#### 一、基础

##### (一)软件设计固有的复杂性

建筑商从来不会去想给一栋已经建好100层高的楼房地下再修一个小地下室。

根本原因:变化

客户需求的变化

技术平台的变化

开发团队的变化

市场环境的变化...

##### **(二)解决复杂性的方法**

1. 分解
2. 抽象

##### (三)软件设计的目标

复用性

##### (四)面向对象设计原则

1. 依赖倒置原则(DIP)

高层模块不应该依赖底层模块，都应该依赖于抽象。

抽象不应该依赖于细节实现。

1. 开闭原则(OCP)

对扩展开放，对修改关闭

1. 单一职责原则(SRP)

一个类仅有一个引起它变化的原因。变化的方向隐含着类的责任。

1. 里氏替换原则(LSP)

父类出现的地方都能使用子类进行替换。

1. 接口隔离原则(ISP)

接口应该小而完备

1. 优先使用对象组合，而不是类继承
2. 封装变化点

将易变化性和非易变化性进行区分归界

1. 针对接口编程，而不是针对实现

##### 重构技法

|  |  |
| --- | --- |
| **重构前** | **重构后** |
| 静态 | 动态 |
| 代码绑定 | 编译绑定、运行时绑定 |
| 继承 | 组合 |
| 编译时依赖 | 运行时依赖 |
| 紧耦合 | 松耦合 |

##### 设计模式注意点

什么时候不用模式?

* 代码可读性很差时
* 需求理解很差时
* 变化没有显现时
* 不是系统的关键依赖点
* 项目没有复用价值时
* 项目将要发布时

使用设计模式时:

* 不要为模式而模式
* 关注抽象类和接口
* 理清变化点和稳定点
* 审视依赖关系
* 要有Framework和Application的隔离四维
* 良好的设计是演化的结果

设计模式成长之路:

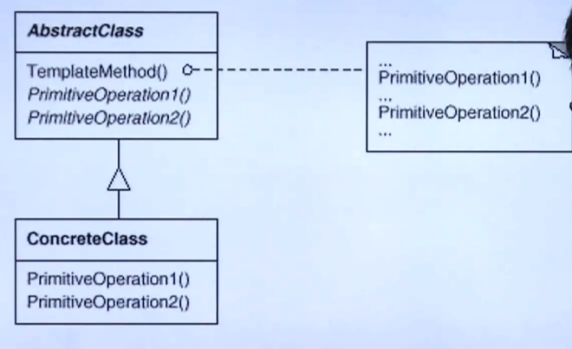
* “手中无剑，心中无剑”：见模式而不知
* “手中有剑，心中无剑”: 可以识别模式，作为应用开发人员使用模式
* “手中有剑，心中有剑”: 作为框架开发人员为应用设计模式
* “手中无剑，心中有剑”: 忘掉模式，只有原则

#### **二、模式组**

##### (一)组件协作模式组

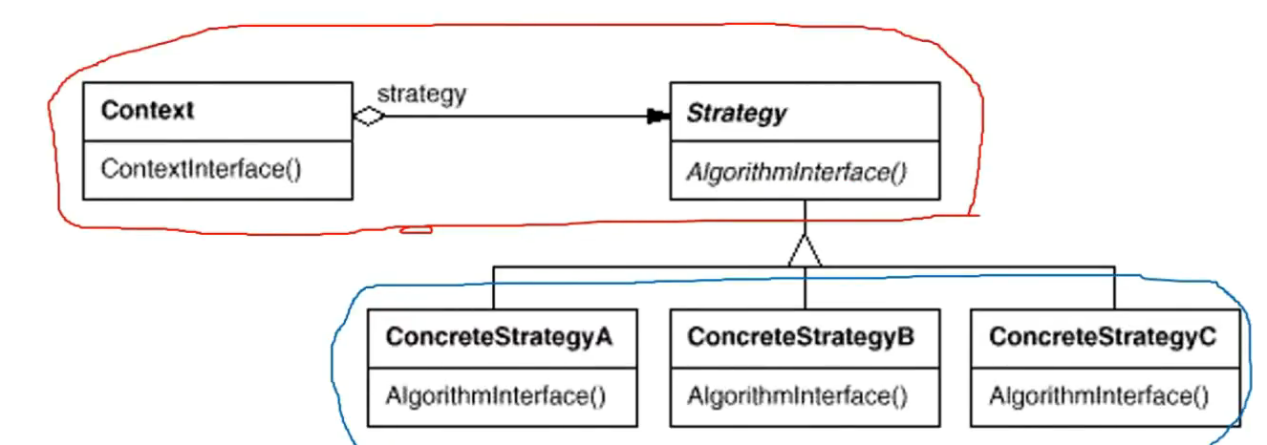
组件协作模式通过晚期绑定，来实现框架与应用之间的松耦合，是二者之间协作常用的模式

1. Template Method

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

1. Strategy

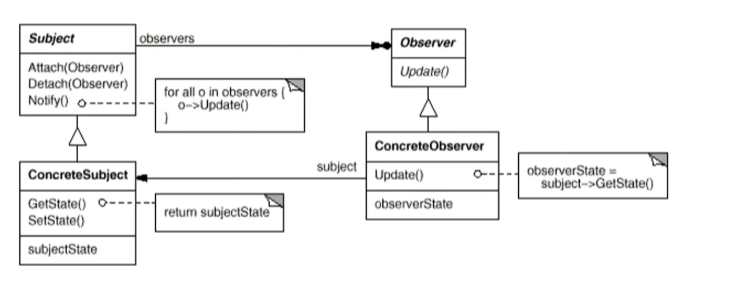
定义一系列算法，把它们一个个封装起来，并使它们可相互替换。使得算法可独立于使用它的客户程序而变化。



* Strategy及其子类为组件提供了一系列可重用的算法，使得类型在运行时方便地根据需要在各个算法之间进行切换。
* Strategy提供了用条件判断语句的另外一种选择，消除条件判断语句，就是在解耦合。含有速度哦条件判断语句的代码通常都需要使用Strategy模式。
* 如果Strategy对象不需要多个实例对象，那么各个上下文可以共享同一个Strategy对象，从而节省对象开销。

1. Observer/Event

定义对象间的一种一对多(变化)的依赖关系，当以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自动更新。

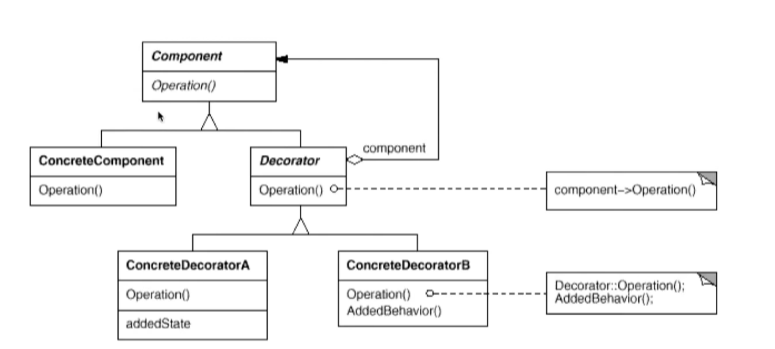


* 使用面向对象的抽象，Observer模式可以独立地改变目标与观察者，从而使二者之间的依赖关系达到松耦合。
* 目标发送通知时，无需指定观察者，通知(可以携带通知信息作为参数)会自动传播。
* 观察者自己决定是否需要订阅通知，目标对象对此一无所知。
* Observer模式是基于事件的UI框架中非常常用的设计模式，也是MVC模式的一个重要组成部分。

##### (二)单一职责模式组

1. Decorator

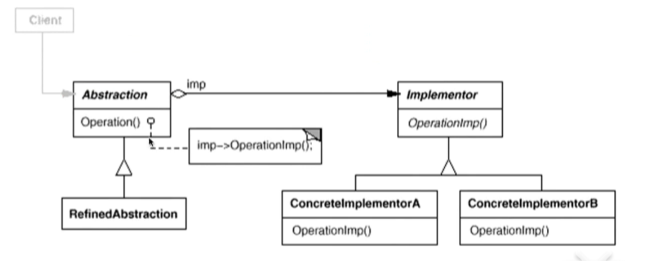
动态(组合)地给一个对象增加一些额外的职责。就增加功能而言，Decorator模式比生成子类(继承)更为灵活(消除重复代码且减少子类个数)。



* 通过采用组合而非继承的手法，实现了在运行时动态扩展对象能力的功能，而且可以根据需要扩展多个功能。避免了使用继承带来的灵活性差和多子类衍生问题。
* 在接口上表现为is-a的继承关系，继承是为了使装饰类具有主体类的接口规范，但在实现上变现为has-a的组合关系，组合是因为只需要依赖于主体类对外的服务进行功能的扩展。
* 要点在于解决主体类在多个方向上的扩展功能，是为“装饰”的含义。

1. Bridge

将抽象部分与实现部分分离，使得可以各自独立地变化。



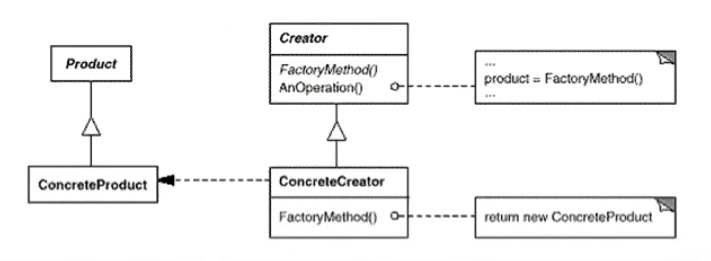
* 使用对象间的组合关系解耦了抽象和实现之间固有的绑定关系，使得抽象和实现可以沿着各自的维度进行变化。
* 有时类似于多继承方案，但是多继承方案往往违背单一职责原则，复用性比较差。
* 适用于多个变化维度。

##### (三)对象创建模式组

通过对象创建模式绕开new,来避免对象创建(new)过程中导致的紧耦合，从而支持对象创建的稳定。是接口抽象之后的第一步工作。

1. Factory

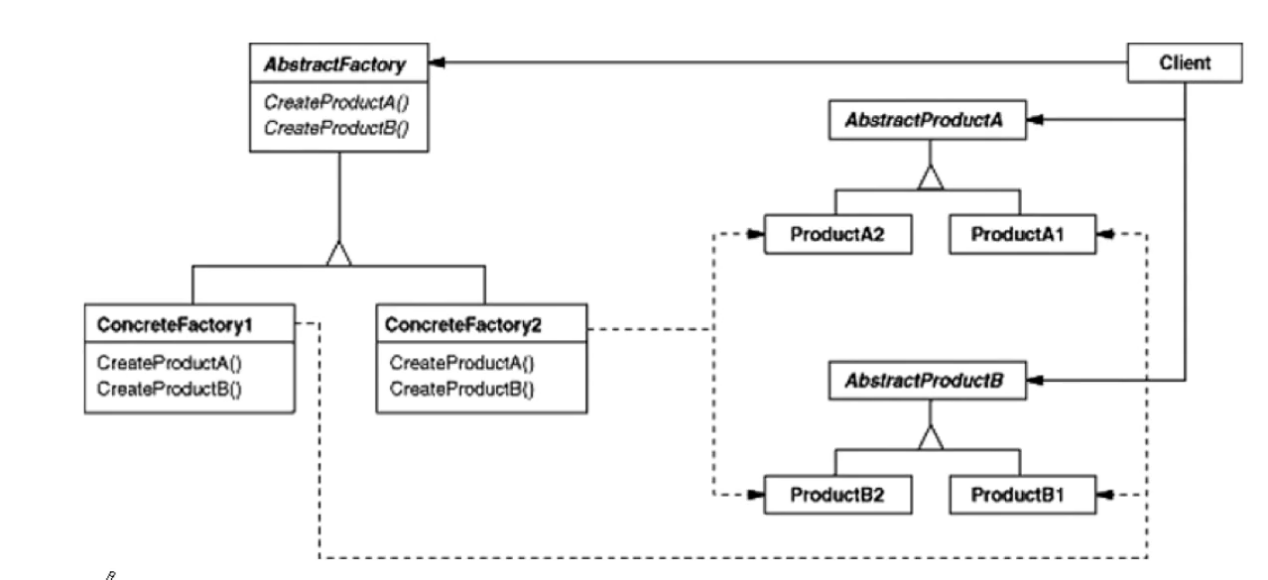
定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory使得一个类的实例化延迟(目的:解耦 手段:虚函数)到子类。



* Factory用于隔离对象的使用者和具体类型之间的耦合关系。面对一个经常变化的具体类型，紧耦合关系(new)会导致软件的脆弱。
* Factory同过面向对象的手法(多态)，将所要创建的具体对象工作延迟到子类，从而实现一种扩展(而非修改)的策略，较好地解决这种紧耦合关系。
* Factory解决单个对象的创建需求变化。缺点在于要求创建方法的参数要一致。

1. Abstract Factory

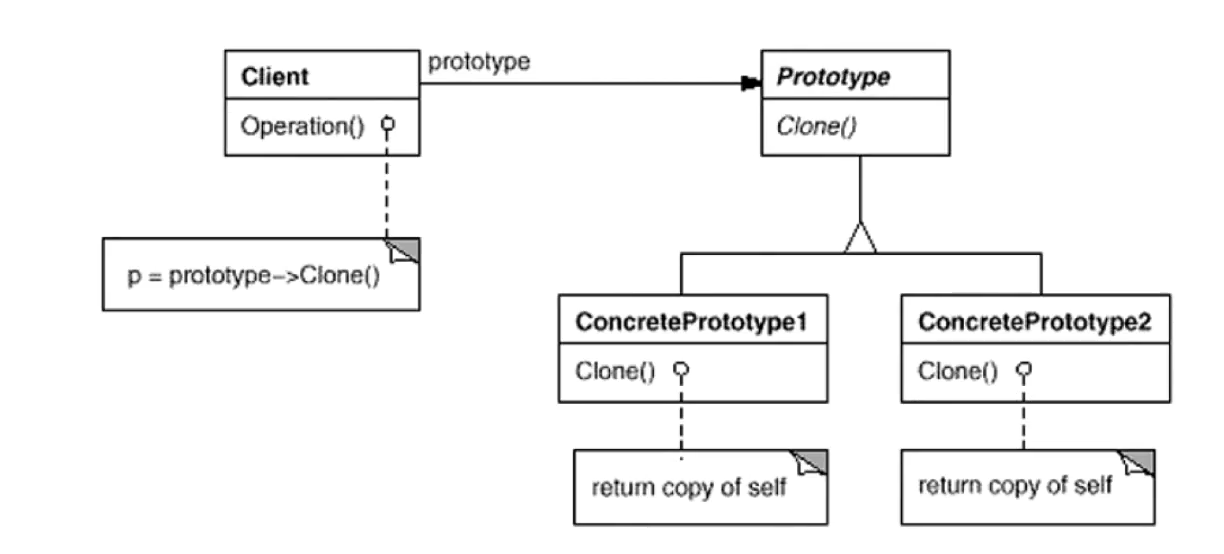
提供一个接口，让该接口负责创建一系列相关或相互依赖的对象，无需指定它们具体的类。



* 如果没有应对多系列对象构建的需求变化，则没有必要使用Abstract Factory。
* “系列对象”指的是在某一特定系列下的对象之间有相互依赖、或作用的关系，不同系列的对象之间不能相互依赖。
* Abstract Factory主要应对于“系列对象”的需求变动，其缺点在于难以应对新系列对象的需求变动。（改变AbstractFactory中的的创建接口）

1. Prototype

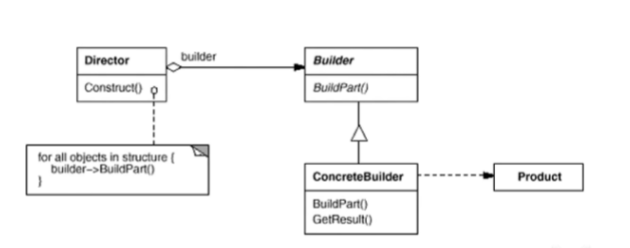
使用原型实例指定创建对象的种类，然后通过深拷贝这些原型来创建新的对象。



* Prototype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，要求易变类拥有稳定的接口。
* Prototype模式对于如何创建易变类的尸体对象采用原型克隆的方法来做，使得可以非常灵活地动态创建拥有某些稳定接口的新对象。
* Prototype模式中的Clone方法可以利用某些框架中的序列化来进行深拷贝。

1. Builder

将一个复杂对象的构建与其表示相分离，使得同样的构建过程(稳定)可以创建不同的表示(变化)。



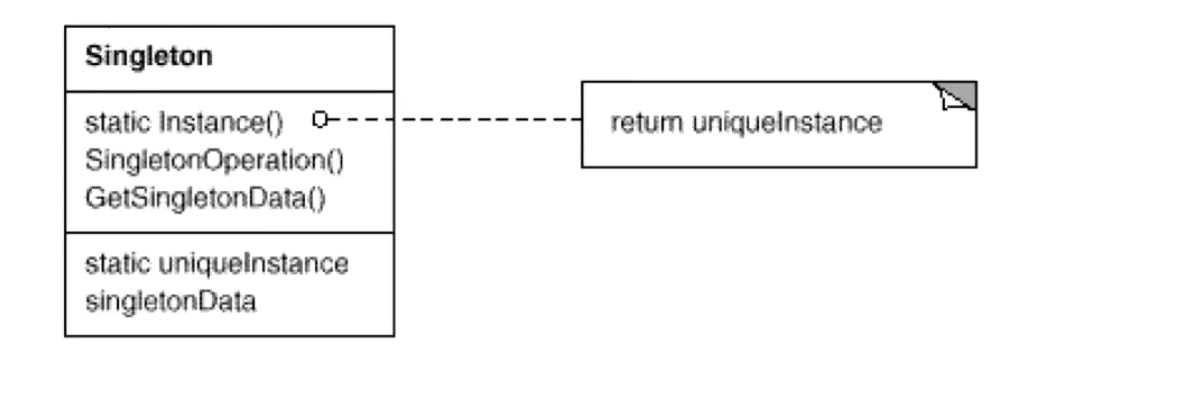
* Builder模式主要用于分步骤构建一个复杂的对象，在这其中分步骤是一个稳定的算法，而复杂对象的各个部分则经常变化。
* 变化点在哪里，封装在哪里。Builder模式主要在于应对复杂对象各个部分的频繁的需求变动，其缺点在于难以应对分步骤构建算法的需求变动。
* 在Builder模式中，注意不同语言中构造器内调用虚函数的差别。

##### (四)对象性能模式组

面向对象很好地解决了抽象的问题，但是不可避免地要付出一定的代价。对于通常情况来讲，面向对象的成本大都可以忽略不记。但是在某些情况，面向对象带来的成本必须进行慎重考虑。

1. Singleton

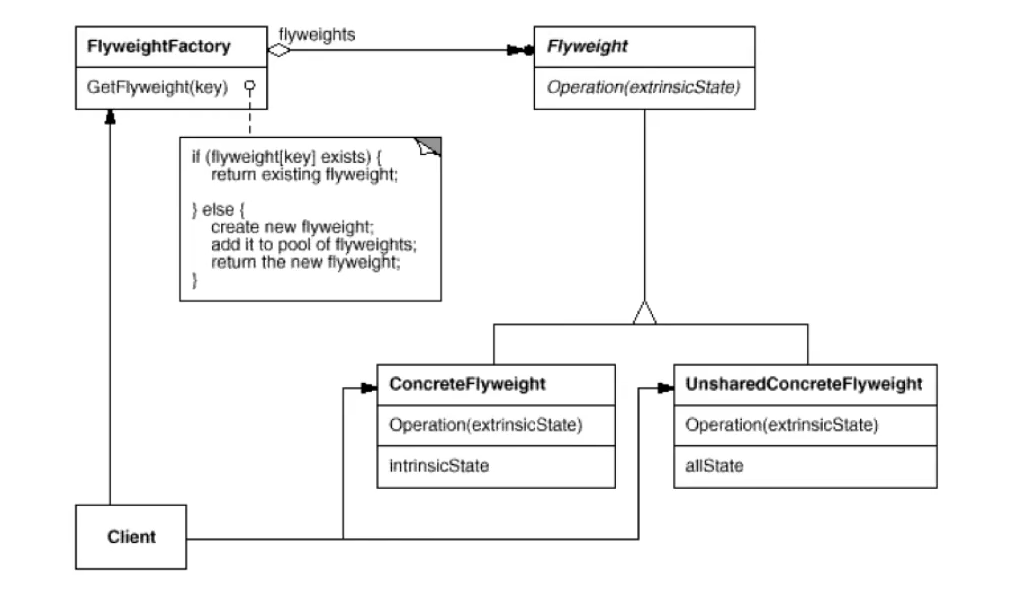
保证一个类仅有一个实例，并提供一个该实例的全局访问点。



* Singleton模式中的实例构造器可以设置为protected以允许子类派生。
* Singleteon模式一般不支持拷贝构造函数和Clone接口，因为这有可能导致多个对象实例。
* 如何实现多线程环境下的安全Singleton?注意对双检查锁的正确实现。

1. Flyweight（享元模式）

运用共享技术有效地支持大量细粒度的对象。



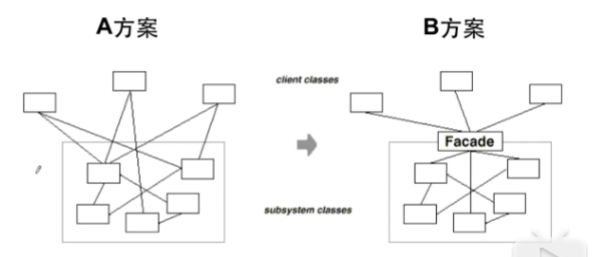
* Flyweight主要解决面向对象的代价问题，一般不触及面向对象的抽象性问题。
* Flyweight采用对象共享的做法来降低系统中对象的个数，从而降低细粒度对象给系统带来的内存压力。在具体实现方面，注意对象状态的处理。
* 对象的数量太大从而导致对象的内存开销加大，数量的级别需要结合具体情况进行甄别。

##### (五)接口隔离模式组

在组件构建过程中，某些接口之间的直接依赖会带来很多问题，甚至根本无法实现。采用添加一层间接(稳定)接口，来隔离本来互相紧密关联的接口。

1. Facade（门面模式）

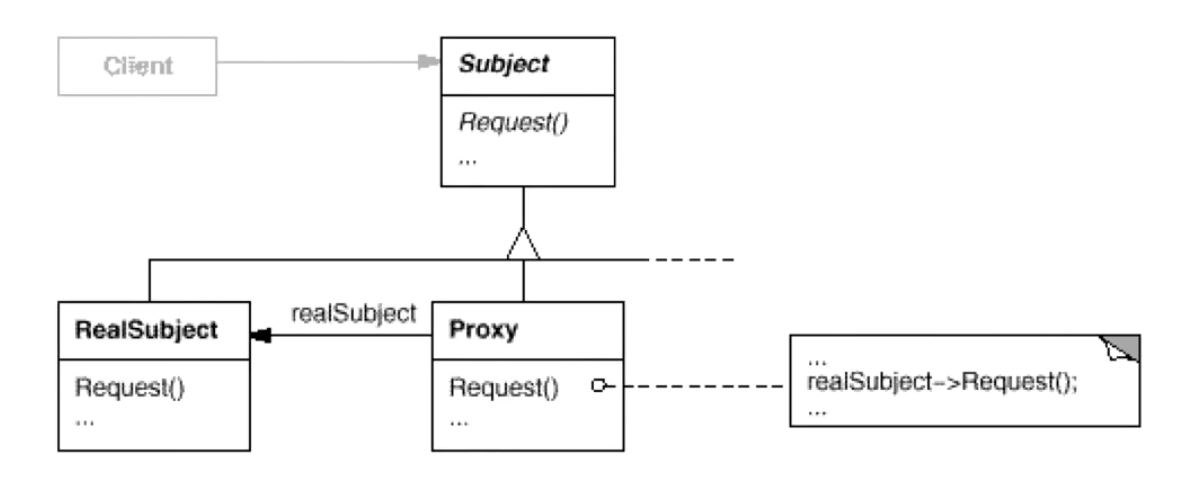
为子系统中的一组接口提供一个一致(稳定)的高层接口，使得这一子系统更加容易进行复用。



* 内部子系统的任何变化不会影响到Facade接口的变化。
* Facade设计模式更注重从架构的层次去看整个系统，而不是单个类的层次。Facade更多时候是一种架构模式。
* Facade并非一个集装箱，可以任意地放进多个对象。子系统内部应该是相互耦合关系比较大的一系列组件，而不是一个简单的功能集合。

1. Proxy

为其他对象提供一种代理以控制(隔离、使用接口)对这个对象的访问。



* 增加一层间接层是软件系统中对许多复杂问题的一种常见解决方法。
* 具体proxy设计模式的实现方法、实现粒度可能相差很大。有对单个对象做细粒度的控制，如copy-on-write技术，有些可能对组件模块提供抽象代理层，在架构层次对对象做proxy.
* proxy并不一定要保持原对象接口完整的一致性，只要能够实现间接控制，有时损及2一些透明性是可以接受的。