



OOP

CHO NGƯỜI ĐI LÀM

- Nguyễn Thế Huy
- huynt57@gmail.com

Lời mở đầu

Giới thiệu về lập trình hướng đối tượng

Lập trình hướng đối tượng là gì

Các trụ cột của lập trình hướng đối tượng

Trừu tượng (Abstraction)

Đóng gói (Encapsulation)

Kế thừa (Inheritance)

Đa hình (Polymorphism)

Giới thiệu về mẫu thiết kế (Design Pattern)

Mẫu Thiết Kế (Design Pattern) là gì?

Tại Sao Tôi Nên Học Các Mẫu Thiết Kế?

Các nguyên tắc trong thiết kế phần mềm

Thiết kế tốt có đặc điểm gì

Tái sử dụng code (Code Reuse)

Khả năng mở rộng (Extensibility) và tính linh hoạt (Flexibility)

Các nguyên tắc thiết kế

Đóng gói

Tập trung vào trừu tượng

Ưu tiên thành phần hơn kế thừa (Composition over Inheritance)

SOLID

Nguyên Tắc Trách Nhiệm Duy Nhất (Single Responsibility Principle)

Nguyên Tắc Mở/Đóng (Open/Closed Principle)

Nguyên Tắc Thay Thế Liskov (Liskov Substitution Principle)

Nguyên Tắc Phân Tách Giao Diện (Interface Segregation Principle)

Nguyên Tắc Đảo Ngược Phụ Thuộc (Dependency Inversion Principle)

Inversion of Control (IoC)

Các Design Pattern quan trọng

Nhóm khởi tạo

Factory Pattern

Dấu hiệu nhận biết:

Ví dụ

Builder

Dấu hiệu nhận biết:

Ví dụ

Singleton

Dấu hiệu nhận biết:

Ví dụ:

Nhóm cấu trúc

Adapter

Dấu hiệu nhận biết:

Ví dụ:

Decorator

Dấu hiệu nhận biết:

Ví dụ:

Nhóm hành vi

Chain of Responsibility

Dấu hiệu nhận biết:

Ví dụ:

Command

Dấu hiệu nhận biết:

Ví dụ:

Observer

Dấu hiệu nhận biết:

Ví dụ:

State

Dấu hiệu nhận biết:

Ví dụ:

Strategy

Dấu hiệu nhận biết:

Ví dụ:

Template Method

Dấu hiệu nhận biết:

Ví dụ:

Bonus: Dependency Injection

Lời kết

Lời mở đầu

Chào bạn, và cảm ơn bạn đã đọc ebook này của mình.

Mình là Huy, hiện tại mình là Technical Leader tại Công ty Cổ phần Giao hàng Tiết kiệm. Ngoài ra, mình còn là Moderator của Group Laravel Việt Nam, một cộng đồng Laravel lớn trên Facebook với gần 30.000 thành viên.

Ebook này của mình nhằm củng cố các kiến thức cơ bản về Lập trình hướng đối tượng (OOP) cho mọi người, phân tích các nguyên lý cốt lõi khi thiết kế và các Design Pattern phổ biến. Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức trong ebook này sẽ giúp bạn rất nhiều trong việc chuẩn bị nền tảng kiến thức về OOP.

Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook này.

Giới thiệu về lập trình hướng đối tượng

Lập trình hướng đối tượng là gì

Lập trình hướng đối tượng (OOP) là một mô hình dựa trên khái niệm gói gọn dữ liệu và hành vi liên quan đến dữ liệu đó vào các gói đặc biệt gọi là đối tượng (object), được xây dựng từ một tập các "bản thiết kế" do lập trình viên định nghĩa, gọi là lớp (class). OOP tập trung vào cách mà các đối tượng giao tiếp với nhau để giải quyết các bài toán nghiệp vụ.

Một điều khác biệt quan trọng khi làm việc với OOP, đó là quan tâm đến trạng thái (State) của đối tượng và dữ liệu trong quá trình chương trình hoạt động, cũng như cách mà các đối tượng đó tương tác (Sending Message). Trong các section bên dưới, bạn sẽ dần hiểu thêm về cách triển khai OOP phù hợp

Các trụ cột của lập trình hướng đối tượng

Mô hình OOP được đại diện bởi bốn trụ cột đặc trưng không thể thiếu:

Đóng gói - Trừu tượng - Kế thừa - Đa hình.



OOP cho người đi làm

Bạn có thể tưởng tượng: Đóng gói (Encapsulation) như một chiếc khiên bảo vệ, Trừu tượng (Abstraction) như đám mây trên bầu trời, Kế thừa (Inheritance) là một cây gia phả, và Đa hình (Polymorphism) là một chú tắc kè hoa. Rất thú vị phải không 😊

Trừu tượng (Abstraction)

Khi làm việc với OOP, chúng ta luôn cố gắng mô tả một đối tượng tồn tại ngoài thực tế. Một chú chó có bốn chân, một chiếc xe ô tô có bốn bánh. Tuy nhiên, không phải lúc nào bạn cũng cần mọi chi tiết của đối tượng: có khi bạn quan tâm đến tốc độ của chiếc xe, nhưng cũng có khi bạn lại chỉ cần quan tâm đến số chỗ ngồi trên xe đó.

Trừu tượng là một mô hình của một đối tượng hoặc hiện tượng trong thế giới thực, giới hạn trong một ngữ cảnh cụ thể, đại diện cho tất cả các chi tiết liên quan đến ngữ cảnh này với độ chính xác cao và bỏ qua tất cả các phần còn lại.

Ví dụ kinh điển sau về Animal, Cat, Dog sẽ giúp bạn dễ dàng hiểu về trừu tượng

```
<?php

// Định nghĩa một Lớp trừu tượng Animal
abstract class Animal {
    // Phương thức trừu tượng makeSound không có triển khai
    abstract public function makeSound();

    // Phương thức thông thường
    public function move() {
        echo "Động vật đang di chuyển\n";
    }
}

// Lớp Dog kế thừa từ Lớp trừu tượng Animal
class Dog extends Animal {
    // Triển khai phương thức trừu tượng makeSound
    public function makeSound() {
        echo "Gâu gâu\n";
    }
}
```

```
}

// Lớp Cat kế thừa từ Lớp trừu tượng Animal
class Cat extends Animal {
    // Triển khai phương thức trừu tượng makeSound
    public function makeSound() {
        echo "Meo meo\n";
    }
}

// Sử dụng các Lớp con
$dog = new Dog();
$cat = new Cat();

$dog->makeSound(); // Output: Gâu gâu
$dog->move();       // Output: Động vật đang di chuyển

$cat->makeSound(); // Output: Meo meo
$cat->move();       // Output: Động vật đang di chuyển

?>
```

Lớp trừu tượng Animal:

- Định nghĩa một phương thức trừu tượng makeSound mà không có triển khai.
- Định nghĩa một phương thức thông thường move có triển khai cụ thể.

Lớp Dog và Cat:

- Kế thừa từ lớp trừu tượng Animal.
- Cung cấp triển khai cụ thể cho phương thức trừu tượng makeSound.

Đóng gói (Encapsulation)

Đặc trưng này của OOP khiến nó trở nên gần gũi hơn với thế giới thật. Trong thực tế, bạn có thể thao tác với hầu hết các loại đồ vật (đối tượng) mà không cần biết bên trong nó có gì. Ví dụ: bật một chiếc TV, bật một chiếc Laptop. Những thứ chi tiết như các vi mạch, các thiết kế phức tạp đều được giấu một cách gọn gàng bên trong lớp vỏ. Bạn chỉ có một lớp giao diện (interface) đơn giản để tương tác với đối tượng: một cái nút bấm, một cái công tắc chẳng hạn. Điều này gợi ý về việc “đóng gói” code của bạn vào đối tượng, và chỉ “chia” ra các “phương thức” cần thiết để tương tác với đối tượng đó.

Trong OOP chúng ta thường sử dụng các từ khoá như “public”, “protected”, “private” để chỉ định mức độ đóng gói

- Public cho phép các thuộc tính được sử dụng thoải mái bởi các lớp khác.
- Protected chỉ cho phép các lớp con của lớp cha được phép sử dụng
- Private chỉ cho phép các method thuộc class đó được phép truy cập (Các lớp con cũng không được phép)

Cùng xem ví dụ dưới đây nhé:

```
<?php
```

```
class Person {  
    // Các thuộc tính private chỉ có thể được truy cập trong class này  
    private $name;  
    private $age;  
  
    // Constructor để khởi tạo đối tượng  
    public function __construct($name, $age) {  
        $this->name = $name;  
        $this->age = $age;  
    }  
  
    // Phương thức public để lấy giá trị của thuộc tính name  
    public function getName() {
```

```
        return $this->name;
    }

    // Phương thức public để đặt giá trị của thuộc tính name
    public function setName($name) {
        $this->name = $name;
    }

    // Phương thức public để lấy giá trị của thuộc tính age
    public function getAge() {
        return $this->age;
    }

    // Phương thức public để đặt giá trị của thuộc tính age
    public function setAge($age) {
        if ($age > 0) { // Đảm bảo tuổi phải là số dương
            $this->age = $age;
        }
    }
}

// Tạo một đối tượng Person
$person = new Person("John Doe", 30);

// Truy cập và thay đổi các thuộc tính thông qua các phương thức
echo "Name: " . $person->getName() . "\n"; // Output: Name: John Doe
echo "Age: " . $person->getAge() . "\n";    // Output: Age: 30

// Đặt lại giá trị cho thuộc tính name và age
```

```
$person->setName("Jane Doe");
$person->setAge(25);

echo "Updated Name: " . $person->getName() . "\n"; // Output: Updated Name: Jane
Doe
echo "Updated Age: " . $person->getAge() . "\n";    // Output: Updated Age: 25

// Thử đặt tuổi không hợp lệ
$person->setAge(-5);
echo "Age after invalid update: " . $person->getAge() . "\n"; // Output: Age
after invalid update: 25

?>
```

Kế thừa (Inheritance)

Kế thừa là khả năng xây dựng các class mới dựa trên các class đã tồn tại. Lợi ích chính của kế thừa là tái sử dụng code. Nếu bạn chỉ muốn tạo một class có một số hành vi khác với class hiện tại, sử dụng kế thừa giúp bạn không cần phải sao chép code của class cha. Thay vào đó, bạn mở rộng class hiện tại, đưa thêm các chức năng bổ sung ở class con. Class con sẽ được kế thừa các phương thức của lớp cha.

Đối với kế thừa, bạn phải đảm bảo rằng class con sẽ có những khả năng giống hệt class cha. Nếu ở class cha có các phương thức trừu tượng (abstract method), bạn sẽ vẫn phải triển khai nó ở class con, kể cả bạn không cần đến nó. Kế thừa có một rủi ro rất lớn, là bạn có thể vô tình khiến cho class con hoạt động không giống như class cha (vi phạm nguyên tắc thay thế Liskov, mình sẽ bàn luận nó sâu hơn ở section SOLID bên dưới). Điều này dẫn tới bug vô tình trong ứng dụng mà bạn khó lường trước, và nó cũng dẫn tới một nguyên tắc quan trọng nữa khi lập trình OOP: Composition over Inheritance (Ưu tiên sử dụng thành phần thay vì kế thừa).

Tiếp tục là một ví dụ đơn giản về Animal, Cat, Dog để minh họa cho kế thừa:

```
<?php

// Class cha
class Animal {
    // Thuộc tính chung cho tất cả các động vật
    protected $name;

    // Constructor để khởi tạo tên động vật
    public function __construct($name) {
        $this->name = $name;
    }

    // Phương thức chung cho tất cả các động vật
    public function eat() {
        echo $this->name . " đang ăn\n";
    }
}

// Clas con Dog kế thừa từ Lớp cha Animal
class Dog extends Animal {
    // Phương thức đặc trưng cho Dog
    public function bark() {
        echo $this->name . " sủa: Gâu gâu\n";
    }
}

// Class con Cat kế thừa từ Lớp cha Animal
class Cat extends Animal {
    // Phương thức đặc trưng cho Cat
```

```
public function meow() {  
    echo $this->name . " kêu: Meo meo\n";  
}  
}  
  
// Sử dụng các class con  
$dog = new Dog("Chó");  
$cat = new Cat("Mèo");  
  
$dog->eat(); // Output: Chó đang ăn  
$dog->bark(); // Output: Chó sủa: Gâu gâu  
  
$cat->eat(); // Output: Mèo đang ăn  
$cat->meow(); // Output: Mèo kêu: Meo meo  
  
?>
```

Đa hình (Polymorphism)

Hãy xem một ví dụ về ô tô. Chúng ta có thể chắc chắn một điều rằng, mọi chiếc ô tô đều có thể di chuyển. Vì vậy khi thiết kế, mình sẽ đặt phương thức cơ bản “move” này lên class cha, khai báo nó là trừu tượng, và buộc các class con tự định nghĩa cách thức di chuyển của nó. Một chiếc siêu xe hoặc một chiếc xe gia đình đều có thể di chuyển, nhưng kết quả và cách thức có thể là khác nhau: siêu xe có thể đi rất nhanh, nhưng xóc hơn một chiếc xe gia đình.

Giờ hãy tưởng tượng bạn đi lạc vào bãi đỗ xe, và có thể chọn bất kỳ chiếc xe nào trước mắt để lái đi. Bạn có thể không biết chắc chắn chiếc xe trước mặt là xe gì, nhưng bạn vẫn hoàn toàn có thể khiến chiếc xe di chuyển: cách thức di chuyển được định nghĩa cụ thể theo từng chiếc xe.

Trong quá trình hoạt động, ứng dụng của bạn có thể không biết chính xác đối tượng Car là gì. Nhưng, nhờ vào cơ chế đa hình, ứng dụng của bạn có thể tìm được chính xác class con của đối

tượng Car đó, và chạy hành vi thích hợp. Đa hình là khả năng của một chương trình để phát hiện lớp thực của một đối tượng và gọi triển khai của nó ngay cả khi kiểu thực sự của nó không được biết trong ngữ cảnh hiện tại. Các đối tượng thuộc class con có thể được sử dụng một cách linh hoạt, miễn là nó đảm bảo các hành vi giống với class cha: Nếu có một class Car mà không thể “move”, chúng ta sẽ không xem xét nó là một chiếc ô tô thông thường nữa mà cần báo lỗi.

```
<?php

// Lớp cha
class Car {
    // Phương thức chung cho tất cả các Loại xe
    public function move() {
        echo "Xe đang di chuyển\n";
    }
}

// Lớp con Sedan kế thừa từ Lớp cha Car
class Sedan extends Car {
    // Triển khai lại phương thức move cho Sedan
    public function move() {
        echo "Sedan đang chạy trên đường\n";
    }
}

// Lớp con SUV kế thừa từ Lớp cha Car
class SUV extends Car {
    // Triển khai lại phương thức move cho SUV
    public function move() {
        echo "SUV đang chạy trên địa hình khó khăn\n";
    }
}
```

```
// Lớp con SportsCar kế thừa từ Lớp cha Car
class SportsCar extends Car {
    // Triển khai lại phương thức move cho SportsCar
    public function move() {
        echo "SportsCar đang chạy ở tốc độ cao\n";
    }
}

// Hàm để kiểm tra tính đa hình
function testCarMovement(Car $car) {
    $car->move();
}

// Sử dụng các Lớp con
$sedan = new Sedan();
$suv = new SUV();
$sportsCar = new SportsCar();

testCarMovement($sedan);           // Output: Sedan đang chạy trên đường
testCarMovement($suv);             // Output: SUV đang chạy trên địa hình khó khăn
testCarMovement($sportsCar);       // Output: SportsCar đang chạy ở tốc độ cao

?>
```

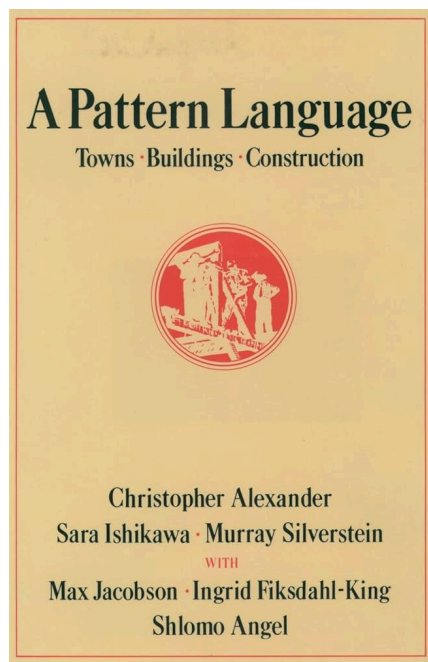
Giới thiệu về mẫu thiết kế (Design Pattern)

Mẫu Thiết Kế (Design Pattern) là gì?

Mẫu thiết kế (Design Pattern hoặc DP) là những giải pháp điển hình cho các vấn đề thường gặp trong thiết kế phần mềm. Chúng giống như các bản thiết kế sẵn mà bạn có thể tùy chỉnh để giải quyết một vấn đề thiết kế lặp đi lặp lại trong code của bạn.

Khác với các thư viện, bạn không thể cứ copy DP về và “paste” vào ứng dụng của mình. Hãy nhớ, DP cho bạn tư tưởng chung để giải quyết vấn đề. Bạn có thể tuân theo các chi tiết hay khuyến cáo trong DP, từ đó tìm ra một giải pháp phù hợp với ứng dụng của mình. DP không mô tả chính xác các bước mà bạn cần thực hiện: Nó cho bạn một giải pháp chung về thiết kế, đầu vào và đầu ra. Phần còn lại, tùy thuộc vào cách mà bạn triển khai.

Fun fact: Design Pattern không phải khái niệm độc quyền xuất hiện trong thiết kế phần mềm. Lần đầu tiên nó được đề cập trong cuốn “**A Pattern Language: Towns, Buildings, Construction**” của Christopher Alexander - một nhà kiến trúc sư và cũng là nhà khoa học máy tính lỗi lạc. Cuốn sách này mô tả các kỹ thuật áp dụng Patterns trong kỹ thuật xây dựng. Nó đã trở thành tiền đề và niềm cảm hứng để các tác giả về sau phát triển các Design Pattern cho lập trình phần mềm.



Nguyễn Thế Huy

Tại sao tôi nên học các mẫu thiết kế?

Bạn có thể làm việc như một lập trình viên trong nhiều năm mà không cần biết đến một mẫu thiết kế nào. Bản thân mình cũng đã từng như vậy. Tuy nhiên, ngay cả trong trường hợp đó, bạn có thể đang triển khai một số mẫu mà không nhận ra. Vậy tại sao bạn nên dành thời gian học chúng?



Mẫu thiết kế là bộ công cụ của các giải pháp đã được kiểm chứng cho các vấn đề phổ biến trong thiết kế phần mềm. Ngay cả khi bạn không gặp phải những vấn đề này, việc hiểu các mẫu thiết kế vẫn hữu ích vì nó giúp bạn học cách giải quyết mọi loại vấn đề bằng cách sử dụng các nguyên tắc của thiết kế hướng đối tượng.

Mẫu thiết kế xác định một ngôn ngữ chung mà bạn và đồng nghiệp có thể sử dụng để giao tiếp hiệu quả. Mỗi khi cần tới giải pháp, bạn chỉ cần nói: “Chỗ này nên sử dụng Strategy”, hoặc “Chỗ kia nên là Observer”. Nếu bạn và đồng nghiệp đã thông thạo các Pattern này, thì sẽ tiết kiệm được rất nhiều thời gian giải thích và mô tả solution.

Các nguyên tắc trong thiết kế phần mềm

Thiết kế tốt có đặc điểm gì

Tái sử dụng code (Code Reuse)

Tái sử dụng được code là chìa khoá quan trọng cho một ứng dụng thành công. Mình hay nói đùa rằng “Thời điểm bạn bắt đầu copy code là thời điểm mà ứng dụng của bạn nên được viết lại”. Code không thể tái sử dụng khiến ứng dụng trở nên cồng kềnh, gia tăng cả về chi phí nhân sự và thời gian phát triển. Chưa kể, nó tiềm ẩn vô số lỗi có thể phát sinh.

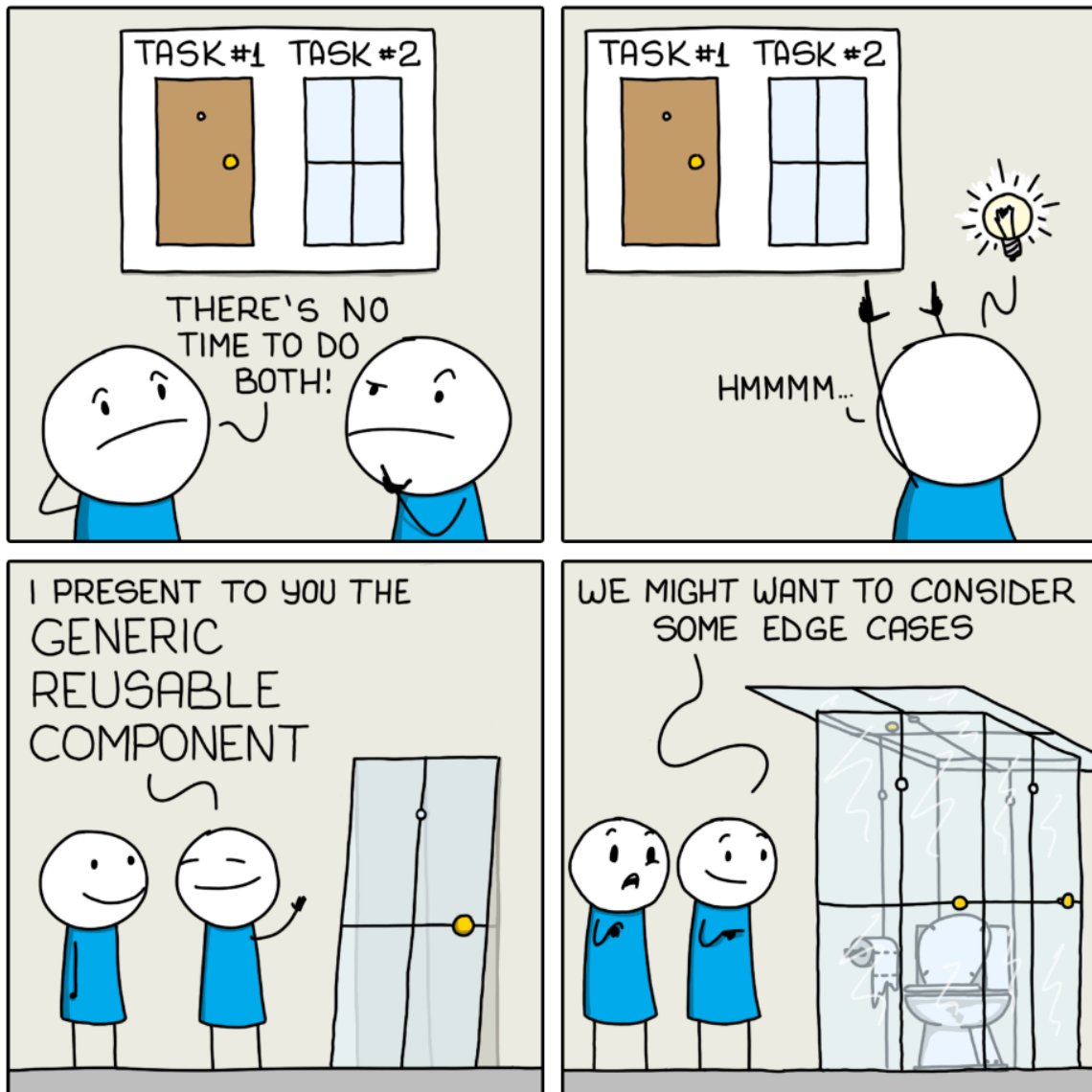
Có một câu nói đùa nữa cũng khá nổi tiếng: “Đừng chế lại cái bánh xe”. Ý tưởng rất đơn giản: thay vì code lại mọi thứ, hãy thử tìm xem có thư viện, hay đoạn code nào có thể sử dụng lại được hay không ?

Nói thì dễ, nhưng việc tái sử dụng code không hề đơn giản chút nào. Các class bị ràng buộc chặt chẽ với nhau mà không qua các lớp trừu tượng hay việc không phân chia trách nhiệm rõ ràng cho các đối tượng (Hiểu nôm na là một class mấy nghìn dòng code). Những điều này là biểu hiện của một thiết kế chưa tốt, và nó hạn chế tính linh hoạt và tái sử dụng của code.

Sử dụng hợp lý các Design Pattern giúp code của bạn dễ dàng tái sử dụng hơn. Mình sẽ bàn luận thêm ở section “Các Design Pattern quan trọng” nhé.

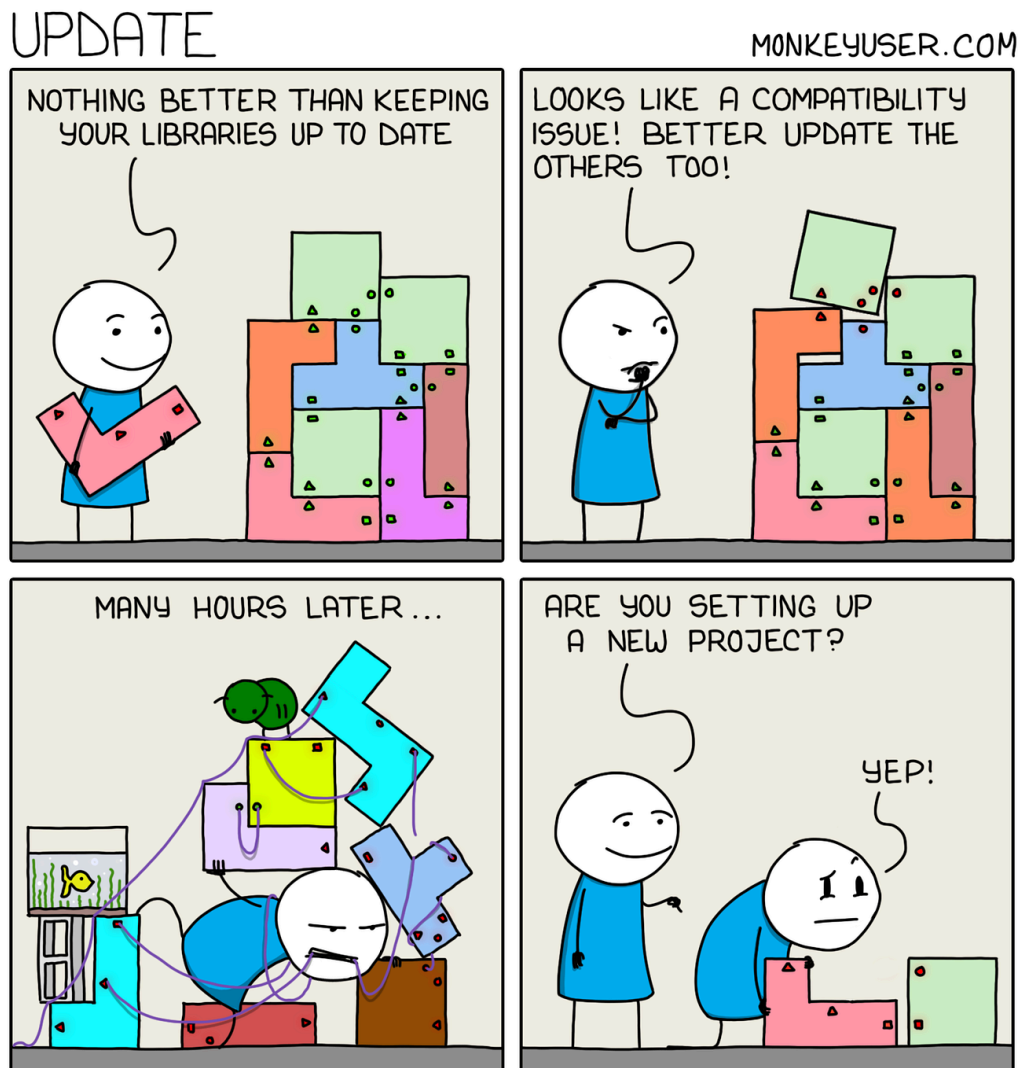
REUSABLE COMPONENTS

MONKEYUSER.COM



Khả năng mở rộng (Extensibility) và tính linh hoạt (Flexibility)

“Thay đổi là thứ cố định duy nhất”. Nếu bạn đã đọc cuốn “Thiết kế hướng nghiệp vụ với Laravel của mình”, bạn sẽ hiểu phần nào áp lực của một ứng dụng phải chịu khi cần “chống chọi” với những thay đổi liên tục từ nghiệp vụ. Những câu chuyện về những đòi hỏi của khách hàng, thậm chí có thể khác hoàn toàn những thiết kế ban đầu, chắc chắn là không thiếu. Vì thế, khả năng mở rộng linh hoạt của thiết kế là cực kỳ quan trọng. Điều này thậm chí còn khó hơn cả tái sử dụng code. Tuy nhiên nhờ có các nguyên tắc như SOLID và các Design Pattern, ứng dụng của chúng ta có thể tăng được khả năng đáp ứng của nó với các nhu cầu phức tạp, và hạn chế tối đa rủi ro khi triển khai.



Các nguyên tắc thiết kế

Đóng gói

Trong thế giới đời thật của chúng ta, mọi thứ đều được “đóng gói” cẩn thận. Mình tin là các bạn sẽ từ chối sử dụng một cái máy giặt trông xù xì xấu xí, hoặc sẽ không thích một chiếc TV với quá nhiều nút bấm khó hiểu. Việc đóng gói tốt giúp bạn rất nhiều điều quan trọng:

- Giảm thiểu tác động chéo lẫn nhau có thể gây lỗi: Hãy tưởng tượng, ứng dụng của bạn giống như một chiếc xe ô tô với bốn bánh xe. Trong trường hợp xấu, chiếc xe của bạn có thể cán phải đinh (những thay đổi bất thường của nghiệp vụ). Nếu được đóng gói không tốt, chiếc xe sẽ không thể di chuyển. Ngược lại, bạn có thể chỉ thay thế chiếc bánh xe bị hỏng, và chiếc xe có thể hoạt động bình thường.
- Cải thiện tính tái sử dụng: Các module được đóng gói có thể được sử dụng lại ở nhiều nơi: các thư viện mà bạn cài với PHP / NodeJS là đại diện tiêu biểu cho việc đóng gói.
- Tăng tính bảo mật: Việc chỉ cho phép tác động thông qua các phương thức nhất định, giúp đảm bảo an toàn cho đối tượng. Ví dụ class Account sẽ chỉ cho phép thay đổi thuộc tính balance duy nhất thông qua phương thức “deposit”. Trong phương thức này mình sẽ cài đặt rõ ràng các quy tắc để đảm bảo an toàn khi thay đổi số dư. Nhờ đó, ứng dụng trở nên chắc chắn và an toàn hơn rất nhiều.

Đóng gói có thể thể hiện ở nhiều mức độ:

- Ở mức độ phương thức: Đảm bảo rằng một phương thức chỉ làm một việc duy nhất. Bạn có thể quan sát ví dụ bên dưới:
- Ở mức độ Class: Đảm bảo rằng một class chỉ làm một việc duy nhất. Tốt nhất Class chỉ nên chứa một phương thức trách nhiệm chính (hàm handler / execute). Bạn có thể tham khảo cuốn “[Thiết kế hướng nghiệp vụ với Laravel](#)”, section Actions của mình để hiểu rõ thêm.

Đóng gói là một nguyên tắc quan trọng trong thiết kế phần mềm, giúp bảo vệ dữ liệu, tăng tính linh hoạt, dễ dàng bảo trì, giảm sự phụ thuộc, cải thiện tính tái sử dụng và tăng cường bảo mật. Việc áp dụng nguyên tắc đóng gói một cách hiệu quả sẽ giúp bạn tạo ra các phần mềm chất lượng cao, dễ dàng mở rộng và bảo trì trong tương lai.



Public property

Private property
with public getter
and setter

Tập trung vào trừu tượng

Lập trình tập trung vào các giao diện (interface), chứ không phải triển khai (implementation).

Phụ thuộc vào trừu tượng, chứ không phải sự cụ thể.

Nguyên tắc thiết kế này đảm bảo hai yếu tố:

- Tính linh hoạt và mở rộng mà không phá vỡ đi code đã có: Hãy tưởng tượng nhà vệ sinh của bạn bị hỏng chiếc đèn. Bạn có thể ra cửa hàng mua một chiếc đèn tương tự (tương tự ở đây nghĩa là triển khai cùng một giao diện kết nối - interface) và thay thế chiếc bị hỏng. Nếu đèn bị ràng buộc chặt chẽ, bạn phải thay thế hoàn toàn cả hệ thống đèn (ác mộng thật sự).
- Khả năng tái sử dụng code: Vì không phụ thuộc chặt chẽ, các thành phần của ứng dụng có thể đem đi tái sử dụng ở nhiều nơi. Vẫn là câu chuyện của cái đèn bên trên: bởi vì bạn đang không tiện ra ngoài, bạn có thể lấy tạm một chiếc đèn nào đó trong nhà để tái sử dụng ở vị trí chiếc đèn bị hỏng.

Việc tập trung vào giao diện và trừu tượng giúp chúng ta giải quyết hai ví dụ trên một cách gọn nhẹ. Khi thiết kế hệ thống điện và cái đèn, mình sẽ tập trung vào cách để hai hệ thống này tương tác được với nhau (interface). Thay vì nghĩ xem nên làm cái đèn như thế nào để gắn được vào hệ thống điện, mình chỉ cần đảm bảo hai đối tượng này giao tiếp lỏng lẻo với nhau (qua chiếc ổ cắm). Miễn là một cái đèn implement interface này, nó đều có thể được sử dụng và hoạt động bình thường.

Ưu tiên thành phần hơn kế thừa (Composition over Inheritance)

Như mình đã nói ở bên trên, Kế thừa sẽ giúp bạn tái sử dụng code. Cách thức thông thường là mình sẽ “bóc” các đoạn code dùng chung lên class cha, và các class con sẽ kế thừa lại. Thật đơn giản và chuẩn xác 😊

Tuy nhiên, khi ứng dụng của bạn ngày càng phình to ra, thì kế thừa sẽ đem lại những rủi ro rất lớn:

- Bạn phải triển khai lại mọi phương thức trừu tượng của class cha nếu có. Đôi khi class của bạn không nhất thiết phải triển khai toàn bộ các phương thức này.
- Rủi ro vi phạm nguyên tắc thay thế Liskov: Class con có thể có những hành vi xung đột với class cha, gây rủi ro lớn cho ứng dụng của bạn. Ngược lại, mọi thay đổi từ class cha cũng có rủi ro gây lỗi tới các class con kế thừa nó.
- Khi hệ thống phình to, cơ chế kế thừa tái sử dụng code có thể tạo thành nhiều luồng kế thừa song song. Kiểu với nghiệp vụ A, bạn có BaseClass A, và một loạt các class kế thừa BaseClass A. Kiểu nghiệp vụ B, bạn có BaseClass B, và tiếp tục có các class con kế thừa nó. Dần dần, bạn sẽ rất khó kiểm soát ứng dụng của mình.

Có một sự thay thế cho kế thừa gọi là sử dụng thành phần. Trong khi kế thừa thể hiện mối quan hệ "là một" giữa các class (một chiếc xe hơi là một phương tiện), sử dụng thành phần thể hiện mối quan hệ "có một" (một chiếc xe hơi có một động cơ). Điều này giúp tăng tính bảo trì, thay thế và linh hoạt, an toàn hơn cho ứng dụng.

Để áp dụng nguyên tắc này, bạn cần chia tách class / module của mình thành các class con, chịu trách nhiệm đơn lẻ, trước khi kết hợp nó theo các nhu cầu khác nhau để tạo thành các đối tượng bạn mong muốn. Hãy nghĩ đến ví dụ mình đã nhắc lại nhiều lần trong cuốn sách này: Bạn có một cái thân đèn, với các bóng đèn được để riêng bên ngoài. Muốn đèn sáng màu gì, bạn chỉ cần mua bóng đèn tương ứng là được, dễ dàng thay thế và linh hoạt. Đây cũng chính là một cách để thực thi Composition over Inheritance thông qua Dependency Injection (Mình có trình bày rất kỹ về Dependency Injection trong cuốn “Thiết kế hướng nghiệp vụ với Laravel”).



```
<?php

interface ShippingMethod {
    public function calculateCost($weight);
}

class TruckShipping implements ShippingMethod {
    public function calculateCost($weight) {
        return $weight * 1.5;
    }
}
```



```
class TrainShipping implements ShippingMethod {
    public function calculateCost($weight) {
        return $weight * 1.2;
    }
}

class ShippingService {
    private $shipping;

    public function __construct(ShippingMethod $shipping) {
        $this->shipping = $shipping;
    }

    public function calculate($weight) {
        return $this->shipping->calculateCost($weight);
    }
}

// Sử dụng ShippingService với TruckShipping
$truckShipping = new TruckShipping();
$shippingService = new ShippingService($truckShipping);
echo $shippingService->calculate(100); // Output: 150

// Sử dụng ShippingService với TrainShipping
$trainShipping = new TrainShipping();
$shippingService = new ShippingService($trainShipping);
echo $shippingService->calculate(100); // Output: 120
```

Trong ví dụ trên:

Nguyễn Thế Huy

OOP cho người đi làm

- Interface ShippingMethod định nghĩa phương thức calculateCost mà các class cụ thể cần triển khai.
- TruckShipping và TrainShipping triển khai interface ShippingMethod.
- Class ShippingService sử dụng dependency injection để nhận một đối tượng ShippingMethod.

Bằng cách này, nếu chúng ta cần thêm phương thức vận chuyển mới (ví dụ: AirShipping), chúng ta chỉ cần tạo một class mới triển khai ShippingMethod mà không cần thay đổi các class hiện có. Điều này giúp hệ thống của chúng ta dễ mở rộng và duy trì, tuân thủ theo nguyên tắc Đóng / Mở.

Nguyên tắc thay thế Liskov (Liskov Substitution Principle)

Có một điều thú vị về nguyên tắc này: Nó được phát biểu lần đầu bởi một người phụ nữ, bà Barbara Liskov, một nhà khoa học máy tính xuất sắc. Nguyên tắc này có thể được mô tả một cách đơn giản: Thằng class cha có hành vi gì, thì class con phải có đầy đủ có hành vi đó. Nghĩa là ta có thể thay thế class cha bằng class con mà hoàn toàn không phá vỡ ứng dụng của mình.

Nguyên tắc này nghe qua thì rất đơn giản, nhưng nó lại rất dễ vi phạm, nếu bạn không cẩn thận khi sử dụng Kế thừa. Hãy cùng xem qua ví dụ sau:

Giả sử bạn có một class với một phương thức `makeNewCar()`: `Car`.

- Tốt: Một class con ghi đè phương thức như sau: `makeNewCar()`: `SuperCar`. Điều này ổn, vì `SuperCar` là một lớp con của `Car`. Chương trình của chúng ta vẫn sẽ chạy bình thường.

Ví dụ:

```
<?php

// Lớp cha
class Car {
    public function drive() {
        return "Lái xe";
    }
}

// Lớp SuperCar
class SuperCar extends Car {
    public function drive() {
        return "Lái siêu xe";
    }
}

// Lớp cha với phương thức makeNewCar
class CarFactory {
```

```

public function makeNewCar(): Car {
    return new Car();
}
}
// Code chạy thử không Lỗi
function testCarFactory(CarFactory $factory) {
    $car = $factory->makeNewCar();
    echo $car->drive() . "\n";
}

```

- Xấu: Một class con ghi đè phương thức như sau: makeNewCar(): Ship. Vấn đề đã xảy ra ở đây: phương thức được ghi đè hoàn toàn không tương thích với class cha. Điều này sẽ gây lỗi tới tất cả lời gọi phương thức này. Tất nhiên, ví dụ này mình đã làm nó “lỗi” một cách rất rõ ràng (các IDE có thể sớm warning bạn). Trong thực tế, nó có thể được ẩn giấu một cách khéo léo, đến mức làm bạn không để ý, và lỗi cũng có thể nghiêm trọng hơn rất nhiều.

Ví dụ:

```

<?php

// Lớp cha
class Car {
    public function drive() {
        return "Lái xe";
    }
}

// Lớp không Liên quan đến Car
class Ship {
    public function sail() {
        return "Đi thuyền";
    }
}

```

```

}

// Lớp cha với phương thức makeNewCar
class CarFactory {
    public function makeNewCar(): Car {
        return new Car();
    }
}

// Lớp con với phương thức makeNewCar trả về Ship (không phải là Lớp con của Car)
class ShipFactory extends CarFactory {
    public function makeNewCar(): Ship {
        return new Ship();
    }
}

// Code chạy thử gây Lỗi
function testCarFactory(CarFactory $factory) {
    $car = $factory->makeNewCar();
    echo $car->drive() . "\n"; // Lỗi vì Ship không có phương thức drive
}

```

Đảm bảo nguyên tắc Liskov giúp code được tái sử dụng an toàn, không làm thay đổi hành vi của đối tượng một cách bất ngờ, từ đó giảm thiểu tối đa lỗi gây ra cho ứng dụng.

Nguyên tắc phân tách giao diện (Interface Segregation Principle)

Client chỉ nên phụ thuộc vào những gì mà nó cần. Có nghĩa là khi triển khai các interface, đừng làm nó trở nên quá “rộng” và cover mọi trường hợp. Hãy thu hẹp nó làm sao để các class con không cần triển khai các hành vi mà nó không cần. Điều này mục đích nhằm tránh việc khi bạn sửa một phương thức nào đó ở interface, nó sẽ không gây lỗi tới các class con không đụng tới các phương thức này.

Hãy quay lại các ví dụ về động vật. Chúng ta đều biết là con mèo và con cá thì đều có thể bơi được, nhưng con cá thì không chạy được. Con mèo và con chim có thể chạy được, nhưng con mèo thì không bay được. Ban đầu, nếu bạn triển khai một Interface lớn như Moveable, bao gồm ba phương thức: swim(), fly(), run() như dưới đây:

thì sẽ rất bối rối nếu Cat phải implement cả fly(), và Fish thì lại phải implement run(). Rủi ro có thể đến nếu có sự thay đổi ở interface và khiến Cat / Fish gặp lỗi.

Thay vì đó, mình chia nhỏ các Interface trên thành ba interface: Flyable, Runnable và Swimmable. Giờ đây, mỗi class con chỉ cần implement interface mà nó cần, đảm bảo rằng hạn chế tối đa các rủi ro xảy ra:

```
// Trước khi áp dụng Interface Segregation Principle  
<?php  
  
interface Moveable {  
    public function move();  
}  
  
class Cat implements Moveable {  
    public function move() {  
        echo "Cat runs";  
    }  
}
```

```
class Fish implements Moveable {
    public function move() {
        echo "Fish swims";
    }
}

class Bird implements Moveable {
    public function move() {
        echo "Bird flies";
    }
}

//Sau khi áp dụng Interface Segregation Principle
<?php

interface Flyable {
    public function fly();
}

interface Runnable {
    public function run();
}

interface Swimmable {
    public function swim();
}

class Cat implements Runnable {
```

```
public function run() {  
    echo "Cat runs";  
}  
}  
  
class Fish implements Swimmable {  
    public function swim() {  
        echo "Fish swims";  
    }  
}  
  
class Bird implements Flyable {  
    public function fly() {  
        echo "Bird flies";  
    }  
}
```


Nguyên tắc đảo ngược phụ thuộc (Dependency Inversion Principle)

Các lớp cấp cao không nên phụ thuộc vào các lớp cấp thấp. Cả hai nên phụ thuộc vào các trừu tượng. Các trừu tượng không nên phụ thuộc vào chi tiết. Chi tiết nên phụ thuộc vào các trừu tượng.

Nguyên tắc này nhằm mục đích giảm sự phụ thuộc giữa các phần của chương trình bằng cách sử dụng các trừu tượng (interface hoặc abstract class) để tách biệt các module cấp cao và cấp thấp. Điều này giúp cho các module có thể thay đổi độc lập mà không ảnh hưởng đến các module khác, tăng tính linh hoạt và khả năng mở rộng của hệ thống. Ví dụ:

Trước Khi Áp Dụng DIP

Giả sử chúng ta có một ứng dụng đơn giản để gửi thông báo, trong đó lớp Notification phụ thuộc trực tiếp vào lớp EmailService.

```
<?php

class EmailService {
    public function sendEmail($message) {
        echo "Sending email: $message\n";
    }
}

class Notification {
    private $emailService;

    public function __construct() {
        $this->emailService = new EmailService();
    }

    public function send($message) {
        $this->emailService->sendEmail($message);
    }
}
```

```
}

// Sử dụng Lớp Notification
$notification = new Notification();
$notification->send("Hello, World!");

?>
```

Trong ví dụ này, lớp Notification phụ thuộc trực tiếp vào lớp EmailService. Nếu chúng ta muốn thay đổi cách gửi thông báo, ví dụ gửi qua SMS thay vì email, chúng ta phải thay đổi mã của lớp Notification, điều này vi phạm DIP.

Sau Khi Áp Dụng DIP

Để áp dụng DIP, chúng ta sẽ giới thiệu một interface (hoặc abstract class) MessageService mà cả EmailService và SmsService đều triển khai. Lớp Notification sẽ phụ thuộc vào MessageService thay vì EmailService.

```
<?php

interface MessageService {
    public function sendMessage($message);
}

class EmailService implements MessageService {
    public function sendMessage($message) {
        echo "Sending email: $message\n";
    }
}

class SmsService implements MessageService {
```

```
public function sendMessage($message) {
    echo "Sending SMS: $message\n";
}
}

class Notification {
    private $service;

    public function __construct(MessageService $service) {
        $this->service = $service;
    }

    public function send($message) {
        $this->service->sendMessage($message);
    }
}

// Sử dụng Lớp Notification với EmailService
$emailService = new EmailService();
$notification = new Notification($emailService);
$notification->send("Hello via Email!");

// Sử dụng Lớp Notification với SmsService
$smsService = new SmsService();
$notification = new Notification($smsService);
$notification->send("Hello via SMS!");

?>
```

Trong ví dụ này:

- Trừu Tượng Hóa (Abstraction): Chúng ta tạo ra interface `MessageService` để trừu tượng hóa hành vi gửi thông báo.
- Module Cấp Cao: Lớp `Notification` bây giờ phụ thuộc vào `MessageService` thay vì phụ thuộc trực tiếp vào `EmailService`.
- Module Cấp Thấp: Các lớp `EmailService` và `SmsService` triển khai `MessageService`.

Bằng cách này, chúng ta có thể dễ dàng thay thế hoặc mở rộng cách gửi thông báo mà không cần thay đổi lớp `Notification`. Điều này làm cho mã linh hoạt hơn, dễ bảo trì hơn và tuân thủ nguyên tắc DIP.

Inversion of Control (IoC)

Nhắc đến nguyên tắc Đảo ngược sự phụ thuộc thì không thể không nhắc tới nguyên lý Inversion of Control hay IoC. Đối với IoC, thay vì flow thông thường là chính ứng dụng khởi tạo và gọi đến các thành phần khác, trong IoC, flow của ứng dụng được điều khiển bởi một framework hay một Container bên ngoài.

Hiểu một cách đơn giản, class của bạn sẽ trao quyền điều khiển ra bên ngoài, qua việc tiêm (inject) các phụ thuộc (dependency). Đối với mô hình thông thường, class sẽ tự khởi tạo các phụ thuộc bên trong để sử dụng.

Đối với Laravel, bạn sẽ không xa lạ gì với bộ IoC được gọi là “Service Container”. Trên SpringBoot, bộ IoC sẽ khởi tạo các “Bean”. Chúng đều tuân theo nguyên tắc: các đối tượng, dịch vụ được đưa vào quản lý bởi một trung tâm được gọi là Service Locator, từ trung tâm đăng ký (registry) này các lớp muốn sử dụng dịch vụ nào thì Locator sẽ cung cấp (khởi tạo nếu chưa, và trả về dịch vụ cho đối tượng sử dụng). Những lợi thế mà IoC sẽ cung cấp cho ứng dụng của bạn:

- Ứng dụng được điều khiển linh hoạt từ bên ngoài thông qua Service Locator. Bạn có thể binding các instance tại thời điểm chạy ứng dụng (Runtime). Ví dụ inject một Interface, mình có thể hoán đổi các class implement interface đó và inject vào class chính bất kỳ lúc nào.
- Tách rời việc khởi tạo và thực thi giúp cấu trúc linh hoạt, dễ dàng thay thế và bảo trì.
- Dễ dàng testing thông qua mocking các dependency

Cách thực hiện IoC phổ biến nhất hiện nay là kết hợp giữa Service Locator (cung cấp bởi các Framework) và [Dependency Injection](#).



Passing a value



Dependency Injection with Inversion of Control

IoC và Dependency Injection ở mọi nơi 😊