Design Patterns

© 2014 Luster

Me

鐘友志

達暉資訊有限公司

軟體工程師

jason@lustertech.net

Reference

啓蒙之書

Object Oriented Basic

抽象 Abstraction

封裝 Encapsulation

繼承 Inheritance

多型 Polymorphism

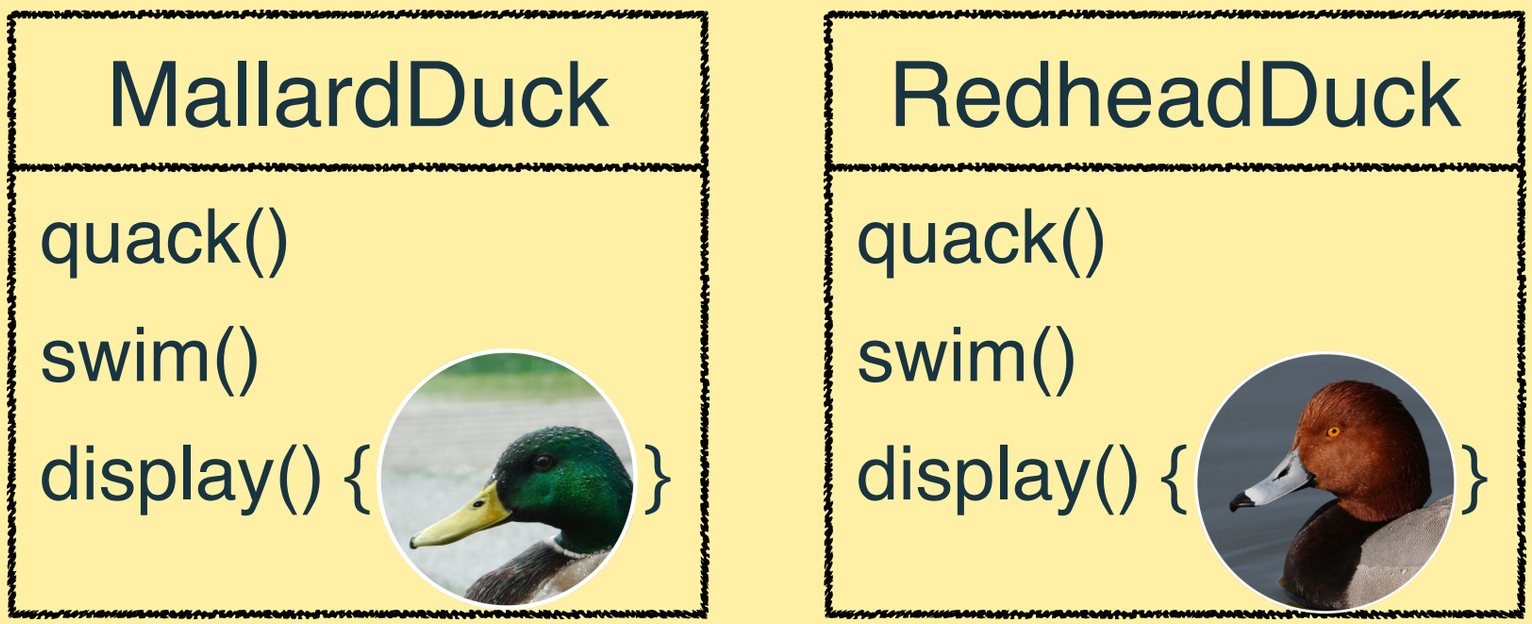
用情境，學設計模式

很久很久以前...

達暉超寫實遊戲公司

模擬鴨子

Class Diagram



Class Diagram

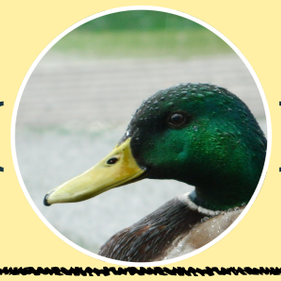
Duck

MallardDuck

quack()

swim()

display() {



}

RedheadDuck

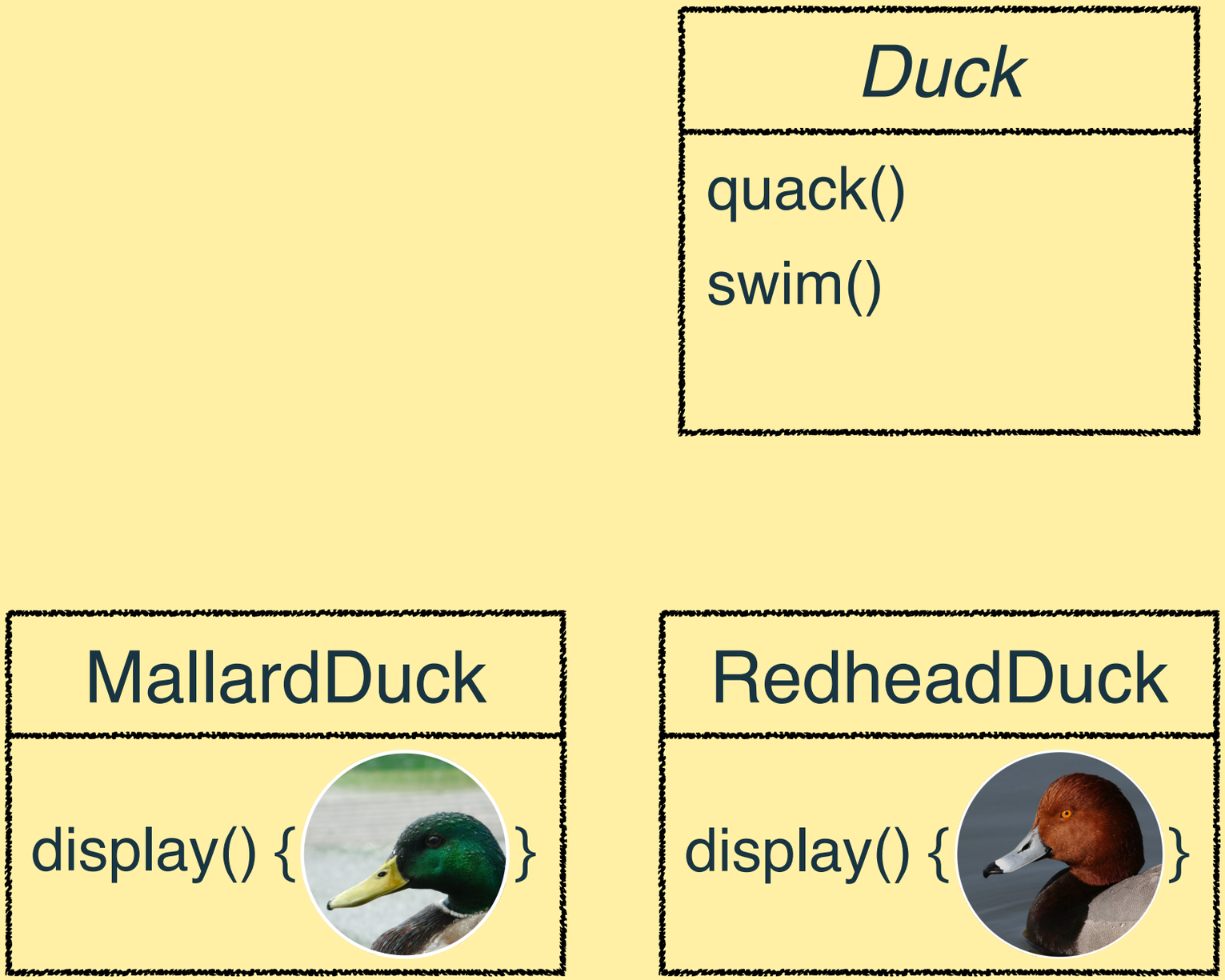
quack()

swim()

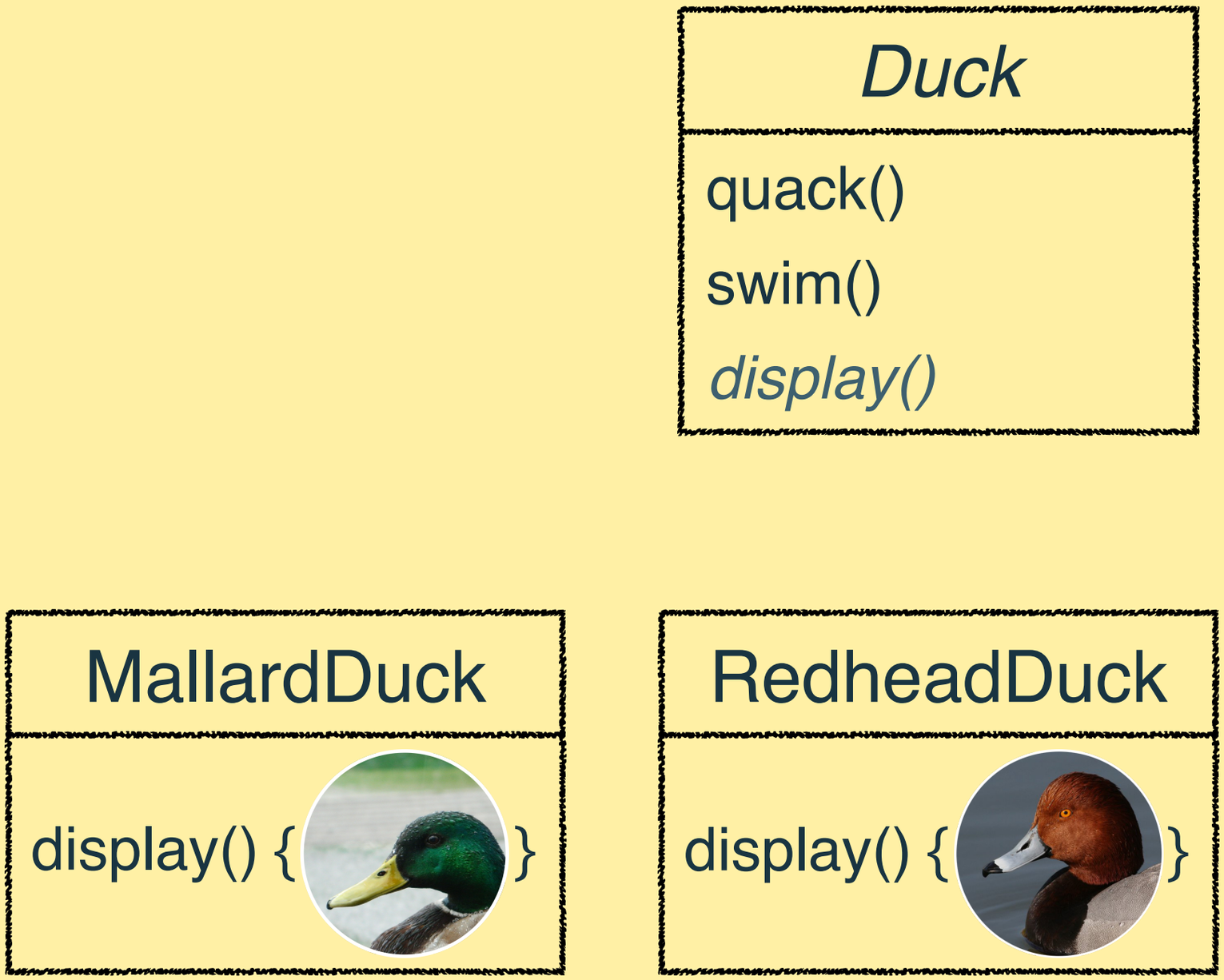
display() {

}

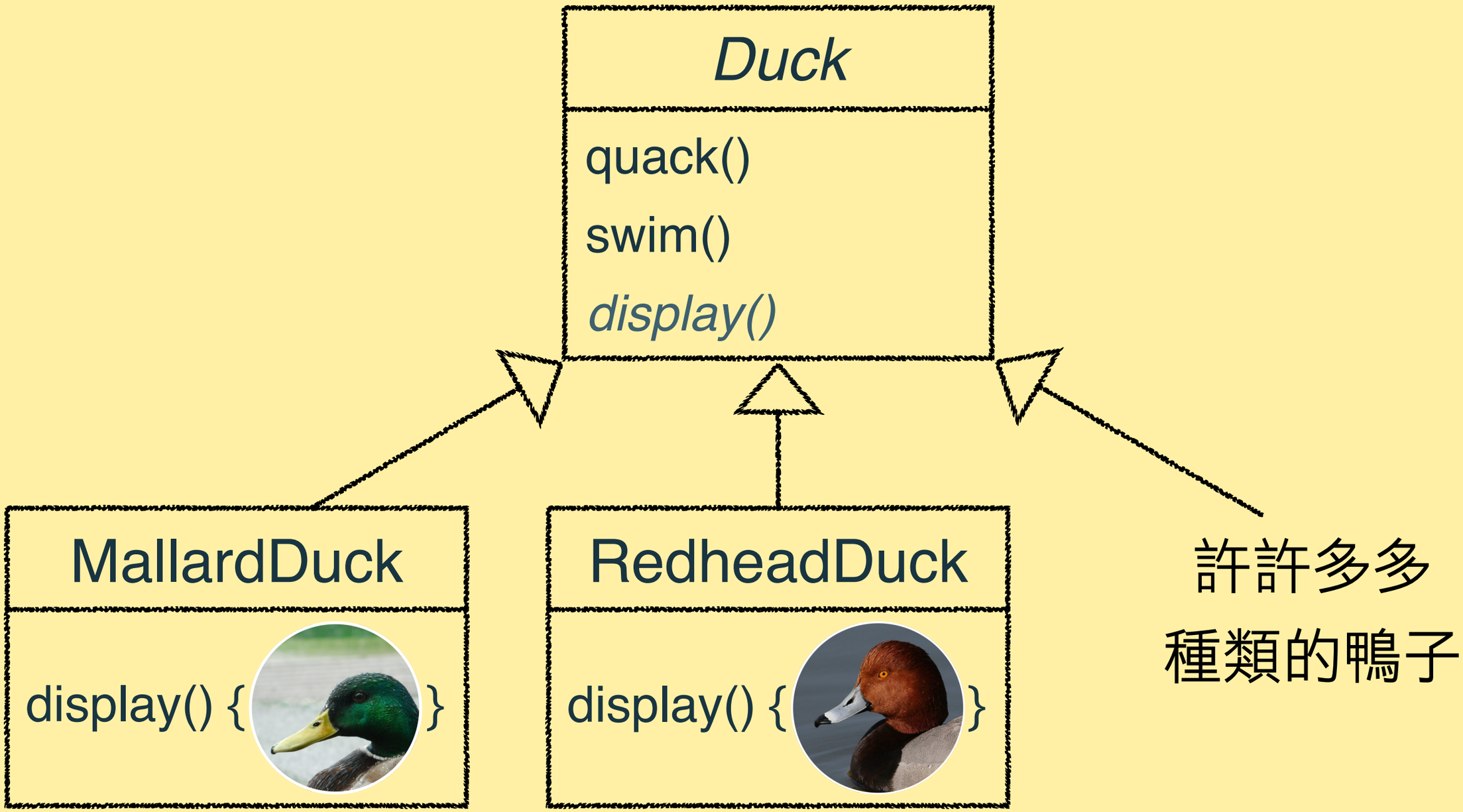
Class Diagram



Class Diagram



Class Diagram



由於市場競爭激烈，
我們決定新增需求。

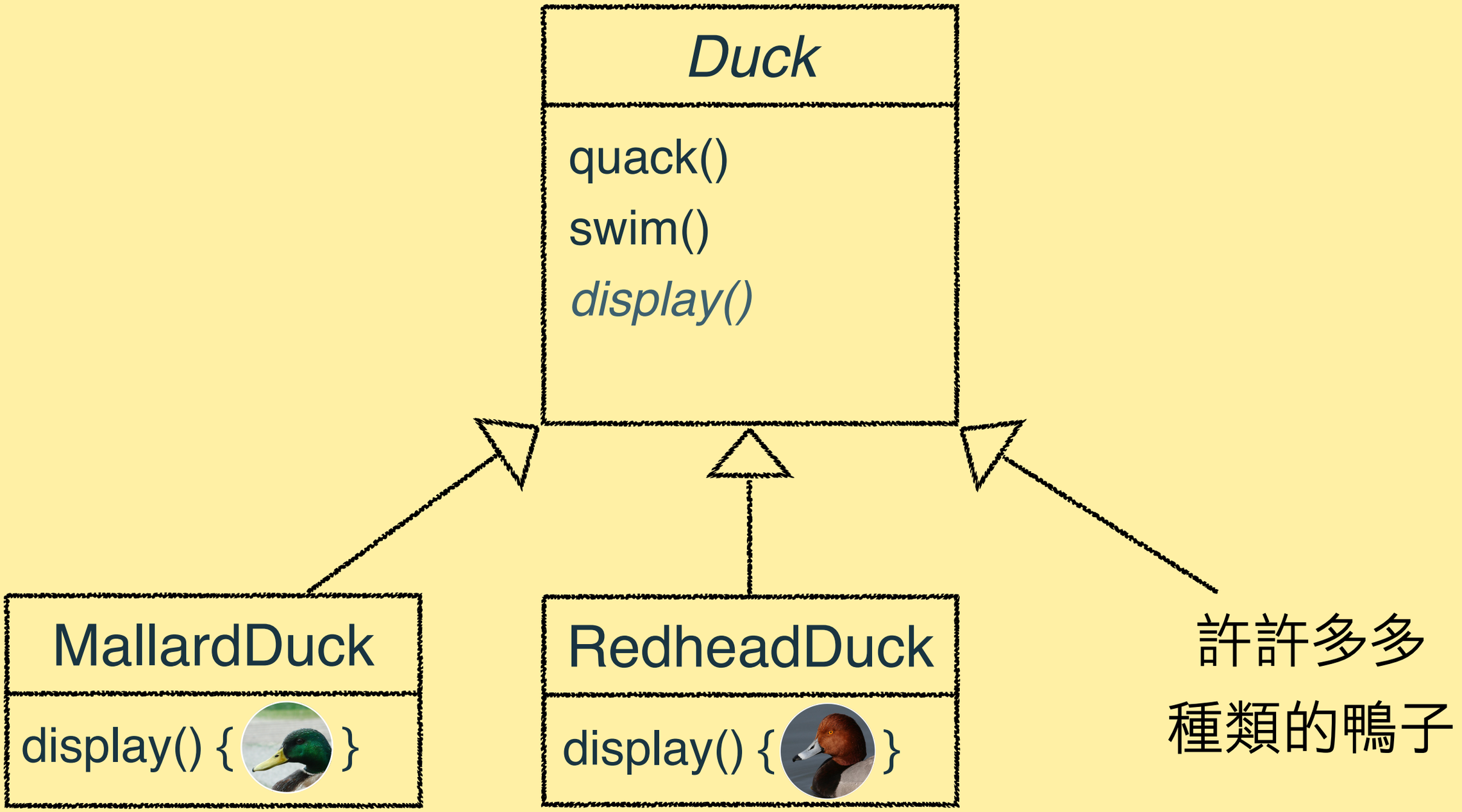
讓鴨子飛

模擬鴨子，全新改版

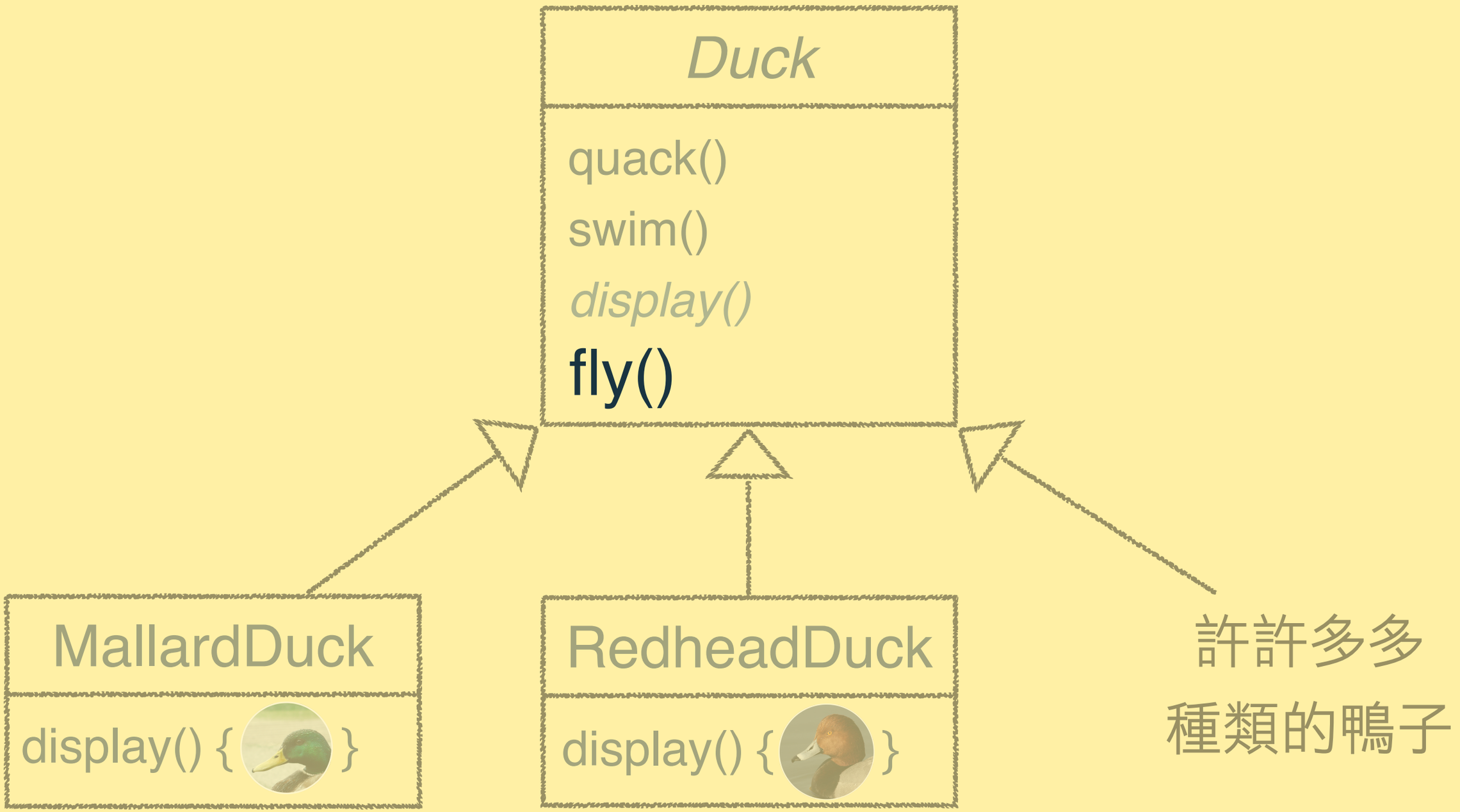
“我只需要
在父類別 Duck 中加上 fly() 方法
所有繼承的鴨子就會飛了
蠻簡單的！”

達暉超寫實遊戲有限公司
菜鳥工程師 Jason

原 Class Diagram



Jason's Class Diagram



公測那一天

黃色塑膠鴨竟然也會飛了－ －III


IT'S NOT A BUG
IT'S A FEATURE!

What's happened ?

Duck

quack()

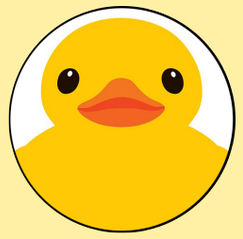
swim()

display()

fly()

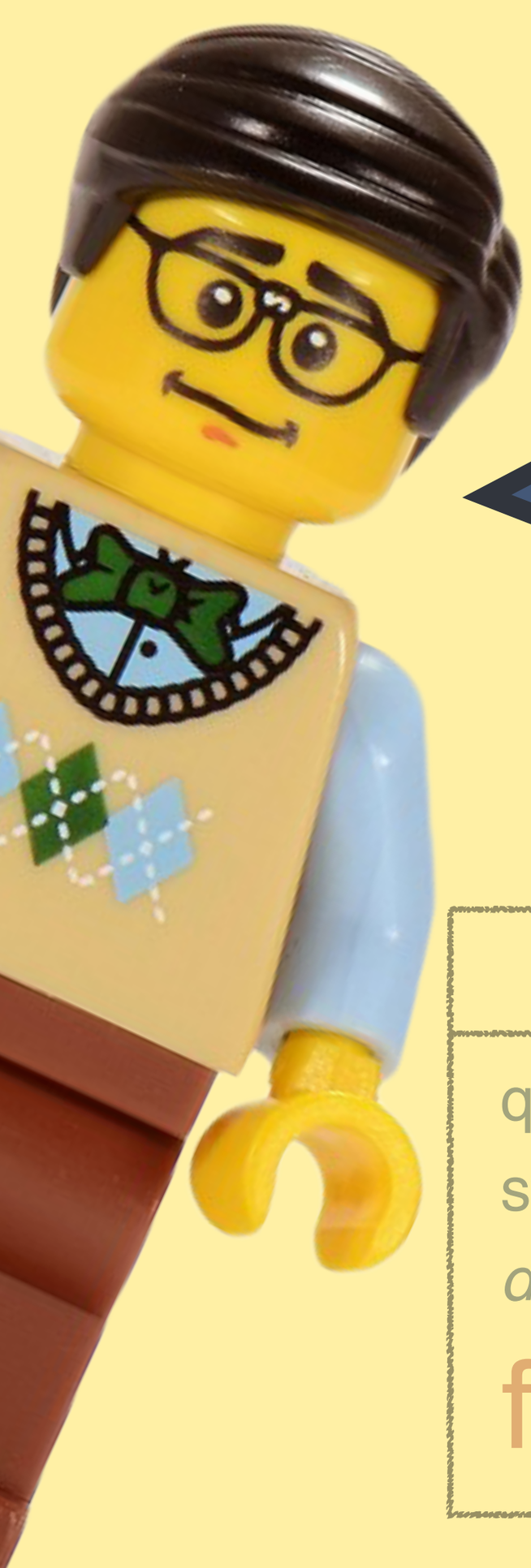
RubberDuck

@Override
quack() { 吱吱叫; }

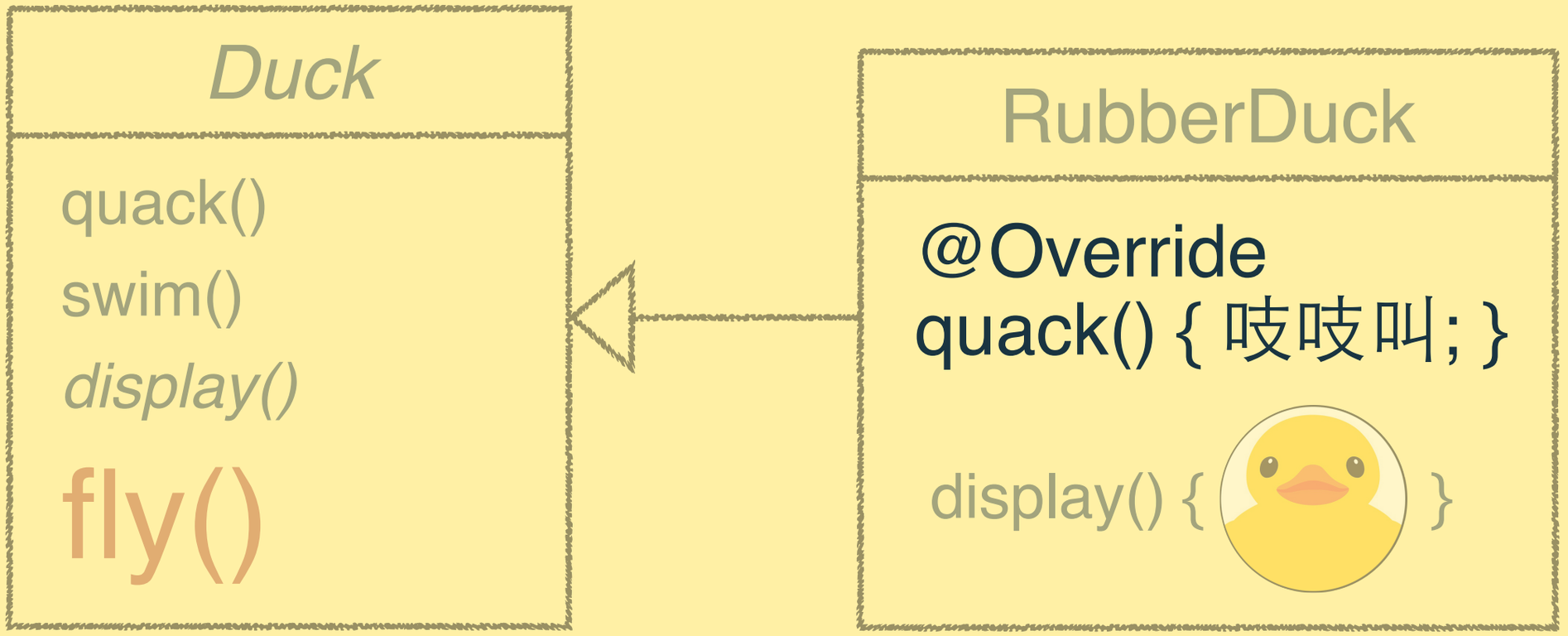
display() {  }

繼承，

一把刀兩面刃。



“事已如此，那我就
把 RubberDuck 的 fly() 推翻掉
就像推翻 quack() 一樣！”



Jason's Class Diagram II

Duck

quack()

swim()

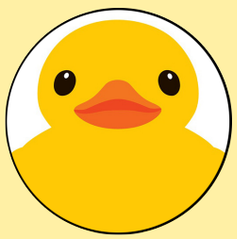
display()

fly()

RubberDuck

@Override

quack() { 吱吱叫; }

display() {  }

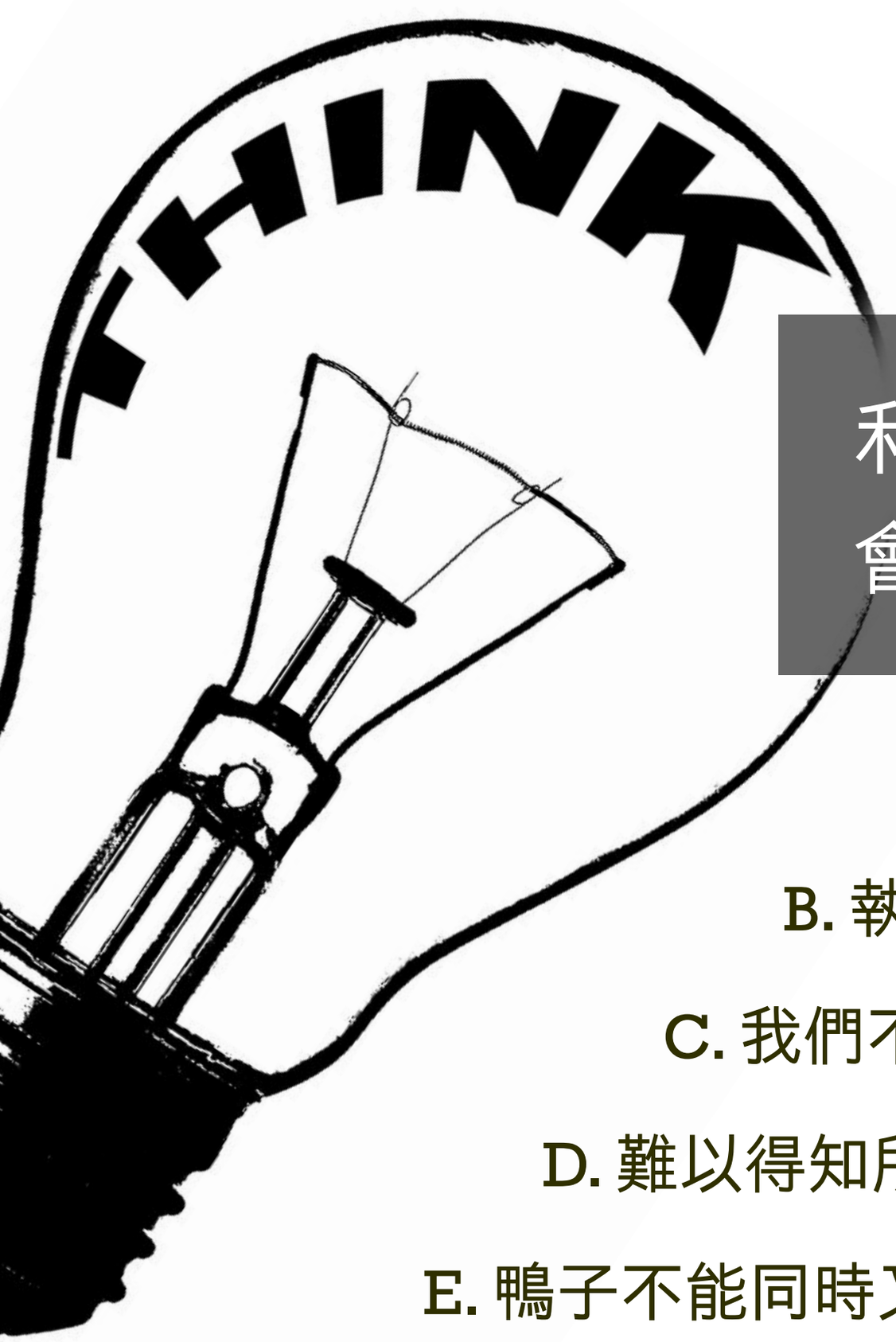
@Override

fly() { // Do nothing! }

“如果以後又加入神魔火鴨，牠不會叫，不會飛，不會游泳，那又會如何呢？”

FireDuck

```
@Override
quack() { // Do nothing! }

@Override
swim() { // Do nothing! }

display() {  }

@Override
fly() { // Do nothing! }
```



利用繼承提供鴨子行為，
會導致下列哪些缺點？（多選）

A. 程式碼在多個次類別中重複

B. 執行期的行為不易改變

C. 我們不能讓鴨子跳舞

D. 難以得知所有鴨子的全部行為

E. 鴨子不能同時又飛又叫

F. 改變會牽一髮動全身，造成其他鴨子不想要的改變

“如果以後加入更多鴨子”

“或者更多的行為”

“每種鴨的行為又不一致”

而且又繼續用繼承的方式，那我豈不是要

而且又繼續用繼承的方式，那我豈不是要

逐鴨審查

“為了避免
新增一個子類別，就須覆寫不需要的行為或
新增一個新行為，就須重新審視子類別，
試試利用介面如何？”

“為了避免
新增一個子類別，就須覆寫不需要的行為或
新增一個新行為，就須重新審視子類別，
試試利用介面如何？”

Flyable
fly()

Quackable
quack()

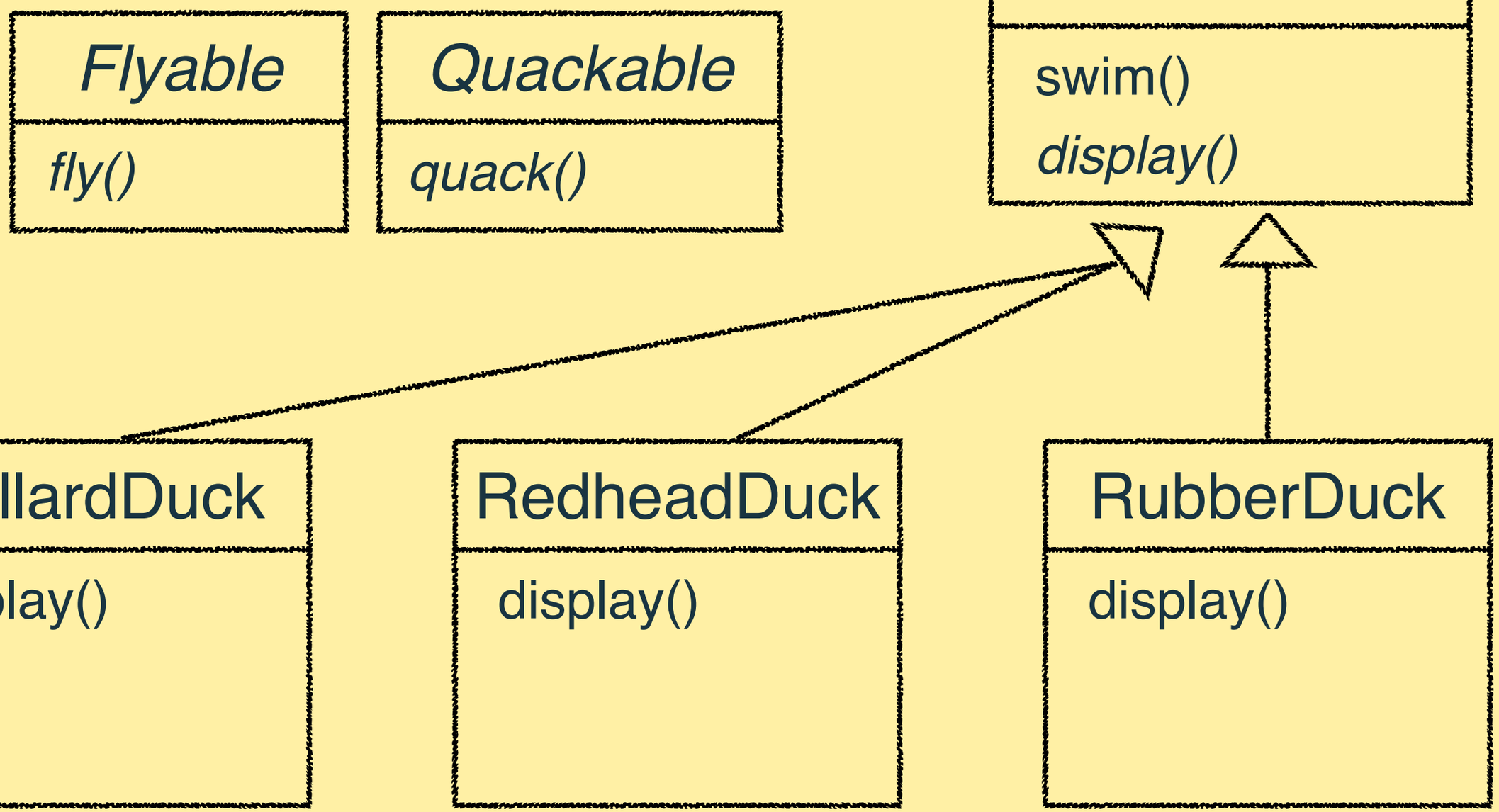
Duck
swim() *display()*

MallardDuck

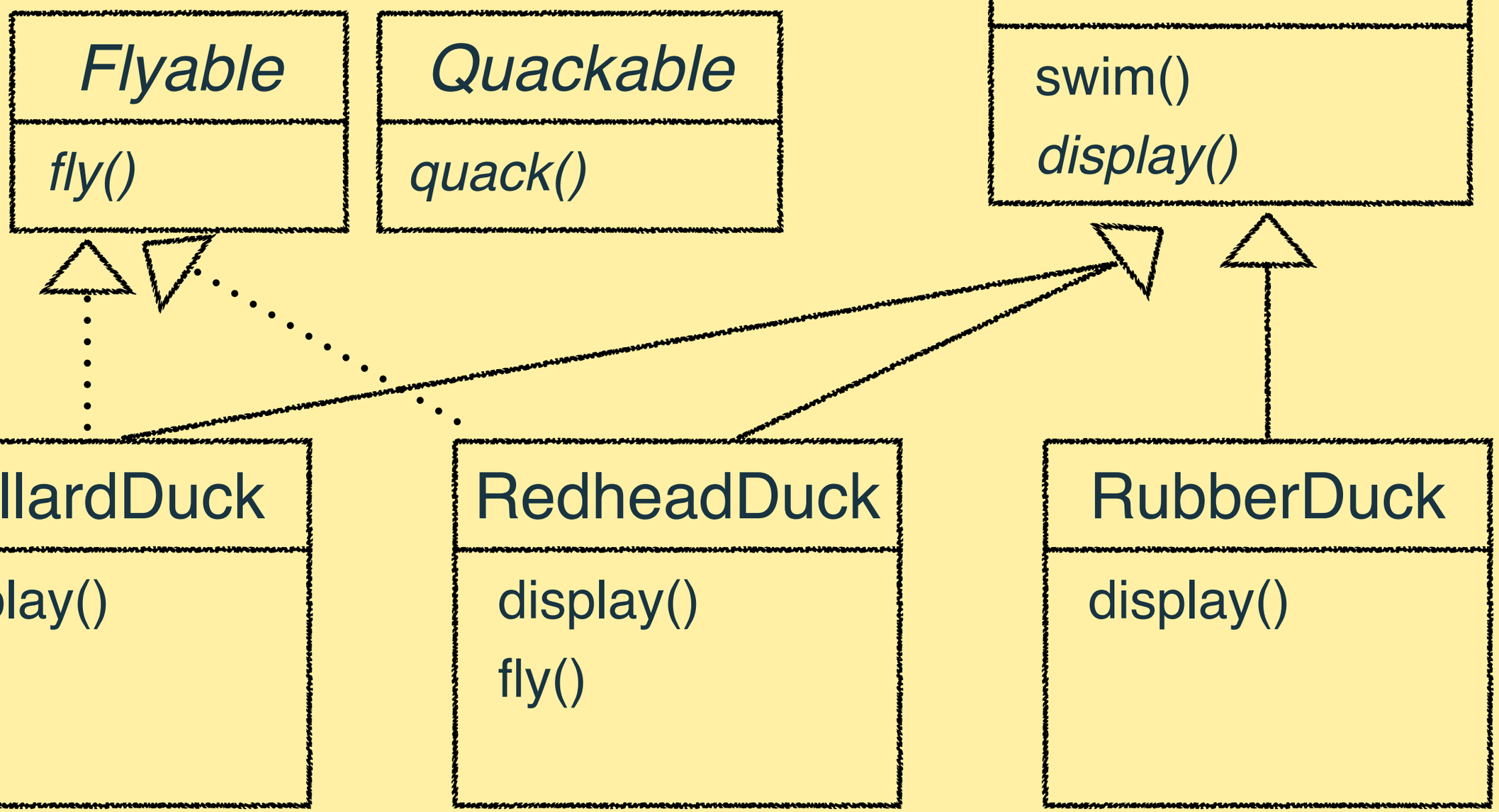
RedheadDuck

RubberDuck

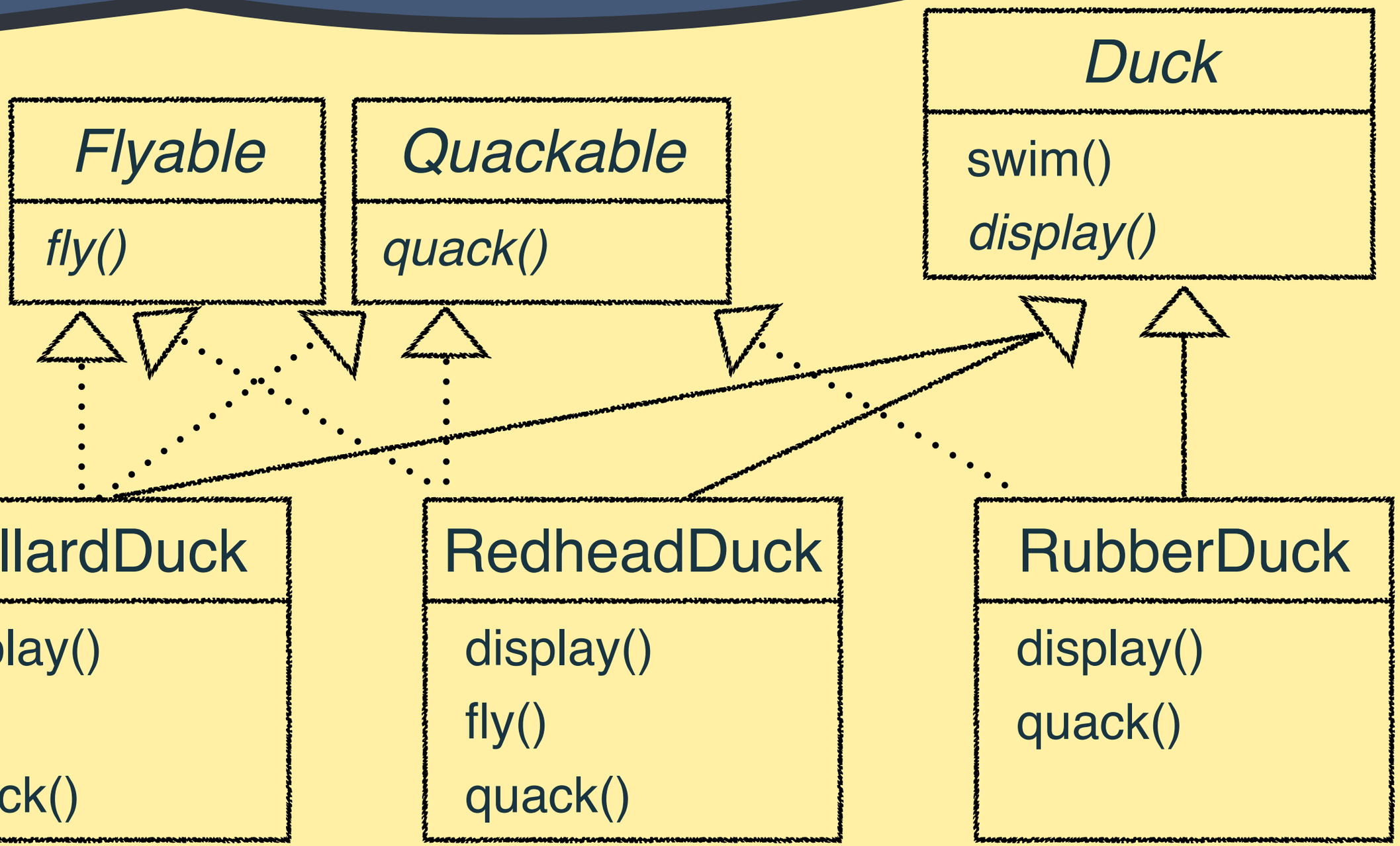
“為了避免
新增一個子類別，就須覆寫不需要的行為或
新增一個新行為，就須重新審視子類別，
試試利用介面如何？”



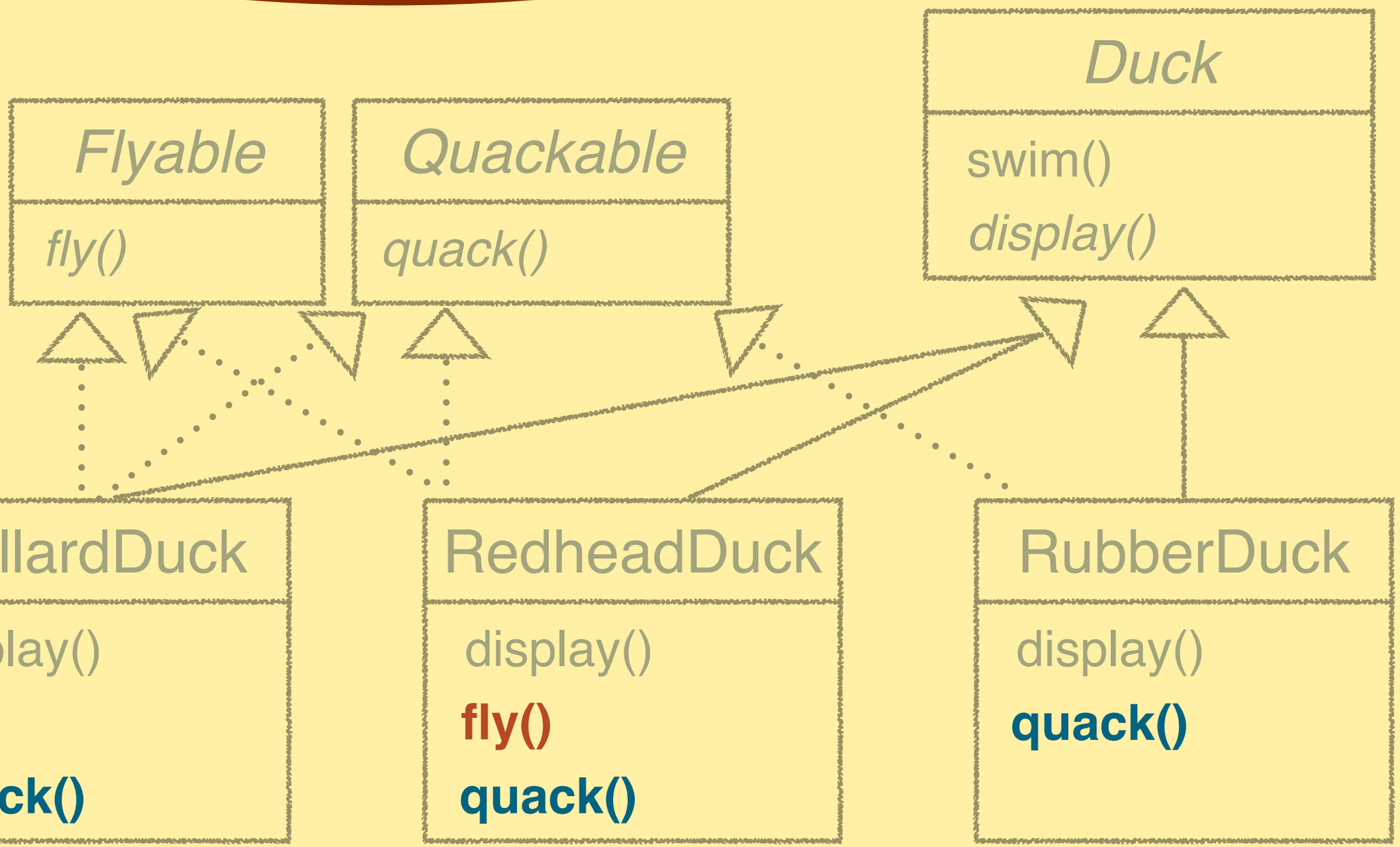
“為了避免
新增一個子類別，就須覆寫不需要的行為或
新增一個新行為，就須重新審視子類別，
試試利用介面如何？”

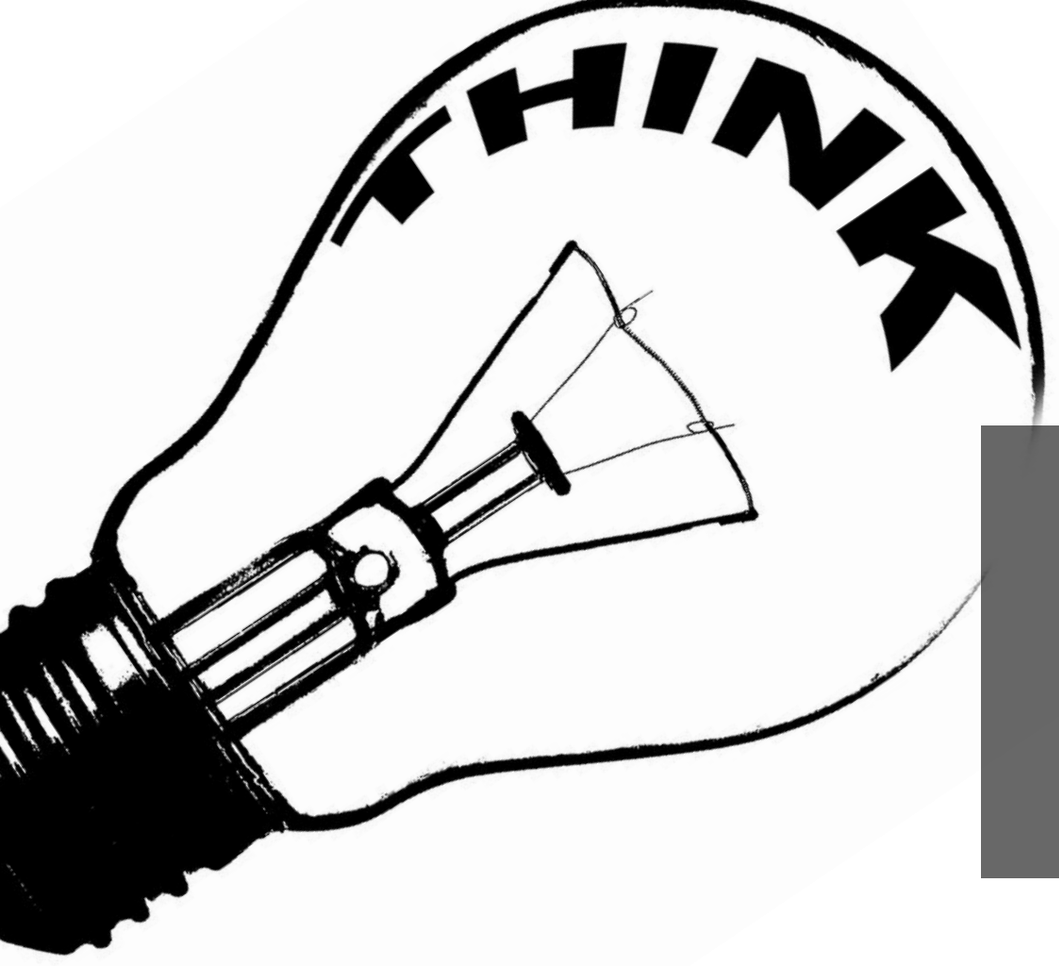


“為了避免
新增一個子類別，就須覆寫不需要的行為或
新增一個新行為，就須重新審視子類別，
試試利用介面如何？”

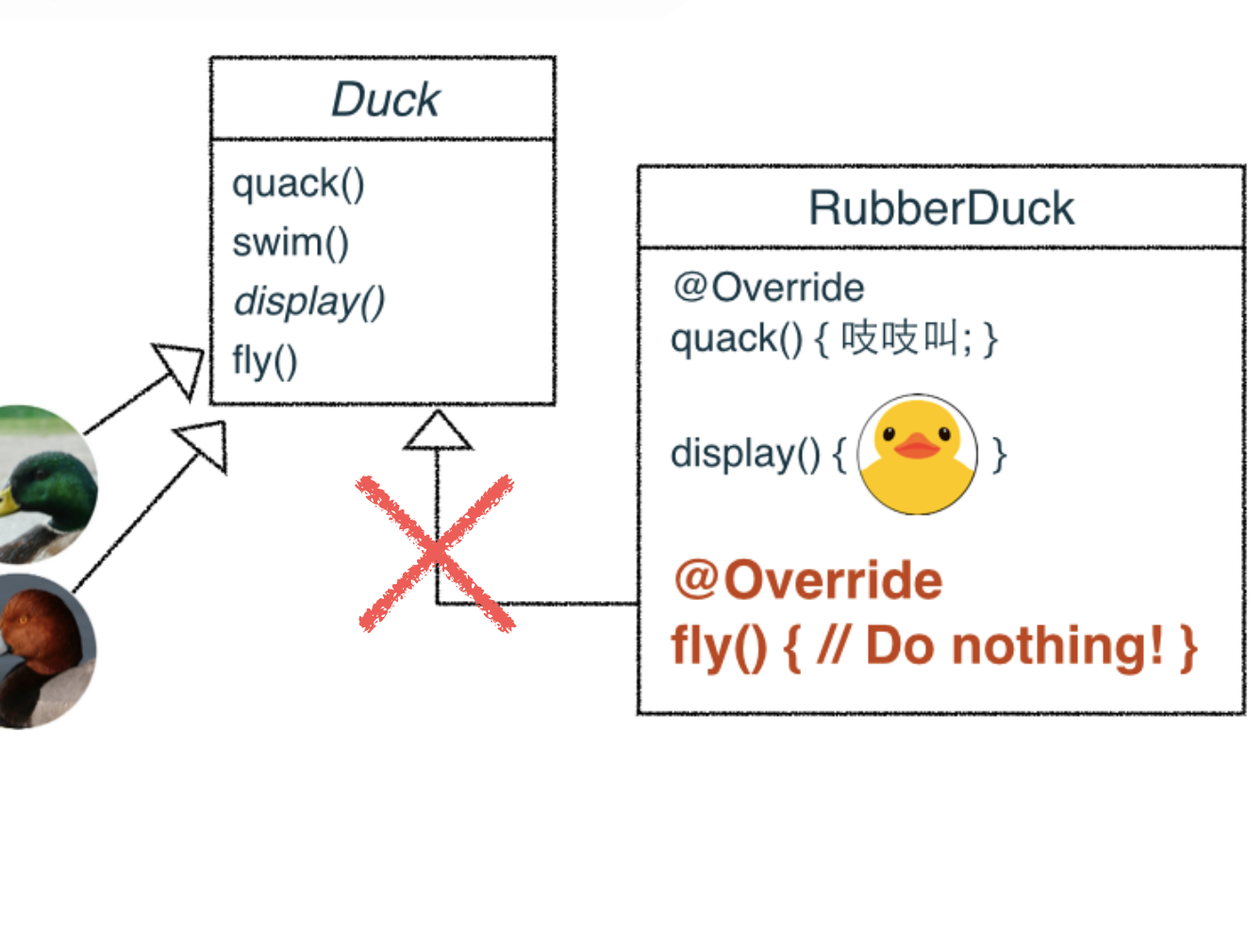


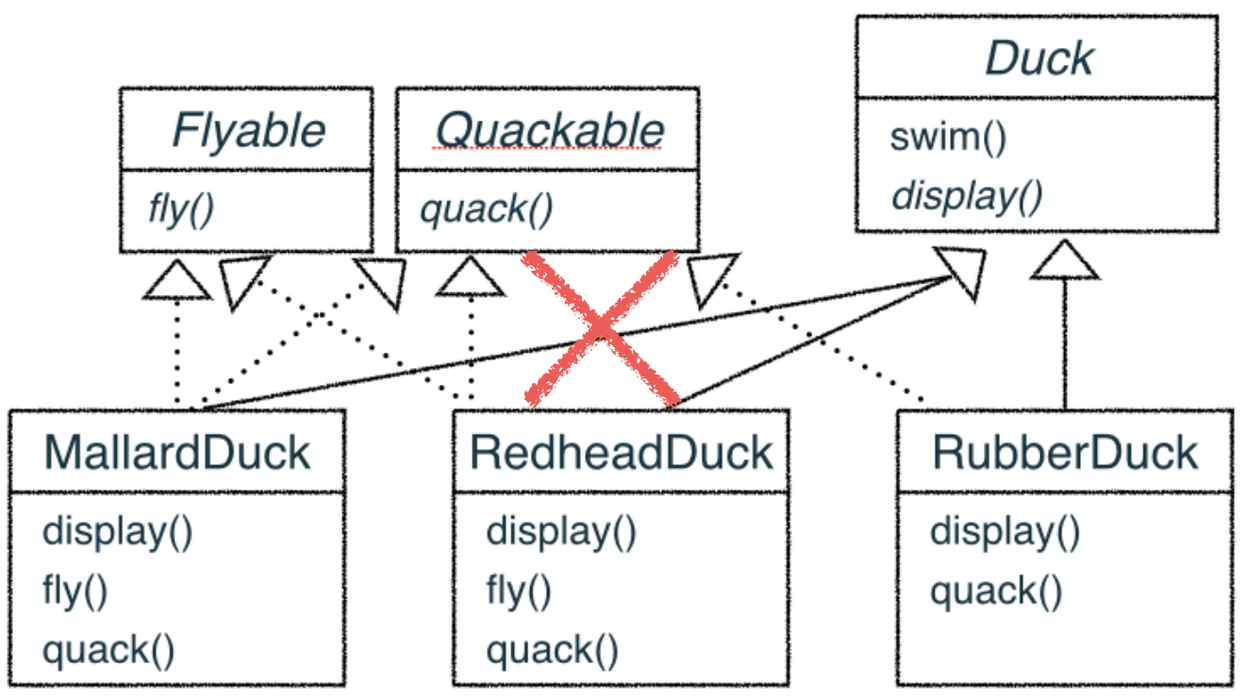
“失敗，程式碼重複了”


Flyable
fly()
Quackable
quack()
Duck
swim()
display()
MallardDuck
display()
fly()
quack()
RedheadDuck
display()
fly()
quack()
RubberDuck
display()
quack()



如果你是 Jason，你要怎麼做呢？





設計守則
Design Principle

1

將程式中

會變化的部分

取出並封裝

...

呱

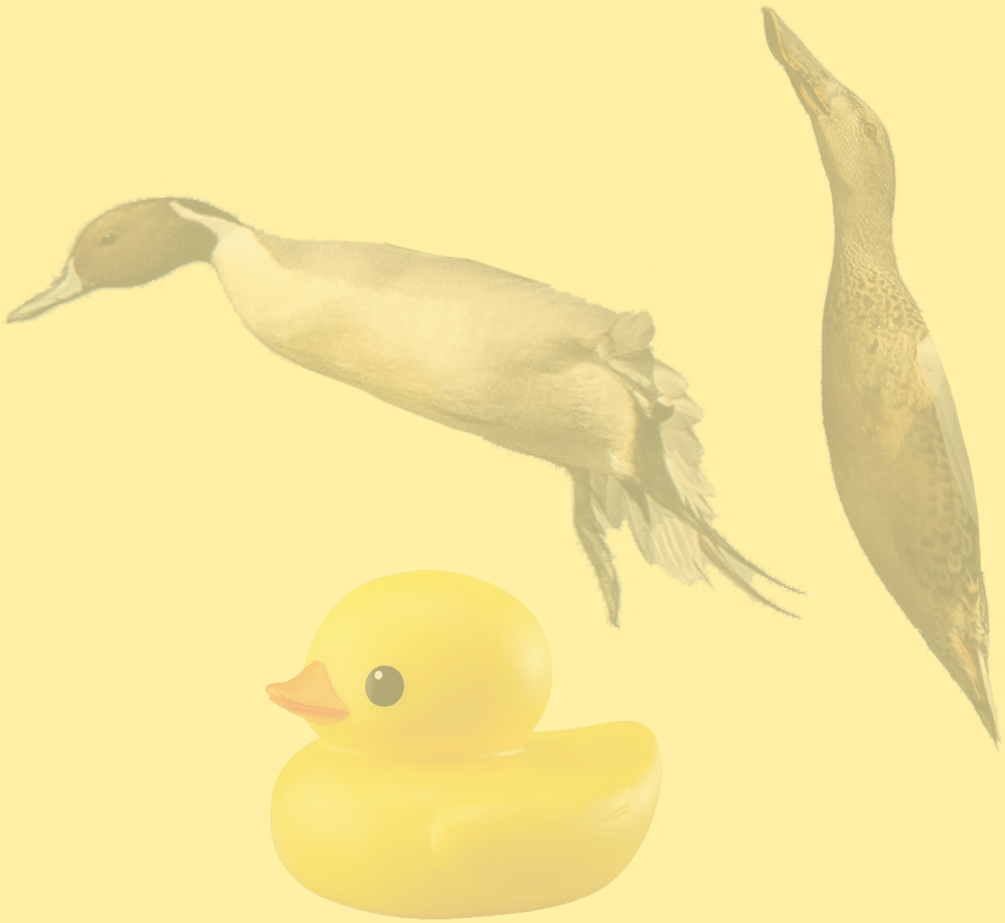
吱

不會飛

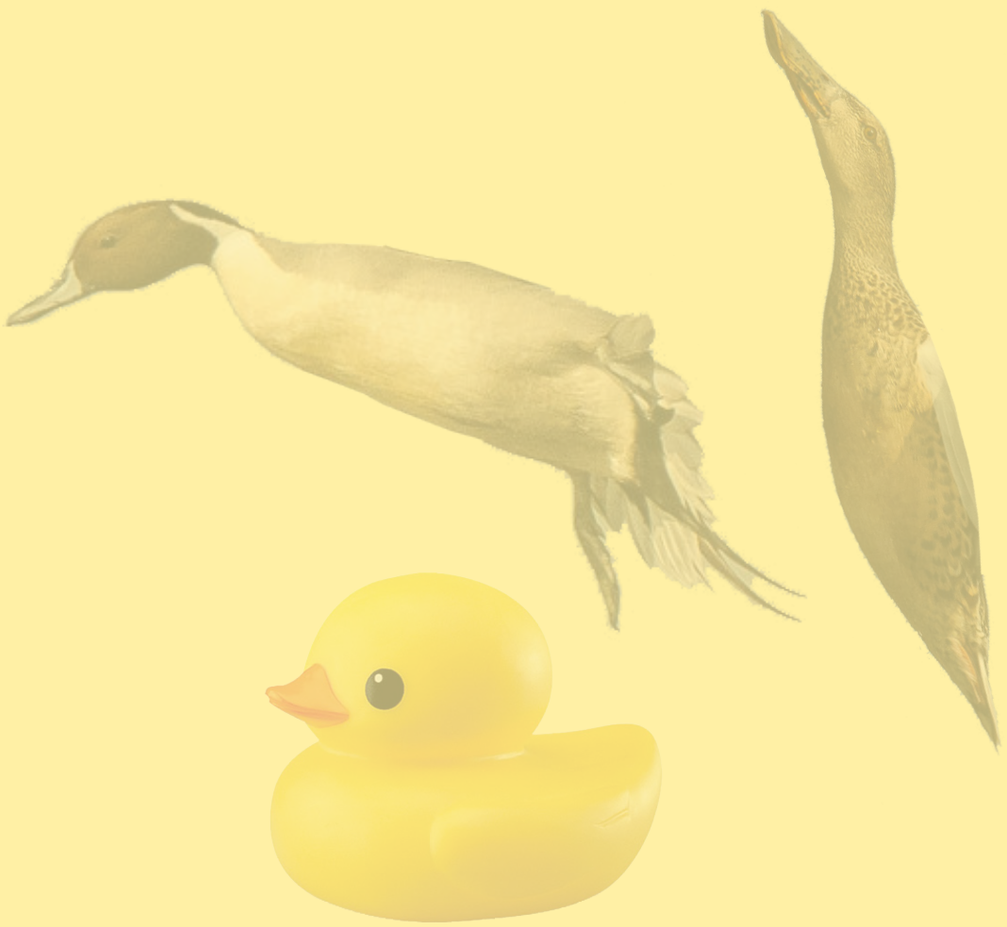
將變化取出，並封裝

將變化取出，並封裝

將變化取出，並封裝



將變化取出，並封裝

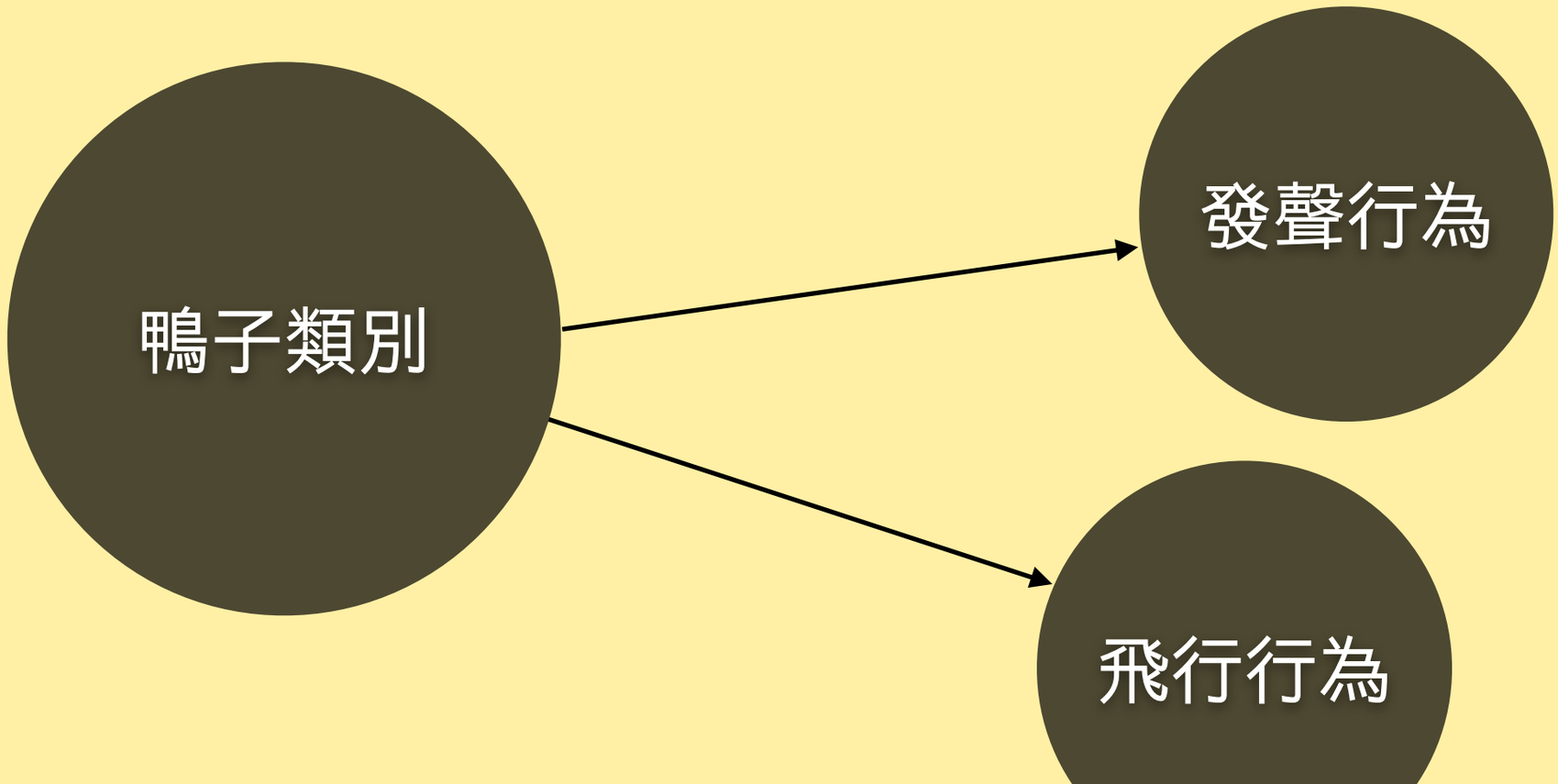


將變化取出，並封裝

1

將變化取出

並封裝



2

寫程式

是針對介面而寫

不是針對實作而寫

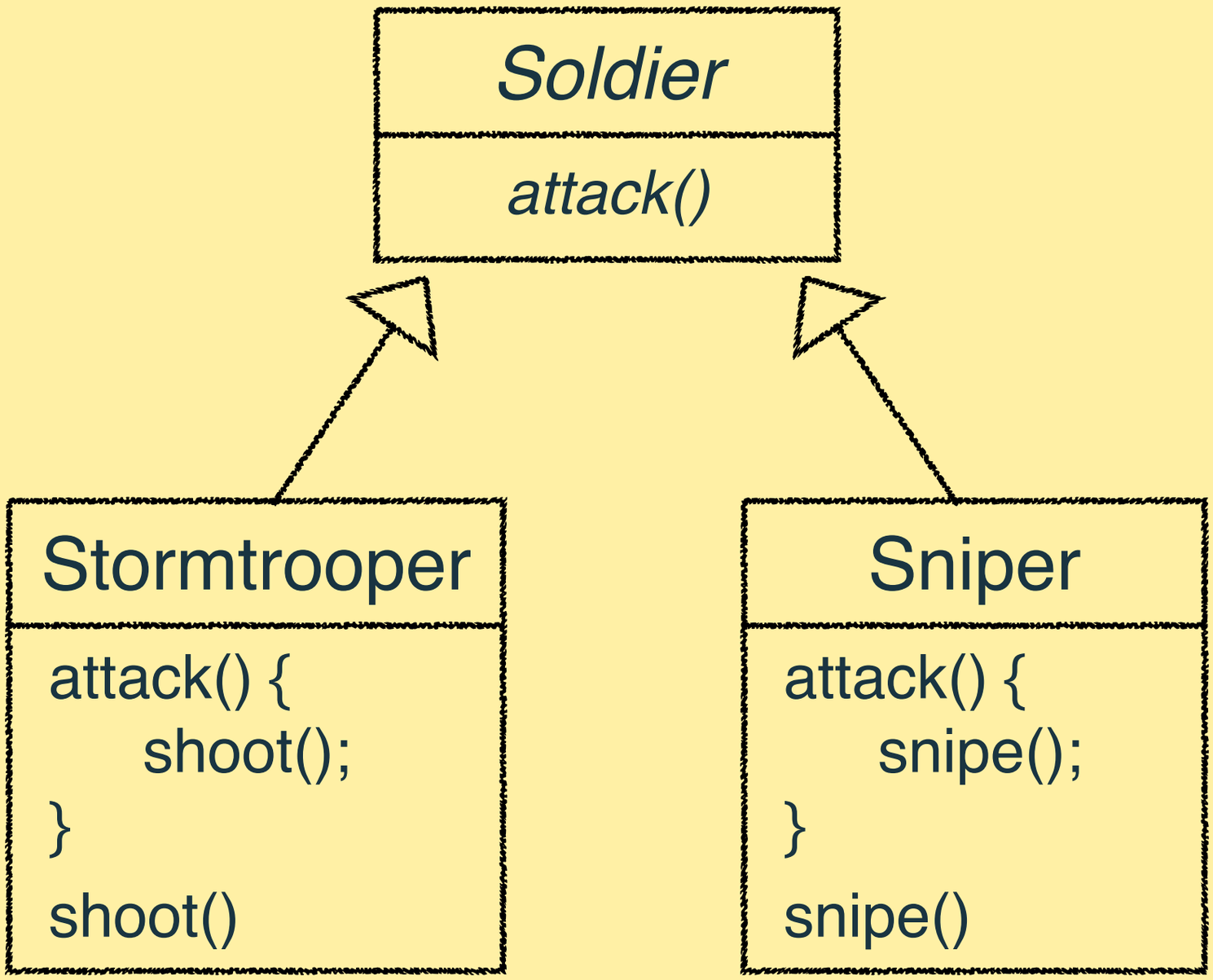
2

Program to an
interface,
not an implementation.

> Program to an interface.
> 真正的意思是
> Program to a supertype.

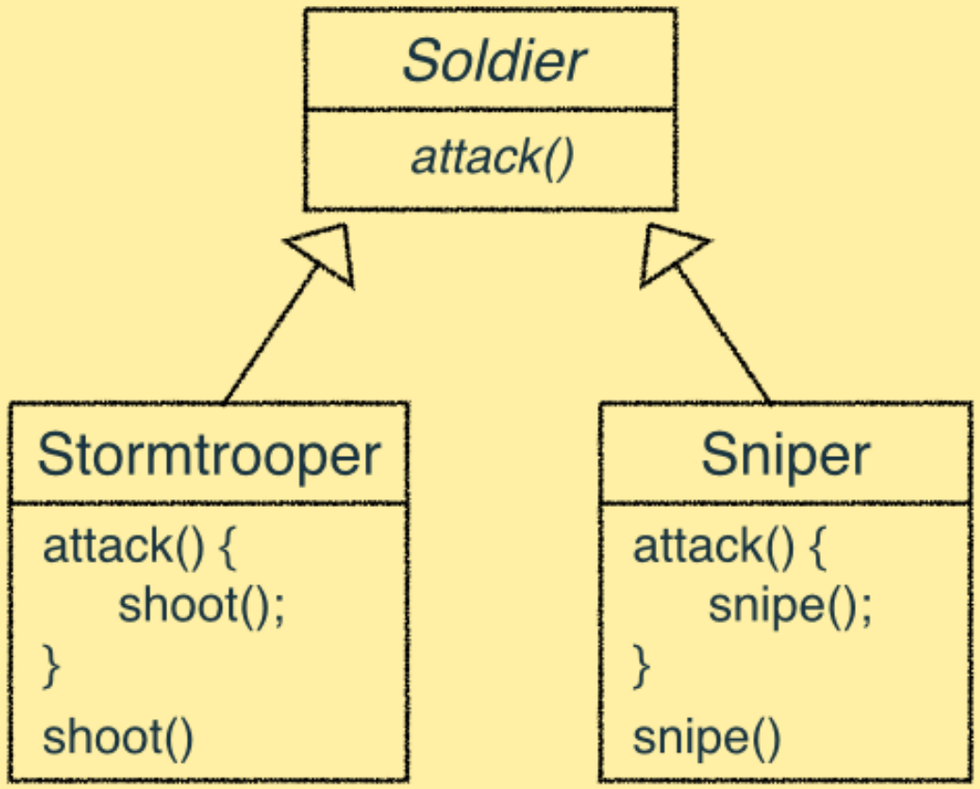
Example

Supertype 可以是
抽象類別或介面



Program to an implement

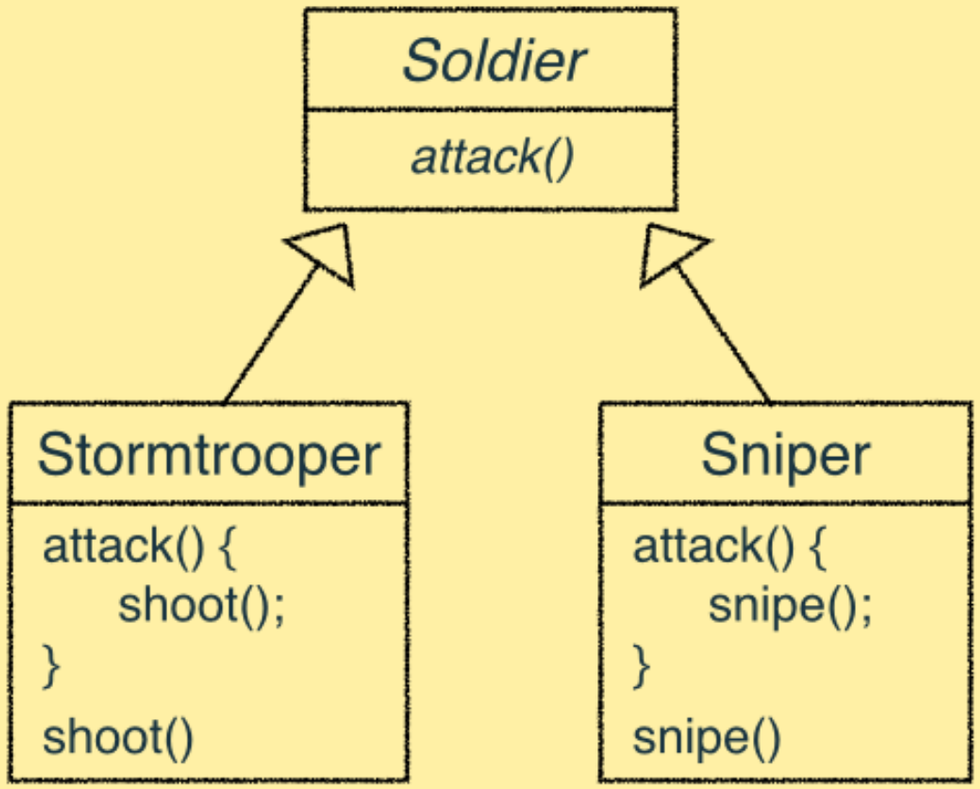
```
Sniper soldier = new Sniper();
soldier.attack();
```



Program to an interface

```
Sniper soldier = new Sniper();
soldier.attack();
```

```
Soldier soldier = new Sniper();
soldier.attack();
```

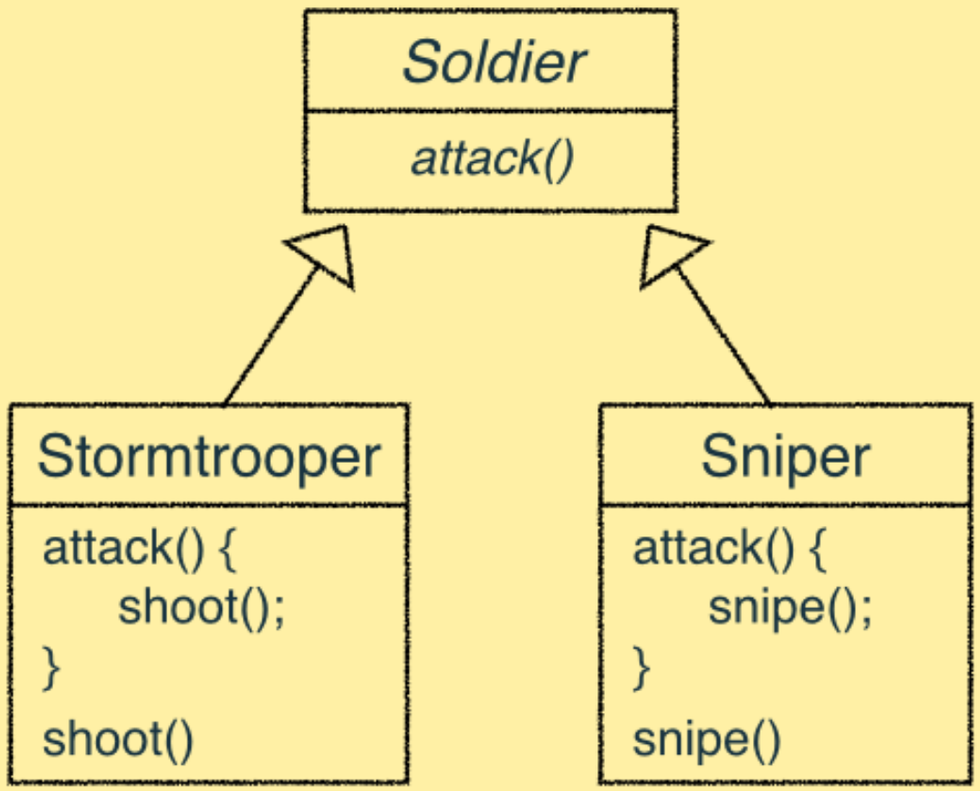


Program to an interface

```
Sniper soldier = new Sniper();
soldier.attack();
```

```
Soldier soldier = new Sniper();
soldier.attack();
```

```
Soldier soldier = getSoldier();
soldier.attack();
```



Better solution：

在執行期間，才指定實踐的物件

回到模擬鴨子

發聲行為

<< interface >>

QuackBehavior

quack()

飛行行為

<< interface >>

FlyBehavior

fly()

發聲行為

<< interface >>

QuackBehavior

quack()

飛行行為

<< interface >>

FlyBehavior

fly()

Quack

quack() { 呱呱叫; }

MuteQuack

quack() { 不會叫; }

Squeak

quack() { 吱吱叫; }

FlyWithWings

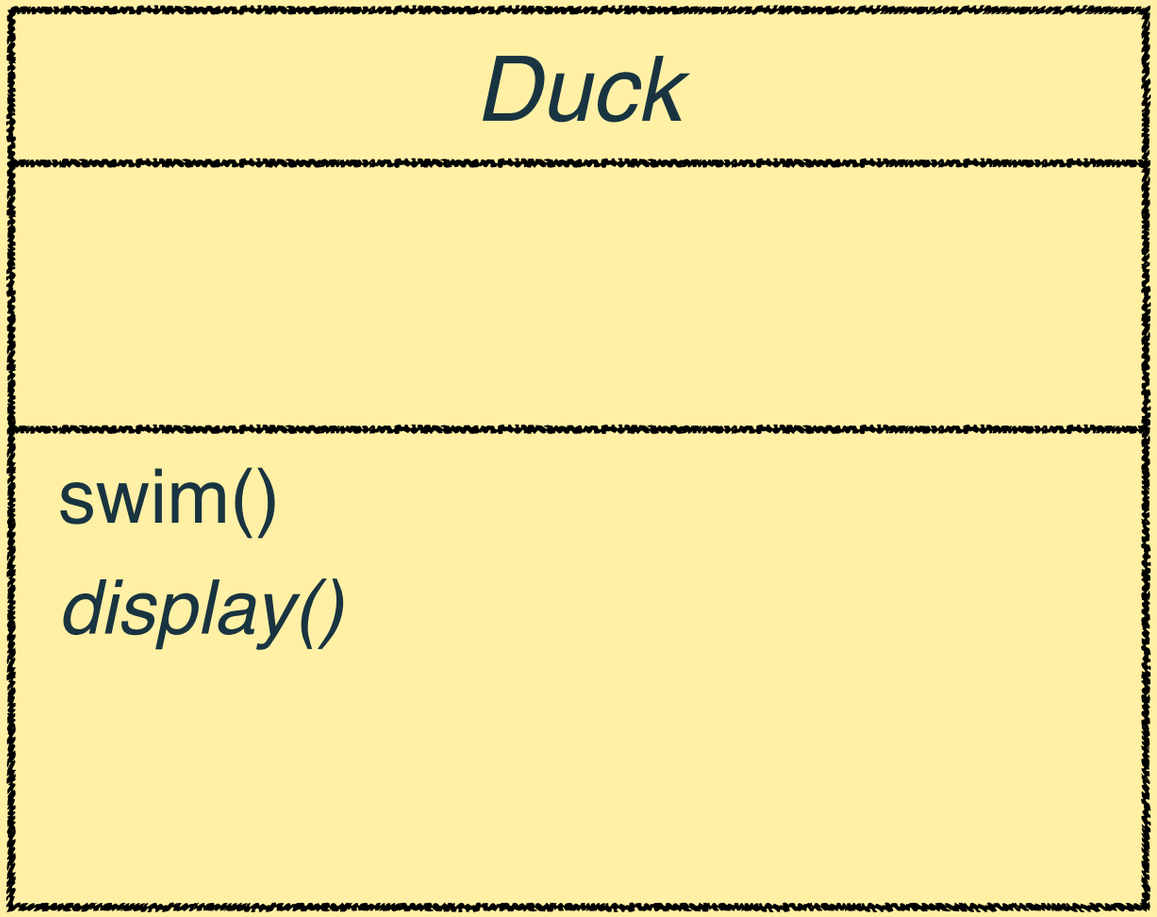
fly() { 用翅膀飛; }

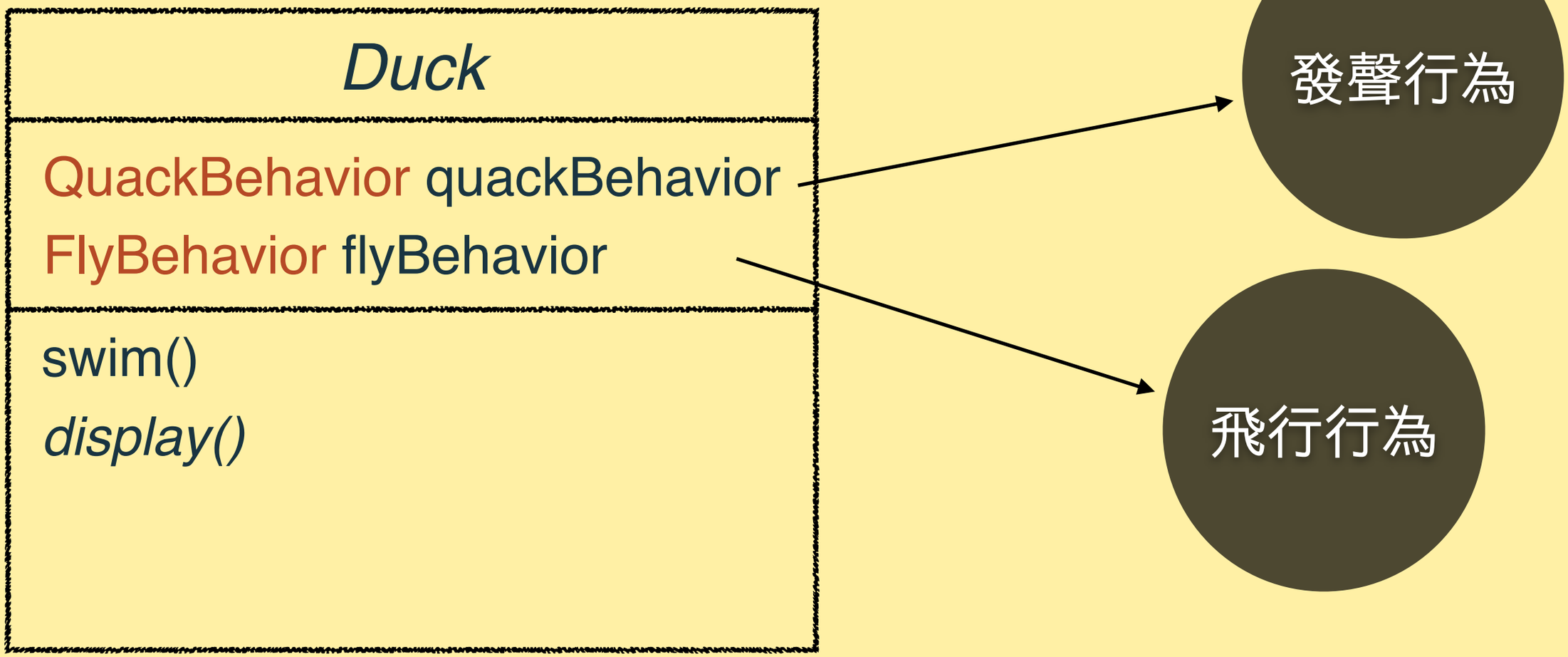
FlyNoWay

fly() { 不會飛; }

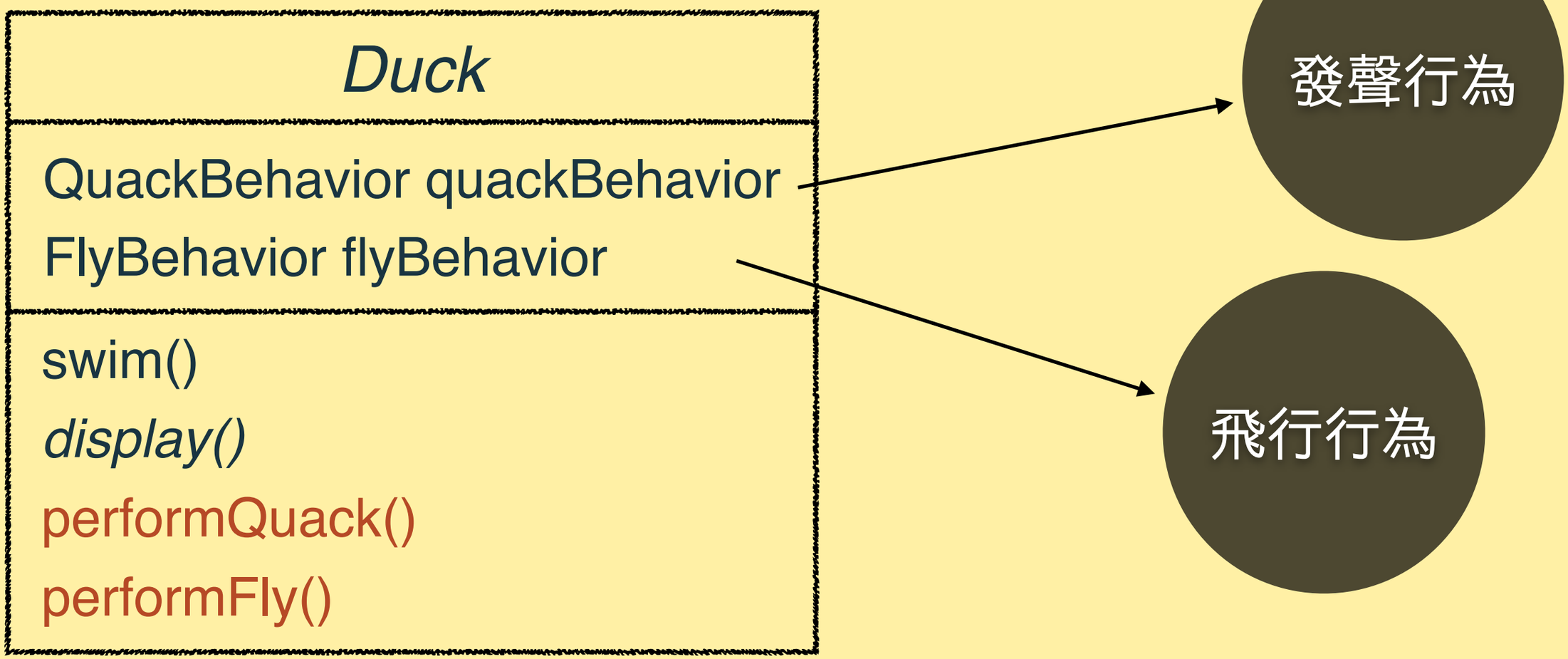
FlyWithRocket

fly() { 用火箭飛; }


Duck
swim()
display()




```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;
}
```




```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}
}
```

Duck

QuackBehavior qu

FlyBehavior flyBe

swim()

display()

performQuack()

performFly()

關鍵在於，

鴨子現在會將它的行為，

委派（delegate）給別人處理，

而不是將它定義在自己或子類別裡！

```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}
}
```

那如何設定鴨子行為的實體變數？

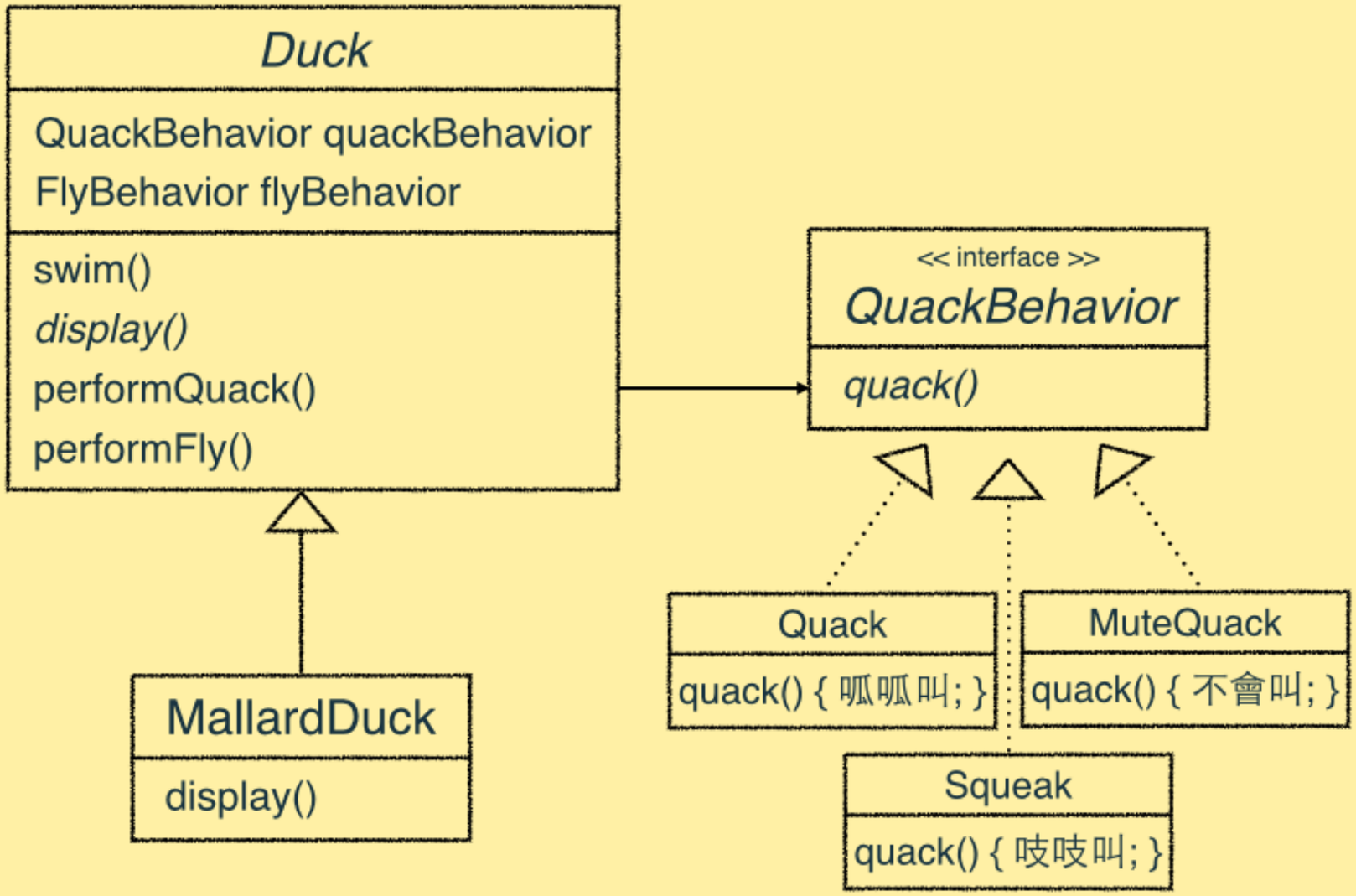
```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}
}
```

在子類別的建構式裡...

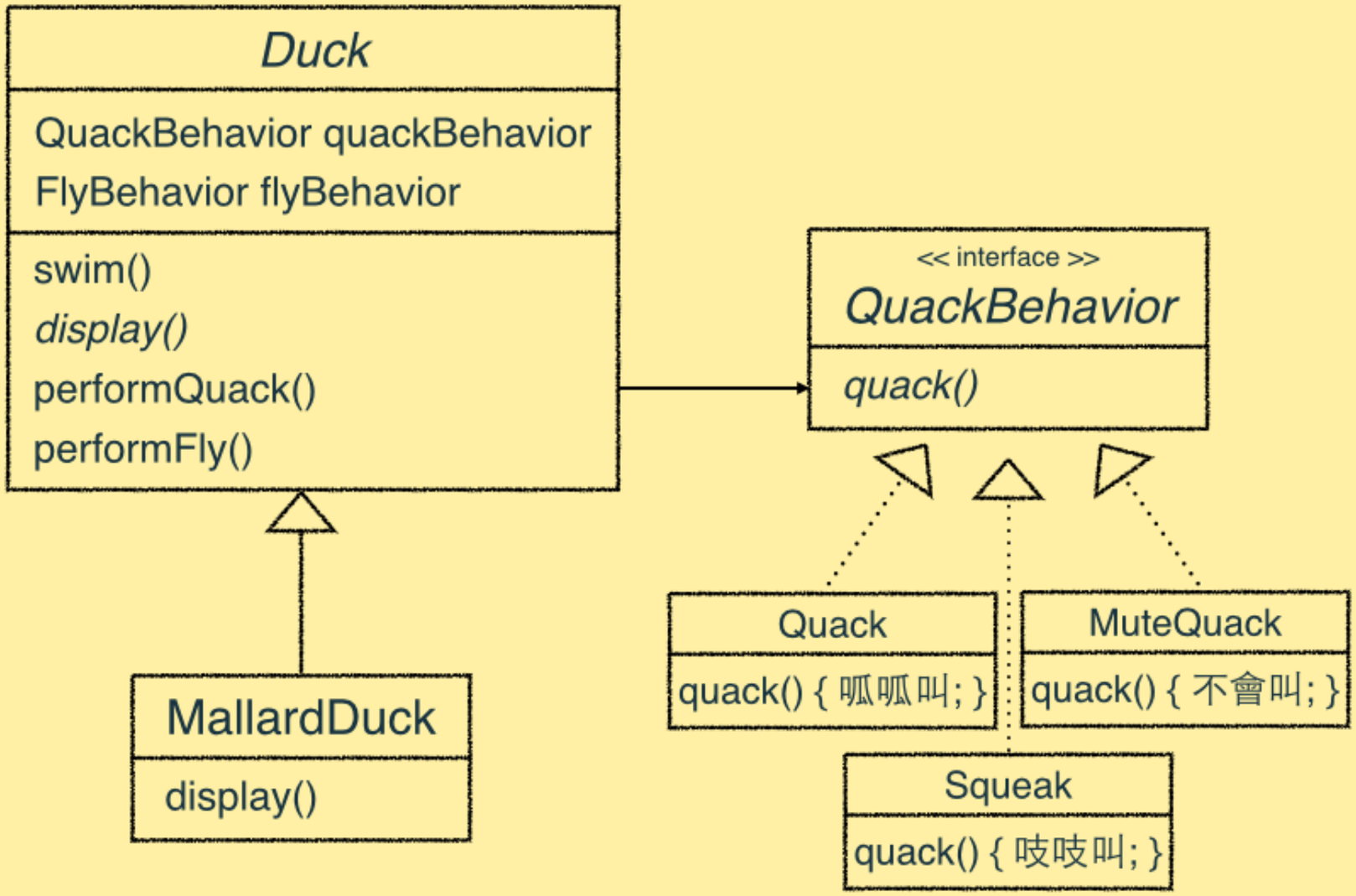
```
public class MallardDuck extends Duck {

    public MallardDuck() {
        quackBehavior = new Quack();
        flyBehavior = new FlyWithNoWings();
    }

    public void display() {…}
}
```

```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}
}
```

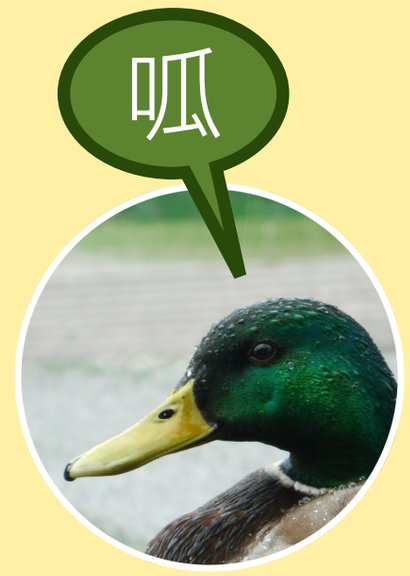
測試



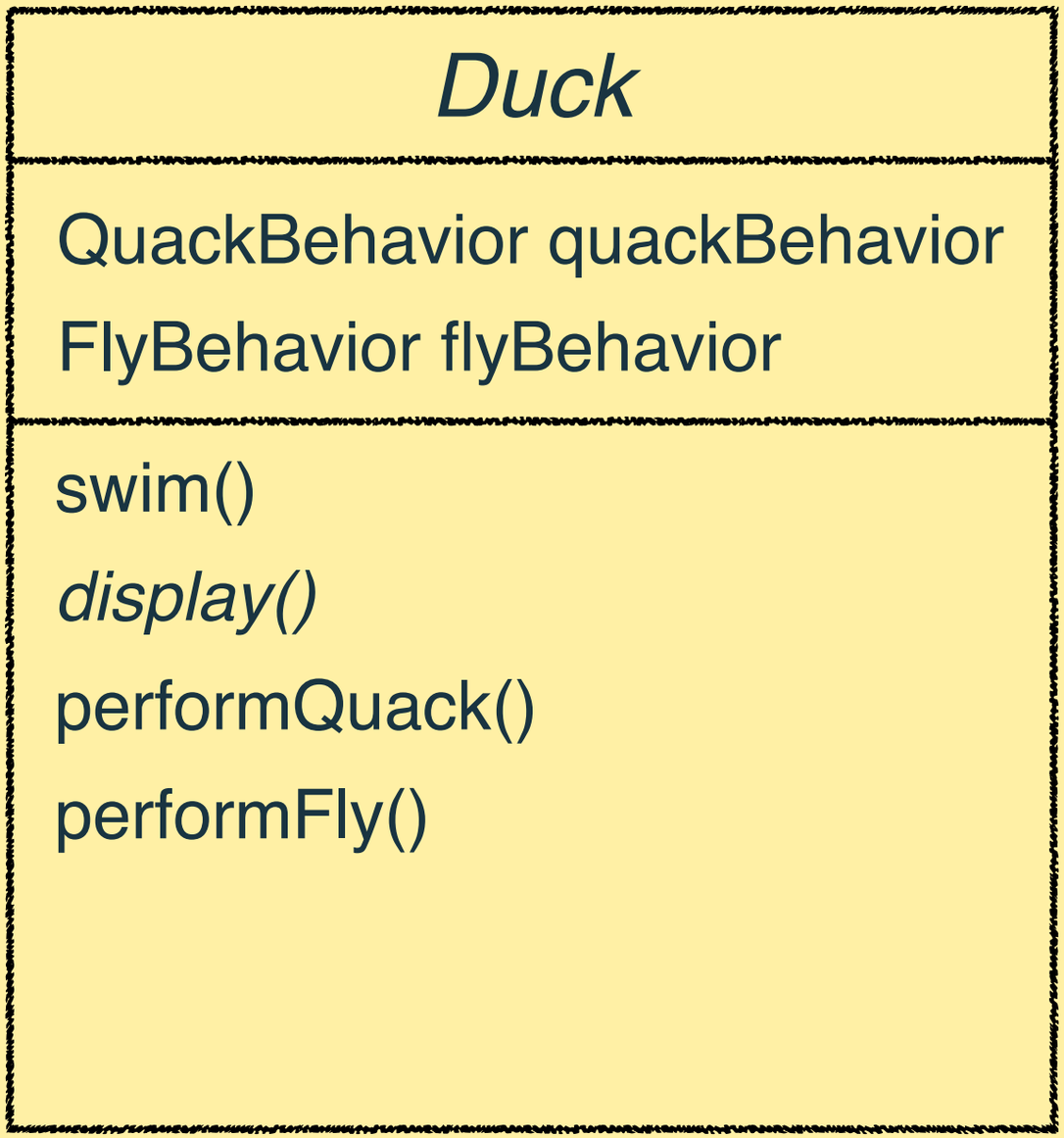

```
public class DuckSimulator {

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



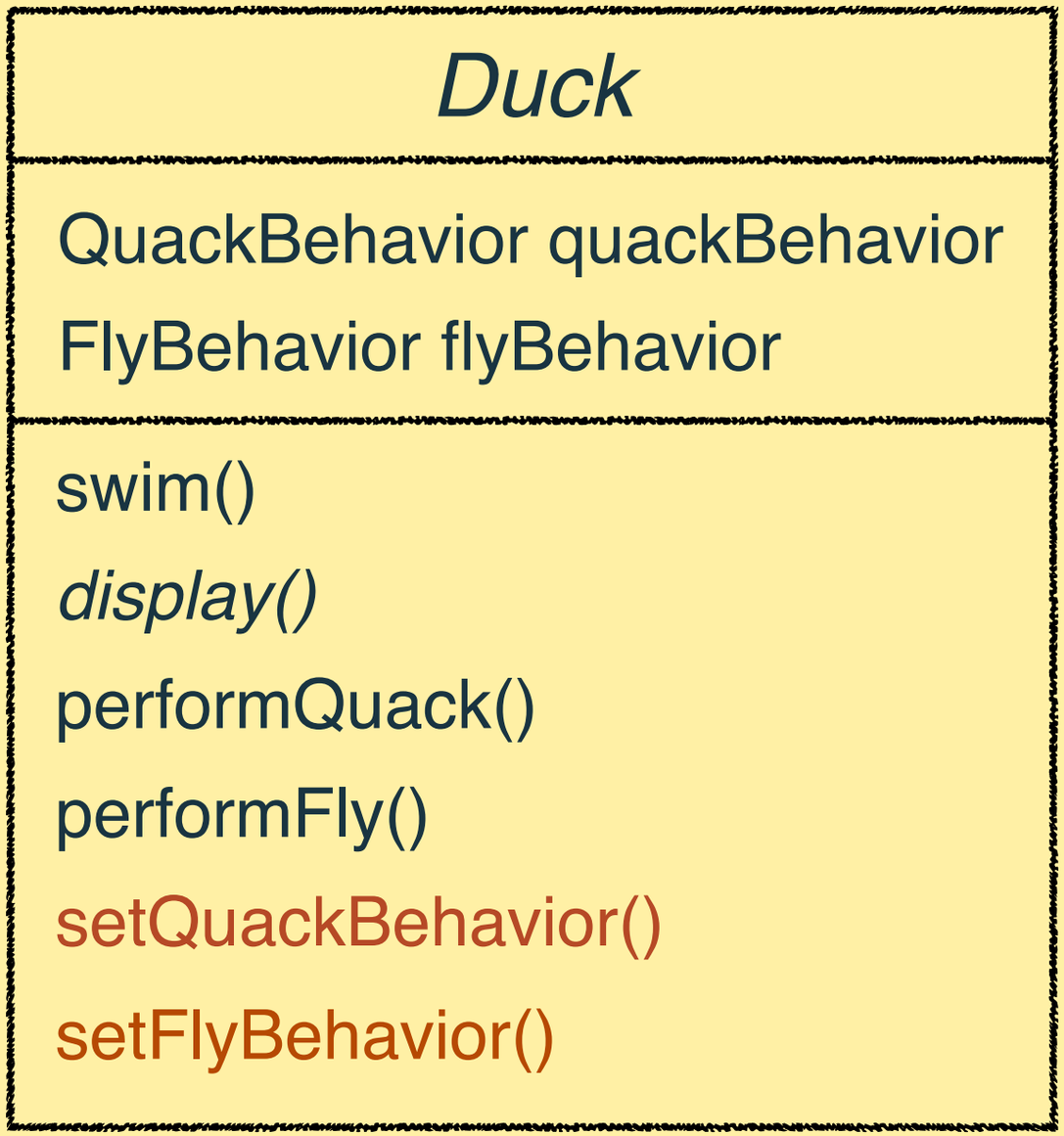

```
public class DuckSimulator {

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



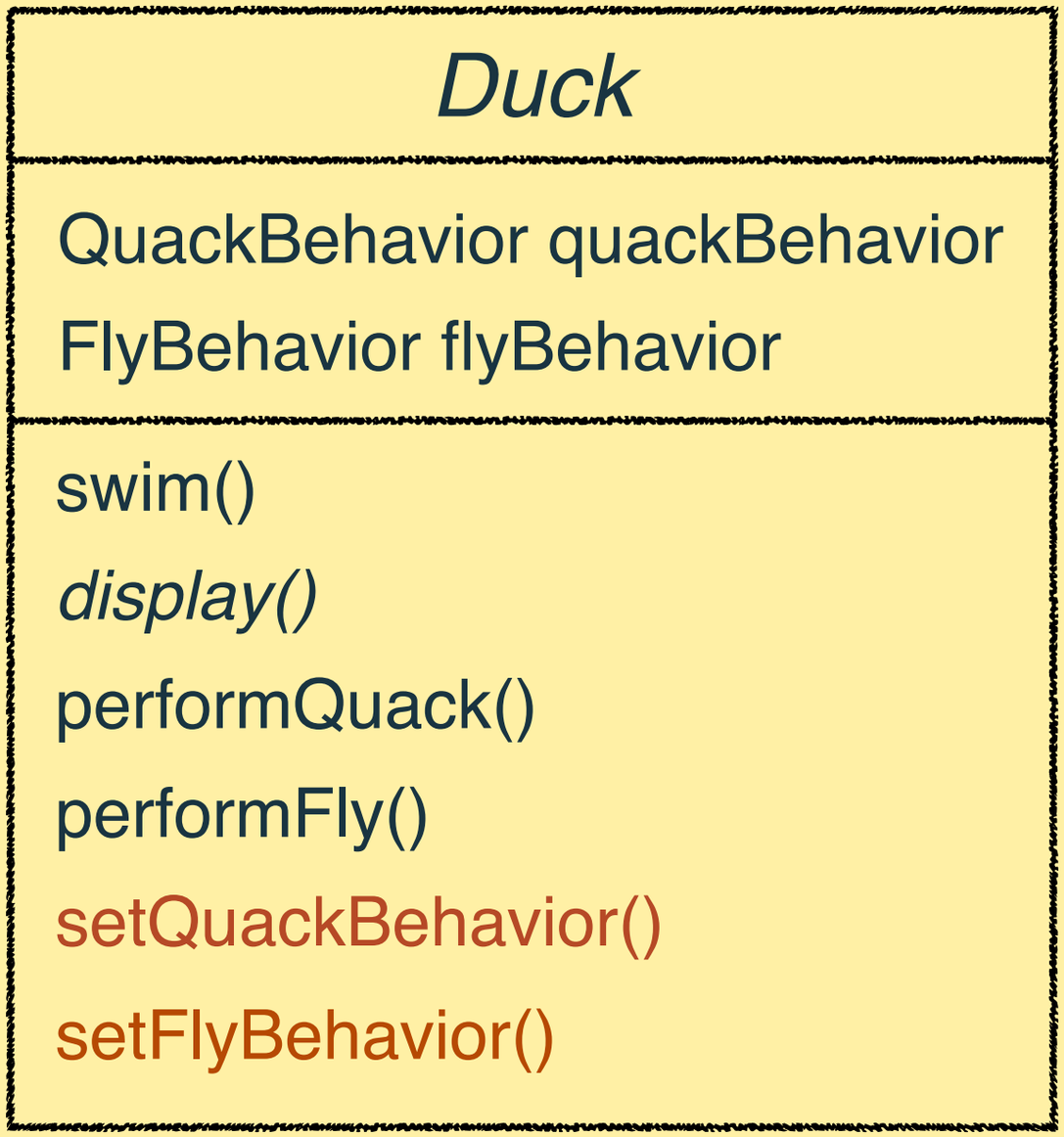
動態設定鴨子行為




```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}

}
```

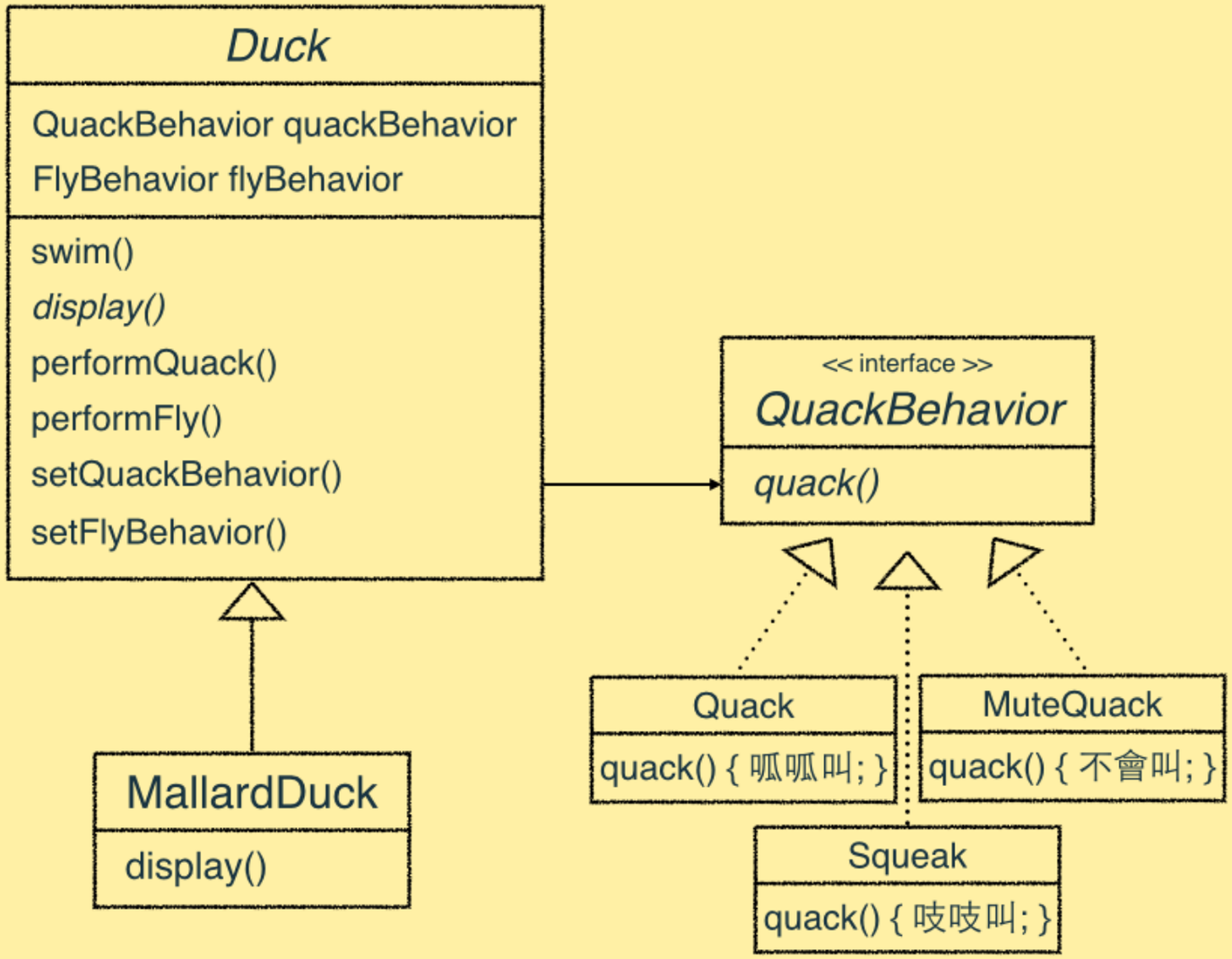



```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}

}
```

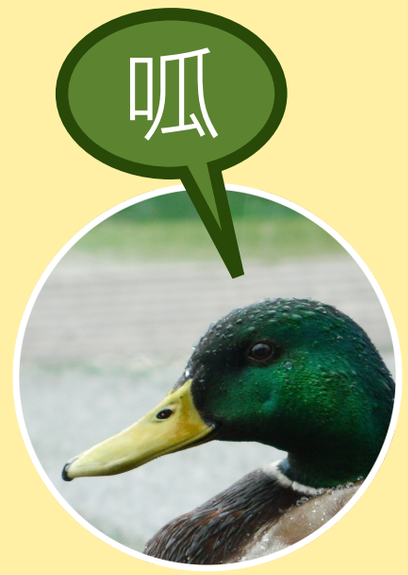


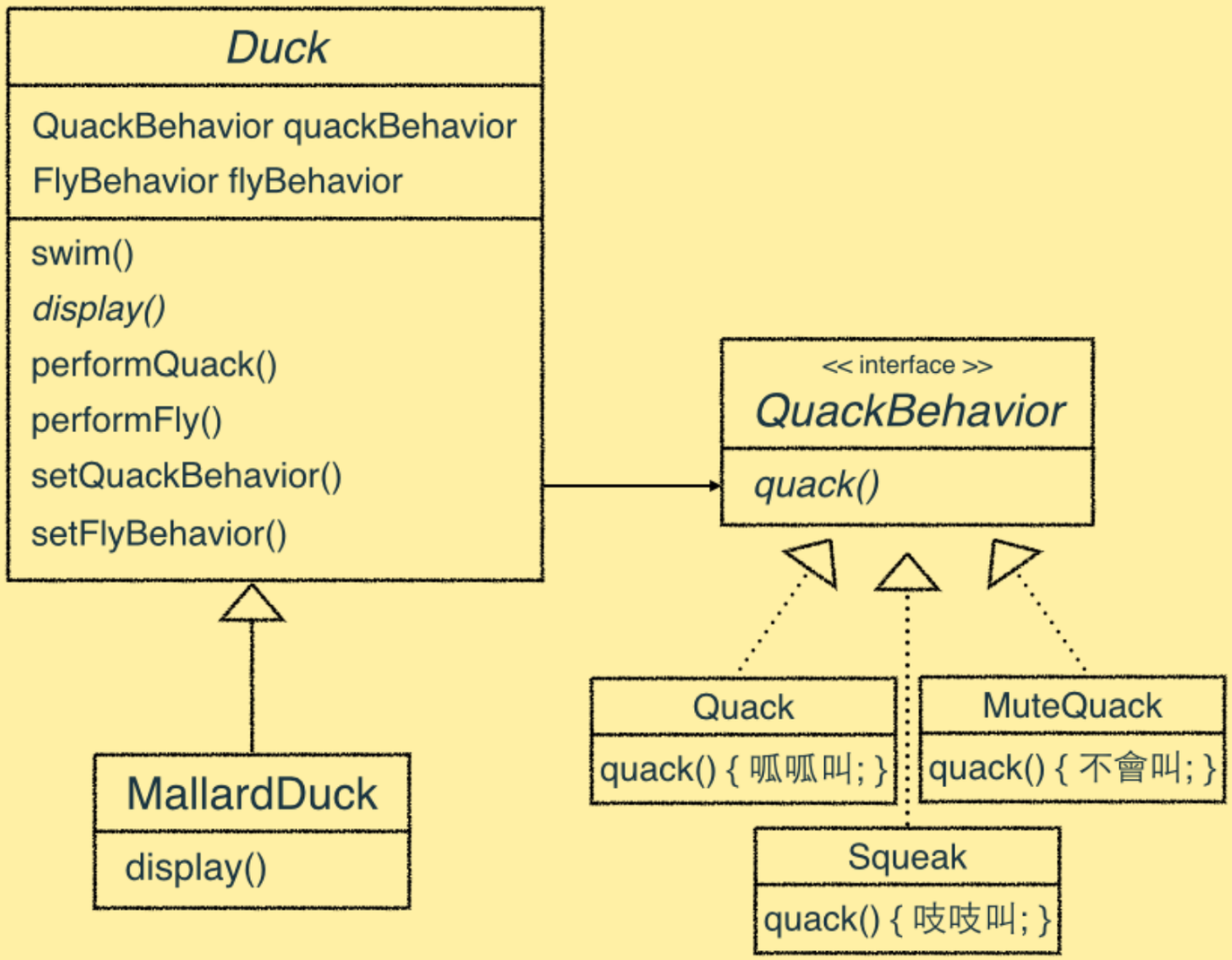

```
public abstract class Duck {
    QuackBehavior quackBehavior;
    FlyBehavior flyBehavior;

    public void performQuack() {
        quackBehavior.quack();
    }
    public void performFly() {…}

    public void setQuackBehavior(
            QuackBehavior behavior) {
        this.quackBehavior = behavior;
    }
    public void setFlyBehavior…
}
```

測試

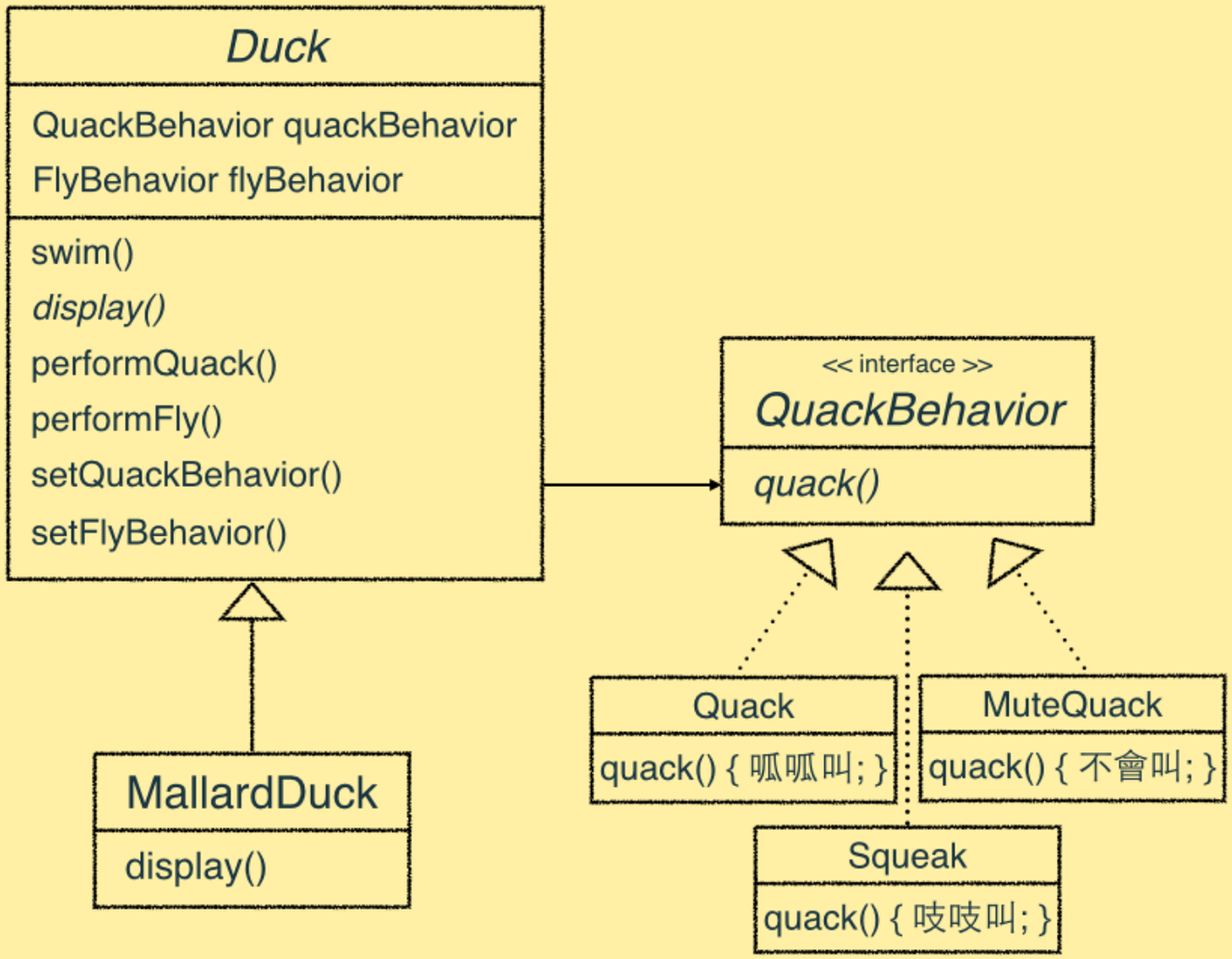



```
public class DuckSimulator {

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

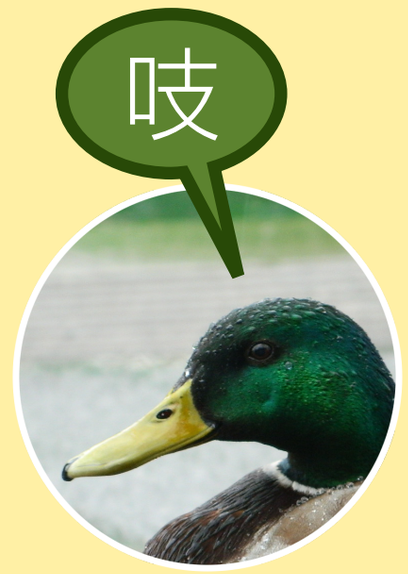





```
public class DuckSimulator {

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

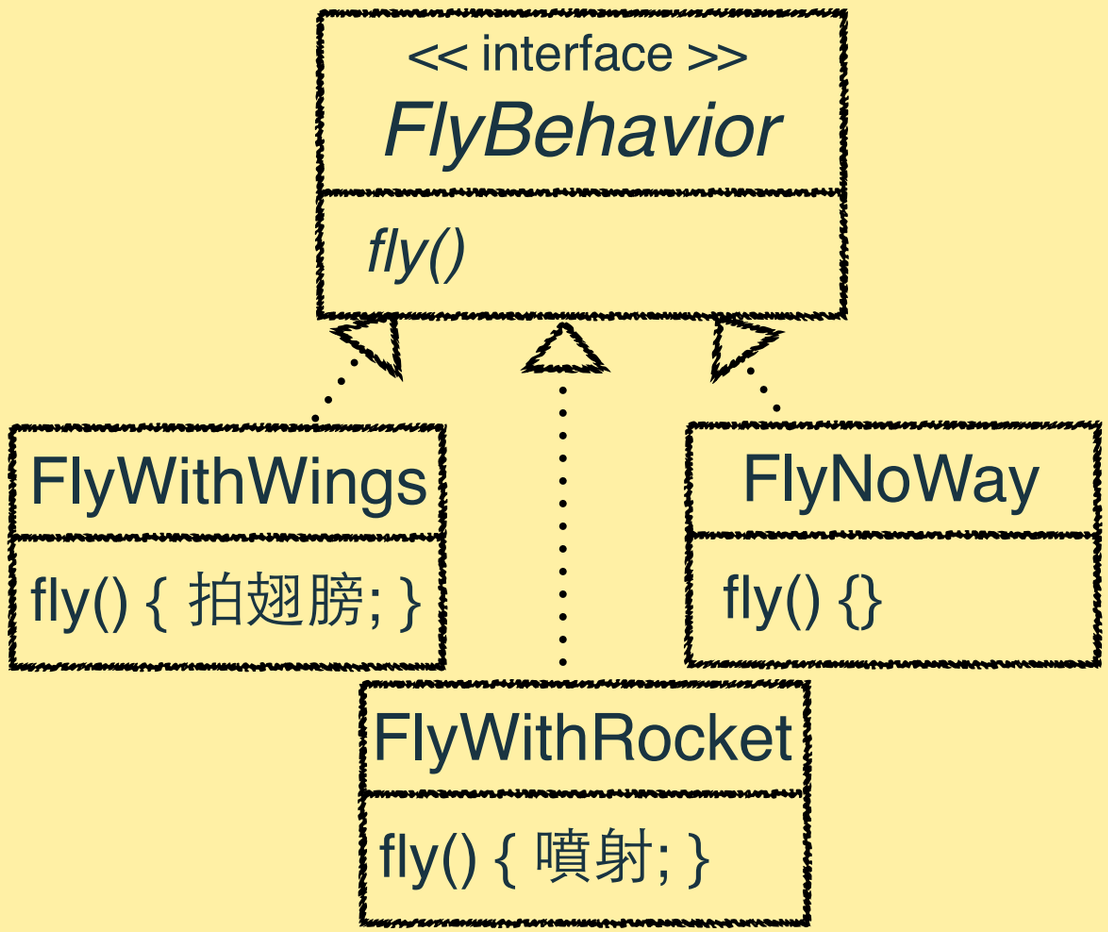


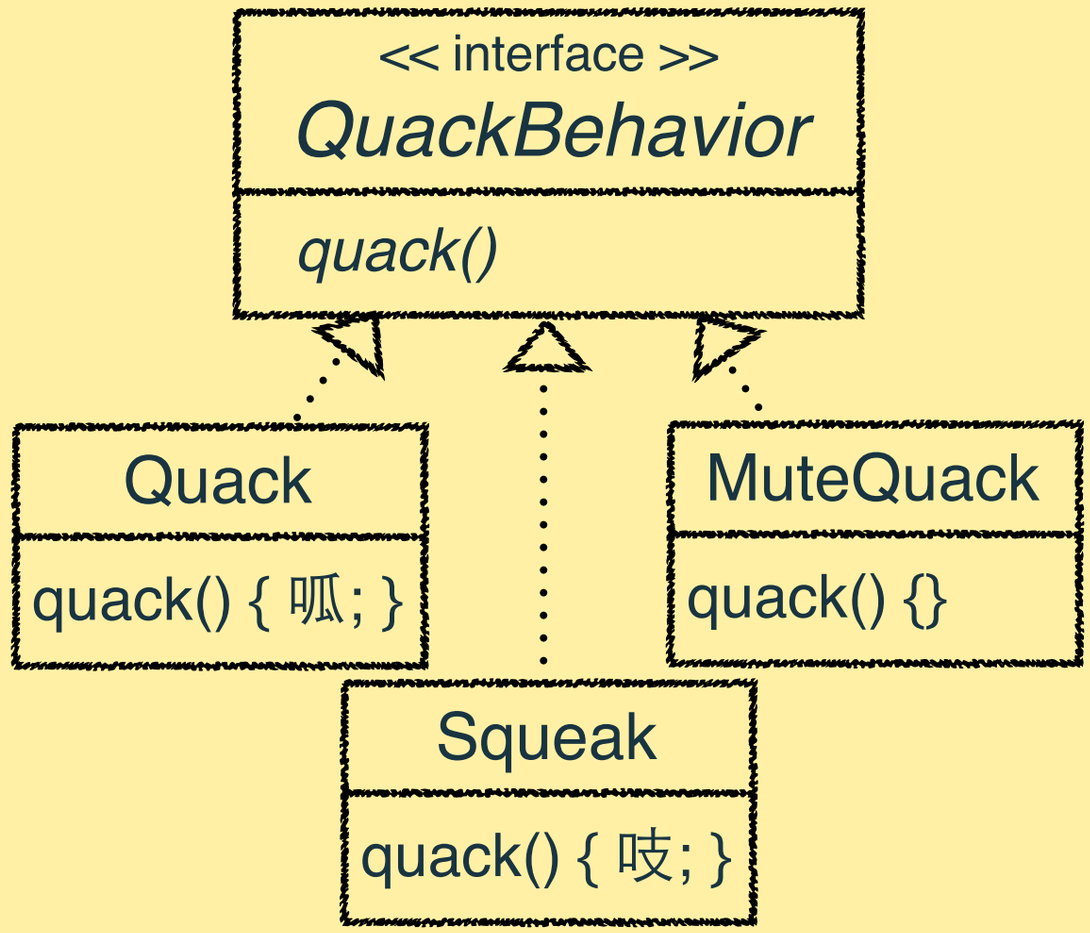

```
public class DuckSimulator {

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

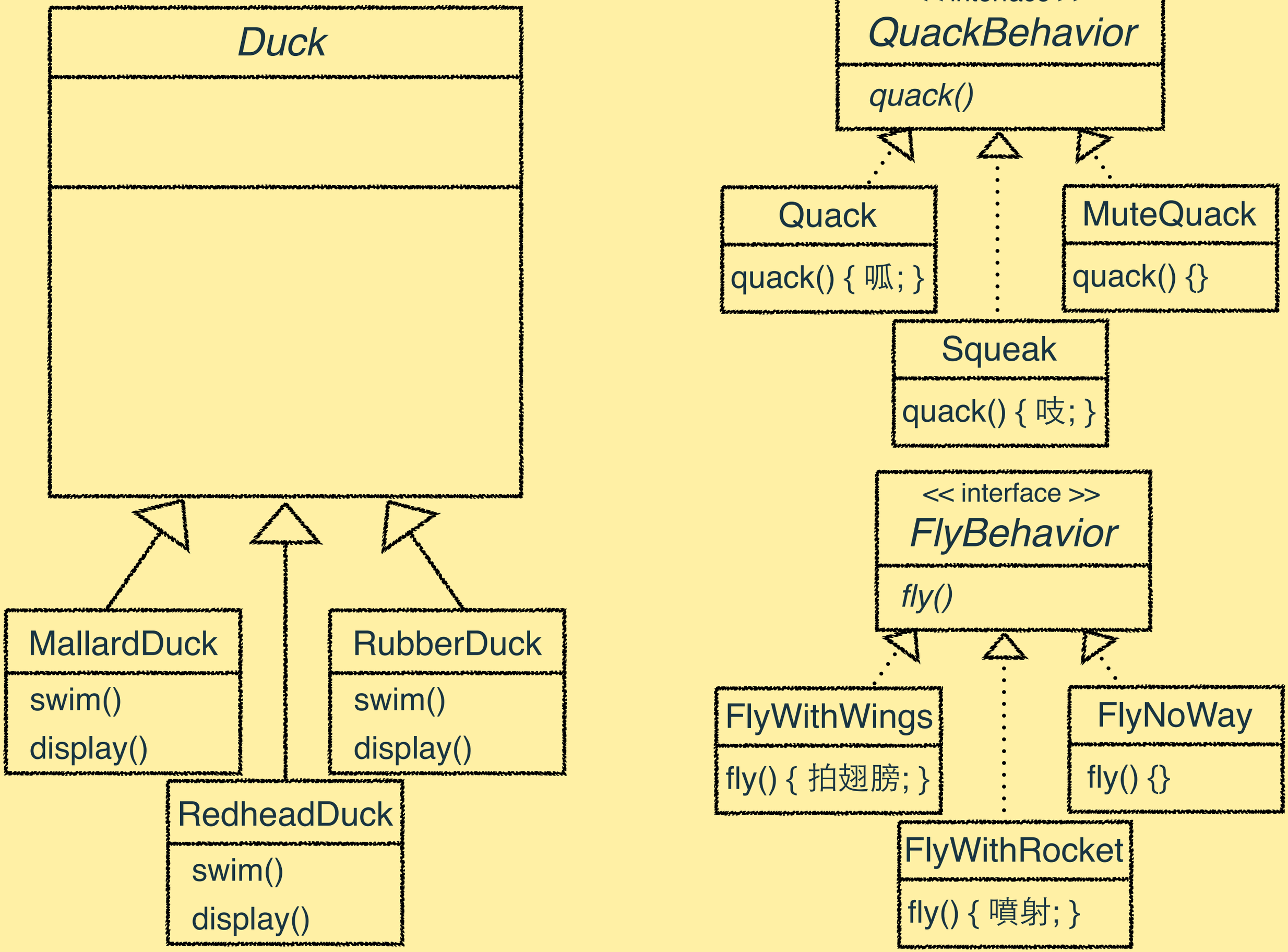


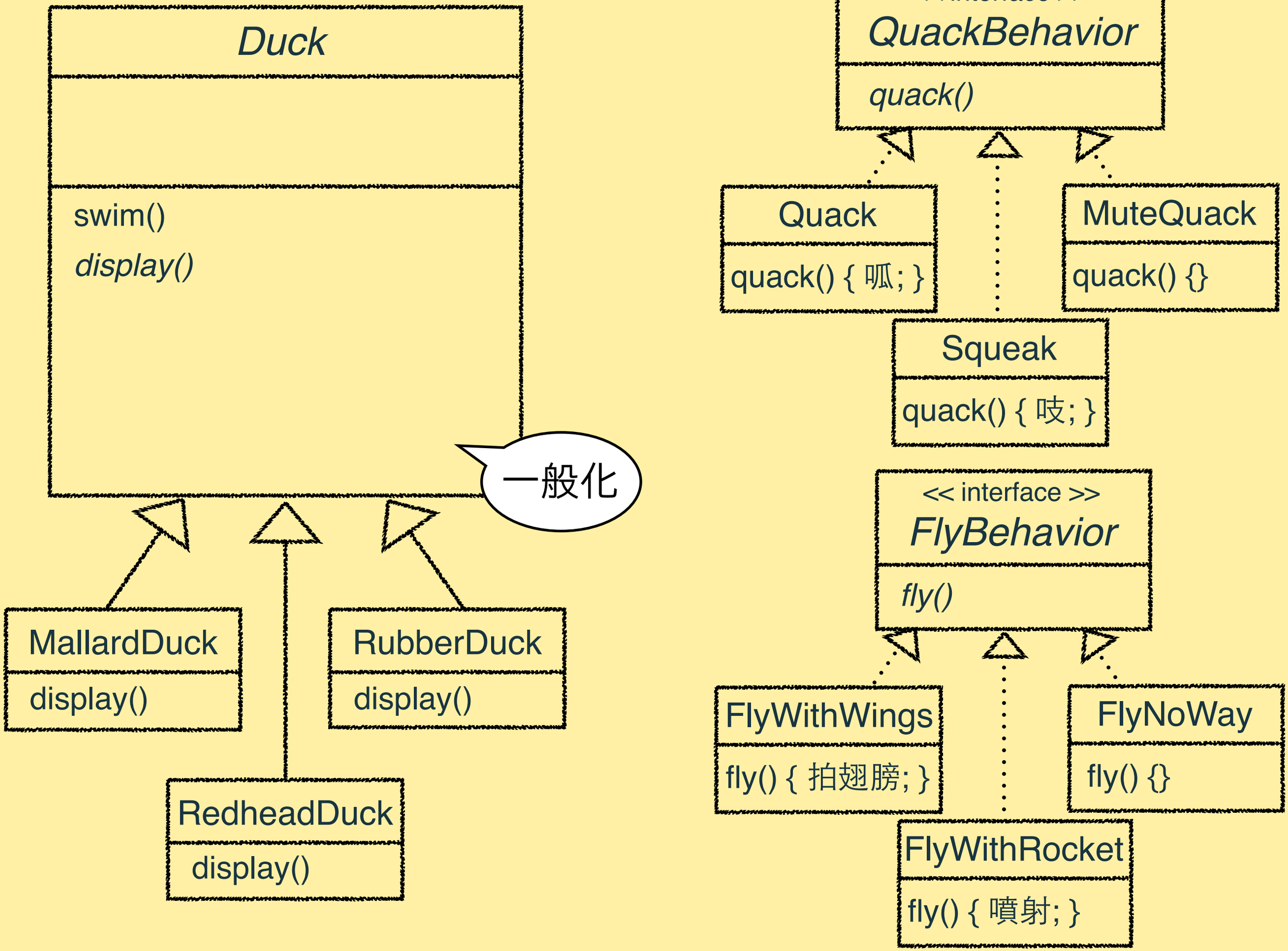
我們到底做了哪些事？

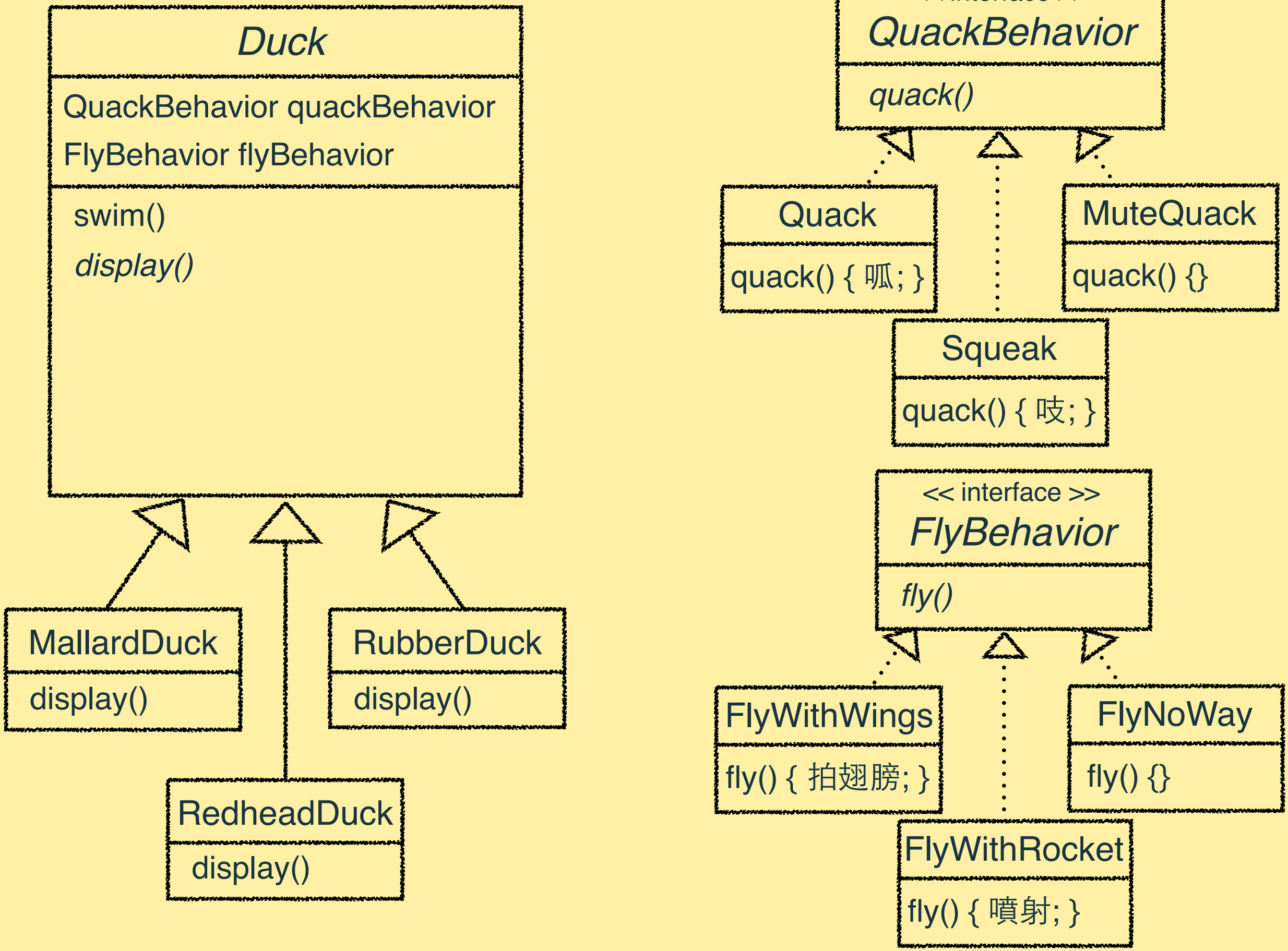
封裝鴨子行為

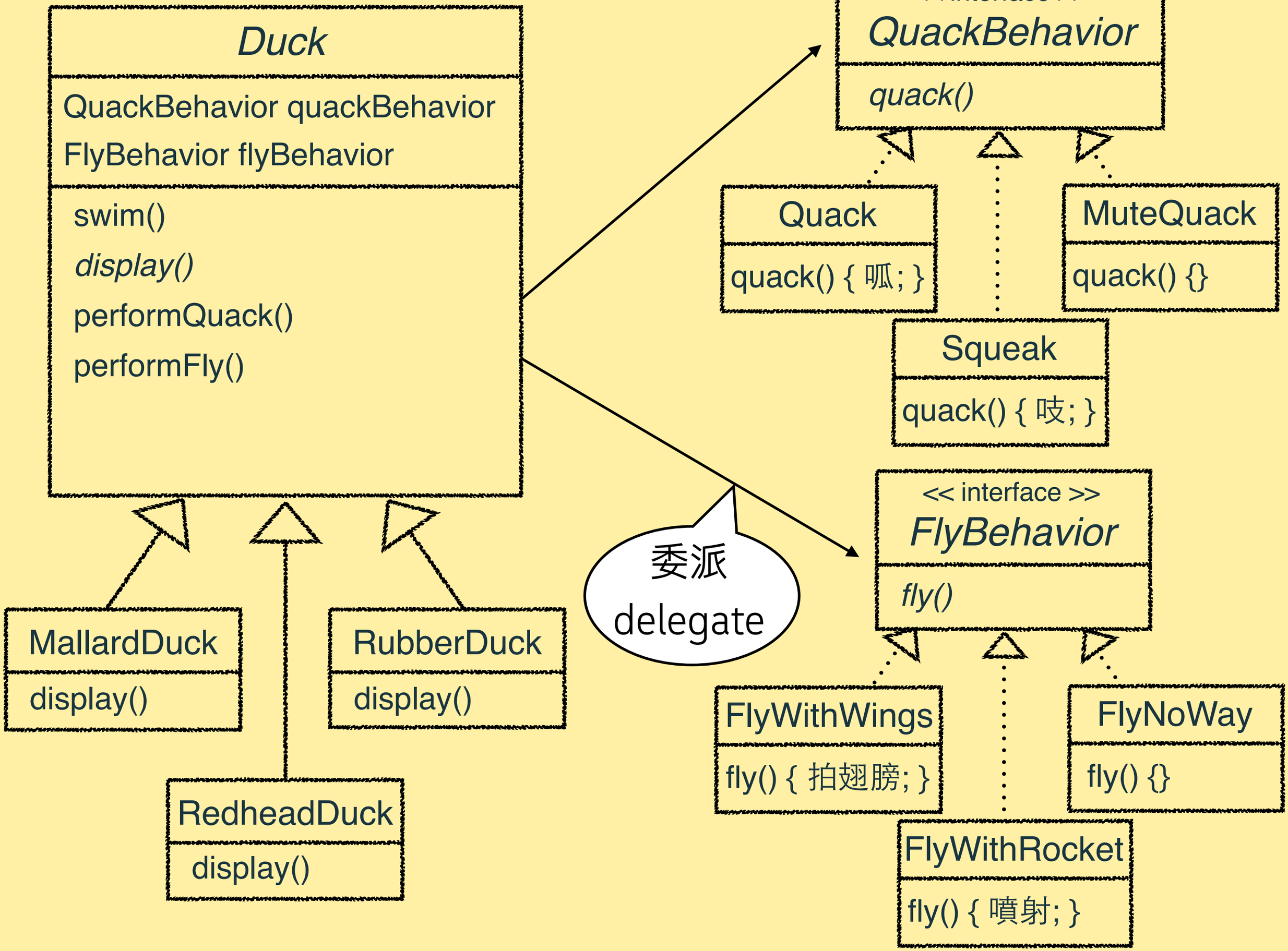

<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }

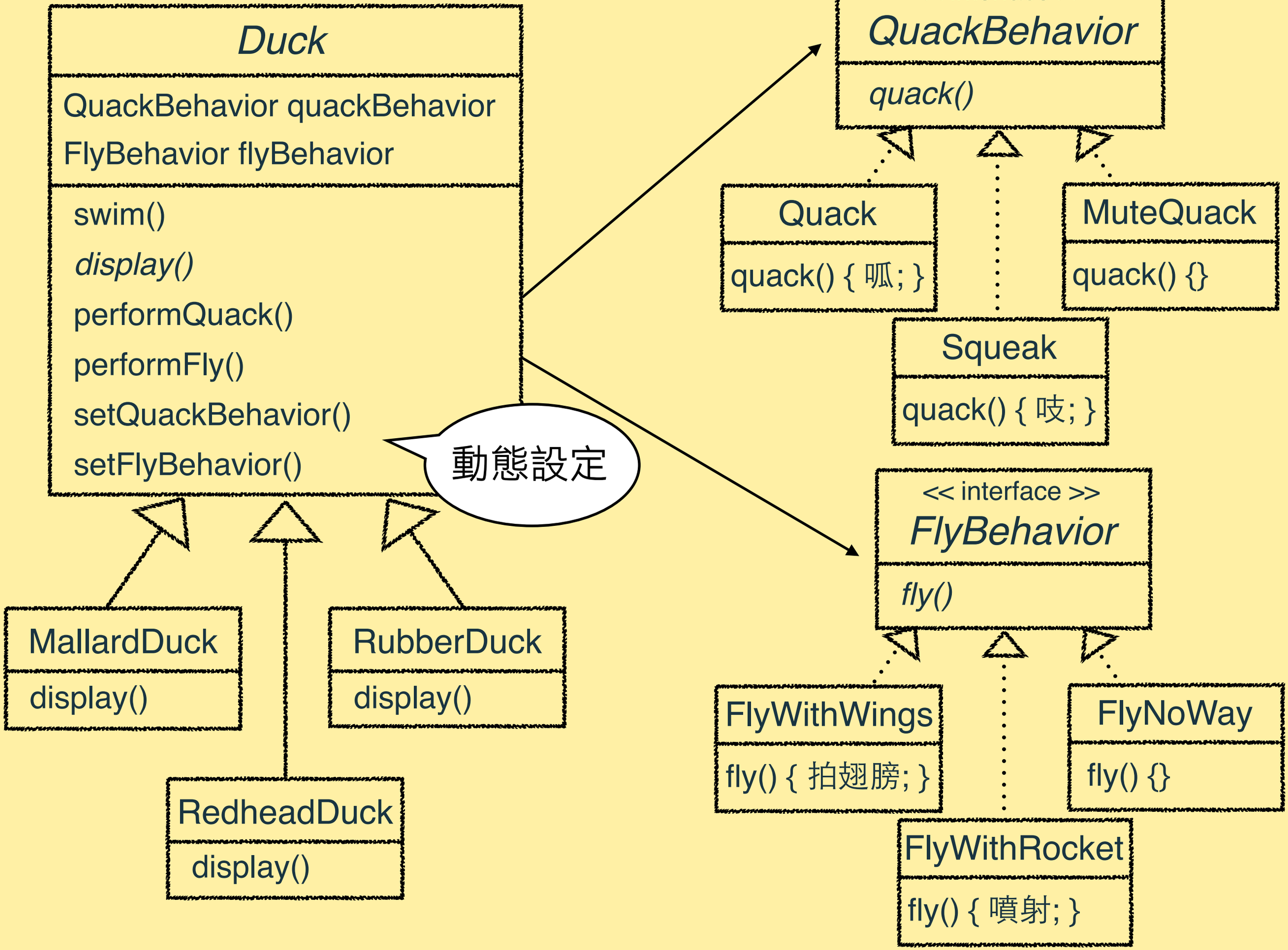

<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }

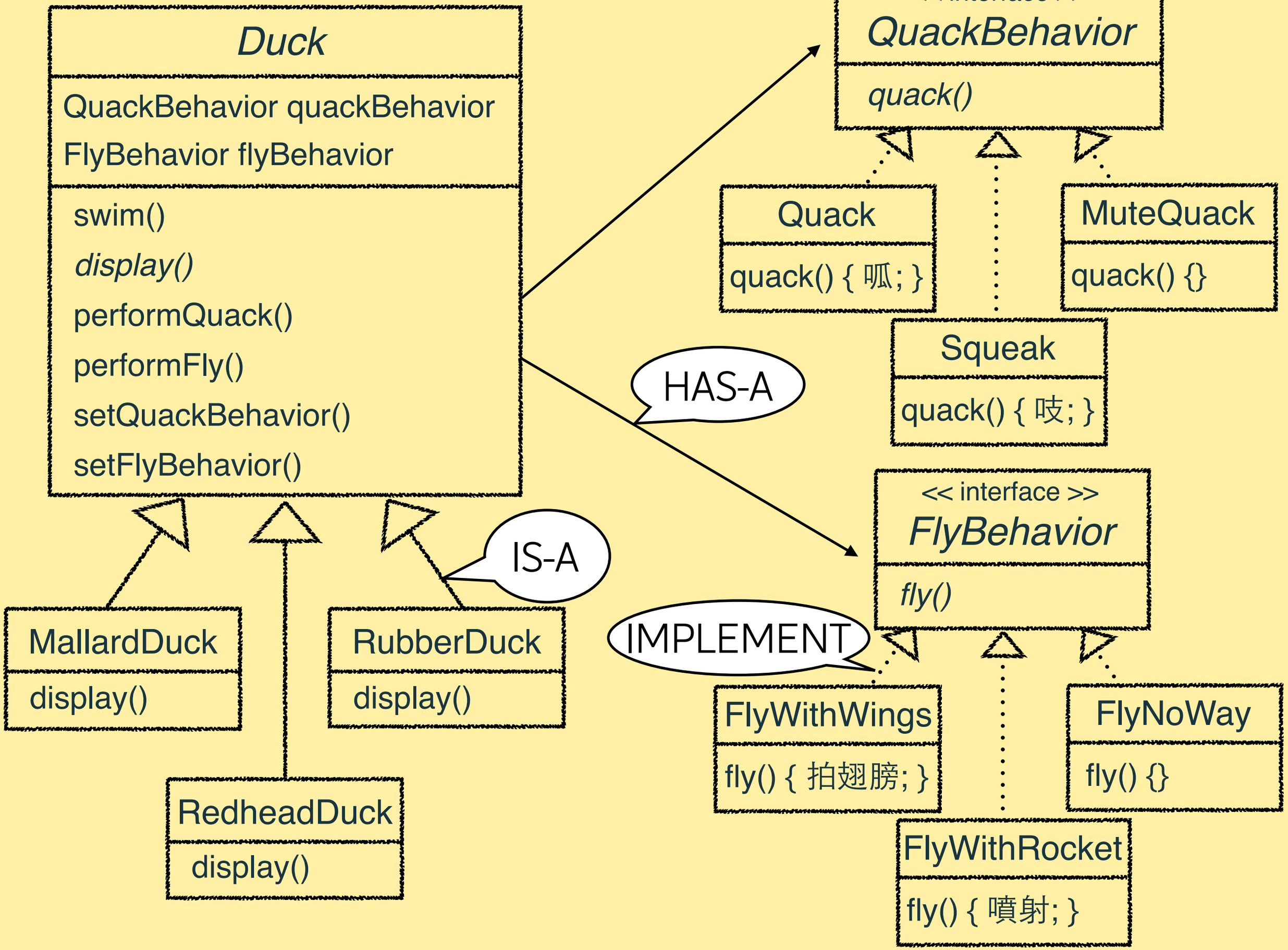

Duck
MallardDuck
swim()
display()
RubberDuck
swim()
display()
RedheadDuck
swim()
display()
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }


Duck
swim()
display()
一般化
MallardDuck
display()
RubberDuck
display()
RedheadDuck
display()
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }


Duck
QuackBehavior quackBehavior
FlyBehavior flyBehavior
swim()
display()
MallardDuck
display()
RubberDuck
display()
RedheadDuck
display()
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }


Duck
QuackBehavior quackBehavior
FlyBehavior flyBehavior
swim()
display()
performQuack()
performFly()
MallardDuck
display()
RubberDuck
display()
RedheadDuck
display()
委派
delegate
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }


Duck
QuackBehavior quackBehavior
FlyBehavior flyBehavior
swim()
display()
performQuack()
performFly()
setQuackBehavior()
setFlyBehavior()
動態設定
MallardDuck
display()
RubberDuck
display()
RedheadDuck
display()
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
<< interface >>
FlyBehavior
fly()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }


Duck
QuackBehavior quackBehavior
FlyBehavior flyBehavior
swim()
display()
performQuack()
performFly()
setQuackBehavior()
setFlyBehavior()
<< interface >>
QuackBehavior
quack()
Quack
quack() { 呱; }
MuteQuack
quack() {}
Squeak
quack() { 吱; }
HAS-A
IS-A
<< interface >>
FlyBehavior
fly()
IMPLEMENT
MallardDuck
display()
RubberDuck
display()
RedheadDuck
display()
FlyWithWings
fly() { 拍翅膀; }
FlyNoWay
fly() {}
FlyWithRocket
fly() { 噴射; }

3

多用組合

少用繼承

組合，

讓物件之間的關係變得更彈性，

你甚至可以在執行期更換它們

恭喜你，學會第一個設計模式

恭喜你，學會第一個
設計模式

策略模式
Strategy

策略模式

Strategy

Strategy

定義一組演算法

Strategy

用翅膀飛、火箭飛、不會飛..

鴨子飛行行為

定義一組演算法

將每個演算法都封裝起來

Strategy

用翅膀飛、火箭飛、不會飛..

鴨子飛行行為

定義一組演算法

將每個演算法都封裝起來

讓他們之間可以相互替換

黃色小鴨可以隨時用火箭飛，或者用翅膀飛

Strategy

定義一組演算法

將每個演算法都封裝起來

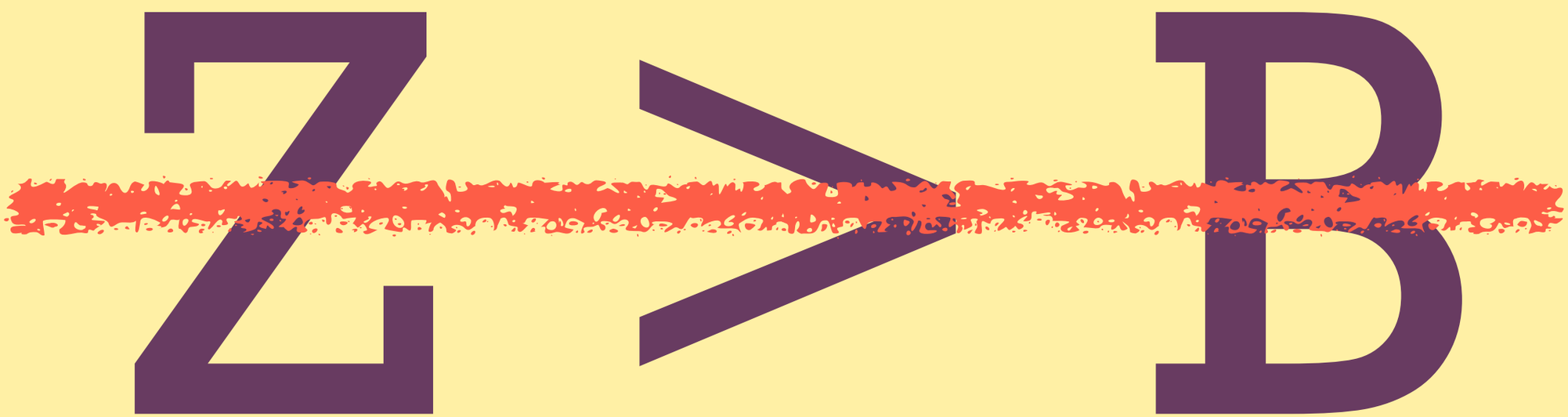
讓他們之間可以相互替換

策略模式讓演算法的變動

不會影響到使用演算法的程式

為什麼我們需要學設計模式

Z > B


Z > B

McDonald

Alice:
我要六塊雞塊，兩包中薯，兩塊麥脆雞，四塊勁辣香雞翅，還有三杯可樂！

Jason: 三人快樂分享餐！

他們其實點一樣的餐點！

Alice:
我要六塊雞塊，兩包中薯，兩塊麥脆雞，四塊勁辣香雞翅，還有三杯可樂！

Jason: 三人快樂分享餐！

Alice:
Jason，請幫我把所有鴨子的飛行行為取出來；將它們封裝成一個一個類別，並且讓它們實作一個飛行的介面！

之後再幫我讓鴨子的類別可以呼叫這些飛行行為...


Jason: Alice，妳只要說策略模式，我就懂了啦！

設計模式

能夠讓夥伴之間快速的溝通

結論


當變化成為事實
設計就是義務

Thanks

達暉資訊

Jason Chung