Viết mã có thể duy trì: Các nguyên tắc SOLID được giải thích trong PHP (Laravel)

Bởi Ankush Thakur
vào ngày 16 tháng 11 năm 2020

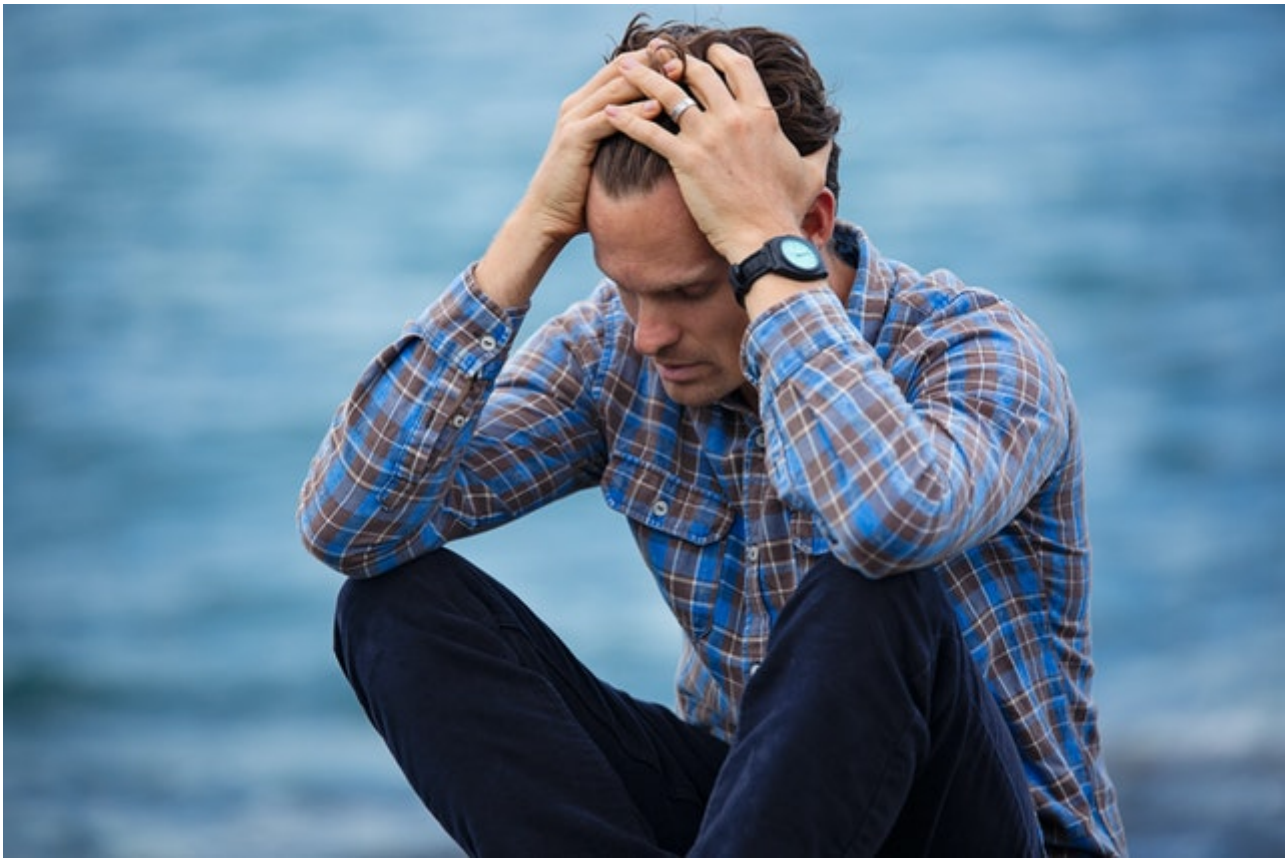
> Viết chương trình máy tính là một niềm vui lớn. Trừ khi bạn phải làm việc với mã của người khác.

Nếu bạn đã làm việc với tư cách là một nhà phát triển chuyên nghiệp trong hơn ba ngày, bạn biết rằng công

việc của chúng tôi là bất cứ điều gì ngoài sáng tạo và thú vị. Một phần lý do là do ban lãnh đạo công ty (đọc: những người không bao giờ hiểu được), trong khi phần khác là sự phức tạp của mã mà chúng tôi phải làm việc. Bây giờ, trong khi chúng ta hoàn toàn không thể làm gì về cái trước, chúng ta có thể làm được nhiều điều về cái sau.

Vì vậy, tại sao các cơ sở mã phức tạp đến mức chúng ta cảm thấy như rút ruột chính mình? Đơn giản là vì những người viết phiên bản đầu tiên đã vội vàng, và những người đến sau cứ thêm vào mớ hỗn độn. Kết quả cuối cùng: một mớ hỗn độn mà rất ít người muốn chạm vào và không ai hiểu.

Chào mừng đến với ngày đầu tiên của công việc!



“Không ai nói rằng nó sẽ khó như vậy. . . ”

Nhưng nó không nhất thiết phải theo cách này.

Viết mã tốt , mã mô-đun và dễ bảo trì, không khó lắm. Chỉ cần năm nguyên tắc đơn giản - đã được thiết lập từ lâu và nổi tiếng - nếu tuân theo kỷ luật, sẽ đảm bảo mã của bạn có thể đọc được, cho người khác và cho bạn khi bạn nhìn vào nó sáu tháng sau. 😂

Các nguyên tắc hướng dẫn này được thể hiện bằng từ viết tắt **SOLID** . Có lẽ bạn đã từng nghe đến thuật ngữ “Nguyên tắc RẮN” trước đây, có lẽ chưa. Trong trường hợp bạn đã học nhưng vẫn tạm dừng việc học

này cho “một ngày nào đó”, thì hãy đảm bảo rằng hôm nay là ngày đó!

Vì vậy, không cần phải lo lắng gì thêm, hãy xem xét nội dung của SOLID này là gì và nó có thể giúp chúng ta viết một đoạn mã chặt chẽ thực sự gọn gàng như thế nào.

“S” là cho Trách nhiệm đơn lẻ

Nếu bạn xem các nguồn khác nhau mô tả Nguyên tắc trách nhiệm duy nhất, bạn sẽ nhận được một số thay đổi trong định nghĩa của nó. Tuy nhiên, nói một cách đơn giản hơn, nó tóm gọn lại điều này: Mỗi lớp trong cơ sở mã của bạn phải có một vai trò rất cụ thể; nghĩa là, nó không nên *chịu trách nhiệm* cho một mục tiêu *duy nhất* . Và bất cứ khi nào cần thay đổi một trong lớp đó, thì chúng ta sẽ chỉ cần thay đổi nó vì một trách nhiệm cụ thể đã thay đổi.

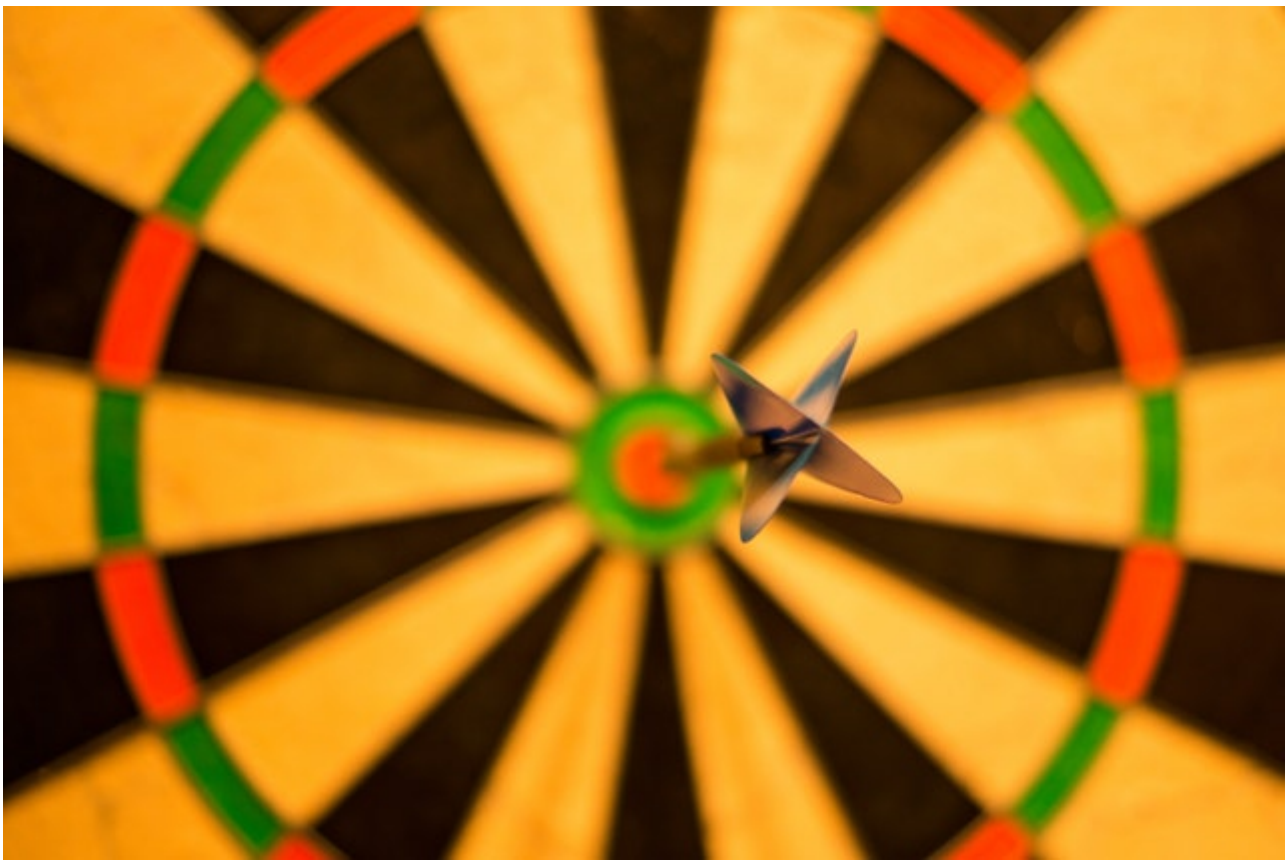
Khi tôi bắt gặp điều này lần đầu tiên, định nghĩa mà tôi đã được trình bày là, “Nên có một và chỉ một lý do để một lớp thay đổi”. Tôi đã giống như, "Cái gì ??! Thay đổi? Có gì thay đổi? Tại sao lại thay đổi? ”, Đó là lý do tại sao tôi đã nói trước đó rằng nếu bạn

đọc về điều này ở những nơi khác nhau, bạn sẽ nhận được những định nghĩa có liên quan nhưng hơi khác nhau và có khả năng gây nhầm lẫn.

Dù sao, đủ của nó. Đã đến lúc cho một điều gì đó nghiêm túc: nếu bạn giống tôi, có lẽ bạn đang tự hỏi, “Được rồi, tất cả đều tốt. Nhưng tại sao tôi phải quan tâm đến cái quái gì? Tôi sẽ không bắt đầu viết mã theo một phong cách hoàn toàn khác từ ngày mai chỉ vì một kẻ điên từng viết sách (và hiện đã chết) nói như vậy ”.

Thông minh!

Và đó là tinh thần chúng ta cần duy trì nếu chúng ta muốn *thực sự* học hỏi mọi thứ. Vì vậy, tại sao tất cả bài hát và điệu nhảy này về "Trách nhiệm duy nhất" lại quan trọng? Những người khác nhau giải thích điều này theo cách khác nhau, nhưng đối với tôi, nguyên tắc này là tất cả nhằm mang lại kỷ luật và sự tập trung vào mã của bạn.



Tập trung, chàng trai của tôi. Tiêu điểm!

Hãy xem một ví dụ trước khi tôi giải thích cách diễn giải của mình. Không giống như các tài nguyên khác được tìm thấy trên web cung cấp các ví dụ mà bạn hiểu nhưng sau đó khiến bạn tự hỏi làm thế nào họ sẽ giúp bạn trong các trường hợp thực tế, hãy đi sâu vào một cái gì đó cụ thể, một phong cách mã hóa mà chúng ta thấy đi xem lại và thậm chí có thể viết trong các ứng dụng Laravel của chúng tôi.

Khi một ứng dụng Laravel nhận được yêu cầu web, URL sẽ được khớp với các tuyến bạn đã xác định `web.php` và `api.php` và nếu có sự trùng khớp , dữ liệu yêu cầu sẽ đến bộ điều khiển. Đây là

phương pháp bộ điều khiển điển hình trông như thế nào trong các ứng dụng thực tế, cấp sản xuất:

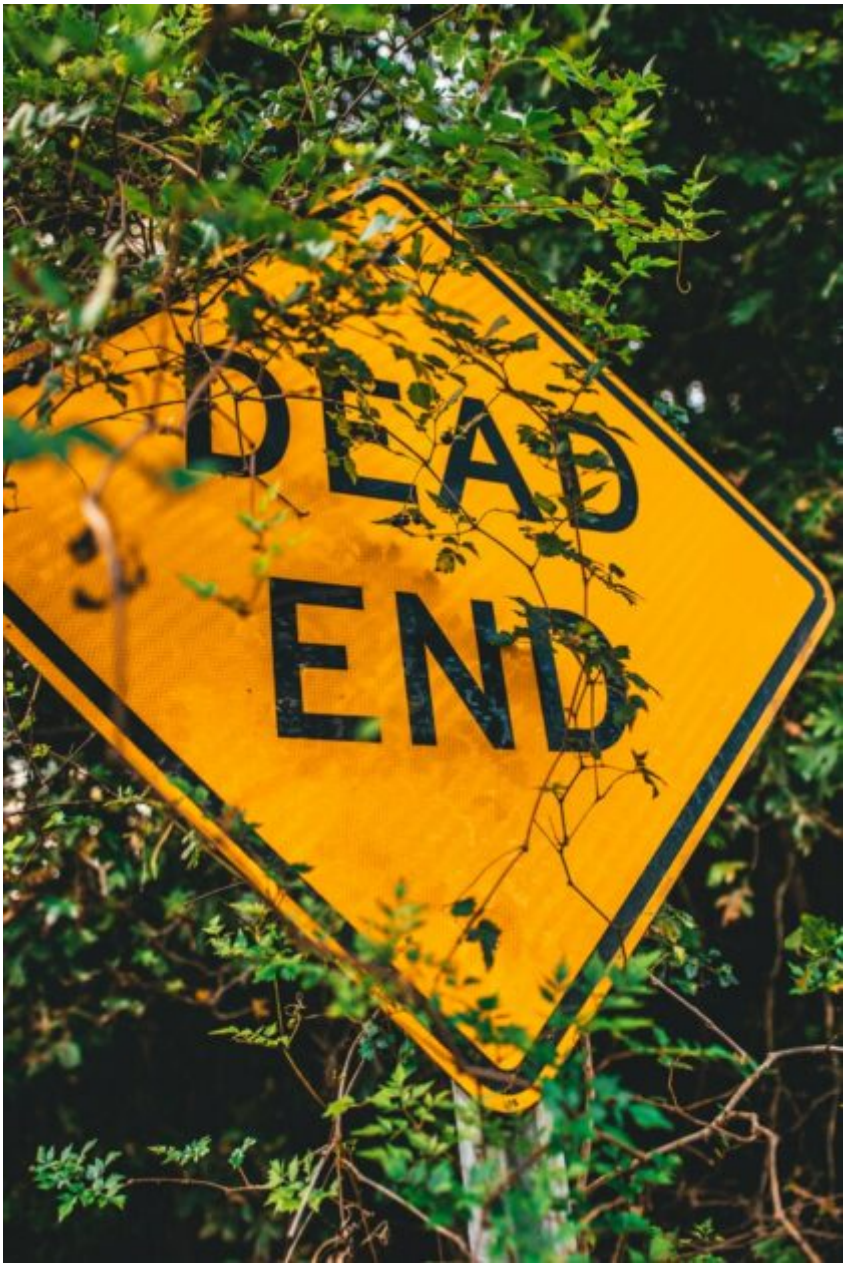
```
class UserController extends Controller {
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
           'first_name' => 'required',
           'last_name' => 'required',
           'email' => 'required|email|unique:users',
           'phone' => 'nullable'
       ]);

       if ($validator->fails()) {
            Session::flash('error', $validator->messages()->first());
            return redirect()->back()->withInput();
       }

       // create new user
       $user = User::create([
           'first_name' => $request->first_name,
           'last_name' => $request->last_name,
           'email' => $request->email,
           'phone' => $request->phone,
       ]);

       return redirect()->route('login');
    }
}
```

Tất cả chúng tôi đã viết mã như thế này. Và thật dễ dàng để xem nó làm gì: đăng ký người dùng mới. Nó trông ổn và hoạt động tốt, nhưng có một vấn đề - nó không thể chứng minh được trong tương lai. Và bằng

chứng trong tương lai, ý tôi là nó chưa sẵn sàng để xử lý sự thay đổi mà không tạo ra một mớ hỗn độn.

Tại sao như vậy?

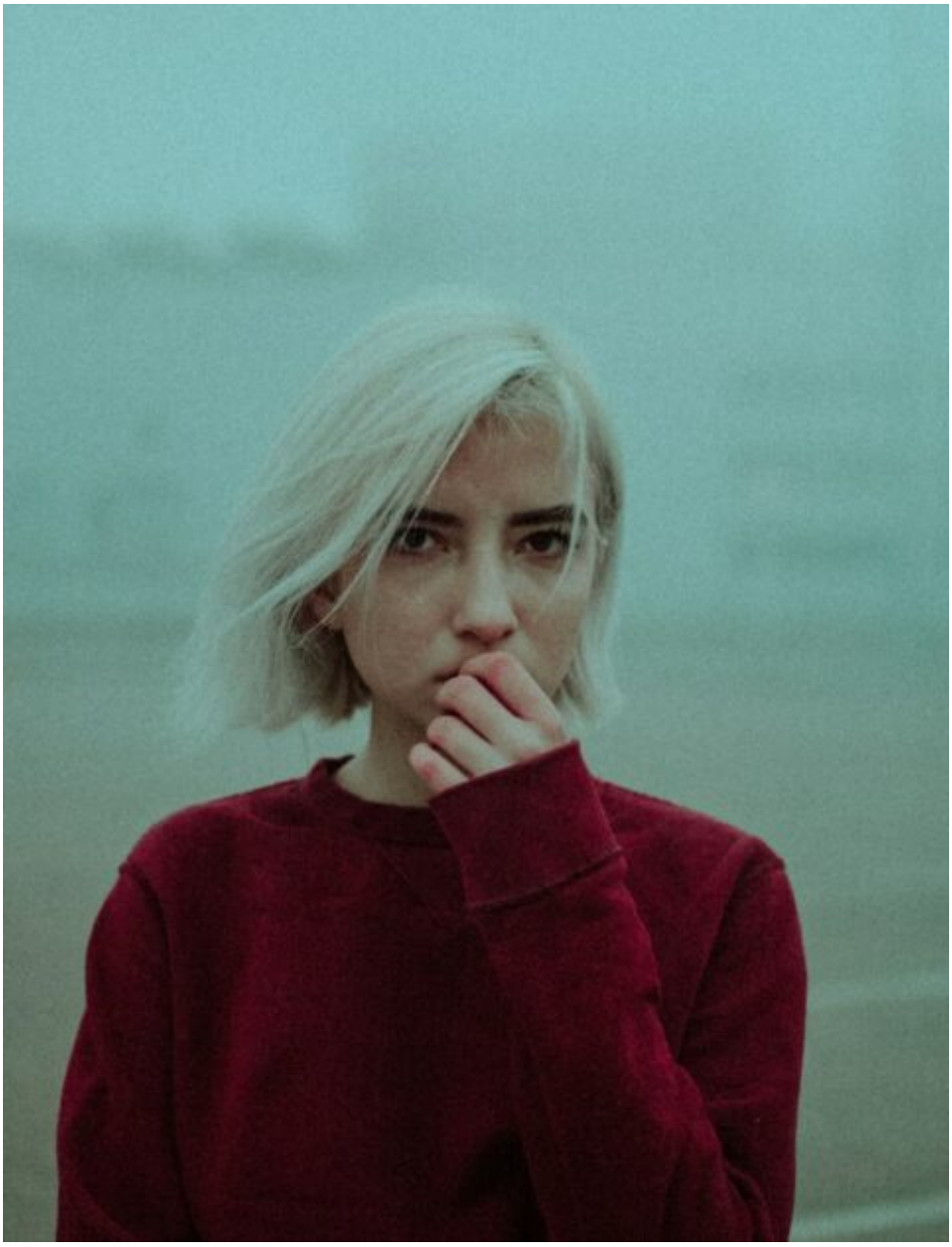
Bạn có thể nói rằng hàm này dành cho các tuyến đường được xác định trong `web.php` tệp; đó là các trang truyền thống, do máy chủ hiển thị. Một vài ngày trôi qua và bây giờ khách hàng / chủ lao động của bạn đang phát triển một ứng dụng dành cho thiết bị di động, có nghĩa là lộ trình này sẽ không có ích cho những người dùng đăng ký từ thiết bị di động. Bạn làm nghề gì? Tạo một tuyến đường tương tự trong `api.php` tệp và viết hàm điều khiển hướng JSON cho nó? Tốt thôi, và sau đó thì sao? Sao chép tất cả mã từ chức năng này, thực hiện một vài thay đổi và gọi nó là một ngày? Đây thực sự là điều mà nhiều nhà phát triển đang làm, nhưng họ đang chuẩn bị cho sự thất bại.



Vấn đề là HTML và JSON không phải là định dạng API duy nhất trên thế giới (chúng ta hãy chỉ coi các trang HTML là một API để tranh luận). Điều gì về một khách hàng có hệ thống kế thừa chạy trên định dạng XML? Và sau đó có một cái khác cho SOAP. Và gRPC. Và Chúa biết điều gì khác sẽ đến vào ngày hôm sau.

Bạn vẫn có thể cân nhắc tạo một tệp riêng cho từng loại API này và sao chép mã hiện có, sửa đổi một chút. Chắc chắn có mười tệp, bạn sẽ tranh luận, nhưng tất cả đều hoạt động tốt, vậy tại sao phải phàn nàn? Nhưng sau đó, cú đấm ruột, kẻ thù không đội trời chung của phát triển phần mềm - thay đổi. Giả sử bây giờ, nhu cầu của khách hàng / nhà tuyển dụng của bạn đã thay đổi. Giờ đây, họ muốn rằng, tại thời điểm đăng ký người dùng, chúng tôi ghi lại địa chỉ IP cũng như thêm một tùy chọn cho trường cho biết họ đã đọc và hiểu các điều khoản và điều kiện.

Ồ ồ! Chúng tôi không có mười tệp để chỉnh sửa và chúng tôi phải đảm bảo rằng logic được xử lý hoàn toàn giống nhau trong tất cả chúng. Ngay cả một lỗi nhỏ cũng có thể gây ra tổn thất lớn trong kinh doanh. Và bây giờ hãy tưởng tượng sự kinh hoàng trong các ứng dụng SaaS quy mô lớn, vì độ phức tạp của mã đã khá cao.



Chỉ trích . . .

Làm thế nào chúng ta đến được địa ngục này?

Câu trả lời là phương thức controller trông có vẻ vô hại này thực sự đang thực hiện một số việc khác nhau: nó xác thực yêu cầu đến, xử lý chuyển hướng và tạo người dùng mới.

Nó đang làm quá nhiều thứ! Và vâng, như bạn có thể nhận thấy, biết cách tạo người dùng mới trong hệ thống thực sự không phải là một công việc của các phương thức điều khiển. Nếu chúng ta loại bỏ logic này ra khỏi hàm và đặt nó vào một lớp riêng biệt, thì bây giờ chúng ta sẽ có hai lớp, mỗi lớp có một trách nhiệm duy nhất để xử lý. Trong khi các lớp này có thể giúp đỡ lẫn nhau bằng cách gọi các phương thức của chúng, chúng không được phép biết những gì đang xảy ra bên trong lớp kia.

```
class UserController extends Controller {
    public function store(Request $request)
    {
        $validator = Validator::make($request->all(), [
           'first_name' => 'required',
           'last_name' => 'required',
           'email' => 'required|email|unique:users',
           'phone' => 'nullable'
       ]);

       if ($validator->fails()) {
            Session::flash('error', $validator->messages()->first());
            return redirect()->back()->withInput();
       }

       UserService::createNewUser($request->all());
       return redirect()->route('login');
    }
}
```

Nhìn mã bây giờ: nhỏ gọn, dễ hiểu hơn nhiều. . . và quan trọng nhất là thích ứng với sự thay đổi. Tiếp tục cuộc thảo luận trước đó của chúng ta, nơi chúng ta có mười loại API khác nhau, mỗi loại hiện gọi một hàm duy nhất và được thực hiện với nó. Nếu cần thay đổi trong logic đăng ký người dùng, lớp sẽ nhìn thấy nó trong khi các phương thức bộ điều khiển không cần thay đổi gì cả. Nếu xác nhận SMS cần được thiết lập sau khi người dùng đăng ký, họ sẽ xử lý nó (bằng cách gọi một số lớp khác biết cách gửi SMS), và một lần nữa, các bộ điều khiển vẫn được giữ nguyên.`UserService::createNewUser($request->all());` `UserService` `UserService`

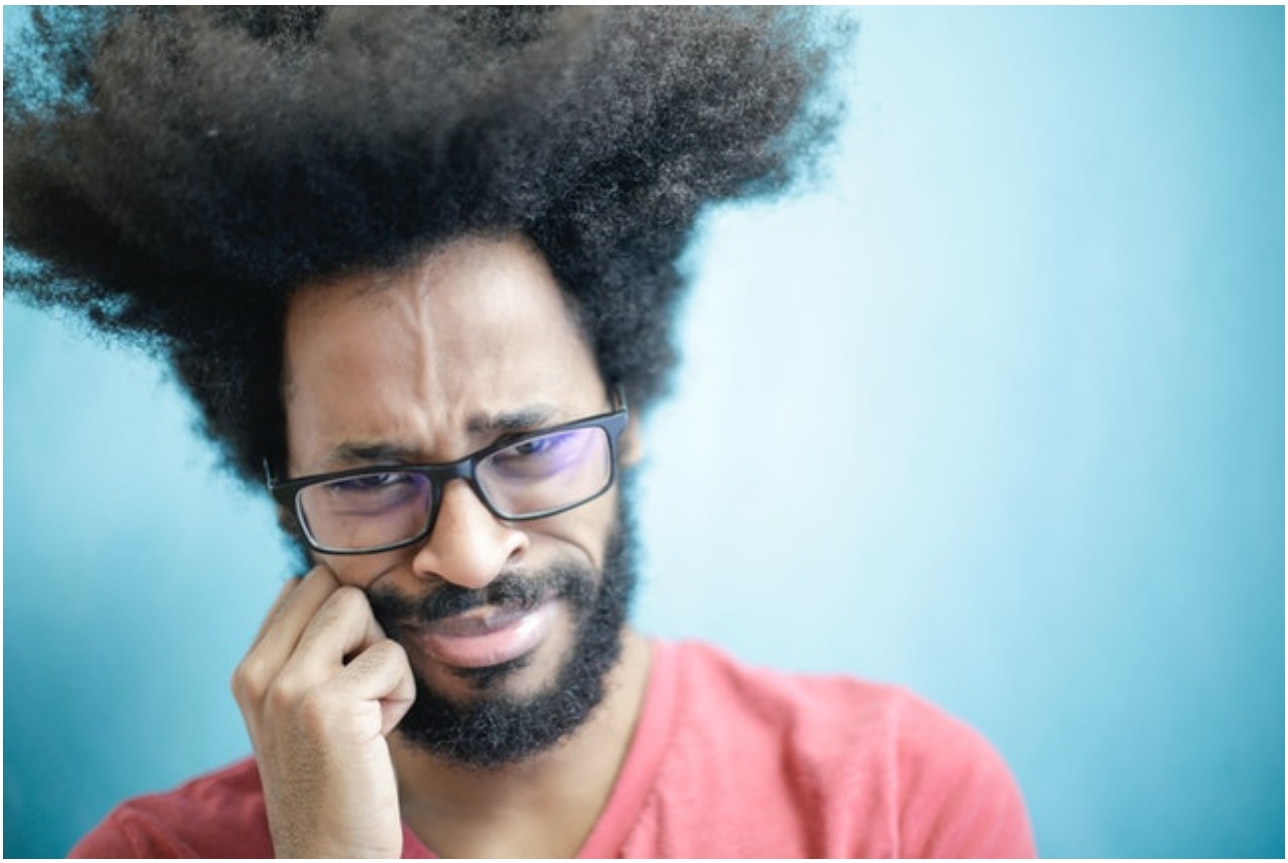
Đây là những gì tôi muốn nói về sự tập trung và kỷ luật: tập trung vào mã (một việc chỉ làm một việc) và kỷ luật của nhà phát triển (không áp dụng cho các giải pháp ngắn hạn).

Chà, đó là một chuyến tham quan! Và chúng tôi đã đề cập chỉ một trong năm nguyên tắc. Tiếp tục nào!

“O” dành cho Đóng mở

Tôi phải nói rằng bất cứ ai đưa ra định nghĩa của những nguyên tắc này chắc chắn không nghĩ đến các nhà phát triển ít kinh nghiệm. Điều tương tự cũng xảy ra với Nguyên tắc Đóng mở, và những nguyên tắc sắp xảy ra đã đi trước một bước về sự kỳ lạ. 😂😂

Bất kể, chúng ta hãy nhìn vào định nghĩa mà mọi người tìm thấy cho nguyên tắc này: Các lớp nên mở để mở rộng nhưng đóng để sửa đổi. Hở?? Vâng, tôi cũng không thích thú khi lần đầu tiên nhìn thấy nó, nhưng theo thời gian, tôi đã hiểu - và ngưỡng mộ - điều mà quy tắc này đang cố gắng nói: mã đã được viết một lần thì không cần phải thay đổi.



Theo nghĩa triết học, quy tắc này rất tuyệt - nếu mã không thay đổi, nó sẽ vẫn có thể dự đoán được và các lỗi mới sẽ không được đưa vào. Nhưng làm thế nào để có thể mơ về những đoạn mã không thay đổi khi mà tất cả những gì chúng ta đang làm với tư cách là các nhà phát triển luôn theo đuổi sự thay đổi?

Trước hết, nguyên tắc không có nghĩa là ngay cả một dòng mã hiện tại cũng không được phép thay đổi; đó sẽ là thẳng ra khỏi một thế giới thần tiên. Thế giới thay đổi, kinh doanh thay đổi, và do đó, mã cũng thay đổi - không thể tránh khỏi điều đó. Nhưng nguyên tắc này có nghĩa là chúng ta hạn chế khả năng thay đổi mã hiện tại càng nhiều càng tốt. Và nó cũng cho bạn biết cách làm điều đó: các lớp nên mở để mở rộng và đóng để sửa đổi.

“Mở rộng” ở đây có nghĩa là sử dụng lại, cho dù việc sử dụng lại ở dạng các lớp con kế thừa chức năng từ lớp cha hay các lớp khác lưu trữ các thể hiện của một lớp và gọi các phương thức của nó.

Vì vậy, quay trở lại câu hỏi hàng triệu đô la: làm thế nào để bạn viết mã tồn tại trước sự thay đổi? Và ở đây, tôi e rằng, không ai có câu trả lời rõ ràng. Trong

Lập trình hướng đối tượng, một số kỹ thuật đã được khám phá và tinh chỉnh để đạt được mục tiêu này, ngay từ các nguyên tắc SOLID này, chúng tôi đang nghiên cứu đến các mẫu thiết kế phổ biến, các mẫu doanh nghiệp, các mẫu kiến trúc, v.v. Không có câu trả lời hoàn hảo và vì vậy một nhà phát triển phải tiếp tục ngày càng cao hơn, thu thập nhiều công cụ nhất có thể và cố gắng làm hết sức mình.

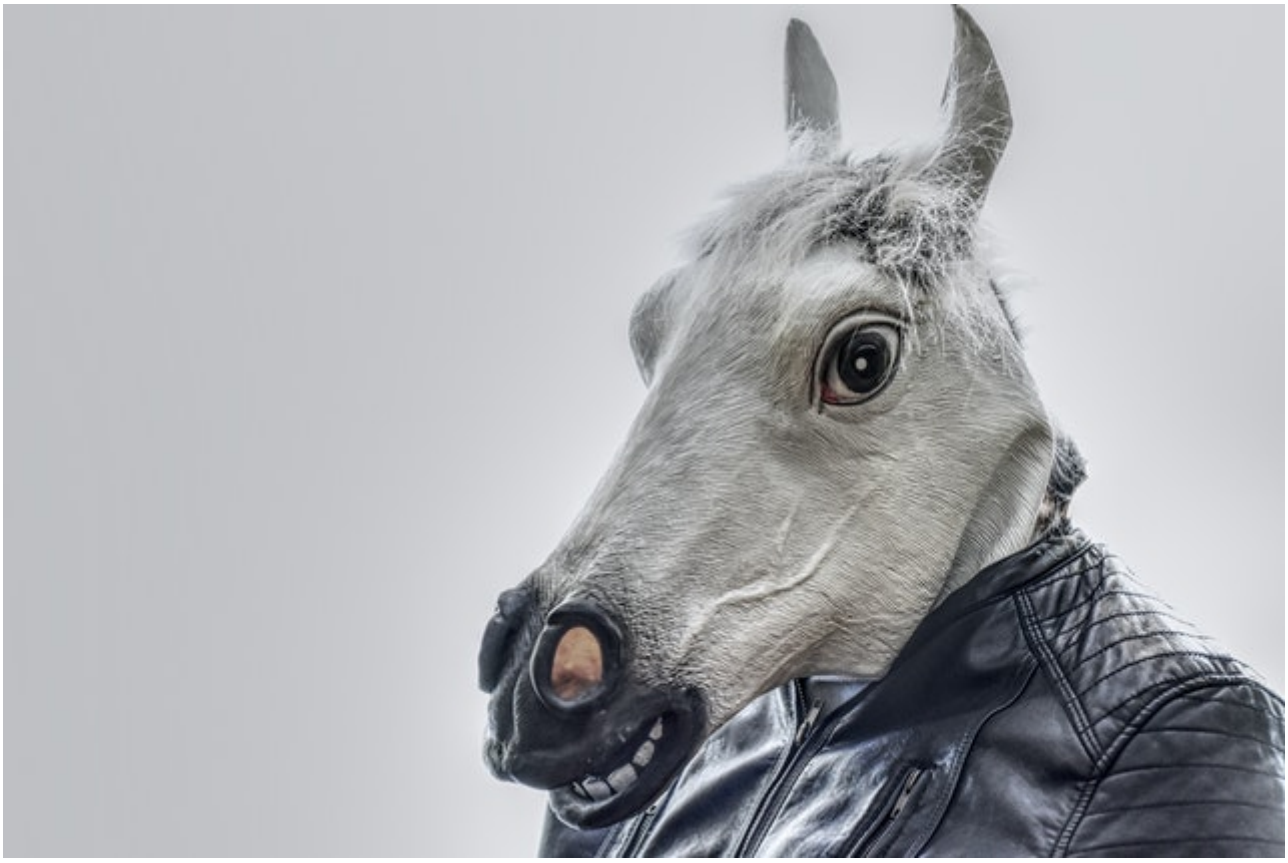
Với ý nghĩ đó, chúng ta hãy xem xét một kỹ thuật như vậy. Giả sử chúng ta cần thêm chức năng để chuyển đổi nội dung HTML nhất định (có thể là hóa đơn?) Thành tệp PDF và cũng buộc tải xuống ngay lập tức trong trình duyệt. Cũng giả sử chúng ta có đăng ký trả phí của một dịch vụ giả định có tên là MilkyWay, dịch vụ này sẽ thực hiện việc tạo PDF thực tế. Chúng tôi có thể kết thúc bằng cách viết một phương thức điều khiển như thế này:

```
class InvoiceController extends Controller {
    public function generatePDFDownload(Request $request) {
        $pdfGenerator = new MilkyWay();
        $pdfGenerator->apiKey = env('MILKY_WAY_API_KEY');
        $pdfGenerator->setContent($request->content); // HTML format
        $pdfFile = $pdfGenerator->generateFile('invoice.pdf');

        return response()->download($pdfFile, [
            'Content-Type' => 'application/pdf',
        ]);
```

```
    }
}
```

Tôi đã bỏ qua xác thực yêu cầu, v.v., để tập trung vào vấn đề cốt lõi. Bạn sẽ nhận thấy rằng phương pháp này thực hiện rất tốt việc tuân theo Nguyên tắc trách nhiệm duy nhất: nó không cố gắng xem qua nội dung HTML được chuyển đến nó và tạo một tệp PDF (thực tế, nó thậm chí không biết rằng nó đã được cấp HTML); thay vào đó, nó chuyển trách nhiệm đó cho `MilkyWay` lớp chuyên biệt và trình bày bất cứ thứ gì nó nhận được, dưới dạng tải xuống.

Nhưng có một vấn đề nhỏ.



Phương thức bộ điều khiển của chúng tôi quá phụ thuộc vào lớp MilkyWay. Nếu phiên bản tiếp theo của API MilkyWay thay đổi giao diện, phương pháp của chúng tôi sẽ ngừng hoạt động. Và nếu chúng tôi muốn sử dụng một số dịch vụ khác vào một ngày nào đó, chúng tôi sẽ phải thực hiện tìm kiếm toàn cầu trong trình soạn thảo mã của mình và thay đổi tất cả các đoạn mã đề cập đến MilkyWay. Và tại sao điều đó lại tồi tệ? Bởi vì nó làm tăng đáng kể khả năng mắc sai lầm và là gánh nặng cho doanh nghiệp (nhà phát triển đã dành thời gian để phân loại mớ hỗn độn).

Tất cả sự lãng phí này bởi vì chúng tôi đã tạo ra một phương pháp không đóng cửa để thay đổi.

Chúng ta có thể làm tốt hơn không?

Có, chúng tôi có thể!

Trong trường hợp này, chúng ta có thể tận dụng lợi thế của một thực hành giống như thế này - chương trình cho các giao diện, không phải triển khai.

Vâng, tôi biết, đó là một OOPSisms khác mà lần đầu tiên không có ý nghĩa gì. Nhưng những gì nó đang nói là mã của chúng ta nên phụ thuộc vào *các loại* thứ

chứ không phải bản thân những thứ cụ thể. Trong trường hợp của chúng tôi, chúng tôi cần giải phóng bản thân khỏi việc phải phụ thuộc vào `MilkyWay` lớp, và thay vào đó phụ thuộc vào một *loại* lớp PDF chung chung (tất cả sẽ trở nên rõ ràng sau một giây).

Bây giờ, chúng ta có những công cụ nào trong PHP để tạo các kiểu mới? Nói rộng ra, chúng ta có Thừa kế và Giao diện. Trong trường hợp của chúng tôi, việc tạo một lớp cơ sở cho tất cả các lớp PDF sẽ không phải là một ý tưởng tuyệt vời vì thật khó để tưởng tượng các loại công cụ / dịch vụ PDF khác nhau có cùng hành vi. Có thể họ có thể chia sẻ `setContent()` phương pháp, nhưng ngay cả ở đó, quá trình thu nhận nội dung có thể khác nhau đối với mỗi lớp dịch vụ PDF, vì vậy việc nhập mọi thứ lên theo hệ thống phân cấp Kế thừa sẽ khiến mọi thứ trở nên tồi tệ hơn.

Với điều đó đã hiểu, hãy tạo một giao diện chỉ định các phương thức mà chúng tôi muốn tất cả các lớp công cụ PDF của chúng tôi chứa:

```
interface IPDFGenerator {
    public function setup(); // API keys, etc.
```

```
    public function setContent($content);
    public function generatePDF($fileName = null);
}
```

Vậy chúng ta có gì ở đây nào?

Thông qua giao diện này, chúng tôi muốn nói rằng tất cả các lớp PDF của chúng tôi có ít nhất ba phương thức đó. Bây giờ, nếu dịch vụ chúng ta muốn sử dụng (trong trường hợp của chúng ta là MilkyWay) không tuân theo giao diện này, thì nhiệm vụ của chúng ta là viết một lớp thực hiện điều đó. Bản phác thảo sơ bộ về cách chúng ta có thể viết một lớp wrapper cho `MilkyWay` dịch vụ của mình như sau:

```
class MilkyWayPDFGenerator implements IPDFGenerator {
    public function __construct() {
        $this->setup();
    }

    public function setup() {
        $this->generator = new MilkyWay();
        $this->generator->api_key = env('MILKY_WAY_API_KEY');
    }

    public function setContent($content) {
        $this->generator->setContent($content);
    }

    public function generatePDF($fileName) {
        return $this->generator->generateFile($fileName);
    }
}
```

Và giống như thế này, bất cứ khi nào chúng tôi có một dịch vụ PDF mới, chúng tôi sẽ viết một lớp wrapper cho nó. Kết quả là, tất cả các lớp đó sẽ được coi là thuộc loại `IPDFGenerator` .

Vì vậy, làm thế nào tất cả những điều này được kết nối với Nguyên tắc Đóng mở và Laravel?

Để đạt được điểm đó, chúng ta phải biết thêm hai khái niệm chính: liên kết vùng chứa của Laravel và một kỹ thuật rất phổ biến được gọi là tiêm phụ thuộc. Một lần nữa, các từ lớn, nhưng việc tiêm phụ thuộc chỉ đơn giản có nghĩa là thay vì tự tạo các đối tượng của các lớp, bạn đề cập đến chúng trong các đối số của hàm và thứ gì đó sẽ tự động tạo chúng cho bạn. Điều này giúp bạn không phải viết mã `$account = new Account();` mọi lúc và làm cho mã dễ kiểm tra hơn (một chủ đề cho ngày khác). “Cái gì đó” mà tôi đã đề cập có dạng Service Container trong thế giới Laravel.

Hiện tại, hãy nghĩ về nó như một thứ có thể tạo ra các thể hiện lớp mới cho chúng ta. Hãy xem điều này giúp ích như thế nào.

Trong Vùng chứa dịch vụ trong ví dụ của chúng ta, chúng ta có thể viết một cái gì đó như sau:

```
$this->app->bind('App\Interfaces\IPDFGenerator', 'App\Services\PDF\MilkyWayPDFGenerato
```

Về cơ bản, điều này nói rằng, bất cứ khi nào ai đó yêu cầu một `IPDFGenerator` , hãy giao cho họ `MilkyWayPDFGenerator` lớp học. Và sau tất cả những bài hát và điệu nhảy đó, thưa quý vị và các bạn, chúng ta đã đến lúc tất cả đã đi vào đúng vị trí và Nguyên tắc Đóng mở được tiết lộ tại nơi làm việc!

Được trang bị tất cả kiến thức này, chúng tôi có thể viết lại phương pháp trình điều khiển tải xuống PDF của mình như sau:

```
class InvoiceController extends Controller {
    public function generatePDFDownload(Request $request, IPDFGenerator $generator) {
        $generator->setContent($request->content);
        $pdfFile = $generator->generatePDF('invoice.pdf');

        return response()->download($pdfFile, [
            'Content-Type' => 'application/pdf',
        ]);
    }
}
```

Chú ý sự khác biệt?

Trước hết, chúng tôi đang nhận được phiên bản lớp trình tạo PDF của chúng tôi trong đối số hàm. Điều này đang được tạo và chuyển cho chúng tôi bởi Vùng

chứa dịch vụ, như chúng tôi đã thảo luận trước đó. Mã cũng sạch hơn và không có đề cập đến các khóa API, v.v. Nhưng, quan trọng nhất, không có dấu vết của `MilkyWay` lớp. Điều này cũng có một lợi ích phụ bổ sung là làm cho mã dễ đọc hơn (ai đó đọc nó lần đầu tiên sẽ không đi, "Whoa! WTF là cái này `MilkyWay` ??" và liên tục lo lắng về nó sau đầu của họ).

Nhưng lợi ích lớn nhất của tất cả?

Phương pháp này hiện đã được đóng lại để sửa đổi và chống thay đổi. Cho phép tôi giải thích. Giả sử ngày mai chúng ta cảm thấy rằng `MilkyWay` dịch vụ quá đắt (hoặc, như thường lệ, bộ phận hỗ trợ khách hàng của họ đã trở nên tồi tệ); do đó, chúng tôi đã thử một dịch vụ khác được gọi là `SilkyWay` và muốn chuyển sang dịch vụ đó. Tất cả những gì chúng ta cần làm bây giờ là viết một `IPDFGenerator` lớp trình bao bọc mới `SilkyWay` và thay đổi ràng buộc trong mã Vùng chứa dịch vụ của chúng tôi:

```
$this->app->bind('App\Interfaces\IPDFGenerator', 'App\Services\PDF\SilkyWayPDFGenerato
```

Đó là tất cả!!

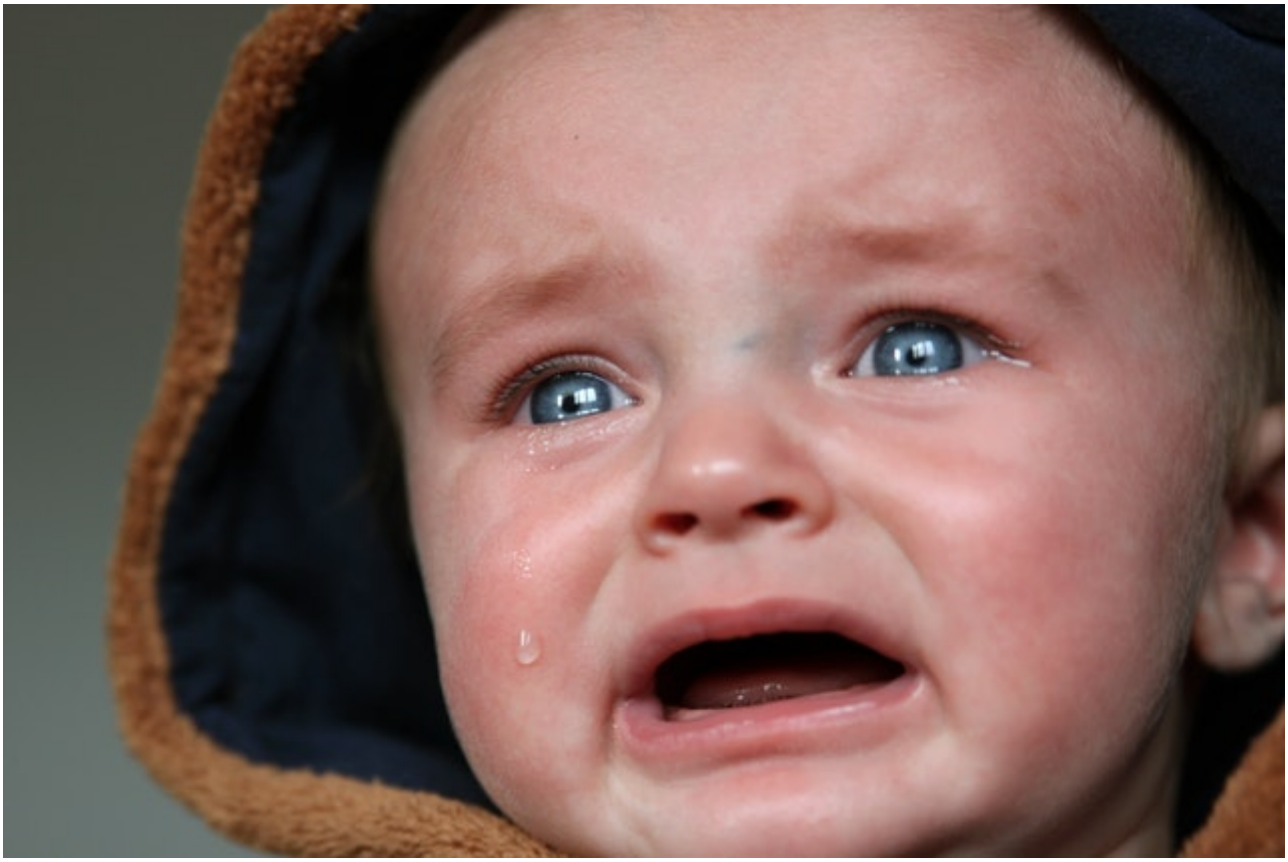
Không có gì khác cần thay đổi, bởi vì ứng dụng của chúng tôi được viết theo một giao diện (giao diện IPDFGenerator) thay vì một lớp cụ thể. Yêu cầu kinh doanh đã thay đổi, một số mã mới được thêm vào (một lớp trình bao bọc) và chỉ một dòng mã được thay đổi - mọi thứ khác vẫn không bị ảnh hưởng và toàn bộ nhóm có thể tự tin về nhà và ngủ yên.

Muốn ngủ yên? Thực hiện theo nguyên tắc đóng mở! 🤭😆

“L” là để thay thế Liskov

Liskov-cái gì ??

Điều này nghe có vẻ giống như một thứ gì đó xuất phát từ một cuốn sách giáo khoa Hóa hữu cơ. Nó thậm chí có thể khiến bạn hối tiếc vì đã chọn phát triển phần mềm như một sự nghiệp vì bạn nghĩ rằng tất cả chỉ là thực tế và không có lý thuyết.



“Nhanh lên, nhóc! Số nguyên tố lớn nhất có thể được biểu diễn dưới dạng tổng của hai số nguyên tố là bao nhiêu? ”

Nhưng hãy giữ những con ngựa của bạn trong một giây! Tin tôi đi, nguyên tắc này dễ hiểu như chính cái tên của nó. Trên thực tế, nó có thể chỉ là nguyên tắc dễ hiểu nhất trong số năm nguyên tắc để hiểu (à, ờ... Nếu không phải là dễ nhất, thì ít nhất nó sẽ có lời giải thích ngắn gọn và đơn giản nhất).

Quy tắc này chỉ đơn giản nói rằng mã hoạt động với các lớp cha (hoặc giao diện) sẽ không bị phá vỡ khi các lớp đó được thay thế bằng các lớp con (hoặc các lớp thực thi giao diện). Ví dụ mà chúng ta đã hoàn thành ngay trước phần này là một minh họa tuyệt vời:

nếu tôi thay thế kiểu chung `IPDFGenerator` trong đối số phương thức
bằng `MilkyWayPDFGenerator` trường hợp cụ thể, bạn có mong đợi mã bị hỏng hoặc tiếp tục hoạt động không?

Tất nhiên, hãy tiếp tục làm việc! Đó là bởi vì sự khác biệt chỉ nằm ở tên, và cả giao diện cũng như lớp đều có các phương thức hoạt động giống nhau, vì vậy mã của chúng ta sẽ hoạt động như trước.

Vì vậy, vấn đề lớn về nguyên tắc này là gì? Nói một cách đơn giản hơn, đó là tất cả những gì mà nguyên tắc này nói: đảm bảo rằng các lớp con của bạn triển khai tất cả các phương thức chính xác theo yêu cầu, với cùng số lượng và kiểu đối số và cùng kiểu trả về. Nếu ngay cả một tham số khác nhau, chúng ta sẽ vô tình tiếp tục xây dựng nhiều mã hơn trên đó, và một ngày nào đó chúng ta sẽ gặp phải một mớ hỗn độn hôi thối mà giải pháp duy nhất là gỡ nó xuống.

Ở đó. Bây giờ điều đó không quá tệ, phải không? 😇

Có thể nói nhiều hơn nữa về Sự thay thế Liskov (hãy tra cứu lý thuyết của nó và đọc trên Các loại Covariant nếu bạn thực sự cảm thấy dũng cảm), nhưng theo tôi,

bấy nhiêu là đủ cho các nhà phát triển trung bình đi qua những vùng đất bí ẩn của các mô hình nguyên tắc lần đầu tiên.

“I” là phân biệt giao diện

Giao diện phân tách. . . hmm, nghe không tệ lắm phải không? Có vẻ như đó là một cái gì đó để làm với sự tách biệt. . . umm, tách ra. . . các giao diện. Tôi chỉ tự hỏi ở đâu và như thế nào.

Nếu bạn đang nghĩ theo những dòng này, hãy tin tôi, bạn gần như đã hiểu và sử dụng nguyên tắc này. Nếu tất cả năm nguyên tắc SOLID đều là phương tiện đầu tư, thì nguyên tắc này sẽ mang lại giá trị lâu dài nhất trong việc học viết mã tốt (được thôi, tôi nhận ra rằng tôi nói vậy về mọi nguyên tắc, nhưng bạn biết đấy, bạn hiểu đấy).

Loại bỏ biệt ngữ highfalutin và chắt lọc xuống dạng cơ bản nhất của nó, nguyên tắc Phân tách giao diện có ý nghĩa như sau: Càng có nhiều giao diện chuyên dụng trong ứng dụng của bạn, thì mã của bạn càng có nhiều mô-đun và ít kỳ lạ hơn.

Hãy xem một ví dụ rất phổ biến và thực tế. Mọi nhà phát triển Laravel đều bắt gặp cái gọi là Mô hình kho lưu trữ trong sự nghiệp của họ, sau đó họ dành vài tuần tiếp theo để trải qua giai đoạn cao và thấp theo chu kỳ, và cuối cùng loại bỏ mô hình đó. Tại sao? Trong tất cả các hướng dẫn về Mẫu Kho lưu trữ, bạn nên tạo một giao diện chung (được gọi là Kho lưu trữ) sẽ xác định các phương thức cần thiết để truy cập hoặc thao tác dữ liệu. Giao diện cơ sở này có thể trông giống như sau:

```
interface IRepository {
    public function getOne($id);
    public function getAll();
    public function create(array $data);
    public function update(array $data, $id);
    public function delete($id);
}
```

Và bây giờ, đối với `User` mô hình của bạn , bạn phải tạo một `UserRepository` triển khai giao diện này; sau đó, đối với `Customer` mô hình của bạn , bạn phải tạo một `CustomerRepository` triển khai giao diện này; bạn có được ý tưởng.

Bây giờ, nó đã xảy ra trong một trong những dự án của tôi rằng một số mô hình không được cho phép bởi

bất kỳ ai ngoài hệ thống. Trước khi bạn bắt đầu đảo mắt, hãy cân nhắc rằng việc ghi nhật ký hoặc duy trì dấu vết kiểm tra là một ví dụ thực tế tốt về các mô hình “chỉ đọc” như vậy. Vấn đề mà tôi phải đối mặt là vì tôi phải tạo các kho lưu trữ tất cả triển khai `IRepository` giao diện, chẳng hạn như `LoggingRepository` , ít nhất hai trong số các phương thức trong giao diện, `update()` và `delete()` không có ích gì đối với tôi.

Đúng, cách khắc phục nhanh chóng là vẫn thực hiện những điều này và để trống hoặc nêu ra một ngoại lệ, nhưng nếu dựa vào các giải pháp băng keo như vậy là ổn đối với tôi, thì ngay từ đầu tôi đã không tuân theo Mô hình Kho lưu trữ!

Trợ giúp! Tôi bị mắc kẹt. : '(

Điều đó có nghĩa là tất cả đều là lỗi của mẫu Kho lưu trữ?

Không hoàn toàn không!

Trên thực tế, Kho lưu trữ là một mẫu nổi tiếng và được chấp nhận, mang lại tính nhất quán, tính linh hoạt và tính trừu tượng cho các mẫu truy cập dữ liệu của bạn. Vấn đề là giao diện chúng tôi đã tạo - hay tôi nên nói giao diện phổ biến trong mọi hướng dẫn thực tế - *quá rộng* .

Đôi khi ý tưởng này được thể hiện bằng cách nói rằng giao diện là "chất béo", nhưng nó cũng có nghĩa tương tự - giao diện tạo ra quá nhiều giả định và do đó thêm các phương thức vô dụng cho một số lớp nhưng các lớp đó vẫn bị buộc phải triển khai chúng, dẫn đến trong mã giòn, khó hiểu. Ví dụ của chúng tôi có thể đơn giản hơn một chút, nhưng hãy tưởng tượng mớ hỗn độn có thể được tạo ra khi một số lớp đã triển khai các phương thức mà họ không muốn hoặc những phương thức họ muốn nhưng bị thiếu trong giao diện.

Giải pháp rất đơn giản và cũng là tên của nguyên tắc chúng ta đang thảo luận: Phân tách giao diện.

Vấn đề là, chúng ta không nên tạo giao diện của mình một cách mù quáng. Và chúng ta cũng không nên đưa ra giả định, cho dù chúng ta nghĩ mình có kinh nghiệm hay thông minh đến đâu. Thay vào đó, chúng ta nên tạo một số giao diện nhỏ hơn, chuyên biệt, cho phép các lớp thực thi những giao diện cần thiết và loại bỏ những giao diện không.

Trong ví dụ chúng ta đã thảo luận, tôi có thể đã tạo hai giao diện thay vì một: `IReadOnlyRespository` (chứa các

hàm `getOne()` và `getAll()`), và `IWriteModifyRepository` (chứa phần còn lại của các hàm). Đối với các kho lưu trữ thông thường, tôi sẽ nói `class UserRepository implements IReadOnlyRepository, IWriteModifyRepository { ... }`. (Lưu ý: Các trường hợp đặc biệt vẫn có thể phát sinh, và điều đó không sao cả vì không có thiết kế nào là hoàn hảo. Bạn thậm chí có thể muốn tạo một giao diện riêng cho mọi phương pháp và điều đó cũng sẽ ổn thôi, giả sử nhu cầu của dự án của bạn là chi tiết.)

Vâng, có nhiều giao diện hơn bây giờ và một số người có thể nói rằng có quá nhiều thứ để nhớ hoặc khai báo lớp bây giờ quá dài (hoặc trông xấu xí), v.v., nhưng hãy nhìn vào những gì chúng ta đã đạt được: chuyên biệt, kích thước nhỏ, tự -giao diện được kiểm soát có thể được kết hợp khi cần thiết và sẽ không cản trở nhau. Miễn là bạn viết phần mềm để kiếm sống, hãy nhớ rằng đó là lý tưởng mà mọi người đang phấn đấu.

“D” là Đảo ngược phụ thuộc

Nếu bạn đã đọc các phần trước của bài viết này, bạn có thể cảm thấy rằng bạn hiểu nguyên tắc này đang muốn nói gì. Và bạn đã đúng, theo nghĩa là nguyên tắc này ít nhiều là sự lặp lại những gì chúng ta đã thảo luận cho đến bây giờ. Định nghĩa chính thức của nó không quá đáng sợ, vì vậy chúng ta hãy nhìn vào nó: Các mô-đun cấp cao không nên phụ thuộc vào các mô-đun cấp thấp; cả hai đều nên phụ thuộc vào sự trừu tượng.

Vâng, có lý. Nếu tôi có một lớp cấp cao (cấp cao theo nghĩa là nó sử dụng các lớp khác nhỏ hơn, chuyên biệt hơn để thực hiện một cái gì đó và sau đó đưa ra một số quyết định), chúng ta không nên có lớp cấp cao đó tùy thuộc vào một mức thấp cụ thể- cấp độ cho một số loại công việc. Thay vào đó, chúng nên được mã hóa để dựa trên các yếu tố trừu tượng (chẳng hạn như các lớp cơ sở, giao diện, v.v.).

Tại sao?

Chúng ta đã thấy một ví dụ tuyệt vời về nó trong phần trước của bài viết này. Nếu bạn đã sử dụng một dịch vụ tạo PDF và mã của bạn có nhiều `new ABCService()` lớp, ngày doanh nghiệp quyết định sử

dụng một số dịch vụ khác sẽ là ngày được ghi nhớ mãi mãi - vì tất cả những lý do sai lầm! Thay vào đó, chúng ta nên sử dụng một dạng chung của sự phụ thuộc này (nghĩa là tạo giao diện cho các dịch vụ PDF) và để một thứ khác xử lý việc khởi tạo nó và chuyển nó cho chúng ta (trong Laravel , chúng ta đã thấy Service Container đã giúp chúng ta làm điều đó như thế nào) .

Nói chung, lớp cấp cao của chúng ta, trước đó có quyền kiểm soát việc tạo các cá thể của lớp cấp thấp hơn, giờ phải nhìn sang một thứ khác. Các bảng đã thay đổi, và đó là lý do tại sao chúng tôi gọi đây là *sự đảo ngược* của các phụ thuộc.

Nếu bạn đang tìm kiếm một ví dụ thực tế, hãy quay lại phần trong bài viết này, nơi chúng ta thảo luận về cách giải cứu mã của chúng ta khỏi việc phải phụ thuộc hoàn toàn vào `MilkyWay` lớp PDF.

. . .

Đoán xem, đó là nó! Tôi biết, tôi biết, nó khá dài và khó đọc, và tôi muốn xin lỗi vì điều đó. Nhưng trái tim của tôi dành cho một nhà phát triển bình thường, người đang làm mọi thứ theo trực giác (hoặc theo

cách họ được dạy cho anh ta) và không thể hiểu được đầu hay đuôi của các nguyên tắc SOLID. Và tôi đã cố gắng hết sức để giữ các ví dụ gần với ngày làm việc của nhà phát triển Laravel nhất có thể; xét cho cùng, những gì sử dụng đối với chúng tôi là các ví dụ có chứa các lớp Xe và Xe - hoặc thậm chí là các phân loại chung nâng cao, phản ánh, v.v. - khi không ai trong chúng ta sẽ là tác giả của các thư viện.

Nếu bạn thấy bài viết này hữu ích, hãy để lại bình luận. Điều này sẽ xác nhận quan điểm của tôi rằng các nhà phát triển đang *thực sự* đấu tranh để hiểu những khái niệm "nâng cao" này và tôi sẽ có động lực để viết về những chủ đề khác như vậy. Cho đến sau này! 🙂