Open closed principle:Software entities (class, modules, functions, etc.) should be open for extension, but closed for modification.

一個軟體個體應該要夠開放使得它可以被擴充，但是也要夠封閉以避免不必要的修改。

套用 OCP 的最高境界就是做到：藉由增加新的程式碼來擴充系統的功能，而不是藉由修改原本已經存在的程式碼來擴充系統的功能

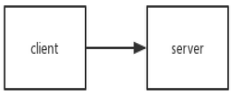
(If the OCP is applied well, then further changes of that kind are achieved by adding new code, not by changing old code that already works.)

英文的例子：http://www.oodesign.com/open-close-principle.html

http://blog.csdn.net/qq_18497495/article/details/52980333

先看一个不遵守OCP原则的例子：

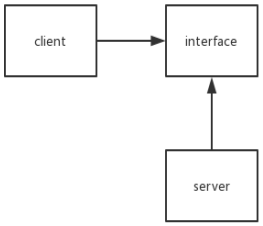
client类调用server类实现某些功能。



如果有新的需求，client需要使用另外一种server，那么就要修改client中的对应部分的代码。

并且与client关联的代码，也需要重新链接、或打包等等。

修改为遵守OCP原则：（strategy 策略模式）



此时，client依赖的是一个接口（也可以是一个抽象类），server只是interface的一个具体实现。

面对同样的需求，只需新增新的server实现，但对client来说，是符合OCP原则的。

OCP 缺點：增加代碼複雜度

OCP 是面向对象的核心所在。遵循这个原则可带来巨大的好处：灵活性、扩展性、重用性、维护性。

但是，在软件的任何地方都遵循 OCP 原则而进行抽象同样不是一个好主意，开发人员应该仅仅对频繁变化的那部分做出抽象。拒绝不成熟的抽象，和抽象本身一样的重要。

OCP

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。

在仔细思考以及仔细阅读很多设计模式的文章后，终于对开闭原则有了一点认识。其实，我们遵循设计模式前面 5 大原则，以及使用 23 种设计模式的目的就是遵循开闭原则。

开闭原则无非就是想表达这样一层意思：用抽象构建框架，用实现扩展细节。因为抽象灵活性好，适应性广，只要抽象的合理，可以基本保持软件**架构**的稳定。而软件中易变的细节，我们用从抽象派生的实现类来进行扩展，当软件需要发生变化时，我们只需要根据需求重新派生一个实现类来扩展就可以了。当然前提是我们的抽象要合理，要对需求的变更有前瞻性和预见性才行。

http://blog.csdn.net/qq_27630169/article/details/52123382

满足开闭原则的模块符合下面两个标准：

对扩展开放 ------- 模块的行为可以被扩展从而满足新的需求。

对修改关闭 ------- 不允许修改模块的源代码。（或者尽量使修改最小化）

怎样实现开闭原则？

抽象

多态

继承

接口

http://blog.163.com/javaee_chen/blog/static/179195077201162775l6279/

design principle

1、Programming to an interface, not an implementation.

這句話表示程式不應該在 compile 的時候就決定好物件的型態，應該在運行時根據子類別回傳的物件再決定型態

2、Favor object composition over class inheritance.（优先使用对象组合,而不是类继承/多用物件複合技術，少用類別繼承）

繼承是 OOP 裡最強烈的耦合（Coupling）關係，

子類別會對父類別瞭若指掌，將會打破封裝，同時也會繼承父類無用的方法

當你動到父類別時，將影響底下所有的子類別。

繼承的關係越長、越複雜時，

所造成的影響就會越大、越難維護。

Composite

簡而言之就是透過 Composite Pattern 讓你可以將個別的 Component 組織成一個樹狀結構. 並且你在處理一個 Component 或者是一個樹狀結構的 Components 並沒有區別. 而所謂的 leaf node 就是沒有 children 的 component.

Component:1、为组合中的对象声明接口；2、在适当情况下实现所有类共有接口的缺省行为；3、声明一个接口用于访问管理 Component 的子组件，在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它

Leaf:1、在组合中表示叶节点对象，叶节点没有子节点，2、在組合中定义其行为

Composite:1、定义有子部件的那些部件的行为 2、存储子部件，实现与子部件有关的操作

Client:通过 Component 接口操作组合件和个别对象。

组合模式的优点

将对象组合成树形结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性。

使用场景

1. 你想表示对象的部分-整体层次结构；
2. 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。

引用大话设计模式的片段：“当发现需求中是体现部分与整体层次结构时，以及你希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑组合模式了。”

总结

通过上面的简单讲解，我们知道了，组合模式意图是通过整体与局部之间的关系，通过树形结构的形式进行组织复杂对象，屏蔽对象内部的细节，对外展现统一的方式来操作对象，是我们处理更复杂对象的一个手段和方式。现在再结合上面的代码，想想文章开头提出的公司OA系统如何进行设计。

在 Component 中声明所有用来管理子对象的方法，其中包括 Add、Remove 等，这样实现 Component 接口的所有子类都具备了 Add 和 Remove。

这样做的好处就是叶节点和枝节点对于外界没有区别，它们具备 完全一致的行为 接口。

但问题也很明显，因为 Leaf 类本身不具备 Add()、Remove()方法的 功能，所以实现它是没有意义的。

何时使用组合模式：

当你发现需求中是体现部分与整体层次的结构时，以及你希望用户可以忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象时，就应该考虑用组合模式了。

```
2 class Component
3 {
4 public:
5     //纯虚函数，只提供接口，没有默认的实现
6     virtual void Operation()=0;
7
8     // 虚函数,提供接口,有默认的实现就是什么都不做
9     virtual void Add(Component*);
0     virtual void Remove(Component*);
1     virtual Component* GetChild(int index);
2     virtual ~Component();
3 protected:
4     Component();
5 };
```

leaf 類要實現的設爲純虛函數，其他設爲虛函數

http://www.cnblogs.com/jiese/p/3168844.html

组合模式允许你将对象组合成树形结构来表现“整体/部分”层次结构。组合能让客户以一致的方式处理个别对象以及对象组合。

在 GOF 的《设计模式:可复用面向对象软件的基础》一书中对组合模式是这样说的：将对象组合成树形结构以表示“部分-整体”的层次结构。组合（Composite）模式使得用户对单个对象和组合对象的使用具有一致性。

简单的理解组合模式，组合模式就是把一些现有的对象或者元素，经过组合后组成新的对象，新的对象提供内部方法，可以让我们很方便的完成这些元素或者内部对象的访问和操作。

https://www.kancloud.cn/digest/walker1/201545　這個裏面的例子，用來管理子類對象的方法，leaf 類別不用的函數 add()等用到了拋例外

组合(Composite)模式的其它翻译名称也很多，比如合成模式、树模式等等。在《设计模式》一书中给出的定义是：将对象以树形结构组织起来，以达成“部分－整体”的层次结构，使得客户端对单个对象和组合对象的使用具有一致性。

从定义中可以得到使用组合模式的环境为：在设计中想表示对象的“部分－整体”层次结构；希望用户忽略组合对象与单个对象的不同，统一地使用组合结构中的所有对象。

看下组合模式的组成。

1)　抽象构件角色 Component：它为组合中的对象声明接口，也可以为共有接口实现缺省行为。

2)　树叶构件角色 Leaf：在组合中表示叶节点对象——没有子节点，实现抽象构件角色声明的接口。

3)　树枝构件角色 Composite：在组合中表示分支节点对象——有子节点，实现抽象构件角色声明的接口；存储子部件。

http://www.tqcto.com/article/framework/67.html (這個有講，管理方法是在 Component 中就声明还是在 Composite 中声明)

一种方式是在 Component 里面声明所有的用来管理子类对象的方法，以达到 Component 接口的最大化（如下图所示）。目的就是为了使客户看来在接口层次上树叶和分支没有区别——透明性。但树叶是不存在子类的，因此 Component 声明的一些方法对于树叶来说是不适用的。这样也就带来了一些安全性问题。

另一种方式就是只在 Composite 里面声明所有的用来管理子类对象的方法（如下图所示）。这样就避免了上一种方式的安全性问题，但是由于叶子和分支有不同的接口，所以又失去了透明性。

《设计模式》一书认为：在这一模式中，相对于安全性，我们比较强调透明性。对于第一种方式中叶子节点内不需要的方法可以使用空处理或者异常报告的方式来解决。

为了防止客户对 Leaf 进行非法的 Add 和 Remove 操作，所以，在实际开发过程中，进行 Add 和 Remove 操作时，需要进行对应的判断，判断当前节点是否为 Composite。

http://blog.csdn.net/lcl_data/article/details/8811101 (有提到實現接口的兩種模式：安全模式與透明模式)

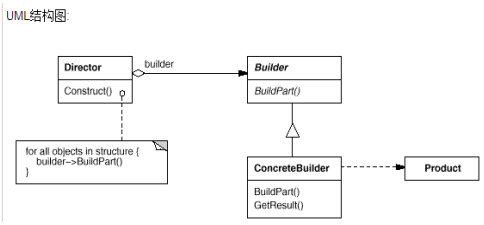
Visitor

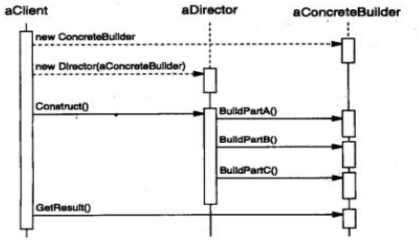
Builder

http://www.cppblog.com/converse/archive/2006/07/21/10305.html （這個不錯）

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。

UML结构图:





解析:

Builder 模式是基于这样的一个情况:一个对象可能有不同的组成部分,这几个部分的不同的创建对象会有不同的表示,但是各个部分之间装配的方式是一致的.比方说一辆单车,都是由车轮车座等等的构成的(一个对象不同的组成部分),不同的品牌生产出来的也不一样(不同的构建方式).虽然不同的品牌构建出来的单车不同,但是构建的过程还是一样的(哦,你见过车轮长在车座上的么?).

也就是说,Director::Construct 函数中固定了各个组成部分的装配方式,而具体是装配怎样的组成部分由 Builder 的派生类实现.

实现:

Builder 模式的实现基于以下几个面向对象的设计原则:1)把变化的部分提取出来形成一个基类和对应的接口函数,在这里不会变化的是都会创建 PartA 和 PartB,变化的则是不同的创建方法,于是就抽取出这里的 Builder 基类和 BuildPartA,BuildPartB 接口函数 2)采用聚合的方式聚合了会发生变化的基类,就是这里 Director 聚合了 Builder 类的指针.

http://blog.csdn.net/i_like_cpp/article/details/8992722

总结一下建造模式：

1、建造者模式的使用使得产品的内部表象可以独立的变化。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、每一个 Builder 都相对独立，而与其它的 Builder 无关。

3、可使对构造过程更加精细控制。

4、将构建代码和实现代码分开。

5、建造者模式的缺点在于难于应付“分步骤构建算法”的需求变动

http://blog.csdn.net/xlf13872135090/article/details/19911589

Template Method