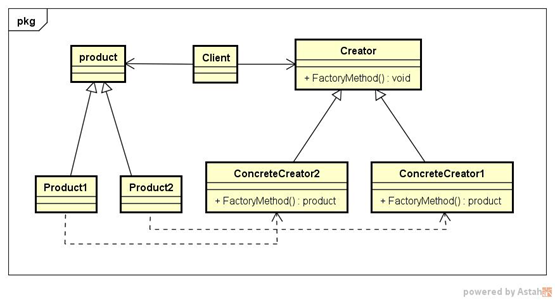
1. 创建型模式

创建型模式都是控制实例化的,比如接管实例化,禁止实例化,限制实例数量等.

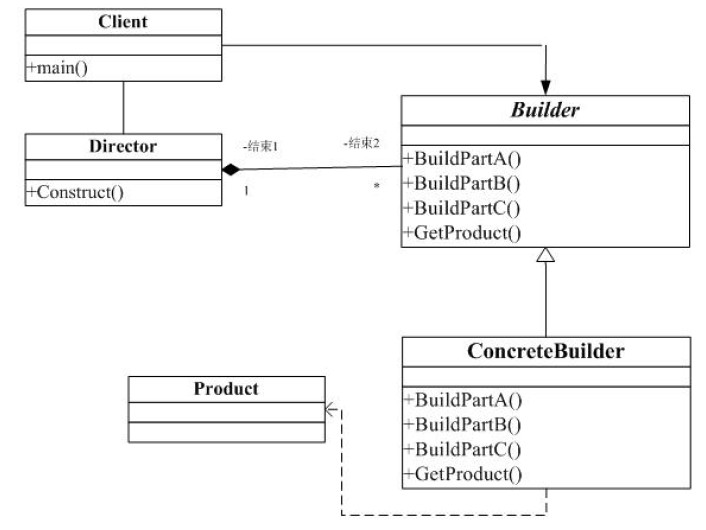
* 1. 抽象工厂模式

产品类具有一个类树,现在想让用户不能直接接触产品类,但是依然能得到产品类的实例,依照产品类的类树,建立对等的工厂类树,并向客户程序开放工厂类树,隐藏产品类树.使用户只能通过工厂类树中的某个节点类创建对应层级的产品实例.



* 1. 构造器模式

一个产品由若干零部件组成,每种零部件类都有可能发生更新,产品的组装过程(零部件的数量,顺序,组合形式等)也可能发生更新,现隔离组装过程和零部件的制造,将零部件的制造工作交给builder,通过重载builder,就可以改变零部件的制造;将零部件的组装交给Director,在Director中给出装配流程,用户并不需要知道这些细节,他只需要从Director那里输入他对这个复杂对象的期望,并接收到一个制造并组装好的复杂对象即可.



* 1. 工厂方法模式

和抽象工厂极为类似,只不过工厂方法模式管理的产品不是产品簇,工厂方法强调将实例的创建延迟到子类中,因此工厂类树中的创建函数是一个虚函数,工厂类树和产品类树保持对等演进,使用虚机制,在产品发生更新时工厂的调用点并不需要更改,自动就生成了对应的新产品,要想最大化利用这个特性,必须是单线继承的,一旦形成类树将会极其复杂.

简单工厂方法就是把控制构造的代码放进产品类本身,并屏蔽掉产品的构造函数.这个技巧在日常编程中很常用

* 1. 原型模式

原型模式允许用克隆代替新实例化,可以提高性能.在产品类向上抽象出原型类,并实现自身克隆方法,当需要与现有对象一致或类似的新对象时,进行克隆而不是拷贝构造.在一些语言的实现层面(例如java),这将会是一个内存层面的复制而没有发生构造函数的运行.

* 1. 单件模式

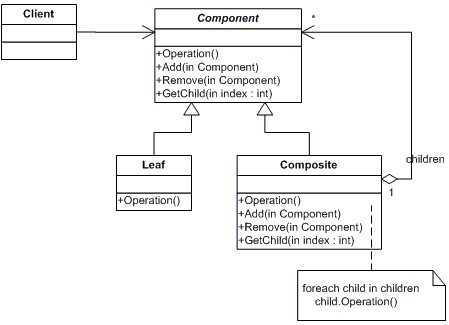
这个没啥好说的.

1. 结构型模式
2. 适配器模式

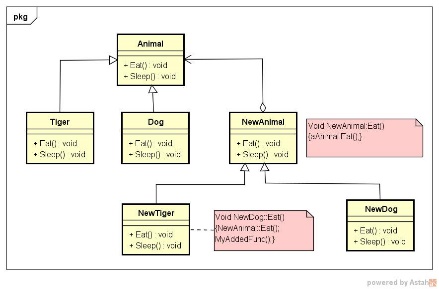
就和现实中的适配器或者称为转换器一样,例如一个typeC转HDMI转换器,我们使用这个东西的动机是:我们需要使用typec的外形和HDMI的功能,适配器总是这样工作的.我们有A类和B类,他们两个类型不兼容,但是我们就是想要一个这样的对象:具备与A兼容的类型却拥有B的方法,此时就得拿一个类C时继承他俩,然后我们在需要使用的类C中委托调用父类B的方法就行了,如果编程语言不支持多继承,那么就只能在需要类型兼容的那个类上抽象出上层接口,并令C类实现这个接口同时继承B类,或者直接继承A类,但是在C类里面放一个B类型的属性来实现委托,同时我们注意到,在可以进行多继承的语言中,我们完全可以用private继承A类.

1. 桥接模式
2. 组合模式

一个很好的例子是目录系统,文件件下面可以有子文件夹,也可以有文件,我们无法预知文件或文件夹的数量以及目录的深度,此时如果把不同层次的目录和文件分别处理将会是非常麻烦的,所以我们把文件和目录抽象出一个统一的上层类,并让这个上层类聚合目录,这样我们的客户端就能用统一的方式处理目录系统了,不用再面对复杂的组合体.



1. 装饰模式

****

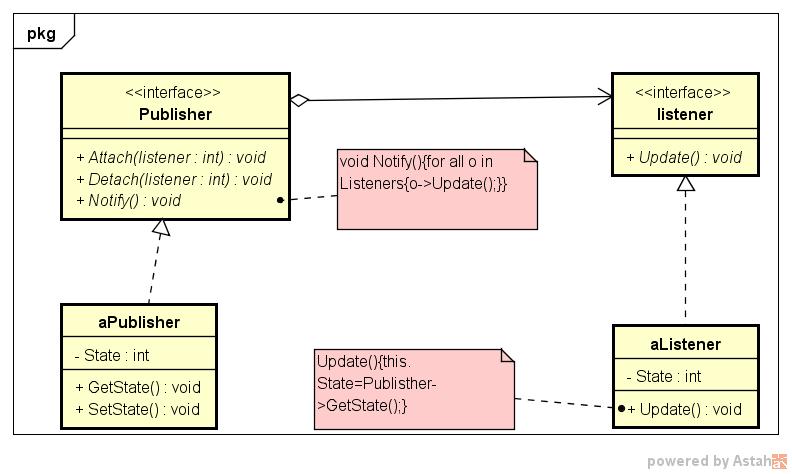
利用聚合替代继承,实现上在新函数中进行委托调用并添加额外代码.

1. 外观模式
2. 享元模式
3. 代理模式
4. 行为模式
5. 职责链模式
6. 命令模式
7. 解释器模式
8. 迭代器模式
9. 中介者模式

多对多变一对多,多个类之间/对象之间的强关联变成这些对象与中介者的关联.

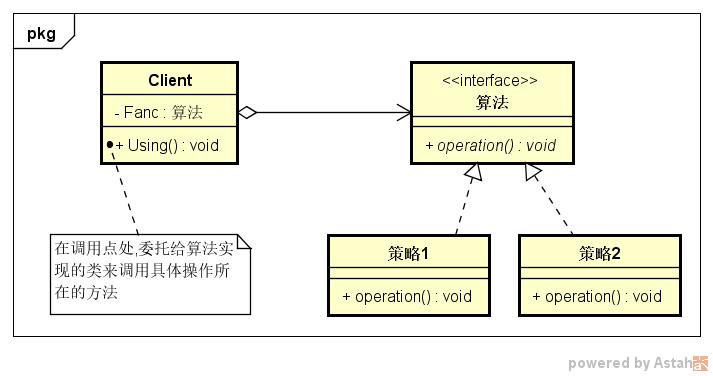
为了能使得多个对象和中介者独立变化,分别向上抽象也是可以的,但实际上中介者模式的核心就是添加一个第三者管理原来的复杂关系.

1. 备忘录模式
2. 观察者模式



如果一个对象的状态发生改变，某些与它相关的对象也要随之做出相应的变化。

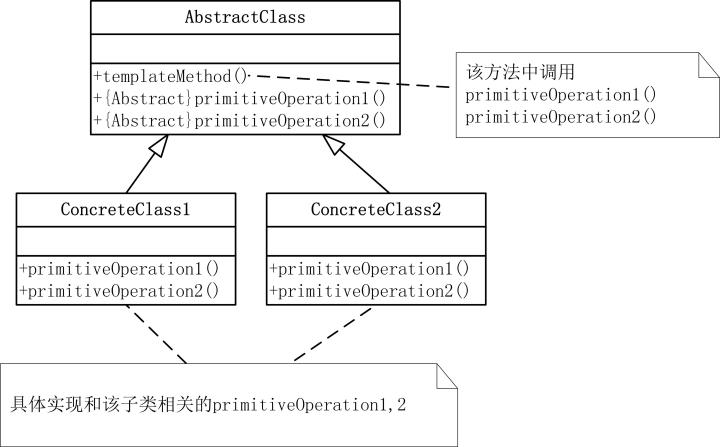
1. 状态模式
2. 策略模式

****

客户程序调用一个业务逻辑方法,但是这个业务逻辑方法可能会有很多版本的实现,可能会有后续更新.例如在一个人机对战游戏中,AI算法可能有多种实现,客户程序希望在运行时决定使用哪个具体实现.我们可以把这些实现通过switch case的方法硬编码在客户程序的调用点,但是这样显然不好.

为需要在运行时动态选择的那些方法抽象出统一接口,用户通过这个接口访问,可以提供统一的调用方式,通过将不同的对象绑定到这个接口类型的指针上,可以实现动态选择.

1. 模板模式



在父类中定义一个算法的骨架结构,并定义对外访问接口函数,子类中给出算法骨架中每一步的实现,好处是程序对于一个庞大而复杂的算法,可以单独变化其中的某些步骤.

1. 访问者模式