**设计模式的六大原则**

1、开闭原则（Open Close Principle）

开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于维护和升级。想要达到这样的效果，我们需要使用接口和抽象类。

2、里氏代换原则（Liskov Substitution Principle）

里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原则中说，任何基类可以出现的地方，子类一定可以出现。 LSP是继承复用的基石，只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基类与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。

3、依赖倒转原则（Dependence Inversion Principle）

这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。

4、接口隔离原则（Interface Segregation Principle）

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

5、迪米特法则（最少知道原则）（Demeter Principle）

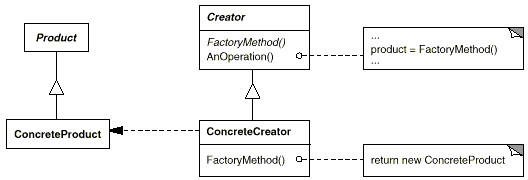
为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）

原则是尽量使用合成/聚合的方式，而不是使用继承。

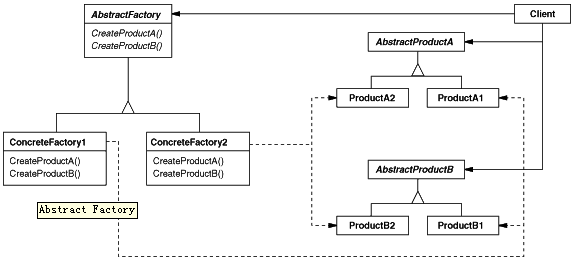
**创建型**

1. Factory Method（工厂方法）



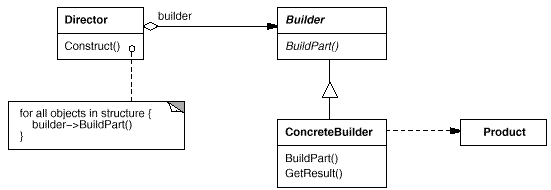
|  |  |  |  |
| --- | --- | --- | --- |
| 意图 | 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method 使一个类的实例化延迟到其子类。 | | |
| 适用性 | 当一个类不知道它所必须创建的对象的类的时候。当一个类希望由它的子类来指定它所创建的对象的时候。当类将创建对象的职责委托给多个帮助子类中的某一个，并且你希望将哪一个帮助子类是代理者这一信息局部化的时候。 | | |
| 主要  解决 | 接口选择的问题 | 如何  解决 | 让其子类实现工厂接口，返回的也是一个抽象的产品 |
| 何时  使用 | 明确地计划不同条件下创建不同实例时 | 关键  代码 | 创建过程在其子类执行 |
| 优点 | 1一个调用者想创建一个对象，只要知道其名称就可以了2扩展性高，如果想增加一个产品，只要扩展一个工厂类就可以3屏蔽产品的具体实现，调用者只关心产品的接口 | | |
| 缺点 | 每次增加一个产品时，都需要增加一个具体类和对象实现工厂，使得系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。这并不是什么好事。 | | |
| 使用  场景 | 1日志记录器：记录可能记录到本地硬盘、系统事件、远程服务器等，用户可以选择记录日志到什么地方  2数据库访问，当用户不知道最后系统采用哪一类数据库，以及数据库可能有变化时  3设计一个连接服务器的框架，需要三个协议，"POP3"、"IMAP"、"HTTP"，可以把这三个作为产品类，共同实现一个接口 | | |
| 注意  事项 | 作为一种创建类模式，在任何需要生成复杂对象的地方，都可以使用工厂方法模式。有一点需要注意的地方就是复杂对象适合使用工厂模式，而简单对象，特别是只需要通过 new 就可以完成创建的对象，无需使用工厂模式。如果使用工厂模式，就需要引入一个工厂类，会增加系统的复杂度。 | | |

2. Abstract Factory（抽象工厂）



|  |  |  |  |
| --- | --- | --- | --- |
| 意图 | 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类 | | |
| 适用性 | 一个系统要独立于它的产品的创建、组合和表示时。一个系统要由多个产品系列中的一个来配置时。当你要强调一系列相关的产品对象的设计以便进行联合使用时。当你提供一个产品类库，而只想显示它们的接口而不是实现时。 | | |
| 主要  解决 | 接口选择的问题 | 如何  解决 | 在一个产品族里面，定义多个产品 |
| 何时  使用 | 系统的产品有多于一个的产品族，而系统只消费其中某一族的产品 | 关键  代码 | 在一个工厂里聚合多个同类产品 |
| 优点 | 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象 | | |
| 缺点 | 产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码 | | |
| 使用  场景 | 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。 | | |
| 注意  事项 | 产品族难扩展，产品等级易扩展 | | |

3. Builder（建造者）



|  |  |  |  |
| --- | --- | --- | --- |
| 意图 | 将一个复杂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 | | |
| 适用性 | 当创建复杂对象的算法应该独立于该对象的组成部分以及它们的装配方式时。当构造过程必须允许被构造的对象有不同的表示时。 | | |
| 主要  解决 | 接口选择的问题 | 如何  解决 | 在一个产品族里面，定义多个产品 |
| 何时  使用 | 系统的产品有多于一个的产品族，而系统只消费其中某一族的产品 | 关键  代码 | 在一个工厂里聚合多个同类产品 |
| 优点 | 当一个产品族中的多个对象被设计成一起工作时，它能保证客户端始终只使用同一个产品族中的对象 | | |
| 缺点 | 产品族扩展非常困难，要增加一个系列的某一产品，既要在抽象的 Creator 里加代码，又要在具体的里面加代码 | | |
| 使用  场景 | 1、QQ 换皮肤，一整套一起换。 2、生成不同操作系统的程序。 | | |
| 注意  事项 | 产品族难扩展，产品等级易扩展 | | |

意图：

。

适用性：

主要解决：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。

何时使用：一些基本部件不会变，而其组合经常变化的时候。

如何解决：将变与不变分离开。

关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。

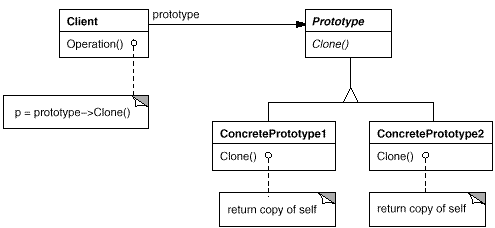
优点： 1、建造者独立，易扩展。 2、便于控制细节风险。

缺点： 1、产品必须有共同点，范围有限制。 2、如内部变化复杂，会有很多的建造类。

使用场景： 1、需要生成的对象具有复杂的内部结构。 2、需要生成的对象内部属性本身相互依赖。

注意事项：与工厂模式的区别是：建造者模式更加关注与零件装配的顺序。

4. Prototype（原型）



意图：

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。

适用性：

当要实例化的类是在运行时刻指定时，例如，通过动态装载；或者

为了避免创建一个与产品类层次平行的工厂类层次时；或者

当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

主要解决：在运行期建立和删除原型。

何时使用： 1、当一个系统应该独立于它的产品创建，构成和表示时。 2、当要实例化的类是在运行时刻指定时，例如，通过动态装载。 3、为了避免创建一个与产品类层次平行的工厂类层次时。 4、当一个类的实例只能有几个不同状态组合中的一种时。建立相应数目的原型并克隆它们可能比每次用合适的状态手工实例化该类更方便一些。

如何解决：利用已有的一个原型对象，快速地生成和原型对象一样的实例。

关键代码： 1、实现克隆操作，在 JAVA 继承 Cloneable，重写 clone()，在 .NET 中可以使用 Object 类的 MemberwiseClone() 方法来实现对象的浅拷贝或通过序列化的方式来实现深拷贝。 2、原型模式同样用于隔离类对象的使用者和具体类型（易变类）之间的耦合关系，它同样要求这些"易变类"拥有稳定的接口。

优点： 1、性能提高。 2、逃避构造函数的约束。

缺点： 1、配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。 2、必须实现 Cloneable 接口。 3、逃避构造函数的约束。

使用场景： 1、资源优化场景。 2、类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等。 3、性能和安全要求的场景。 4、通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。 5、一个对象多个修改者的场景。 6、一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。 7、在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者。原型模式已经与 Java 融为浑然一体，大家可以随手拿来使用。

注意事项：与通过对一个类进行实例化来构造新对象不同的是，原型模式是通过拷贝一个现有对象生成新对象的。浅拷贝实现 Cloneable，重写，深拷贝是通过实现 Serializable 读取二进制流。

深复制与浅复制

浅复制与深复制概念

⑴浅复制（浅克隆）

被复制对象的所有变量都含有与原来的对象相同的值，而所有的对其他对象的引用仍然指向原来的对象。换言之，浅复制仅仅复制所考虑的对象，而不复制它所引用的对象。

⑵深复制（深克隆）

被复制对象的所有变量都含有与原来的对象相同的值，除去那些引用其他对象的变量。那些引用其他对象的变量将指向被复制过的新对象，而不再是原有的那些被引用的对象。换言之，深复制把要复制的对象所引用的对象都复制了一遍。

Java的clone()方法

⑴clone方法将对象复制了一份并返回给调用者。一般而言，clone（）方法满足：

①对任何的对象x，都有x.clone() !=x//克隆对象与原对象不是同一个对象

②对任何的对象x，都有x.clone().getClass()= =x.getClass()//克隆对象与原对象的类型一样

③如果对象x的equals()方法定义恰当，那么x.clone().equals(x)应该成立。

⑵Java中对象的克隆

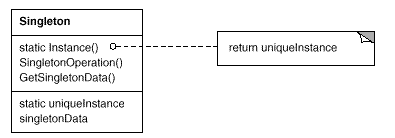
①为了获取对象的一份拷贝，我们可以利用Object类的clone()方法。

②在派生类中覆盖基类的clone()方法，并声明为public。

③在派生类的clone()方法中，调用super.clone()。

④在派生类中实现Cloneable接口。

5. Singleton（单例）



意图：

保证一个类仅有一个实例，并提供一个访问它的全局访问点。

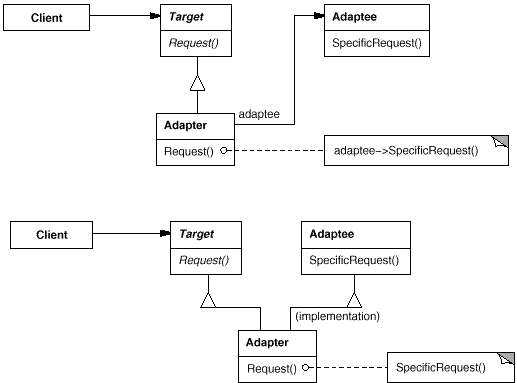
适用性：

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它时。

当这个唯一实例应该是通过子类化可扩展的，并且客户应该无需更改代码就能使用一个扩展的实例时。

**结构型**

6. Adapter Class/Object（适配器）

  
意图：

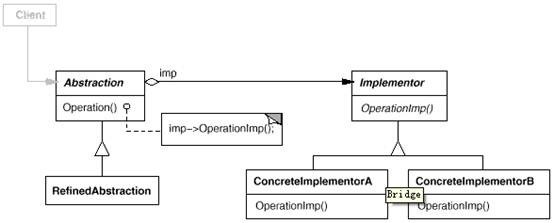
将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。   
适用性：

你想使用一个已经存在的类，而它的接口不符合你的需求。

你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。

（仅适用于对象Adapter ）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

7. Bridge（桥接）



意图：

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。

适用性：

你不希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。

类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时Bridge 模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。

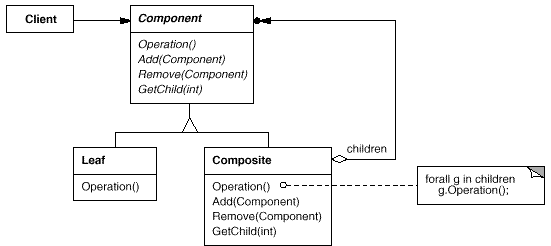
对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。

（C++）你想对客户完全隐藏抽象的实现部分。在C++中，类的表示在类接口中是可见的。

有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。Rumbaugh 称这种类层次结构为“嵌套的普化”（nested generalizations ）。

你想在多个对象间共享实现（可能使用引用计数），但同时要求客户并不知道这一点。一个简单的例子便是Coplien 的String 类[ Cop92 ]，在这个类中多个对象可以共享同一个字符串表示（StringRep ）。

8. Composite（组合）

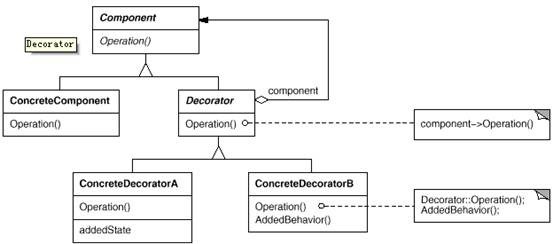
  
意图：

将对象组合成树形结构以表示“部分-整体”的层次结构。C o m p o s i t e 使得用户对单个对象和组合对象的使用具有一致性。   
适用性：

你想表示对象的部分-整体层次结构。

你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

9. Decorator（装饰）

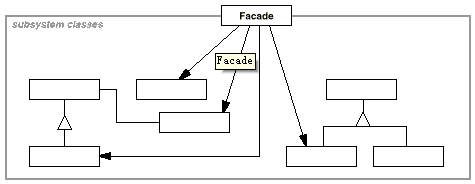
  
意图：   
动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。   
适用性：

在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。

处理那些可以撤消的职责。

当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

10. Facade（外观）



意图：

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

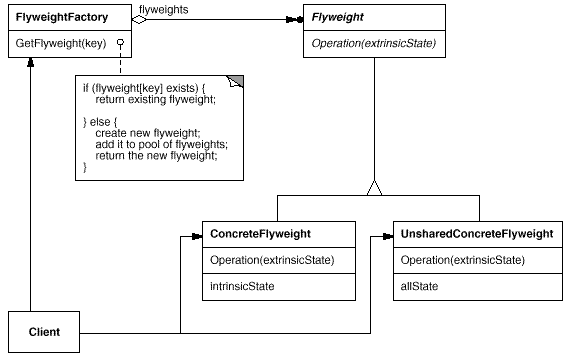
适用性：

当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统进行定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade 可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过facade层。

客户程序与抽象类的实现部分之间存在着很大的依赖性。引入facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。

当你需要构建一个层次结构的子系统时，使用facade模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过facade进行通讯，从而简化了它们之间的依赖关系。

11. Flyweight（享元）



意图：

运用共享技术有效地支持大量细粒度的对象。

适用性：

一个应用程序使用了大量的对象。

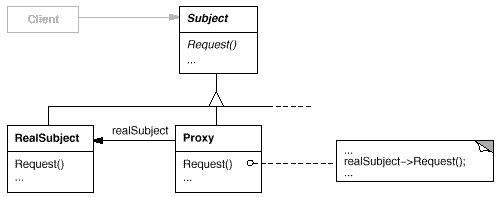
完全由于使用大量的对象，造成很大的存储开销。

对象的大多数状态都可变为外部状态。

如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。

应用程序不依赖于对象标识。由于Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

12. Proxy（代理）



意图：

为其他对象提供一种代理以控制对这个对象的访问。

适用性：

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用Proxy模式。下面是一 些可以使用Proxy 模式常见情况：   
1) 远程代理（Remote Proxy ）为一个对象在不同的地址空间提供局部代表。 NEXTSTEP[Add94] 使用NXProxy 类实现了这一目的。Coplien[Cop92] 称这种代理为“大使” （Ambassador ）。   
2 )虚代理（Virtual Proxy ）根据需要创建开销很大的对象。在动机一节描述的ImageProxy 就是这样一种代理的例子。   
3) 保护代理（Protection Proxy ）控制对原始对象的访问。保护代理用于对象应该有不同 的访问权限的时候。例如，在Choices 操作系统[ CIRM93]中KemelProxies为操作系统对象提供 了访问保护。   
4 )智能指引（Smart Reference ）取代了简单的指针，它在访问对象时执行一些附加操作。 它的典型用途包括：

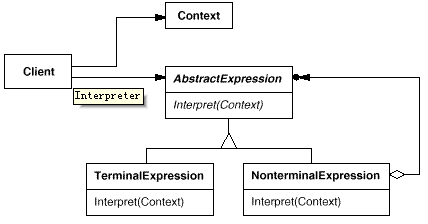
对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它(也称为SmartPointers[Ede92 ] )。

当第一次引用一个持久对象时，将它装入内存。

在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

**行为型**

13. Interpreter（解释器）



意图：

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

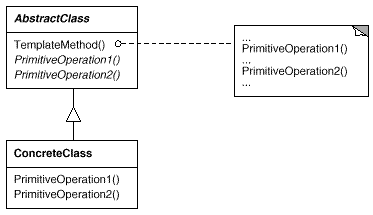
适用性：

当有一个语言需要解释执行, 并且你可将该语言中的句子表示为一个抽象语法树时，可使用解释器模式。而当存在以下情况时该模式效果最好：

该文法简单对于复杂的文法, 文法的类层次变得庞大而无法管理。此时语法分析程序生成器这样的工具是更好的选择。它们无需构建抽象语法树即可解释表达式, 这样可以节省空间而且还可能节省时间。

效率不是一个关键问题最高效的解释器通常不是通过直接解释语法分析树实现的, 而是首先将它们转换成另一种形式。例如，正则表达式通常被转换成状态机。但即使在这种情况下, 转换器仍可用解释器模式实现, 该模式仍是有用的。

14. Template Method（模板方法）



意图：

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

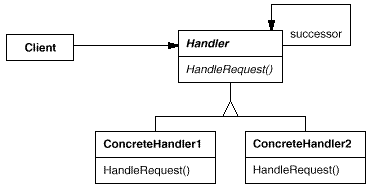
适用性：

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke 和Johnson 所描述过的“重分解以一般化”的一个很好的例子[ OJ93 ]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook ”操作（参见效果一节），这样就只允许在这些点进行扩展。

15. Chain of Responsibility（责任链）



意图：

使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

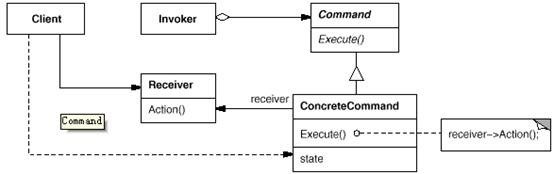
适用性：

有多个的对象可以处理一个请求，哪个对象处理该请求运行时刻自动确定。

你想在不明确指定接收者的情况下，向多个对象中的一个提交一个请求。

可处理一个请求的对象集合应被动态指定。

16. Command（命令）



意图：

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；对请求排队或记录请求日志，以及支持可撤消的操作。

适用性：

抽象出待执行的动作以参数化某对象，你可用过程语言中的回调（call back）函数表达这种参数化机制。所谓回调函数是指函数先在某处注册，而它将在稍后某个需要的时候被调用。Command 模式是回调机制的一个面向对象的替代品。

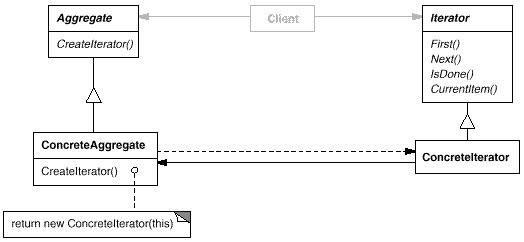
在不同的时刻指定、排列和执行请求。一个Command对象可以有一个与初始请求无关的生存期。如果一个请求的接收者可用一种与地址空间无关的方式表达，那么就可将负责该请求的命令对象传送给另一个不同的进程并在那儿实现该请求。

支持取消操作。Command的Excute 操作可在实施操作前将状态存储起来，在取消操作时这个状态用来消除该操作的影响。Command 接口必须添加一个Unexecute操作，该操作取消上一次Execute调用的效果。执行的命令被存储在一个历史列表中。可通过向后和向前遍历这一列表并分别调用Unexecute和Execute来实现重数不限的“取消”和“重做”。

支持修改日志，这样当系统崩溃时，这些修改可以被重做一遍。在Command接口中添加装载操作和存储操作，可以用来保持变动的一个一致的修改日志。从崩溃中恢复的过程包括从磁盘中重新读入记录下来的命令并用Execute操作重新执行它们。

用构建在原语操作上的高层操作构造一个系统。这样一种结构在支持事务( transaction)的信息系统中很常见。一个事务封装了对数据的一组变动。Command模式提供了对事务进行建模的方法。Command有一个公共的接口，使得你可以用同一种方式调用所有的事务。同时使用该模式也易于添加新事务以扩展系统。

17. Iterator（迭代器）



意图：

提供一种方法顺序访问一个聚合对象中各个元素, 而又不需暴露该对象的内部表示。

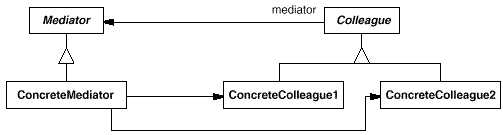
适用性：

访问一个聚合对象的内容而无需暴露它的内部表示。

支持对聚合对象的多种遍历。

为遍历不同的聚合结构提供一个统一的接口(即, 支持多态迭代)。

18. Mediator（中介者）



意图：

用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

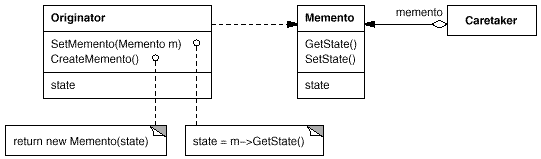
适用性：

一组对象以定义良好但是复杂的方式进行通信。产生的相互依赖关系结构混乱且难以理解。

一个对象引用其他很多对象并且直接与这些对象通信,导致难以复用该对象。

想定制一个分布在多个类中的行为，而又不想生成太多的子类。

19. Memento（备忘录）



意图：

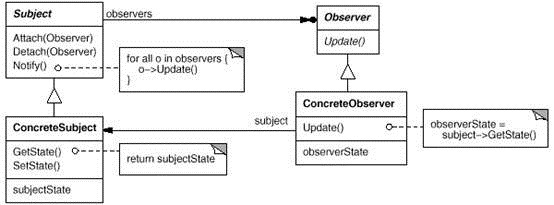
在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

适用性：

必须保存一个对象在某一个时刻的(部分)状态, 这样以后需要时它才能恢复到先前的状态。

如果一个用接口来让其它对象直接得到这些状态，将会暴露对象的实现细节并破坏对象的封装性。

20. Observer（观察者）



意图：

定义对象间的一种一对多的依赖关系,当一个对象的状态发生改变时, 所有依赖于它的对象都得到通知并被自动更新。

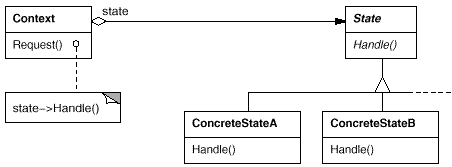
适用性：

当一个抽象模型有两个方面, 其中一个方面依赖于另一方面。将这二者封装在独立的对象中以使它们可以各自独立地改变和复用。

当对一个对象的改变需要同时改变其它对象, 而不知道具体有多少对象有待改变。

当一个对象必须通知其它对象，而它又不能假定其它对象是谁。换言之, 你不希望这些对象是紧密耦合的。

21. State（状态）



意图：

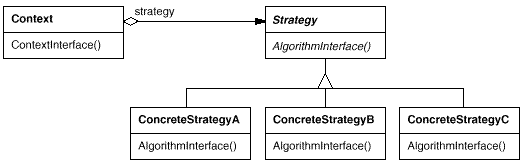
允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

适用性：

一个对象的行为取决于它的状态, 并且它必须在运行时刻根据状态改变它的行为。

一个操作中含有庞大的多分支的条件语句，且这些分支依赖于该对象的状态。这个状态通常用一个或多个枚举常量表示。通常, 有多个操作包含这一相同的条件结构。State模式将每一个条件分支放入一个独立的类中。这使得你可以根据对象自身的情况将对象的状态作为一个对象，这一对象可以不依赖于其他对象而独立变化。

22. Strategy（策略）



意图：

定义一系列的算法,把它们一个个封装起来, 并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

适用性：

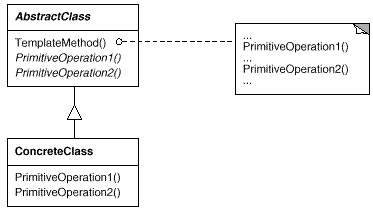
许多相关的类仅仅是行为有异。“策略”提供了一种用多个行为中的一个行为来配置一个类的方法。

需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时[H087] ,可以使用策略模式。

算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。

一个类定义了多种行为, 并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

23. Visitor（访问者）



意图：

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。TemplateMethod 使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

适用性：

一次性实现一个算法的不变的部分，并将可变的行为留给子类来实现。

各子类中公共的行为应被提取出来并集中到一个公共父类中以避免代码重复。这是Opdyke和Johnson所描述过的“重分解以一般化”的一个很好的例子[OJ93]。首先识别现有代码中的不同之处，并且将不同之处分离为新的操作。最后，用一个调用这些新的操作的模板方法来替换这些不同的代码。

控制子类扩展。模板方法只在特定点调用“hook ”操作（参见效果一节），这样就只允许在这些点进行扩展。