**DESIGN PATTERN**

1. Strategy Design Pattern

1.1. Phân tích vấn đề

- Giả sử ta có một ứng dụng về quản lý nhóm Vịt

+ Với giải pháp bình thường kế thừa:

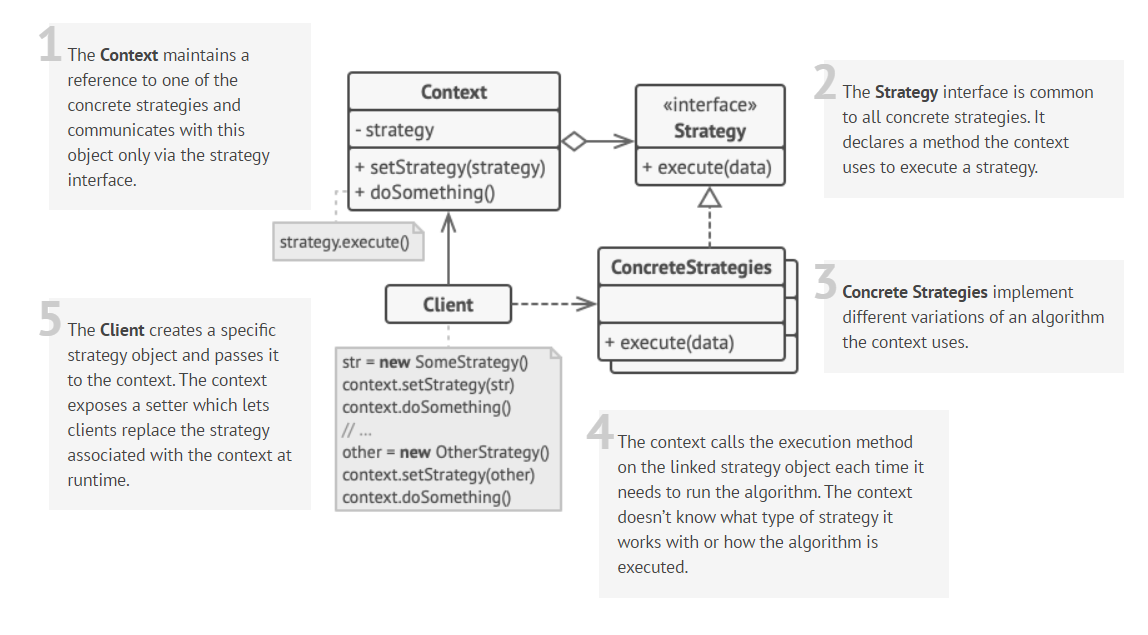
* Ta sẽ tạo ra một class Vit chứa các thuộc tính và phương thức (ăn, uống, bay, chạy,..).
* Sau đó ta sẽ tiến hành tạo các class con Vịt trời,… kế thừa Vịt ban đầu.
* Vấn đề sảy ra khi Vịt Cỏ xuất hiện. Bởi Vịt Cỏ không hề biết bay nên ta phải override lại phương thức bay. Và sau đó các con Vịt Xiêm, Vịt Nhà,… xuất hiện, ta lại tiếp tục override lại phương thức bay.
* Tuy nhiên tiếp theo vấn để lại tiếp tục xuất hiện khi Khách Hàng của chúng ta yêu cầu trong phương thức Bay của chúng ta sẽ xuất ra một câu “Tao biết bay” nếu vịt biết bay, và “Tao không biết bay” với vịt không biết bay. Thế là ta ngồi và chỉnh từng phương thức Bay trong các class ta tạo ra. Công việc như vậy quá tốn thời gian.
* Tuy nhiên, sáng mai thức dậy, ta lại tiếp tục nhận được yêu cầu của khách hàng. Khách hàng lại muốn phân chia nhóm Vịt biết bay ra thành nhóm vịt bay cao và bay thấp, và lại tiếp tục in ra câu “Tao bay cao vãi”, “Tao bay thấp quá”. Đến lúc này không còn cách nào khác ta phải sửa lại từ logic đến các chuổi String trong phương thức Bay.
* Không dừng lại ở đó, Khách hàng của chúng ta rất thích kiềm chuyện. Ngày hôm sau, ông ta lại bảo. “Thôi, tôi thấy chia vịt bay cao thấp quá vớ vẫn, bạn sửa lại như cũ giúp tôi”. What the f\*\*\* man. Nhưng thôi, bạn có chửi thề hay làm gì thì bạn cũng phải sửa lại theo yêu cầu khách hàng. Cuộc sống mà biết sao giờ.
* Như bạn thấy đấy, việc sử dụng kế thừa trong hướng đối tượng cơ bản trong trường hợp này, sẽ khiền code chúng ta trước hết là không clean dẫn đến ta phải sữa đi sữa lại rất nhiều lần. Và tất nhiên điều đó là không chấp nhận được với một tín đồ Java như tôi. Ở java có một câu rất nổi tiếng “Write One, Run AnyWhere” nghĩa là code một lần và chạy bất cứ đâu. Ta phải tìm ra phương pháp để chúng ta chỉ cần sửa một lần và các lớp Vịt con đều sẽ nhận được. Và chúng ta sẽ biết đến nó ở phần tiếp theo Giải pháp.

1.2. Giải pháp

- Như vấn đề chúng ta đã phân tích, ta đang gặp vấn đề với phương thức Bay, nó khiến chúng ta phải cực khổ thay đi thay lại n lần.

- Vậy sao chúng ta không định nghĩa một Interface/Abtract class Bay. Sau đó các Class Bay đủ thể loại của ta hoàn toàn có thể implement Bay. Trong Bay chúng ta sẽ có Không biết bay, Bay Cao, Bay thấp, và thậm chí sau này Khách Hàng có muốn Bay Xuống Địa Phủ hay Bay lên Thương Đường, ta cũng chỉ định nghĩa implement Bay. Và Vit cũng chỉ cần một phương thức get để nhận kết quả từ Bay mà chẳng cần quan tâm vấn đề khác.

1.3. Kiến trúc (Class Diagram)



1.4. Ví dụ

- Ta sẽ tiếp tục thực hiện với ví dụ ta đã nhắc tới trong phần Phân tích.

- Đầu tiên ta sẽ tạo một Class Duck

public class Duck {  
 private int \_id;  
 private String \_gender;  
 private String \_price;  
 private IFlyStrategy \_flyStrategy;  
  
 public Duck() {  
  
 }  
  
 public Duck(IFlyStrategy *flyStrategy*) {  
 \_flyStrategy = *flyStrategy*;  
 }  
  
 public int get\_id() {  
 return \_id;  
 }  
  
 public void set\_id(int *\_id*) {  
 this.\_id = *\_id*;  
 }  
  
 public String get\_gender() {  
 return \_gender;  
 }  
  
 public void set\_gender(String *\_gender*) {  
 this.\_gender = *\_gender*;  
 }  
  
 public String get\_price() {  
 return \_price;  
 }  
  
 public void set\_price(String *\_price*) {  
 this.\_price = *\_price*;  
 }  
  
 public IFlyStrategy get\_flyStrategy() {  
 return \_flyStrategy;  
 }  
  
 public void set\_flyStrategy(IFlyStrategy *\_flyStrategy*) {  
 this.\_flyStrategy = *\_flyStrategy*;  
 }  
  
 public void fly() {  
 this.\_flyStrategy.doFly();  
 }  
}

- Trong Duck ngoài các thông tin cơ bản ta sẽ khao báo thêm một IflyStrategy là một interface chứa phương thức doFly().

- Ta sẽ bỏ qua các thông tin khác, trong ví dụ này, ta chỉ tập trung vào phương thức fly() trong Duck. Phương thức fly của chúng ta đang gọi đến doFly() trong Interface

public interface IFlyStrategy {  
 void doFly();  
}

- Và tất nhiên Interface của chúng ta định nghĩa chỉ có khai báo phương thức, ta sẽ tiến hành khởi tạo các class implement interface này để thực hiện các loại Bay.

- Đầu tiên ta dịnh nghĩa lớp Không biết bay

public class NoFlyStrategy implements IFlyStrategy{  
  
 *@Override* public void doFly() {  
 System.out.println("I don't know how to fly, please!");  
 }  
}

- Lớp này sẽ implement Interface của chúng ta, và override lại phương thức doFly()

- Tương tự ta tạo các lớp khác

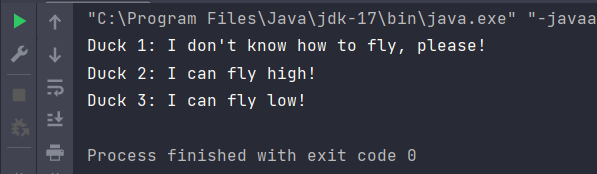
public class FlyHighStrategy implements IFlyStrategy{  
 *@Override* public void doFly() {  
 System.out.println("I can fly high!");  
 }  
}

public class FlyLowStrategy implements IFlyStrategy{  
 *@Override* public void doFly() {  
 System.out.println("I can fly low!");  
 }  
}

- Tiếp đó ta sẽ khai báo một vài duck để test fly()

public class Main {  
 public static void main(String[] *args*) {  
 Duck duck1 = new Duck();  
 duck1.set\_flyStrategy(new NoFlyStrategy());  
 System.out.print("Duck 1: ");  
 duck1.fly();  
  
 Duck duck2 = new Duck();  
 duck2.set\_flyStrategy(new FlyHighStrategy());  
 System.out.print("Duck 2: ");  
 duck2.fly();  
  
 Duck duck3 = new Duck();  
 duck3.set\_flyStrategy(new FlyLowStrategy());  
 System.out.print("Duck 3: ");  
 duck3.fly();  
 }  
}

- Ta tạm khai báo ba Duck và ta setFly cho từng loại là các loại bay mà ta muốn. Sau đó ta cho fly và kết quả:



- Các con vịt của ta đã bay đúng như định nghĩa, và bạn thấy đấy ta chỉ cần sửa các strategy implement từ IflyStrategy của chúng ta, nếu muốn thay đổi String. Không như kế thừa, ta phải đi sửa trong từng class con.

- Hoặc nếu ta có một thêm Bay lên trời bay xuống đất bay đi đâu cũng được, ta chỉ cần tạo thêm lớp FlyToSkyStrategy, FlyToAnywhereStrategy, … thì hoàn toàn code của chúng ta vẫn không bị thay đổi, và việc thêm cũng không tốn thời gian.

**Tổng kết lại:** Trên đây là ví dụ đơn giản về Strategy Pattern, trong thực tế, khi nhận một task chúng ta nên phân tích sẽ nghĩ đến các tình huống có thể diễn ra để có thể cân nhắc chúng ta có nên dùng Pattern hay không. Không nên dùng Pattern một cách tùy tiện. Nên nhớ Pattern sinh ra là để chúng ta có thể dễ dàng thay đổi maintance code, không phải làm phức tạp hóa code.