對本書的讚譽

「從昨天翻開這本書的第一頁開始，我就停不下來了！於是我把它帶到健身房，一邊運動、一邊閱讀、笑個不停，這本書超酷的，它不僅有趣、涵蓋許多層面，也切中要點，實在令人印象深刻。」

— Erich Gamma，IBM 傑出工程師、《設計模式》四人幫作者之一，
其他三位成員是 Richard Helm、Ralph Johnson 與 John Vlissides

「《深入淺出設計模式》結合有趣的、令人捧腹大笑的元素、深刻的見解和技術，以及實用的建議，是一本有趣且引人深思的讀物。無論你對設計模式一無所知，還是已經使用它們好幾年了，你都可以在物件村裡面學到一些東西。」

— Richard Helm，《設計模式》四人幫作者之一，
其他三位成員是 Erich Gamma、Ralph Johnson 與 John Vlissides

「這本書彷彿從我的腦海卸下一千磅重的書本。」

— Ward Cunningham，維基百科的發明者，Hillside Group 的創始人

「這本書近乎完美，因為它可以在提供專業知識的同時，維持高度的易讀性，行文兼具權威與優雅，是我認為不可或缺的少數幾本軟體書籍之一（除了它之外大概只有 10 本吧）。」

— David Gelernter，耶魯大學電腦科學系教授，
《*Mirror Worlds*》與《*Machine Beauty*》作者

「在設計模式的王國裡，複雜的東西變得簡單，但是簡單的東西也有可能變得複雜。本書帶你快速融入這個王國，我認為 Eric 與 Elisabeth 是最好的導遊。」

— Miko Matsumura，產業分析師，
曾經在 Sun Microsystems 擔任中介軟體公司首席 Java 傳道者

「它深深打動我，讓我歡笑，也讓我哭。」

— Daniel Steinberg，java.net 主編

「我的第一反應是笑得在地上打滾，平復心情之後，我才意識到，這本書不僅十分精確地講解技術面，也是我讀過的設計模式書籍中，最簡單易讀的一本。」

— Timothy A. Budd 博士，俄勒岡大學的電腦科學系副教授，
十餘本書籍的作者，包括《*C++ for Java Programmers*》

「雖然 Jerry Rice 是美式足球聯盟最擅長跑位（run pattern）的接球員，但是他連 Eric 與 Elisabeth 的車尾燈都看不到。說實在的…這本是我看過的軟體設計書籍中，最有趣、最有智慧的一本。」

— Aaron LaBerge，ESPN 技術與產品開發部資深副總裁

碁峰 www.gotop.com.tw

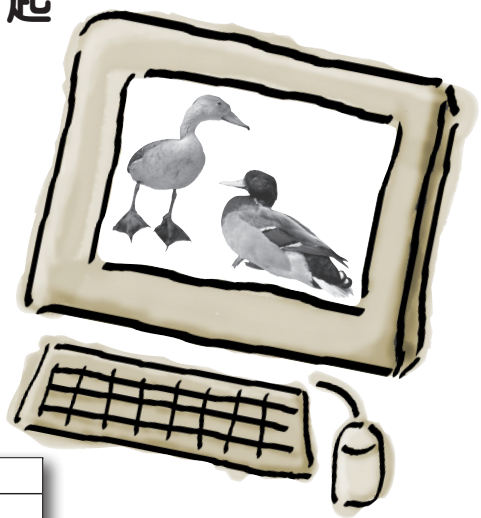
設計模式歡迎您！

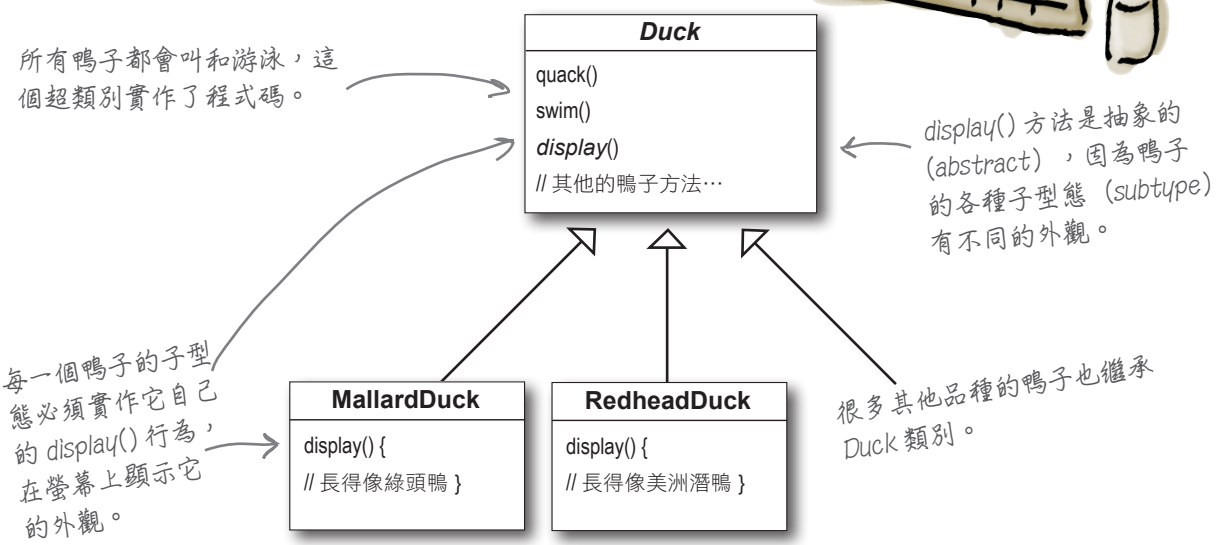
有人已經解決你的問題了。在這一章，你會學到為什麼要利用其他開發者的智慧和經驗，以及如何利用它們。那些開發者都遇過同樣的問題，也曾經順利地解決過它們。在本章結束前，我們會看看設計模式的用途和優點，以及一些重要的物件導向（OO）設計原則，並且透過一個範例來了解設計模式是怎麼運作的。使用模式最好的辦法是：「把模式放入你的腦中，然後在你的設計和應用程式裡認出哪裡可以運用它們。」運用設計模式並非重複使用程式碼，而是重複使用經驗。

www.gotop.com.tw

我們從一個簡單的 SimUDuck app 看起

Joe 的公司做出一個超夯的鴨子池塘模擬遊戲，稱為 *SimUDuck*。遊戲裡面有各種鴨子品種，牠們會一邊游泳，一邊鳴叫。當初設計這個系統的人使用了標準的物件導向技術，並且建立一個 Duck 超類別，來讓所有其他的鴨子品種可以繼承。



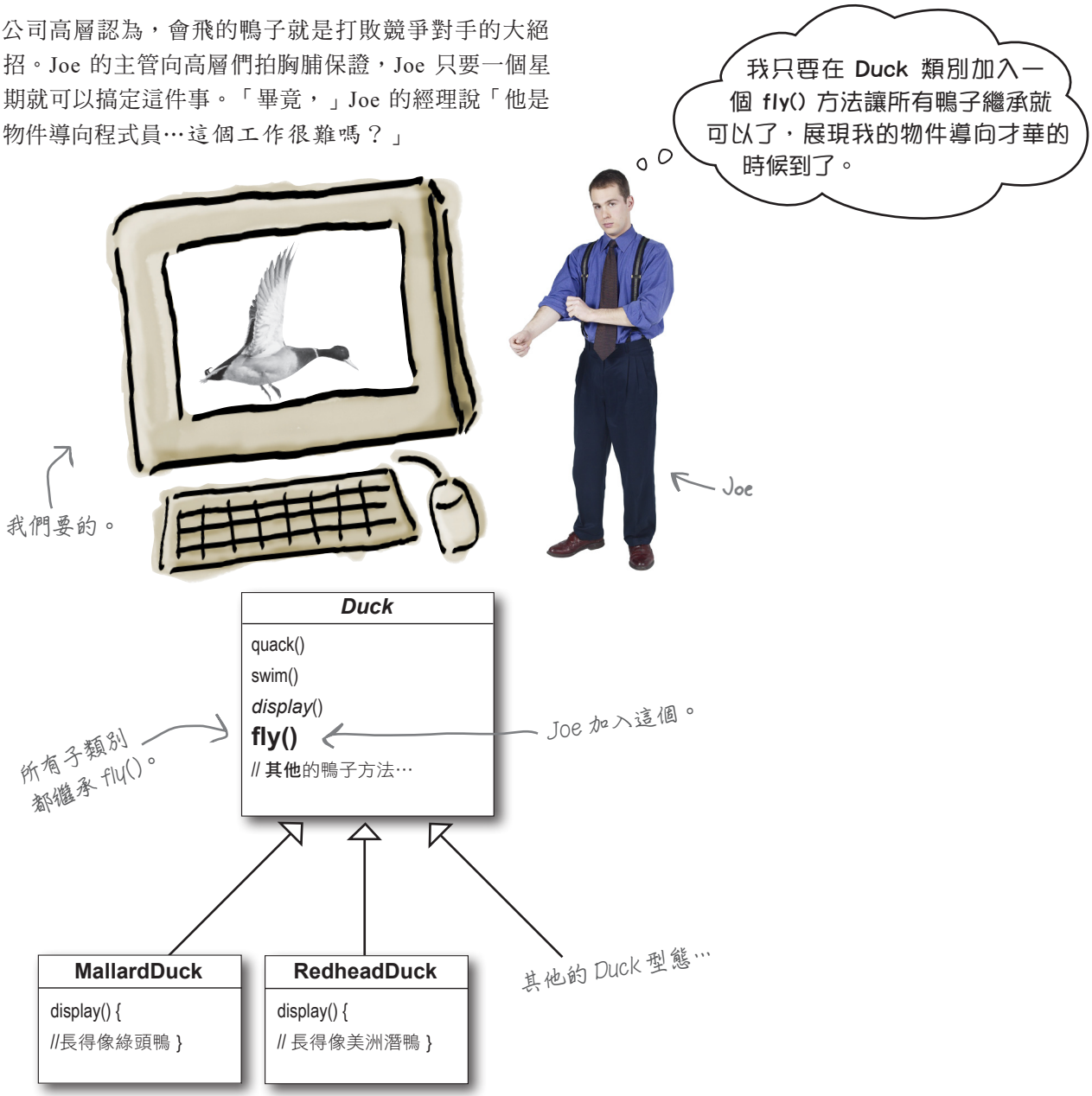


但是公司在去年遇到強大的競爭對手，在為期一週的高爾夫假期兼腦力激盪會議之後，公司高層認為該創新了，他們打算在下週的茂宜島股東大會上，展示一些真的令人印象深刻的東西。

碁峰 www.gotop.com.tw

現在我們要讓鴨子飛起來

公司高層認為，會飛的鴨子就是打敗競爭對手的大絕招。Joe 的主管向高層們拍胸脯保證，Joe 只要一個星期就可以搞定這件事。「畢竟，」Joe 的經理說「他是物件導向程式員…這個工作很難嗎？」



www.gotop.com.tw

但是，可怕的問題出現了…

Joe，我在股東大會，有一隻橡皮鴨在展示畫面上飛來飛去，你在搞笑嗎？

為什麼會這樣？

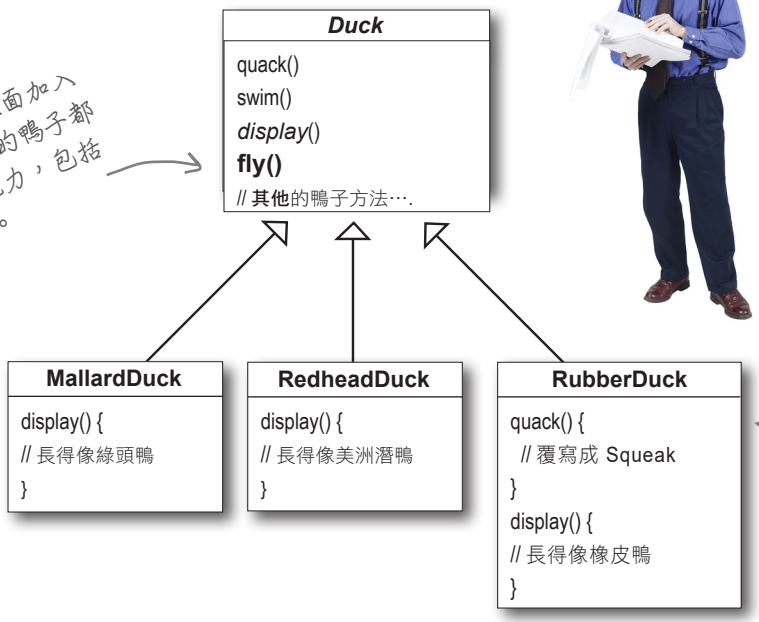
Joe 忽略一件事：並非 Duck 的子類別都會飛。Joe 在 Duck 超類別加入新行為也會幫不適合該行為的 Duck 子類別加入那個行為，現在可好，在 SimUDuck 程式裡面，有一個會飛的無生命物件。

局部修改程式碼，卻造成非局部性的影響（會飛的橡皮鴨）！

好吧，看來我的設計有一個小缺失，為什麼那些人不把它視為一種「功能」就好了？其實它蠻可愛的啊…

Joe 認為善用繼承可以重複使用程式碼，但是這種做法卻造成維護方面的問題。

他在超類別裡面加入 fly() 讓所有的鴨子都有飛行的能力，包括不能飛的。



還有，橡皮鴨不會發出鴨叫聲，所以 quack() 被覆寫成「Squeak（唧唧叫）」。

碁峰 www.gotop.com.tw

Joe 想到繼承…

RubberDuck

```
quack()  { // squeak}
display() { // 橡皮鴨 }
fly() {
  // 覆寫成不做任何事情
}
```

這是在繼承階層裡面的另一個類別，注意，它與 RubberDuck 一樣不會飛，但是它也不會叫。

DecoyDuck

```
quack() {
  // 覆寫成不做任何事情
}

display() { // 誘餌鴨}

fly() {
  // 覆寫成不做任何事情
}
```

削尖你的鉛筆

用繼承來提供 Duck 的行為有哪些缺點？（多選題）

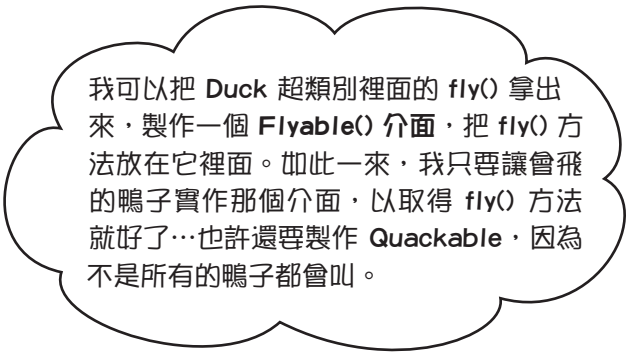
- ❑ A. 在子類別裡面有重複的程式碼。
- ❑ B. 在執行期很難改變行為。
- ❑ C. 無法讓鴨子跳舞。
- ❑ D. 很難知道所有鴨子行為。
- ❑ E. 無法讓鴨子一邊飛，一邊鳴叫。
- ❑ F. 修改一個地方可能會不小心影響其他的鴨子。

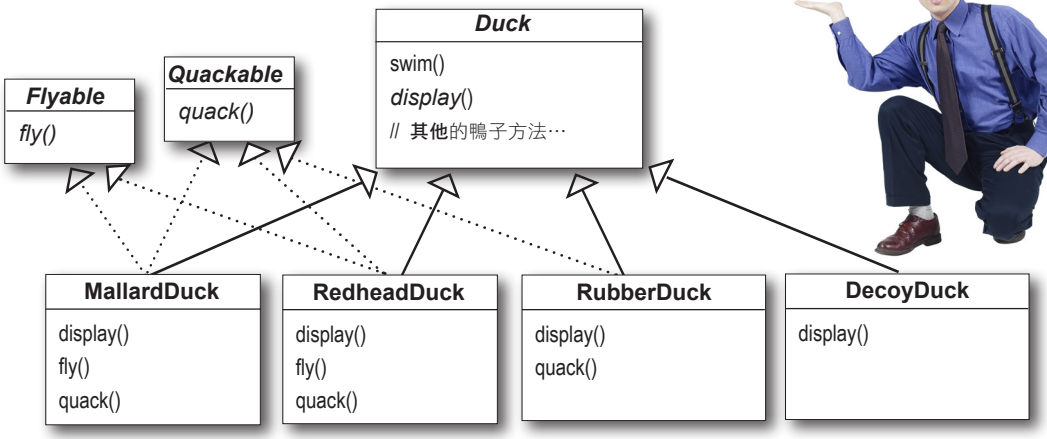
碁峰 www.gotop.com.tw

那麼，使用介面如何？

Joe 意識到繼承應該不是解決問題的辦法，因為他收到一張備忘錄，上面說高層決定每六個月就更新一次產品（但是怎麼更新還不知道）。Joe 知道規格會經常改變，所以，每當有新的 Duck 子類別加入時，他就要檢查 fly() 與 quack()，並且視情況覆寫它們…這簡直是無止盡的惡夢。

所以，他要用更清楚的方法，只讓一些（不是全部）鴨子類型可以飛或叫。





你覺得這個設計怎麼樣？

碁峰 www.gotop.com.tw

這簡直是最愚蠢的想法，**可不可以跟著我講一遍「重複的程式碼」**？你認為覆寫幾個方法不是好事，那如果你想要稍微修改「飛」這個行為呢…難道你想要修改全部的 48 個會飛的 Duck 子類別？！

如果你是 Joe，你會怎麼做？

我們知道，並非所有的子類別都必須有飛或叫的行為，所以繼承不是正確的答案。雖然讓子類別實作 Flyable 或 Quackable 可以解決部分的問題（會飛的橡皮鴨不會出現了），但是它會造成這些行為的程式碼無法重複使用，這只是從一個維護惡夢跳入另一個維護惡夢。而且，會飛的鴨子絕對不會只有一種飛行行為…

此時，你應該已經開始期待設計模式會像超人一樣現身，把你從水深火熱中救出來，但是，直接讓你知道答案有什麼樂趣？我們接下來要用一種老方法來找出解決方案—藉著運用優良的物件導向軟體設計原則。

要是有一種技巧可以讓我在修改軟體時，對既有的程式碼造成最小的影響，那該有多好？如此一來，我就不用花那麼多時間重寫程式了，我可以用更多時間來讓程式做更酷的事情…

碁峰 www.gotop.com.tw

軟體開發的不變真理

OK ！在軟體開發裡面，哪件事一定是對的？

無論你在哪裡工作、你在建構什麼東西、你使用哪種程式語言，與你永遠形影不離的不變真理是什麼？

變化

（用鏡子來看答案）

無論你的應用程式設計得多好，一段時間之後，它一定需要成長並改變，否則它就會死亡。

削尖你的鉛筆

驅動改變的因素很多，有哪些因素曾經讓你改變應用程式碼？把它寫下來（我們列出一些我們的原因，幫你起個頭）。先比對你的答案與本章結尾的答案，再翻到下一頁。

我們的顧客或用戶要求別的東西，或是想要新功能。

公司決定採用另一種資料庫產品，並且向另一家供應商購買資料，但是那些資料使用不同的格式，唉！

碁峰 www.gotop.com.tw

把問題歸零…

我們已經知道繼承的效果不太好了，因為在子類別裡面的鴨子行為會不斷改變，而且讓所有子類別都擁有那些行為並不洽當。使用 Flyable 與 Quackable 介面乍看之下還不賴（需要飛的鴨子才需要使用 Flyable），但是 Java 的介面裡面通常沒有實作碼，所以無法重複使用程式碼。當你需要修改某種行為時，無論採取哪種做法，通常你都要在定義該行為的所有子類別裡面檢查並修改那個行為，況且，一不小心還會寫出新的 bug ！

幸運的是，有一種設計原則可以處理這種情況。

設計原則

找出應用程式中會變的部分，把它們和不會變的部分隔開。

這只是諸多設計原則裡面的一條，這本書會用更多時間來討論它們。

換句話說，當你的程式碼有某些部分會變（例如每當有新的需求出現時），你就有一個需要拉出來並且和其他不變的部分分開的行為了。

你可以用另一種方式來看待這條原則：**把會變的部分封裝起來，如此一來，你就可以修改或擴展會變的部分，同時不會影響不變的部分**。

這個簡單的概念幾乎是所有設計模式的基礎。所有模式都提供一種方式來讓系統的某個部分可以改變，卻又不影響任何其他的部分。

好，是時候把鴨子行為從 Duck 類別裡面拉出來了！

把會變的部分「封裝」起來，讓它不會影響其餘的部分。

這有什麼效果？它可以減少修改程式造成的意外後果，也可以讓系統更靈活！

www.gotop.com.tw

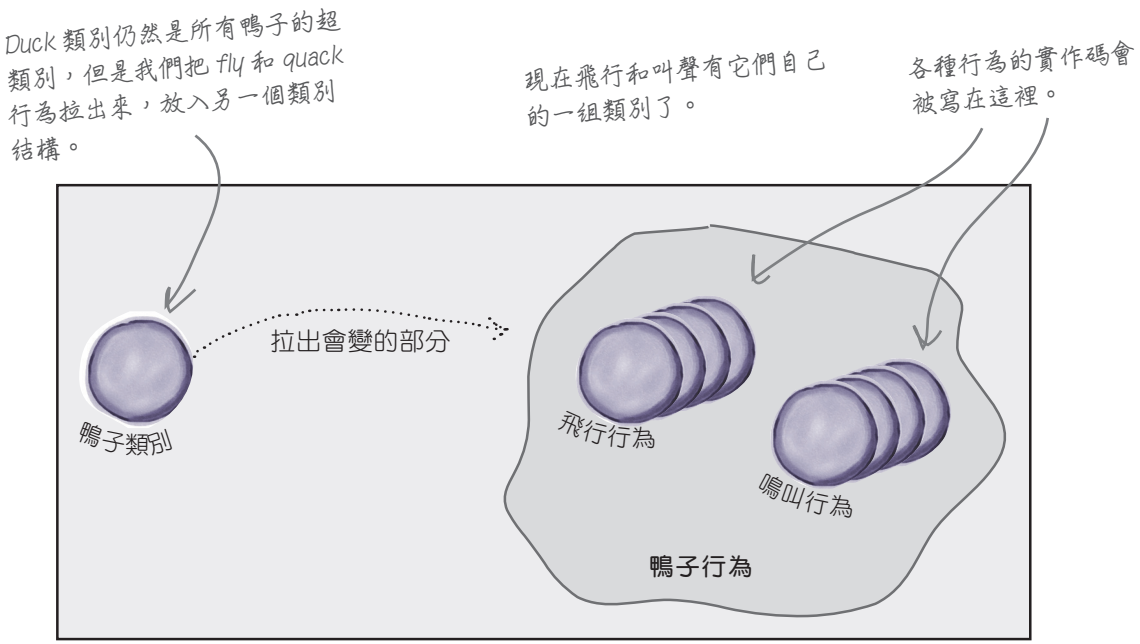
把會變的部分和不變的部分分開

我們該從何處下手？我們知道，除了 fly() 與 quack() 的問題之外，Duck 類別還算正常，其他的地方看起來都不需要經常改變或修改。所以，除了一些小改變之外，我們不打算對 Duck 類別做太多的處理。

為了分開「會變的部分與不變的部分」，我們要建立兩組類別（完全和 Duck 分開），一個是用來飛的，一個是用來鳴叫的。每一組類別都會實作各自的行為。舉例來說，我們可能在一個類別中實作叫聲，在另一個類別中實作啾啾叫，在第三個類別中實作不出聲。

我們知道，在 Duck 類別裡面的 fly() 與 quack() 會隨著鴨子的不同而改變。

為了把這兩個行為與 Duck 類別分開，我們把它們抽出 Duck 類別，並建立一組類別來代表那些行為。



碁峰 www.gotop.com.tw

設計鴨子的行為

如何設計一組類別，並且用它們來實作飛行和鳴叫行為？

我們想要保持彈性，畢竟，當初讓我們陷入困境的禍首，就是不靈活的鴨子行為。我們想要將行為指派給 Duck 的實例，例如，也許我們可以實例化一個新的 MallardDuck 實例，並且用特定的飛行行為型態來將它初始化。既然如此，何不順便讓我們可以動態地改變鴨子的行為？也就是在 Duck 類別裡面加入行為設定方法，如此一來，我們就可以在執行期改變 MallardDuck 的飛行行為了。

設定這些目標之後，我們來看一下第二條設計原則：

設計原則

針對介面寫程式，而不是針對實作寫程式。

我們將使用介面來代表各種行為，例如 FlyBehavior 與 QuackBehavior，而且實作每一種行為的程式都必須實作其中的一個介面。

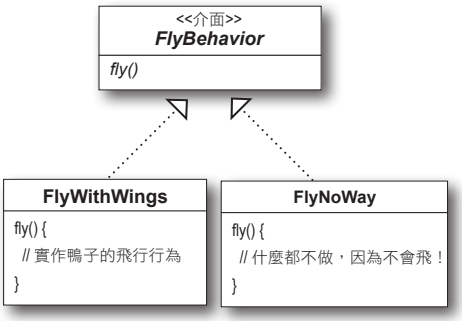
所以，現在不是讓 *Duck* 類別實作飛行和鳴叫介面了，我們用一組專門的類別來代表行為（例如「鳴叫」），並且讓行為類別實作行為介面，而不是讓 Duck 類別實作它們。

這種做法與之前不一樣，之前的行為來自超類別 Duck 裡面的具體實作，或是在子類別裡面量身打造的實作。無論如何，它們都依靠一個實作，那個特定的實作讓我們動彈不得，沒有改變行為的餘地（除非寫更多程式）。

在新設計中，Duck 子類別會使用介面所代表的行為（FlyBehavior 與 QuackBehavior），所以行為的實作（也就是為了實作 FlyBehavior 或 QuackBehavior 而在類別裡面編寫的具體行為）不會被綁死在 Duck 的子類別裡面。

從此之後，Duck 的行為會被放在分開的類別裡面，該類別實作了特定的行為介面。

如此一來，Duck 類別就不需要知道自己的行為的實作細節了。



www.gotop.com.tw

「針對介面寫程式」的意思其實是「針對超型態（supertype）寫程式」。

介面這個字有很多種用法，它可以代表介面這個概念，也可以代表 Java 的介面結構。不使用 Java 的介面也可以針對介面寫程式。重點在於藉著「針對超型態寫程式」來利用多型，如此一來，執行期的物件就不會被鎖死在程式碼裡面了。我們可以將「針對超型態寫程式」這句話改成「用來宣告變數的型態應該是超型態」（超型態通常是抽象類別或介面），如此一來，那一個變數就可以設成那一個超型態的任何一個具體物件，這意味著，宣告該變數的類別不需要知道實際的物件型態是什麼！」

你應該早就知道這些事情了，重述它們只是為了確保我們說的是同一件事，舉一個使用多型型態的例子—假如有一個抽象類別 Animal，它有兩個具體實作，Dog 與 Cat。

針對實作寫程式是：

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

將變數「d」宣告成 Dog 型態（Animal 的具體實作）讓我們針對具體實作寫程式。

針對介面 / 超型態寫程式則是：

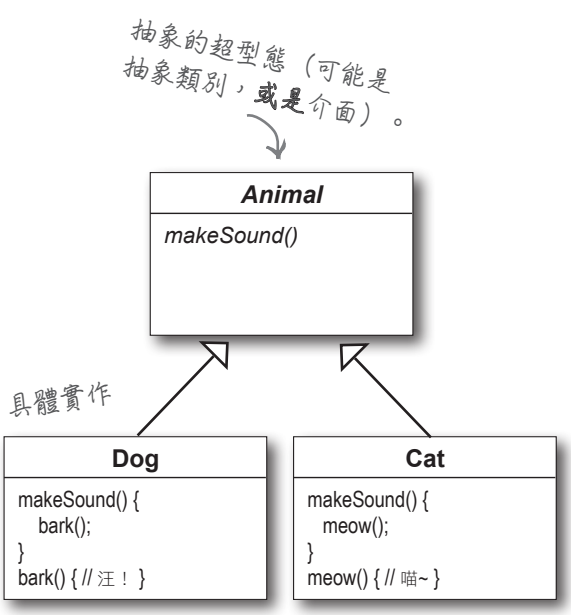
```
Animal animal = new Dog();
animal.makeSound();
```

我們知道它是 Dog，但是現在我們使用多型的 animal 參考。

更好的寫法是**在執行期指派具體的實作物件**，而不是用固定的程式來實例化子型態（例如 new Dog()）：

```
a = getAnimal();
a.makeSound();
```

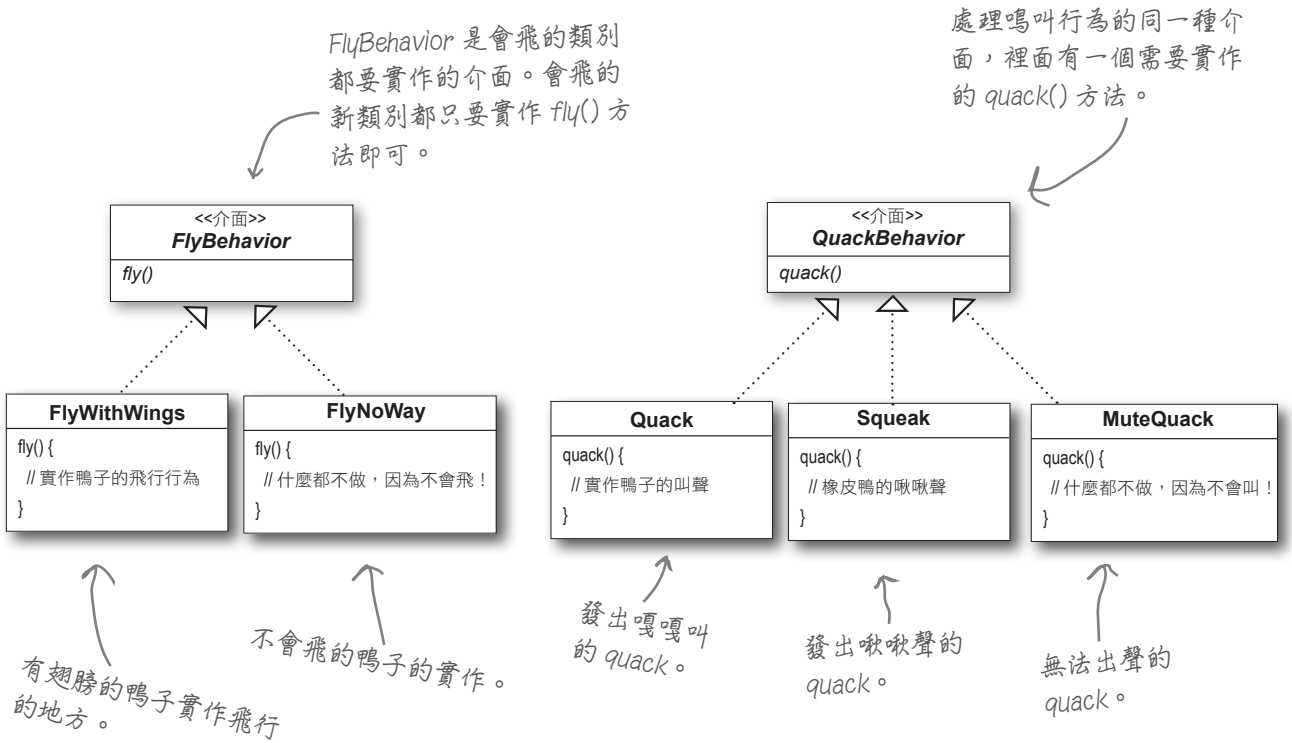
我們不知道 animal 子型態到底是什麼…只知道它曉得如何回應 makeSound()。



碁峰 www.gotop.com.tw

實作 Duck 行為

現在有兩個介面：FlyBehavior 與 QuackBehavior，以及實作具體行為的類別：



使用這種設計可讓其他的物件型態重複使用飛行和鳴叫行為，因為這些行為沒有被埋在 Duck 類別裡面了！

我們也可以加入新行為，而且不需要修改既有的行為類別，或修改使用飛行行為的 Duck 類別。

如此一來，我們不僅獲得重複使用的好處，也免除繼承帶來的所有負擔。

www.gotop.com.tw

問：我一定要先寫好應用程式，看看哪些地方會改變，再回過頭來隔離並封裝會變的地方嗎？

答：不一定，在設計應用程式時，通常你可以預料哪些部分會改變，並提前加入處理它的彈性機制。你將發現，設計原則和模式可以在開發週期的任何階段運用。

問：我們是不是也要把 Duck 做成介面？

答：在這個例子不需要。把所有部分組合起來之後，你會看到，不把 Duck 做成介面，並且讓具體的鴨子（例如 MallardDuck）繼承共同的屬性和方法是有好處的。我們已經把 Duck 的繼承機制之中會變的部分移開，並且獲得這個結構帶來的好處了，所以不需將 Duck 設計成介面。

問：只有一個行為的類別有點奇怪，類別不是用來表示某一種「東西」的嗎？它應該擁有狀態與行為，不是嗎？

答：在物件導向系統中，你說得對，類別通常代表兼具狀態（實例變數）與方法的東西，在這個例子裡，那個東西碰巧是一種行為。但是，行為同樣有狀態與方法，飛行行為可能用實例變數來代表飛行行為的屬性（每分鐘揮動幾下翅膀、最大飛行高度、速度…等）。

削尖你的鉛筆

❶ 如何使用新設計，在 SimUDuck app 裡面加入火箭動力的飛行行為？

❷ 除了鴨子之外，你能不能想到其他使用 Quack 行為的類別？

答案：

1) 建立 FlyRocketPowered 類別，讓它實作 FlyBehavior 介面。

2) 例如鴨鳴器（一種可以發出鴨叫聲的器具）。

碁峰 www.gotop.com.tw

整合 Duck 的行為

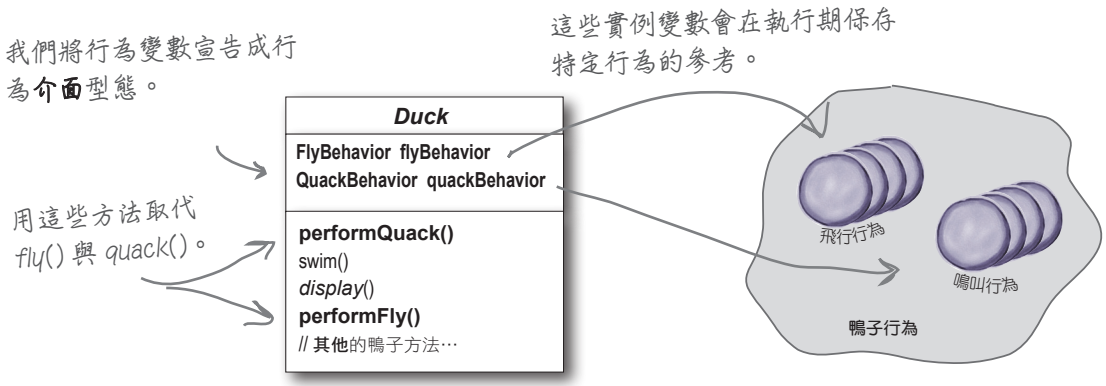
重點整理：現在 Duck 將它的飛行和鳴叫的行為委託（*delegate*）出去，而不是在 Duck 類別（或子類別）裡面定義鳴叫或飛行的方法。

做法是：

❶ **加入型態為 FlyBehavior 與 QuackBehavior 的實例變數**—我們稱之為 flyBehavior 與 quackBehavior。在執行期，每個具體的鴨子物件都會將特定的行為指派給那些變數。

我們也將 Duck 類別的（和它的任何子類別的）fly() 與 quack() 方法移除，因為這些行為已經被搬到 FlyBehavior 與 QuackBehavior 類別裡面了。

我們把 Duck 類別裡面的 fly() 與 quack() 換成類似的方法，稱為 performFly() 與 performQuack()，接下來你會看到它們是如何運作的。



❷ **實作 performQuack()：**

```
public abstract class Duck {
   QuackBehavior quackBehavior;
   // 其他程式碼

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

每一個 Duck 都保存一個參考，指向實作了 QuackBehavior 介面的東西。

Duck 物件不是自己鳴叫，而是將這個行為委託給 quackBehavior 所參考的物件。

很簡單吧？當 Duck 需要鳴叫時，只要請 quackBehavior 參考的物件幫它鳴叫就即可。在這部分的程式裡，我們不在乎具體 Duck 是哪一種物件，**只在乎它知道怎麼 *quack()*！**

www.gotop.com.tw

更多的整合…

❸ OK，是時候關心**如何設定** flyBehavior 與 quackBehavior 實例變數了。看一下 MallardDuck 類別：

```
public class MallardDuck extends Duck {

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

   public void display() {
       System.out.println("I'm a real Mallard duck");
   }
}
```

MallardDuck 使用 Quack 類別來處理它的叫聲，所以當你呼叫 performQuack() 時，鳴叫的工作會被委託給 Quack 物件，讓你得到真實的叫聲。

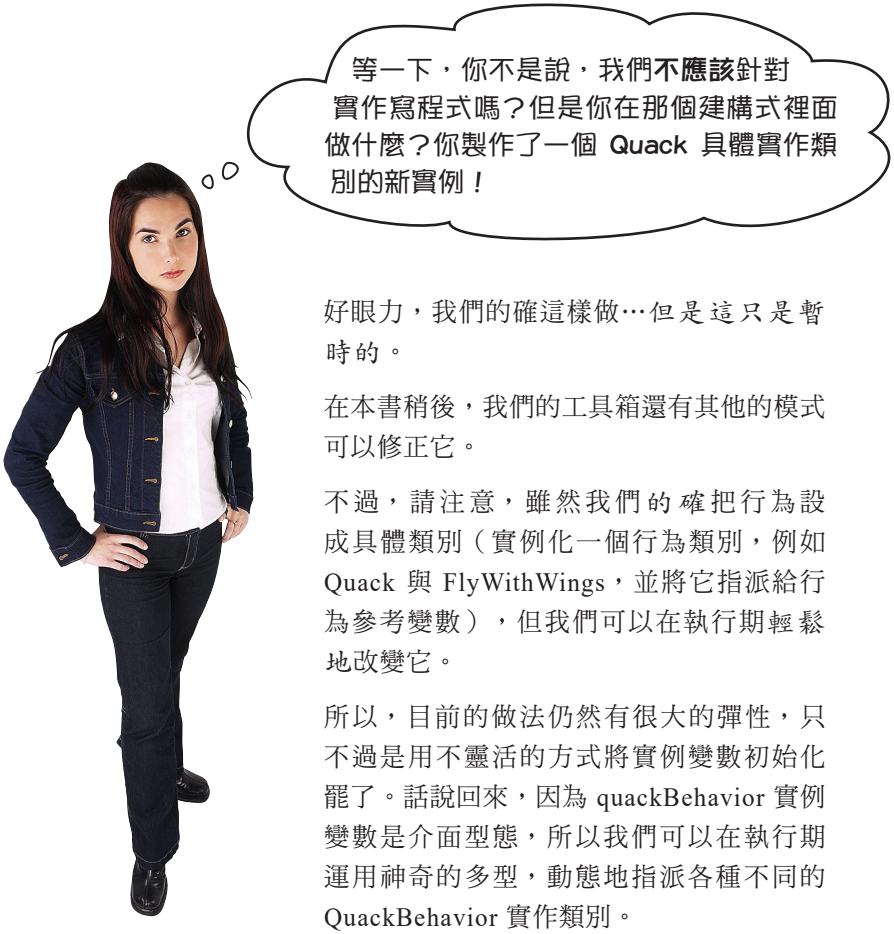
別忘了，MallardDuck 的 quackBehavior 與 flyBehavior 實例變數是從 Duck 類別繼承來的。

它也將 FlyWithWings 當成 FlyBehavior 型態。

MallardDuck 的 quack 是貨真價實的鴨子**嘎嘎聲（quack）**，不是**啾啾聲（squeak）**，也不是**默不出聲（mute quack）**。MallardDuck 被實例化時，它的建構式會將 quackBehavior 實例變數設成 Quack 型態的新實例，Quack 是 QuackBehavior 的具體實作類別。

鴨子的飛行行為也一樣—MallardDuck 的建構式會將繼承來的 flyBehavior 實例變數設成 FlyWithWings 型態的實例，FlyWithWings 是 FlyBehavior 的具體實作類別。

碁峰 www.gotop.com.tw



好眼力，我們的確這樣做…但是這只是暫時的。

在本書稍後，我們的工具箱還有其他的模式可以修正它。

不過，請注意，雖然我們的確把行為設成具體類別（實例化一個行為類別，例如 Quack 與 FlyWithWings，並將它指派給行為參考變數），但我們可以在執行期輕鬆地改變它。

所以，目前的做法仍然有很大的彈性，只不過是用不靈活的方式將實例變數初始化罷了。話說回來，因為 quackBehavior 實例變數是介面型態，所以我們可以在執行期運用神奇的多型，動態地指派各種不同的 QuackBehavior 實作類別。

花一點時間想一下如何寫出可以在執行期改變行為的鴨子（幾頁之後，你就會看到做這件事的程式了）。

www.gotop.com.tw

測試 Duck 程式碼

❶ 輸入下面的 Duck 類別（Duck.java）還有兩頁之後的 MallardDuck 類別（MallardDuck.java），並編譯它們。

```
public abstract class Duck {

   FlyBehavior flyBehavior;
   QuackBehavior quackBehavior;
   public Duck() { }

   public abstract void display();

   public void performFly() {
      flyBehavior.fly();
   }

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

   public void swim() {
      System.out.println("All ducks float, even decoys!");
   }
}
```

宣告兩個行為介面型態的參考變數。在同一個程式包裡面的鴨子子類別都會繼承它們。

委託給行為類別。

❷ 輸入 FlyBehavior 介面（FlyBehavior.java）與兩個行為實作類別（FlyWithWings.java 與 FlyNoWay.java），並編譯它們。

```
public interface FlyBehavior {
   public void fly();
}
```

讓所有飛行行為類別實作的介面。

```
public class FlyWithWings implements FlyBehavior {
   public void fly() {
       System.out.println("I'm flying!!");
   }
}
```

飛行行為的實作，這是讓**會飛的**鴨子使用的…

```
public class FlyNoWay implements FlyBehavior {
   public void fly() {
       System.out.println("I can't fly");
   }
}
```

飛行行為的實作，這是讓**不會飛的**鴨子使用的（例如橡皮鴨與誘餌鴨）。

碁峰 www.gotop.com.tw

測試 Duck 的程式碼，續…

❸ 輸入 QuackBehavior 介面（QuackBehavior.java）與三個行為實作類別（Quack.java、MuteQuack.java 與 Squeak.java），並編譯它們。

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

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

```
public class MuteQuack implements QuackBehavior {
   public void quack() {
       System.out.println("<< Silence >>");
   }
}
```

```
public class Squeak implements QuackBehavior {
   public void quack() {
       System.out.println("Squeak");
   }
}
```

❹ 輸入測試類別（MiniDuckSimulator.java）並編譯它。

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

它會呼叫 MallardDuck 繼承來的 performQuack() 方法，那個方法會將工作委託給物件的 QuackBehavior（呼叫鴨子繼承來的 quackBehavior 所參考的 quack()）。

我們也讓 MallardDuck 繼承來的 performFly() 方法做同樣的事情。

❺ 執行程式！

```
File Edit Window Help Yadayadayada
%java MiniDuckSimulator
Quack
I'm flying!!
```


www.gotop.com.tw

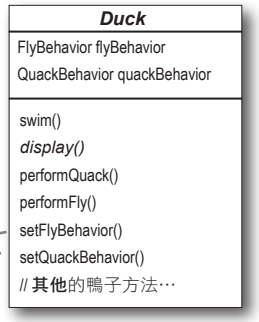
動態地設定行為

為鴨子設計這些動態機制卻不使用它真的很可惜！假設我們想用 Duck 類別的 set 方法來設定鴨子的行為，而不是在鴨子的建構式裡面將它實例化。

❶ 在 Duck 類別裡面加入兩個新方法：

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

public void setQuackBehavior(QuackBehavior qb) {
    quackBehavior = qb;
}
```



接下來，我們就可以隨時呼叫這些方法，動態（on the fly）地改變鴨子的行為了。

編注：這是多此一舉的雙關語，改一下

❷ 製作新的 Duck 型態（ModelDuck.java）。

```
public class ModelDuck extends Duck {
    public ModelDuck() {
        flyBehavior = new FlyNoWay();
        quackBehavior = new Quack();
    }

    public void display() {
        System.out.println("I'm a model duck");
    }
}
```

我們的模型鴨在一開始是陸棲的…它不會飛。

❸ 製作新的 FlyBehavior 型態（FlyRocketPowered.java）。

沒關係的，我們建立以火箭驅動的飛行行為。

```
public class FlyRocketPowered implements FlyBehavior {
    public void fly() {
        System.out.println("I'm flying with a rocket!");
    }
}
```

碁峰 www.gotop.com.tw

❹ 修改測試類別（MiniDuckSimulator.java），加入 ModelDuck，並且幫 ModelDuck 裝上火箭。

```
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());
        model.performFly();
    }
}
```

之前

第一次呼叫 performFly() 時，我們將工作委託給在 ModelDuck 建構式裡設定的 flyBehavior 物件，它是個 FlyNoWay 實例。

這會呼叫 model 繼承來的行為 setter（設定方法），然後…哇！model 突然可以用火箭來飛行了！

成功的話，模型鴨可以動態地改變飛行行為！將實作放在 Duck 類別裡面的話，就不能這樣做了。

之後

❺ 執行程式！

```
File Edit Window Help Yabbadabbadoo
%java MiniDuckSimulator
Quack
I'm flying!!
I can't fly
I'm flying with a rocket!
```

要在執行期改變鴨子的行為，你只要呼叫鴨子的行為的 setter 即可。

碁峰 www.gotop.com.tw