Lecture4 设计模式1

1. 什么是设计模式

定义

- 在某种背景下解决问题的方法
- 用于交流解决方案的语言
- 模式语言存在于很多问题中，但是我们主要关注设计方面

说明

- 设计模式不是思考的替代品
- 类名和目录结构并不等于良好的设计
- 设计模式有它的权衡
    - 它并没有完全消除交互中的复杂性，只是提供了一个结构
- 设计模式取决于编程语言
    - 特定的语言限制可能需要特定的模式（例如，与对象创建和销毁相关的模式）

设计模式的使用动机

- 它们可以提供一种设计经验的抽象
    - 可以经常作为一个可重复使用的经验基础
- 它们提供了一种讨论系统设计的通用语言
- 它们通过命名抽象来降低系统的复杂性
    - 从而提高了程序的理解能力，减少了一段新代码的学习时间
- 它们为类层次结构的重组或重构提供了一个目标

2. 策略模式 Strategy Pattern

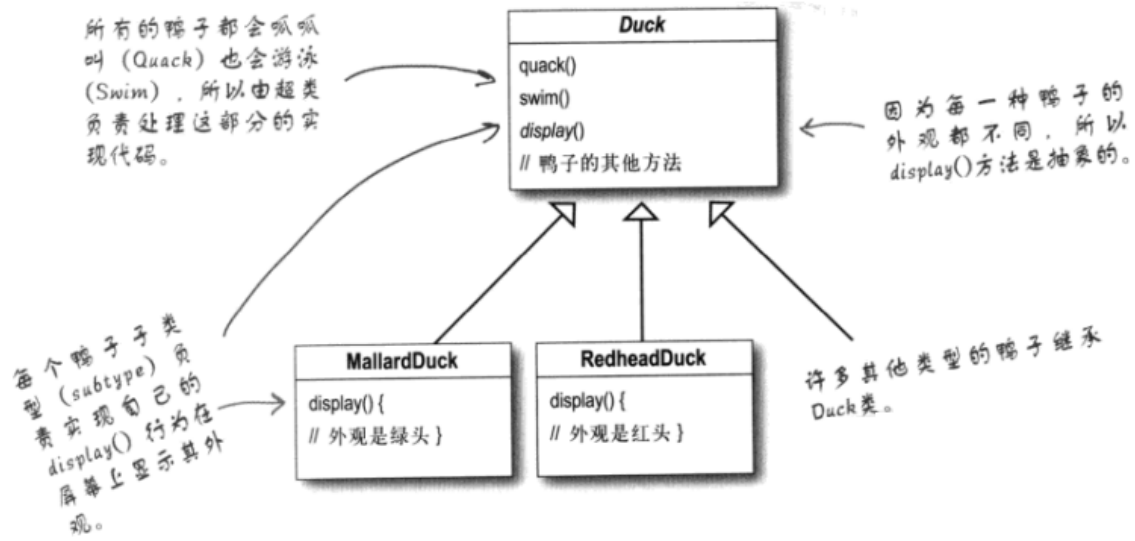
定义

> The Strategy Pattern defines a family of **algorithms**, **encapsulates** each one, and makes them **interchangeable**. Strategy lets the **algorithm vary independently** from the clients that use it.

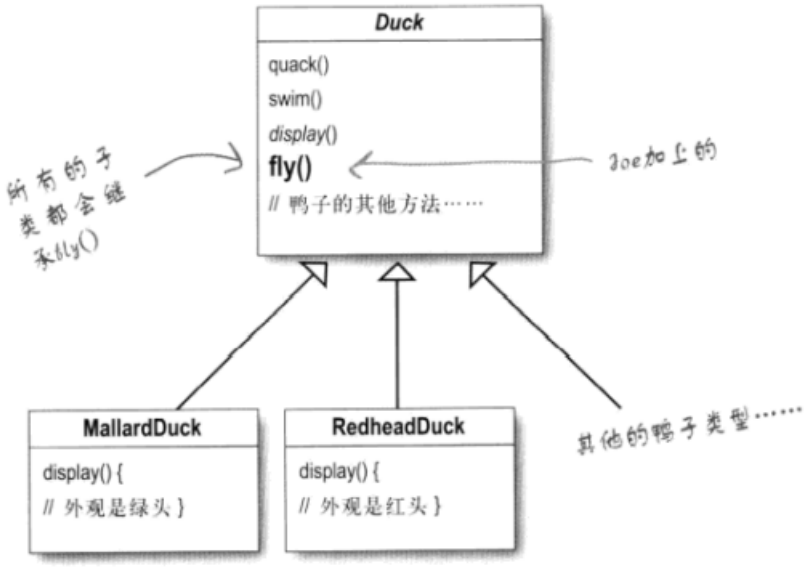
问题引入

为了制作一个模拟鸭子的游戏，游戏中会出现各种鸭子，有的会游泳戏水，有的会呱呱叫

现在设计的系统内部使用了标准的 OO 技术，设计了一个鸭子父类，并让各种鸭子继承这个父类

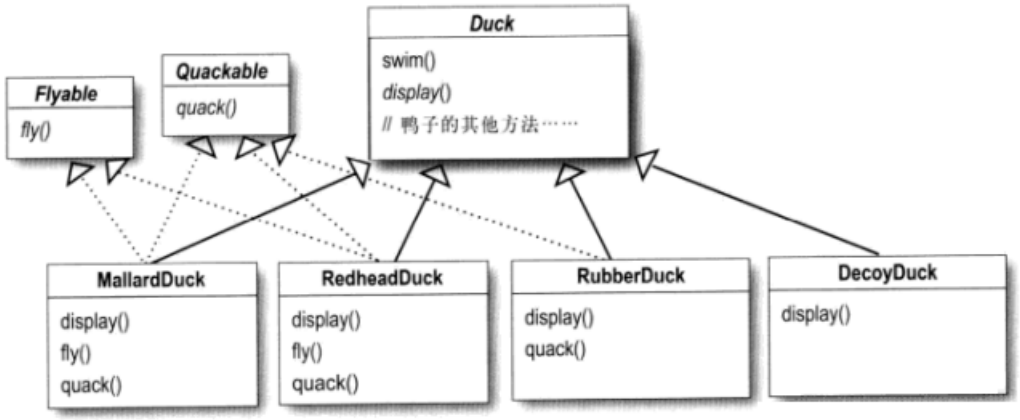


现在有了新的需求，需要**有些鸭子会飞，有些不会，有些会叫，有些不会**



这并不是很好的一种解决方法，因为鸭子的功能各异，会导致所有的继承类可能都要重复重写这些方法

如果设计成 `Flayable` 和 `Quackable` 接口，让不同的鸭子子类实现这些接口呢?



这更加的不好，因为所有的实现类都需要取重写这些方法，如果有 10 种鸭子都会飞，那就需要重写 `Flayable` 实现它的方法 10 次

设计原则

在软件开发上，**唯一不变的真理就是改变**，不论当初软件设计得多好，一段时间之后，总是需要改变

在鸭子示例中

- 使用继承并不能很好的解决问题，因为鸭子的行为在子类中不断的改变，使所有的子类都有这些行为是不恰当的
- `Flayable` 和 `Quackable` 接口开始似乎不错，但是接口不能达到代码的复用，这意味着无论何时你需要修改某个行为，必须追踪到定义的子类中去修改它

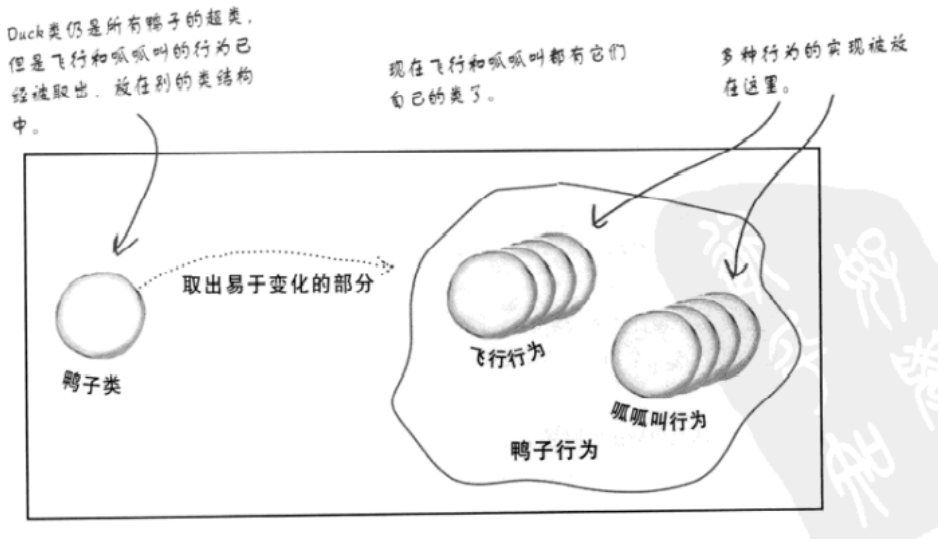
不过，有一个设计原则，恰好适用于这种情况

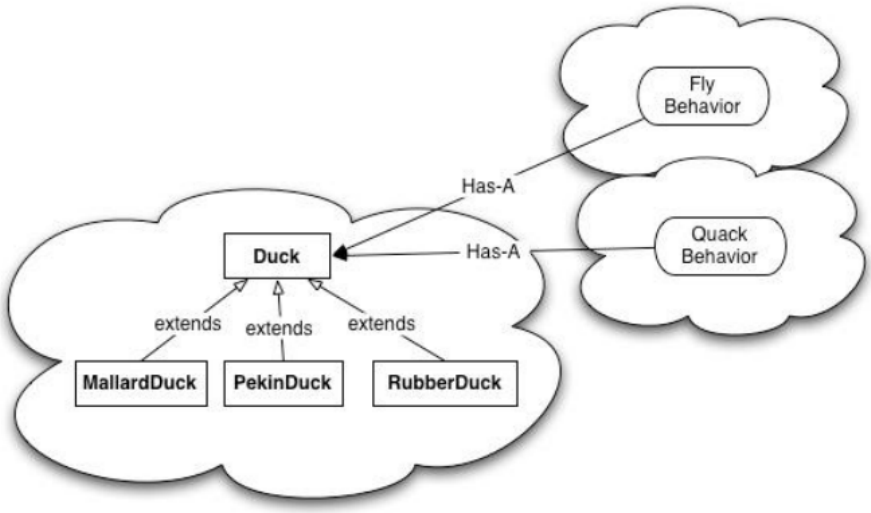
> Identify the aspects of your application that vary and separate them from what stays the same
> 找出应用中可能需要变化的地方，把它们独立出来，不要和那些不需要变化的代码混在一起

- 找出并封装那些经常变化的部分
- 更倾向于组合，而不是继承

在示例中，**鸭子的行为**是一个不断变化的部分，应该把它们取出来

准备建立两种类，一种是与 `fly` 相关的，一种是与 `quack` 相关的，每组类将实现各自的具体行为





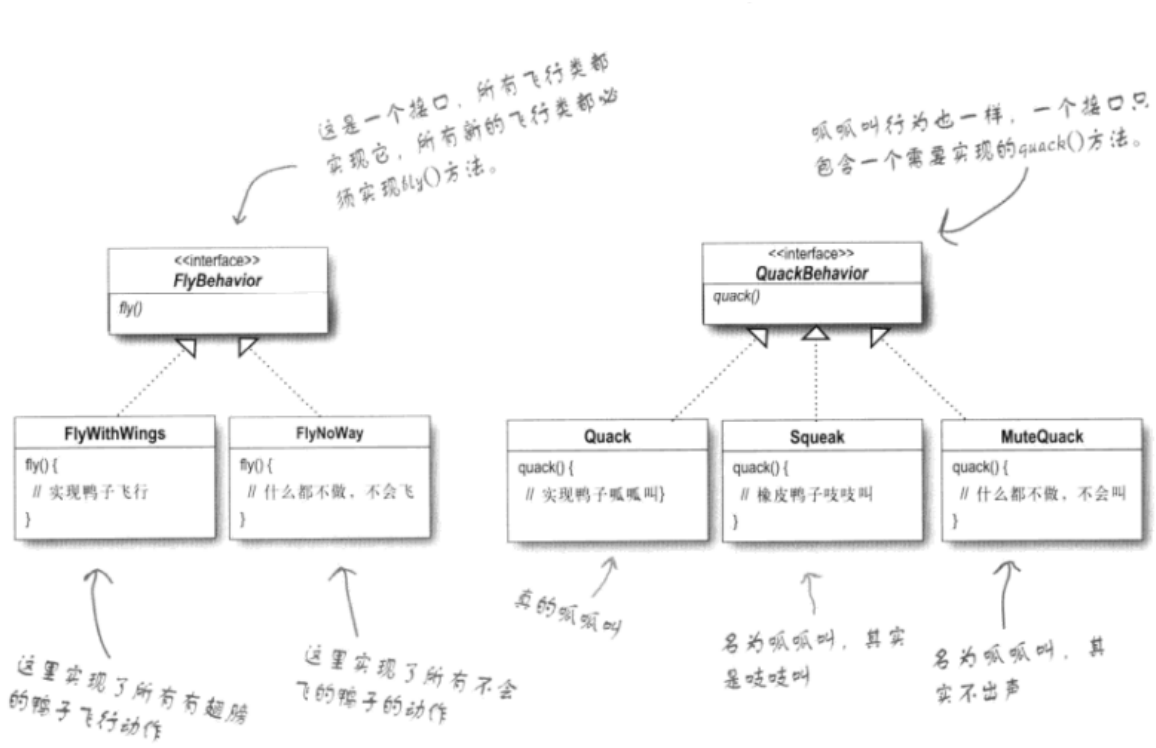
我们希望这些行为更有弹性，比方说，我们想要产生一个新的绿头鸭实例，并且指定特定的 `fly` 行为给它，干脆让它的行为可以动态的改变

> Program to an interface, not to an implementation
> 面向接口编程，而不是面向实现编程

我们用接口代表每个行为，比如 `FlyBehavior` 和 `QuackBehavior`，而行为的每个实现都将实现其中的一个接口

这次，鸭子类不会负责实现 `FlyBehavior` 和 `QuackBehavior` 接口，而是制造的一组专门的类负责实现这些接口，这些类我们可以把它们称为**行为类**

在新设计中，鸭子的子类将使用接口所表示的行为，而**实际的实现不会被绑死**在鸭子的子类中



鸭子类**委托 delegate**了 `fly` 和 `quack` 的行为

- 这样的设计，可以让飞和叫的动作被其它的对象复用，而它们的行为与鸭子类本身无关
- 我们可以新增一些飞行行为，既不会影响到既有的行为类，也不会影响“使用”到飞行行为的类

`has-a` 关系可能比 `is-a` 更好，每一个鸭子有一个 `FlyBehavior` 和一个 `QuackBehavior`，鸭子委托它们进行处理

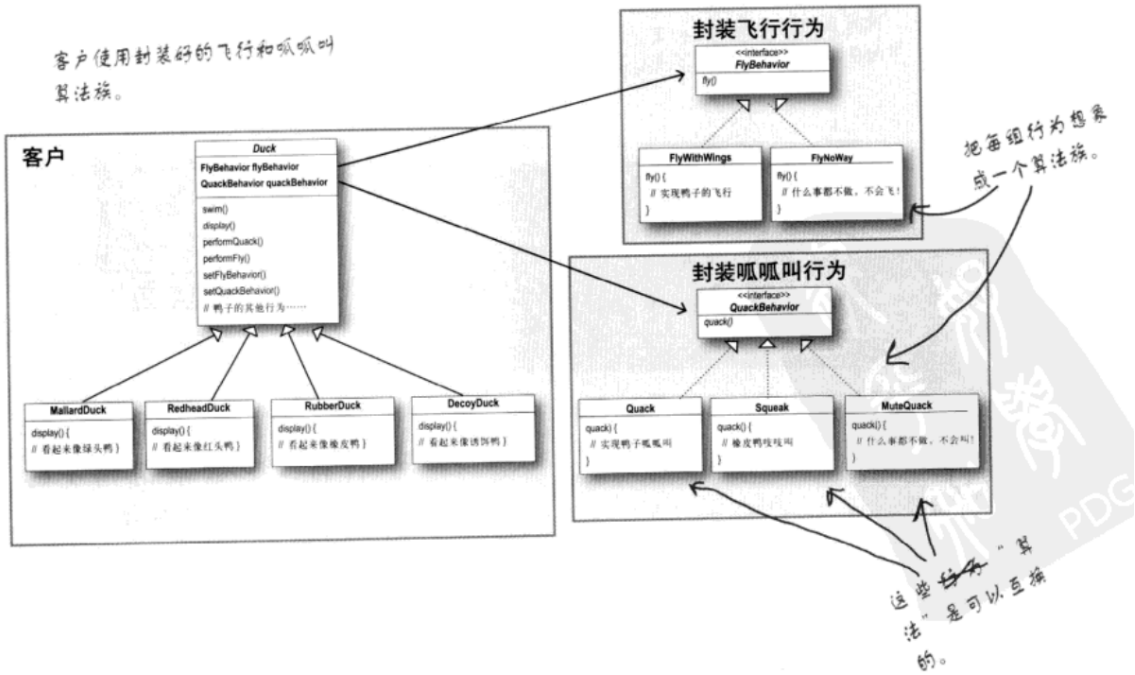
当你将两个类组合起来使用，这就叫做组合 composition，与继承不同，鸭子的是适当的拼装成的

> Favor composition over inheritance
> 多用组合，少用继承

组合建立的系统具有很大的弹性，不仅可以将**算法族封装成类**，更可以“**在运行时动态地改变行为**”，只要组合行为对象符合正确的接口标准即可

实现

UML



Duck 类

```java
public class Duck{
    QuackBehavior quackBehavior; // 每个鸭子都会引用实现 QuackBehavior 接口的对象
    FlyBehavior flyBehavior;

    // 鸭子对象不直接处理 quack 行为，而是委托给 quackBehavior 处理
    public void performQuack(){
        quackBehavior.quack();
    }

    // 鸭子对象不直接处理 fly 行为，而是委托给 flyBehavior 处理
    public void performFly(){
        flyBehavior.fly();
    }
}
```

QuackBehaivor 接口和它的实现类

```
public interface QuackBehavior{
    public void quack();
}

public class Quack implements QuackBehavior{
    public void quack(){
        System.out.println(“Quack”);
    }
}
```

Duck 的子类

```
public class MallardDuck extends Duck {
    public MallardDuck() {
        quackBehavior = new Quack(); // 绿头鸭使用 quack 类处理呱呱叫
        flyBehavior = new FlyWithWings(); // 使用 FlyWithWings 作为 FlyBehavior 类型
    }
    public void display() {
        System.out.println(“I’m a real Mallard duck!”);
    }
}
```

测试类

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
    }
}
```

动态实现

不仅如此，行为类甚至可以自动组装

Duck 类

加入两个新方法

```
public abstract class Duck{
    QuackBehavior quackBehavior; // 每个鸭子都会引用实现 QuackBehavior 接口的对象
    FlyBehavior flyBehavior;

    public void SetQuackBehavior(QuackBehavior qb){
```

```
        quackBehavior = qb;
    }

    public void setFlyBehavior(FlyBehavior fb){
        flyBehavior = fb;
    }

    // 鸭子对象不直接处理 quack 行为，而是委托给 quackBehavior 处理
    public void performQuack(){
        quackBehavior.quack();
    }

    // 鸭子对象不直接处理 fly 行为，而是委托给 flyBehavior 处理
    public void performFly(){
        flyBehavior.fly();
    }
}
```

之后，我们可以随时调用这两个方法改变鸭子的行为

测试类

```
public class MiniDuckSimulator {
    public static void main(String[] args) {
        Duck mallard = new MallardDuck();
        mallard.performQuack();
        mallard.performFly();
        Duck model = new ModelDuck();
        model.performFly();
        model.setFlyBehavior(new FlyRocketPowered()); // 改变模型鸭的行为
        model.performFly();
    }
}
```