



现在有一个对象Object1 --- o1

 ---- 添加了功能

动态的第一个含义：不是预先做好的 运行期间添加的 ---- 加的时机是动态的

还有动态的第二个含义就是：每次添加的功能都不一样 这一次添加的功能3 下次添加的就是功能5

添加的功能本身也是动态变化的

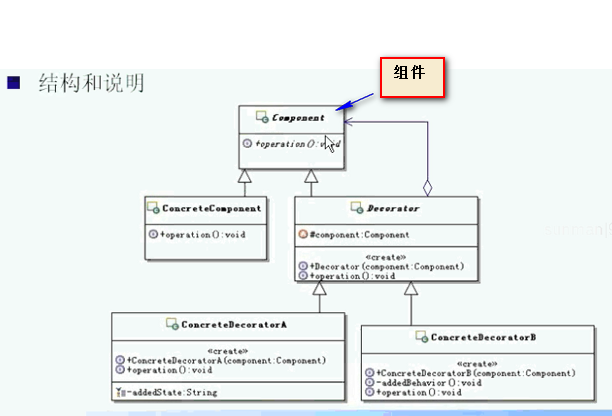
这就是动态的两层含义

装饰模式两个含义：为对象添加功能 ----- 添加功能的方式是动态 添加可能是动态的 添加的功能也是动态变化的

装饰模式为对象添加功能比生成子类灵活

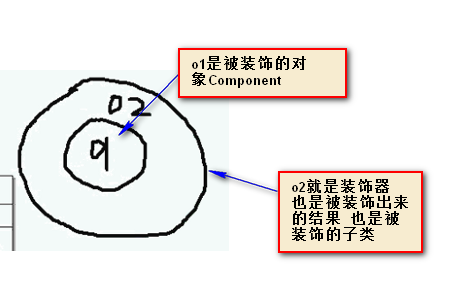
下面是装饰模式的类图

Component是被添加功能的对象

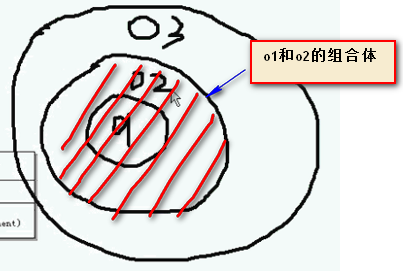


上面的o1就是被动态增添职责的对象

Decorator是用来装饰组件的装饰器



为啥一个装饰器又是一个component呢？原因就是 如果在o1 o2的基础上 再添加新功能 装饰出来03

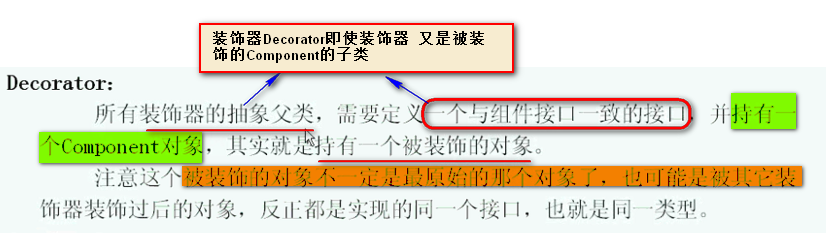


论证一下 为什么装饰器本身又是Component呢？

在O2这个装饰器的基础上 又添加了新功能变成了o3 ---- 那么o2相对于o2不就是Component么？ 这样 o2即是decorator 有事component了

所以 Decorator本身也是Component







组件可以被动态添加功能

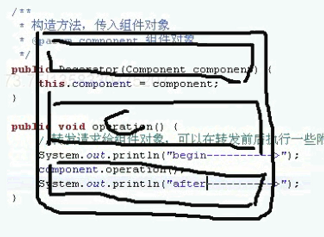




要在组件对象前后增加功能

持有component的原因就是要调用组件对象原有的方法 并进行增强



 这样 从外部看 是一个整体的功能 新的功能

这个就是参考示例代码

总的Decorator里面是缺省实现 真正的业务在ConcreteA和ConcreteB 才进行装饰

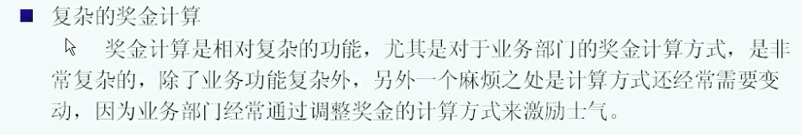


这样对装饰模式有一定的认识

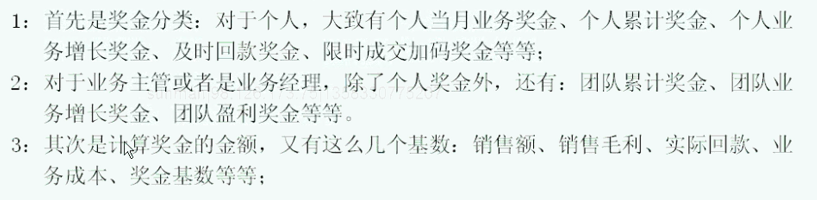
下面体会装饰模式 看下装饰模式在实际的开发中 如何利用

奖金计算是相对复杂的 尤其是对销售部门更复杂 计算方式经常需要变动 通过调整 奖金的计算方式 来激励士气

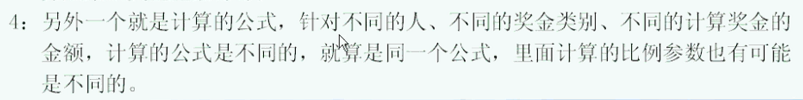
计算复杂 + 灵活的变动



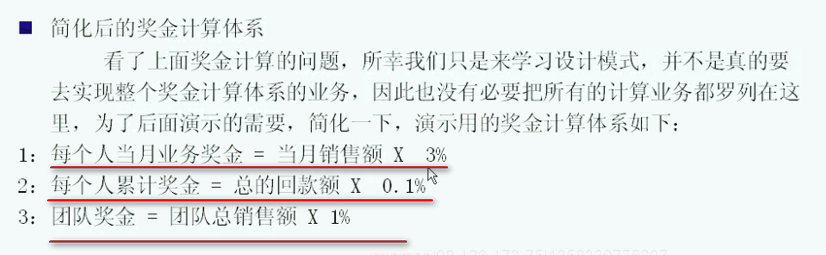
业务上的东西非常多 还要管理团队的奖金



总而言之 这些计算方式 从业务上看 就非常复杂



我们现在简化一下



业务有了 现在就是计算这个奖金

我们先不使用模式



模拟一个数据



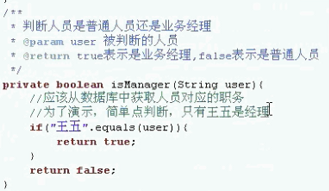
Map用来记录每个人的月度销售额







假定的累计业务额是10w块



这里面就认为王五是Manager



现在就从客户端来调用

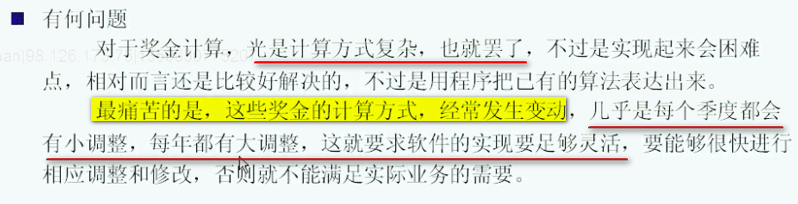




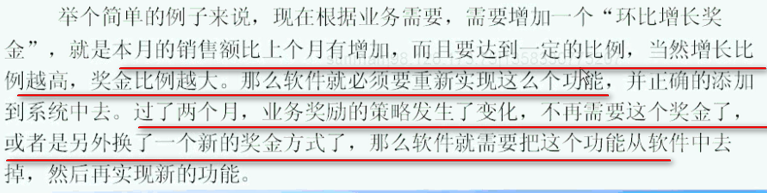
这是当月的奖金

看起来实现很简单

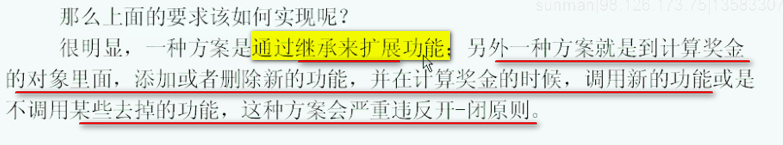
现在想想 有没有什么问题？



比如增加环比增长奖金

这个意味着一旦有变动 计算起来非常麻烦 修改原有的类的方法

CalcPrice就非常麻烦

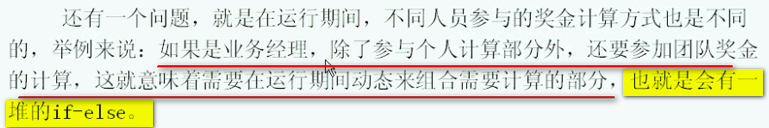


继承来扩展

覆盖这些monthPrice进行修改 需要的地方进行覆盖 不要的地方置为0 ---- 这完全是一种补丁方式的做法 ---- 实际上 不要就应该拿掉 不应该置为0

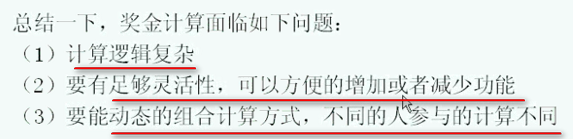
----- 继承的方式不太好

另外就是直接进去修改 加加减减 添加新功能 删除旧功能 这严重违反了开闭原则



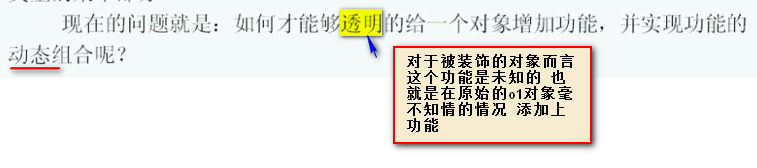


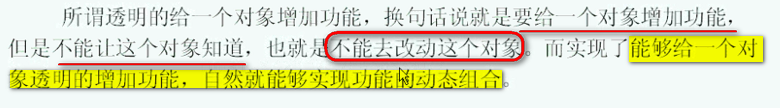
这些问题 就罗列出来了 目前奖金计算有如下问题



假设有一个计算奖金的对象 灵活增加或者减少功能 还能针对不同的人动态组合

这一看实现就依靠装饰模式来完成



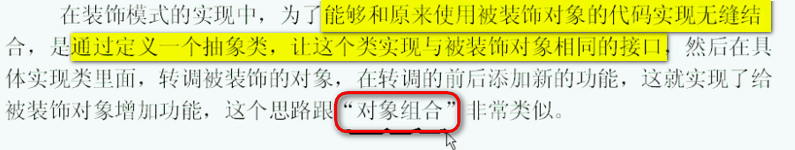


不能去改动这个对象

事实上 这个就是类似于功能的动态组合

在这个具体的实现上 为了让装饰对象和被装饰对象无缝结合 通过一个抽象类约束类型

实际上装饰模式的思路就是从对象组合来的



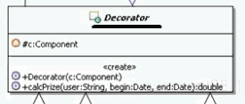
这是面向对象设计的基本功

在这样的情况下 大概思路就有了

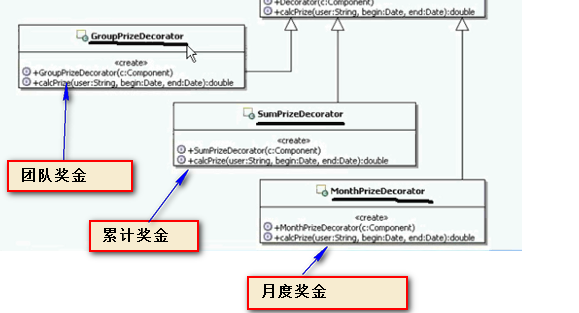
不需要的时候 直接去掉 ---- 原来的功能太烂 就废掉

谁来原始的被装饰的对象 ---- 就是ConcreteComponent

 --- 缺省情况下 奖金为0 没有任何奖金 这个用来充当被装饰得对象

用来表示每一种奖金的抽象装饰类

比如



以后还有别的奖金方式 比如 环比奖金 同比奖金

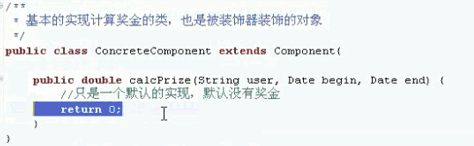
以后的变化 就是在Decorator下面增加就好

现在 代码实现



计算某人在某段时间的

看一下ConcreteComponent缺省实现 默认返回0



重点在装饰器这边



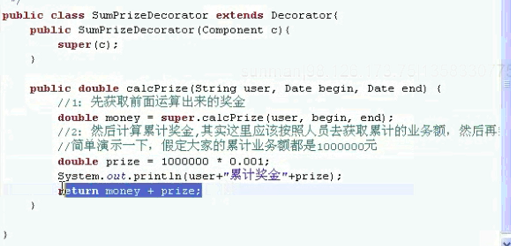
这样Decorator和Component的接口就统一了

每一个奖金的具体的实现 就相当于奖金的计算



这样设计就非常简单了

再看看累计奖金 先取出前面计算的奖金



最后看一下团队奖金



先获取团队的结果

【我们的root业务 每一个过滤 是不是可以做成装饰器 更灵活呢 ---- 有点可插拔的意思】

这些都有了之后 有一个问题 现在这样的计算奖金 都是单独的

现在 要组合这个过程

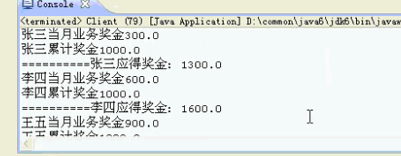
在客户端





这样非常的灵活 最后就看你如何的组合

运行看一下



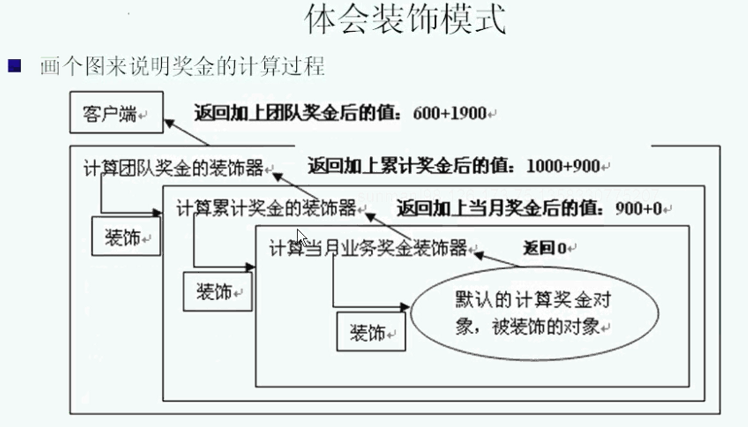


看看怎么运算



D2传进来的是d1

调用的是d1.calcPrice



调用的时候是从外层到内层 递归的调用

返回的时候 就从里面到外面一个个往外返回

【new的比较多 看看能不能把new的地方初始化】

【至于能否用到大数据web端的高并发 需要leader的第二部分的多线程知识】

装饰模式的调用过程是从外到内走

计算过程是从里到外返回

对于这个示例而言 是很简洁的

这个明显比计算奖金增加子类非常灵活

每一个奖金的计算规则都变成了装饰器

并且在client端进行了动态的组合 组合了几个

还可以调整顺序

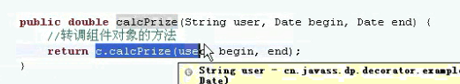
透明如何体现？

比如给c1增强功能的话



完全是在c1之外添加的功能

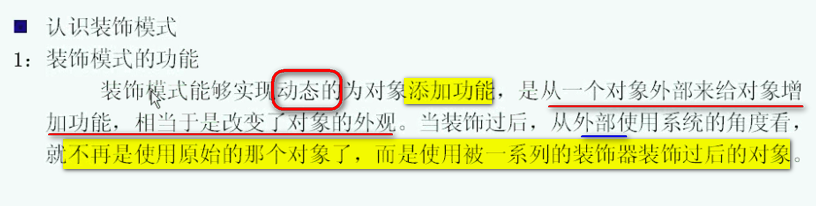




这个地方就是 调用的是c1的功能 ----- 原始对象c1是什么都不知情的 就是传进来给别人用的 c1只管自己内部的实现 别人怎么用c1 c1不清楚

所以是透明的

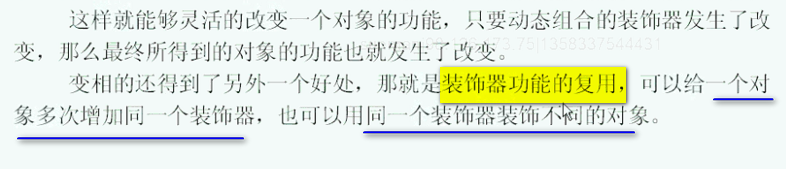
认识装饰模式



这样一来 就能灵活改变对象的功能

只要动态组合的装饰器的功能 最终的就发生了改变

这写都会导致功能的不同



可以把这个功能在不同的地方添加

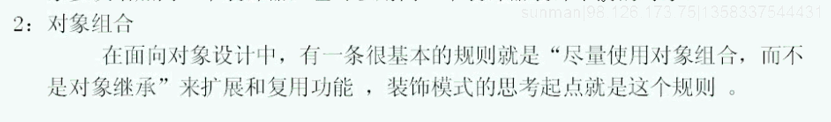
老板一高兴 就给张三的月度奖金加倍



再传一遍



使用一个装饰器多次复用



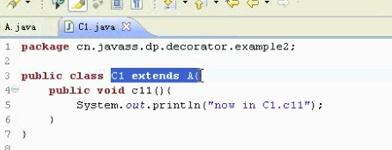
复合了多聚合 少继承

一个是面向接口 另一个就是多聚和 少继承

什么是对象组合？

现在有一个对象A

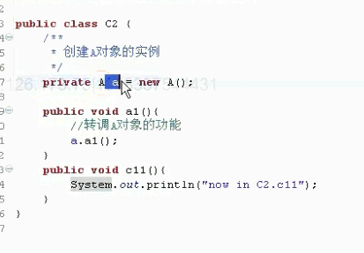
C1继承A



C1就是为了扩展A

这种是继承

对象组合 就是再C1中 不去继承



转过来调用 可以前后做点啥事

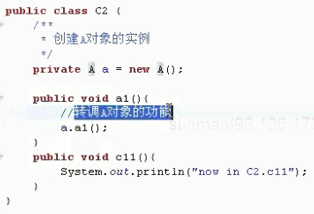
相当于给给原始的a1做扩展

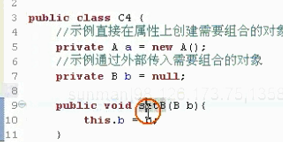
这个理解了过后 实际上 装饰模式的思考起点 就是通过对象组合得到的

这么做有一个额外的好处 java不能多继承

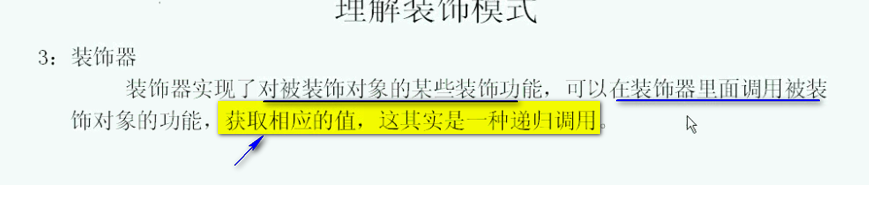
现在c2还有一个b1

Java就不能实现多继承 所以 这个时候 组合就起到了作用



 这个成员 通过容器向里面注入实例

下面看一下装饰器



11:05