1. **设计模式六大原则**

单一职责原则

里氏替换原则

依赖倒置原则

接口隔离原则

迪米特法则

开闭原则

* 1. **单一职责原则**

定义：不要存在多于一个导致类变更的原因。即一个类只负责一项职责。

问题由来：类T负责两个不同职责：职责P1，职责P2。当由于职责P1需求发生改变而需要修改类T时，有可能会导致原本运行正常的职责P2功能发生故障。

解决方案：遵循单一职责原则。分别建立两个类T1、T2，使T1完成职责P1功能，T2完成职责P2功能。这样，当修改类T1时，不会使职责P2发生故障风险；同理，当修改T2时，也不会使职责P1发生故障风险。

* 1. 里氏替换原则

定义：所有引用基类的地方必须能透明地使用其子类的对象。

问题由来：有一功能P1，由类A完成。现需要将功能P1进行扩展，扩展后的功能为P，其中P由原有功能P1与新功能P2组成。新功能P由类A的子类B来完成，则子类B在完成新功能P2的同时，有可能会导致原有功能P1发生故障。

解决方案：当使用继承时，遵循里氏替换原则。类B继承类A时，除添加新的方法完成新增功能P2外，尽量不要重写父类A的方法，也尽量不要重载父类A的方法。

* 1. **依赖倒置原则**

定义：高层模块不应该依赖低层模块，二者都应该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象。

问题由来：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。

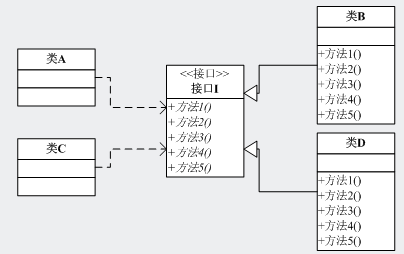
解决方案：将类A修改为依赖接口I，类B和类C各自实现接口I，类A通过接口I间接与类B或者类C发生联系，则会大大降低修改类A的几率。

* 1. **接口隔离原则**

定义：客户端不应该依赖它不需要的接口；一个类对另一个类的依赖应该建立在最小的接口上。

问题由来：类A通过接口I依赖类B，类C通过接口I依赖类D，如果接口I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。

解决方案：将臃肿的接口I拆分为独立的几个接口，类A和类C分别与他们需要的接口建立依赖关系。也就是采用接口隔离原则。



* 1. **迪米特法则**

定义：一个对象应该对其他对象保持最少的了解。

问题由来：类与类之间的关系越密切，耦合度越大，当一个类发生改变时，对另一个类的影响也越大。

解决方案：尽量降低类与类之间的耦合。

* 1. **开闭原则**

定义：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。

问题由来：在软件的生命周期内，因为变化、升级和维护等原因需要对软件原有代码进行修改时，可能会给旧代码中引入错误，也可能会使我们不得不对整个功能进行重构，并且需要原有代码经过重新测试。

解决方案：当软件需要变化时，尽量通过扩展软件实体的行为来实现变化，而不是通过修改已有的代码来实现变化

1. **建造者模式**

建造者模式：是将一个复杂的对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

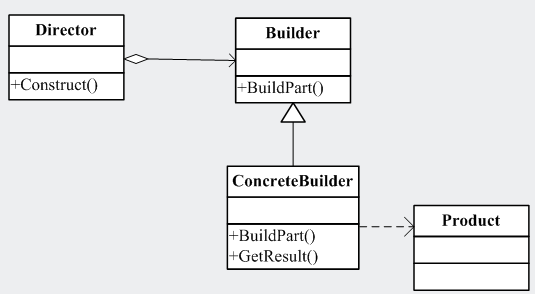
建造者模式通常包括下面几个角色：

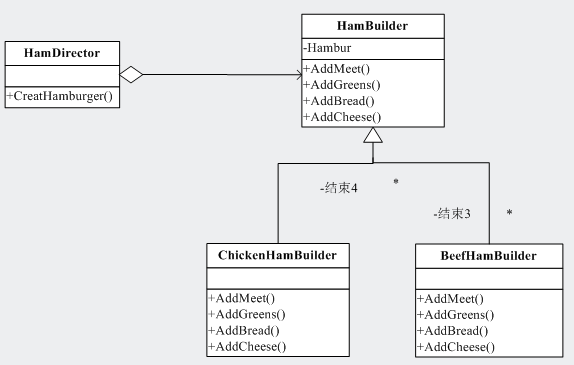
1. builder：给出一个抽象接口，以规范产品对象的各个组成成分的建造。这个接口规定要实现复杂对象的哪些部分的创建，并不涉及具体的对象部件的创建。

2. ConcreteBuilder：实现Builder接口，针对不同的商业逻辑，具体化复杂对象的各部分的创建。 在建造过程完成后，提供产品的实例。

3. Director：调用具体建造者来创建复杂对象的各个部分，在指导者中不涉及具体产品的信息，只负责保证对象各部分完整创建或按某种顺序创建。

4. Product：要创建的复杂对象。





使用建造者模式的好处：

1.使用建造者模式可以使客户端不必知道产品内部组成的细节。

2.具体的建造者类之间是相互独立的，对系统的扩展非常有利。

3.由于具体的建造者是独立的，因此可以对建造过程逐步细化，而不对其他的模块产生任何影响。

使用建造模式的场合：

1.创建一些复杂的对象时，这些对象的内部组成构件间的建造顺序是稳定的，但是对象的内部组成构件面临着复杂的变化。

2.要创建的复杂对象的算法，独立于该对象的组成部分，也独立于组成部分的装配方法时。

1. **装饰模式**

装饰模式(Decorator)，动态地给一个对象添加一些额外的职责，就增加功能来说，装饰模式比生成子类更加灵活。

装饰模式分为3个部分：

1. 抽象组件 -- 对应Coffee类

2. 具体组件 -- 对应具体的咖啡，如：Decaf，Espresso

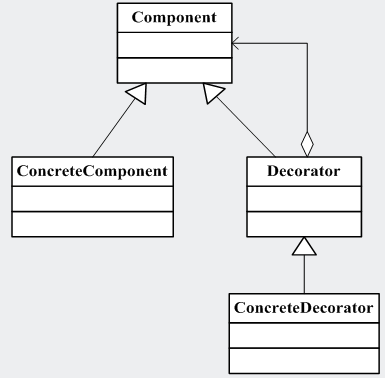
3. 装饰者 -- 对应调味品，如：Mocha，Whip

装饰模式有3个特点：

1. 具体组件和装饰者都继承自抽象组件(Decaf、Espresson、Mocha和Whip都继承自Coffee)，并且装饰者持有抽象组件的引用

2. 可以使用装饰者组合具体组件创造出新的类(Mocha组合Decaf创造出MochaDecaf)

3. 过程2可以重复，直到创造出需要的类



1. **策略模式**

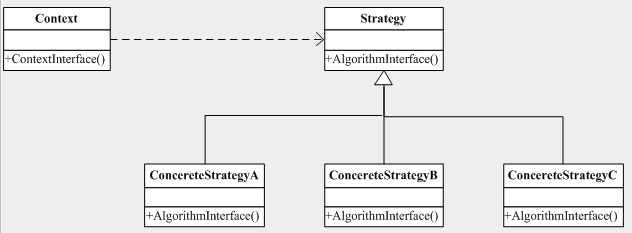
它定义了算法家族，分别封装起来，让它们之间可以互相替换，此模式算法的变化，不会影响到使用算法的客户。

策略模式通常包括下面几个角色：

1. Strategy：抽象策略类，通常由一个接口或者抽象类实现。

2. ConcereteStrategy：具体策略角色，包装了相关的算法和行为。

3. Context：环境角色，持有一个策略类的引用，最终给客户端调用。



**2.3 使用场景**

策略模式的好处：

相关算法系列 Strategy类层次为Context定义了一系列的可供重用的算法或行为。 继承有助于析取出这些算法中的公共功能。

提供了可以替换继承关系的办法

可以避免使用多重条件转移语句

使用策略模式的场合：

多个类只区别在表现行为不同，可以使用Strategy模式，在运行时动态选择具 体要执行的行为。

需要在不同情况下使用不同的策略(算法)，或者策略还可能在未来用其它方式来实现。

对客户隐藏具体策略(算法)的实现细节，彼此完全独立。

1. **观察者模式**

观察者模式(Observer)，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。这个主题对象在状态发生变化时，会通知所有观察者对象，使它们能够自动更新自己。

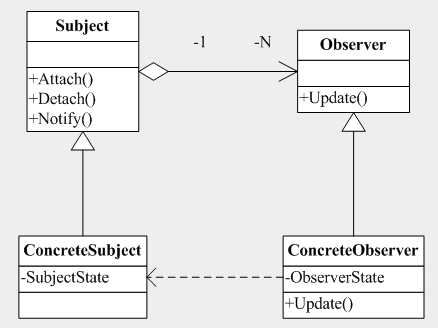
观察者模式4个角色：

1. 抽象主题（Subject）：它把所有观察者对象的引用保存到一个聚集里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。

2. 具体主题（ConcreteSubject）：将有关状态存入具体观察者对象；在具体主题内部状态改变时，给所有登记过的观察者发出通知。

3. 抽象观察者（Observer）：为所有的具体观察者定义一个接口，在得到主题通知时更新自己。

4. 具体观察者（ConcreteObserver）：实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题状态协调。



1. **桥接模式**

桥接模式(Bridge)，将抽象部分与它的实现部分分离，使它们都可以独立地变化

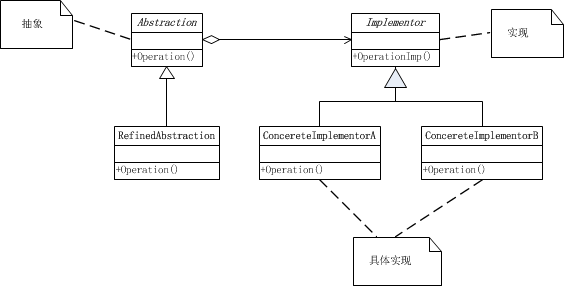
4个角色：

1. 抽象化角色（Abstraction）：抽象化给出的定义，并保存一个对实现化对象的引用。

2. 扩展抽象化（Refined Abstraction）：扩展抽象化角色，改变和修正父类对抽象化的定义。

3. 实现化角色（Implementor）：这个角色给出实现化角色的接口，但不给出具体的实现。

4. 具体实现化角色（Concrete Implementor）：这个角色给出实现化角色接口的实现。



优点：

分离抽象接口及其实现部分。提高了比继承更好的解决方案。

桥接模式提高了系统的可扩充性，在两个变化维度中任意扩展一个维度，都不需要修改原有系统。

实现细节对客户透明，可以对用

缺点：

桥接模式的引入会增加系统的理解与设计难度，由于聚合关联关系建立在抽象层，要求开发者针对抽象进行设计与编程

桥接模式要求正确识别出系统中两个独立变化的维度，因此其使用范围具有一定的局限性。

使用桥接模式的场合：

如果一个系统需要在构件的抽象化角色和具体化角色之间增加更多的灵活性，避免在两个层次之间建立静态的继承联系，通过桥接模式可以使它们在抽象层建立一个关联关系。

对于那些不希望使用继承或因为多层次继承导致系统类的个数急剧增加的系统，桥接模式尤为适用。

一个类存在两个独立变化的维度，且这两个维度都需要进行扩展。

1. **命令模式**

命令模式(Bridge)，将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化：对请求队列或记录请求日志，以及支持可撤销的操作。

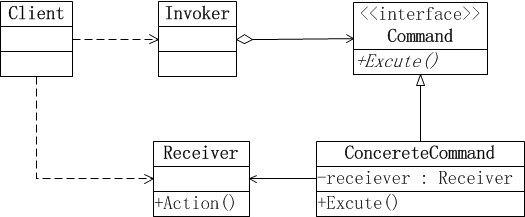
命令模式4个角色：

1、 Command ：定义命令的统一接口。

2、ConcreteCommand ： Command接口的实现者，用来执行具体的命令，某些情况下可以直接用来充当Receiver。

3、 Receiver：命令的实际执行者。

4、Invoker：命令的请求者，是命令模式中最重要的角色。这个角色用来对各个命令进行控制。



使用命令模式的场合：

命令的发送者和命令执行者有不同的生命周期。命令发送了并不是立即执行。

命令需要进行各种管理逻辑。

需要支持撤消\重做操作。