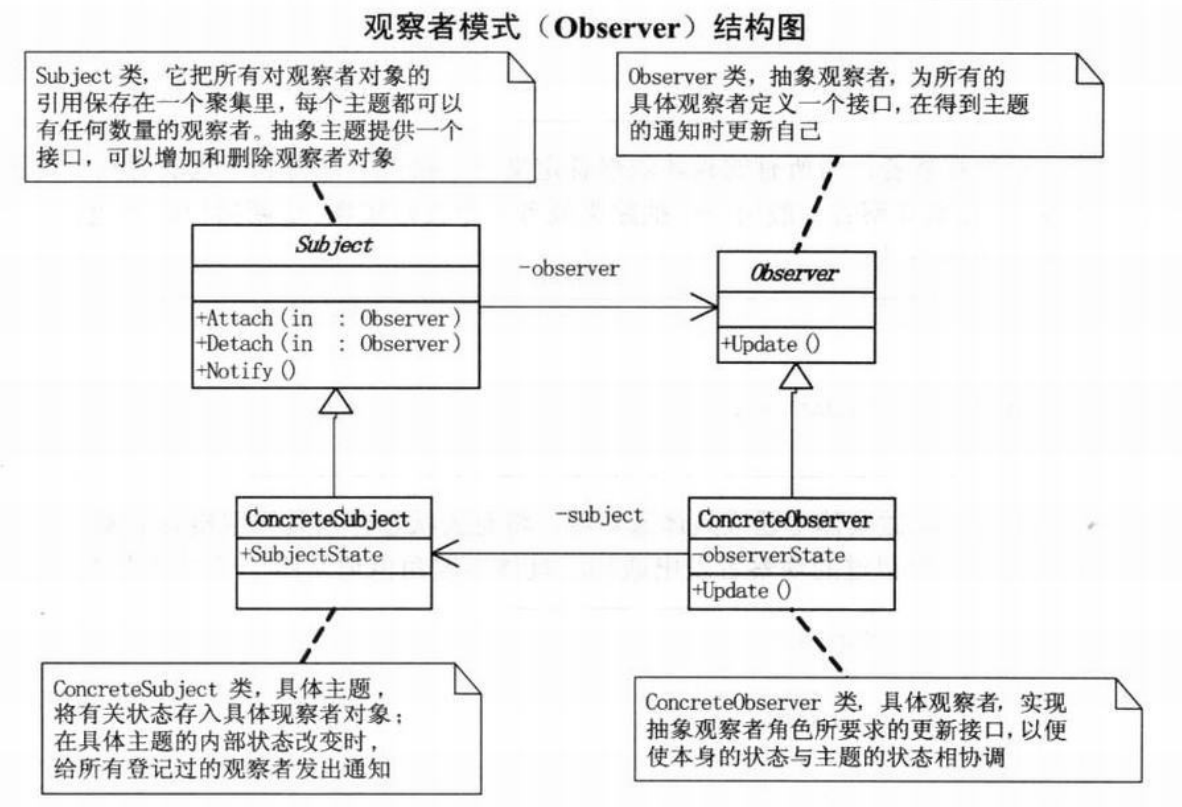
**简介：**

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：

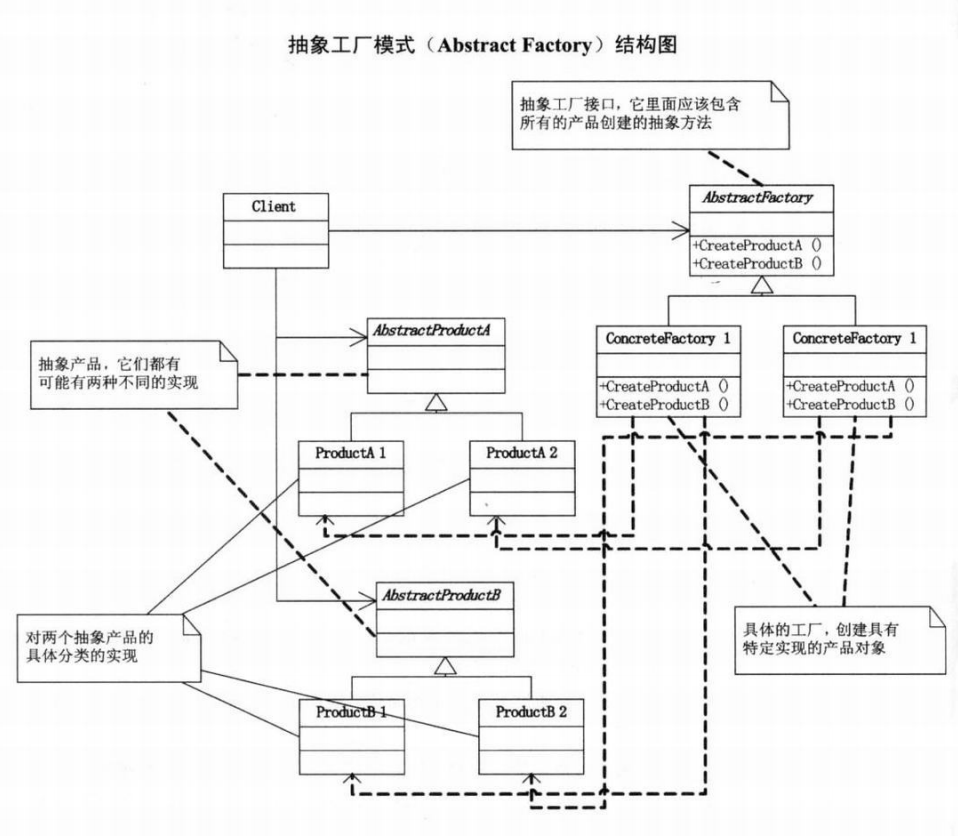
**简介：**

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。

**UML图：**

