Object Oriented Analysis & Design
面向对象分析与设计

Lecture_10 GOF设计模式（二）

1）策略模式　　2）工厂

主讲: 姜宁康 博士

- 2、修改后的设计方案…

复习：面向对象设计原则

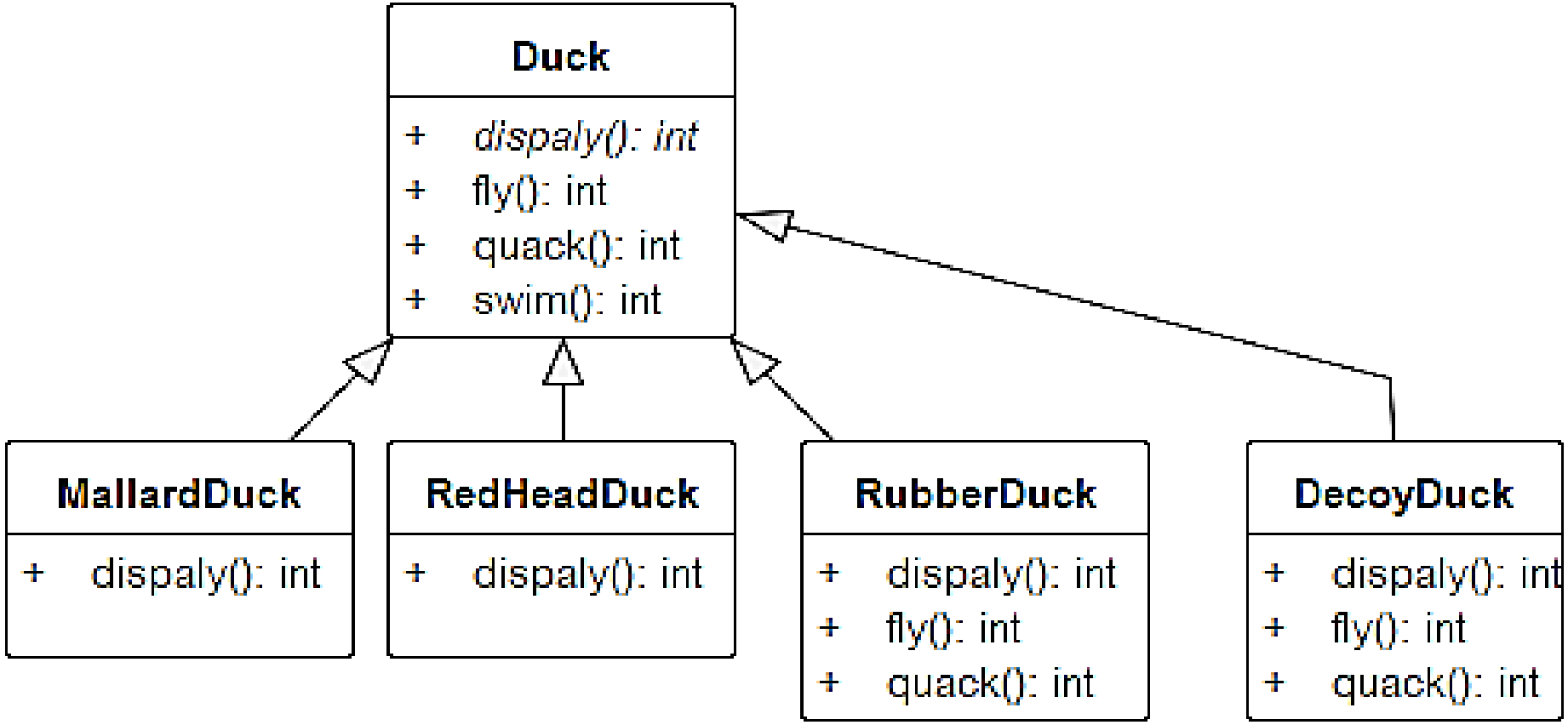
- 1、把变化的部分，封装起来

- 2、面向接口设计（编程），而不是面向实现
  Program to an interface，not to an implementation

请同学们思考一下，你会如何修改设计方案?

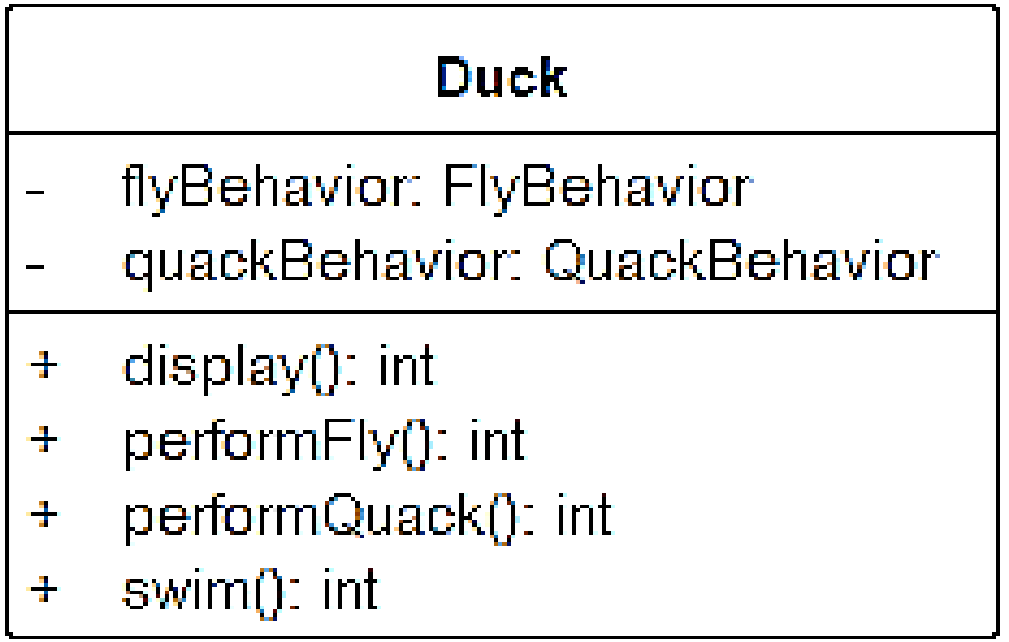
2.1 新的设计

- 原来的设计
  - Duck自己具有行为 fly()、quack（）
  - 子类有特殊性的时候，子类覆盖父类的实现



2.1 新的设计

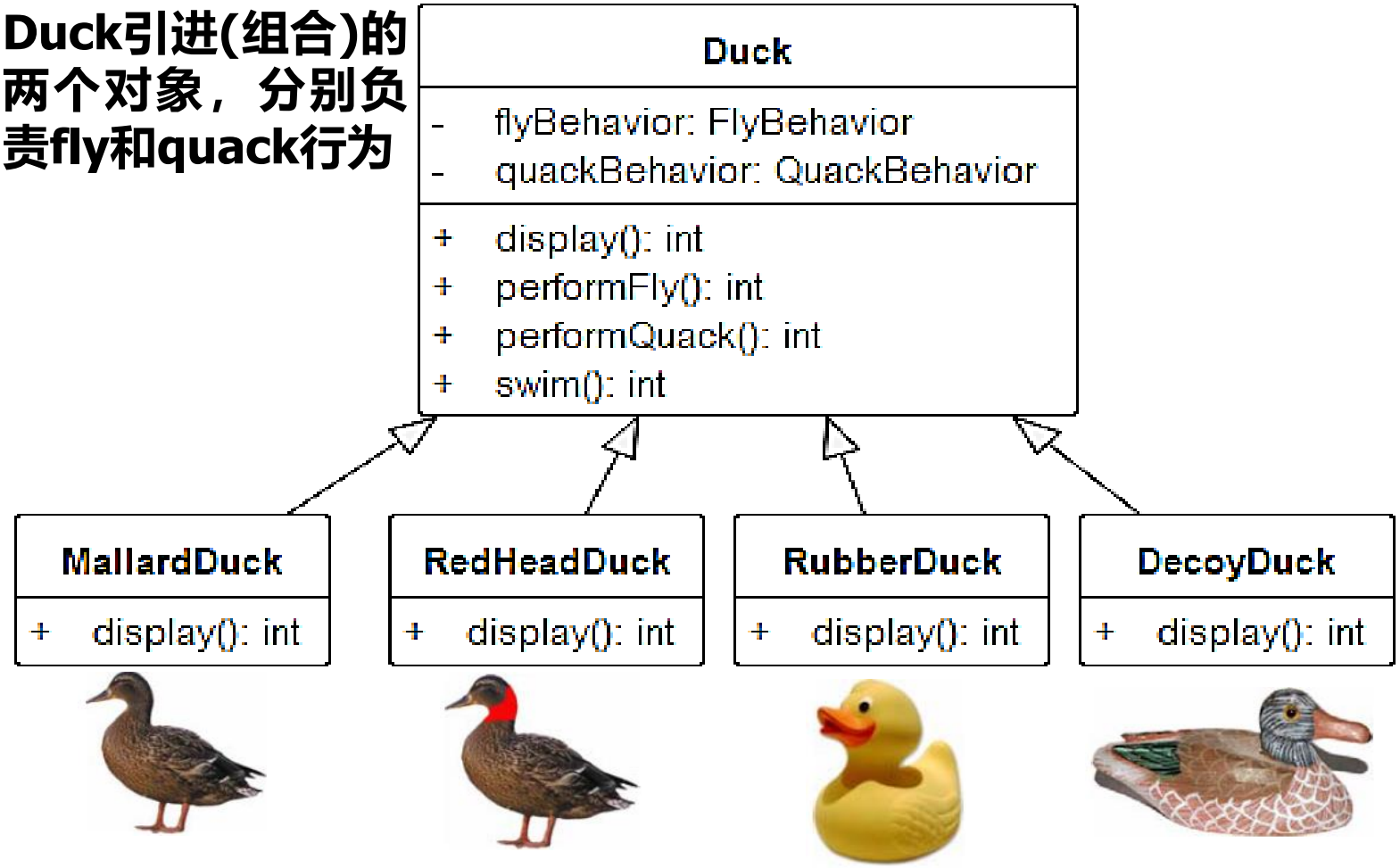
- 新的设计
  - Duck自己不再去fly( )、不再去quack( )
  - 增加了两个属性：行为对象
    - 当需要fly的时候，委托给flyBehavior对象去处理
    - 当需要quack的时候，委托给quackBehavior对象去处理
    - 当这些变量所引用的对象不同时，有不同的执行效果，以满足需求变化



- performFly( ) 代替了 fly( )
- performQuack( ) 代替了 quack （ ）

2.1 新的设计

Duck引进(组合)的两个对象，分别负责fly和quack行为




```
class Duck{
    public:
        FlyBehavior *flyBehavior;
        QuackBehavior *quackBehavior;
        ...
        void performFly();
        void performQuack();
        ...
};
```

```
void Duck::performFly(){
        flyBehavior->fly();
}
void Duck::performQuack(){
        quackBehavior->quack();
}
```

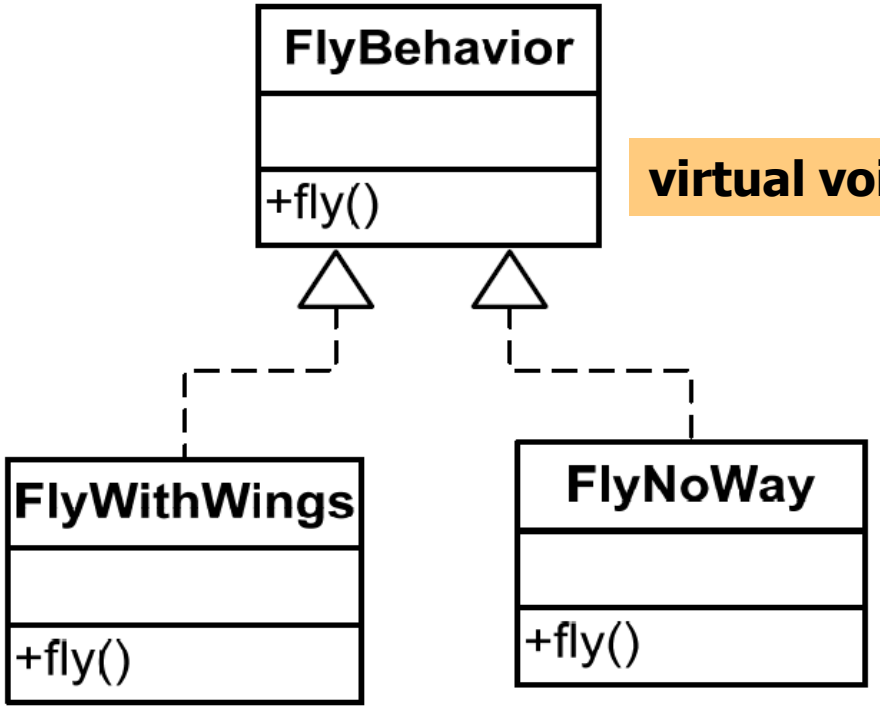
```
MallardDuck::MallardDuck(){
   flyBehavior = new FlyWithWings();
   quackBehavior = new Quack();
}
```

```
RubberDuck::RubberDuck(){
   flyBehavior = new FlyNoWay();
   quackBehavior = new Squick();
}
```

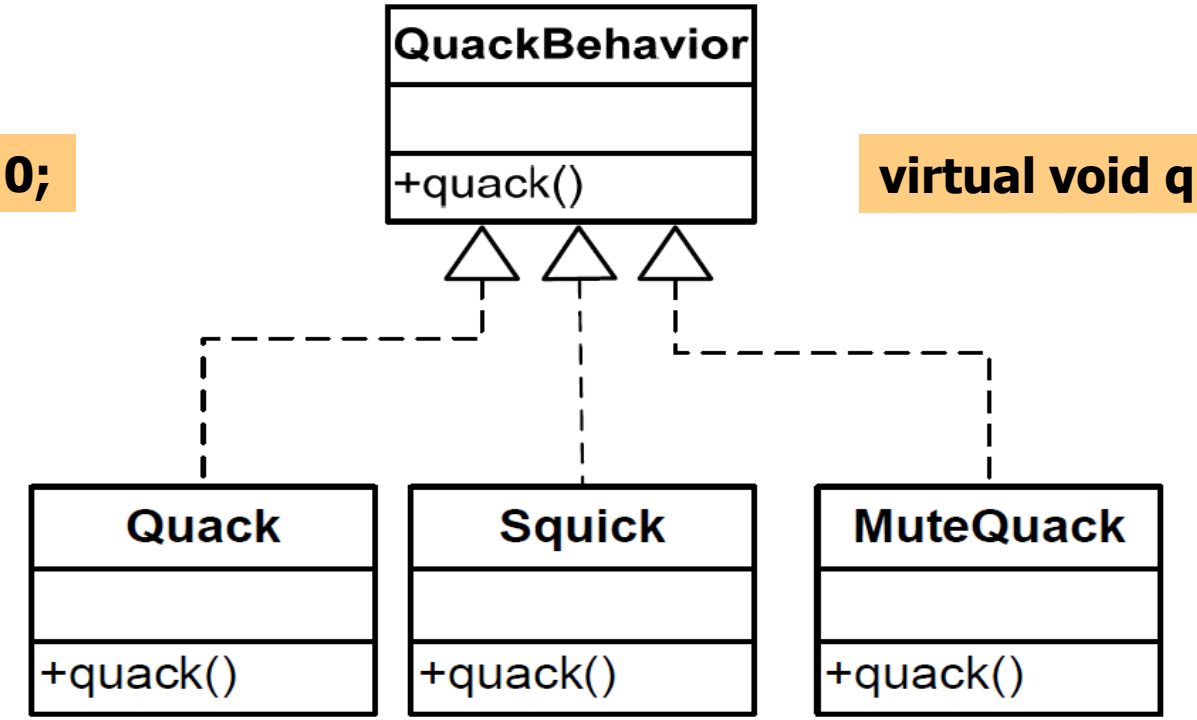
具体鸭子在初始化时，确定自己需要的fly和quack行为

回顾：FlyBehavior、QuckBehavior 的定义

- Fly() 和 quack() 行为在变化，所以，为每一种行为创建新类



virtual void fly()=0;



virtual void quack()=0;

```
void FlyWithWings::fly(){
    cout << "I'm flying!" << endl;
};
```

```
void FlyNoWay::fly(){
    cout << "I can't fly." << endl;
};
```

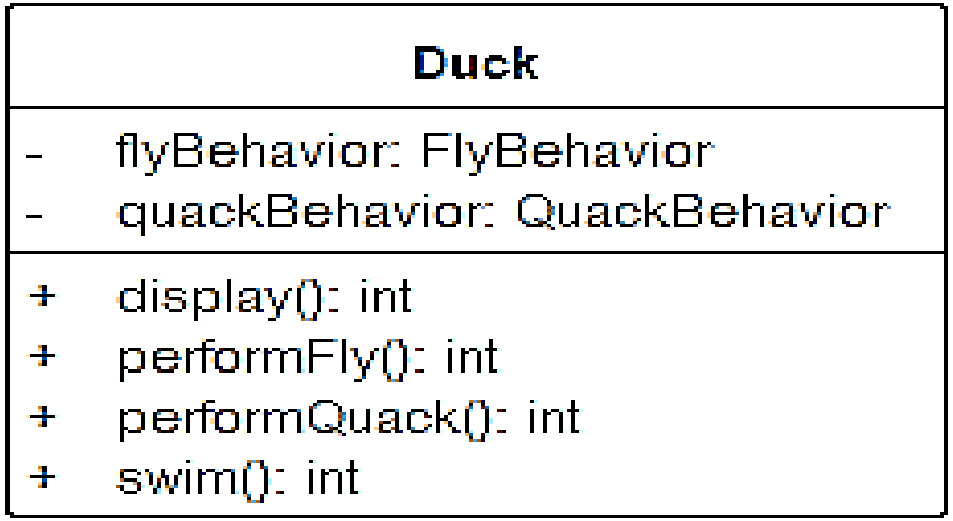
```
void Quack::quack(){
cout << "Quack" << endl;
};
```

```
void Squeak::quack(){
    cout << "Squeak" << endl;
};
```

```
void MuteQuack::quack(){
    cout << "....." << endl;
};
```

复习：面向对象设计原则

- 1、把变化的部分，封装起来
- 2、面向接口设计（编程），而不是面向实现



- 3、能用组合的地方，不要用继承

  Favor Composition over Inheritance

  - 每个 Duck 都有一个 flyBehavior、 quackBehavior 对象
  - 当 Duck 需要 fly 的时候，委托给 flyBehavior 对象去处理
  - 当 Duck 需要 quack 的时候，委托给 quackBehavior 对象去处理
  - Duck 没有使用继承机制，而是组合了具有所需行为的对象！

2.2 测试一下SimUDuck软件

```
int main(){
    cout << "Testing the Duck Simulator"
    << endl << endl;
    Duck *mallard = new MallardDuck();
    mallard->display();
    mallard->swim();
    mallard->performFly();
    mallard->performQuack();
    cout << endl;

    Duck *rubberduck = new RubberDuck();
    rubberduck->display();
    rubberduck->swim();
    rubberduck->performFly();
    rubberduck->performQuack();

    return 0;
}
```

```
MallardDuck::MallardDuck(){
   flyBehavior = new FlyWithWings();
   quackBehavior = new Quack();
}
```

```
void FlyWithWings::fly(){
   cout << "I'm flying!" << endl;
};
```

```
RubberDuck::RubberDuck(){
   flyBehavior = new FlyNoWay();
   quackBehavior = new Squick();
}
```

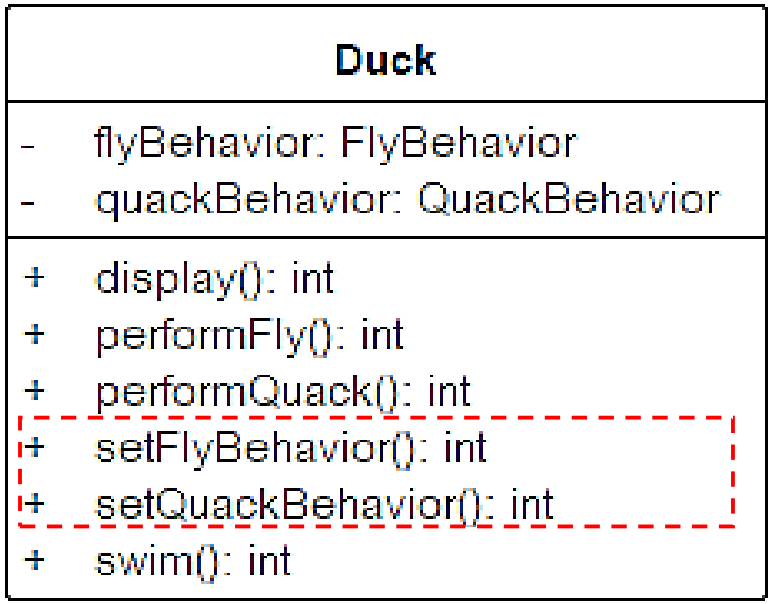
```
void FlyNoWay::fly(){
   cout << "I can't fly." << endl;
};
```

```
Testing the Duck Simulator

I'm a mallard duck
All ducks float, even decoys
I'm flying!!!
Quack

I'm a rubber duck
All ducks float, even decoys
I can't fly.
Squeak
Press any key to continue . . .
```

2.3 需求又变化了...

Joe, 我又在开董事会了。会上讨论到我们的竞争对手已经超过我们了，他们刚刚发布了一个新版本。你得做点什么！
能不能把哪些飞行中的鸭子打下来?

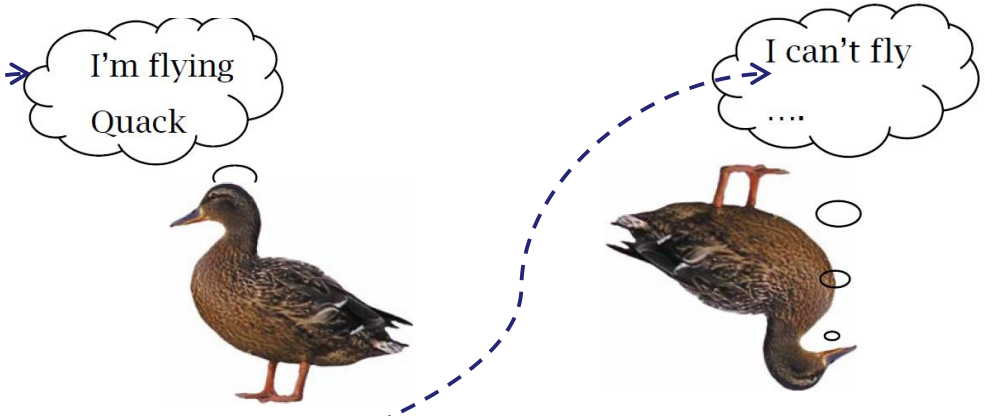
Joe：知道了，马上解决问题！我甚至可以把我们的鸭子仿真游戏改为“野鸭狩猎”游戏



2.4 新方案

```
void Duck::setFlyBehavior(FlyBehavior *fb){
        flyBehavior = fb;
}
void Duck::setQuackBehavior(QuackBehavior *qb){
        quackBehavior = qb;
}
```

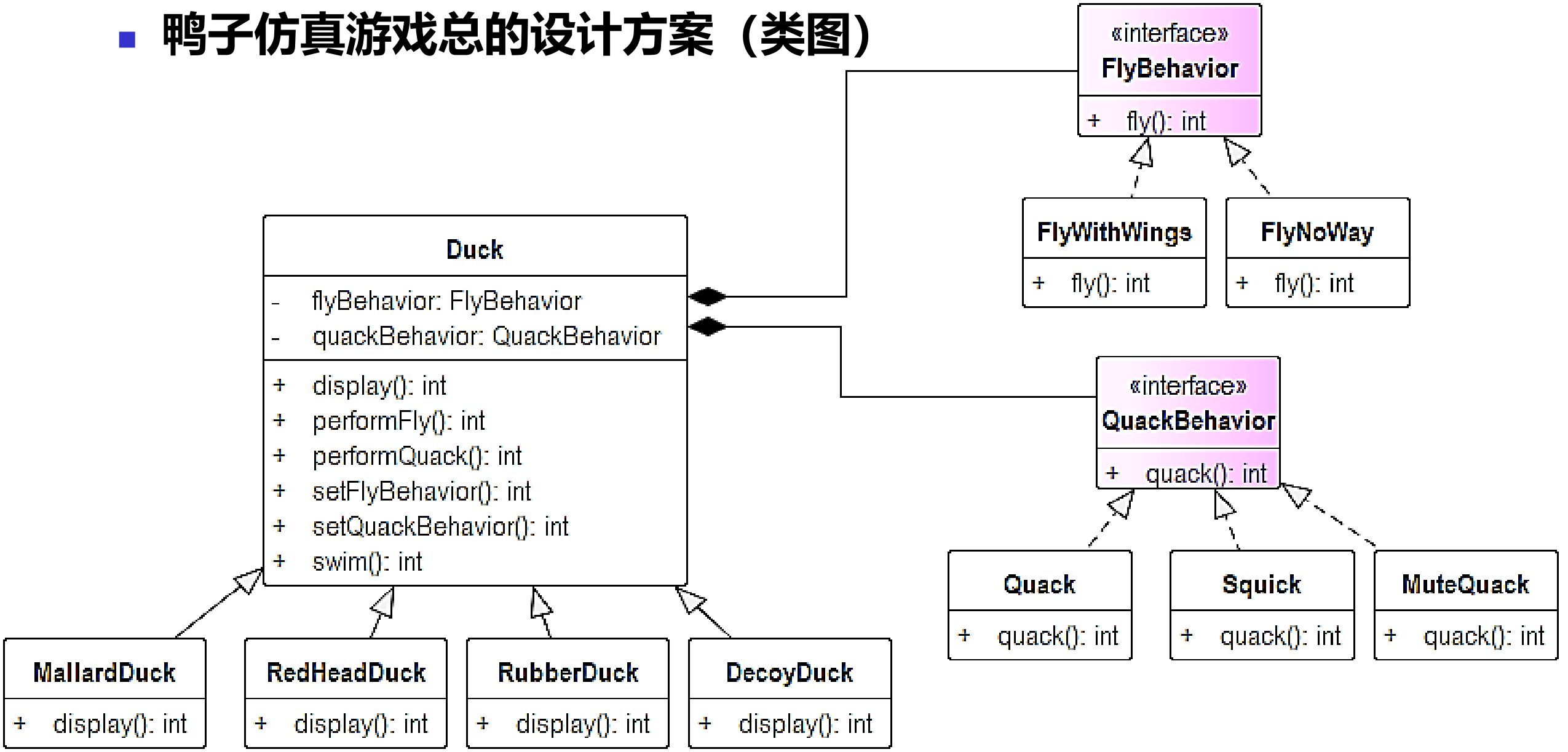
```
int main(){
  Duck *mallard = new MallardDuck();
  mallard->display();
  mallard->swim();
  mallard->performFly();
  mallard->performQuack();
  cout << endl;
   //鸭子还是这只鸭子。但它把fly、quack行为对象换了
  mallard->setFlyBehavior(new FlyNoWay());
  mallard->setQuackBehavior(new MuteQuack());
  mallard->performFly();
  mallard->performQuack();

return 0;
}
```




```
Testing the Duck Simulator
I'm a mallard duck
All ducks float, even decoys
I'm flying?!?
Quack

I can't fly.
.....
Press any key to continue . . .
```

2.4 新方案

- 鸭子仿真游戏总的设计方案（类图）



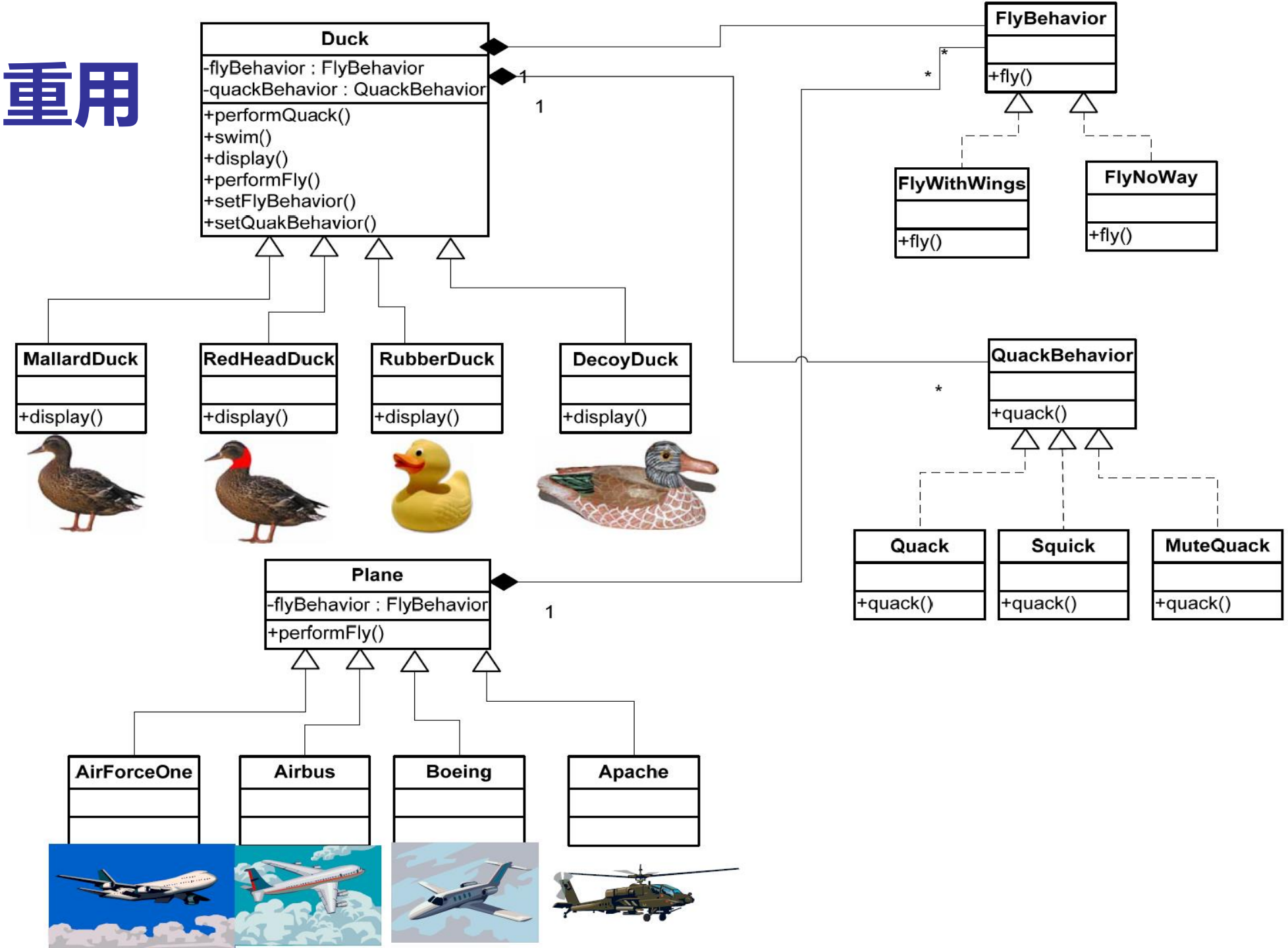
2.5 需求再次变化了…

Joe, 我在电影院看电影 “新球大战”。电影太精彩了！我在想，能不能把一些漂亮的飞机加到我们的游戏中来，彻底打败我们的竞争对手?

Joe：没问题，马上解决问题!

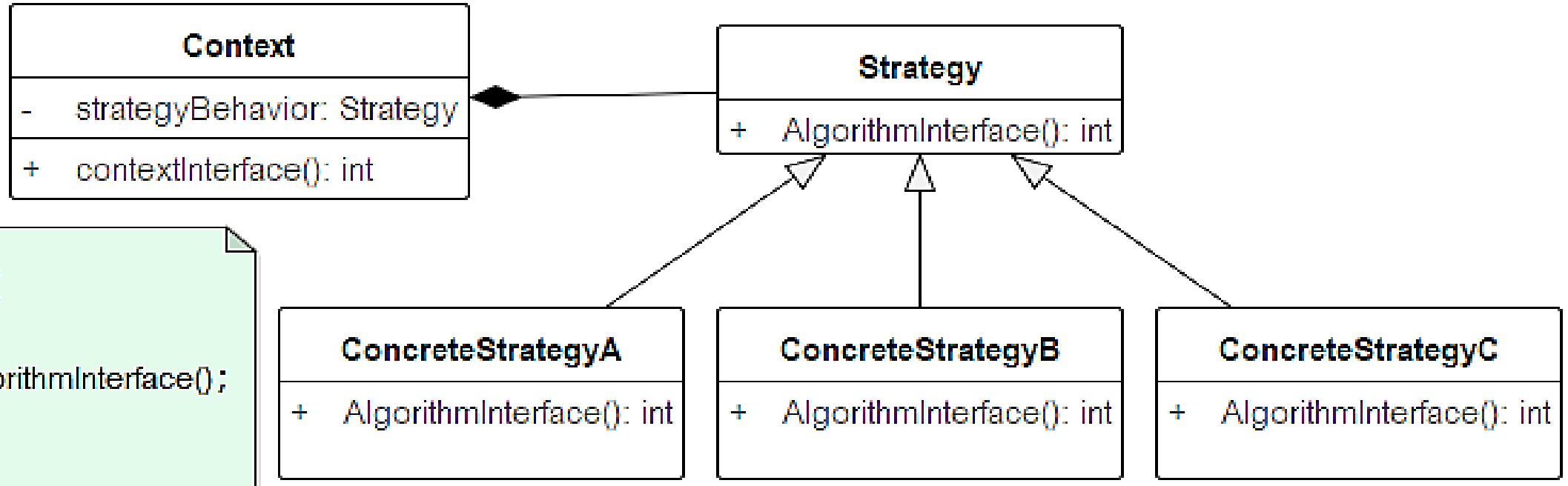
嗯，嗯，另外，… 是不是要给我加点薪水啦….

2.6 行为重用


Duck
-flyBehavior : FlyBehavior
-quackBehavior : QuackBehavior
+performQuack()
+swim()
+display()
+performFly()
+setFlyBehavior()
+setQuakBehavior()
FlyBehavior
+fly()
FlyWithWings
+fly()
FlyNoWay
+fly()
MallardDuck
+display()
RedHeadDuck
+display()
RubberDuck
+display()
DecoyDuck
+display()
QuackBehavior
+quack()
Quack
+quack()
Squick
+quack()
MuteQuack
+quack()
Plane
-flyBehavior : FlyBehavior
+performFly()
AirForceOne
Airbus
Boeing
Apache
1
1
*
*
1
*

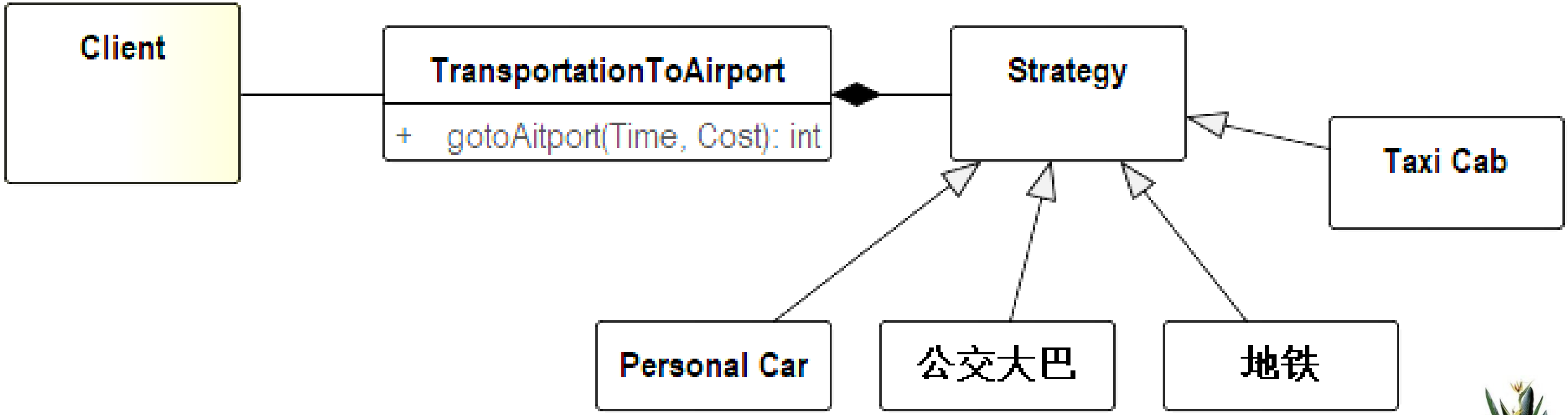
2.7 这就是策略模式

- 策略模式
  - 定义了算法族，分别封装起来，让它们之间可以互相替换，此模式让算法的变化独立于使用算法的客户
  - Strategy – defines a family of algorithms, encapsulate each one, and makes them interchangeable. Strategy lets the algorithm vary independently from the clients that use it




```
contextInterface () {

strategyBehavior.AlgorithmInterface();
}
```

2.8 例：生活中的策略模式



- 本讲结束