**STRATEGY PATTERN**

Để có thể hiểu rõ về Strategy Pattern, chúng ta có thể theo dõi quá trình xử lý thiết kế các đối tượng sau.

Yêu cầu đặt ra của chúng ta là thiết kế 1 class Car. Ở đây ta ví dụ có abstract class Vehicle như sau

public abstract class Vehicle

{

public Vehicle()

{

}

**public void go()**

**{**

**System.out.println(“Now I’m driving.”);**

**}**

}

Sau đó chúng ta có thêm 1 class khác StreetRacer thừa kế từ Vehicle

public class StreetRacer extends Vehicle

{

public StreetRacer()

{

}

}

Tới đây ta thấy là StreetRacer được thừa kế từ Vehicle và khi gọi hàm go() thì đối tượng StreetRacer sẽ trả về “Now I’m driving”.

Nếu chúng ta có thêm 2 class FomulaOne và Helicopter thừa kế từ class Vehicle thì hàm go() của các đối tượng này vẫn là “Now I’m driving.” Điều này nghe có vẻ không hợp lý đối với Helicopter và chúng ta cần hàm go() ở đây trả về “Now I’m flying.” cho hợp lý, việc này được giải quyết dể dàng bằng 1 hàm override trong class Helicopter.

public class Helicopter implements IFly

{

public Helicopter()

{

}

**public void go()**

**{**

**System.out.println(“Now I’m flying.”);**

**}**

}

Vấn đề được giải quyết rất dễ dàng, tuy nhiên chúng ta thử nhìn xa hơn một chút, nếu chúng ta có chục class được thừa kế từ Vehicle và trong mỗi class có hàm override riêng của nó. Tới đây mọi chuyện vẫn tốt đẹp, tuy nhiên nếu một ngày đẹp trời chúng ta mún thay đổi nội dung của nó. Gỉả sử hàm go() của Helicopter bây giờ sửa lại là “Now I’m flying fast” chúng ta sẽ mắc phải một vấn đề là phải chỉnh hết cho từng class mà ta muốn sửa nội dung, rõ ràng cách này sẽ rất dễ xảy ra sai sót khi ta phải chỉnh lại một số lượng lớn các class. Strategy Pattern sẽ giúp ta giải quyết vấn đề này.

Trước khi nói tiếp, chúng ta nên hiểu một chút về khái niệm “is-a”, từ đầu tới giờ khi nói tới thừa kế là chúng ta đang sử dụng khái niệm “is-a” Có nghĩa là Helicopter is a Vehicle, Street Racer is a Vehicle … và hàm go() của các class con cũng “is a” go() của class Vehicle. Khi sử dụng Strategy Pattern, chúng ta sẽ sử dụng khái niệm “has-a”.

Vậy bây giờ chúng ta sẽ coi “has-a” tức là has a cái gì. Nguyên tắc sử dụng Strategy Pattern như sau, chúng ta sẽ xây dựng bộ giao diện (interface) Algorithms gồm các thành phần thừa kế mà các class sẽ has-a nó, khi có bất kỳ sự thay đổi nào, ta chỉ cần thay đổi algorithms thì các hàm trong class cũng sẽ thay đổi theo, để có thể thấy rõ cách sử dụng, ta quay trở lại với ví dụ trên, bây giờ chúng ta sẽ tạo 1 interface algorithms như sau

public interface GoAlgorithm

{

public void go();

}

Sau đó, ta thêm vào các tùy chọn khác nhau cho interface này bằng cách tạo ra các class thừa kế nó.

public class GoByDrivingAlgorithm implements GoAlgorithm

{

public void go()

{

**System.out.println(“Now I’m driving.”);**

}

}

Thêm 1 Algorithms khác

public class GoByFlying implements GoAlgorithm

{

public void go() {

**System.out.println(“Now I’m flying.”);**

}

}

public class GoByFlyingFast implements GoAlgorithm

{

public void go()

{

**System.out.println(“Now I’m flying fast.”);**

}

}

Bây giờ sau khi chúng ta đã có một số Algorithms, chúng ta sẽ sử dụng nó trong các class với khái niệm “has-a” như sau, bắt đầu từ class cha

public abstract class Vehicle

{

private GoAlgorithm goAlgorithm;

public Vehicle()

{

}

public void setGoAlgorithm (GoAlgorithm algorithm)

{

goAlgorithm = algorithm;

}

public void go() {

goAlgorithm.go();

}  
}

Ta thấy ở đây class Vehicle has-a interface GoAlthrithm và thiết lập thuộc tính set interface này đồng thời hàm go() trả về kết quả gọi hàm go() của interface này. Chúng ta coi tiếp các class tiếp theo

public class StreetRacer extends Vehicle

{

**public StreetRacer()**

**{**

**setGoAlgorithm(new GoByDrivingAlgorithm());**

**}**

}

public class FormulaOne extends Vehicle

{

**public FormulaOne()**

**{**

**setGoAlgorithm(new GoByDrivingAlgorithm());**

**}**

}

public class Helicopter extends Vehicle

{

**public Helicopter()**

**{**

**setGoAlgorithm(new GoByFlyingAlgorithm());**

**}**

}

Chúng ta thấy việc thiết lập các class tiếp theo tương đối đơn giản, chúng ta chỉ cần thiết đặt lại giao diện của nó ứng với chức năng mà nó cần. Như vậy nếu chúng ta muốn nâng cấp lên thì chúng ta chỉ cần tạo thêm chức năng mới cho interface hoặc thay đổi dễ dàng trong interface. Rõ ràng kiểu Strategy Pattern rất hữu ích và tạo ra năng suất cao hơn hẳn so với cách thông thường.

Ngoài ra, khi chúng ta mún thay đổi chức năng của interface, chúng ta chỉ cần set lại thuộc tính của nó với chức năng phù hợp. Ví dụ sau mô tả RealJet thể hiện khả năng vừa chạy vừa bay …

public class RealJet

{

public static void main(String[] args)

{

Jet jet = new Jet();

jet.setGoAlgorithm(new GoByDrivingAlgorithm());

jet.go();

**jet.setGoAlgorithm(new GoByFlyingFastAlgorithm());**

**jet.go();**

**jet.setGoAlgorithm(new GoByDrivingAlgorithm());**

**jet.go();**

}

}

**Tổng kết**

Ta tổng kết 1 vài tình huống cần xài Strategy Pattern

* Khi ta mún thay đổi chức năng của nó lúc chương trình đang chạy (bằng cách set chức năng khác nhau như đã nói ở trên)

Strategy Pattern được miêu tả như.

* Tạo 1 interface (gọi là Algorithm) gồm nhiều Algorithms khác nhau bên trong nó.
* Sử dụng Algorithm này theo khái niệm “has-a”, nhớ tạo thuộc tính thích hợp để có thể thay đổi chức năng (Algorithm) giao diện.