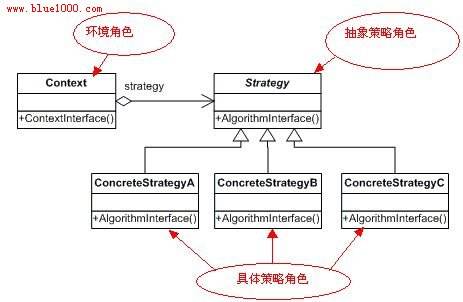
设计原则：

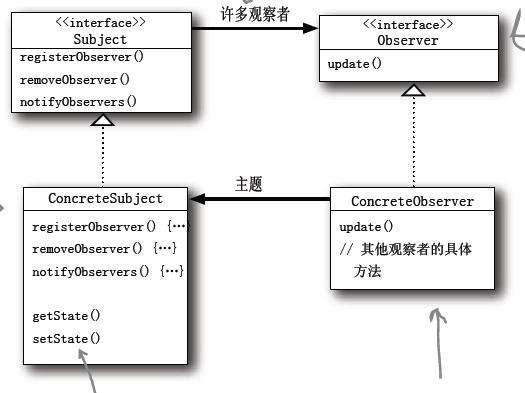
1. 找出应用中可能需要变化之处，把他们独立出来，不要和那些不需要变化的代码混在一起（开放封闭原则）把会变化的部分取出并封装起来，以便以后可以轻易地改动或扩充此部分，而不影响不需要变化的其他部分。所有的模式都提供了一套方法让“系统中的某部分改变不会影响其他部分”。
2. 针对接口编程而不是针对实现编程
3. 多用组合，少用继承
4. 为了交互对象之间的搜耦合设计而努力
5. 类应该对拓展开放，对修改关闭
6. 要依赖抽象，不要依赖具体类（依赖倒置原则）
7. 单一职责原则的优点：类的复杂性低；可读性高；可维护性高；变更风险低
8. 里氏替换原则：

设计模式：

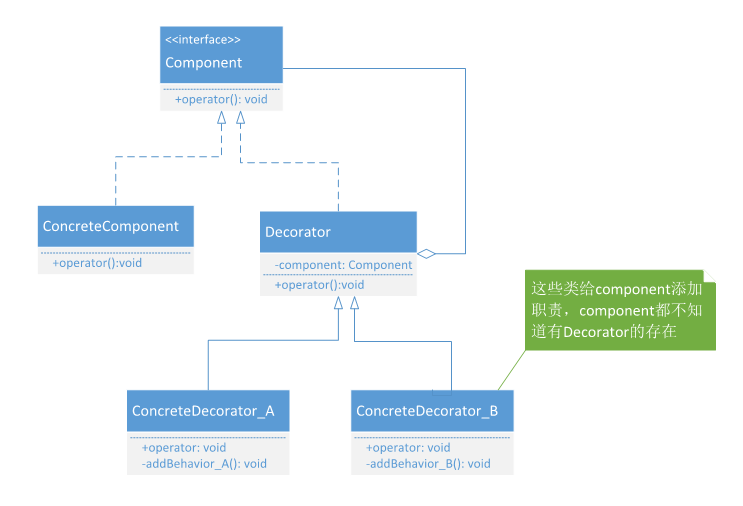
1. 策略模式定义了算法族，分别封装起来，让他们之间可以互相替换，此模式让算法的变化独立于使用算法的客户。



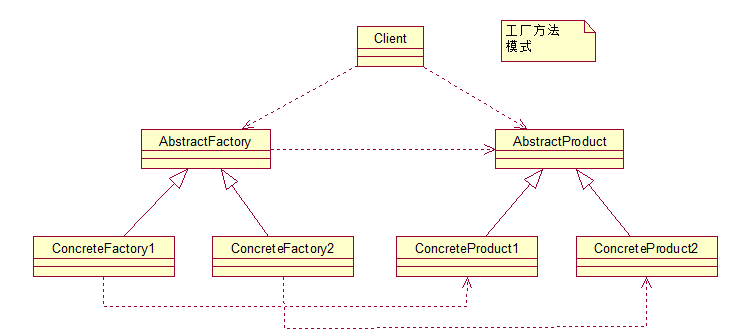
1. 观察者模式定义了对象之间的一对多依赖，这样一来，当一个对象改变状态时，它的所有依赖者都会受到通知并自动更新。



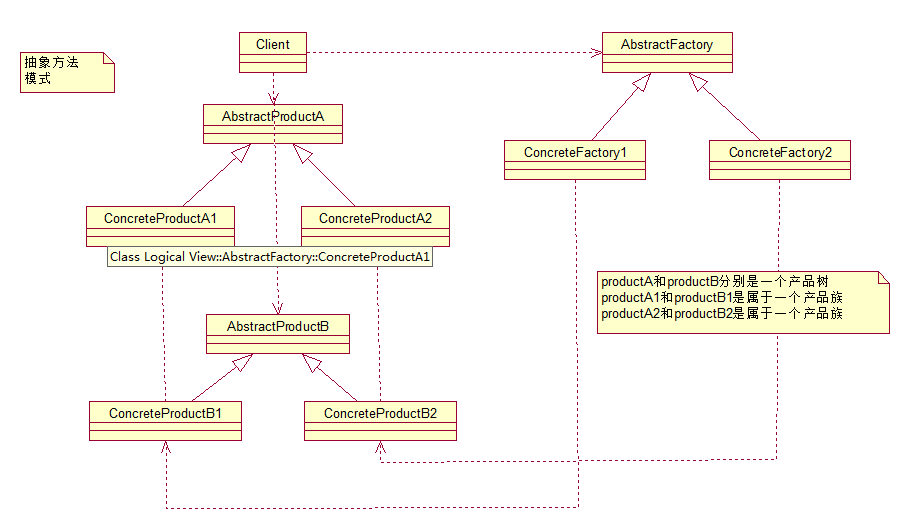
1. 装饰者模式动态地将责任附加到对象上。若要拓展功能，装饰者提供了比继承更有弹性的替代方案



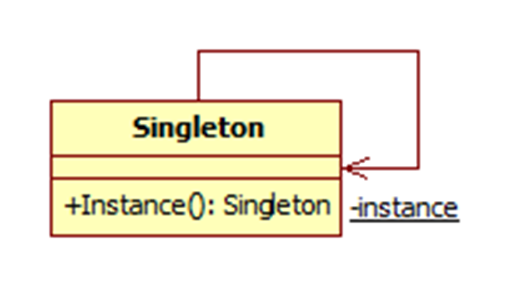
1. 工厂方法模式定义了一个创建对象的接口，但子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类



5、抽象工厂模式提供一个借口，用于创建相关或依赖对象的家族，而不需要明确指定具体类



1. 单例模式确保一个类只有一个实例，并提供一个全局访问点



单例模式的优点：

1. 由于只有一个实例，减少内存开支，减少系统性能的开销，避免对资源的多重占用。
2. 对唯一实例的受控访问，因为Singleton类封装它的唯一实例，所以可以严格的控制客户怎样以及何时访问它
3. 缩小名空间，Singleton模式是对全局变量的一种改进。它避免了哪些存储唯一实例的全局变量污染名空间。
4. 跨平台：使用合适的中间件，可以把singleton模式拓展为跨多个JVM和多个计算机工作
5. 适用于任何类：只需要把一个类的构造函数变成私有的，并且在其中增加相应的静态函数和变量，就可以把这个类变为SINGLETON
6. 延迟求职：如果SINGLETON从未使用过，那么就不会创建它

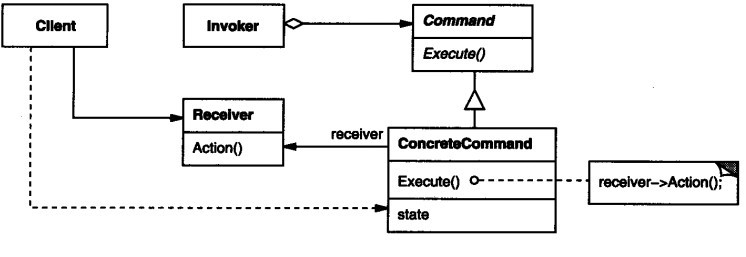
单例模式的缺点：

1. 摧毁方法未定义：没有好的方法去摧毁一个SINGLETON，或者解除其职责。
2. 不能继承：
3. 效率问题：每次调用instance方法都会执行if语句。
4. 不透明
5. 单例模式一般没有接口，拓展困难。

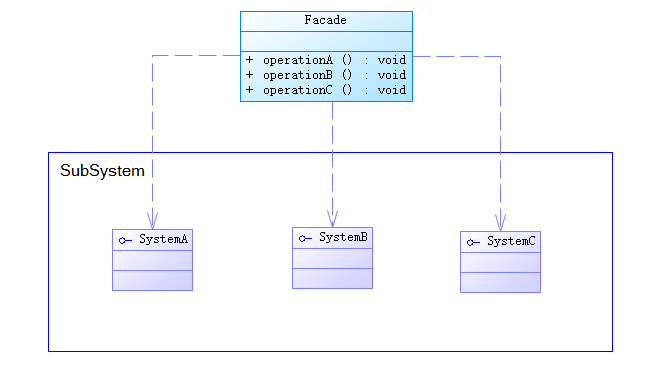
适用场景：

* 当类只能有一个实例而且客户可以从一个众所周知的访问点访问时
* 当这个唯一实例应该是通过子类化拓展的，并且客户应该无需要更改代码就能使用一个拓展的实例时
* 要求生成唯一序列号的环境
* 在整个项目中需要一个共享方位点或共享数据。
* 创建一个对象需要消耗的资源过多，如访问IO和数据库等资源
* 需要定义大量的静态常量和静态方法的环境

1. 命令模式将"请求"封装成对象，以便使用不同的请求、队列或者日志来参数化其他对象。命令模式也支持可撤销的操作



1. 适配器模式将一个类的接口，转换成客户期望的另一个接口。适配器让原本接口不兼容的类可以合作无间（必须是面向接口的编程才合适）
2. 外观模式提供了一个统一的接口，用来访问子系统中的一群接口。外观定义了一个高层接口，让子系统更容易使用。



1. 模板方法模式在一个方法中第一个算法的骨架，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。

优点：封装不变部分，拓展可变部分。

提取公共部分代码便于维护

行为由父类控制，子类实现

使用场景：多个子类有共有的方法，并且逻辑基本相同

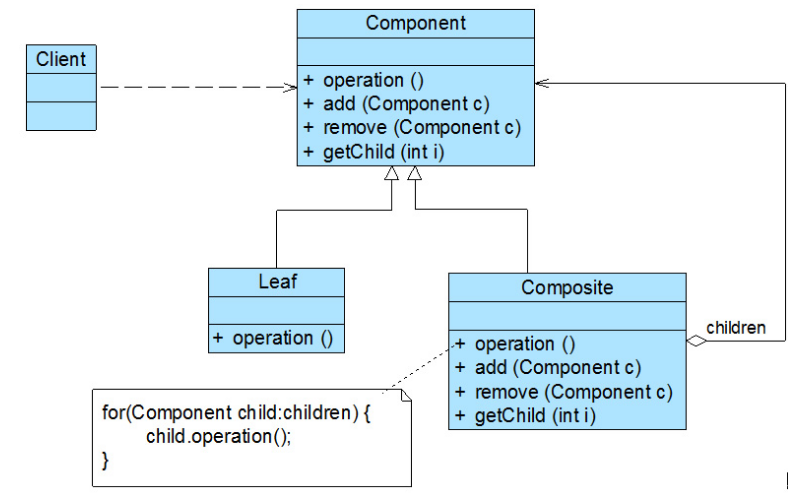
重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现

重构时，模板方法是一个经常使用的模式，把相同的代码抽取到父类中，然后通过钩子函数约束其行为

1. 迭代器模式提供一种方法顺序访问一个聚合对象中的各个元素，而又不暴露其内部的表示
2. 原型模式：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优点：性能优良；逃避构造函数的约束

使用场景：资源优化场景；性能和安全要求的场景；一个对象多个修改者的场景

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



使用场景：

**1.** 在具有整体和部分的层次结构中，希望通过一种方式忽略整体与部分的差异，客户端可以一致地对待它们。   
**2.**  在一个使用面向对象语言开发的系统中需要处理一个树形结构。   
**3**. 在一个系统中能够分离出叶子对象和容器对象，而且它们的类型不固定，需要增加一些新的类型

组合模式的优点：

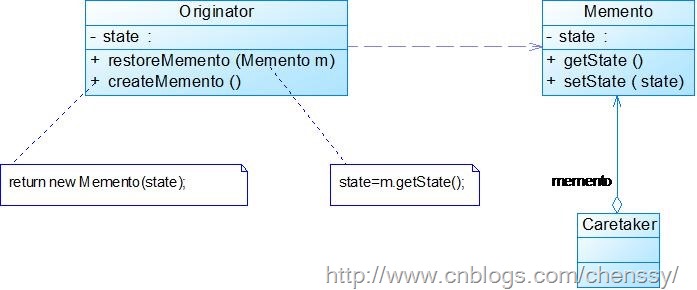
* 高层模块调用简单

一棵树形机构中的所有节点都是Component，局部和整体对调用者来说没有任何区别，要就是说，高层模块不必关系自己处理的是单个对象还是整个组合结构，简化了高层模块的代码

* 节点自由增加

使用了组合模式后，我们可以看看，如果想增加一个树枝节点、树叶节点都很容易，只要找到它的父节点就可以，非常容易拓展，符合开闭原则，对以后的维护非常有利。

13、备忘录模式：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。这样以后就可将对象恢复到原先保存的状态。



Originator发起人角色

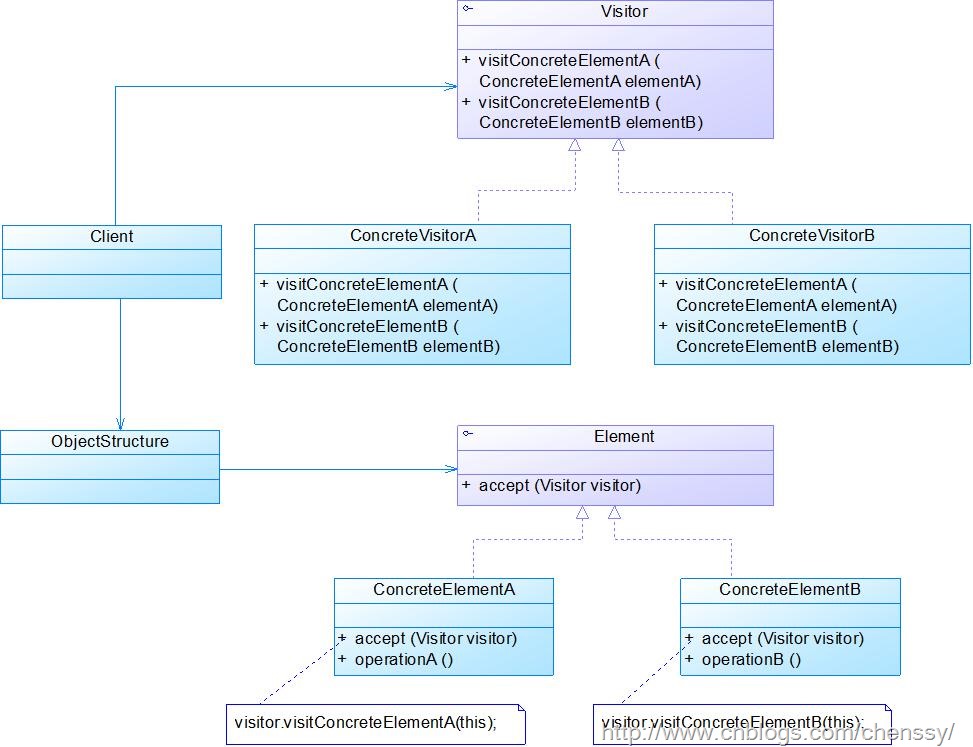
记录当前时刻的内部状态，负责定义哪些属于备份范围的状态。负责创建和恢复备忘录数据。

Memento备忘录角色

负责存储Originator发起人对象的内部状态，在需要的时候提供发起人需要的内部状态。

Caretaker备忘录管理员角色

对备忘录进行管理、保存和提供备忘录

14、访问者模式：封装一些作用于某种数据结构中的各元素的操作，他可以在不改变数据结构的前提下定义作用域这些元素的新的操作访问者模式的优点：

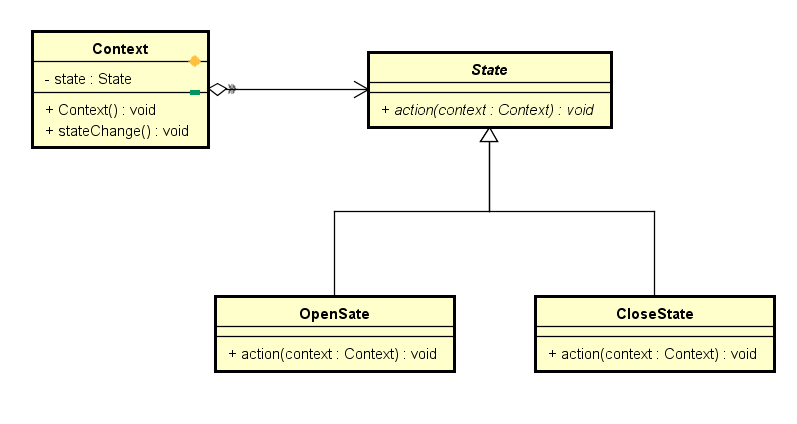
* 符合单一职责原则
* 优秀的拓展性：由于职责分开，继续增加对数据的操作是非常快捷的
* 灵活性非常高

访问者模式的缺点：

* 具体元素对访问者公布细节
* 具体元素变更比较困难
* 违背了依赖倒置原则

使用场景

* 一个对象结构包含很多类对象，他们有不同的接口。而你相对这些对象实施一些依赖于其具体类的操作，也就是说用迭代器已经不能胜任的情景
* 需要对一个对象结构中的对象进行很多不同并且不相关的操作，而你想避免让这些操作污染这些对象的类

15、状态模式：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

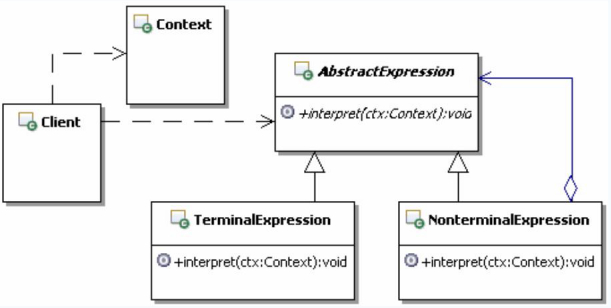
状态模式的优点：

* 结构清晰，避免了过多的switch…case或者if…else语句的使用，避免了程序的复杂性，提高系统的可维护性
* 遵循设计原则，很好地体现了开闭原则和单一职责原则，每个状态都是一个子类，你要增加状态就要增加子类，要修改状态，只需要修改一个子类
* 封装性好，这也是状态模式的基本要求，状态变化放置到类的内部来实现，外部的调用不用知道类内部如何实现状态和行为的变换

状态模式的缺点：会产生过多的子类，即发生类膨胀。

使用场景：

* 行为随状态改变而改变的场景
* 条件、分支判断语句的替代者

16、解释器模式：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

17、享元模式：使用共享对象可有效地支持大量的细粒度对象

