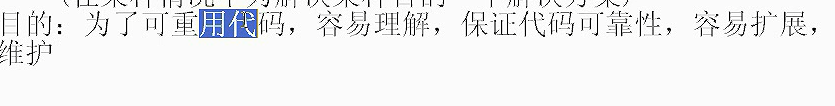


特殊的情况下 在某种情况下 为解决某种目的的解决方案而已

这是设计模式的概念

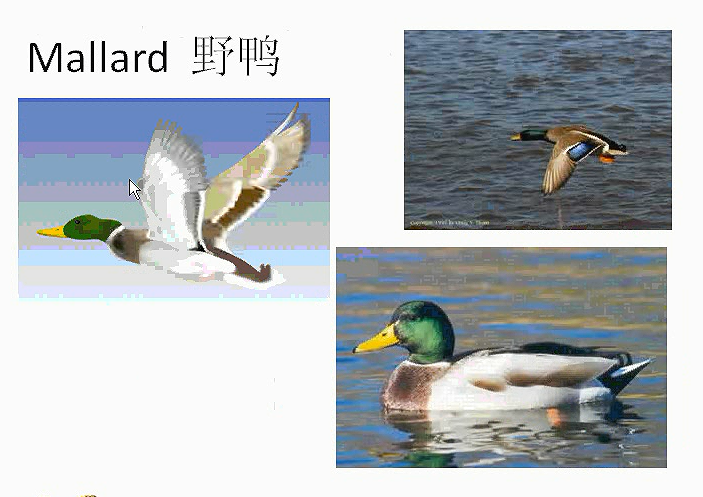
设计模式的用处 --- 为了让我们的代码可重用 让我们的代码更容易理解 ---- 而且还要保证代码可靠性 --- 容易扩展才可以 容易维护 这就是学设计模式的目的

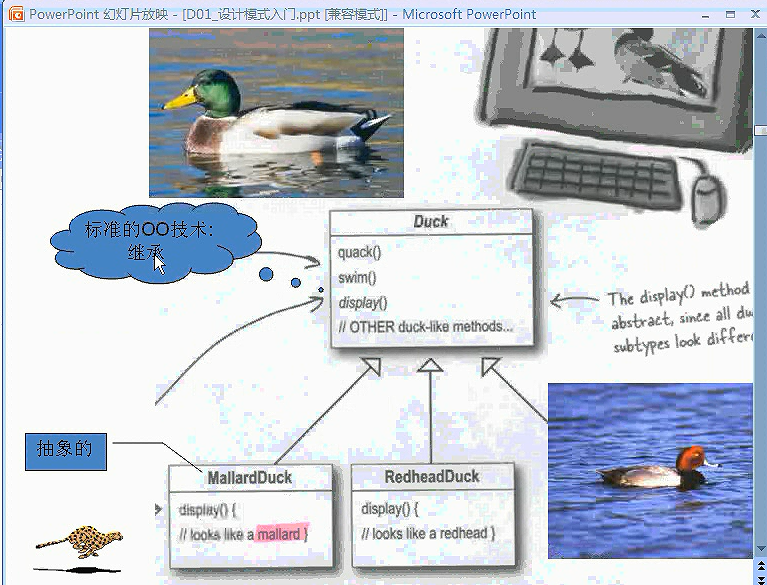


可读性 可扩展性 和 可维护性

------------------ 策略模式 -------------

有一家公司 做了一套的模拟鸭子的游戏 在游戏中出现各种鸭子 – 一遍游泳 一边叫



这款软件 里面 采用了标准的OOP的设计 面向对象的技术

抽象 封装 继承

每一种具体的鸭子 在显示的时候 每种鸭子的状态是不一样 显示的时候 有各自的display的方法

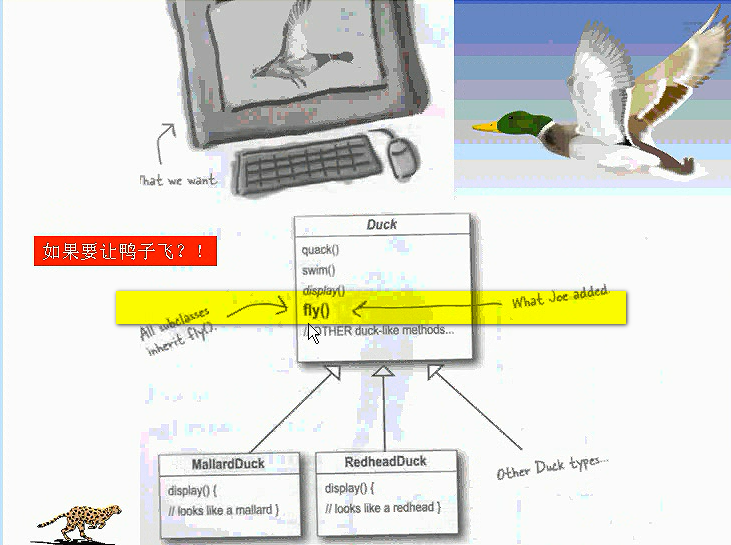
还有其他类型的鸭子

去年的时候 公司的竞争压力增加了 公司认为系统的该创新了

需要在会议上展示更加振奋的东西 来振奋人心

这个时候 让系统的鸭子会飞 --- Joe的经理说 给一周的时间 Joe可以完成 -----Joe面向OOP 没有困难

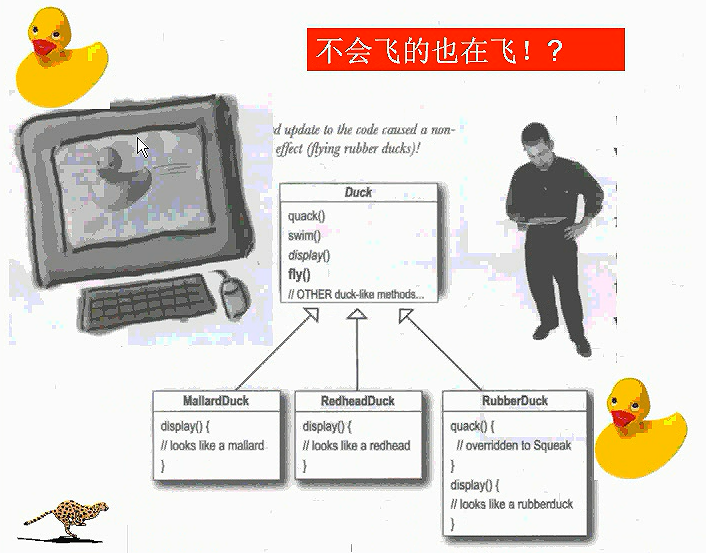
Joe在鸭子类中添加一个会飞的方法 所有的鸭子就会飞了



体现了继承的特色 --- 但是可怕的问题发生了

Joe接到了一个电话 正在开股东会议 发现很多橡皮鸭子在屏幕上飞来飞去 ---- 怎么回事？Joe忽略了一件事 不是所有的鸭子子类都会飞



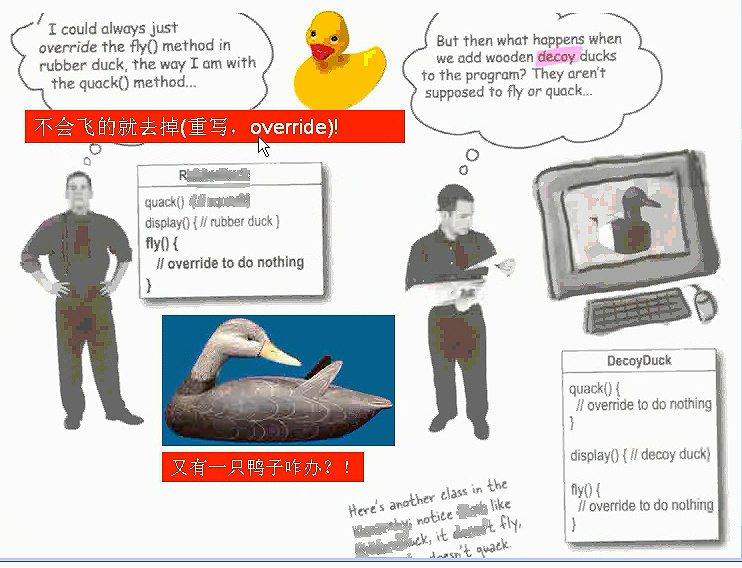
导致了 不该飞的鸭子 也在飞

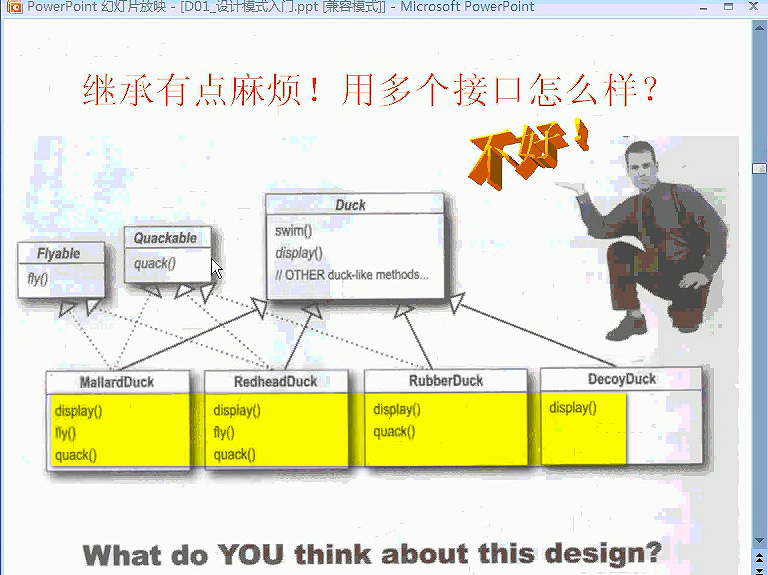
当你维护的过程中 为了复用的目的 使用继承 就不完美

Joe可以把橡皮鸭子的飞的方法覆盖掉

--- 可以采用覆盖的方法 把飞的方法 覆盖掉 就是不会飞

但是 涉及到以后新来的鸭子 加入新的子类 都不会飞 不会叫 --- 这种用重写的方法 覆盖 不是最好的方式 因为我们的代码 要扩展 要加新的鸭子 ----- 这样就会多个子类代码重复---- 很难知道所有鸭子的全部行为 ---- 有所改变 就会牵一发动全身





希望每6个月更新产品 增加新的功能 ---- 规格会常常改变

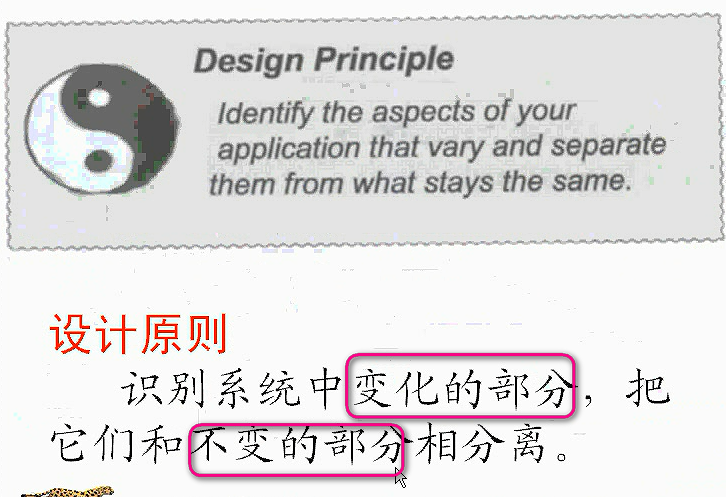
需要一个更清晰的方法 让某些鸭子即可以飞 又可以叫 ---- 就把这些动作从超类中提取【实际上是单一职责原则】 编程接口

这样会飞的鸭子 实现会飞的接口 用到哪个就实现对应的接口

这是一个超级笨的主意 ---- 这样重复的代码会变多 --- 需要对所有的鸭子的子类都要进行修改 --- 每一个都变成接口 这样工作量不是很少的

并非所有的子类都具有呱呱叫和飞行的行为 – 所以继承不是解决问题的办法 ---- 同时 使用多个接口会造成代码重复【因为你不用继承了】 ---- 只能算是一个噩梦到另一个噩梦

到底怎么办？用一个老方法



从这个设计原则入手

在鸭子类中 虽然鸭子游戏很好



软件开发的真理 就是变化 --- 不变的真理就是变化

把变化的部分 从不变的部分分离出来 ---- 飞和呱呱叫都不同 是变化的

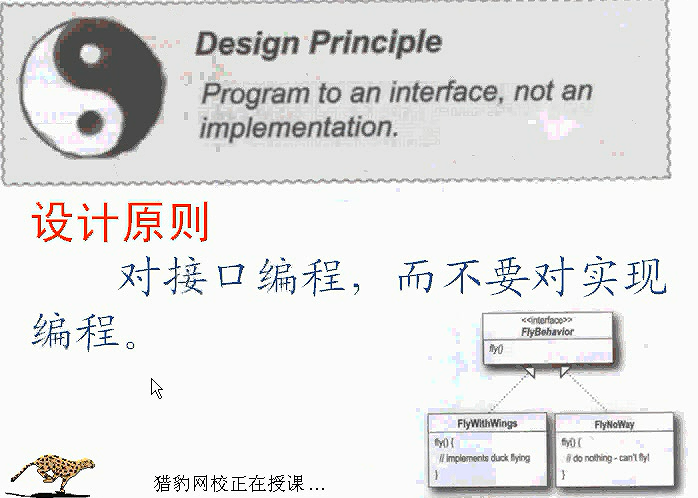
飞的行为也是变化的 -----

飞的行为和叫的行为都是变化 所以 从不变的部分分离出来

---- 如何设计？

希望一切都有弹性 ---- 正因为之前的鸭子的类没有弹性 -----

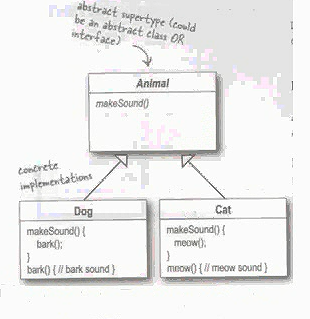
所以 第二个原则

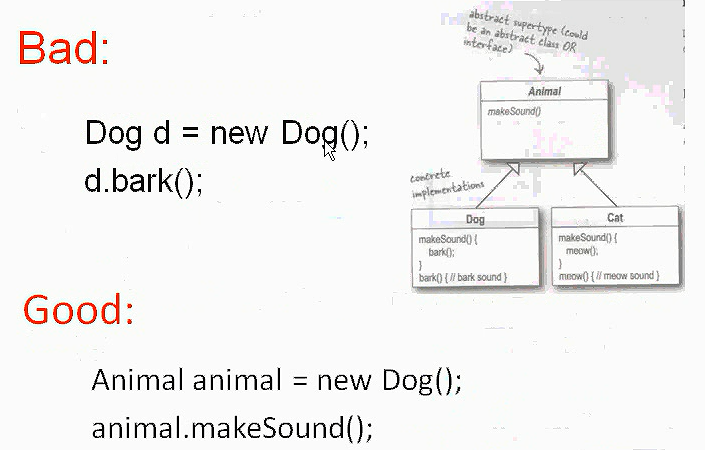


真正的意思 就是对超类编程

接口是一个概念 ---可以在不涉及到java interface ---- 利用多态的方式

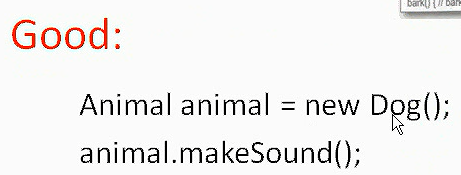
变量的声明类型是超类型的





 这个d就绑定死了 必须知道dog中的代码 才能使用

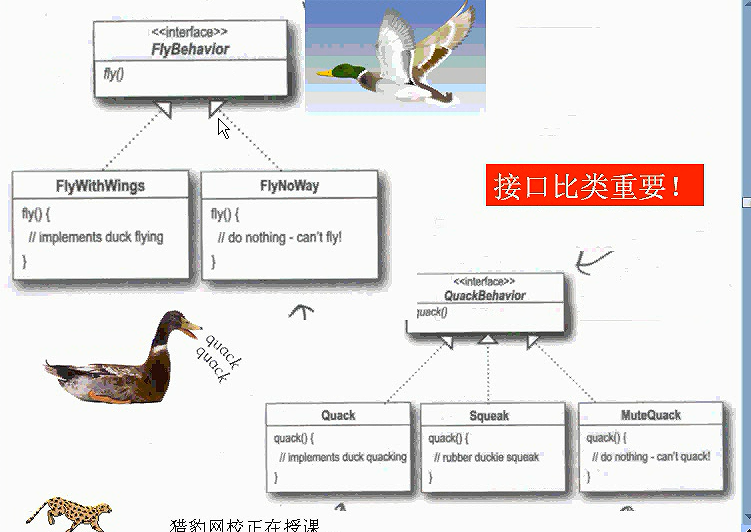
我们针对超类型编程



【计算机网络中 说到 接口就是一种规格】

声明类型是一种超类 指定的是一种超类型 到底是哪一种子类类型 自己的状态是不同的

对于飞和叫 这两组变化的部分 我们就设计成两组接口和子类

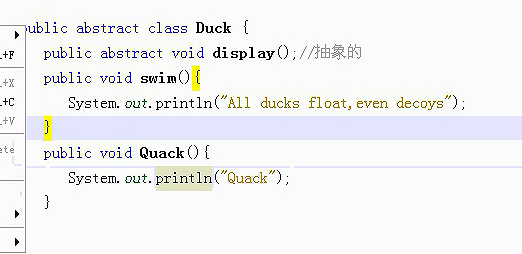


一种是关于飞的行为 一种会飞 另一种不会

呱呱叫也这样做

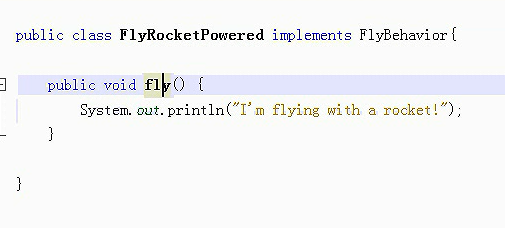
接口这种设计 飞行和呱呱叫可以被复用 和鸭子就没有关系了 --- 直接复用就好

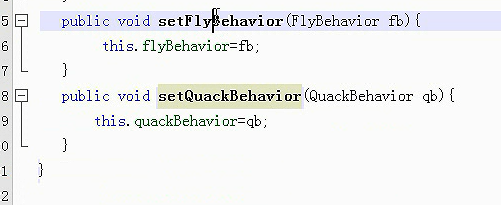
----- 现在开始编码 -----

 定义了一个抽象的鸭子类 分别是显示 游泳 和 呱呱叫

定义两个子类 野鸭 mallar 实现叫的行为 Quack()行为 输出一个呱呱叫

除了呱呱叫 还有一个吱吱叫的行为 实现





【分离出来的行为 和原来的】

这些代码就应用到了策略模式

客户端这部分