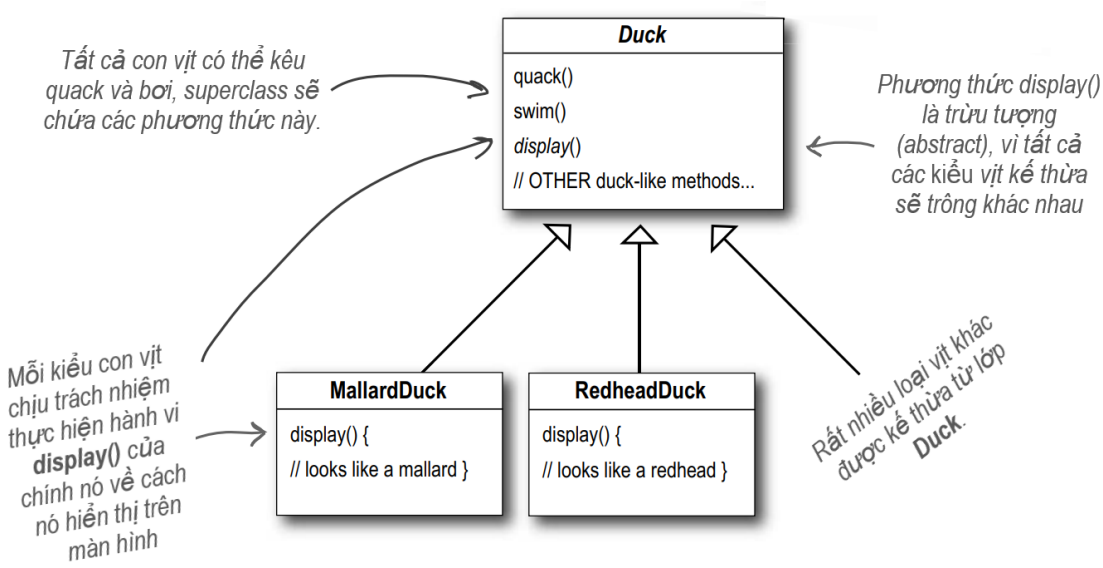
Chương 1: Strategy Pattern (Mẫu chiến lược)

Strategy Pattern – Chào mừng đến với Design Patterns

Ai đó đã giải quyết những vấn đề của bạn. Trong chương này **Strategy Pattern (Mẫu chiến lược)**, bạn sẽ học được lý do tại sao (và làm thế nào) bạn có thể khai thác kiến thức và kinh nghiệm của những developer khác. Trước khi chúng ta hoàn thành, chúng ta sẽ xem xét việc sử dụng và lợi ích của các mẫu thiết kế, xem xét một số nguyên tắc thiết kế OO chính và xem qua một ví dụ về cách một mẫu hoạt động. Cách tốt nhất để sử dụng các mẫu là suy nghĩ về chúng và sau đó nhận ra nơi nào cần sử dụng trong thiết kế của bạn và các ứng dụng hiện có, nơi bạn có thể áp dụng chúng. Thay vì tự viết code, với các mẫu thiết kế bạn có được kinh nghiệm sử dụng lại code (reuse code).

Bắt đầu với một ứng dụng SimUDuck đơn giản

Joe làm việc cho một công ty rất thành công trò chơi mô phỏng vịt, **SimUDuck**. Trò chơi có thể hiển thị rất nhiều loài vịt bơi và tạo ra âm thanh "quack quack". Thiết kế ban đầu của hệ thống đã sử dụng các kỹ thuật OO tiêu chuẩn và tạo ra một superclass Duck mà tất cả các loại vịt khác sẽ kế thừa nó.

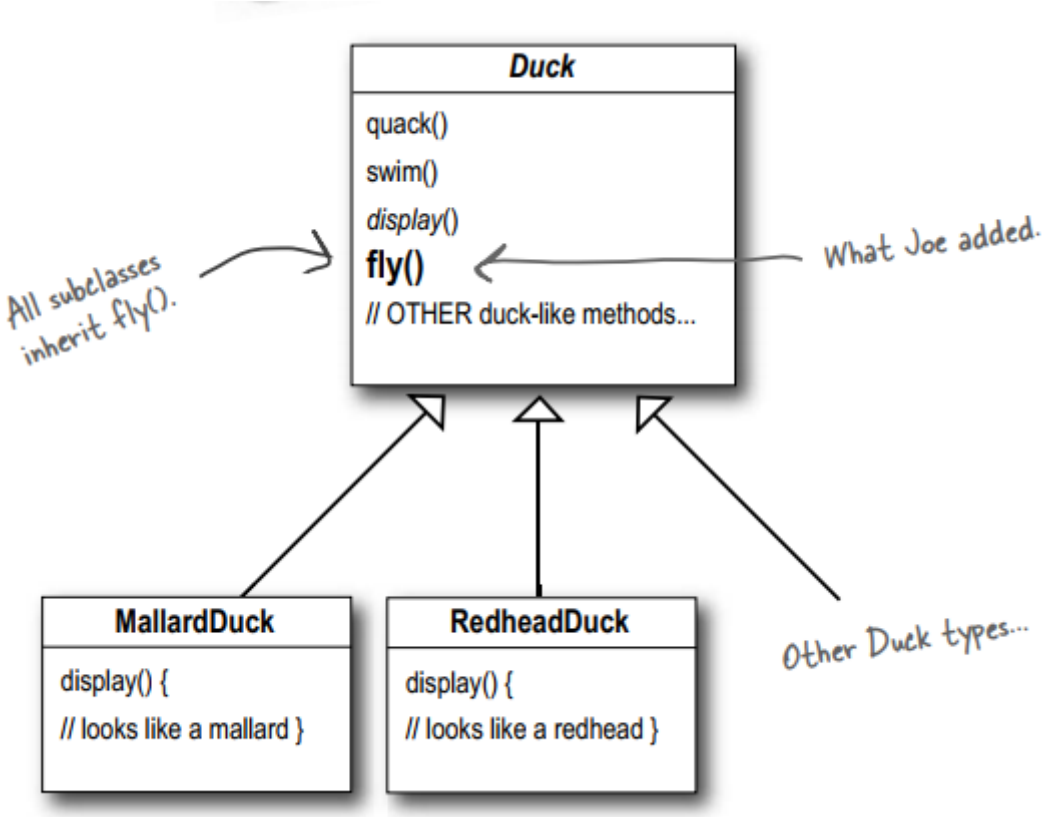


Trong năm ngoái, công ty đã chịu áp lực ngày càng tăng từ các đối thủ cạnh tranh. Sau một tuần "động não" ngoài sân golf, các giám đốc điều hành của công ty nghĩ rằng đây là thời gian cho một sự thay đổi lớn. Công ty cần một cái gì đó thực sự ấn tượng để trình bày tại cuộc họp cổ đông sắp tới ở Maui.

Nhưng bây giờ chúng tôi cần những con vịt BAY

Giám đốc điều hành đã quyết định rằng vịt biết bay là thứ mà trình giả lập cần để thổi bay các đối thủ cạnh tranh khác. Và tất nhiên, người quản lý của Joe, nói với anh rằng nó sẽ không có vấn đề gì nên chỉ cần làm gì đó trong một tuần. "Sau tất cả", ông chủ của Joe nói, "Khi anh ấy là một lập trình viên OO … nó sẽ khó đến mức nào?"

*"Tôi chỉ cần thêm một phương thức **fly()** trong lớp **Duck** và sau đó tất cả các con vịt sẽ thừa hưởng nó. Bây giờ, thời gian của tôi để thực sự thể hiện tài năng OO thực sự của tôi."*



Nhưng một cái gì đó đã sai khủng khiếp ...

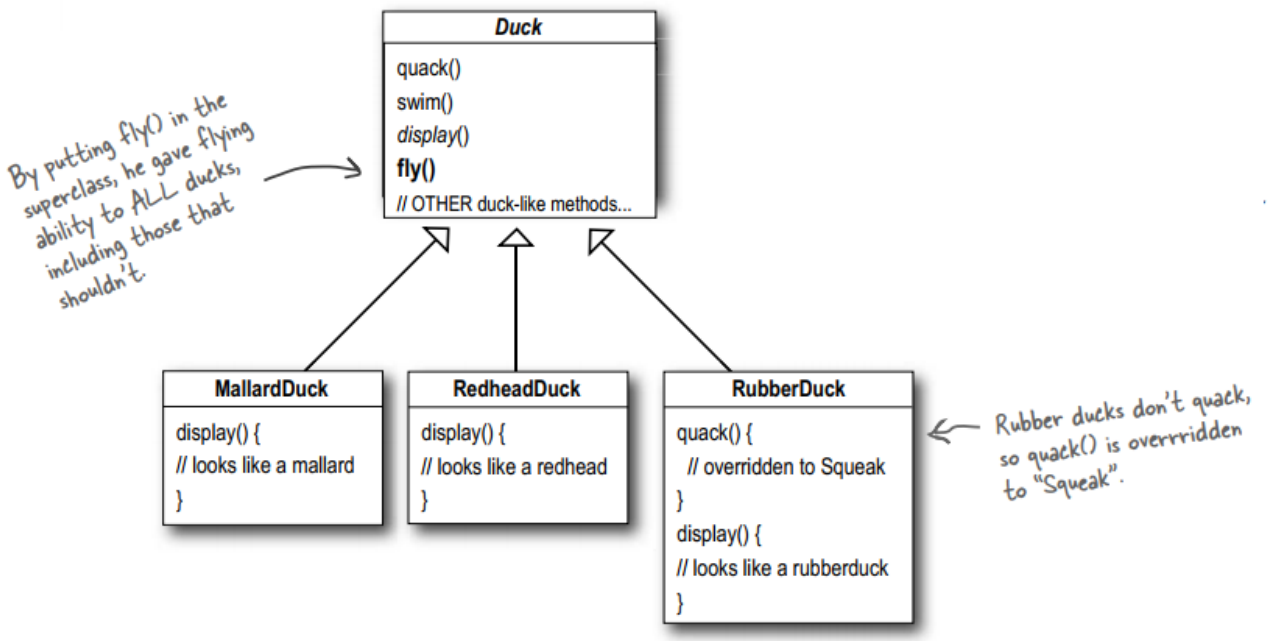
"Joe, tôi đang ở cuộc họp cổ đông. Họ đưa ra một bản demo và có những chú vịt cao su (rubber ducks) đang chạy quanh màn hình. Đây có phải ý tưởng của bạn về một trò đùa? Bạn có thể muốn dành một chút thời gian cho trang Monster.com..."

CHUYỆN GÌ ĐÃ XẢY RA?

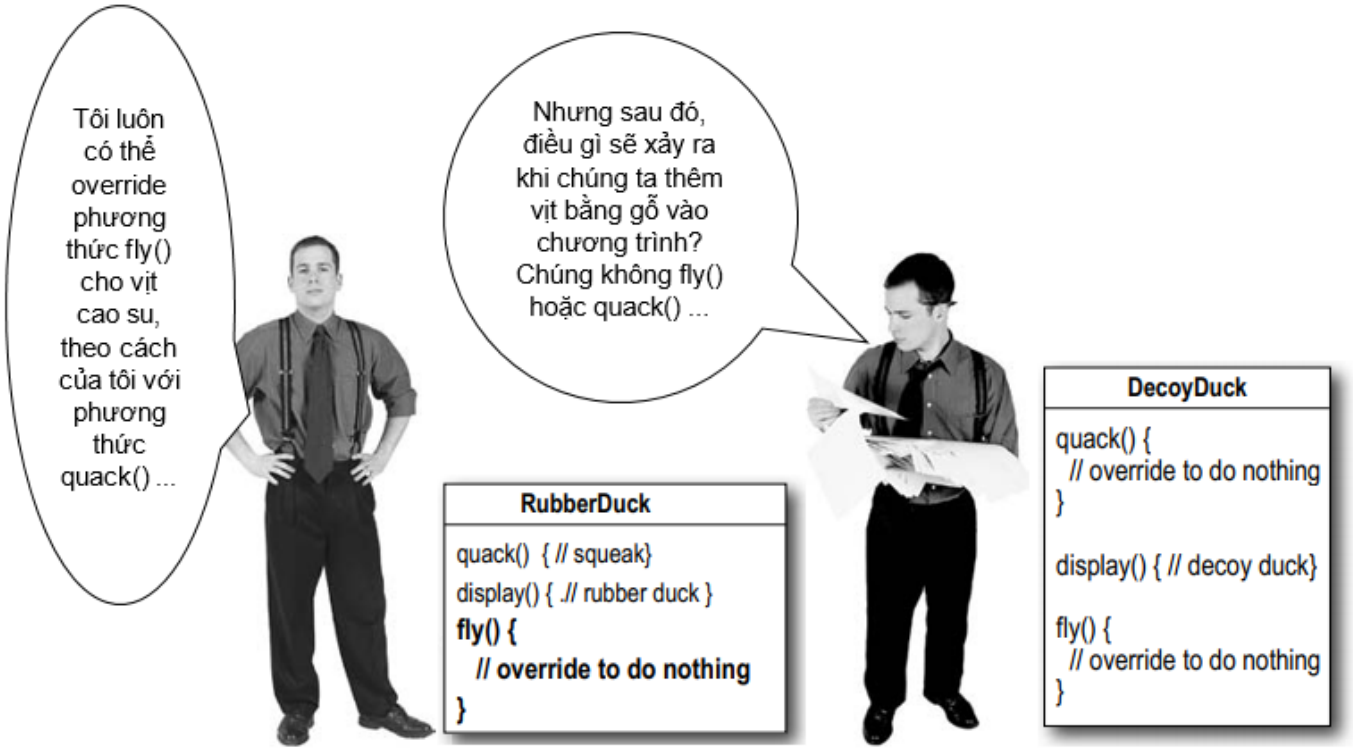
Joe không biết rằng không phải tất cả các loại vịt đều phải BAY. Khi Joe thêm hành vi mới vào lớp Duck, anh ta cũng đang thêm hành vi không phù hợp với một số lớp con của Duck. Bây giờ Joe có các "thứ vô tri vô giác" trong chương trình SimUDuck. Một bản cập nhật gây ra hành vi phụ không đúng (vịt cao su bay)!

Joe: *"OK, có một lỗ hổng nhỏ trong thiết kế của tôi. Tôi không thể thấy được lý do tại sao họ có thể gọi đó là "feature". Nó rất dễ thương ..."*

Joe đã nghĩ việc sử dụng thừa kế là điều tuyệt vời cho mục đích tái sử dụng code, nhưng đã không thành công lắm khi nói đến bảo trì.



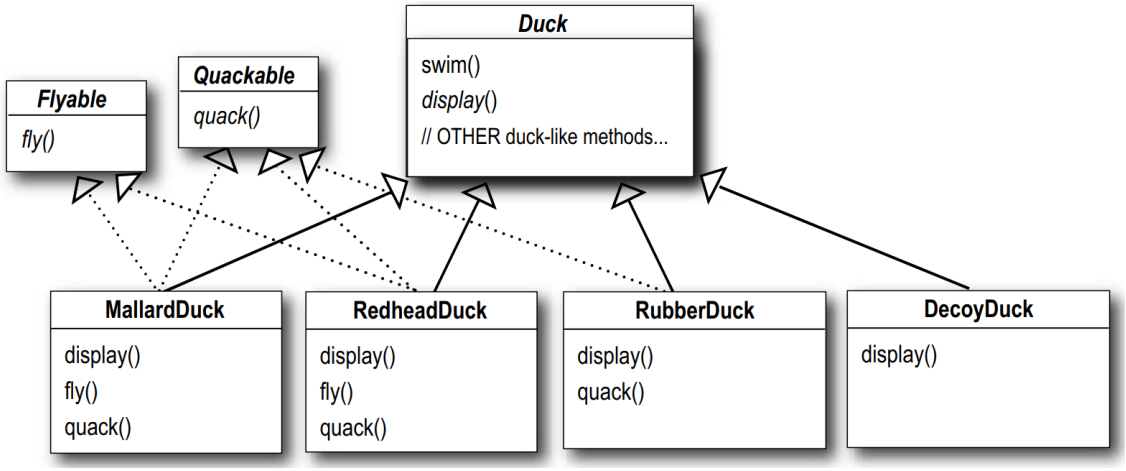
Joe nghĩ về kế thừa...



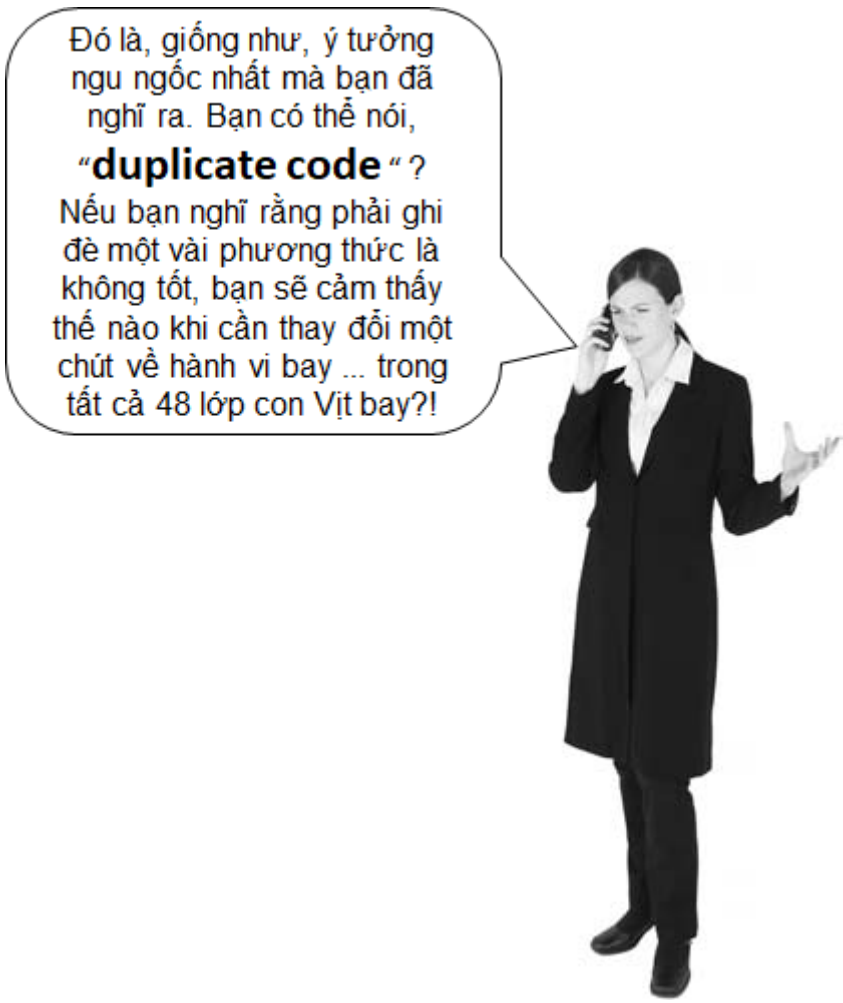
How about an interface?

Joe nhận ra rằng sự kế thừa có lẽ không phải là câu trả lời đúng, bởi vì anh ta vừa nhận được một bản ghi chú nói rằng giám đốc điều hành hiện muốn cập nhật sản phẩm 6 tháng một lần (theo cách mà họ sẽ quyết định). Joe biết rằng yêu cầu sẽ tiếp tục thay đổi và anh ấy sẽ bị buộc phải xem xét và có thể override fly() và quack() cho mỗi lớp Vịt mới sẽ thêm vào chương trình … *mãi mãi*.

Vì vậy, anh ta cần tìm một cách thông minh hơn để chỉ có một vài (không phải tất cả) loại vịt có thể bay hay kêu quack quack.



Bạn nghĩ gì về thiết kế trên?



Bạn sẽ làm gì nếu bạn là Joe?

Chúng tôi biết rằng không phải tất cả các lớp con nên có hành vi Fly() hoặc quack(), do đó, kế thừa không phải là câu trả lời đúng. Nhưng nếu các lớp con implement Flyable và/hoặc Quackable thì sẽ giải quyết được một phần của vấn đề (không có vịt cao su bay), nhưng nó sẽ phá hủy hoàn toàn việc sử dụng lại code cho các hành vi đó, vì vậy nó chỉ tạo ra một cơn ác mộng bảo trì khác. Và tất nhiên, có thể có nhiều kiểu bay ngay cả trong số những con vịt bay …

Tại thời điểm này, bạn có thể chờ đợi một Mẫu thiết kế giúp bạn tiết kiệm thời gian và công sức. Nhưng niềm vui đó sẽ là gì? Không, chúng tôi sẽ tìm ra giải pháp theo cách lỗi thời bằng cách áp dụng các *nguyên tắc thiết kế phần mềm (software design principles)* OO tốt.

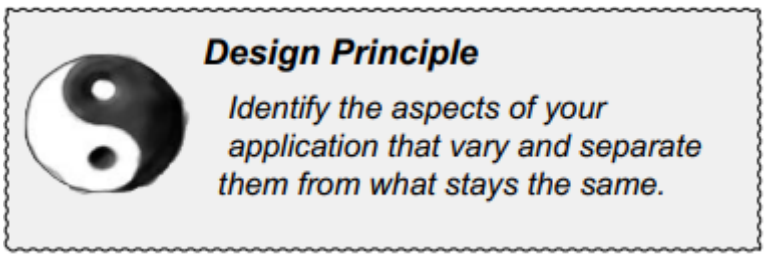
Một hằng số trong phát triển phần mềm

Được rồi, một thứ bạn luôn có thể dựa vào trong phát triển phần mềm là gì?
Bất kể bạn làm việc ở đâu, bạn đang xây dựng thứ gì, hoặc bạn đang lập trình ngôn ngữ gì, đâu là một hằng số thực sự sẽ luôn ở bên bạn?
THAY ĐỔI
Cho dù bạn thiết kế ứng dụng tốt đến đâu, theo thời gian, một ứng dụng phải phát triển và thay đổi nếu không nó sẽ chết.

Đưa vấn đề về số 0...

Chúng tôi biết việc sử dụng tính kế thừa sẽ rất hiệu quả, vì hành vi của vịt liên tục thay đổi trên các lớp con và nó không phù hợp với tất cả các lớp con để có những hành vi đó. Giao diện Flyable và Quackable nghe có vẻ hứa hẹn. Ở những con vịt thực sự bay sẽ là Flyable, v.v., ngoại trừ việc interface của Java không chứa code, do đó không thể dùng lại code trong trường hợp này. Và điều đó có nghĩa là bất cứ khi nào bạn cần sửa đổi một hành vi, bạn sẽ buộc phải theo dõi và thay đổi nó trong tất cả các lớp con khác nhau nơi hành vi đó được định nghĩa, và có thể sinh ra các lỗi mới trong tương lai!

May mắn thay, có một nguyên tắc thiết kế cho tình huống này.



Nói cách khác, nếu bạn có một số đoạn code đang thay đổi, hãy xem xét với tất cả những yêu cầu mới, sau đó bạn biết bạn sẽ có một hành vi cần phải được tách ra khỏi tất cả những thứ không thay đổi.

Ở đây, một cách khác để suy nghĩ về nguyên tắc này: lấy các thành phần thay đổi và đóng gói chúng, để sau này bạn có thể thay đổi hoặc mở rộng các thành phần khác nhau mà không ảnh hưởng đến những phần còn lại.

Đơn giản như khái niệm này, nó tạo thành nền tảng cho hầu hết mọi mẫu thiết kế. Tất cả các mẫu cung cấp một cách để cho một phần của hệ thống thay đổi độc lập với tất cả các phần khác.

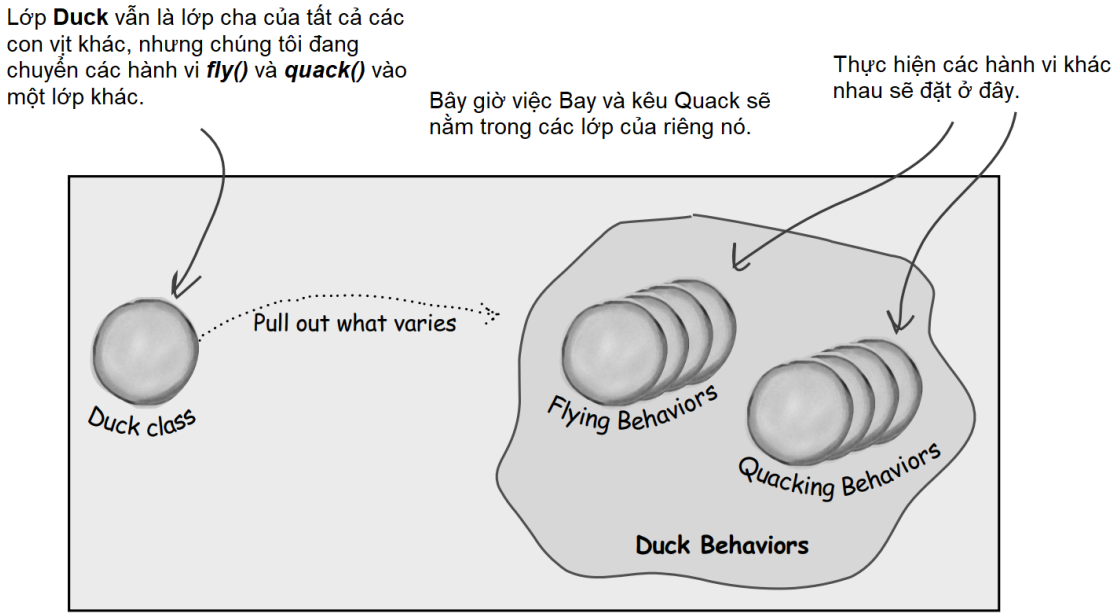
Được rồi, thời gian để đưa hành vi của những con vịt cần biến ra khỏi các lớp Duck!

Tách những thứ thay đổi ra khỏi những thứ không thay đổi

Chúng ta bắt đầu từ đâu? Theo như chúng tôi, có thể nói, ngoài các vấn đề với **fly()** và **quack()**, lớp **Duck** đang hoạt động tốt và không có phần nào khác của nó thay đổi hoặc thay đổi thường xuyên. Vì vậy, ngoài một vài thay đổi nhỏ, chúng tôi sẽ chuyển chúng ra khỏi lớp **Duck**.

Bây giờ, để phân tách các phần thay đổi từ các phần không thay đổi, chúng tôi sẽ tạo ra hai **bộ lớp** (hoàn toàn tách biệt với **Duck**), một bộ các lớp cho việc bay và một bộ các lớp cho việc kêu quack. Mỗi bộ sẽ đảm nhiệm tất cả hành vi tương ứng của chúng. Ví dụ, với bộ các lớp cho việc bay, chúng ta có thể có một lớp thực hiện việc quacking (vịt thật kêu), lớp khác thực hiện squeaking (vịt giả kêu) và lớp khác thực hiện việc im lặng (silence).

Chúng ta biết rằng **fly()** và **quack()** là các thành phần có thể thay đổi giữa các con vịt.
Để tách các hành vi này khỏi lớp Vịt, chúng tôi sẽ lấy cả hai phương thức ra khỏi lớp **Duck** và tạo một tập hợp các lớp mới để thể hiện mỗi hành vi.



Thiết kế hành vi của Duck

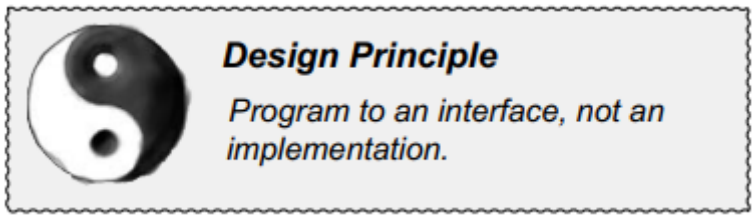
VẬY LÀM THẾ NÀO CHÚNG TA SẼ THIẾT KẾ BỘ CÁC LỚP THỰC HIỆN CÁC HÀNH VI FLY() VÀ QUACK()?

Chúng tôi muốn giữ mọi thứ khả thi; Rốt cuộc, chính sự linh hoạt trong hành vi của vịt đã khiến chúng tôi gặp rắc rối lúc đầu. Và chúng tôi biết rằng chúng tôi muốn gán các hành vi cho các trường hợp của Vịt. Ví dụ: chúng tôi có thể muốn khởi tạo một đối tượng MallardDuck mới với một loại hành vi cố định. Và

trong khi làm điều đó, tại sao không thay đổi hành vi của một con vịt một cách linh hoạt hơn? Nói cách khác, chúng ta nên bao gồm các phương thức setter hành vi trong các lớp Duck để chúng ta có thể thay đổi hành vi của MallardDuck trong khi chạy chương trình.

Từ giờ trở đi, các hành vi của Vịt sẽ đặt trong một lớp riêng biệt, Một lớp implement một interface hành vi cụ thể.
Theo cách đó, các lớp Vịt không cần phải biết bất kỳ implementation chi tiết nào thực hiện cho hành vi của nó.

Đưa ra những mục tiêu này, chúng ta hãy nhìn vào nguyên tắc thiết kế thứ hai của chúng tôi:



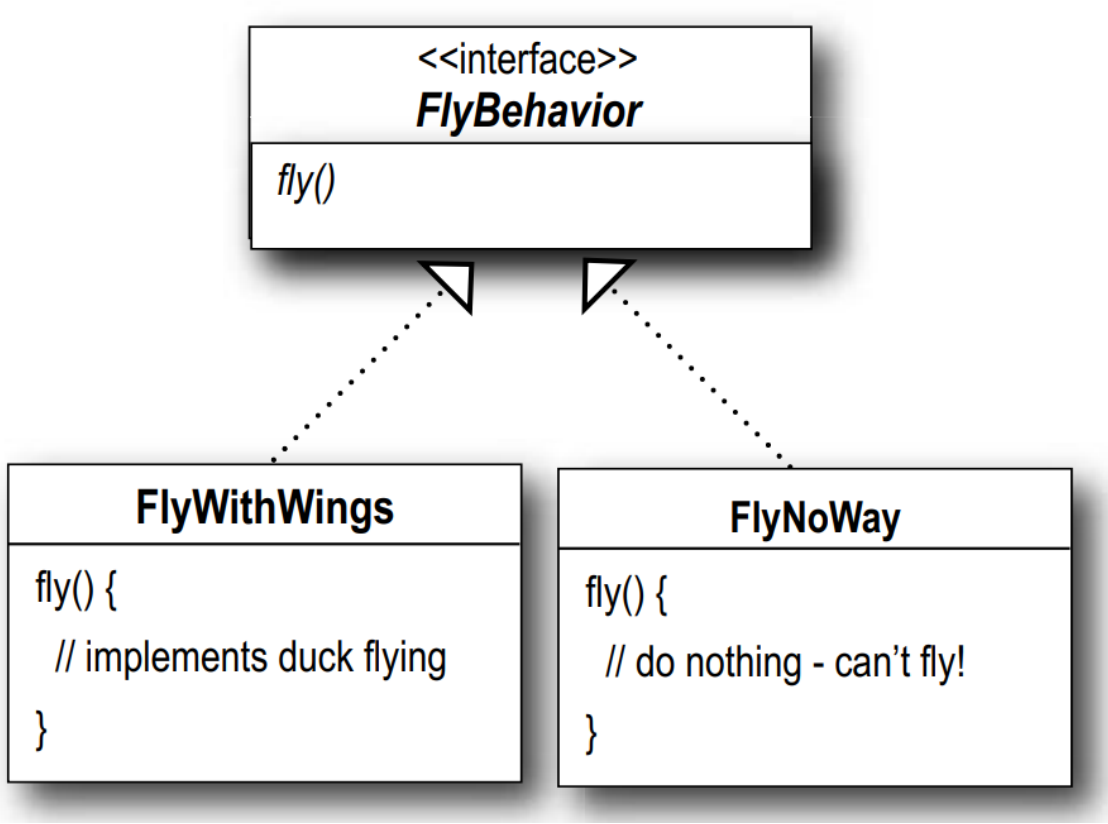
Lập trình với một interface, không phải một lớp con

Chúng tôi sẽ sử dụng một interface để thể hiện từng hành vi – ví dụ: **FlyBehavior** và **QuackBehavior** – và mỗi lần cần một hành vi thì sẽ implement một trong những interface đó.

Vì vậy, lần này lớp Duck sẽ không chứa phương thức flying và quacking. Thay vào đó, chúng tôi sẽ tạo ra một tập hợp các lớp để thể hiện một hành vi (ví dụ squeaking), và khi cần thay đổi, chỉ phải điều chỉnh các lớp hành vi, thay vì phải điều chỉnh lớp Duck.

Điều này trái ngược với cách chúng ta đã làm trước đây, trong đó một hành vi xuất phát từ việc triển khai trong lớp Duck, thay bằng cách implements một interface độc lập trong chính lớp con. Trong cả hai trường hợp, chúng tôi đều dựa vào implementation. Nhưng chúng tôi đã bị sa lầy vào việc implements trước đó và không có chỗ để thay đổi hành vi (ngoài việc viết thêm code).

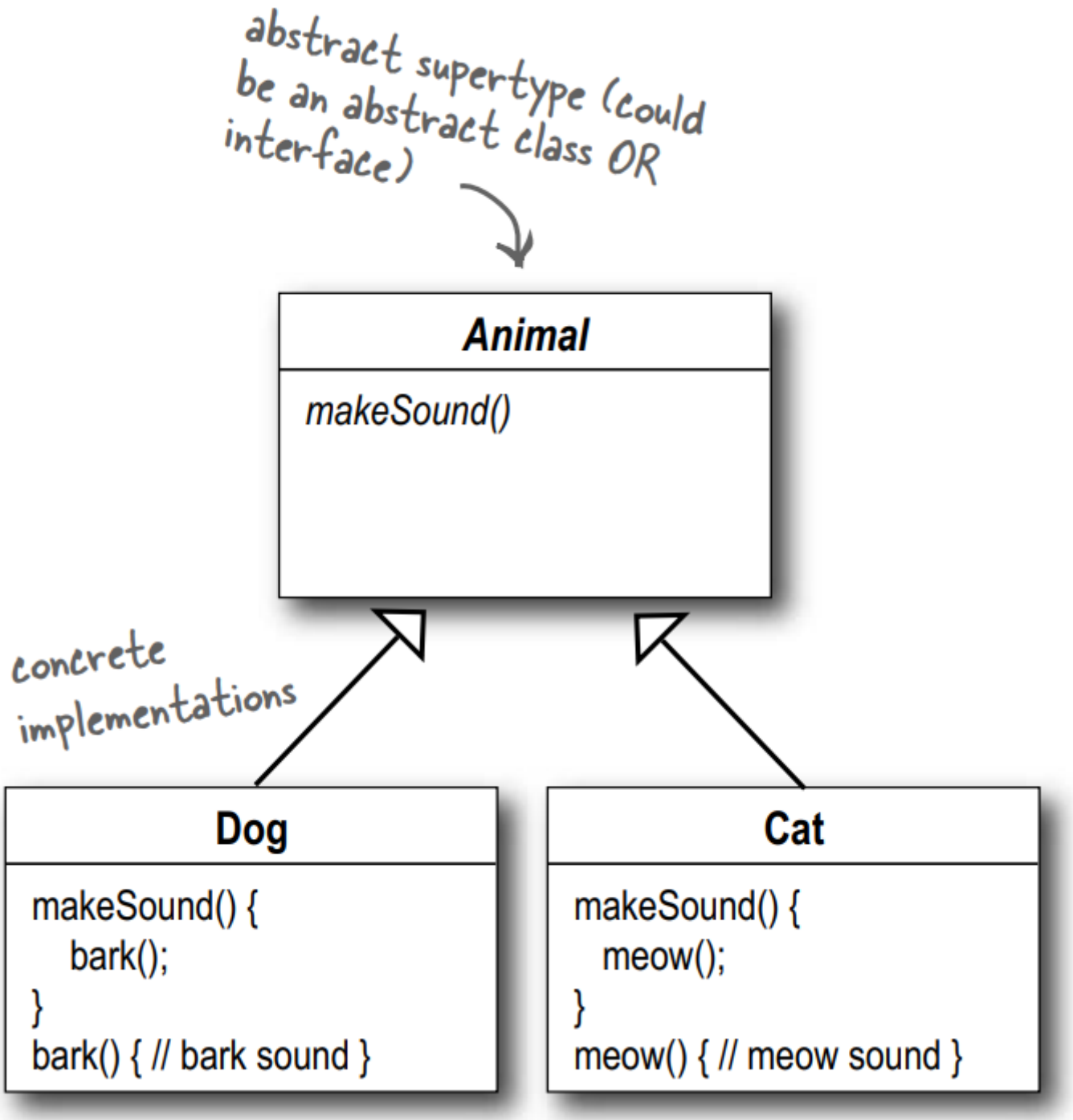
Với thiết kế mới của chúng tôi, các lớp con của Duck sẽ sử dụng một hành vi được đại diện bởi một interface (**FlyBehavior** và **QuackBehavior**), vì vậy thứ thực sự thực hiện các hành vi sẽ không bị khóa chặt trong các lớp con của Duck .



Thắc mắc: Tôi không thấy lý do tại sao bạn phải tạo interface cho FlyBehavior. Bạn có thể làm điều tương tự với một abstract class. Interface không phải là toàn bộ cách để sử dụng đa hình đúng chứ?

"PROGRAM TO AN INTERFACE" THỰC SỰ CÓ NGHĨA LÀ "PROGRAM TO A SUPERTYPE"

Từ *interface* đã bị sử dụng quá nhiều ở đây. Có một khái niệm về interface, nhưng Java cũng có từ khóa ***interface***. Bạn có thể "program to an interface" mà không cần phải thực sự sử dụng từ khóa interface trong Java. Vấn đề là khai thác tính đa hình bằng cách lập trình thành một supertype để đối tượng runtime không bị gắn cứng trong code. Và chúng ta có thể định nghĩa lại "program to a supertype" thành một siêu kiểu dữ liệu vì kiểu loại biến được khai báo phải là siêu kiểu, thường là một lớp trừu tượng hoặc giao diện, để các đối tượng được gán cho các biến đó có thể là bất kỳ triển khai cụ thể nào của siêu kiểu, có nghĩa là lớp khai báo chúng không cần phải biết về các loại đối tượng thực tế!
Dưới đây có lẽ là kiến thức cũ đối với bạn, nhưng chỉ để đảm bảo rằng tất cả chúng ta đều nói về cùng một thứ, ở đây, một ví dụ đơn giản về việc sử dụng một loại đa hình – hãy tưởng tượng một lớp **Animal** trừu tượng, với hai triển khai cụ thể là **Dog** và **Cat**.



"**Programming to an implementation**" sẽ là:

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

Khai báo biến "d" kiểu Dog (một triển khai cụ thể của Animal) buộc chúng ta phải code thành một triển khai cụ thể (Programming to an implementation).

Nhưng "**programming to an interface/supertype**" sẽ là:

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

Chúng ta biết nó là Dog, nhưng bây giờ chúng ta có thể sử dụng Animal tham chiếu một cách đa hình.

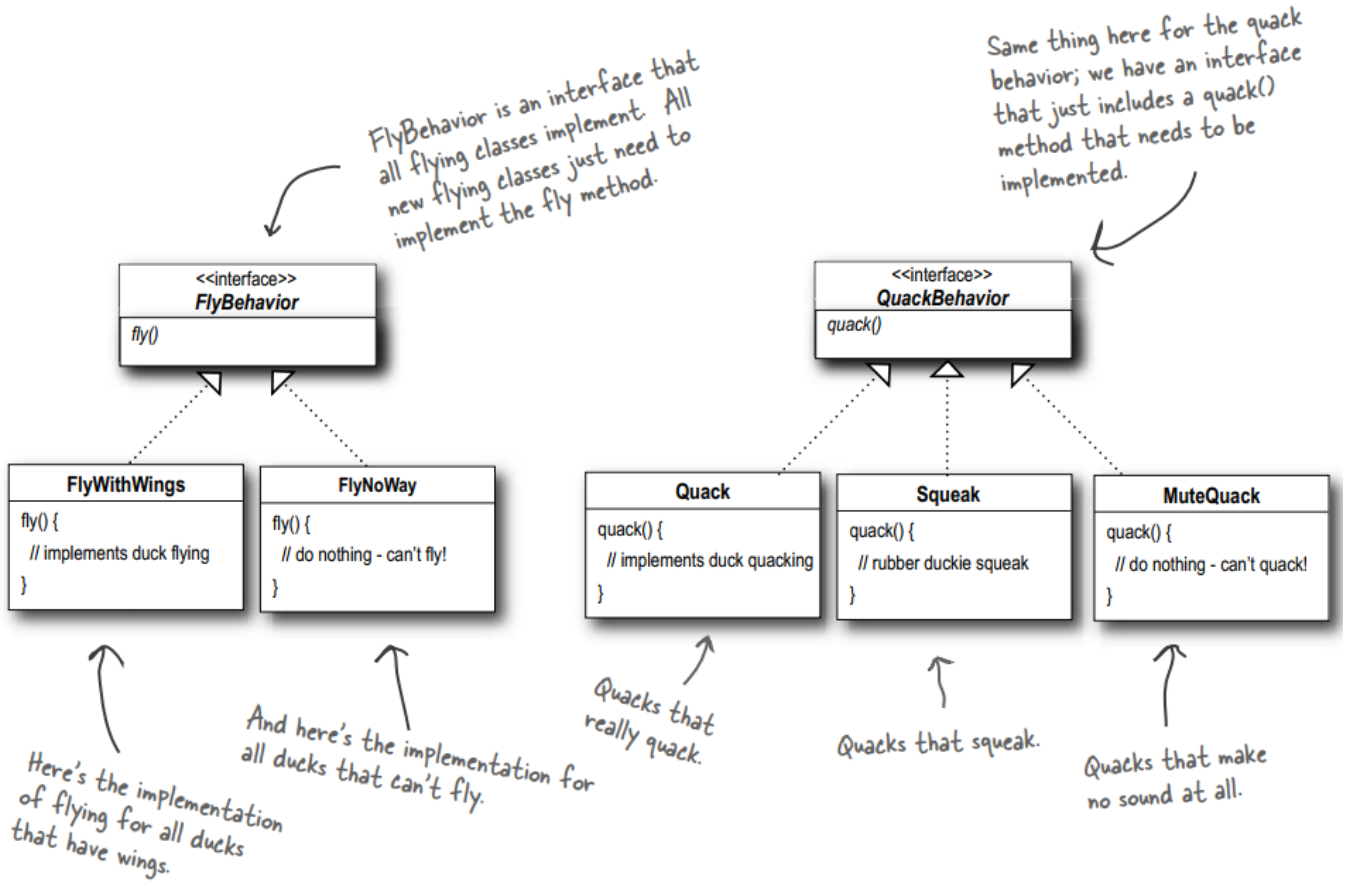
Thậm chí tốt hơn, thay vì code cứng việc khởi tạo kiểu lớp con (như *new Dog()*) vào code, hãy gán đối tượng triển khai cụ thể trong runtime:

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

Chúng tôi không biết rằng loại animal thực tế là gì … tất cả những gì chúng tôi quan tâm là nó biết cách phản ứng với makeSound().

Implementing những hành vi của Duck

Ở đây chúng ta có hai giao diện, **FlyBehavior** và **QuackBehavior** cùng với các lớp tương ứng implement từng hành vi cụ thể:



Với thiết kế này, các loại đối tượng khác có thể sử dụng lại các hành vi fly và quack, bởi vì chúng không còn bị che giấu trong các lớp Duck của chúng ta! Và chúng ta có thể thêm các hành vi mới mà không cần sửa đổi bất kỳ hành vi hiện có nào hoặc sửa đổi vào bất kỳ lớp Duck nào sử dụng hành vi fly().

Không câu hỏi nào ngớ ngẩn

Hỏi: Tôi có luôn phải triển khai ứng dụng của mình trước không, xem mọi thứ đang thay đổi ở đâu, sau đó quay lại và phân tách & gói gọn những thứ đó?
Đáp: Không phải lúc nào cũng vậy; thông thường khi bạn đang thiết kế một ứng dụng, bạn dự đoán

những khu vực sẽ thay đổi và sau đó tiếp tục và xây dựng tính linh hoạt để xử lý nó vào code của bạn. Bạn có thể thấy rằng các nguyên tắc và mô hình có thể được áp dụng ở bất kỳ giai đoạn nào trong vòng đời phát triển phần mềm.

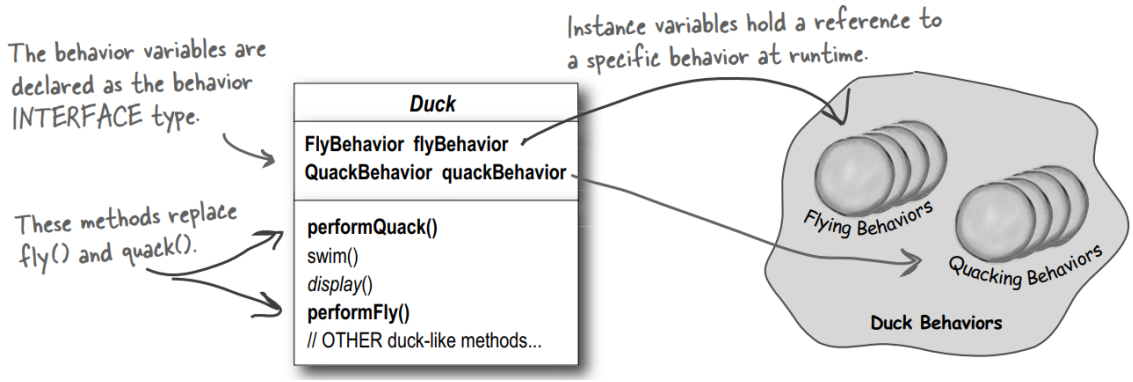
Hỏi: Chúng ta có nên tạo cho Duck một interface không?
Đáp: Không trong trường hợp này. Như bạn sẽ thấy khi chúng tôi kết nối mọi thứ lại với nhau, chúng tôi có lợi khi Duck không phải là một interface và có những con vịt cụ thể, như MallardDuck, thừa hưởng các đặc tính và phương thức chung. Bây giờ chúng tôi đã loại bỏ những gì thay đổi từ thừa kế Vịt, chúng tôi nhận được lợi ích của cấu trúc này mà không gặp vấn đề gì.

Hỏi: Cảm thấy hơi kỳ lạ khi có một lớp chỉ là một hành vi. Không phải các lớp được cho là đại diện cho mọi thứ? Không phải các lớp được cho là có cả trạng thái VÀ hành vi?
Đáp: Trong một hệ thống OO, có, các lớp đại diện cho những thứ thường có cả trạng thái (biến thể hiện) và phương thức. Và trong trường hợp này, sự việc xảy ra là một hành vi. Nhưng ngay cả một hành vi vẫn có thể có trạng thái và phương thức; một hành vi bay có thể có các biến thể hiện đại diện cho các thuộc tính cho hành vi bay (nhịp đập mỗi phút, độ cao tối đa và tốc độ, v.v.).

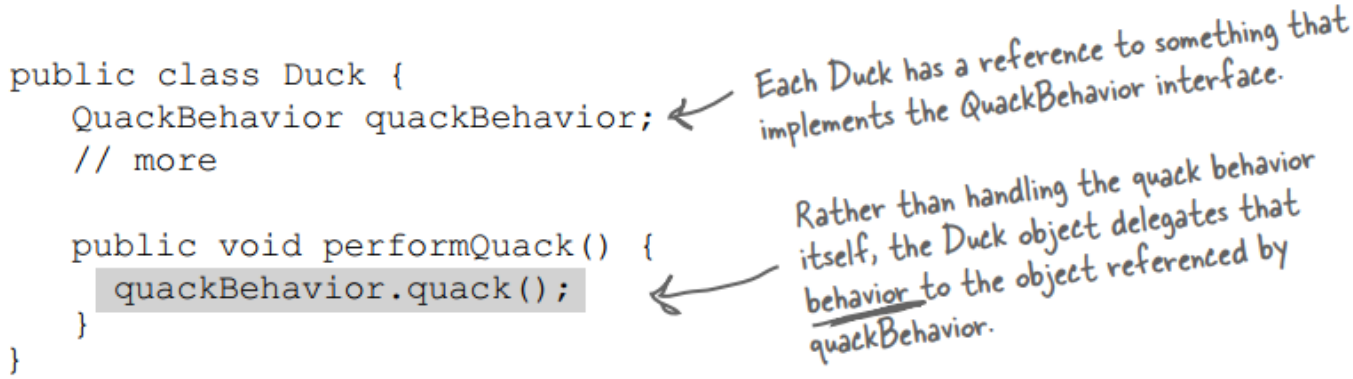
Tích hợp hành vi cho những chú vịt

Điều quan trọng bây giờ, một con Vịt sẽ ủy thác (delegate) hành vi bay và tiếng kêu của nó, thay vì sử dụng các phương thức quack() và fly() được định nghĩa trong lớp Duck (hoặc lớp con).
Đây là cách:

1. Đầu tiên, chúng tôi sẽ thêm hai biến instance vào lớp Duck được gọi là flyBehavior và quackBehavior, được khai báo theo kiểu ***interface*** (không phải là kiểu implement lớp cụ thể). Mỗi đối tượng duck sẽ đặt các biến này một cách đa hình để tham chiếu loại hành vi cụ thể mà nó muốn trong thời gian chạy (**FlyWithWings**, **Squeak**, v.v.).
Chúng tôi cũng sẽ loại bỏ các phương thức **fly()** và **quack()** khỏi lớp Duck (và bất kỳ lớp con nào của Duck) bởi vì chúng tôi đã chuyển hành vi này ra các lớp **FlyBehavior** và **QuackBehavior**.
Chúng tôi sẽ thay thế **fly()** và **quack()** trong lớp **Duck** bằng hai phương thức tương tự, được gọi là **PerformanceFly()** và **PerformanceQuack()**; bạn sẽ thấy cách nó làm việc bên dưới.



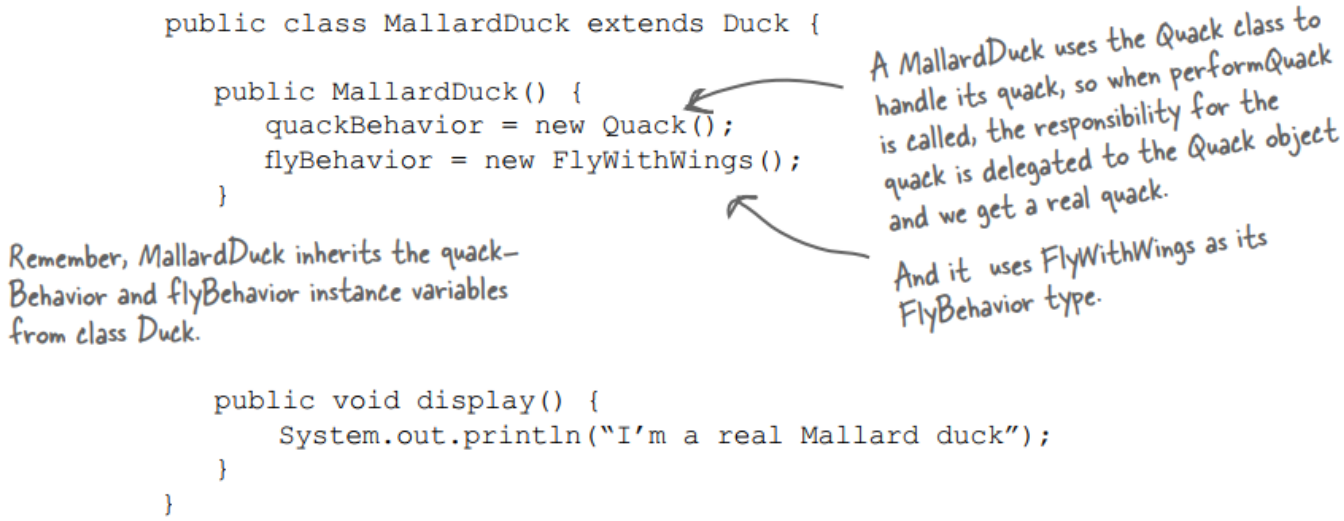
2. Bây giờ chúng ta implement performQuack():



Khá đơn giản nhỉ? Để thực hiện quack, Duck chỉ cho phép đối tượng được tham chiếu bởi quackBehavior thực hiện phương thức quack().

Trong phần code này, chúng tôi không quan tâm đến loại đối tượng quackBehavior, tất cả những gì chúng tôi quan tâm là nó biết cách quack().

3. Được rồi, thời gian để lo lắng về **cách các biến đối tượng flyBehavior và quackBehavior** được cài đặt. Hãy cùng xem lớp MallardDuck:



Bây giờ, MallardDuck có tiếng kêu **quack** như là một con vịt sống thật sự, không phải là **squeak** hay **mute quack**. Vậy chuyện gì xảy ra ở đây? Khi một MallardDuck được khởi tạo, hàm constructor của nó khởi tạo biến đối tượng **quackBehavior** với một instance của Quack (một lớp con của **QuackBehavior**).

Và điều tương tự cũng đúng với hành vi fly() của Duck, hàm khởi tạo MallardDuck, khởi tạo biến đối tượng **flyBehavior** với một instance của **FlyWithWings** (lớp con của **FlyBehavior**).

Đợi một chút, không phải bạn nói chúng ta KHÔNG NÊN "program to an implementation" sao? Nhưng chúng ta đang làm gì trong constructor đó? Chúng ta đang tạo một new instance của một lớp con kế thừa từ lớp Quack!

Nắm bắt tốt, đó chính xác là những gì chúng tôi đang làm … bây giờ.
Phần sau trong cuốn sách, chúng tôi sẽ có nhiều mẫu hơn trong có thể giúp chúng ta sửa nó.

Tuy nhiên, lưu ý rằng trong khi chúng ta đang đặt các hành vi thành các lớp cụ thể (bằng cách khởi tạo một lớp hành vi như Quack hoặc FlyWithWings và gán nó cho biến tham chiếu hành vi của chúng ta), chúng ta có thể dễ dàng thay đổi điều đó khi chạy.

Vì vậy, chúng tôi vẫn có rất nhiều tính linh hoạt ở đây, nhưng chúng tôi làm rất kém trong việc khởi tạo các biến thể hiện một cách linh hoạt. Nhưng hãy nghĩ về nó, vì biến đối tượng quackBehavior là một loại giao diện, chúng ta có thể (thông qua phép đa hình) tự động gán một lớp triển khai QuackBehavior khác trong thời gian chạy.

Dành một chút thời gian và suy nghĩ về cách bạn sẽ thực hiện một con vịt để hành vi của nó có thể thay đổi khi chạy. (Bạn sẽ thấy code thực hiện điều này ở một vài trang phía sau.)

Kiểm tra code con Vịt

1. Nhập và biên dịch lớp Duck (Duck.java) và lớp MallardDuck (MallardDuck.java).

```
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!");
   }
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.

2.Nhập và biên dịch giao diện FlyBehavior (FlyBehavior.java) và hai lớp thực hiện hành vi (FlyWithWings.java và FlyNoWay.java).

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

The interface that all flying behavior classes implement.

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

Flying behavior implementation for ducks that DO fly...

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

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

3. Nhập và biên dịch giao diện QuackBehavior (QuackBehavior.java) và ba lớp implement hành vi (Quack.java, MuteQuack.java và Squeak.java).

```
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");
   }
}
```

4. Nhập và biên dịch lớp để test (MiniDuckSimulator.java).

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

This calls the MallardDuck's inherited performQuack() method, which then delegates to the object's QuackBehavior (i.e. calls quack() on the duck's inherited quackBehavior reference).

Then we do the same thing with MallardDuck's inherited performFly() method.

5. Run the code!

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

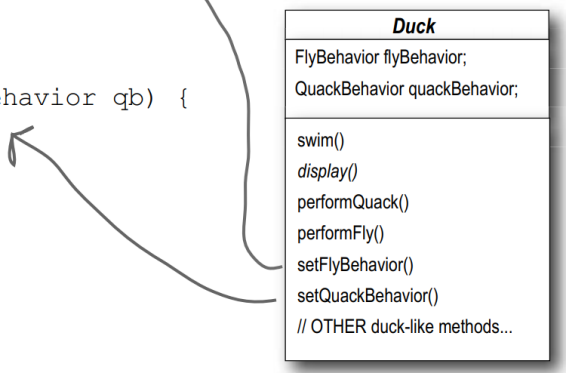
Cài đặt hành vi động

Hãy tưởng tượng bạn muốn thiết lập loại hành vi của vịt thông qua một phương thức setter trong lớp con của Duck, thay vì khởi tạo nó trong hàm constructor.

1. THÊM 2 PHƯƠNG THỨC MỚI VÀO LỚP DUCK:

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

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



Chúng ta có thể gọi những phương thức này bất cứ lúc nào chúng ta muốn thay đổi hành vi của một con vịt đang bay.

2. TẠO RA MỘT LOẠI DUCK MỚI (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");
    }
}
```

Our model duck begins life grounded... without a way to fly.

3. TẠO RA MỘT LOẠI FLYBEHAVIOR MỚI (FLYROCKETPOWERED.JAVA).

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

4. THAY ĐỔI LỚP TEST (MINIDUCKSIMULATOR.JAVA), THÊM MODELDUCK VÀ GỌI ĐẾN HÀNH VI CỦA NÓ.

ModelDuck, and make the ModelDuck rocket-enabled.

```
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();
    }
}
```

before

The first call to performFly() delegates to the flyBehavior object set in the ModelDuck's constructor, which is a FlyNoWay instance.

This invokes the model's inherited behavior setter method, and...voila! The model suddenly has rocket-powered flying capability!

If it worked, the model duck dynamically changed its flying behavior! You can't do THAT if the implementation lives inside the duck class.

after

Run it!

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

Để thay đổi hành vi một con Vịt trong runtime, chỉ cần gọi phương thức setter vịt cho hành vi mong muốn.

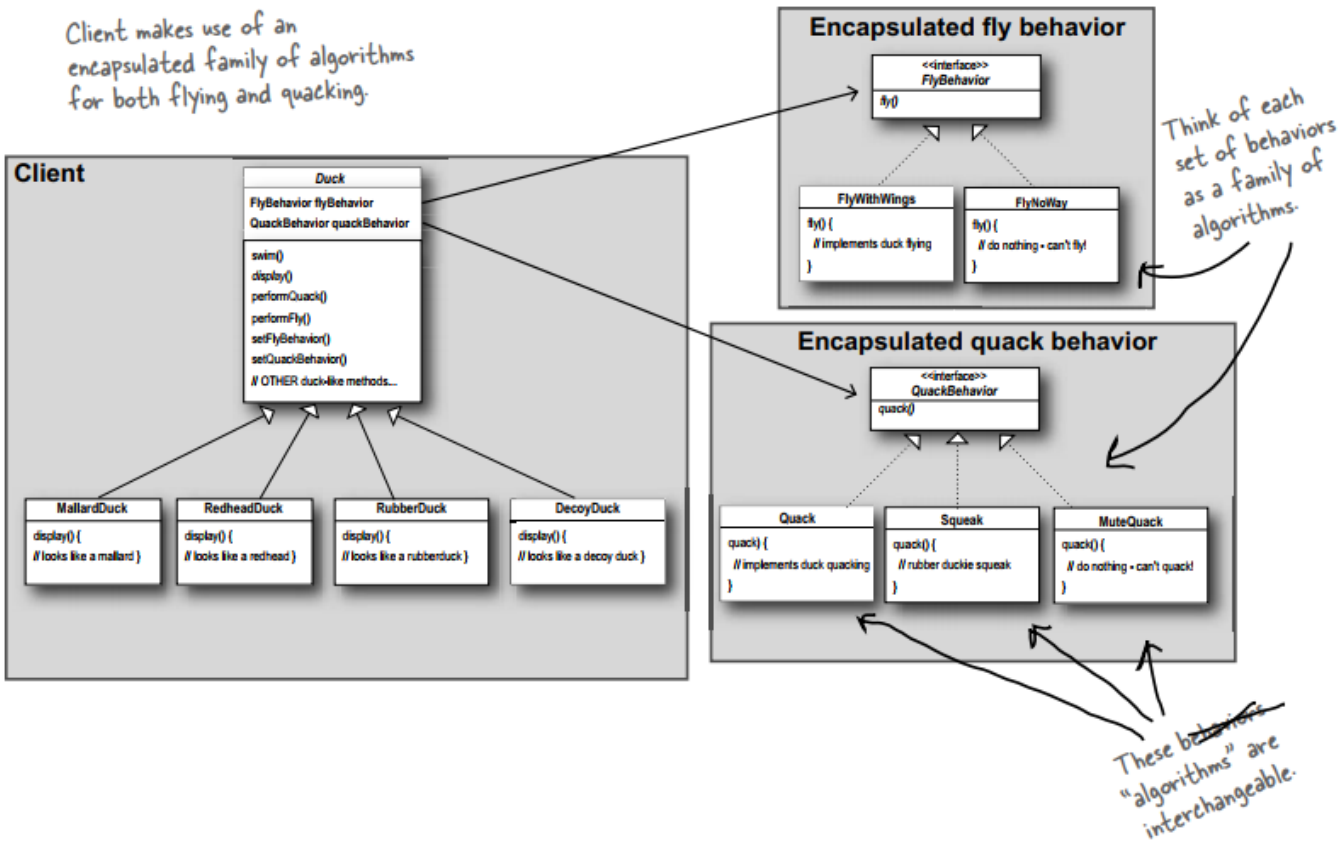
Nhìn tổng quan về việc đóng gói các hành vi

Được rồi, bây giờ chúng tôi đã thực hiện công việc "đào sâu" trên thiết kế giả lập vịt, đã đến lúc quay trở lại để lên cao và nhìn tổng quan hơn.

Dưới đây là toàn bộ cấu trúc lớp được làm lại. Chúng tôi có tất cả mọi thứ bạn mong đợi: một con vịt sẽ extends Duck, hành vi bay implementing **FlyBehavior** và hành vi kêu quack implementing **QuackBehavior**.

Cũng lưu ý rằng chúng tôi đã bắt đầu mô tả mọi thứ khác đi một chút. Thay vì nghĩ về các hành vi của vịt như một tập hợp các hành vi, chúng tôi sẽ bắt đầu nghĩ về chúng như một tập hợp các thuật toán (family of algorithms). Hãy suy nghĩ về điều này: trong thiết kế SimUDuck, các thuật toán đại diện cho những việc mà một con vịt sẽ làm (các cách khác nhau để bay hoặc kêu), nhưng chúng ta có thể dễ dàng sử dụng các kỹ thuật tương tự cho một "nhóm các lớp" implement các cách thực hiện bởi các trạng thái khác nhau.

Hãy chú ý cẩn thận đến các mối quan hệ giữa các lớp. Trong thực tế, lấy bút của bạn và viết mối quan hệ phù hợp (**IS-A**: là một, **HAS-A**: có một và **IMPLEMENTS**) trên mỗi mũi tên trong sơ đồ lớp.

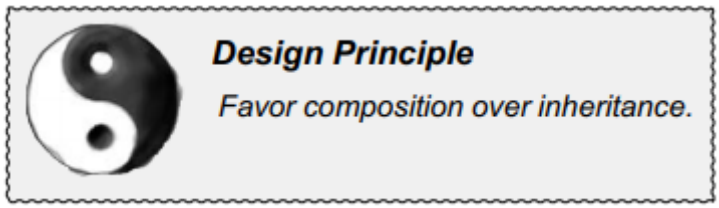


HAS-A có thể tốt hơn IS-A

Mối quan hệ **HAS-A** là một mối quan hệ thú vị: mỗi con vịt **CÓ MỘT** hành vi bay và hành vi kêu mà nó ủy nhiệm bay và kêu.

Khi bạn đặt hai lớp với nhau như thế này, bạn sẽ sử dụng kết hợp (composition). Thay vì kế thừa hành vi của chúng, những con vịt có được hành vi của chúng bằng cách được kết hợp (compose) với object hành vi phù hợp.

Đây là một kỹ thuật quan trọng; Trên thực tế, chúng tôi đã sử dụng nguyên tắc thiết kế thứ ba của mình:



Sử dụng việc kết hợp hơn là kế thừa

Như bạn đã thấy, việc tạo các hệ thống bằng cách sử dụng việc kết hợp giúp bạn linh hoạt hơn rất nhiều. Nó không chỉ cho phép bạn đóng gói một nhóm thuật toán vào tập hợp các lớp của riêng chúng mà còn cho phép bạn thay đổi hành vi khi runtime miễn là đối tượng bạn đã composing (kết hợp) với hành vi của interface khác.

Sự kết hợp (Composition) được sử dụng trong nhiều mẫu thiết kế và bạn sẽ thấy nhiều hơn về những lợi thế và bất lợi của nó trong suốt cuốn sách.

Note: ở đoạn này mình không dịch được hết nghĩa của từ Composition được, vì thế mình để nguyên. Các bạn có thể hiểu các lớp Duck và DuckBehavior đang kết hợp (composition) với nhau.

Động não một chút

Một “duck call” là công cụ là mà các thợ săn sử dụng để bắt chước tiếng kêu (quacks) của vịt. Làm thế nào bạn có thể tạo lớp “duck call” của riêng bạn mà không kế thừa từ lớp Duck?

Sư phụ và học trò...

Sư phụ: Grasshopper, hãy cho thầy biết những gì con đã học được về hướng đối tượng.

Học trò: Sư phụ, con đã học được rằng hướng đối tượng là tái sử dụng lại code.

Sư phụ: Tiếp tục đi…

Học trò: Sư phụ, thông qua thừa kế, tất cả những gì cần thiết có thể được tái sử dụng và vì vậy chúng sẽ giảm đáng kể thời gian viết code.

Sư phụ: Grasshopper, và sẽ tốn nhiều thời gian hơn trước hay sau công việc develop?

Học trò: Câu trả lời là sau, thưa thầy. Con luôn dành nhiều thời gian hơn khi bảo trì và thay đổi phần mềm so với phát triển ban đầu.

Sư phụ: Vậy Grasshopper, có nên nỗ lực tái sử dụng khi tốn thời gian bảo trì sau này?

Học trò: Sư phụ, con tin rằng không nên trong việc này.

Sư phụ: Thầy có thể thấy rằng con vẫn còn nhiều điều phải học. Thầy muốn con đi và suy ngẫm về kế thừa hơn nữa. Như con đã thấy, kế thừa có vấn đề của nó, và còn nhiều cách khác để đạt được mục đích tái sử dụng.

Nói về các mẫu thiết kế ...

Bạn vừa áp dụng mẫu thiết kế đầu tiên của mình, mẫu **STRATEGY**. Đúng vậy, bạn đã sử dụng **STRATEGY** để làm lại ứng dụng **SimUDuck**.

Nhờ mẫu này, trình giả lập đã sẵn sàng cho mọi thay đổi mà những người thực hiện có thể sẽ làm trong chuyến công tác tiếp theo tới Vegas.

Cho tới hiện tại, chúng tôi đã đưa bạn đi một đoạn đường dài để áp dụng nó, và ở đây, định nghĩa chính thức của mẫu này:

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

Mẫu Strategy xác định một họ các thuật toán, gói gọn từng cái và làm cho chúng có thể hoán đổi cho nhau. Strategy cho phép thuật toán thay đổi độc lập với các client sử dụng nó.

Design Puzzle

Bên dưới bạn sẽ tìm thấy một mớ hỗn độn các class và interface cho một game phiêu lưu hành động. Bạn sẽ tìm thấy các class cho các nhân vật trong game cùng với các class cho các hành vi vũ khí mà các nhân vật có thể sử dụng trong trò chơi. Mỗi lần, nhân vật có thể sử dụng một vũ khí, nhưng có thể thay đổi vũ khí bất cứ lúc nào trong trò chơi. Công việc của bạn là sắp xếp tất cả …

NHIỆM VỤ CỦA BẠN:

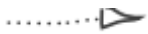
1. Sắp xếp các class.

2. Xác định một lớp trừu tượng, một giao diện và 8 class.

3. Vẽ mũi tên giữa các lớp.

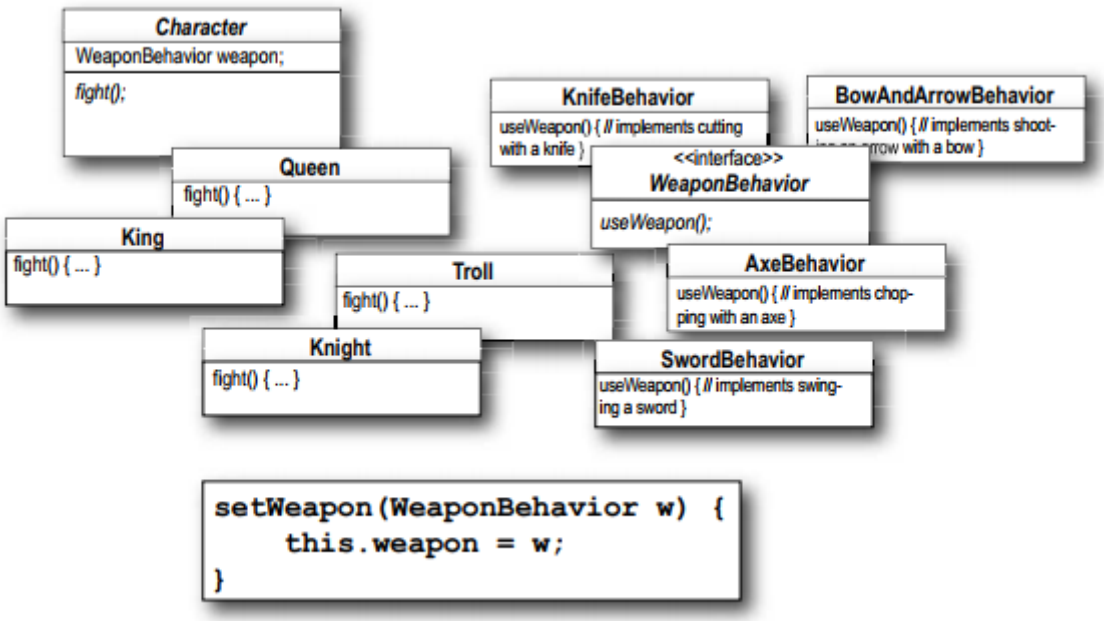
a. Vẽ loại mũi tên này để thừa kế (extends).

b. Vẽ loại mũi tên này cho giao diện (implements).

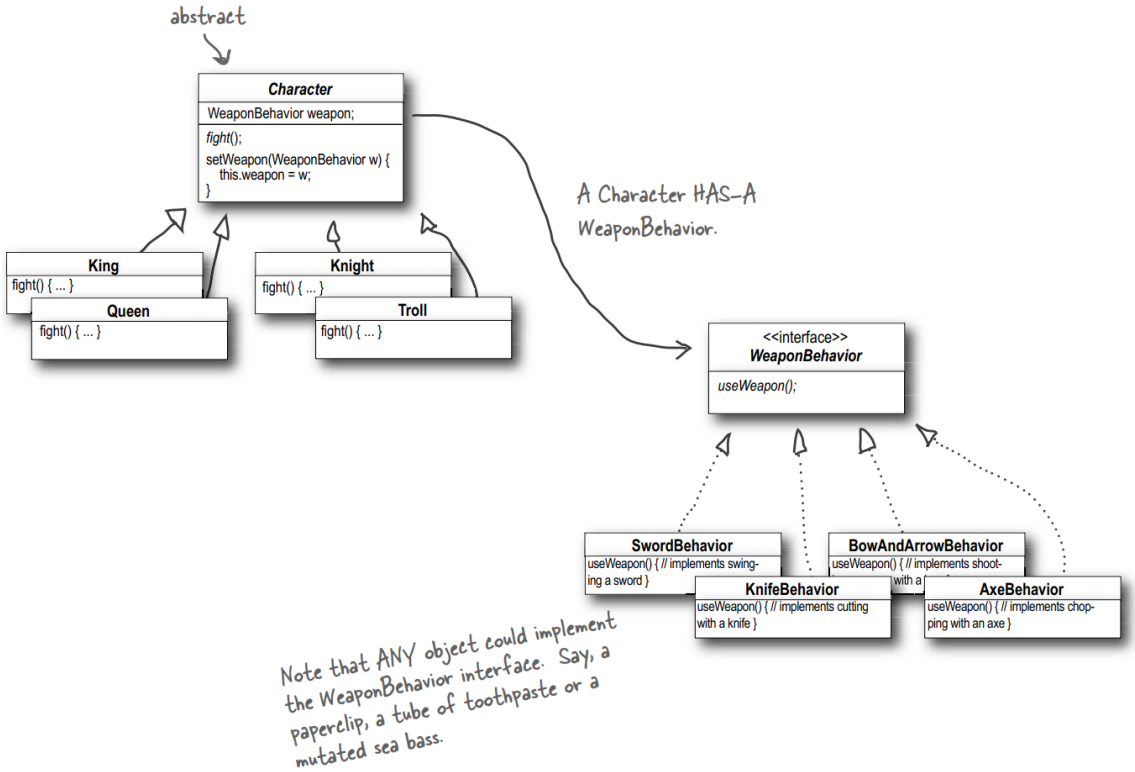


c. Vẽ loại mũi tên này cho "HAS-A".

4. Đặt phương thức *setWeapon()* vào đúng lớp.



Đây là câu trả lời:



Tình cờ nghe thấy ở một quán ăn địa phương...

Điều gì khác biệt giữa hai đơn đặt hàng này? Không một thứ gì! Cả hai cùng một thứ tự, ngoại trừ Alice đang sử dụng gấp đôi số từ và thử sự kiên nhẫn của một đầu bếp.

Những gì Flo nhận được mà Alice không có? Một "từ vựng" đơn hàng ngắn. Không chỉ dễ dàng hơn để giao tiếp với người nấu ăn, mà nó còn giúp người nấu ít nhớ hơn bởi vì anh ấy có tất cả các công thức nấu ăn trong đầu.

Mẫu thiết kế cũng cung cấp cho bạn một "từ vựng" được chia sẻ với các developer khác. Khi bạn đã có được vốn từ vựng, bạn có thể dễ dàng giao tiếp với các nhà phát triển khác và truyền cảm hứng cho những người không biết mẫu thiết kế để bắt đầu học chúng. Nó cũng nâng cao suy nghĩ của bạn về kiến trúc bằng cách cho phép bạn suy nghĩ ở cấp độ mẫu chứ không phải cấp độ đối tượng.

Nghe lỏm trong tủ bên cạnh ...

Sức mạnh của một "từ vựng pattern"

Khi bạn giao tiếp bằng cách sử dụng các mẫu bạn đang làm nhiều hơn là chia sẻ lời nói.

Chia sẻ "Từ vựng Pattern" là siêu sức mạnh.
Khi bạn giao tiếp với developer khác hoặc nhóm của bạn bằng các mẫu thiết kế, bạn đang giao tiếp không chỉ là tên mẫu mà là toàn bộ các phẩm chất, đặc điểm và ràng buộc mà mẫu đó thể hiện.

Các mẫu cho phép bạn nói nhiều hơn với ít hơn.
Khi bạn sử dụng một mẫu trong một mô tả, các nhà phát triển khác nhanh chóng biết chính xác thiết kế mà bạn có trong đầu.

Nói ở pattern level cho phép bạn ở lại trong thiết kế của lâu hơn.
Nói về các hệ thống phần mềm bằng cách sử dụng các mẫu cho phép bạn giữ cuộc thảo luận ở mức thiết kế, mà không cần phải đi sâu vào các chi tiết nghiệt ngã của việc thực hiện các đối tượng và các lớp.

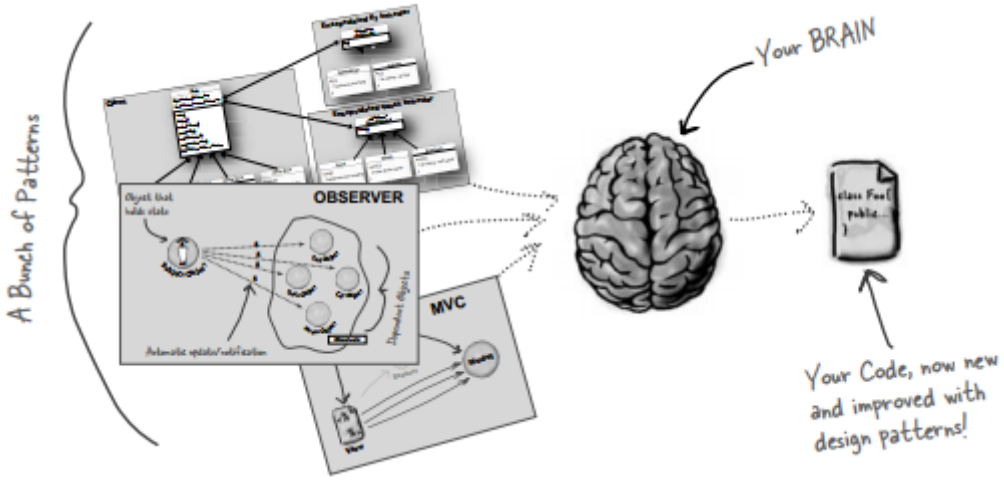
Từ vựng được chia sẻ có thể tăng tốc cho đội ngũ phát triển của bạn.
Một nhóm thành thạo các mẫu thiết kế có thể di chuyển nhanh hơn với ít chỗ hơn cho sự hiểu lầm.

Các từ vựng được chia sẻ khuyến khích nhiều developer trẻ hơn để đạt được tốc độ.
Các developer trẻ tìm đến các developer có kinh nghiệm. Khi các developer có kinh nghiệm sử dụng các mẫu thiết kế, các developer trẻ cũng trở nên có động lực để tìm hiểu chúng. Xây dựng một cộng đồng mẫu người dùng tại tổ chức của bạn.

Làm cách nào để sử dụng Mẫu thiết kế?

Chúng tôi đã sử dụng tất cả các thư viện và frameworks sẵn có. Chúng tôi lấy chúng, viết một số dòng code từ API của chúng, biên dịch chúng vào các chương trình của chúng tôi và được hưởng lợi từ rất nhiều code mà người khác đã viết. Hãy suy nghĩ về các API Java và tất cả các chức năng mà chúng cung cấp cho bạn: network, GUI, IO, v.v. Các thư viện và framework hướng tới một mô hình phát triển nơi chúng ta có thể chọn và chọn các thành phần và sử dụng chúng. Nhưng … họ không giúp chúng ta cấu trúc các ứng dụng của riêng mình theo những cách dễ hiểu hơn, dễ bảo trì hơn và khả thi hơn. Đó là những nơi mà các mẫu thiết kế cần đến.

Các mẫu thiết kế không đi trực tiếp vào code của bạn, đầu tiên chúng đi vào NÃO của bạn. Khi bạn đã load bộ não của mình với kiến thức làm việc tốt về các mẫu, bạn có thể bắt đầu áp dụng chúng cho các thiết kế mới của mình và xử lý lại code cũ khi bạn biến nó thành một mớ hỗn độn "spaghetti code".

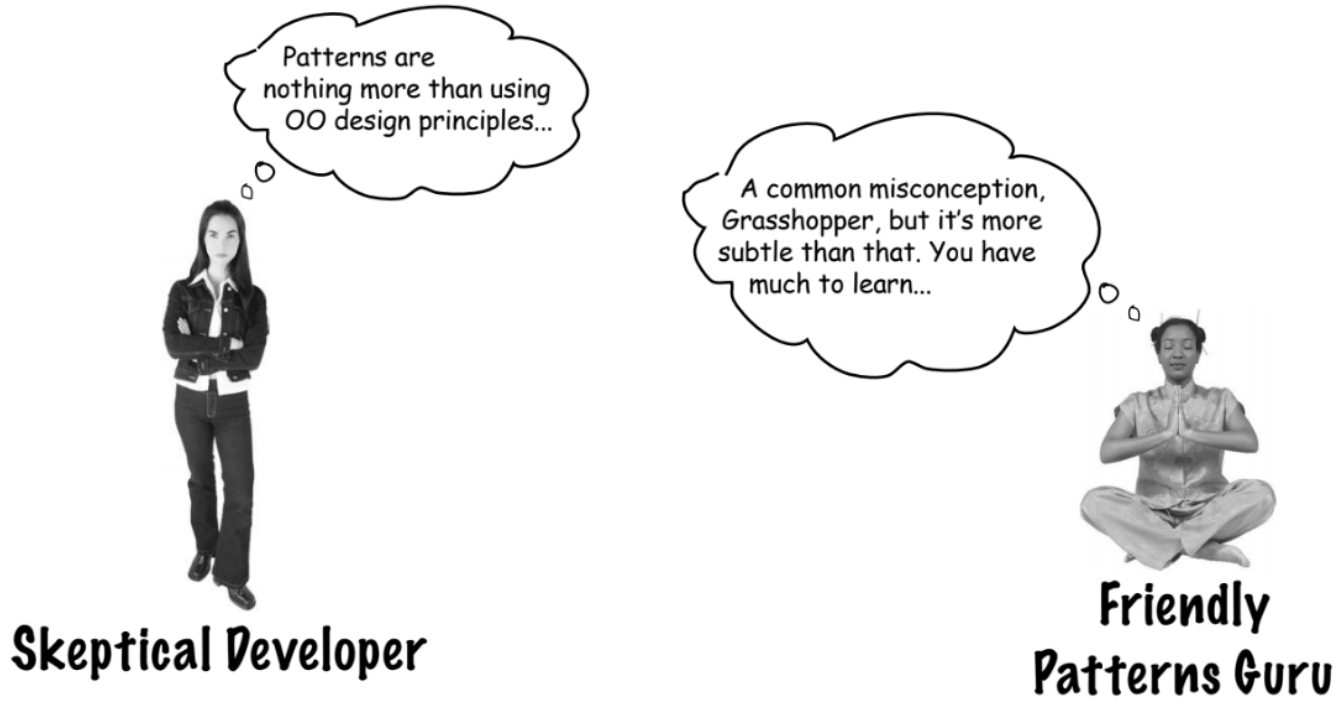


Không có câu hỏi ngớ ngẩn

Hỏi: Nếu các mẫu thiết kế quá tuyệt vời, tại sao không xây dựng một thư viện của chúng?
Đáp: Các mẫu thiết kế ở cấp độ cao hơn so với các thư viện. Các mẫu thiết kế cho chúng ta biết cách cấu trúc các lớp và các đối tượng để giải quyết các vấn đề nhất định và công việc của chúng ta là điều chỉnh các thiết kế đó để phù hợp với đặc điểm của chúng ta.

Hỏi: Libraries và frameworks cũng là một design pattern?
Đáp: Libraries và frameworks không phải là mẫu thiết kế; nó cung cấp các triển khai cụ thể mà bạn sẽ dùng chúng trong code của bạn. Tuy nhiên, đôi khi, các Libraries và frameworks sử dụng các mẫu thiết kế trong việc triển khai của chúng. Thật tuyệt vời, bởi vì một khi bạn hiểu các mẫu thiết kế, bạn sẽ hiểu các API nhanh hơn.

Hỏi: Vì vậy, không có Libraries của các mẫu thiết kế?
Đáp: Không, nhưng bạn sẽ tìm hiểu sau về danh mục mẫu với danh sách các mẫu mà bạn có thể áp dụng cho các ứng dụng của mình.



Developer: Được rồi, hmm, nhưng đây không phải là thiết kế hướng đối tượng tốt; Ý tôi là miễn là tôi tuân theo sự đóng gói và tôi biết về sự trừu tượng, tính kế thừa và tính đa hình, tôi có thực sự cần phải suy nghĩ về các Mẫu thiết kế không? Có phải nó khá đơn giản? Đây có phải là lý do tại sao tôi tham gia tất cả các khóa học OO đó không? Tôi nghĩ Mẫu thiết kế chỉ hữu ích cho những người không biết thiết kế hướng đối tượng.

Giáo sư: Ah, đây là một trong những hiểu lầm thực sự về phát triển hướng đối tượng: rằng bằng cách biết các kiến thức cơ bản về OO, chúng tôi sẽ tự động xây dựng các hệ thống linh hoạt, có thể tái sử dụng và bảo trì.

Developer: Đúng vậy chứ?

Giáo sư: Như vậy, việc xây dựng các hệ thống OO có các tính chất này không phải lúc nào cũng rõ ràng và chỉ được phát hiện thông qua làm việc.

Developer: Tôi nghĩ rằng tôi đã bắt đầu có được nó. Những cách này, đôi khi không rõ ràng, để xây dựng các hệ thống hướng đối tượng đã được có được…

Giáo sư: … vâng, thành một tập các mẫu gọi là Mẫu thiết kế.

Developer: Vì vậy, bằng cách biết các mẫu, tôi có thể bỏ qua công việc khó khăn và nhảy thẳng đến các thiết kế luôn hoạt động tốt?

Giáo sư: Vâng, ở một mức độ nào đó, nhưng hãy nhớ rằng, thiết kế là một nghệ thuật. Sẽ luôn có sự đánh đổi. Nhưng, nếu bạn làm theo các mẫu thiết kế được suy nghĩ kỹ lưỡng và được thử nghiệm theo thời gian, bạn sẽ đi trước những người khác.

Developer: Tôi phải làm gì nếu tôi có thể tìm thấy một mẫu thiết kế?

Giáo sư: Có một số nguyên tắc hướng đối tượng làm nền tảng cho các mẫu và biết những điều này sẽ giúp bạn đối phó khi bạn có thể tìm thấy một mẫu phù hợp với vấn đề của bạn.

Developer: Nguyên tắc? Ý bạn là ngoài sự trừu tượng, đóng gói và …

Giáo sư: Vâng, một trong những bí mật để tạo ra các hệ thống OO có thể bảo trì là suy nghĩ về cách chúng có thể thay đổi trong tương lai và những nguyên tắc này sẽ giải quyết những vấn đề đó.