SOLID – 5 nguyên tắc của thiết kế hướng đối tượng

August 27, 2016

1. Single responsibility principle

Tạm dịch: Nguyên tắc đơn nhiệm

Nguyên tắc đơn nhiệm phát biểu rằng: mỗi class hay method chỉ nên có một và chỉ một công năng (a.ka. chịu trách nhiệm cho một việc). Thường thì nguyên tắc này rất dễ bị vi phạm, kiểu như một class User chịu trách nhiệm cho rất nhiều việc trong ứng dụng, từ validate email cho tới đăng nhập, gửi email v.v. Những class làm quá nhiều việc khác nhau như vậy thường được gọi là **god object**.

Nguyên tắc này được lần đầu giới thiệu bởi Robert C. Martin (Principles of Object Oriented Design), ông định nghĩa rằng "responsibility" là **một lý do để thay đổi**

IN THE CONTEXT OF THE SINGLE RESPONSIBILITY PRINCIPLE (SRP) WE DEFINE A RESPONSIBILITY TO BE "A REASON FOR CHANGE." IF YOU CAN THINK OF MORE THAN ONE MOTIVE FOR CHANGING A CLASS, THEN THAT CLASS HAS MORE THAN ONE RESPONSIBILITY. – BOB MARTIN

Đổi với method, tương đối dễ để áp dụng nguyên tắc này. Ví dụ trong lập trình game hàm kiểm tra va chạm (tạm gọi `detectCollision()`) không nên thay đổi điểm của người chơi, nếu

hàm `detectCollision()` làm một việc khác ngoài việc "detect collision" thì nó đã vi phạm nguyên tắc đơn nhiệm.

2. Open/closed principle

Tạm dịch: Nguyên tắc đóng/mở

> *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*

Nguyên tắc này có thể được hiểu rằng: các class hoặc hàm nên được thiết kế để mở rộng, nhưng đóng lại để tránh sự thay đổi trực tiếp mã nguồn. Điều này có nghĩa là hệ thống nên được thiết để hỗ trợ các lập trình viên sau này có extend các class có sẵn để cung cấp thêm chức năng thay vì chỉnh sửa trên mã nguồn tồn tại sẵn trong hệ thống.

Việc này có lợi là những lập trình viên sau này cùng làm việc trên một code base không cần phải viết test lại cho code có sẵn và hạn chế tối đa *side effects*. Bởi vì đối với những code base lớn thì một thay đổi nhỏ cũng có thể gây ra hậu quả lớn, kiểu hiệu ứng domino.

Để minh hoạ cho ví dụ này, hãy xem một ví dụ về **strategy design pattern** sau:

```
<?php
interface GreetingStrategyInterface
{
    function greet($name);
}

class EnglishGreetingStrategy implements GreetingStrategyInterface
{
    function greet($name)
    {
        printf("Hello, %s", $name);
```

```
    }
}

class VietnameseGreetingStrategy implements GreetingStrategyInterface
{
    function greet($name)
    {
        printf("Xin chào, %s", $name);
    }
}

class GreetingContext
{
    private $greetingStategy = null;

    public function __construct($context)
    {
        switch($context)
        {
            case "Vietnamese":
                $this->greetingStategy = new VietnameseGreetingStrategy();
                break;
            case "English":
                $this->greetingStategy = new EnglishGreetingStrategy();
                break;
            default:
                $this->greetingStategy = new EnglishGreetingStrategy();
                return;
        }
    }

    public function greet($name)
```

```
    {
        $this->greetingStategy->greet($name);
    }
}

$vnGreeter  = new GreetingContext("Vietnamese");
$enGreeter  = new GreetingContext("English");

$vnGreeter->greet("thế giới!");
$enGreeter->greet("World!");
//Xin chào, thế giới!
//Hello, World!
```

3. Liskov substitution principle

Tạm dịch: Nguyên tắc hoán đổi Liskov

> *The Liskov substitution principle (LSP) states that objects should be replacable with instances of their subtypes without alterting the correctness of that program*

Nguyên tắc này nghe có vẻ phức tạp, nhưng thực chất có thể diễn ý đơn giản lại như sau: Nếu một class có sử dụng một implemtation của một interface, thì nó phải được thay thế dễ dàng bởi các implementation của interface đó mà không cần sửa gì thêm.

Ví dụ này được lấy lại từ bài viết trước. Chúng ta có thể thấy rằng `StandardLogger` và `FileLogger` cùng implement một interface và chúng có thể thay đổi dễ dàng với nhau mà không cần chỉnh sửa mã nguồn của `MyLog` .

```
<?php
interface LoggerInterface
{
```

```
    function info($message);
}

class StandardLogger implements LoggerInterface
{

    public function info($message)
    {
        printf("[INFO] %s \n", $message);
    }
}

class FileLogger implements LoggerInterface
{

    public function info($message)
    {
        file_put_contents('app.log', sprintf("[INFO] %s \n", $message), FILE_APPEND);
    }
}

class MyLog
{
    public $logger;

    public function __construct(LoggerInterface $logger)
    {
        $this->logger = $logger;
    }

    public function info($string)
    {
```

```
        return $this->logger->info($string);
    }
}
// Print to standard input/output device
$myLog = new MyLog(new StandardLogger);
$myLog->info('This object depend on another object');
// Write to file
$myFileLog = new MyLog(new FileLogger);
$myFileLog->info('This object depend on another object');
```

4. Interface segregation principle

Tạm dịch: Nguyên tắc tách rời giao diện (lập trình)

> *The interface-segregation principle (ISP) states that no client should be forced to depend on methods it does not use.*

Nguyên tắc này phát biểu rằng implementation của một interface không nên bị phụ thuộc vào những methods mà nó không dùng. Điều này có nghĩa là các interface phải được sắp xếp và phân chia hợp lý. Thay vì có một **FAT** interface chứa tất cả các methods cần được thi công thì nó nên được chia nhỏ ra mà class nào implement nó cũng không có method **thừa**.

Trong cuốn **Laravel: From Apprentice To Artisan**, Taylor Otwell có đưa ví dụ về `SessionHandlerInterface` như sau:

```
<?php
interface SessionHandlerInterface
{
    public function close();
    public function destroy($sessionId);
    public function gc($maxLifeTime);
```

```
    public function open($savePath, $name);
    public function read($sessionId);
    public function write($sessionId, $sessionData);
}
```

Nhìn interface SessionHandlerInterface ở trên có vẻ không có vấn đề gì, nhưng thực ra nó ép conrete class sử dụng khá nhiều method thừa ví dụ như `open`, `close`, `gc`. Ví dụ chúng ta sử dụng memcached để lưu session thì nó tự động expire dữ liệu mà chúng ta không cần implement hàm `gc` (garbage collector). Để giải quyết vấn đề này, `SessionHandlerInterface` nên được chia nhỏ ra thành nhiều interface nhỏ hơn và tập trung vào chức năng không thể tách rời. Ví dụ:

```
<?php
intferface GarbageCollectorInterface
{
    public function gc($maxLifeTime);
}
```

5. Dependency inversion principle

Tạm dịch: Nguyên tắc nghịch đảo phụ thuộc

THE DEPENDENCY INVERSION PRINCIPLE (DIP) STATES THAT HIGH-LEVEL CODE SHOULD NOT DEPEND ON LOW-LEVEL CODE, AND THAT ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS

Để dễ hiểu thì code càng gần với ngôn ngữ máy hoặc làm những việc càng cơ bản thì gọi là low-level code (ví dụ: truy vấn trực tiếp database, đọc ghi xuống file, network v.v). Còn code nào xử lý logic và sử dụng các thư viện low-level thì được gọi là...high-level code.

Nguyên tắc này nói rằng, high-level code không nên phụ thuộc vào low-level code. Thay vào đó high-level nên phụ thuộc vào một abstraction mà hỗ trợ tương tác với low-level code đó, nhưng không được phụ thuộc chi tiết của low-level code đó.

Nghe thì có vẻ lằn nhằn, nhưng chúng ta có thể hiểu nguyên tắc này dễ dàng hơn qua ví dụ vi phạm DIP sau:

Giả sử chúng ta có một class `Authenticator`, class này tương tác trực tiếp với MySQL, và sử dụng hàm `md5` để mã hoá password. Dễ thấy class đó vi phạm cả 2 nguyên tắc trên của DIP. Vì theo nguyên tắc trên, `Authenticator` là high-level code, nó không nên phụ thuộc vào code tương tác với MySQL (có thể là Eloquent hay raw SQL) và chi tiết cụ thể là sử dụng hàm `md5` để hash password.

```
<?php
class Authenticator
{
    public function __construct($DatabaseConnection $db)
    {
        $this->db = $db;
    }

    public function findUser($id)
    {
        return $this->db->exec("select * from users where id = ?", array($id));
    }

    public function authenticate($credentials)
    {
        //Authenticate the users
    }
}
```

Thay vào đó cả hai nên được inject vào `Authenticator`. Điều này có lợi là chúng ta có thể dễ dàng thay thế `MySQL` bằng `NoSQL` (hay làm những việc khác như connect với Facebook, Google), và thay thế `md5` bằng một hàm băm hay mã hoá khác an toàn hơn.

```php
<?php
class Authenticator
{
    public function __construct(UserProviderInterface $users,
                                HasherInterface $hash)
    {
        $this->users = $users;
        $this->hash = $hash;
    }
}
```

Để hiểu rõ và đi vào chi tiết hơn về Dependency Inversion/Injection. Các bạn có thể tham khảo bài viết trước Laravel : Dependency Injection và IoC container

Kết luận

Có thể nói, Nguyên tắc SOLID giúp chúng ta xây dựng những hệ thống lớn, dễ mở rộng và bảo trì hơn. Nhưng mà có thể thấy, chúng ta phải trừu tượng hoá ứng dụng một chút, sử dụng nhiều interface hơn, gõ phím nhiều hơn một chút. Một số người cho rằng: **Too much Java**. Cái này đúng nếu như code base của chúng ta nhỏ và không bao giờ thay đổi. Nhưng thực tế, chẳng ai mong muốn ứng dụng/ business mình không phát triển cả. Một lần nữa có lẽ nó là **nice problem to have**. Các nguyên tắc này không mới, nhưng mình tin rằng nó mang lại khá nhiều lý luận cũng như công cụ để xây dựng các dụng lớn và dễ bảo trì nâng cấp cho mọi lập trình viên (trừ mấy tay chơi pure functional programming ra :grin:).

Techtalk via butchiso