source:https://gist.github.com/mhfireapps/8d403558ca88154a7b1d947467c69baa#m%E1%BB%A5c-l%E1%BB%A5c

---------------------------------BIẾN--------------------------------

1.Biến

\* Chữ cái đầu viết thường, từ chữ thứ 2 viết hoa

---Sử dụng tên biến có ý nghĩa và dễ hiểu

-> đặt tên liên quan đến ý nghĩa của biến

---Sử dụng cùng từ vựng cho cùng một loại biến

->getUserInfo();getUserData(); =>>>> getUser();

---Đặt tên sao cho dễ tìm kiếm (phần 1)

-> không truyền số trực tiếp mà phải gán cái số đó vào một biến có tên cụ thể

-> Số nào mà không thay đổi giá trị thì nên gán vào biến CONST

---Đặt tên biến có dễ hiểu

->Tức là đặt tên biến sao cho đọc vô là hiểu nó là gì và nó dùng để làm gì. Không cần phải suy nghĩ, suy diễn.

-> không nên $matches[1]; ==>> hãy sử dụng $matches["name"];

---Tránh lồng (nesting) quá nhiều và nên return sớm:

->Quá nhiều if else lồng nhau sẽ khiến code tăng độ phức tạp, khó debug.

->Giảm sự phức tạp bằng cách giảm số if else lồng nhau xuống ít nhất có thể.

->Return sớm chính là một cách giảm số lần lồng nhau.

-->in\_array(strtolower($day), $openingDays, true); // xem lại

---Tránh hack não người đọc

->Đừng khiến người đọc code phải khó khăn để hiểu ý nghĩa của biến.

->Tên biến càng rõ ràng càng tốt.

---Đừng thêm những nội dung không cần thiết

->Nếu tên của class/object đã rõ ràng, không nên lặp lại chúng trong tên biến.

-> $carMake ==>> $make;

---Sử dụng đối số mặc định thay vì phải kiểm tra bằng biểu thức điều kiện

->function createMicrobrewery($name = null): void // xem lại void

{

$breweryName = $name ?: 'Hipster Brew Co.';

// ...

}

//sử dụng type hinting và chắc chắn $breweryName sẽ không bị NULL.

->function createMicrobrewery(string $breweryName = 'Hipster Brew Co.'): void

{

// ...

}

----------------------------------SO SÁNH--------------------------------

2.So sánh

---Sử dụng identical comparison // xem lại chưa rõ lắm

----------------------------------HÀM--------------------------------

3.Hàm

------ public function getCruisingAltitude(): int // trả về kiểu

------ public function getCruisingAltitude(): void // không có giá trị trả về

---Đối số của hàm (ít hơn hoặc bằng 2 là lý tưởng) // xem lại ví dụ để học hỏi

->Giới hạn số lượng đối số (parameters) của hàm vô cùng quan trọng bởi vì nó giúp dễ test hơn.

->Có nhiều hơn 3 đối số dẫn đến một tổ hợp rất nhiều trường hợp khác nhau cần phải test.

->Lý tưởng nhất là khi hàm không có đối số nào.

->Một hoặc hai đối số là ok, còn ba thì nên hạn chế.

->Bất cứ khi nào hàm có nhiều hơn 3 đối số thì cần phải xem xét tìm cách giảm bớt lại.

->Bởi vì nếu hàm có nhiều hơn hai đối số thì nó phải xử lý rất nhiều.

---Hàm chỉ thực hiện một chức năng

->Đây là nguyên tắc quan trọng nhất trong phát triển phần mềm.

->Khi hàm thực hiện nhiều hơn một chức năng, chúng khó biên dịch, kiểm tra và biết được nguyên nhân lỗi.

->Khi bạn tạo hàm chỉ với một chức năng, sẽ dễ dàng refactor hơn và code sẽ dễ đọc hơn.

->Nếu làm được điều này thì bạn sẽ tốt hơn nhiều lập trình viên khác.

->array\_walk($array,"call\_back\_function"); array\_filter('tên mảng','tên hàm gọi call back') // xem lại ví dụ hay vãi loen luôn

---Tên hàm nên thể hiện chức năng của hàm

->Hàm này dùng làm gì? Có phải nó xử lý.. không?

---Hàm chỉ nên chứa một cấp trừu tượng

->Khi bạn có độ trừu tượng nhiều hơn một cấp thì hàm thường phải làm quá nhiều việc.

->Hãy chia tách hàm ra thành nhiều phần để dễ sử dụng lại và dễ test. // xem lại ví dụ

-> Hàm explode() //chuyển một chuỗi thành một mảng và mỗi phần tử được cắt bởi một chuỗi con nào đó.

-> tạo contructor

-> public function \_\_construct(Tokenizer $tokenizer, Lexer $lexer)

{

$this->tokenizer = $tokenizer;

$this->lexer = $lexer;

}

---Đừng sử dụng cờ như là một đối số của hàm

->Cờ dùng để nói rằng hàm này thực hiện nhiều hơn một công việc.

->Nhưng hàm thì chỉ nên xử lý một công việc mà thôi.

->Do đó hãy chia tách hàm của bạn nếu như chúng có nhiều luồng code phân biệt bằng boolean(true/false).

---Tránh tác dụng phụ // xem ví dụ khá hay

->Hãy tránh những sai lầm phổ biến như:

- chia sẻ trạng thái giữa các object mà không tuân theo cấu trúc nào,

-sử dụng kiểu dữ liệu có thể thay đổi/bị thay đổi dễ dàng

--> không tổng hợp các tác dụng phụ có thể xảy ra khi viết hàm.

---Đừng viết hàm global // xem ví dụ

---Đừng sử dụng Singleton pattern // xem ví dụ(hiểu sâu hơn)

---Đóng gói điều kiện

-> viết điều kiện trong môt hàm rồi gọi hàm đó ra

---Tránh điều kiện phủ định

-> nên trả về true/false,không nên (!)

---Tránh dùng điều kiện

->Điều này có vẻ không khả quan. Hầu hết mọi người sẽ thắc mắc, "làm sao có thể làm gì đó mà không có if?"

->Bạn có thể dùng tính đa hình để hoàn thành việc đó trong khá nhiều trường hợp.

->Câu hỏi thứ hai là, "ồ ngon nhưng tại sao phải làm thế?"

->Bởi vì khái niệm clean code mà ta đã học trước đây: một hàm chỉ nên thực hiện một chức năng.

->Khi bạn có một lớp hoặc hàm chứa if, tức là bạn đang muốn nó thực hiện nhiều việc.

->Luôn nhớ, chỉ một mà thôi.

---Tránh kiểm tra kiểu dữ liệu

-> xem ví dụ

->function combine(int $val1, int $val2): int // nên định kiểu cho tham số truyền vào

---Xóa dead code

->Dead code thì cũng củ chuối giống như duplicate code.

->Không có lý do gì để giữ chúng.

->Nếu đoạn code nào đó không được gọi, hãy xóa đi!

->Sau này cần thì chỉ cần tìm lại phiên bản trước bằng git là được.

----------------------------OBJECT------------------------------------

5.Đối tượng và kiến trúc dữ liệu

---Sử dụng đối tượng đóng gói

->Trong PHP bạn có thể khai public, protected và private cho phương thức và thuộc tính.

->Hãy sử dụng chúng để kiểm soát được sự thay đổi thuộc tính trong object.

->Khi bạn muốn nhiều hơn là chỉ nhận được thuộc tính của object, thì ưu điểm là ta không cần phải tìm kiếm và thay đổi quyền mỗi khi truy cập vào object.

->Giúp tạo validation đơn giản mỗi khi thực hiện set.

->Đóng gói các thành phần bên trong.

->Dễ dàng ghi log và xử lý lỗi khi get và set.

->Kế thừa lớp, bạn có thể ghi đè những phương thức mặc định.

->Bạn có thể lazy load các thuộc tính của object, giả sử nó được lấy từ máy chủ chẳng hạn.

->Thêm vào đó, đây là một phần của nguyên tắc Open/Closed.

-> ví dụ khá hay

---Tạo đối tượng có chứa thuộc tính hoặc phương thức private/protected

->Phương thức và thuộc tính public khá nguy hiểm

->bởi vì vài dòng code phía bên ngoài có thể dễ dàng thay đổi chúng và bạn không thể kiểm soát được nó bị thay đổi những gì.

->Thay đổi trong một lớp thì nguy hiểm cho tất cả các người dùng của lớp đó.

->protected cũng nguy hiểm không kém, bởi vì chúng được cấp quyền ở tất cả các lớp con.

->Điều này có nghĩa là sự khác nhau giữa public và protected chỉ là cơ chế truy cập

->Nhưng tính đóng gói đảm bảo vẫn giữ nguyên.

->Sửa đổi trong lớp thì rất nguy hiểm cho các lớp con.

->private sửa đổi đảm bảo rằng code sửa đổi chỉ nguy hiểm trong lớp đó (bạn sẽ được an toàn khi sửa và không có hiệu ứng Jenga).

->Do đó, hãy mặc định sử dụng private và public/protected khi bạn cần cung cấp sự truy cập cho các class bên ngoài.

----------------------------CLASS------------------------------------

6.Lớp

Ưu tiên thành phần hơn kế thừa

-> ở lớp con thì parent::\_\_construct($name, $email);

->xem ví dụ

Tránh viết fluent interfaces

--> chưa hiểu ví dụ cho lắm

SOLID

Nguyên lý trách nhiệm duy nhất (SRP)

Nguyên lý Đóng/Mở (OCP)

Nguyên lý thay thế Liskov (LSP)

Nguyên lý phân tách interface (ISP)

Nguyên lý đảo ngược dependencies (DIP)

Nguyên lý đừng lặp lại chính mình (DRY)

Các ngôn ngữ khác