**Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 1)**

# Mục lục

* Phần 1: [Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 1)](https://viblo.asia/p/lap-trinh-huong-doi-tuong-voi-php-va-nhung-dieu-can-biet-phan-1-GrLZDbzV5k0) 1. Các đặc điểm cơ bản của lập trình hướng đối tượng. Chúng được thể hiện như thế nào trong PHP. 2. Sự khác biệt giữa Abstract Class và Interface. 3. Thế nào là một hàm static. Phân biệt cách dùng từ khoá static::method() với self::method().
* Phần 2: [Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 2)](https://viblo.asia/p/lap-trinh-huong-doi-tuong-voi-php-va-nhung-dieu-can-biet-phan-2-Eb85oXq0K2G) 4. Thế nào là Trait 5. Thế nào là Namespaces 6. Thế nào là magic functions 7. Tìm hiểu về các quy tắc trong PSR2
* Phần 3: [Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 3)](https://viblo.asia/p/lap-trinh-huong-doi-tuong-voi-php-va-nhung-dieu-can-biet-phan-3-bJzKmWBPl9N) 8. Các phương pháp thiết kế hướng đối tượng (SOLID).

# Nội dung

## 1. Các đặc điểm cơ bản của lập trình hướng đối tượng. Chúng được thể hiện như thế nào trong PHP.

Lập trình hướng đối tượng có 4 đặc điểm cơ bản sau.

### Tính đóng gói

* Tính đóng gói (encapsulation) "đóng gói" thuộc tính và phương thức của đối tượng (hoặc lớp) thông qua việc giới hạn quyền truy cập (hoặc thay đổi) giá trị của thuộc tính hoặc quyền gọi phương thức. Nói cách khác tính đóng gói cho phép kiểm soát quyền truy cập (và thay đổi) giá trị của thuộc tính hoặc quyền gọi phương thức của đối tượng (hoặc lớp) và đối tượng (hoặc lớp) con.
* Trong PHP việc đóng gói được thực hiện nhờ sử dụng các từ khoá public, private và protected:
  + public: Cho phép truy cập (và thay đổi giá trị) của thuộc tính và phương thức ở mọi phạm vi, có thể hiểu là mang đặc tính cộng đồng.
  + protected: Chỉ cho phép truy cập (hay thay đổi) giá trị của thuộc tính và phương thức ở phạm vi của đối tượng con (hoặc lớp con), có thể hiểu là mang đặc tính dòng họ.
  + private: Chỉ cho phép truy cập (hay thay đổi) giá trị của thuộc tính và phương thức ở phạm vi của đối tượng (hoặc lớp), có thể hiểu là mang đặc tính gia đình.

### Tính kế thừa

* Tính kế thừa trong lập trình hướng đối tượng cho phép một lớp (class) có thể kế thừa các thuộc tính và phương thức từ các lớp khác đã được định nghĩa. Lớp được kế thừa còn được gọi là lớp cha và lớp kế thừa được gọi là lớp con.
* Điều này cho phép các đối tượng có thể tái sử dụng hay mở rộng các đặc tính sẵn có mà không phải tiến hành định nghĩa lại.
* Trong PHP, một lớp có thể kế thừa từ một lớp khác, việc kế thừa được thực hiện thông qua sử dụng từ khóa extends. Đối tượng thuộc lớp con sẽ có các thuộc tính và phương thức protected và public của lớp mà nó kế thừa.

### Tính trừu tượng.

* Tính trừu tượng (abstraction) trong lập trình hướng đối tượng giúp giảm sự phức tạp thông qua việc tập trung vào các đặc điểm trọng yếu hơn là đi sâu vào chi tiết.
* Như vậy khi tương tác với đối tượng chỉ cần quan tâm đến các thuộc tính, phương thức cần thiết. Chi tiết về nội dung không cần chú ý đến.
* PHP có abstract class và interface để trừu tượng hóa các đối tượng. Ví dụ khi ta tạo một lớp (class) dùng đại diên cho các tài khoản tiền gửi ngân hàng của các khách hàng và đặt tên cho lớp này là BankAccount. Lớp này có hai thuộc tính là $balance và $interest dùng để lưu dữ liệu số tiền dư và lãi suất tiền gửi của tài khoản.

class BankAccount {

public $balance;

public $interest;

}

Tiếp theo ta thêm các phương thức gửi tiền deposit và rút tiền withdraw như sau:

class BankAccount

{

public $balance; // số dư tài khoản

public $interest; // lãi suất

public function deposit ($amount)

{

// TODO

}

public function withdraw ($amount)

{

// TODO

}

}

Với tính trừu tượng (abstraction) thì toàn bộ sự phức tạp của việc xử lý quá trình gửi tiền và rút tiền sẽ được thực hiện trong 2 phương thức deposit và withdraw. Các lập trình viên không cần phải quan tâm tới sự phức tạp (hay nội dung chi tiết) của việc xử lý các công việc gửi tiền và rút tiền trên mà chỉ cần biết mục đích của từng phương thức là gì.

Dưới đây là một cách thực hiện (implementation) của phương thức deposit:

// nạp tiền vào tài khoản

public function deposit ($amount)

{

if ($amount < 50000) { // số tiền nạp vào dưới mức tối thiểu 50 ngàn

return "Error! The minimum amount is 50";

}

if ($amount > 100000000) { // tài khoản này cho phép nạp tối đa 100 triệu một lần

return "Error! You exceed the maximum amount, please upgrade your account";

}

$this->balance += $amount; // tăng số dư tài khoản

}

Với tính trừu tượng thì lập trình viên chỉ cần quan tâm tới mục đích của phương thức deposit là để nạp tiền vào tài khoản. Toàn bộ chi tiết của quy trình xử lý gửi tiền sẽ được thực hiện ở bên trong phương thức deposit.

### Tính đa hình

* Thể hiện qua việc có thể định nghĩa một đặc tính, hoặc phương thức cho một loạt các đối tượng gần giống nhau. Nhưng khi thực hiện thì các đối tượng khác nhau sẽ có cách thực hiện khác nhau và cho ra kết quả khác nhau.
* Tính đa hình (polymorphism) trong lập trình hướng đối tượng cho phép các lớp con có thể viết lại (override) các thuộc tính hoặc phương thức từ lớp cha. Ví dụ: Đối tượng "hình vuông" và "hình tròn" có chung 1 phương thức là "chu\_vi". Khi gọi phương thức này thì với mỗi đối tượng sẽ có 1 công thức tính khác nhau và cho ra kết quả khác nhau. Trong PHP:
  + Các lớp con có thể viết lại hoặc mở rộng phương thức của lớp cha mà nó kế thừa.
  + Các class cùng implement một interface nhưng chúng có cách thức thực hiện khác nhau cho các method của interface đó.
  + Qua đó cùng 1 phương thức sẽ cho kết quả khác nhau khi được gọi bởi các đối tượng thuộc lớp khác nhau.

## 2. Sự khác biệt giữa  Abstract Class và Interface.

Interface và Abstract class là 2 khái niệm cơ bản trong lập trình OOP. Nhưng phân lớn mọi người cảm thấy mơ hồ và lẫn lộn 2 khái niệm này. Vậy chúng là gì, khác nhau như nào? Tại sao dùng cái này Interface mà không phải Abstract và ngược lại ??? Bỏ qua tất cả những phần về lý thuyết của việc tạo một abstract classvà interface. Bạn không cần quan tâm nhiều đến việc abstract có thể khai báo những gì, hay interface có được phép định nghĩa nội dung phương thức hay không. Điểm cơ bản khi bạn được hỏi về sự khác biệt giữa chúng là gì? Đó chính là mục đích mà chúng được sử dụng: – Abstract class: là một class cha cho tất cả các class có cùng bản chất. Bản chất ở đây được hiểu là kiểu, loại, nhiệm vụ của class. Hai class cùng hiện thực một interface có thể hoàn toàn khác nhau về bản chất. Hiểu đơn giản như một thằng con (child class) chỉ có thể là con của một thằng cha, có tính cách giống cha (abstract class) nó. – Interface: là một chức năng mà bạn có thể thêm và bất kì class nào. Từ chức năng ở đây không đồng nghĩa với phương thức (hoặc hàm). Interface có thể bao gồm nhiều hàm/phương thức và tất cả chúng cùng phục vụ cho một chức năng. Vậy, bạn không nên nhầm lẫn khi nói về việc một class được implement hay extend. Nhiều người thường hay đồng nhất là không phân biệt hai từ này, nhưng chính chúng đã nói lên sự khác biệt giữa interface và abstract class. Bạn chỉ có thể thừa kế (extend) từ một class và chỉ có thể hiện thực (implement) các chức năng (interface) cho class của mình.

Cuối cùng, cũng nên liệt kê các điểm khác biệt giữa hai khái niệm này để bạn có thể sử dụng được khi cần thiết. Các điểm khác biệt này có thể khác nhau tùy vào ngôn ngữ mà bạn sử dụng. Vì vậy bạn chỉ cần nhớ các điểm căn bản sau:

|  | **Interface** | **Abstract class** |
| --- | --- | --- |
| Multiple inheritance | Một class có thể hiện thực nhiều interface.(tạm coi là thừa kế) | Không hỗ trợ đa thừa kế |
| Access Modifiers | Mọi phương thức, property đều mặc định là public. | Có thể xác định modifier. |
| Adding functionality | Mọi phương thức, property của interface cần được hiện thực trong class. | Không cần thiết. |
| Fields and Constants | Không | Có |

Nếu một class implements nhiều interface mà các interface có những phương thức cùng tên thì sẽ không lỗi nếu các phương thức đó truyền vào số lượng biến bằng nhau. Còn nếu các phương thức đó khác số lượng biến truyền vào thì sẽ sinh ra lỗi "Fatal error".

## 3. Thế nào là một phương thức static. Phân biệt cách dùng từ khoá static::method() với self::method().

### Phương thức static.

* Phương thức static là phương thức có thể truy cập mà không cần khởi tạo một đối tượng của class.
* Phương thức static gắn liền với class hơn là với object (là thành phần khởi tạo class bằng từ khóa new), đây là những phương thức chỉ có một, có địa chỉ xác định và không thay đổi địa chỉ trên vùng nhớ (tĩnh).
* Khi chương trình chạy, nó sẽ được sinh ra đầu tiên trước tất cả các truy nhập tới nó và tồn tại cho tới khi chương trình kết thúc.
* Khai báo một phương thức static trong một class như sau:

public static function staticMethod()

{

//TODO

}

Việc thực thi hàm static trong class có thể thực hiện bằng lệnh: static::staticMethod(), self::staticMethod() hoặc $this->staticMethod(), trong đó self và static là đại diện của class, còn $this là đại diện của object. Trong phương thức static không thể gọi phương thức hoặc thuộc tính non-static. Nhưng phương thức non-static có thể gọi phương thức hoặc thuộc tính static. Bởi vì có thể hiểu đơn giản như sau:

* Phương thức static có thể gọi ngay cả khi chưa khởi tạo object, do đó nếu phương thức staticgọi đến một phương thức non-static thì khi chưa khởi tạo object, sẽ không có biến $this (là đại diện của object) để gọi đến phương thức 'non-static`.
* Đương nhiên phương thức non-static luôn luôn có thể gọi đến phương thức static vì phương thức static đã tồn tại ngay từ khi chạy chương trình, khi object chưa được khởi tạo.

### Phân biệt cách dùng từ khoá static::method() với self::method().

Như ở trên chúng ta có nhắc đến từ khóa static::method() với self::method(), chúng đều là đại diện cho class để gọi đến các phương thức và thuộc tính static. Câu hỏi đặt ra là Thế tại sao lại phải có tới 2 từ khóa?. Nếu chỉ đơn thuần như trong nội bộ class thì cả 2 từ khóa trên đều cho ra kết quả giống nhau. Nhưng chúng có thể sẽ khác nhau nếu khi class này extends từ class khác. Để rõ hơn sự khác nhau giữa 2 từ khóa này, chúng ta cùng xem xét ví dụ sau.

class Person

{

protected static $className = 'classPerson';

public static function getClassName()

{

echo self::$className; // từ khóa self

echo ' - ';

echo static::$className; // từ khóa static

}

}

class Girl extends Person

{

protected static $className = 'classGirl';

}

Person::getClassName(); //classPerson - classPerson

Girl::getClassName(); // classPerson - classGirl

* Trong ví dụ trên class Girl đã extends từ class Person, tức là nó mang toàn bộ các thuộc tính protected và public của class Person, trong class Person ta có thuộc tính $className = 'classPerson', class Girl khi kế thừa đã override lại thuộc tính $className = 'classGirl'.
* Khi ta gọi Girl::getClassName() thì từ khóa sefl trả về kết quả là 'classPerson', điều này cho thấy từ khóa self đại diện cho chính đối tượng khai báo nó, phương thức getClassName() được khai báo tại class Person nên thuộc tính self trong phương thức này sẽ đại diện cho class Person nên trả về kết quả là 'classPerson'.
* Còn từ khóa static nó sẽ đại diện cho chính đối tượng đang gọi đến nó, Girl::getClassName() thì class Girl đang gọi đến phương thức getClassName() nên từ khóa static sẽ trả về kết quả là classGirl. Có thể hiểu static tương đương với $this, chỉ khác là từ khóa static sẽ gọi đến các phương thức staticcủa class, còn $this thì sẽ gọi đến cả phương thức non-static và static của object.

Các bạn xem thêm ví dụ dưới đây để hiểu rõ hơn về từ khóa static và self.

class Person

{

public static function getSeflObject()

{

return new self;

}

public static function getStaticObject()

{

return new static;

}

}

class Girl extends Person

{

}

echo get\_class(Girl::getSeflObject()); //Person

echo get\_class(Girl::getStaticObject()); //Girl

Tóm tắt lại như sau

* self: đại diện cho class khai báo nó.
* static: đại diện cho class gọi đến nó.

# Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 2)

# Nội dung

## 4. Thế nào là Trait

PHP là ngôn ngữ chỉ hỗ trợ đơn kế thừa trong hướng đối tượng thế nên bạn muốn sử dụng lại source code một cách nhiều lần là rất khó khăn. Và để khắc phục điều đó thì từ PHP 5.4 trở lên, PHP đã hỗ trợ chúng ta Traits. Một khái niệm được dùng rất là nhiều trong các framework PHP hiện nay, điển hình là Laravel.

### Trait là gì?

* Trait là một module giúp cho chúng ta có thể sử dụng lại các phương thức được khai báo trong traitvào các class khác nhau một cách đơn giản hơn là kế thừa như trước.
* Các đặc điểm của Traits:
  + Trait có chức năng gom lại các phương thức và thuộc tính mà chúng ta muốn sử dụng lại nhiều lần.
  + Traits như một abstract class ( đều không thể khởi tạo được) nhưng không hoàn toàn giống nhau.
  + Các phương thức trong Trait có thể bị override lại trong class sử dụng nó.
* Ưu điểm của Trait:
  + Giảm việc lặp code đáp ứng được nguyên tắc (DRY - Don't Repeat Yoursefl).
  + Khắc phục được điểm yếu đơn kế thừa của PHP.
* Nhược điểm của Trait: Nhược điểm duy nhất mà mình thấy được khi sử dụng traits đó là sẽ gây khó khăn có chúng ta đọc được các phương thức từ một class có sử dụng trait.

### Khai báo và sử dụng trait.

Để khai báo một trait trong PHP các bạn sử dụng cú pháp:

trait TraitName

{

public function methodTrait($param)

{

// TODO

}

}

Bạn cũng có thể sử dụng trait lồng nhau.

trait A

{

//TODO

}

trait B

{

use A;

// TODO

}

Để sử dụng trait trong class thì các bạn chỉ cần sử dụng cú pháp:

class ClassName

{

use TraitName;

//TODO

}

## 5. Thế nào là Namespaces (không gian tên).

* Namespace là một cơ chế cho phép chúng ta phân nhóm các thực thể như class, object, function thành những nhóm riêng biệt, mỗi nhóm đó được đặt cho một cái tên.
* Việc dùng Namespace sẽ để tránh xung đột khi sử dụng nhiều thư viện, trong code có các class với tên giống nhau.
* Namespace có thể hiểu đơn giản như là thư mục trên các hệ điều hành, để phân cấp và phân biệt các thực thể có tên giống nhau.

## 6. Thế nào là magic functions

Magic functions là các hàm đặc biệt được tự động thực thi khi thực hiện một số hành động nhất định lên đối tượng của class đó. Magic functions luôn có dạng \_\_<method\_name>.

Ưu điểm của Magic functions:

* Giúp cho chúng ta tùy biến được các hành vi.
* Nó giúp cho chúng ta có thể khởi tạo một đối tượng theo cách mình muốn.

Nhược điểm: Một magic functions có tốc độ chậm hơn các phương thức bình thường.

Một số Magic functions:

* \_\_construct(): Thực hiện khi khởi tạo đối tượng.
* \_\_destruct(): Thực hiện khi hủy bỏ đối tượng.
* \_\_get(), \_\_set(), \_\_unset(): Khi cố gắng truy cập, thay đổi, xóa thuộc tính không có sẵn trong đối tượng.
* \_\_call(), \_\_callStatic(): Khi gọi hàm không có sẵn.
* \_\_toString(): Thực hiện khi được sử dụng như string.
* \_\_invoke(): Khi đối tượng được sử dụng như một hàm.
* \_\_sleep(), \_\_awake(): Khi gọi hàm serialize, unserialize lên đối tượng.
* \_\_clone(): Khi clone đối tượng.
* \_\_set\_state(), \_\_debugInfo(): Khi gọi var\_export, var\_dump lên đối tượng.

## 7. Tìm hiểu về các quy tắc trong PSR2

### Code theo chuẩn PSR2 phải tuân thủ các quy tắc sau

* Code PHẢI tuân thủ PSR-1.
* Code PHẢI sử dụng 4 ký tự space để lùi khối (không dùng tab).
* Mỗi dòng code PHẢI dưới 120 ký tự, NÊN dưới 80 ký tự.
* PHẢI có 1 dòng trắng sau namespace, và PHẢI có một dòng trắng sau mỗi khối code.
* Ký tự mở lớp { PHẢI ở dòng tiếp theo sau tên class, và đóng lớp } PHẢI ở dòng tiếp theo của thân class.
* Ký tự { cho hàm PHẢI ở dòng tiếp theo sau tên hàm, và ký tự } kết thúc hàm PHẢI ở dòng tiếp theo của thân hàm.
* Các visibility (public, private, protected) PHẢI được khai báo cho tất cả các hàm và các thuộc tính của lớp.
* Các từ khóa điều khiển khối(if, elseif, else) PHẢI có một khoảng trống sau chúng, hàm và lớp thì KHÔNG ĐƯỢC làm như vậy.
* Mở khối { cho cấu trúc điều khiển PHẢI trên cùng một dòng và đóng khối này } với ở dòng tiếp theo của thân khối.
* Hằng số true, false, null PHẢI viết với chữ thường.
* Từ khóa extends và implements PHẢi cùng dòng với class.
* Implements nhiều lớp, thì mỗi lớp trên một dòng.
* Keyword var KHÔNG ĐƯỢC dùng sử dụng khai báo property.
* Tên property KHÔNG NÊN có tiền tố \_ nhằm thể hiện thuộc protect hay private.
* Tham số cho hàm, phương thức: KHÔNG được thêm space vào trước dấu , và PHẢI có một space sau ,.
* Các tham số CÓ THỂ trên nhiều dòng, nếu làm như vậy thì PHẢI mỗi dòng 1 tham số.
* abstract, final PHẢI đứng trước visibility, còn static PHẢI đứng sau.
* Phải có 1 khoảng trắng trước và sau phép toán, khi ép kiểu thì phải có 1 khoảng trắng ngăn cách giữa kiểu dữ liệu và biến được ép kiểu.
* Đối với khối lệnh switch case thì case phải lùi 4 khoảng trắng so với switch, và các lệnh trong casecũng phải lùi 4 khoảng trắng so với case. Phải có từ khóa break hoặc return, trong trường hợp nào không có thì phải comment //no break.

### Các quy tắc liên quan

Tiêu chuẩn PSR-1 chuẩn cơ bản:

* Các file code PHẢI sử dụng thẻ <?php hoặc <?.
* File code PHP sử dụng encode: UTF-8 without BOOM.
* Các Files NÊN hoặc dùng để khai báo các thành phần PHP (các lớp, hàm, hằng ...) hoặc dùng với mục đích làm hiệu ứng phụ (như include, thiết lập ini cho PHP ...), nhưng KHÔNG NÊN dùng cả 2 cùng lúc trong 1 file.
* Các Namespace và classes PHẢI theo chuẩn "autoloading" PSR: [PSR-0 và PSR-4].
* Tên lớp PHẢI có dạng NameClass (chứ không được nameclass, Nameclass, namClass ...).
* Hằng số trong class tất cả PHẢI viết HOA và chia ra bởi dấu \_.
* Tên phương thức của lớp PHẢI ở dạng camelCase (từ đầu viết thường, ví dụ: helloWorld).

Tiêu chuẩn PSR-0 chuẩn Autoloading:

* Một namespace và class đầy đủ điều kiện (fully qualified) PHẢI có cấu trúc như sau \<Vendor Name>\(<Namespace>\)\*<Class Name>
* Mỗi namespace PHẢI có một top-level namespace ("Vendor name" - có thể hiểu là namespace gốc)
* Mỗi namespace CÓ THỂ có nhiều sub-namespace (namespace con)
* Mỗi namespace phân biệt được chuyển đến thư mục phân biệt khi load từ hệ thống.
* Mỗi kí tự \_ trong TÊN CLASS được chuyển đến một thư mục riêng biệt. Kí tự \_ không có ý nghĩa đặc biệt gì trong namespace.
* Namespace và class đầy đủ điều kiện có hậu tố .php khi load từ hệ thống
* Kí tự chữ cái trong Vendor name, namespaces, và class CÓ THỂ là tổ hợp của kí tự thường và hoa.

Tiêu chuẩn PSR-4 quy tắc lưu trữ file được load:

* "Class" ở đây ám chỉ cho classes, interface, traits và những cấu trúc tương tự khác.
* Tên xác định đầy đủ của 1 class có dạng: \<NamespaceName>(\<SubNamespaceNames>)\*\<ClassName>.
  + Tên xác định đầy đủ PHẢI có một namespace gốc (hiểu là tên vendor).
  + Tên xác định đầy đủ CÓ THỂ có một hoặc nhiều namespace con.
  + Tên đầy đủ nó PHẢI có một tên class kết thúc (ClassName).
  + Kí tự \_ không có ý nghĩa đặc biệt trong tên class đầy đủ.
  + Kí tự chữ cái trong tên class đầy đủ CÓ THỂ là tổ hợp của kí tự thường và hoa.
  + Tất cả tên class PHẢI được tham chiếu trong một cách phù hợp.
* Khi nạp một file thì nó phải tương ứng với một tên xác định đầy đủ của class.
  + Một loạt liên tiếp của 1 hoặc nhiều leading namespace và sub-namespace không bao gồm các dấu phân cách trong tên class đầy đủ (có thể hiểu là "namespace prefix") tương ứng với ít nhất 1 "thư mục cơ sở".
  + Một loạt liên tiếp sub-namespace sau "namespace prefix" tương ứng với 1 thư mục con trong "thư mục cơ sở", mỗi namespace riêng biệt tương ứng với thư mục riêng biệt. Tên thư mục con PHẢI phù hợp với sub-namespace.
  + Tên class kết thúc tương ứng với tên file <ClassName>.php
  + Triển khai tự động nạp PHẢI KHÔNG throw exception, PHẢI KHÔNG gây ra lỗi ở bất kì level, và KHÔNG NÊN trả về giá trị.

# Lập trình hướng đối tượng với PHP và những điều cần biết (Phần 3)

# Nội dung

Lập trình hướng đối tượng (OPP) có 4 tính chất cơ bản, nếu chưa rõ 4 tính chất này thì các bạn có thể đọc phần 1 trong loạt bài viết này. Với các tính chất này của OOP giúp chúng ta xây dựng được các chương trình giải quyết được nhiều vấn đề cụ thể khác nhau trong thế giới thực. Tuy nhiên, để vận dụng và phối hợp các tính chất này với nhau để tăng hiệu quả của ứng dụng thì quả thật cũng phải cần một thời gian khá dài và không phải ai cũng nắm được. Một trong những nguyên tắc (chỉ dẫn) để giúp chúng ta xây dựng được các ứng dụng OOP hiệu quả hơn đó là phương pháp thiết kế SOLID (đây là từ viết tắt), nó là một bộ 5 chỉ dẫn đã được nhắc tới từ lâu bởi các nhà phát triển phần mềm và được tổng hợp và phát biểu thành nguyên tắc bởi Robert C. Martin, cũng chính là tác giả của cuốn sách The Clean Coder nổi tiếng. Năm nguyên tắc (chỉ dẫn) này bao gồm:

* S: Single Responsibility principle (nguyên lý đơn nhiệm).
* O: Open-Closed principle (nguyên lý mở rộng - hạn chế).
* L: Liskov substitution principle (nguyên lý thay thế Liskov).
* I: Interface segregation principle (nguyên lý giao diện phân biệt - hay phân tách interface).
* D: Dependency inversion principle (nguyên lý nghịch đảo phụ thuộc).

Đây là những nguyên lý được đúc kết bởi máu xương vô số developer, rút ra từ hàng ngàn dự án thành công và thất bại. Một project áp dụng những nguyên lý này sẽ có code dễ đọc, dễ test, rõ ràng hơn. Và việc quan trọng nhất là việc maintainace code sẽ dễ hơn rất nhiều (Ai có kinh nghiệm trong ngành IT đều biết thời gian code chỉ chiếm 20-40%, còn lại là thời gian để maintainance: thêm bớt chức năng và sửa lỗi). Nắm vững những nguyên lý này, đồng thời áp dụng chúng trong việc thiết kế + viết code sẽ giúp bạn tiến thêm 1 bước trên con đường thành senior nhé.

## 8. Các phương pháp thiết kế hướng đối tượng (SOLID).

### S: Single Responsibility principle (nguyên lý đơn nhiệm).

Mỗi class chỉ nên chịu trách nhiệm về một nhiệm vụ cụ thể nào đó mà thôi. Hay chỉ có thể sửa đổi class với 1 lý do duy nhất.

Giả sử rằng nhân viên của một công ty phần mềm cần phải làm 1 trong 3 việc sau đây: lập trình phần mềm (developer), kiểm tra phần mềm (tester), bán phần mềm (salesman). Mỗi nhân viên sẽ có một chức vụ và dựa vào chức vụ sẽ làm công việc tương ứng. Khi đó bạn có nên thiết kế lớp “Employee” với thuộc tính “position” và 3 phương thức developSoftware(), testSoftware() và saleSoftware() không?

class Employee

{

public $position;

public function developSoftware(){};

public function testSoftware(){};

public function saleSoftware(){};

}

Có lẽ bạn sẽ thấy rằng cách trên là nhanh và có thể xử lý được bài toán ở trên. Nhưng câu trả lời cho cách thiết kế trên là không. Thử hình dung nếu có thêm một chức vụ nữa là quản lí nhân sự, không lẽ ta phải vào sửa lại lớp “Employee” và thêm phương thức mới vào? Nếu có thêm 10 chức vụ nữa thì sao? Ngoài ra, khi đối tượng được tạo ra sẽ dư thừa rất nhiều phương thức: Developer thì đâu cần dùng hàm testSoftware() và saleSoftware() đúng không nào, lỡ may dùng lầm phương thức cũng sẽ gây hậu quả khôn lường.

Nguyên tắc này nói rằng: mỗi lớp thực hiện 1 trách nhiệm. Ta sẽ tạo 1 lớp trừu tượng là “Employee” có phương thức là “working()”, từ đây bạn kế thừa ra 3 lớp cụ thể là Developer, Tester và Salesman. Ở mỗi lớp này bạn sẽ implement phương thức “working()” cụ thể tuy theo nhiệm vụ của từng người.

abstract class Employee

{

abstract public function working();

}

class Developer extends Employee

{

public function working()

{

// TODO

}

}

// Tương tự cho 2 class Tester và Salesman

Với cách thiết kế này thì ta đã tuân thủ nguyên tắc S trong SOLID, mỗi class Develop, Tester và Salesman sẽ chỉ đảm nhiệm một nhiệm vụ duy nhất.

### O: Open-Closed principle (nguyên lý mở rộng - hạn chế).

Có thể thoải mái mở rộng 1 class, nhưng hạn chế sửa đổi class đó.

Nhìn vào phát biểu trên, chúng ta thấy rằng, có hai vấn đề cần lưu tâm trong nguyên lí này:

* Hạn chế sửa đổi: Ta không nên chỉnh sửa source code của một module hoặc class có sẵn, vì sẽ ảnh hưởng tới tính đúng đắn của chương trình.
* Ưu tiên mở rộng: Khi cần thêm tính năng mới, ta nên kế thừa và mở rộng các module/class có sẵn thành các module con lớn hơn. Các module/class con vừa có các đặc tính của lớp cha (đã được kiểm chứng đúng đắn), vừa được bổ sung tính năng mới phù hợp với yêu cầu.

Hình dung rằng “tiện nghi sống” của bạn đang là 1 căn nhà, bây giờ bạn muốn có thêm 1 tính năng là “hồ bơi” để thư giãn. Bạn có 2 cách để làm điều này:

* Cách 1: thay đổi hiện trạng của căn nhà, xây thêm 1 tầng nữa để làm hồ bơi.
* Cách 2: không làm thay đổi căn nhà đang có, mua thêm 1 mảnh đất cạnh nhà bạn và xây hồ bơi ở đó.

Mặc dù cả 2 cách đều giải quyết được vấn đề nhưng cách đầu tiên có vẻ rất thiếu tự nhiên và kì cục. Cách này làm thay đổi hiện trạng của căn nhà, và nếu không cẩn thận có thể làm hư luôn những thứ đang có. Cách thứ 2 an toàn hơn rất nhiều và đáp ứng tốt được nhu cầu muốn có hồ bơi của bạn.

Nguyên tắc này có ý rằng: không được thay đổi hiện trạng của các lớp có sẵn, nếu muốn thêm tính năng mới, thì hãy mở rộng bằng cách kế thừa để xây dựng class mới. Làm như vậy sẽ tránh được các tình huống làm hỏng tính ổn định của chương trình đang có.

Bây giờ giả sử bạn muốn xây dựng một chương trình tạo ưu đãi cho customer. Ban đầu bạn chỉ có 2 loại customer là 'normal' và 'vip'. Chúng ta sẽ giảm cho khách hàng 'vip' và 'normal' theo một mức nào đó bằng phương thức getDiscount().

class Customer

{

protected $typeCustomer; // 'normal' hoặc 'vip'

public function setTypeCustomer($type)

{

if ($type == 'vip') {

$this->typeCustomer = 'vip';

} else {

$this->typeCustomer = 'nomal';

}

}

public function getDiscount($total)

{

if ($this->typeCustomer == 'vip') {

return $total \* 0.2;

} else {

return $total \* 0.1;

}

}

}

Bây giờ ta muốn thêm một kiểu customer mới là "silver", thì ta cần thay đổi lại hàm "setTypeCustomer()" và "getDiscount()" để tính toán lại cho loại customer mới này, như thế là ta đã thay đổi class "Customer". Nếu giờ muốn thêm nhiều loại customer mới nữa thì sao? Ta lại phải tiếp tục thay đổi class "Customer". Như thế liệu có đảm bảo các đoạn code khác đã sử dụng class này có hoạt động bình thường. Đây chính là điều mà nguyên lý này muốn nói để giải quyết vấn đề này. Ta có thể giải quyết vấn đề mở rộng bằng cách sau:

abstract class Customer

{

abstract public function getDiscount($total);

}

class VipCustomer extends Customer

{

public function getDiscount($total)

{

return $total \* 0.2;

}

}

class SilverCustomer extends Customer

{

public function getDiscount($total)

{

return $total \* 0.15;

}

}

class NormalCustomer extends Customer

{

public function getDiscount($total)

{

return $total \* 0.1;

}

}

Với các trên thì ta thì khi thêm một loại customer mới thì ta có thể extends class "Customer", có thể thoải mái mở rộng các loại customer mà không cần thay đổi lại class "Customer". Lớp “Customer” bây giờ được hiểu là “Đóng” với mọi thay đổi, nhưng nó luôn “Mở” đối với việc mở rộng các thuộc tính hay phương thức của lớp “Customer".

### L: Liskov substitution principle (nguyên lý thay thế Liskov).

Trong một chương trình, các object của class con có thể thay thế class cha mà không làm thay đổi tính đúng đắn của chương trình

Giờ ta có ví dụ sau: Giả sử, ta muốn viết một chương trình để mô tả các loài chim bay. Đại bàng, chim sẻ bay được, nhưng chim cánh cụt không bay được. Do chim cánh cụt cũng là chim, ta cho nó kế thừa class "Bird". Tuy nhiên, vì cánh cụt không biết bay, khi gọi hàm bay của chim cánh cụt, ta sẽ quăng "NoFlyException".

class Bird

{

public function fly()

{

return "Fly";

}

}

class EagleBird extends Bird

{

public function fly()

{

return "Eagle Fly";

}

}

class SparrowBird extends Bird

{

public function fly()

{

return "Sparrow Fly";

}

}

class PenguinBird extends Bird

{

public function fly()

{

throw new NoFlyException();

}

}

$birds = [new Bird(), new EagleBird(), new SparrowBird(), new PenguinBird()];

foreach ($birds as $bird) {

$bird->fly();

}

Ta tạo 1 mảng chứa các loài chim rồi duyệt các phần tử. Khi gọi hàm fly() của class "PenguinBird", hàm này sẽ quăng lỗi. Class "PenguinBird" gây lỗi khi chạy, không thay thế được class cha của nó là "Bird", do đó nó đã vi phạm LSP.

### I: Interface segregation principle (nguyên lý giao diện phân biệt - hay phân tách interface).

Thay vì dùng 1 interface lớn, ta nên tách thành nhiều interface nhỏ, với nhiều mục đích cụ thể

Interface là một lớp rỗng chỉ chứa khai báo về tên phương thức không có khai báo về thuộc tính hay thứ gì khác và các phương thức này cũng là rỗng. Bởi vậy bất kỳ lớp nào sử dụng lớp interface đều phải định nghĩa các phương thức đã khai báo ở lớp interface.

Để thiết kế một hệ thống linh hoạt, dễ thay đổi, các module của hệ thống nên giao tiếp với nhau thông qua interface. Mỗi module sẽ gọi chức năng của module khác thông qua interface mà không cần quan tâm tới implementation bên dưới. Như đã nói ở trên, do interface chỉ chứa khai báo rỗng về method, khi một class implement một interface, class đó phải implement toàn bộ các method được khai báo trong interface đó.

Đây là nguyên lý dễ hiểu nhất trong SOLID, các bạn chỉ đọc code một chút là hiểu ngay! Giả sử ta muốn viết một chương trình giới thiệu thuộc tính của các loài động vật. Động vật nào cũng có thể ăn, uống, ngủ, ta thiết kế interface "AnimalInterface" như sau:

interface AnimalInterface

{

public function eat();

public function drink();

public function sleep();

}

class Dog implements AnimalInterface {

public function eat () {}

public function drink () {}

public function sleep () {}

}

class Cat implements AnimalInterface {

public function eat () {}

public function drink () {}

public function sleep () {}

}

Khi ta muốn thêm 1 số loài động vật mới và tính năng vào, ta phải thêm có thêm method vào trong interface như: bơi lội, bay, săn mồi, … Điều này làm interface phình to ra. Khi một loài động vật kế thừa interface, nó phải implement luôn cả những hàm không dùng đến. Khi thêm method vào interface "AnimalInterface", những class cũ như Dog, Cat đều phải implement những method mới nên mất thời gian. Giải pháp trong tình huống này là tách interface "AnimalInterface" ra thành các interface nhỏ như sau:

interface AnimalInterface

{

public function eat();

public function drink();

public function sleep();

}

interface BirdInterface

{

public function fly();

}

interface FishInterface

{

public function swim();

}

// Các class chỉ cần kế thừa những interface có chúc năng chúng cần

class Dog implements AnimalInterface

{

public function eat() {}

public function drink() {}

public function sleep() {}

}

class Bird implements AnimalInterface, BirdInterface

{

public function eat() {}

public function drink() {}

public function sleep() {}

public function fly() {}

}

class Fish implements AnimalInterface, FishInterface

{

public function eat() {}

public function drink() {}

public function sleep() {}

public function swim() {}

}

Để có thể phân tách một interface lớn thành các interface nhỏ một cách hợp lý, các bạn nên áp dụng nguyên tắc đầu tiên trong nguyên tắc SOLID trong bài viết này. Tuy nhiên, đôi khi việc tách ra nhiều interface có thể làm tăng số lượng interface, tăng số lượng class, ta cần cân nhắc lợi hại trước khi áp dụng nhé.

### D: Dependency inversion principle (nguyên lý nghịch đảo phụ thuộc).

* Các module cấp cao không nên phụ thuộc vào các modules cấp thấp. Cả 2 nên phụ thuộc vào abstraction.
* Interface (abstraction) không nên phụ thuộc vào chi tiết, mà ngược lại. Các class giao tiếp với nhau thông qua interface, không phải thông qua implementation.

Nguyên tắc này nói cho bạn rằng bạn không nên viết code gắn chặt với nhau bởi vì sẽ là cơn ác mộng cho việc bảo trì khi ứng dụng trở lên lớn dần. Nếu một class phụ thuộc một class khác, bạn sẽ cần phải thay đổi class đó nếu một trong những class phụ thuộc phải thay đổi. Chúng ta nên cố gắng viết các class ít phụ thuộc nhất có thể.

Chúng ta đều biết 2 loại đèn: đèn tròn và đèn huỳnh quang. Chúng cùng có đuôi tròn, do đó ta có thể thay thế đèn tròn bằng đèn huỳnh quanh cho nhau 1 cách dễ dàng. Ở đây, interface chính là đuôi tròn, implementation là bóng đèn tròn và bóng đèn huỳnh quang. Ta có thể swap dễ dàng giữa 2 loại bóng vì ổ điện chỉ quan tâm tới interface (đuôi tròn), không quan tâm tới implementation.

Giả sử chúng ta có một hệ thống thông báo sau khi lưu vài thông tin vào DB.

class Email

{

public function sendEmail()

{

// code to send mail

}

}

class Notification

{

private $email;

public function notification()

{

$this->email = new Email();

}

public function promotionalNotification()

{

$this->email->sendEmail();

}

}

Giờ class Notification hoàn toàn phụ thuộc vào class Email, vì nó chỉ gửi một loại của thông báo. Nếu bạn muốn thêm một cách thông báo mới như SMS chẳng hạn? Chúng ta cũng phải thay đổi cả hệ thống thông báo? Đó gọi là liên kết chặt (tightly coupled). Bạn có thể làm gì để giúp nó giảm phụ thuộc vào nhau. OK, bạn xem ví dụ sau đây:

interface MessengerInterface

{

public function sendMessage();

}

class Email implements MessengerInterface

{

public function sendMessage()

{

// code to send email

}

}

class SMS implements MessengerInterface

{

public function sendMessage()

{

// code to send SMS

}

}

class Notification

{

private $messenger;

public function notification(MessengerInterface $messenger)

{

$this->messenger = $messenger;

}

public function doNotify()

{

$this->messenger->sendMessage();

}

}

Như vậy lớp Notification không còn phụ thuộc vào các lớp cụ thể (SMS hay Email) mà chỉ phụ thuộc vào 1 lớp trừu tượng MessengerInterface, ta có thể thêm các lớp cụ thể (SMS hay Email) mà không hề ảnh hưởng đến lớp Notification.