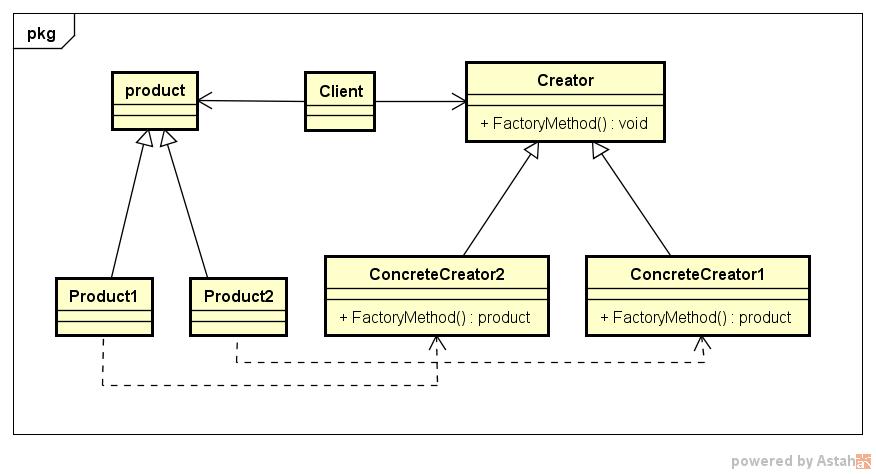
设计模式复习文档

**1,创建型模式**

**1.1工厂方法模式**

**问题**:有这样一种类体系,他的构造方式很有可能会发生变化,所以想把对象创建部分代码和调用点代码分离开,使得构造方式的变化不会影响调用位置.定义一个用于创建对象的接口,让子类决定实例化那一个类.

**方法**:



**特点：**

优点：扩展产品时，完全支持开闭原则；新增子类产品、新增产品树、用同构的产品树替换原有产品树；具有对应的层次结构，Client只通过产品的抽象接口交互，不必明确知道具体的产品

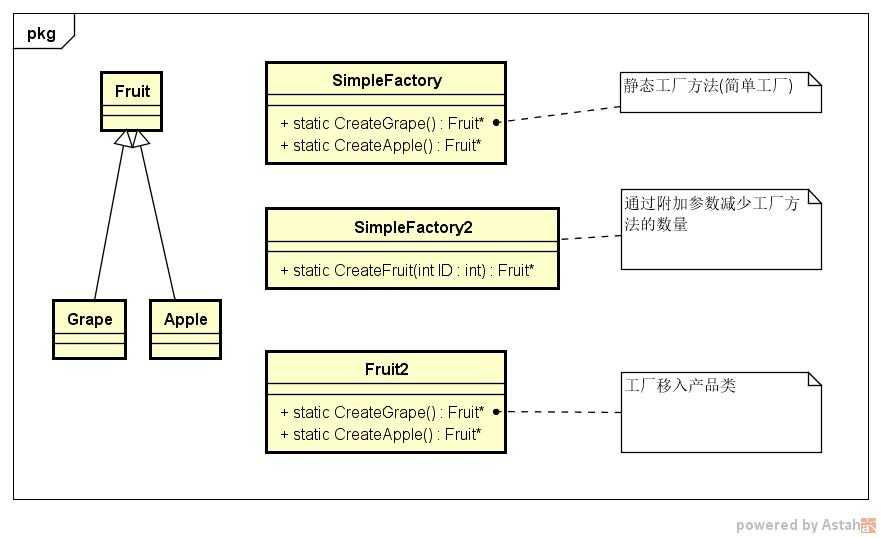
不足：子类数量可能较多（有时可用类模版减少子类数量）

**1.2简单工厂模式**

* **问题**：1. 产品种类稳定,则从Factory类派生子类,失去意义;Factory中的工厂方法,成为类方法更合理.

2. 某个类有很多种构造方法,这些构造方法的参数不同,构造出的具体对象也不同,然而名字是一样的,在使用上,不能通过名字来判断某个构造的具体作用.

* **解决方案**:使用简单工厂方法,也称**静态工厂方法**,能使得构造现场不需要工厂构造器的实例即可构造所需对象,核心是工厂类,工厂类可以有多个工厂方法,也可以附加参数减少工厂方法数量,工厂类可简单移入产品类里,当一个产品有多个重载的构造函数，可应用简单工厂方法在该产品类中对构造方法进行一次封装.
* **实例:**



**1.3单件模式**

* **问题**:确保一个类仅有一个实例,能够从外部访问该实例,使用全局变量有局限性和不足.
* **解决方案**:使用单件模式:

**延迟式:**

Singleton \* Singleton::instance = 0；

Singleton \* Singleton::Instance( ){

if ( instance == 0)

instance = new instance;

else

return instance;

}

**饥饿式:**

public class Singleton {

private static final Singleton instance = new Singleton;

private Singleton() { }

public static Signleton getInstance() {

return instance;

}

}

**注册式:**

public class Singleton {

static private HashMap reg = new HashMap( );

static { reg.put(“MyClassName”,new Singleton); }

protected Singleton() { }

public static Signleton getInstance(String name) {

if (name == null )

name = “DefaultClassName”;

if (reg.get(name)== null)

reg.put(Class.forName(name).newInstance();

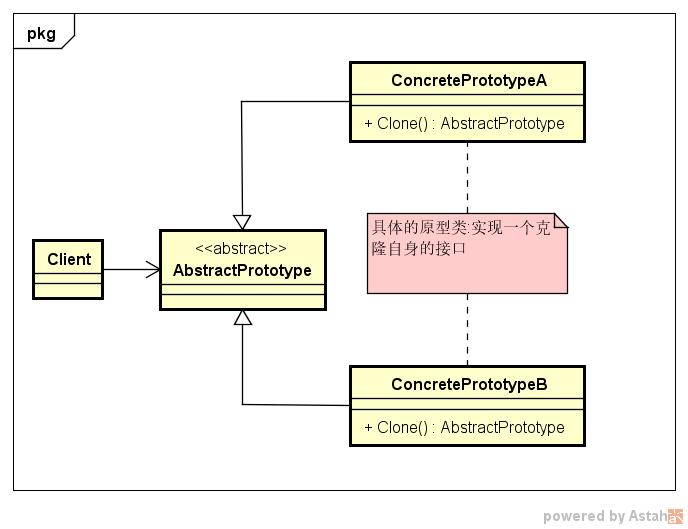
return (Singleton)(reg.get(name));

}

}

**1.4 原型模式**

* **问题:**需要创建多次,多个对象，但都“一个样”或“**差不多**”,例如一个房间的若干片墙壁,虽然都是墙壁,但是其长短,是否有门窗等属性是不同的,不想直接使用构造函数,虽可应用工厂方法,但是增加产品子类会使得工厂的子类过多且不能使用动态改变产品子类类型.
* **解决方案:**使用原型方法,为每个可能用到的产品实例化出一个“样本”,需要产品时,“复制”相应的“样本”,必要的话,再细微“修改”.
* **实现:**



**登记式原型方法的代码实现:**

**class ShapeManager {**

**private:**

**static Rectang \* aRect;**

**static Circle \* aCircle;**

**public:**

**Shape \* Create(int id ) {**

**if (id==1) return aRect->Clone();**

**else if (id==2) return aCricle->Clone();**

**else return NULL;**

**}**

**};**

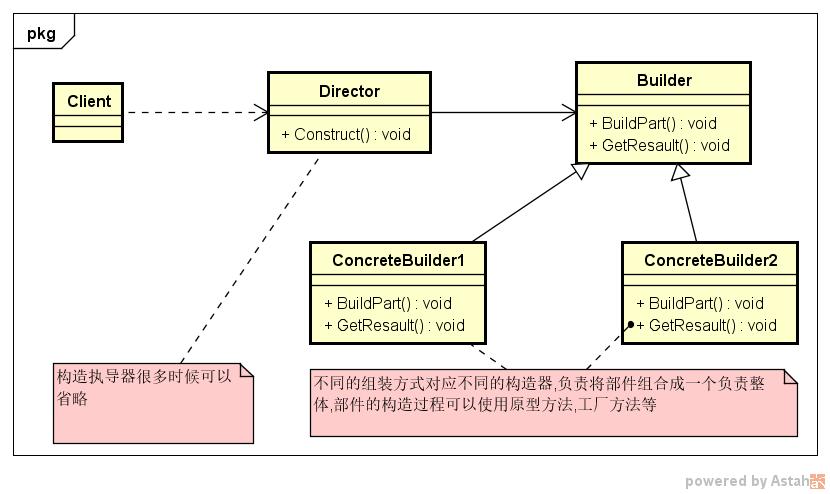
**可以通过一个”注册表”来记录全部实例对象,增删注册表，就可以实现动态修改产品,改变注册对象，可以相当于新定义一个产品类,创建对象时,通过索引,从注册表中查找，再克隆.**

**Shape \* Create(int id ) {**

**return vec->get(id)->Clone();**

**}**

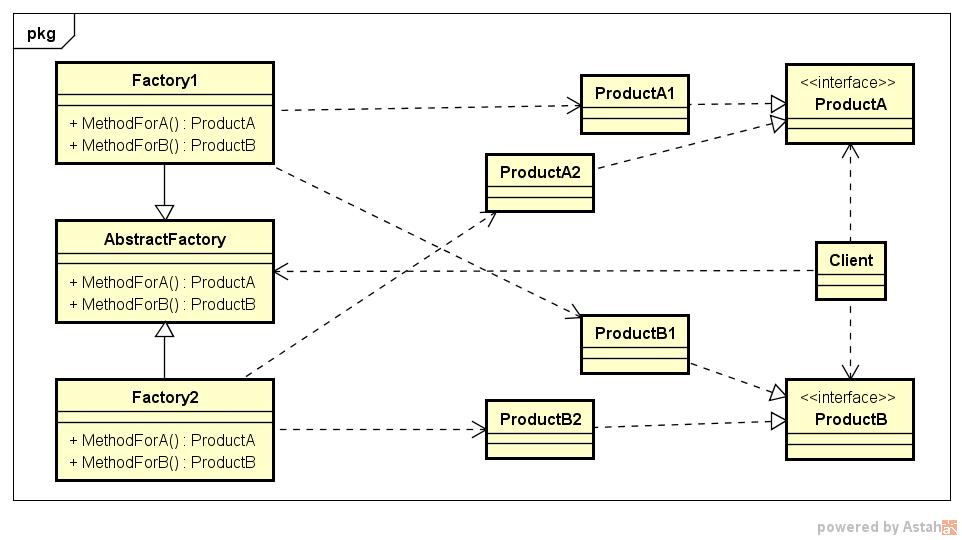
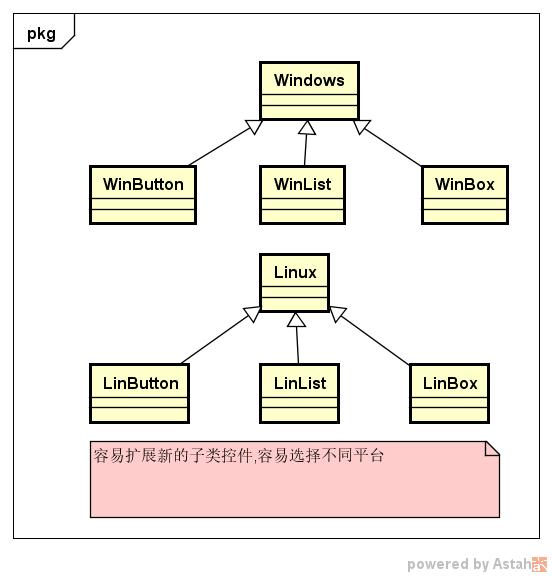
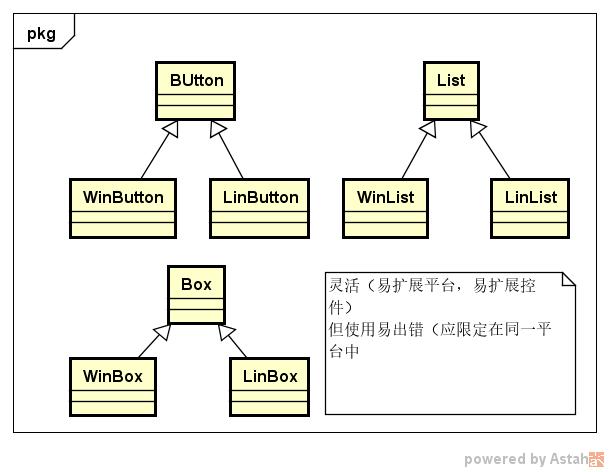
* **对比工厂方法：有效减少子类数量,当有需求要求减少子类的数量的时候,可以使用原型方法,注意应用条件是”差不多”,即:如果某一个类的子对像之间的差别非常大,那么为他们分别建立工厂方法的子类相对而言更整齐一点.**
  1. **构造器(生成器)模式**
* **问题:**创建**复杂产品**或**复合产品**的时候,构成复合产品的子产品(部件)种类,数量可能会变化,**部件装配成整体**的过程也可能变化,如果单纯使用工厂方法,装配过程就会被固定在工厂的调用点.
* **解决方案:在”部件工厂”和使用最终产品的地点之间,增加一层Builder,组装产品的细节封装在Builder中.**
* **实现:**



* 构造器模式隐藏了构建的细节和装配过程,分离了构造代码和产品表示,使用相同的部件，但使用不同的构造器，可以构造不同的复杂对象,在使用中,可以根据情况省略抽象Builder,或省略director,或将Builder移动到Product中.

**1.6 抽象工厂模式**

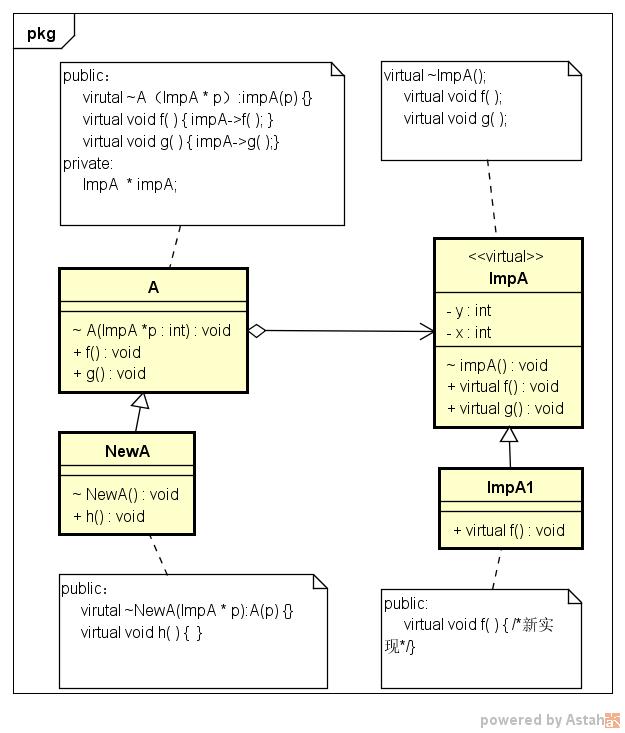
* 问题:**不同的类树,具有相同的层次结构**.比如某跨平台软件既有Win平台版也有Linux平台版,但是这两个平台不可能同时使用.
* 解决方案:



**2.结构型模式**

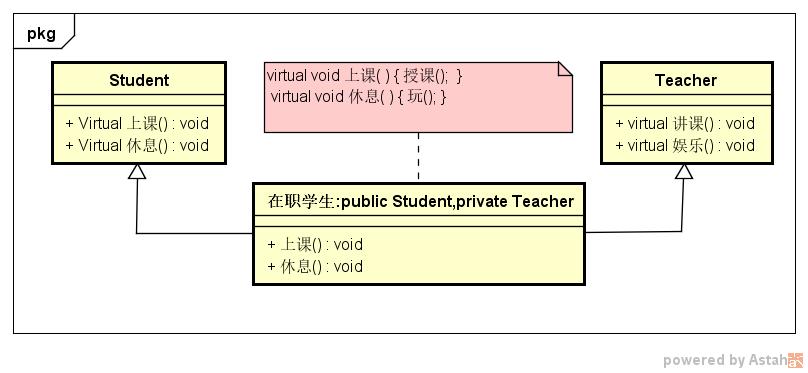
**2.1 桥接模式**

* **问题:**某个类的**接口,实现,属性,都有可能发生变化**,考虑到各种因素,不能单独的使用组合和继承等基本复用方式.
* **解决方案:**1.分离接口和实现,2.用组合连接接口和实现,3.使接口和实现的变化独立.分离接口及其实现,使得各部分可独立扩展,可隐藏实现,也称Handle模式/Handle-Body模式.
* **实例:**



**2.2 适配器模式**

* 问题:两个不同的类A和B,接口不同,但又希望构造一个这样的对象,他**能够使用A的接口来管理,又能使用B类中定义的方法**,同时不更改原接口.例如已有学生类和老师类,希望创建一个实例,用来表示正在进修学习的老师.
* **解决方案:1.类适配器模式;**

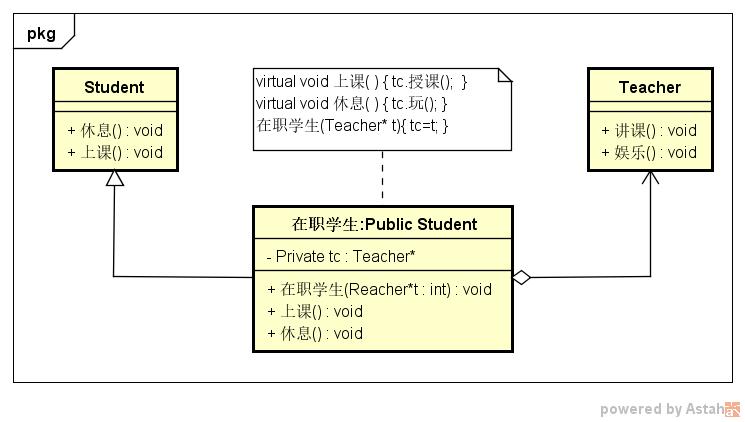


**类适配器:**新建一个类**,public继承A**或者A的上层接口(使用接口) **private继承B(使用方法)**,这样这个新的类就做到了使用A的接口和Ｂ的方法，但是他有局限性：

1.语言必须支持多重继承;

2.不能使用B的子类中定义的方法

**对象适配器模式:**



**对象适配器:**新建一个类,**继承A类,在里面放一个B类的对象**(指针/引用),这样,由于继承的作用,心累可以使用A类的接口进行管理,需要使用B类的方法的时候,编写一个壳方法,在里面直接将实现委托给B类的对象中的相应的方法.由于对象的向上转型作用,可以进行多态调用**.**

**总结适配器模式:**

**在不改变原有类的前提下，调整原有类的接口**

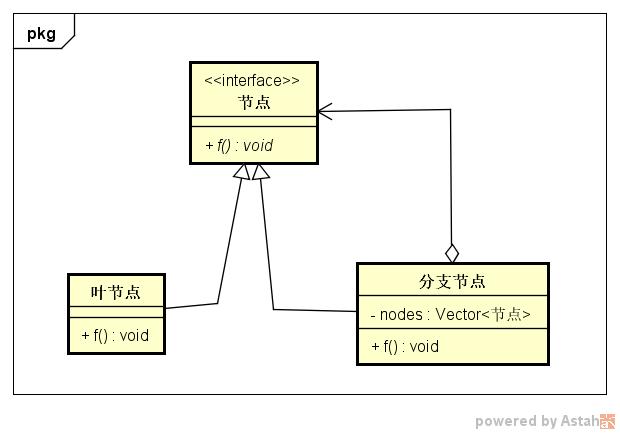
**适用性：**

**使用已有的,但类的接口不符合需要；**

**对象适配器,通常比类适配器更灵活；**

**2.3 合成模式**

* **问题: 有一些类的聚合层次,是树状的,但是出现在各个层次上的元素可能会跨越层次,比如电脑是由中央处理器,主板,显卡,电源等部件组成,其中显卡是由主板,电源,处理器组成.这个时候,横向的聚合关系可能会出现问题且不清晰.**
* **解决方案:在组件上方抽象出一个公共接口,让带有子系统的类来聚合这个抽象接口.**

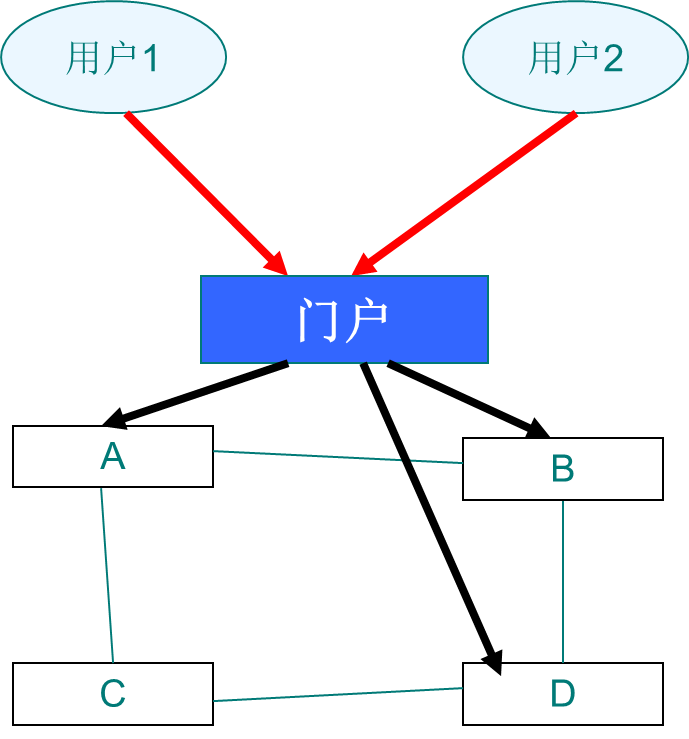


* **用树形结构表示线性的整体-部分关系,由于抽象公共接口的存在,访问任意一个结点时不需要判定或明确结点类型.**
* **AddChild(),RemoveChild( ),GetChild(int),GetChildren()等只对树枝节点有意义这些管理子节点的操作放哪个类里呢？**

**放抽象接口中(透明式)**

**放在对这些操作有意义的类中(安全式)**

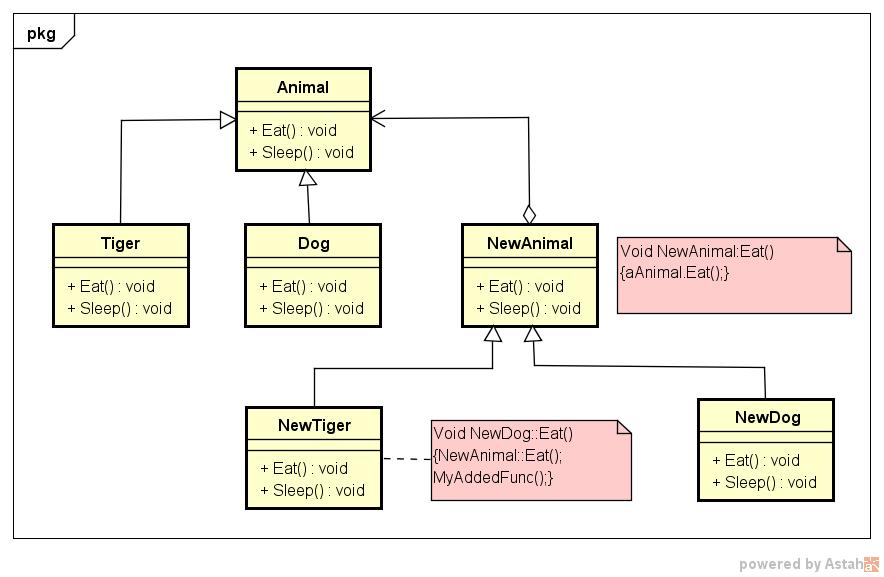
* 1. **门面模式(Facade Pattern)**
* **问题:一个系统内部有许多类,该系统通过这些类和它们之间的交互作用提供功能.用户必须清楚子系统中的各部分的职责,但是用户可能不想关心具体实现,而只是用系统中的已有功能;用户的访问,通常是单向的;**
* **解决方案:**

****

* **对用户隐藏了系统的实现，易用；**
* **降低了Client与子系统之间的耦合；**
* **仍然可以直接与子系统内部交互；**
* **适用性：**
  + **为复杂系统提供简单接口；**
  + **为多个子系统提供统一的“门面”，保持子系统的独立变化性；**
  + **便于构建层次化的系统；**

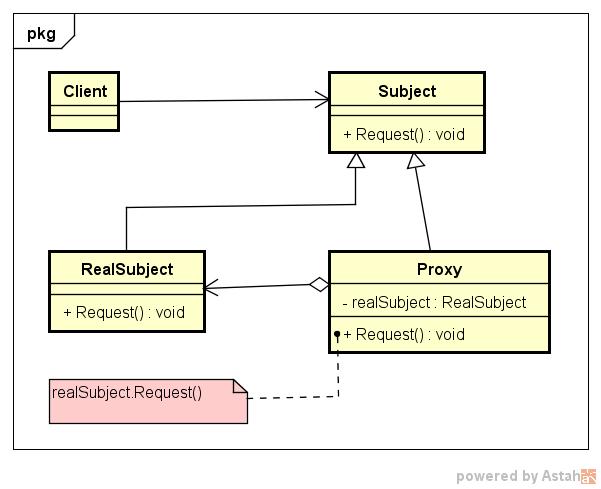
**2.5装饰模式(Decorator Pattern)**

* **问题:想要完成这样一种操作:给某个类的某个操作添加一些操作,原有的操作不想抛弃,而且希望通过对象的类型控制这种操作添加与否,添加几次,何时添加,而不想通过类继承的方式静态绑定到原有的类上.**
* **解决方案:将这种添加行为所需的增量操作(方法)写在另一个新的类中,为了能够复用原来的操作,这个新的类(或其接口)需要聚合被装饰类(或其接口),在新的方法实现中,通过委托聚合对象的方法复用原来的操作,并添加新的增量操作.为了能够统一管理,这个类需要成为被装饰类的平行类.通过控制传入装饰类的对象的类型,可以控制这个装饰绑定在什么对象上,从而控制何时装饰,装饰谁,装饰几次.**



**2.5 代理模式**

* **问题:在A对象访问B对象时，为B对象提供一个中介(代理)Proxy，使得A对象不再直接访问B对象，而是直接访问Proxy，由Proxy自行决定是否与B对象交互。这样就可以控制对B对象的访问。**



* **代理的种类:远程代理(隐藏目标对象的位置信息)**

**虚拟代理(隐藏具体的访问过程及具体实现细节),保护代理(保证访问的安全性),灵巧指针.**

class SmartPtr {  
public:  
    SmartPtr (A \*p): ptr(p) { }  
    ~ SmartPtr( ) { delete ptr; }

A\* operator->( ) { return prt;}  
private:

SmartPtr(const SmartPtr&);

SmartPtr& operator=(const SmartPtr&);

private:  
    A \*ptr;

};

**main() {**

**SmartPtr pt(new A); pt->f( );**

**}**

**对比：**

**main() {**

**A \* pt = new A; pt->f( );**

**delete pt;**

**}**

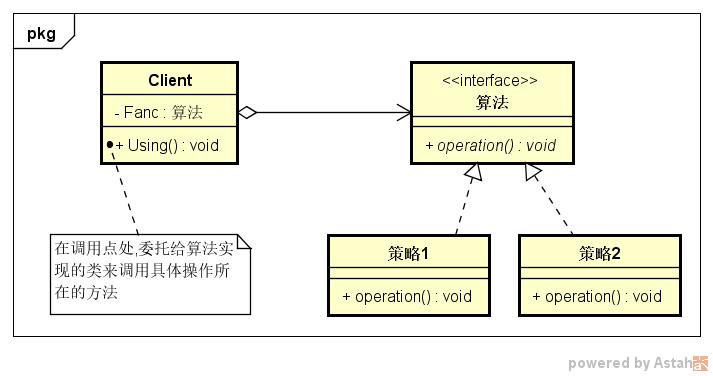
**2.5享元模式(flyweight)**

* **问题: 一个类的对象很多不便全部以对象的形式保存,对象粒度小(超轻量级),对象状态之间的差别不复杂.例如一个字处理程序中,每个字母都重复了很多遍,每次出现的区别很小(字号,颜色,位置等).**
* **解决方案:不知道………**

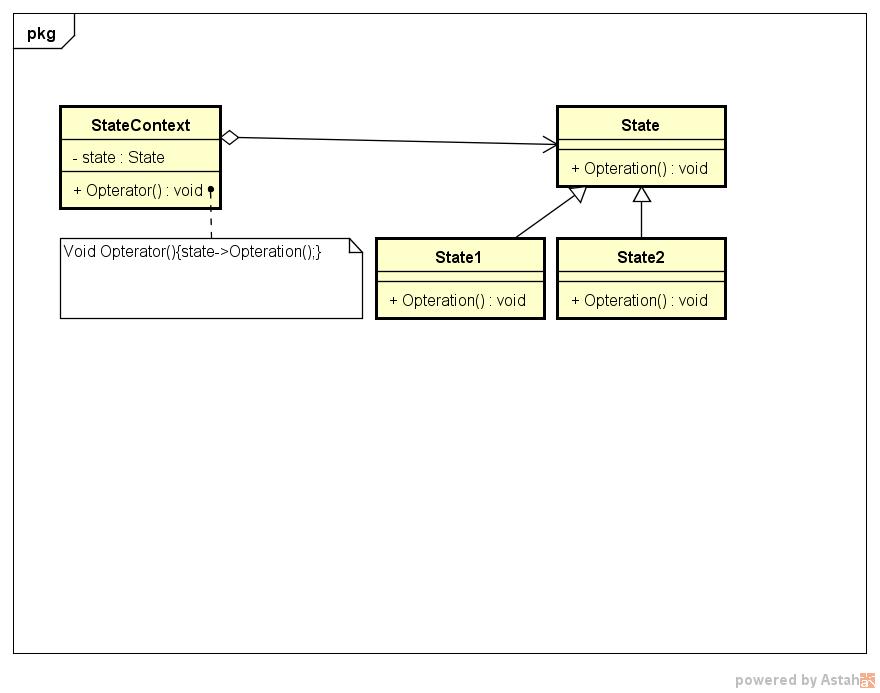
**3.行为型模式**

**3.1策略模式**

* **问题:定义多个实现或实现的一部分,并一一封装起来.在行为的动态执行过程中,可以替换各实现.这样，各实现可独立于使用者(客户)而变化.比如,一个人机对战游戏在游戏开始的时候创建了机器人玩家,要在不重新创建机器人(意味着游戏的重置)的情况下更换其AI算法以调整难度.**
* **解决方案:很自然的想到,既然要独立变化,就要把变化部分分离出来成为一个类,这样就可以使用继承的方法进行修改,要实现调用点的快捷切换,只需进行一个委托,通过调用点处的初始化参数和set方法就可以切换委托的具体对象.**



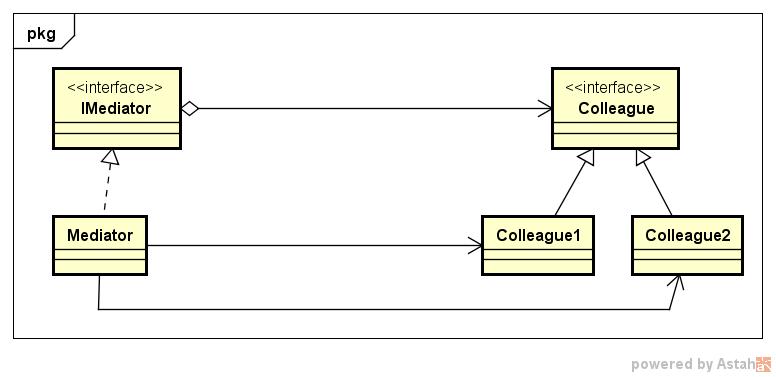
* **封装一部分实现细节**
* **允许在不新建上下文(Context)的情况下，改变行为。**
* **允许相同的行为有不同的实现，而且可动态地选择并改变。**
* **切换的具体细节需要商讨~**
  1. **状态模式**
* **问题:**一个对象的行为取决于它的状态,**且必须在运行时刻根据状态改变它的行为**;或者这个对象的某个操作中含有庞大的**多分支条件语句**,且这些分支实际上是**在对象的不同状态启动不同的功能**,这些状态通常是用**枚举变量**来区分的,甚至,这个对象的**很多方法都具有这样的特点:根据类中定义的枚举变量来执行不同的操作**.
* **解决方案:将类中的方法进行拆解分类,把所有那些在同一种状态下执行的动作放在一个类中,把不同状态下的这些类抽象出一个公共接口,在调用点聚合这个接口管理的引用,这样,原来用枚举管理的状态分支就变成了用对象的具体类型管理.**



* 将与行为相关的状态，单独抽象出来。
* 状态改变时，使用者不用判断状态再执行相应的行为。
* 状态的改变,可放在具体状态中实现,也可放在上下文
* 状态对象的创建与销毁
* **比较策略模式和状态模式**

3.3中介者模式

* 问题:定义好了一些列的类/对象,但是他们之间的通信和调用太复杂, 或者在某些对象中拥有过多的其他对象的引用,想把行为分布在多个类中但是并不像拥有太对子类.比如一个GUI程序的组件之间有一个制约关系(按A按钮之后B按钮不能按下,文本框有输入才能按下某个菜单项等等),直接编写会导致这些对象产生强关联而且很难从代码层面理解到底发生了什么.
* 解决方案:设置一个中介者,把所有这些对象之间需要沟通的地方全都放在中介者中,这样,所有的其他的类只需要知道中介者的存在就可以了.



* 好处
  + 将多对多的交互，改变成一对多的，易于维护和理解
  + 降低各Colleague间的耦合度
  + 集中处理对象间的交互和通信，不破坏对象的复用性
* 不足
  + 中介者类的复杂度很高,难以复用

3.4观察者模式

* 问题:多个对象间是一种一对多的关系,当一个对象的状态改变时,所有依赖于它的对象都将得到通知,并自动更新,例如,当网络时钟服务器的时间更新之后,所有的联网时钟的时间都会随之调整.又叫订阅发布模式.
* 解决方案:目标与观察者间的关联管理可直接关联,也可将关联关系存入一个Map表中**,**目标可以多个.即二者为多对多关系.但是一般为一对多.

