

面向对象设计模式 入门

曹东刚

caodg@pku.edu.cn

北京大学信息学院研究生课程 - 面向对象的分析与设计

<http://sei.pku.edu.cn/~caodg/course/oo>



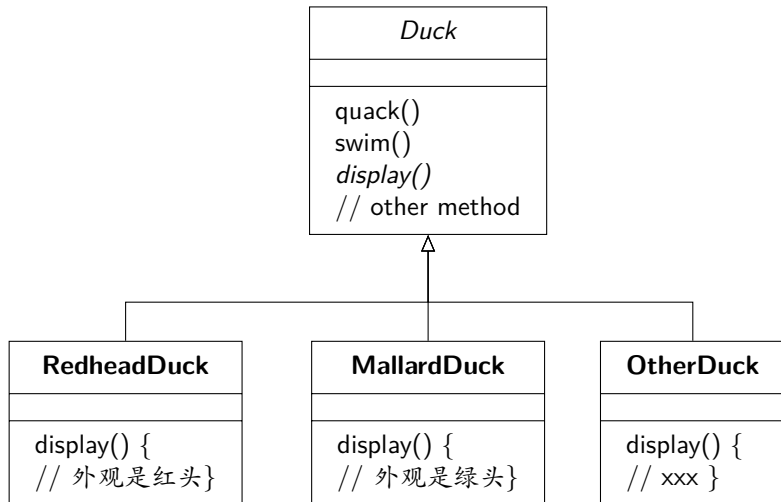
从一个简单的模拟鸭子应用开始

问题描述

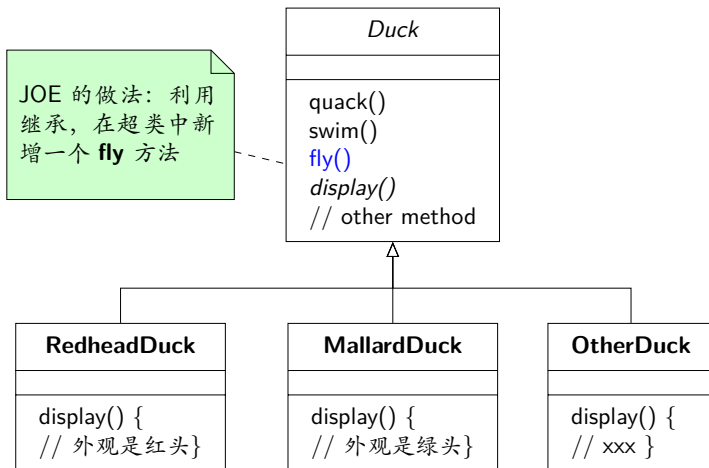
Joe 的公司做了一套模拟鸭子游戏: SimUDuck。游戏中会出现各种鸭子, 一边游泳戏水, 一边呱呱叫。

此系统的内部设计使用了标准的 OO 技术, 设计了一个鸭子超类 (Superclass), 并让各种鸭子继承此超类, 复用超类的代码。

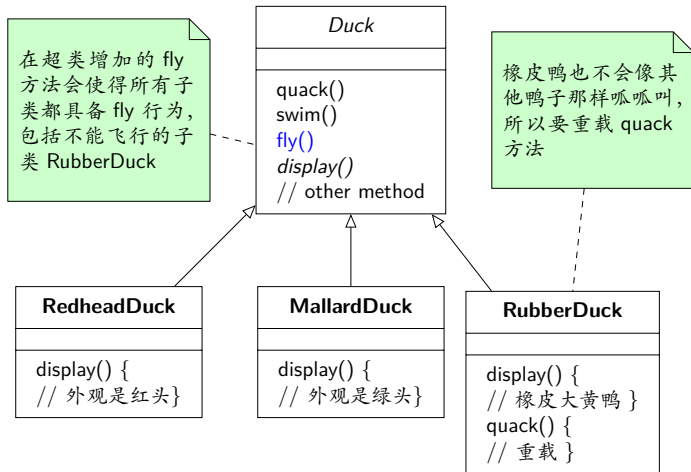
从一个简单的模拟鸭子应用开始



新的需求：让鸭子飞



问题：不会飞的鸭子也都会飞了！



如果用继承来解决

| RubberDuck |
|---|
| |
| <pre>quack() { // 重载, 吱吱叫 } display() { // 橡皮大黄鸭 } fly() { // 重载, 什么都不做 }</pre> |

问题：如果系统新增加一个既不会飞也不会叫的诱饵鸭
DecoyDuck 怎么办？

如果用继承来解决

DecoyDuck 不得不将超类中的 quack 和 fly 方法重载

| DecoyDuck |
|---|
| |
| <pre>quack() { // 重载, 什么都不做 } display() { // 诱饵鸭 } fly() { // 重载, 什么都不做 }</pre> |

新的问题：如果每隔半年更新产品，改变鸭子的行为，或者增加新的鸭子，会如何？

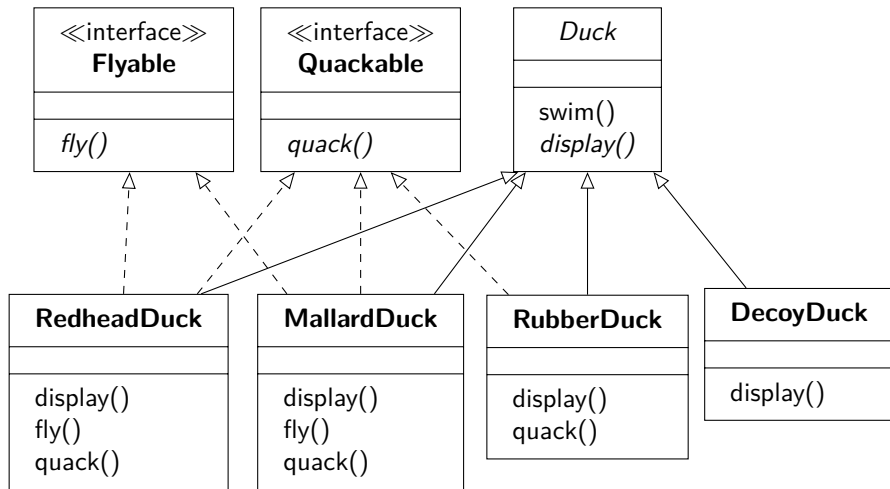
小结

为了“复用”目的而使用的“继承”机制，在解决系统维护的问题方面，没有想像中的那样便利

利用继承实现鸭子的行为，导致

- 代码在多个子类中重复
- 运行时的行为不容易改变
- 很难知道所有鸭子的全部行为
- 改变会牵一发而动全身，造成其他鸭子不想要的改变

让某些鸭子可飞可叫：采用接口技术



让某些鸭子可飞可叫：采用接口技术

问题：

- 重复代码变多
- 代码难以复用

期待：

能否有一种对现有代码影响最小的方式来修改代码？

答案：

良好的面向对象设计原则和方法

应对变化性

鸭子示例两种设计的问题

- 继承方式: 共同超类无法适应子类行为不断改变的情况
- 接口方式: 实现接口无法达到代码复用

问题的根源: 如何应对变化性!

设计原则 1: 封装变化性

找出应用中可能需要变化之处, 把它们独立出来, 不要和那些不需要变化的代码混在一起。

即: 系统中的某部分改变不要影响其他部分。

新的设计：分开变化部分和不变化的部分

Duck 类只有 quack、fly 等行为是易变化的

⇒ 将 quack 和 fly 相关的行为实现在另外的类里

⇒ quack 和 fly 行为有了自己的类

⇒ Duck 仍然是所有鸭子的超类

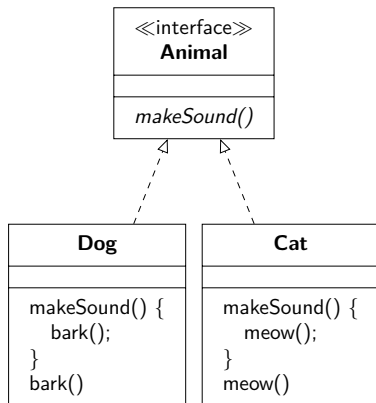
更进一步：能否让鸭子的行为可以在运行时刻动态指定？

设计原则 2: 针对接口编程

针对接口编程，而不是针对实现编程

设计行为类来实现行为接口，让易变的行为细节对对象透明

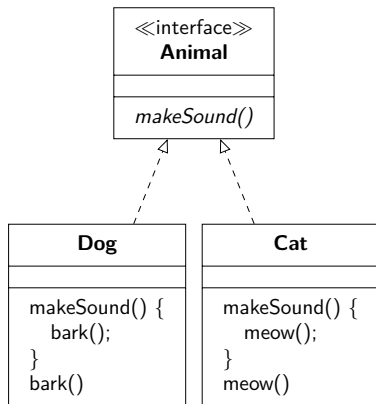
针对接口编程：实质是针对超类型编程



—— 针对实现编程 ——

```
1  Dog d = new Dog( );
2  d.bark( );
```

针对接口编程：实质是针对超类型编程



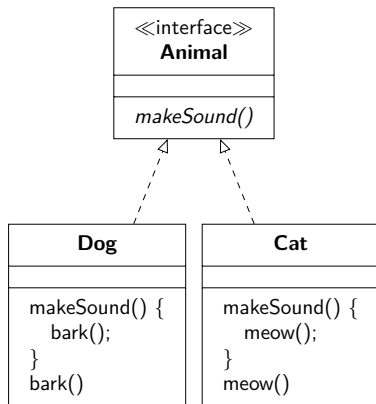
—— 针对实现编程 ——

```
1 Dog d = new Dog( );
2 d.bark( );
```

—— 针对接口编程 ——

```
1 Animal a = new Dog();
2 a.makeSound();
```

针对接口编程：实质是针对超类型编程



针对实现编程

```
1 Dog d = new Dog( );
2 d.bark( );
```

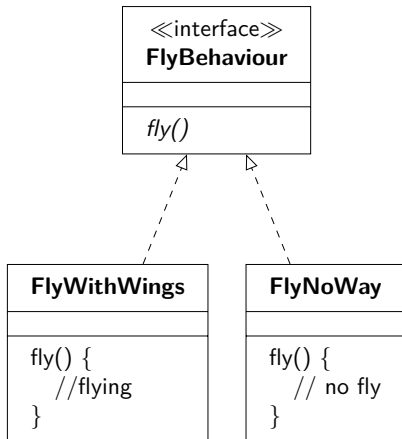
针对接口编程

```
1 Animal a = new Dog();
2 a.makeSound();
```

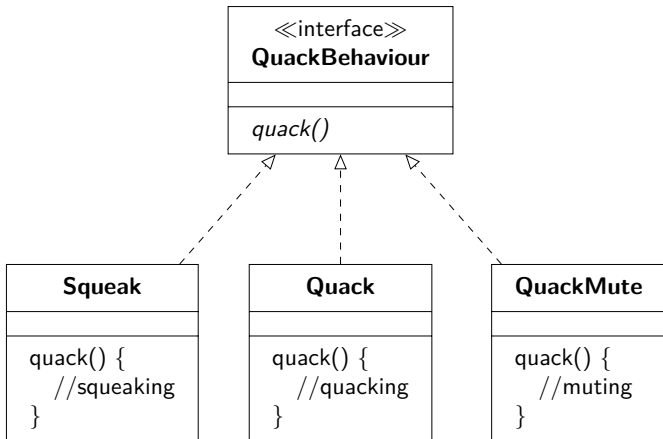
动态实例化

```
1 Animal a = getAnimal();
2 a.makeSound();
```

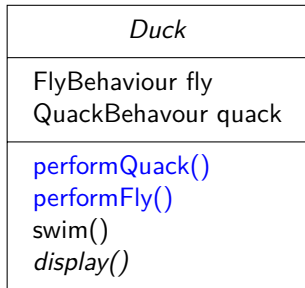
实现鸭子的行为



实现鸭子的行为



整合鸭子的行为



定义两个接口类型的实例变量, 运行时持有特定行为的引用

整合鸭子的行为

实现 performQuack

```
1 public class Duck
2 {
3     QuackBehaviour quack ;
4     // more variable
5
6     public void performQuack() {
7         quack.quack() ;
8     }
9 }
```

整合鸭子的行为

绑定具体的行为

```
1 public class MallardDuck extends Duck
2 {
3     public MallardDuck() {
4         quack = new Quack();
5         fly = new FlyWithWings();
6     }
7     public void display() {
8         System.out.println("I'm a real Mallard duck");
9     }
10 }
```

整合鸭子的行为

测试类

```
1 public class MiniDuckSimulator
2 {
3     public static void main(String[] args)
4     {
5         Duck mallard = new MallardDuck();
6         mallard.performQuack();
7         mallard.performFly();
8     }
9 }
```

动态设定行为

| <i>Duck</i> |
|---|
| FlyBehaviour fly QuackBehaviour quack |
| setFlyBehaviour() setQuackBehaviour() performQuack() performFly() swim() display() |

Duck 新增行为设定方法

```
1 public void setFlyBehavior(  
2     FlyBehavior fb) {  
3     fly = fb;  
4 }  
5  
6 public void setQuackBehavior(  
7     QuackBehavior qb) {  
8     quack = qb;  
9 }
```

动态设定行为

构造新的鸭子子类型

```
1 public class ModelDuck extends Duck
2 {
3     public ModelDuck() {
4         fly = new FlyNoWay();
5         quack = new Quack();
6     }
7
8     public void display() {
9         System.out.println("I'm a model duck");
10    }
11 }
```

动态设定行为

—— 构造一个新的 FlyBehaviour ——

```
1 public class FlyRocketPowered implements FlyBehavior
2 {
3     public void fly()
4     {
5         System.out.println("I'm flying with a rocket!");
6     }
7 }
```


动态设定行为

新的测试类

```
1 public class MiniDuckSimulator {  
2     public static void main(String[] args) {  
3         Duck mallard = new MallardDuck();  
4         mallard.performQuack();  
5         mallard.performFly();  
6  
7         Duck model = new ModelDuck();  
8         model.performFly();  
9         model.setFlyBehavior(new FlyRocketPowered());  
10        model.performFly();  
11    }  
12 }
```

再次总结

我们的目标是让系统更有弹性, 应对变化性
继承并不总是好的
聚合有时比继承更好

设计原则 3: 多用聚合

多用聚合, 少用继承

回到设计模式 (Design Pattern)



前例就是一个典型的设计模式: 策略模式 (Strategy Pattern)

策略模式

策略模式定义了算法族, 分别封装起来, 让他们彼此之间可以替换, 使算法的变化独立于使用算法的客户

- 软件设计是创造性、艺术性的工作
- 学过 OO 不代表能设计出良好的 OO 系统
- 良好的 OO 设计必须具备
可复用、可扩展、可维护的特性
- 模式能帮助我们建造出设计良好的 OO 系统

- 模式被认为是历经验证的面向对象设计经验
- 模式不是代码, 而是针对设计应用问题的通用解决方案
- 大多数模式和原则, 都着眼于应对软件变化
- 模式利于开发人员之间的沟通
- 学习模式能帮助开发人员成长, 提升层次

-  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-oriented Software. Addison Wesley Longman. 1995
-  Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns. O'Reilly Media. Oct 2004

