# Object Oriented Programing

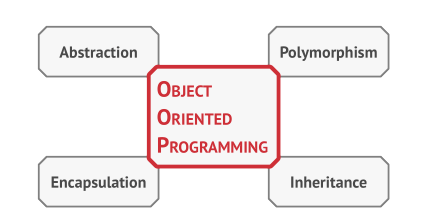
## CLASS

|  |  |
| --- | --- |
|  | **UML Summary**  + Class ( class of instances )  + Fields ( State )  + pulic  + private  + Methods ( Behavior )  Mỗi một lớp sẽ có |
|  | Mỗi một instance của class sẽ chứa các thuộc tính **standard attributes**: name, sex, age, weight, color, favorite food, etc. được định nghĩa ở trong **CLASS CAT ( CAT FIELDS )**  Bên cạnh đó mỗi instance đều có cùng hành vi được định nghĩa ở **CLASS CAT (Methods)** |

## Class hierarchies ( phân lớp )

|  |  |
| --- | --- |
|  | **UML Summary**  + Super Class  + Fields ( State )  + pulic  + private  + Methods ( Behavior )  SuperClass là lớp cha có những thành phần chung nhất để cho SubClass (lớp con) kế thừa. Trong UML SubClass có mũi tên hình tam giác trống trỏ đến SuperClass ngầm hiểu là kế thừa từ SuperClass. Cũng như vậy mà nét đứt thì sẽ chỉ ra rằng subclass có thể overelap |
|  | Subclasses can override the behavior of methods that they inherit from parent classes. A subclass can either completely replace the default behavior or just enhance it with some extra stuff.  Các lớp con có thể ghi đè hành vi của các phương thức mà chúng kế thừa từ các lớp cha. Một lớp con có thể thay thế hoàn toàn hành vi mặc định hoặc chỉ nâng cao nó bằng một số công cụ bổ sung. |

## Pillars of OOP - Main Concept



|  |  |
| --- | --- |
| **Abstraction ( ap ‘trak sn ) – Tính trừ tượng**  Lớp máy bay có thể tồn tại ở 2 trường hợp sau đây:   * Mô phỏng chuyển bay   Tốc độ  Độ cao  Góc cuộn  Góc sân  Góc nghiên   * Ứng dụng đặt vé máy bay   Chỗ ngồi  Chỗ ngồi chống  Trừu tượng là một mô hình của một đối tượng hoặc hiện tượng trong thế giới thực, giới hạn trong một bối cảnh cụ thể, thể hiện tất cả các chi tiết liên quan đến bối cảnh này với độ chính xác cao và bỏ qua tất cả các phần còn lại. | **Abstraction**    Abstraction is a model of a real-world object or phenomenon, limited to a specific context, which represents all details relevant to this context with high accuracy and omits all the rest.. |
| **Encapsulation (in kap su ‘lei sn) Đóng gói**  Tính đóng gói là khả năng của một đối tượng che giấu các phần trạng thái và hành vi của nó khỏi các đối tượng khác, chỉ để lộ một giao diện giới hạn cho phần còn lại của chương trình.  Thực tế ***Interface*** là từ viết tắt của một phần công khai của đối tượng. | **Encapsulation**  Encapsulation is the ability of an object to hide parts of its state and behaviors from other objects, exposing only a limit­ed interface to the rest of the program. |
| **Inheritance**  In most programming languages a subclass can extend only one superclass. On the other hand, any class can implement several interfaces at the same time. But, as I mentioned before, if a superclass implements an interface, all of its subclasses must also implement it.  Một subclass chỉ được kế thừa từ một lớp cha, mặt khác các class có thể triển khi nhiều interface cùng một lúc. Nhưng lếu một superclass triển khai một interface thì tất cả lớp con của nó cũng phải triển khai các interface đó | **Inheritance** |
| **Polymorphism ( po li ˈmɔr fɪ zm - Tính đa hình)**  Most Animals can make sounds. We can guess that all subclasses will need to override the base makeSound method so each subclass can emit ( phát ra ) the correct sound. Therefore, we can **declare it abstract right away**. This lets us **omit any default implementation** of the method in the superclass, but force all **subclasses to come up with their own.**  Tất cả các subclass của superclass Animal đều có thể phát ra âm thanh và chúng cho ra kết quả khác nhau. Vì vậy chúng ta có thể **khai báo lớp trù tượng ngay lập tức .** Nó cho phép **bỏ qua việc triển khai trong lớp cha,** nhưng buộc tất cả các **lớp con phải đưa ra phương thức riêng của chúng** | **Polymorphism** |

## Relations between objects

UML Relations between objects

# Software design principles

<https://viblo.asia/p/mot-so-design-principles-trong-lap-trinh-ma-ban-nen-biet-eW65GvJOlDO>

Design principle các nguyên tắc thiết kế phần mềm, đưa ra những gợi ý, lời khuyên hướng dẫn ( Low level )

Design pattern đưa cho chúng ta một khuôn mẫu trong một ngữ cảnh thực tế ( High Level )

* Thường nhầm lẫn với design pattern
* Các Design pattern tốt thường sẽ thỏa mãn các Design Principles



## DRY | WET

**DRY : Don’t Repeat your-self**

Đừng lặp lại chính bạn , mang hàm ý đừng lặp lại những đoạn code ở nhiều nơi khác nhau. DRY không chỉ ứng dụng trong việc code, mà còn cả trong việc viết documentation, hay thiết kế database schemas ...

**WET :** DRY Every time, Vi phạm DRY thì sẽ WET

## KISS

**KISS: Keep It Simple, Stupid**

KISS mang ý nghĩa rất giống với một số câu nói nổi tiếng khác như "Simple is the best", "Simplicity is the ultimate sophistication", "Make Simple Tasks Simple" ... chúng đều đề cao tính "đơn giản" trong việc giải quyết vấn đề.

## YAGNI

**YAGNI** là viết tắt của từ **You aren't gonna need it**, tức bạn có thể sẽ không cần đến nó đâu. **YAGNI** là một principle được giới thiệu trong extreme programming

# Design Pattern (GOF)

## Design pattern

Design patterns are typical solutions to commonly occurring problems in software design. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code. Patterns are often confused with algorithms, because both concepts describe typical solutions to some known problems. The pattern is not a specific piece of code, but a general concept for solving a particular problem. You can follow the pattern details and implement a solution that suits the realities of your own program.

* Pattern là những mẫu thiết kế được tạo ra sẵn nhằm giải quyết các vấn đề phổ biến trong lập trình.
* Các pattern thường bị nhầm lẫn với thuật toán vì cả 2 khái niệm trên đều mô tả giải pháp điển hình cho một số vấn đề đã biết.
* Patterns không phải những đoạn code cụ thể. Bạn chỉ có thể làm theo và truển khai một giải pháp phù hợp với chương trình thực tế của bạn

An algorithm is like a cooking recipe: both have clear steps to achieve a goal. On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

## Facade pattern [fa sa:d]

<https://github.com/doctor-blue/design-patterns/tree/master/Structural/Facade>

* Mỗi một hàm chỉ làm một việc duy nhất. Nguyên tắc S trong solid ( Single Responsibility Principles)
* Gói một subsystem vào một class và cung cấp interface để client sử dụng đơn giản hơn
* Trong trường hợp bạn có nhiều hệ thống con và mỗi lần gọi tới những hệ thống này lại quá sức phức tạp và tốn nhiều thời gian. Vậy **Facade** sẽ giúp đỡ bạn trong việc này. Cũng giống như [**Adapter**](https://github.com/doctor-blue/design-patterns/tree/master/Structural/Adapter)**Facade** cung cấp cho chúng ta một lớp để làm chung gian giữa hai thành phần, nhưng khác nhau ở chỗ Adapter thì giúp chúng ta làm việc giữa hai mã nguồn, còn đối với Facade thì thì sẽ là thành phần bao bọc che giấu đi các hệ thống con để giúp giao tiếp với thành phần khác dễ dàng hơn.
* Như vậy ta có thể hiệu Facade sẽ cung cấp một giao diện cho việc giao tiếp giữa các hệ thống con, giúp người dùng có thể giao tiếp với chúng một cách dễ dàng. Facade giúp che dấu đi xử lý logic phức tạp bên trong hệ thống con.

## Strategy pattern

* Để giải quyết một việc (tùy thuộc arguments) thì sẽ cho một kết quả trong một lần thực thi ( Switch case )

1. a