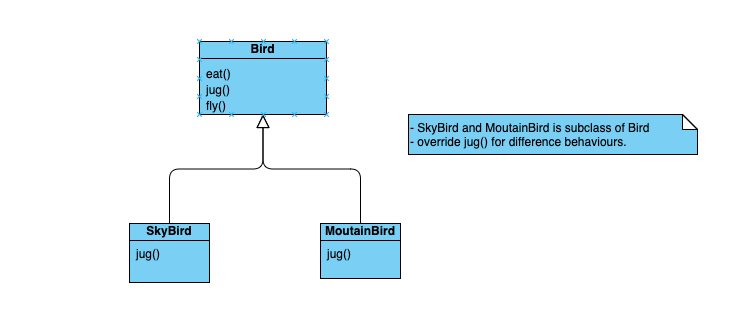
**Composition over inheritance**

1. **Vấn đề về sử dụng Inheritance**

Inheritance là 1 trong những khái niệm cốt lõi của OOP. 1 vài lợi ích mà Inheritance mang lại đó là:

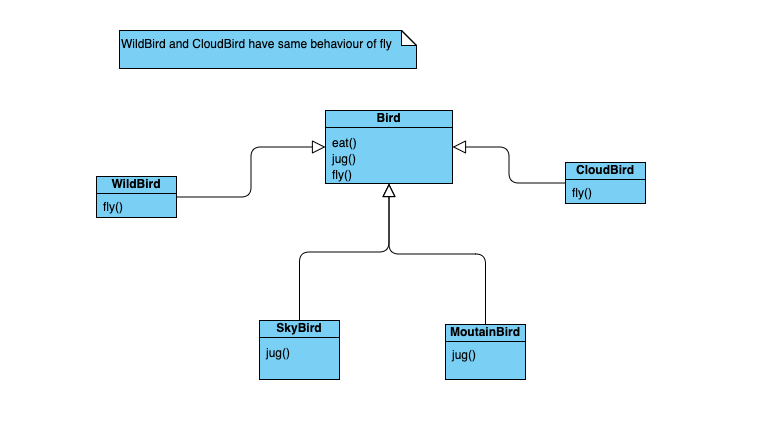
* Khả năng tái sử dụng code: Các lớp con có các properties và functions của lớp cha 🡪 Có thể giảm sự trùng lặp code giữa các lớp con bằng cách đặt các phần code bị trùng lặp vào lớp cha.
* Code dễ đọc và dễ hiểu hơn.
* Inheritance đại diện cho mối quan hệ IS-A relation ship 🡪 Các lớp con có thể thay thế cho lớp cha

Mặc dù Inheritance được sử dụng rộng rãi, nhưng liệu nó có phải là 1 ý tưởng mạnh mẽ nên áp dụng mọi lúc mọi nơi? Hãy cùng suy ngẫm về 1 ví dụ sau:



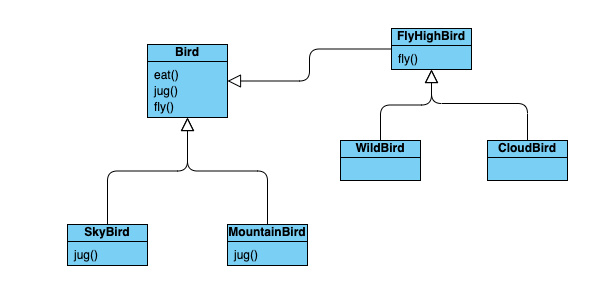
* SkyBird và MountainBird là 2 class kế thừa từ class Bird. SkyBird và MountainBird có chung hành vi eat() và fly(), do đó chúng ta đặt 2 phương thức này ở lớp Bird để tái sử dụng code
* Hành vi jug() là khác nhau nên chúng ta sẽ tự override lại ở lớp con.

Nghe có vẻ ổn. Bây giờ, khách hàng muốn chúng ta thêm 2 class là WildBird và CloudBird. Chúng có hành vi fly() khác so với lớp Bird, vì vậy chúng ta sẽ override lại hành vi fly(), từ đó có thiết kế:



* Nhưng vấn đề ở đây là WildBird and CloudBird có cùng hành vi fly(). Nếu chúng ta thiết kế như Diagram ở trên thì chúng ta đã tạo ra 1 sự trùng lặp code hành vi fly giữa lớp WildBird và CloudBird.

Bạn có thể nghĩ 1 cách để tránh trùng lặp code đó là tạo 1 class cha cho WildBird và CloudBird, và đặt hành vi fly() chung vào đó, như sau:



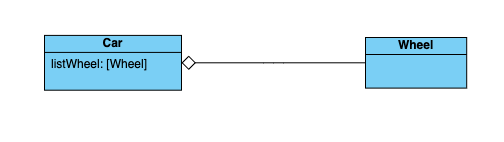
* Khá OK. Nhưng giờ khách hàng tiếp tục muốn 1 số hành vi khác như WildBird và SkyBird có chung hành vi eat() mới – khác hành vi eat của Bird, CloudBird và MoutainBird có chung hành vi jug(), …
* Nếu bạn tiếp tục cố gắng tạo ra các lớp cha mới thì bạn sẽ tạo ra 1 kiến trúc ngày càng mở rộng, càng rối cho hệ thống của bạn. Ngoài ra bạn còn phải sửa lại kiến trúc hiện có 🡪 Dẫn đến sửa lại rất nhiều class, code hiện có (Vi phạm nguyên tắc **OCP - Open/Closed Principle**).

Nhược điểm của kế thừa:

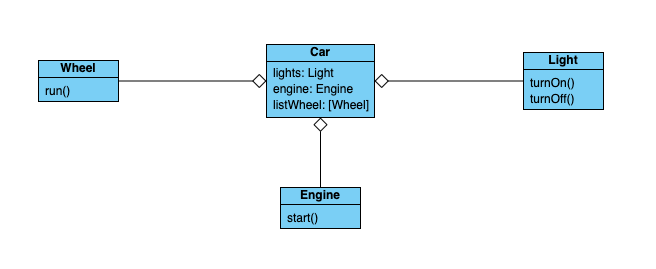
* 1 object chỉ có thể kế thừa từ 1 lớp cha.
* Dễ dàng tạo ra 1 kiến trúc lớn, phức tạp
* Thay đổi 1 function ở lớp cha 🡪 Ảnh hưởng tới toàn bộ lớp con.
* Thường thì chúng ta cố gắng nhét tất cả các function chung vào lớp cha 🡪Dẫn đến lớp cha bị phình to, đồng thời có những function của lớp cha mà lớp con này không cần dùng đến. (Điều này rất dễ thấy trong chính project hiện tại của các bạn).
* Khi mở rộng source code thường dẫn đến phải thay đổi các code hiện có.

1. **Composition là gì?**

Khác với Inheritance đại diện cho quan hệ IS-A giữa 2 class, thì Composition đại diện cho quan hệ HAS-A giữa 2 class:



* Composition có nghĩa là "thành phần". Ở ví dụ trên, Wheel là 1 thành phần của Car, nói cách khác, Car chứa 1 instance kiểu Wheel.
* Có thể hiểu Composition như việc xếp hình. Thay vì đặt các chức năng vào đối tượng gốc, bạn sẽ tách các chức năng ra thành các đối tượng riêng lẻ, sau đó ghép các thành phần đó lại để bổ sung thêm chức năng cho đối tượng gốc của bạn.



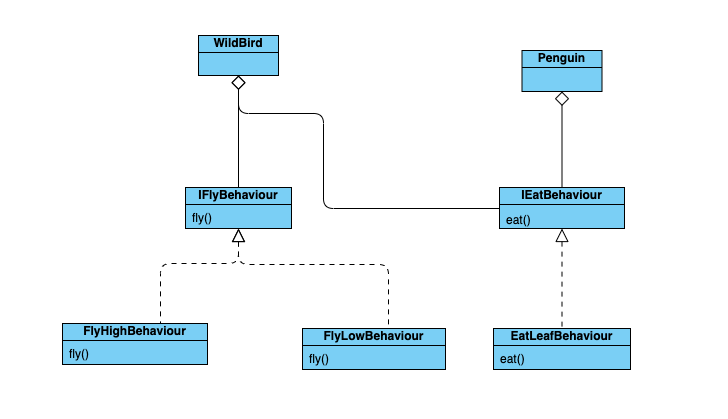
Thay vì đặt các login run, start,… vào đối tượng gốc Car thì ta sẽ tách ra thành

các thành phần riêng lẻ.

Lợi ích của Composition:

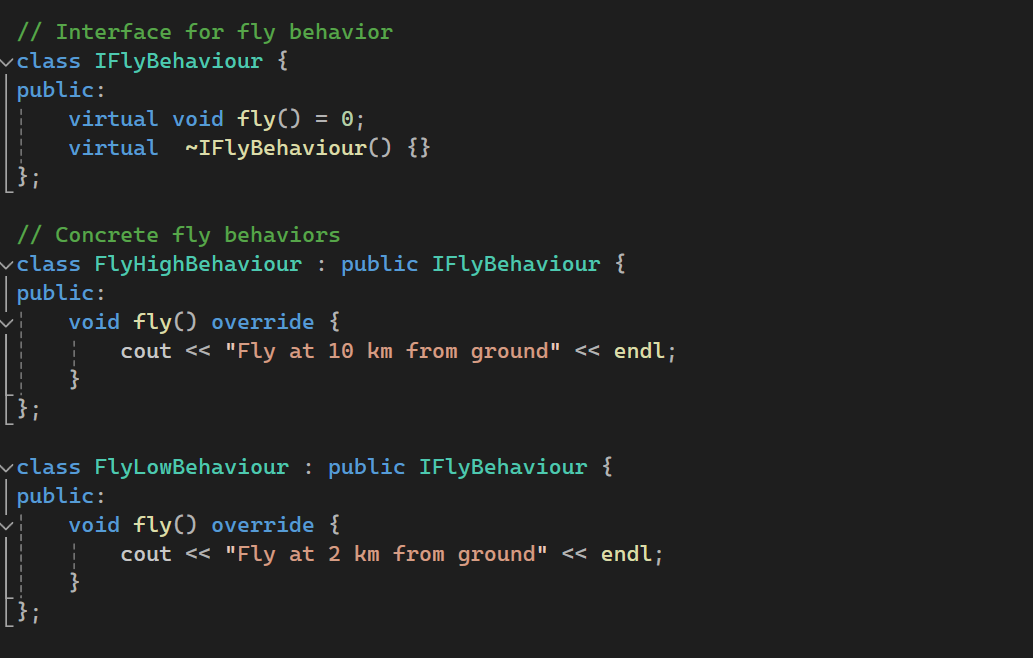
* Giải quyết được vấn đề tạo ra một hệ thống phân cấp lớn khi dùng kế thừa.
* Dễ dàng tái sử dụng code giữa các đối tượng một cách linh hoạt.
* Ví dụ: Ta có thể tái sử dụng các logic turnOn, turnOff giữa các loại xe mà không cần tạo class cha.
* Khi thêm mới/ thay đổi các hành vi của đối tượng thì không cần phải thay đổi quá nhiều code hiện có, chỉ cần viết thêm code và thay thế các hành vi mới vào hành vi hiện tại. (Đảm bảo nguyên lí **SRP - Single Responsibility Principle** và **OCP - Open/Closed Principle**).
* Để tái sử dụng code mà có liên quan đến UI kéo thả thì sử dụng Inheritance là bất khả thi, còn Composition vẫn có thể giải quyết được

1. **Áp dụng Composition để giải quyết vấn đề ở đầu bài viết.**



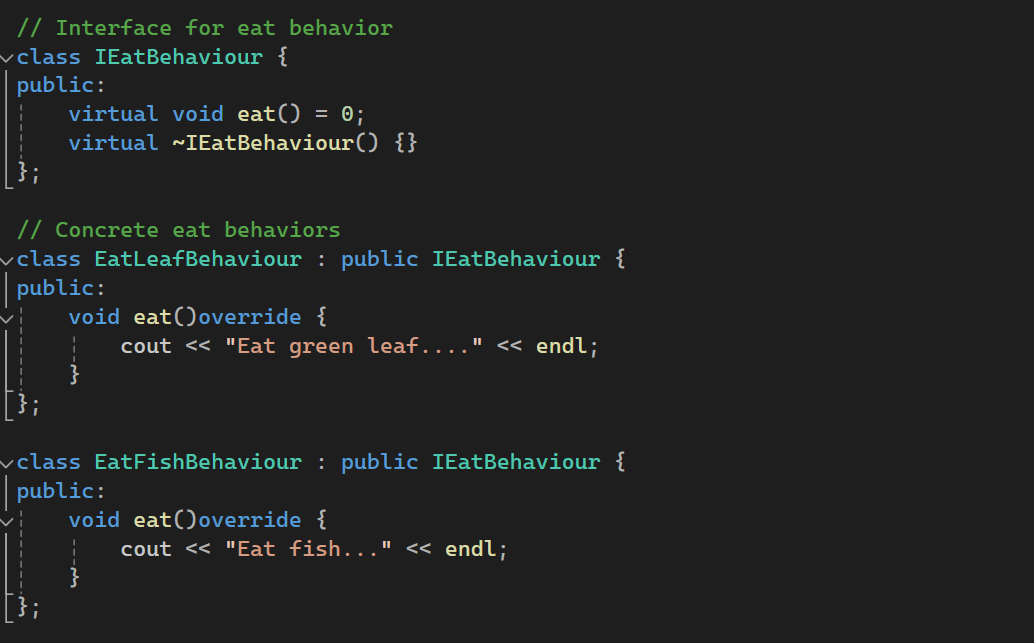
* Tạo ra các interface cho các hành vi fly và eat lần lượt là **IFlyBehaviour** và **IEatBehaviour**.
* Các class **FlyHighBehaviour** và **FlyLowBehaviour** tuân thủ theo giao thức **IFlyBehaviour**, chúng sẽ khai báo lại phương thức fly() với các hành vi riêng. Ngoài ra có thể tạo thêm các class có phương thức fly khác tùy theo yêu cầu bài toán.
* Class **EatLeafBehaviour** tuân thủ theo giao thức **IEatBehaviour**, khai báo lại phương thức eat riêng. Ngoài ra có thể tạo thêm các class có phương thức eat khác tùy theo yêu cầu bài toán.
* WildBird vừa có thể bay và ăn nên nó sẽ chứa 2 instance kiểu **IFlyBehaviour** và **IEatBehaviour**; Penguin không thể bay nên chỉ cần chứa 1 instance kiểu **IEatBehaviour**.

1. **Code triển khai**



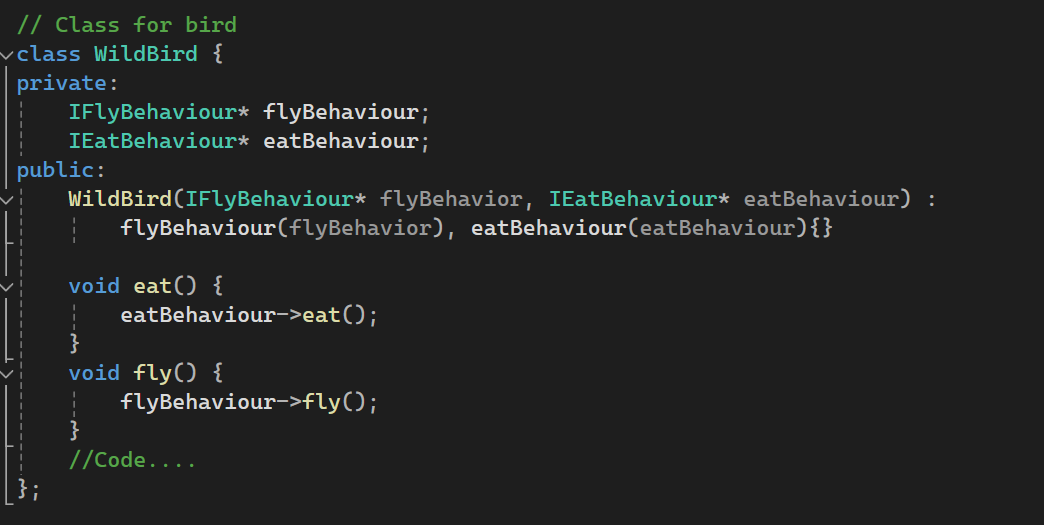
**Khi cần thêm các hành vi fly mới, chỉ cần viết thêm các class kế thừa**

**IFlyBehaviour.**

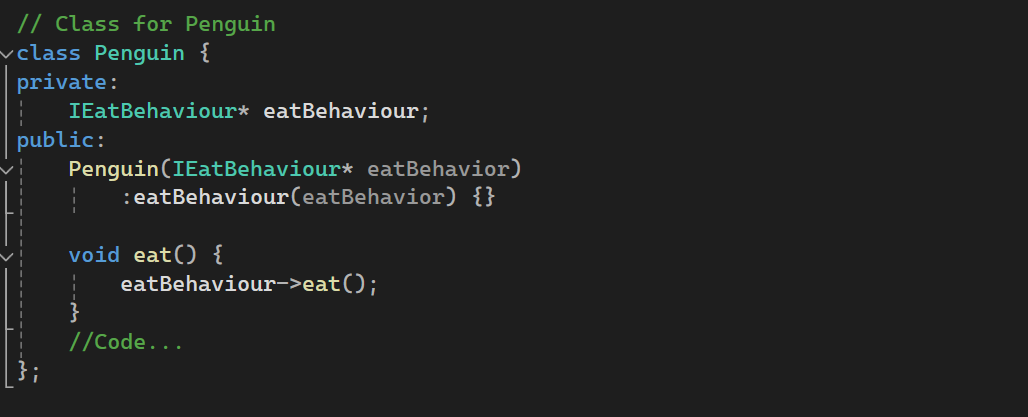


**Khi cần thêm các hành vi eat mới, chỉ cần viết thêm các class kế thừa**

**IEatBehaviour.**



**Ví dụ tạo 2 đối tượng WildBird có hành vi Fly khác nhau. Các loài chim với các hành vi fly/eat khác thì chỉ cần khởi tạo class chim đó với các hành vi mong muốn**



**Khởi tạo đối tượng Penguin không có hành vi bay, chỉ có hành vi ăn**