OOP

CHO NGƯỜI ĐI LÀM

- Nguyễn Thế Huy
- huynt57@gmail.com

Lời mở đầu

Chào bạn, và cám 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.

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 □l

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
<?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";
    }
```

```php
}

// 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 "chìa" 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
<?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");
```

```php
$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 hoạ cho kế thừa:

```php
<?php
```

```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
<?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
```

```php
class SportsCar extends Car {
    // Triển khai lại phương thức move cho SportsCar
    public function move() {
        echo "SportsCar đang chạy ở tốc độ cao\n";
    }
}1

// 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

?>
```

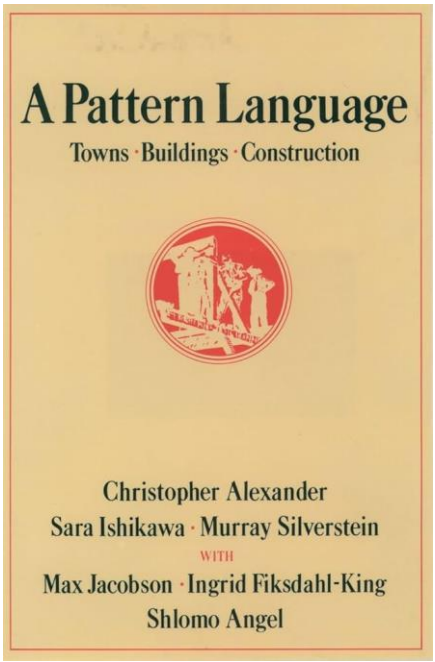
Giới thiệu về mẫu thiết kế (Design Pattern)

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 cao 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, tuỳ 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.



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

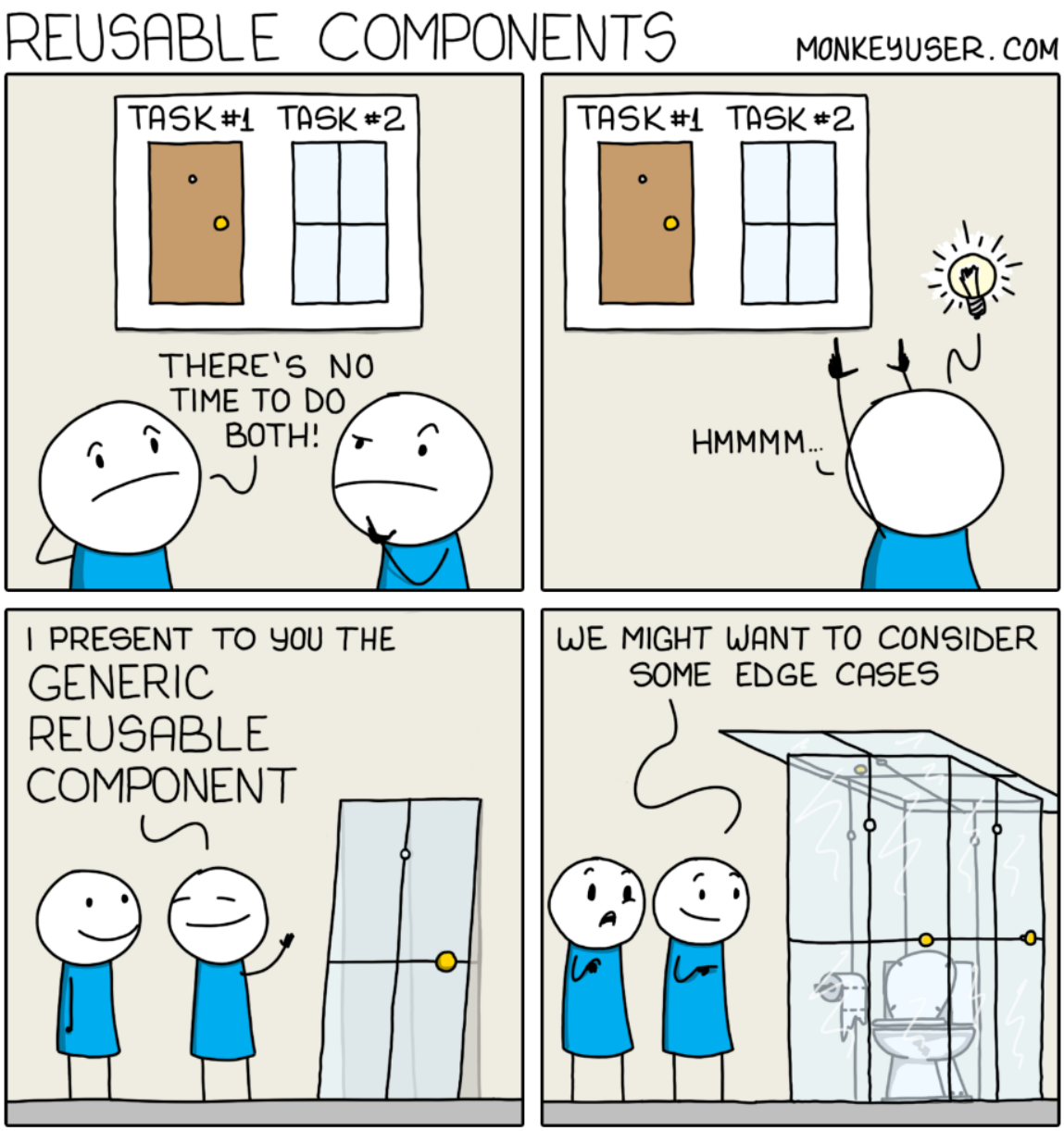
Tái sử dụng code (Code Reuse)

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 ?

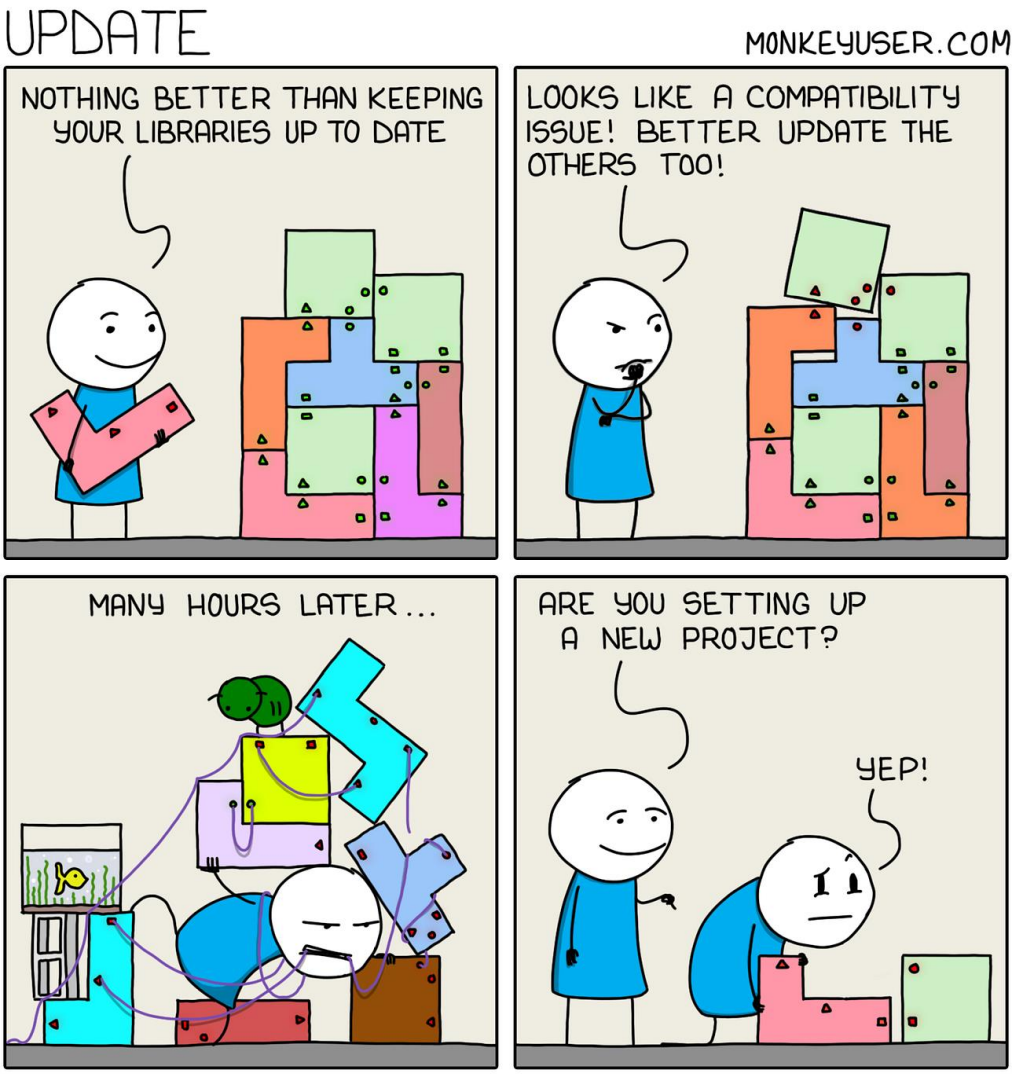
Nói thì dễ, nhưng việc tái sử dụng code không hề đơn giản chút nào. Các class bị ràng buộc chặt chẽ với nhau mà không qua các lớp trừu tượng hay việc không phân chia trách nhiệm rõ ràng cho các đối tượng (Hiểu nôm na là một class mấy nghìn dòng code). Những điều này là biểu hiện của một thiết kế chưa 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
TASK #1 TASK #2
THERE'S NO TIME TO DO BOTH!
TASK #1 TASK #2
HMMMM...
I PRESENT TO YOU THE GENERIC REUSABLE COMPONENT
WE MIGHT WANT TO CONSIDER SOME EDGE CASES

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 ràng 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.

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 điền (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”).


COMPOSITION OVER INHERITANCE
PREFER YOU MUST
memegenerator.net

SOLID

SOLID là các nguyên tắc quan trọng trong Lập trình hướng đối tượng, được Robert Martin (Uncle Bob) giới thiệu trong quyển "**Agile Software Development, Principles, Patterns, and Practices**" của ông. SOLID bao gồm năm nguyên tắc đại diện cho năm chữ cái, nhằm mục tiêu giúp các thiết kế phần mềm trở nên dễ hiểu, linh hoạt và dễ bảo trì.

Tuy nhiên, áp dụng SOLID cần cẩn thận và linh hoạt theo tình huống. Chi phí áp dụng và độ phức tạp có thể tăng nếu bạn áp dụng nó một cách máy móc. Hãy cùng đi một vòng các nguyên tắc này, mình sẽ giúp bạn phân tích, trước khi chúng ta sang section về Design Pattern nhé.

Nguyên Tắc Trách Nhiệm Duy Nhất (Single Responsibility Principle)

“Một Class chỉ nên có một lý do để thay đổi.” - Uncle Bob

Hãy cố gắng đảm bảo rằng một Class chỉ làm một việc, và duy nhất một việc mà thôi. Công việc này cũng cần được đóng gói một cách gọn gàng bên trong Class đó, và chỉ cho bên ngoài truy cập khi cần thiết.

Mục tiêu chính của nguyên tắc này là giảm bớt đi sự phức tạp. Áp dụng triệt để nguyên tắc này, bạn sẽ không còn lo sợ việc phải viết những đoạn code dài hàng gang tay, hoặc phải maintain những ứng dụng nghiệp vụ khù khoằm. Nếu bạn đã đọc cuốn: “Thiết kế hướng nghiệp vụ với Laravel” của mình, chắc hẳn bạn đã hiểu phần nào mindset này.

Có một tips mình thường xuyên áp dụng để đảm bảo Single Responsibility, đó là thiết kế theo hướng “Use case”. Các class của mình chỉ có một hàm duy nhất là “handle” (hoặc “execute”) và nó chỉ làm duy nhất một công việc. Nếu có một nghiệp vụ mới phát sinh, mình sẽ tạo Class mới (với hàm “handle”/”execute”) mới, cứ như vậy. Nếu bạn quan sát Laravel, bạn cũng sẽ thấy họ áp dụng rất thường xuyên cách thức này: Các class Jobs / Listeners đều đơn lẻ, và được kích hoạt qua hàm handle duy nhất. Đó cũng là cách áp dụng nguyên tắc đóng gói ở mức Class, mình đã trình bày ở bên trên.

```php
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Services\UserService;

class RegisterUserController extends Controller
{
    protected $userService;

    public function __construct(UserService $userService)
```

```
    {
        $this->userService = $userService;
    }

    public function __invoke(Request $request)
    {
        $data = $request->validate([
            'name' => 'required|string|max:255',
            'email' => 'required|string|email|max:255|unique:users',
            'password' => 'required|string|min:8|confirmed',
        ]);

        $this->userService->register($data);

        return response()->json(['message' => 'User registered successfully'],
201);
    }
}
```

Trong ví dụ bên trên, bạn có thể thấy mình sử dụng Single Action Controller (Một khái niệm được sử dụng rất phổ biến trong cộng đồng Laravel). Trong class RegisterUserController được kích hoạt duy nhất qua hàm __invoke(), và nó sẽ chỉ làm duy nhất công việc đăng ký User.

Khi bạn không đảm bảo tính trách nhiệm duy nhất, cơn ác mộng sẽ tới khi ứng dụng của bạn gặp phải những thay đổi. Một class Service lớn, khiến bạn bối rối khi phải sửa nó để đáp ứng nghiệp vụ. Và rất có thể, bạn có nguy cơ thay đổi những thứ mà bạn không mong muốn, dẫn tới những lỗi tiềm ẩn trong tương lai.

Nguyên Tắc Mở/Đóng (Open/Closed Principle)

Các lớp nên mở để mở rộng nhưng đóng để sửa đổi.

Ý tưởng chính của nguyên tắc này là giữ cho mã hiện có không bị hỏng khi bạn triển khai các tính năng mới.

Mình nghĩ đây là một nguyên tắc khá dễ hiểu. Mình hay nói vui với đồng nghiệp: “Cái gì đang chạy rồi thì đừng sửa nó” □. Tư tưởng của nguyên tắc này phần nào phản ánh câu nói đùa trên: chúng ta cần hạn chế tối đa sửa code cũ, mà ưu tiên mở rộng, và “thay thế” nếu có thể. Hãy tưởng tượng bạn đang làm việc với một website Thương mại điện tử. Một ngày đẹp trời, công ty yêu cầu bạn phải thêm một phương thức vận chuyển mới. Sẽ là ác mộng nếu mọi thứ liên quan đến phương thức vận chuyển đều được viết chung trong class Order (Đơn hàng). Bạn sẽ phải tác động trực tiếp vào code cũ, cố gắng hy vọng rằng mình không làm hỏng điều gì (và chắc chắn là nó sẽ hỏng rồi, định lý Murphy mà □). Lúc này, Strategy pattern sẽ giải cứu bạn: chúng ta sẽ trích xuất các phương thức vận chuyển này sang các class chịu trách nhiệm riêng biệt, implement chung interface để dễ dàng thay đổi khi cần.

Bạn có thể tham khảo ví dụ minh hoạ dưới đây:

Giả sử chúng ta có một hệ thống tính toán chi phí vận chuyển cho các loại phương tiện khác nhau. Ban đầu, hệ thống chỉ hỗ trợ vận chuyển bằng xe tải:

```php
<?php

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

class ShippingService {
    private $shipping;

    public function __construct(TruckShipping $shipping) {
```

```
        $this->shipping = $shipping;
    }

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

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

Khi cần thêm loại phương tiện vận chuyển mới như tàu hỏa, thay vì sửa đổi các class hiện có, chúng ta sẽ mở rộng hệ thống bằng cách sử dụng interface:

```
<?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:

- 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
<?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 tác 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:

```php
// 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
<?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 {
```

```php
    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 thưởng, 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.

IoC và Dependency Injection ở mọi nơi

Các Design Pattern quan trọng

Có rất nhiều các Design Pattern khi thiết kế hướng đối tượng. Cuốn “**Design Patterns: Elements of Reusable Object-Oriented Software**” của Gang of Four (Erich Gamma, Richard Helm, Ralph Johnson và John Vlissides) là cuốn tiên phong áp dụng các Pattern vào thiết kế phần mềm, đề cập đến 23 patterns, chia vào các nhóm khởi tạo, cấu trúc, hành vi (cuốn này là kinh điển về Design Pattern, bạn nên tìm đọc). Trong section dưới đây, mình chỉ xin chia sẻ về các Pattern mình cho là quan trọng, thường xuyên được sử dụng trong thực tế, cũng như cách nhận biết tình huống nào bạn cần sử dụng chúng.

Nhóm khởi tạo

Mục tiêu chính của hầu hết các Pattern thuộc nhóm khởi tạo là “kéo” việc khởi tạo trực tiếp đối tượng ra thành một class riêng biệt. Nhờ đó, việc thay thế các đối tượng sẽ trở nên linh hoạt hơn, đảm bảo các nguyên tắc SOLID. Chúng ta cung cấp một giao diện chung cho việc khởi tạo đối tượng và có thể thay thế đối tượng đó ở class con khi cần thiết.

Factory Pattern

Factory Method là một mẫu thiết kế khởi tạo cung cấp một giao diện để tạo các đối tượng trong một lớp cha, nhưng cho phép các lớp con thay đổi loại đối tượng sẽ được tạo ra. Mục đích chính của Factory Design Pattern là giảm sự phụ thuộc giữa code và lớp cụ thể. Điều này giúp dễ dàng mở rộng và bảo trì hệ thống hơn.

Dấu hiệu nhận biết:

Khi bạn cần khởi tạo nhiều loại đối tượng khác nhau mà không cần biết chính xác đối tượng nào sẽ được khởi tạo. Ví dụ trong việc xây dựng các phương thức vận chuyển cho website Thương mại điện tử: Bạn có thể gửi hàng qua GHTK, GHN, ViettelPost, ….. Lúc này, Factory sẽ đóng vai trò khởi tạo đối tượng tương ứng của từng phương thức vận chuyển khác nhau.

Ví dụ

```php
<?php

// Lớp Dog
class Dog {
    public function speak() {
        return "Gâu gâu";
    }
}

// Lớp Cat
class Cat {
```

```
    public function speak() {
        return "Meo meo";
    }
}

// Tạo lớp Factory
class AnimalFactory {
    // Phương thức để tạo đối tượng Animal dựa trên loại (type)
    public function createAnimal($type) {
        switch ($type) {
            case 'dog':
                return new Dog();
            case 'cat':
                return new Cat();
            default:
                throw new Exception("Invalid animal type");
        }
    }
}

// Sử dụng Factory để tạo đối tượng Dog
$factory = new AnimalFactory();

$dog = $factory->createAnimal('dog');
echo $dog->speak(); // Output: Gâu gâu

// Sử dụng Factory để tạo đối tượng Cat
$cat = $factory->createAnimal('cat');
echo $cat->speak(); // Output: Meo meo
```

```
?>
```

Chú ý: Mình không đi sâu vào hai Pattern riêng biệt là Factory Method và Abstract Factory vì tính tương tự nhau của nó. Ở đây, mình đưa đến một cách đơn giản hơn để áp dụng Factory. Các bạn có thể tìm hiểu thêm để đi sâu hơn vào hai Pattern trên nhé.

Builder

Builder là một mẫu thiết kế khởi tạo cho phép bạn xây dựng các đối tượng phức tạp từng bước một. Pattern này cho phép bạn tạo ra các loại và biểu diễn khác nhau của một đối tượng bằng cách sử dụng cùng một đoạn code.

Dấu hiệu nhận biết:

- Bạn có một đối tượng lớn, phức tạp. Ví dụ một đơn hàng với nhiều thuộc tính về khách hàng, thời gian gửi hàng, ghi chú …. Quá trình khởi tạo đối tượng này cần một Builder riêng.
- Đối tượng lớn nhưng đôi khi không cần toàn bộ các thuộc tính. Bạn có thể sử dụng các Builder khác nhau để xây dựng các đối tượng Đơn hàng phù hợp với context (ngữ cảnh mà bạn đang sử dụng).

Ví dụ

```
<?php

// Định nghĩa class Car
class Car {
    private $model;
    private $year;
    private $color;

    public function setModel($model) {
        $this->model = $model;
    }
```

```
    public function setYear($year) {
        $this->year = $year;
    }

    public function setColor($color) {
        $this->color = $color;
    }

    public function __toString() {
        return "{$this->color} {$this->year} {$this->model}";
    }
}

// Định nghĩa class CarBuilder
class CarBuilder {
    private $car;

    public function __construct() {
        $this->car = new Car();
    }

    public function setModel($model) {
        $this->car->setModel($model);
        return $this;
    }

    public function setYear($year) {
        $this->car->setYear($year);
        return $this;
```

```
    }

    public function setColor($color) {
        $this->car->setColor($color);
        return $this;
    }

    public function build() {
        return $this->car;
    }
}

// Sử dụng CarBuilder để tạo đối tượng Car
$builder = new CarBuilder();
$car = $builder->setModel('Corolla')
               ->setYear(2020)
               ->setColor('Red')
               ->build();

echo $car; // Output: Red 2020 Corolla

?>
```

Singleton

Singleton là một mẫu thiết kế khởi tạo đảm bảo rằng một class chỉ có một instance duy nhất, và nó sẽ trở thành instance toàn cục cho toàn bộ lifecycle của ứng dụng. Điều này rất hữu ích khi bạn cần một đối tượng duy nhất để điều phối các hành động trong hệ thống.

Dấu hiệu nhận biết:

Những trường hợp bạn cần tái sử dụng trạng thái của đối tượng trong ứng dụng: ví dụ mở Database Connection, hoặc tạo một HTTP Client. Trong những trường hợp này, bạn không mong muốn khởi tạo lại instance mới từ đầu, mà chuyển sang sử dụng instance đã có.

Ví dụ:

Một đoạn code sử dụng Singleton đơn giản để khởi tạo Database Connection

```
<?php

class Database {
    // Thuộc tính tĩnh lưu trữ thể hiện duy nhất của lớp Database
    private static $instance = null;
    private $connection;
    private $host = 'localhost';
    private $username = 'root';
    private $password = '';
    private $database = 'example_db';

    // Constructor được khai báo private để ngăn việc khởi tạo đối tượng từ bên ngoài
    private function __construct() {
        $this->connection = new mysqli($this->host, $this->username, $this->password, $this->database);

        // Kiểm tra kết nối
        if ($this->connection->connect_error) {
            die("Connection failed: " . $this->connection->connect_error);
        }
    }
```

```php
    // Sử dụng static để đảm bảo chỉ khởi tạo duy nhất một instance
    public static function getInstance() {
        if (self::$instance === null) {
            self::$instance = new Database();
        }
        return self::$instance;
    }

    // Phương thức để lấy kết nối
    public function getConnection() {
        return $this->connection;
    }
}

// Sử dụng Database Singleton
$db1 = Database::getInstance();
$conn1 = $db1->getConnection();

$db2 = Database::getInstance();
$conn2 = $db2->getConnection();

// Kiểm tra xem hai kết nối có trỏ đến cùng một thể hiện không
if ($conn1 === $conn2) {
    echo "Both connections are the same.\n";
}

?>
```

Nhóm cấu trúc

Nhóm cấu trúc của các mẫu thiết kế (Structural Design Patterns) liên quan đến cách tổ chức các lớp và đối tượng để tạo thành các cấu trúc lớn hơn và phức tạp hơn trong phần mềm. Các mẫu thiết kế này giúp đảm bảo rằng các mối quan hệ giữa các đối tượng và lớp được tổ chức một cách hiệu quả và dễ bảo trì.

Adapter

Adapter là một mẫu thiết kế cấu trúc cho phép các đối tượng với interface không tương thích làm việc cùng nhau. Nó hoạt động như một cầu nối giữa các đối tượng không tương thích bằng cách chuyển đổi interface của một lớp thành interface mà một lớp khác có thể làm việc cùng. Hiểu nôm na, class Adapter sẽ implement interface A, và triển khai lại phương thức của class B. Khi một class con triển khai interface A cần kết nối tới B, nó chỉ cần gọi tới Adapter và gọi phương thức tương ứng là được.

Dấu hiệu nhận biết:

Bài toán dễ nhận thấy nhất với Adapter là khi bạn cần chuyển đổi hệ thống, migrate dữ liệu giữa các module khác nhau. Lúc này, Adapter đóng vai trò cây cầu nối đảm việc chuyển đổi trơn tru, không gặp sự cố bất thường.

Ví dụ:

Chúng ta có một class EmailService đang chạy. Giờ mình muốn sử dụng class SendEmailAWS mới, class này có hành vi khác với class cũ (do sử dụng dịch vụ SES của AWS). Chúng ta sẽ sử dụng Adapter để làm cầu nối hai hệ thống này

```php
<?php
interface EmailServiceInterface {
    public function sendEmail($to, $subject, $message);
}

class EmailService implements EmailServiceInterface {
```

```
    public function sendEmail($to, $subject, $message) {
        // Gửi email bằng thư viện đã có
        echo "Đã gửi email đến $to với chủ đề: $subject và nội dung: $message";
    }
}

class SendEmailAWS {
    public function send($email) {
        // Gửi email bằng API của AWS SES
        echo "Đã gửi email thông qua AWS SES";
    }

}
class AWSAdapter implements EmailServiceInterface {
    private $awsService;

    public function __construct(SendEmailAWS  $awsService) {
        $this->awsService = $awsService;
    }

    public function sendEmail($to, $subject, $message) {
        // Chuyển đổi từ giao diện hiện có sang giao diện của AWS
        $email = [
            'to' => $to,
            'subject' => $subject,
            'message' => $message
        ];
        $this->awsService->send($email);
    }
}
```

```
// Sử dụng EmailService nhưng bên dưới nó sẽ gửi qua AWS
$emailService = new AWSAdapter(new SendEmailAWS());
$emailService->sendEmail('recipient@example.com', 'Chủ đề', 'Nội dung');
```

Bạn có thể thay thế $emailService được khởi tạo từ Adapter vào bất cứ chỗ nào đang sử dụng code cũ mà không làm break hệ thống.

Decorator

Decorator là một mẫu thiết kế cấu trúc cho phép bạn thêm các hành vi mới vào các đối tượng bằng cách đặt chúng bên trong các đối tượng bao bọc đặc biệt chứa các hành vi này. Điều này cho phép bạn thêm chức năng vào các đối tượng hiện có mà không cần sửa đổi code của chúng, giảm thiểu rủi ro gây lỗi.

Dấu hiệu nhận biết:

Các yêu cầu bổ sung thêm hành vi cho đối tượng, dựa trên hành vi gốc có thể là dấu hiệu cần sử dụng tới Decorator.

Ví dụ:

Bạn đang làm một hệ thống bán xe ô tô. Giá cơ bản của một chiếc xe là 20,000 USD. Nếu lắp thêm các hệ thống phụ trợ, giá xe sẽ thay đổi. Chúng ta sẽ áp dụng Decorator như sau:

```
<?php
// Interface đại diện cho ô tô
interface Car {
    public function cost();
    public function description();
}

// Class cơ bản đại diện cho ô tô
class BasicCar implements Car {
```

```
    public function cost() {
        return 20000; // Giá của ô tô cơ bản là 20,000 USD
    }

    public function description() {
        return "Ô tô cơ bản";
    }
}

// Decorator
abstract class CarDecorator implements Car {
    protected $car;

    public function __construct(Car $car) {
        $this->car = $car;
    }

    public function cost() {
        return $this->car->cost();
    }

    public function description() {
        return $this->car->description();
    }
}

// Class decorate thêm hệ thống định vị
class NavigationSystem extends CarDecorator {
    public function cost() {
        return $this->car->cost() + 2000; // Giá của hệ thống định vị là 2,000
```

```
USD
    }

    public function description() {
        return $this->car->description() . ", có hệ thống định vị";
    }
}

// Class decorate thêm ghế da
class LeatherSeats extends CarDecorator {
    public function cost() {
        return $this->car->cost() + 1500; // Giá của ghế da là 1,500 USD
    }

    public function description() {
        return $this->car->description() . ", có ghế da";
    }
}

// Sử dụng
$basicCar = new BasicCar();
echo "Giá của ô tô cơ bản: " . $basicCar->cost() . " USD\n";
echo "Mô tả của ô tô cơ bản: " . $basicCar->description() . "\n\n";

$carWithNavigation = new NavigationSystem($basicCar);
echo "Giá của ô tô có hệ thống định vị: " . $carWithNavigation->cost() . "
USD\n";
echo "Mô tả của ô tô có hệ thống định vị: " . $carWithNavigation->description()
. "\n\n";
```

```
$carWithLeatherSeats = new LeatherSeats($basicCar);
echo "Giá của ô tô có ghế da: " . $carWithLeatherSeats->cost() . " USD\n";
echo "Mô tả của ô tô có ghế da: " . $carWithLeatherSeats->description() .
"\n\n";

$carWithBothOptions = new NavigationSystem(new LeatherSeats($basicCar));
echo "Giá của ô tô có cả hệ thống định vị và ghế da: " . $carWithBothOptions-
>cost() . " USD\n";
echo "Mô tả của ô tô có cả hệ thống định vị và ghế da: " . $carWithBothOptions-
>description() . "\n";
```

Nhóm hành vi

Mẫu thiết kế hành vi xử lý việc giao tiếp hiệu quả và phân công trách nhiệm giữa các đối tượng.

Chain of Responsibility

Chain of Responsibility là pattern cho phép dữ liệu được truyền qua một loạt các đối tượng xử lý trước khi ra được kết quả cuối cùng. Những lợi ích của pattern này:

- Phân chia trách nhiệm rõ ràng ra các Class riêng biệt.
- Tách biệt các bước giúp

Dấu hiệu nhận biết:

Các công việc cần chia thành từng bước (giống như trên một dây chuyền) có thể nghĩ tới Pattern này. Một ví dụ tiêu biểu là Middleware cho một ứng dụng web (HTTP). Khi HTTP Request tới, nó đi qua một loạt các Middleware như xác thực, phân quyền … trước khi đến bước xử lý cuối. Nếu một trong các bước đó gặp vấn đề, Request đó sẽ bị tạm dừng.

Ví dụ:

Mình sẽ sử dụng Pipeline trong Laravel để mô tả Pattern này:

```
use Illuminate\Support\Facades\Pipeline;
use App\Pipelines\CheckAuthentication;
use App\Pipelines\LogRequest;
use App\Pipelines\HandleResponse;

// Tạo $request sẽ đi qua các class xử lý
$request = [];

// Sử dụng Pipeline để xử lý yêu cầu
$response = Pipeline::send($request)
    ->through([
```

```
        CheckAuthentication::class,
        LogRequest::class,
        HandleResponse::class,
    ])
    ->then(function ($request) {
        return "Request handled successfully.";
    });

echo $response;
```

Command

Command là một mẫu thiết kế hành vi biến một yêu cầu thành một đối tượng độc lập chứa tất cả thông tin về yêu cầu đó, và sử dụng nó trở thành đầu vào cho class xử lý. Command giúp đóng gói nghiệp vụ, phân chia rõ ràng giữa luồng yêu cầu - xử lý (Vì yêu cầu đã đóng vai trò là tham số cho luồng xử lý, nên nó được tách bạch rõ ràng các class xử lý với nhau).

Dấu hiệu nhận biết:

- Bạn cần tách biệt yêu cầu của Hành động (Command) ra khỏi phần xử lý Hành động (Handler)
- Bạn cần tham số hoá các đối tượng dựa theo Hành động (Command)

Trong cuốn “Thiết kê hướng nghiệp vụ với Laravel” của mình, mình có mô tả rất rõ về cách áp dụng CommandBus Design Pattern trong thiết kế (Tương tự Pattern Command này). Các bạn có thể tham khảo nhé.

Ví dụ:

```
<?php

// Định nghĩa interface Command
```

```php
interface Command {
    public function execute();
}

// Tạo các class Command cụ thể
class TurnOnLightCommand implements Command {
    private $light;

    public function __construct(Light $light) {
        $this->light = $light;
    }

    public function execute() {
        $this->light->turnOn();
    }
}

class TurnOffLightCommand implements Command {
    private $light;

    public function __construct(Light $light) {
        $this->light = $light;
    }

    public function execute() {
        $this->light->turnOff();
    }
}

// Tạo class Light được tương tác bởi Command
```

```php
class Light {
    public function turnOn() {
        echo "The light is on\n";
    }

    public function turnOff() {
        echo "The light is off\n";
    }
}

// Class nhận vào Command và gọi đến Handler tương ứng
class RemoteControl {
    private $command;

    public function setCommand(Command $command) {
        $this->command = $command;
    }

    public function pressButton() {
        $this->command->execute();
    }
}


// Tạo đối tượng Light
$light = new Light();

// Tạo đối tượng Command
$turnOnCommand = new TurnOnLightCommand($light);
$turnOffCommand = new TurnOffLightCommand($light);
```

```
// Tạo đối tượng RemoteControl
$remote = new RemoteControl();

// Bật đèn
$remote->setCommand($turnOnCommand);
$remote->pressButton(); // Output: The light is on

// Tắt đèn
$remote->setCommand($turnOffCommand);
$remote->pressButton(); // Output: The light is off
```

Observer

Observer là một mẫu thiết kế hành vi cho phép bạn xác định một cơ chế đăng ký để cho phép nhiều đối tượng nhận thông báo về các sự kiện xảy ra đối với đối tượng mà chúng đang quan tâm. Điều này giúp tách biệt code, đảm bảo rằng các đối tượng lắng nghe (observe) không phụ thuộc chặt chẽ vào đối tượng gốc, giúp ứng dụng của bạn linh hoạt hơn rất nhiều.

Dấu hiệu nhận biết:

Bạn cần kích hoạt một loạt các hành vi theo sau bởi một sự kiện nào đó. Ví dụ khi một đơn đặt hàng thành công, mình cần gửi Mail thông báo, tiến hành trừ kho hàng hoá, …. Nếu không sử dụng Observer, bạn cần kích hoạt nó trực tiếp trong phương thức xử lý đặt hàng. Với observer, bạn chỉ cần gửi một sự kiện, và tuỳ ý các class đã đăng ký xử lý theo nghiệp vụ riêng biệt.

Ví dụ:

```
<?php
// Định nghĩa interface subject, để chuẩn bị cho việc thêm (attach) observer,
bớt (detach), và gửi thông báo (notify) tới các Observer
interface Subject {
```

```
    public function attach(Observer $observer);
    public function detach(Observer $observer);
    public function notify();
}

interface Observer {
    public function update($state);
}

// Class Order cho phép các observer đăng ký nhận sự kiện. Khi trạng thái đơn hàng thay đổi qua hàm setState, tiến hành gọi hàm notify() để thông báo đến tất cả các Observer.
class Order implements Subject {
    private $observers = [];
    private $state;

    public function attach(Observer $observer) {
        $this->observers[] = $observer;
    }

    public function detach(Observer $observer) {
        $this->observers = array_filter($this->observers, function($o) use ($observer) {
            return $o !== $observer;
        });
    }

    public function notify() {
        foreach ($this->observers as $observer) {
            $observer->update($this->state);
```

```
        }
    }

    public function setState($state) {
        $this->state = $state;
        $this->notify();
    }

    public function getState() {
        return $this->state;
    }
}

// Các Observer sẽ đăng ký nhận thông tin
class CustomerObserver implements Observer {
    public function update($state) {
        echo "Customer: Your order status has changed to $state.\n";
    }
}

class AdminObserver implements Observer {
    public function update($state) {
        echo "Admin: Order status has changed to $state.\n";
    }
}

class WarehouseObserver implements Observer {
    public function update($state) {
        echo "Warehouse: Order status has changed to $state. Prepare for
shipping.\n";
```

```
    }
}

// Sử dụng Observer
// Tạo đối tượng Order
$order = new Order();

// Tạo các đối tượng Observer
$customerObserver = new CustomerObserver();
$adminObserver = new AdminObserver();
$warehouseObserver = new WarehouseObserver();

// Gắn các Observer vào Order
$order->attach($customerObserver);
$order->attach($adminObserver);
$order->attach($warehouseObserver);

// Thay đổi trạng thái của Order
$order->setState('Processing'); // Output:
                                // Customer: Your order status has changed to
Processing.
                                // Admin: Order status has changed to
Processing.
                                // Warehouse: Order status has changed to
Processing. Prepare for shipping.

// Thay đổi trạng thái của Order lần nữa
$order->setState('Shipped');    // Output:
                                // Customer: Your order status has changed to
Shipped.
```

```
                                // Admin: Order status has changed to Shipped.
                                // Warehouse: Order status has changed to
Shipped. Prepare for shipping.

// Tháo một Observer
$order->detach($warehouseObserver);

// Thay đổi trạng thái của Order lần nữa
$order->setState('Delivered');  // Output:
                                // Customer: Your order status has changed to
Delivered.
                                // Admin: Order status has changed to Delivered.



?>
```

State

State là một mẫu thiết kế hành vi cho phép một đối tượng thay đổi hành vi của nó khi trạng thái bên trong của nó thay đổi. Điều này giúp gia tăng tính linh hoạt, phân chia trách nhiệm rõ ràng giữa khi đối tượng thay đổi trạng thái và tăng khả năng bảo trì.

Dấu hiệu nhận biết:

Bài toán kinh điển nhất có lẽ là thay đổi trạng thái đơn hàng: Khi đơn hàng đó mới được tiếp nhận, nhân viên lấy hàng sẽ được quyền thao tác với đơn hàng đó. Tuy nhiên khi đơn hàng chuyển sang trạng đã giao hàng, nhân viên giao hàng sẽ không được phép tác động vào đơn hàng nữa. Bất kể khi nào hành vi của đối tượng có thể được thay đổi bởi trạng thái của nó, bạn có thể nghĩ tới State Pattern.

Ví dụ:

```
<?php
```

```
// Interface trạng thái đơn, trong đó có cho phép đơn được thao tác hay không
interface OrderState {
    public function getStatus();
    public function allowProcessing();
}
// Các trạng thái được tách thành các class riêng biệt để phân chia trách nhiệm
xử lý logic.
//Class xử lý đơn mới, nó cho phép tác động thêm (allowProcessing trả ra true)

class NewOrderState implements OrderState {
    public function getStatus() {
        return 'New';
    }

    public function allowProcessing() {
        return true;
    }
}
//Class xử lý đơn đã được giao, nó không cho phép tác động thêm (allowProcessing
trả ra false)

class ShippedOrderState implements OrderState {
    public function getStatus() {
        return 'Shipped';
    }

    public function allowProcessing() {
        return false;
    }
}
```

```php
//Tạo class Order để lưu trạng thái

class Order {
    private $state;

    // Khởi tạo bằng trạng thái New
    public function __construct() {
        $this->state = new NewOrderState();
    }

    public function setState(OrderState $state) {
        $this->state = $state;
    }

    public function getStatus() {
        return $this->state->getStatus();
    }

    public function allowProcessing() {
        return $this->state->allowProcessing();
    }
}

// Tạo đối tượng Order
$order = new Ordert();

// Thử kiểm tra xem có được thao tác không
$order->allowProcessing(); // true
```

```php
// Chuyển trạng thái đến Shipped
$order->setState(new ShippedOrderState());
echo "Current Order Status: " . $order->getStatus() . "\n"; // Output: Shipped

// Thử kiểm tra xem có được thao tác không
$order->allowProcessing(); // false
```

Strategy

Strategy là một mẫu thiết kế hành vi cho phép bạn xác định một tập hợp các công việc tương tự nhau, được đặt trong các class riêng biệt (implement chung một interface), và cho phép chúng có thể hoán đổi lẫn nhau. Điều này giúp bạn thay đổi các class này tại thời gian chạy (Runtime) mà không làm ảnh hưởng tới chức năng của ứng dụng.

Dấu hiệu nhận biết:

Strategy rất dễ để nhận biết. Trong các tình huống mà bạn cảm thấy mình có nhiều phương án (và các phương án này tương tự nhau ở input / output) để xử lý cho một vấn đề, bạn có thể nghĩ tới Strategy. Ví dụ khi implement các phương thức vận chuyển, các phương thức thanh toán cho đơn hàng trên website thương mại điện tử. Hoặc cách mà Laravel có thể thay đổi giữa các driver khác nhau trên Cache chẳng hạn (file, Redis, array, …), cũng là nhờ sử dụng đến Strategy.

Ví dụ:

```php
<?php

// Định nghĩa Strategy Shipping (Chiến lược vận chuyển)
interface ShippingStrategy {
    public function calculate($order);
}

// Tạo ra các phương thức vận chuyển khác nhau
```

```
class GHTKShipping implements ShippingStrategy {
    public function calculate($order) {
        // Giả lập logic tính phí vận chuyển của GHTK
        return 1000;
    }
}

class GHNShipping implements ShippingStrategy {
    public function calculate($order) {
        // Giả lập logic tính phí vận chuyển của GHN
        return 12000;
    }
}

class ViettelPostShipping implements ShippingStrategy {
    public function calculate($order) {
        // Giả lập logic tính phí vận chuyển của ViettelPost
        return 15000;
    }
}

Tạo lớp Order và áp dụng Strategy
class Order {
    private $shippingStrategy;

    public function __construct(ShippingStrategy $shippingStrategy) {
        $this->shippingStrategy = $shippingStrategy;
    }

    public function setShippingStrategy(ShippingStrategy $shippingStrategy) {
```

```
        $this->shippingStrategy = $shippingStrategy;
    }

    public function calculateShipping() {
        // Giả lập một đối tượng đơn hàng
        $order = [];
        return $this->shippingStrategy->calculate($order);
    }
}

// Tạo đối tượng Order với GHTKShipping Strategy
$order = new Order(new GHTKShipping ());
echo "GHTK  Shipping Cost: $" . $order->calculateShipping() . "\n"; // Output:
GHTK Shipping Cost: 1000

// Thay đổi Strategy thành GHNShipping
$order->setShippingStrategy(new GHNShipping());
echo "GHN Shipping Cost: $" . $order->calculateShipping() . "\n"; // Output: GHN
Shipping Cost: 12000

// Thay đổi Strategy thành ViettelPostShipping
$order->setShippingStrategy(new ViettelPostShipping());
echo "ViettelPost  Shipping Cost: $" . $order->calculateShipping() . "\n"; //
Output: ViettelPost Shipping Cost: 15000
```

Template Method

Template Method là một mẫu thiết kế hành vi định nghĩa một bộ khung (template) tại class cha, và đưa việc triển khai xuống dưới các class con. Điều này giúp bạn định nghĩa các bước trong một template duy nhất, và trong đó có thể có các bước được sử dụng chung, và các bước sẽ cần được triển khai tuỳ thuộc các class con.

Dấu hiệu nhận biết:

Template Method cũng khá dễ nhận biết. Khi bạn có nghiệp vụ bao gồm một tập hợp các bước, bạn có thể đưa nó vào một Abstract Class. Các bước sử dụng chung sẽ được đưa vào các phương thức thông thường. Các bước cần sử dụng riêng sẽ là các abstract method và được triển khai tại các lớp con. Nói chung cái khó là bạn cần phân tích và chia tách nghiệp vụ của mình một cách hợp lý.

Ví dụ:

```php
<?php
// Định nghĩa bộ khung (Template) xử lý đơn hàng
abstract class OrderProcessor {
    // Phương thức template định nghĩa bộ khung
    public function processOrder() {
        $this->selectItem();
        $this->processPayment();
        $this->shipItem();
    }

    // Các phương thức trừu tượng sẽ được ghi đè bởi các lớp con
    abstract protected function selectItem();
    abstract protected function processPayment();
    abstract protected function shipItem();
}
```

```php
class OnlineOrderProcessor extends OrderProcessor {
    protected function selectItem() {
        echo "Selecting item from online store.\n";
    }

    protected function processPayment() {
        echo "Processing online payment.\n";
    }

    protected function shipItem() {
        echo "Shipping item to customer's address.\n";
    }
}

class InStoreOrderProcessor extends OrderProcessor {
    protected function selectItem() {
        echo "Selecting item from store shelf.\n";
    }

    protected function processPayment() {
        echo "Processing in-store payment.\n";
    }

    protected function shipItem() {
        echo "Customer takes the item home.\n";
    }
}

// Xử lý đơn hàng trực tuyến
```

```php
$onlineOrder = new OnlineOrderProcessor();
echo "Processing online order:\n";
$onlineOrder->processOrder();

// Xử lý đơn hàng tại cửa hàng
$inStoreOrder = new InStoreOrderProcessor();
echo "Processing in-store order:\n";
$inStoreOrder->processOrder();
?>
```

Bonus: Dependency Injection

(Section này mình lấy hoàn toàn từ cuốn “Thiết kế hướng nghiệp vụ với Laravel” của mình):
Hãy cùng nhắc lại một bài toán kinh điển là cách thiết kế "bóng đèn - đui đèn": Một mẫu thiết kế tốt cho phép bạn linh hoạt thay đổi giữa bóng và đui, trong khi thiết kế tồi khiến cái đèn của bạn không thể thay thế (tight coupling), khiến bạn buộc phải mua một cái đèn mới, thay vì chỉ cần thay bóng. Hãy tưởng tượng cái đèn nhà vệ sinh của mình bị hỏng, và nếu phải thay tuốt tuồn tuột từ trên xuống dưới thì thực sự là cơn ác mộng !
Thành thạo các design pattern cơ bản là điều kiện tiên quyết để bạn phát triển kỹ năng coding của mình. DI là giải pháp thiết kế cho chính bài toán bóng đèn - đui đèn bên trên. Nếu bạn làm Laravel, chắc chắn bạn đã nằm lòng khái niệm Service Container - một cái hộp thần kỳ giúp ứng dụng của bạn trở nên uyển chuyển một cách thú vị thông qua việc binding các lớp trừu tượng. Tuy nhiên, nếu bạn là người mới bắt đầu, hầu hết chúng ta chỉ cảm nhận DI qua việc inject interface từ __construct, binding trong Service Container. Rất nhiều bạn thắc mắc rốt cuộc DI có gì hơn việc sử dụng từ khóa new trong chính class đó (Việc inject nhiều thực sự là rối rắm, đúng không ?).

Nếu không sử dụng DI, bài toán bóng đèn của chúng ta có thể được thực thi như thế này:

```
// Class đèn
class Lamp {
    public function __construct()
    {
        // Class bóng đèn được gắn chặt vào class Đèn qua từ khóa new
        $this->bulb = new Bulb();
    }

    public function turnOn() { ... }
}
```

Về cốt lõi, DI đại diện cho triết lý "Composition Over Inheritance". Một đối tượng nên được tạo thành từ các viên gạch nhỏ hơn từ bên ngoài, thay vì chúng ta tự khởi tạo trong chính đối tượng đó. Hãy quay trở lại với ví dụ trên: Nếu ta đã có một cái đui, ta có thể inject bất kỳ cái đèn nào vào cái đui đó

để thu được một cái đèn như ý muốn. Cần đèn xanh, có. Cần đèn vàng, có. Cần đèn nhấp nháy, easy (Đoạn này nghe quen quen phải không, DI giúp chúng ta tạo nên tính Đa hình (Polymorphism). Ta có thế chế ra đủ loại đèn mà mình muốn). Nếu cái đui gắn chặt vào một cái đèn (giống như nó tự khởi tạo new một cái đèn bên trong nó), rõ ràng là rất tồi tệ (ko thể thay đèn mà sẽ phải vứt nguyên mua cái mới). Khi ta tách được đèn và đui, việc của chúng ta là thiết kế một lớp trừu tượng, ở đây chính là cái trôn để xoáy đèn vào đui của nó. Cái đèn chỉ cần "implement" "interface đui đèn", và miễn là nó implement interface này, nó sẽ luôn xoáy được vào đui, và chắc chắn nó sẽ hoạt động được. Rất trực quan và dễ hiểu đúng không các bạn ?
Ví dụ giờ mình viết class Lamp bên trên như thế này:

```
class Lamp
{
    // Injection thông qua Interface Bulb, hoặc cũng có thể là Abstract Class
Bulb
    public function __construct(
        Bulb $bulb
    ) {
        $this->bulb = $bulb;
  }

    public function turnOn()
    {
        $this->bulb->turnOn();
  }
}
```

Giờ nếu muốn tạo một cái đèn màu xanh, bạn có thể khởi tạo như thế này:

```
$lamp = new Lamp(new GreenBulb());
```

Một cái đèn màu đỏ, thì sẽ là:

```
$lamp = new Lamp(new RedBulb());
```

Hãy nhớ nhé: “Composition over Inheritance” là nguyên tắc mà các lớp phải đạt được hành vi đa hình và tái sử dụng code bằng thành phần của chúng thay vì kế thừa từ lớp cơ sở hoặc lớp cha.

DI giúp bạn test dễ hơn nhiều. Tôi muốn test cái đui, hoặc đèn, tôi có sẵn đèn mẫu (Mocking Object), cắm vào, sáng, và thế là pass Test. Nếu không có DI, thì class sẽ rất khó để test được.
Việc áp dụng triệt để triết lý này khi code giúp chúng ta có tư duy phân tách code thành các đơn vị nhỏ, để lắp ghép nó thành những cái đèn của chúng ta. Nhiều cái đèn thì có thể làm thành một ngọn hải đăng chẳng hạn. Và nếu có cái đèn nào vỡ / hỏng (bug), ta dễ dàng cô lập, thay thế, và sửa chữa nó. Ở các section bên dưới, tư duy ưu tiên việc chia nhỏ, lắp ghép sẽ đóng vai trò rất quan trọng trong mindset thiết kế của mình.

Lời kết

Bạn đã đọc hết cuốn “OOP cho người đi làm” của mình.
Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. OOP là một trong những kiến thức nền tảng cực kỳ quan trọng, cho dù bạn làm với techstack nào. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường học lập trình nhé □.
Cám ơn bạn rất nhiều !
Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:
Email: huynt57@gmail.com
Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !