DESIGN PATTERNS

FOR

BEGINNERS

Phần 2 - Các Pattern

Biên tập bởi Đội ngũ CodeGym

DESIGN PATTERNS FOR BEGINNERS

PHẦN 2 - CÁC PATTERN

ĐỘI NGŨ BIÊN TẬP

Nguyễn Khắc Nhật

Nguyễn Bình Sơn

Nguyễn Khánh Tùng

Phan Văn Luân

LỜI TỰA

Làm trong ngành công nghiệp phát triển phần mềm, bạn đã từng gặp phải trường hợp như thế này?

Bạn làm qua nhiều dự án khác nhau và nhận ra rằng, trong những dự án đó, mình luôn dùng một phương pháp, cách làm theo một khuôn mẫu nào đó để giải quyết các vấn đề gặp phải, khi chúng tương đồng với nhau và đây là những thứ thường xuyên. Lặp lại, lặp lại…

Nếu có, đó chính là lúc mà bạn cần đến Design Pattern!

Được xây dựng theo dạng "template" - Design pattern là các giải pháp tổng thể đã được tối ưu hóa, được tái sử dụng cho các vấn đề phổ biến trong thiết kế phần mềm mà chúng ta thường gặp phải hàng ngày. Đây là tập các giải pháp đã được suy nghĩ, đã giải quyết trong tình huống cụ thể.

Tiếp theo đây, bạn có nghĩ mình phù hợp để đọc cuốn eBook này?

Điều quan trọng tôi muốn nói rằng: Design Pattern không dành cho những bạn mới bắt đầu tìm hiểu về lập trình. Muốn tìm hiểu và học được Design Pattern, bạn cần nắm cơ bản được kiến thức OOP đặc biệt là về abstract class, interface và static.

Không dành cho người mới tìm hiểu về lập trình, vậy tại sao tựa sách lại là *"for Beginners"***.** Ở đây, chúng tôi muốn đem những người mới, những kẻ "dummy" đến với Design Pattern. Họ sẽ là những người bắt đầu làm quen với các "mẫu" và áp dụng nó để phát triển hơn kỹ năng, tay nghề của mình!

--

Nối tiếp Phần 1: Tổng quan về Design Pattern, ban biên tập tiếp tục xây dựng và gửi tới các bạn Phần 2: Các Pattern trong cuốn Design Patterns for Beginners.

*Các bạn có thể tham khảo Design Patterns for Beginners TẠI ĐÂY!

Những nội dung dưới đây là kiến thức được tổng hợp và biên tập lại từ đội ngũ sản xuất. Bằng tất cả sự nỗ lực để đem tới những kiến thức thực sự cần thiết và trọng tâm nhất cho những người bắt đầu tìm hiểu về Design Pattern.

Trong quá trình biên tập, đôi khi không tránh khỏi những sai sót, rất mong nhận được sự đóng góp của các anh, chị, em để cuốn eBook được hoàn thiện hơn.

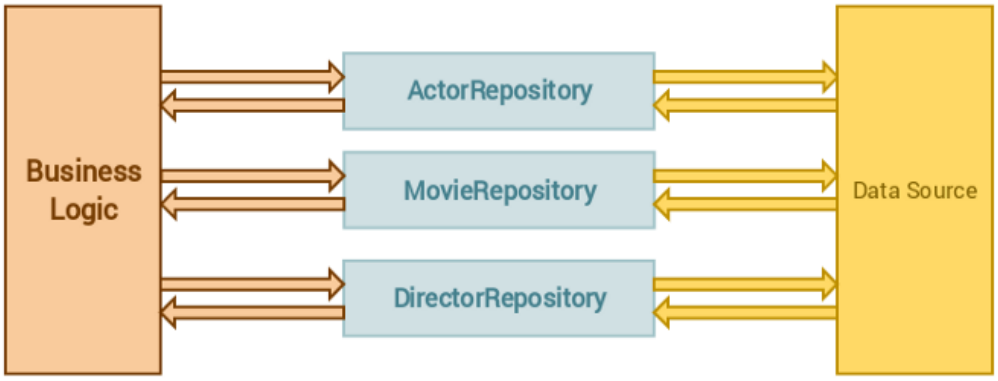
Thân mến,

Đội ngũ biên tập!

MỤC LỤC

Bài 1. [Repository Pattern] Repository design pattern hoàn thiện trong Laravel

Trong bài viết này tôi sẽ chỉ cho bạn cách thiết lập Repository design pattern trong Laravel từ đầu. Tôi sẽ sử dụng phiên bản Laravel 5.8.3, nhưng phiên bản Laravel cũng không thực sự quá quan trọng. Trước khi chúng tôi bắt đầu code, có một vài điều bạn cần biết về repository design pattern.



Repository Pattern là lớp trung gian giữa tầng Data Access và Business Logic. Repository design pattern cho phép bạn sử dụng các đối tượng mà không cần phải biết các đối tượng này được duy trì như thế nào. Về cơ bản nó là một sự trừu tượng hóa của lớp dữ liệu.

Điều này có nghĩa là Business Logic của bạn không cần phải biết cách lấy lại dữ liệu hay nguồn dữ liệu là gì. Business Logic dựa trên repository để lấy dữ liệu chính xác.

Một quan niệm sai lầm mà tôi thấy rất nhiều là các repository đang được thực hiện theo cách tạo hay cập nhật các bản ghi. Đây không phải là những gì mà repository nên làm. Các kho lưu trữ không nên tạo hoặc cập nhật dữ liệu, nhưng chỉ nên được sử dụng để truy xuất dữ liệu.

Hãy bắt tay vào việc code bây giờ!

Bởi vì tôi hướng dẫn các bạn làm từ đâu, đầu tiên ta phải tạo một project laravel trước.

```
composer create-project --prefer-dist laravel/laravel repository
```

Đối với phần hướng dẫn này, tôi sẽ tạo ra một blog nhỏ. Bây giờ chúng tôi đã tạo một project, chúng tôi cần tạo `Controller` và `Model` cho blog.

```
php artisan make:controller BlogController
```

Cái này sẽ tạo `BlogController` trong tệp `app/Http/Controllers`

```
php artisan make:model Models/Blog -m
```

Ghi chú: Lựa chọn -m sẽ tạo ra một Data Migration (chuyển đổi dữ liệu). File này có thể được tìm thấy trong `database/migrations`

Điều này sẽ tạo Model cho Blog của bạn và lưu trữ nó trong thư mục App / Models. Đây chỉ là một trong những cách để lưu trữ các Model của bạn, và là phương pháp mà tôi thích.

Bây giờ chúng ta đã có Controller và Model, đã đến lúc xem tệp chuyển đổi mà chúng ta đã tạo. Bây giờ blog cần một tiêu đề, nội dung và trường `user_id;` bên cạnh các trường `timestamp` (dấu thời gian) mà là mặc định của Laravel.

```
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateBlogsTable extends Migration
{
   public function up()
   {
       Schema::create('blogs', function (Blueprint $table) {
           $table->bigIncrements('id');
           $table->string('title');
           $table->text('content');
           $table->integer('user_id');
           $table->timestamps();

           $table->foreign('user_id')
                 ->references('id')
                 ->on('users');
       });
   }

   public function down()
   {
       Schema::dropIfExists('blogs');
   }
}
```

Ghi chú: Nếu bạn đang dùng phiên bản cũ hơn Laravel 5.8, bạn nên thay thế dòng

```
$table->bigIncrements('id'); ---> $table->increments('id');
```

Thiết lập kho database:

Tôi dùng MySQL cho ví dụ này. Bước đầu tiên là tạo một cơ sở dữ liệu mới.

```
mysql -u root -p
create database laravel_repository;
```

Điều này sẽ tạo ra một `database` gọi là `laravel_repository`. Tiếp theo chúng ta phải thêm thông tin cơ sở dữ liệu vào tệp `.env`

```
DB_DATABASE=laravel_repository
DB_USERNAME=root
DB_PASSWORD=secret
```

Sau khi bạn đã thay đổi tệp `.env`, chúng tôi phải xóa bộ đệm cấu hình:

```
php artisan config:clear
```

Chạy chuyển đổi dữ liệu

Sau khi ta đã set-up xong phần database, ta có thể bắt đầu chạy phần chuyển đổi dữ liệu

```
php artisan migrate
```

Điều này sẽ tạo blog với các trường tiêu đề, nội dung và `user_id` mà chúng tôi đã khai báo trong chuyển đổi dữ liệu.

Thực hiện repository design pattern

Với những gì ta đã làm, bây giờ chúng ta có thể bắt đầu thực hiện repository design pattern. Chúng tôi sẽ bắt đầu bằng cách tạo thư mục Repositories trong thư mục App. Tiếp theo chúng ta sẽ tạo thư mục `Interfaces`. Thư mục này sẽ được đặt trong thư mục `Repositories` mà chúng ta vừa tạo.

Trong thư mục Interfaces, chúng ta tạo lớp `BlogReositoryInterface` với hai phương thức:

- Phương thức `all` sẽ trả về tất cả các blog
- Phương thức `getByUser` sẽ trả về tất cả các blog mà được tạo bởi một user cụ thể.

```
<?php

namespace App\Repositories\Interfaces;

use App\User;

interface BlogRepositoryInterface
{
   public function all();

   public function getByUser(User $user);
}
```

Lớp cuối cùng mà chúng ta sẽ tạo là lớp `BlogRepository`, lớp này sẽ triển khai lớp `BlogRepositoryInterface`. Chúng tôi sẽ thực hiện việc nay đơn giản nhất có thể.

```
<?php

namespace App\Repositories;

use App\Models\Blog;
use App\User;
use App\Repositories\Interfaces\BlogRepositoryInterface;
```

```
class BlogRepository implements BlogRepositoryInterface
{
   public function all()
   {
       return Blog::all();
   }

   public function getByUser(User $user)
   {
       return Blog::where('user_id'. $user->id)->get();
   }
}
```

Thư mục Repositories của bạn nên trông như thế này:

```
app/
└── Repositories/
    ├── BlogRepository.php
    └── Interfaces/
        └── BlogRepositoryInterface.php
```

Bây giờ bạn đã thành công tạo repository! Giờ ta sẽ bắt đầu sử dụng nó.

Sử dụng Repository

Để bắt đầu sử dụng `BlogRepository`, chúng ta nên đưa nó vào `BlogController`. Vì `repository` sẽ được chèn, nên sẽ dễ dàng trao đổi nó với một thực thi khác. `Controller` sẽ trông như sau:

```
<?php

namespace App\Http\Controllers;

use App\Repositories\Interfaces\BlogRepositoryInterface;
use App\User;

class BlogController extends Controller
{
   private $blogRepository;

   public function __construct(BlogRepositoryInterface
$blogRepository)
   {
       $this->blogRepository = $blogRepository;
   }

   public function index()
   {
       $blogs = $this->blogRepository->all();

       return view('blog')->withBlogs($blogs);
   }

   public function detail($id)
   {
       $user = User::find($id);
       $blogs = $this->blogRepository->getByUser($user);
```

```
        return view('blog')->withBlogs($blogs);
    }
}
```

Như bạn có thể thấy mã trong `controller` ngắn và do đó có thể đọc được. Bạn không cần mười dòng mã để có được bộ dữ liệu bạn muốn, tất cả có thể được thực hiện trong một dòng mã nhờ vào `repository`. Điều này cũng rất tốt cho kiểm thử đơn vị, vì các phương thức của repository có thể dễ dàng kiểm tra qua.

Repository design pattern cũng giúp dễ dàng thay đổi giữa các nguồn dữ liệu. Trong ví dụ này, chúng tôi đang sử dụng cơ sở dữ liệu để truy xuất blog của mình. Chúng ta dựa vào `Eloquent` để làm điều đó cho. Nhưng giả sử ta tìm thấy một blog API tuyệt vời ở đâu đó trên mạngvà chúng tôi muốn sử dụng API này. Tất cả những gì chúng ta phải làm là viết lại `BlogRepository` để sử dụng API đó thay vì Eloquent.

RepositoryServiceProvider

Thay vì chèn BlogRepository vào trong BlogController, ta có thể chèn `BlogRepositoryInterface` và sau đó để Service Container quyết định repository nào sẽ được sử dụng. Điều này có thể được thực hiện trong phương thức boot của `AppServiceProvider`, nhưng tôi thích tạo 1 provider mới cho việc này để giữ mọi thứ clean.

```
php artisan make:provider RepositoryServiceProvider
```

Lý do ta tạo ra một provider mới cho việc này là vì mọi thứ trở nên thực sự lộn xộn khi dự án của bạn bắt đầu phát triển. Hãy tưởng tượng một dự án với hơn 10 model và mỗi model đều có repository riêng. `AppServiceProvider` của bạn sẽ trở nên không thể đọc được.

RepositoryServiceProvider của bạn sẽ như sau:

```
<?php

namespace App\Providers;

use App\Repositories\BlogRepository;
use App\Repositories\Interfaces\BlogRepositoryInterface;
use Illuminate\Support\ServiceProvider;

class RepositoryServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->bind(
            BlogRepositoryInterface::class,
            BlogRepository::class
        );
    }
}
```

Hãy nhớ rằng việc hoán đổi BlogRepository với một repository khác dễ như thế nào.

Đừng quên thêm RepositoryServiceProvider vào danh sách các provider trong tệp `config/app.php`. Sau đó, chúng ta phải xóa bộ đệm cấu hình một lần nữa.

```
php artisan config:clear
```

Và bạn đã thực hiện thành công repository design pattern. Cũng không khó quá đúng không?

Bài 2. [Observer Pattern] Sử dụng Observer pattern trong JavaScript

Observer pattern là một pattern khá hữu ích cho các dự án web nói chung và dự án ngôn ngữ lập trình JavaScript nói riêng. Qua kinh nghiệm làm việc, tác giả nhận thấy pattern này giải quyết được nhiều tình huống thường gặp. Để đọc hiểu bài viết này, bạn đọc cần có một số kiến thức cơ bản sau:

- Có hiểu biết cơ bản về Design Pattern
- Nắm vững cú pháp cơ bản trong JavaScript

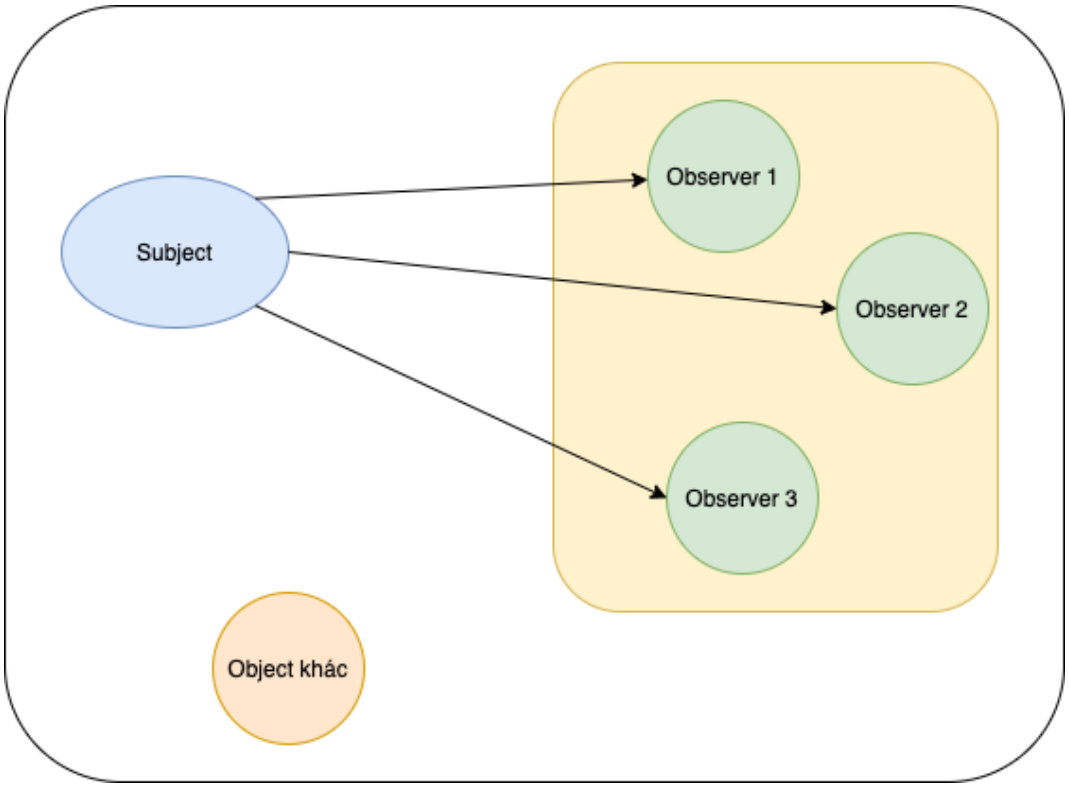
Giới thiệu

Khái niệm

Observer pattern là một pattern trong thiết kế phần mềm, trong đó có một đối tượng được gọi là Subject (chủ thể) có nhiệm vụ quản lý các đối tượng phụ thuộc (được gọi là Observer). Khi trạng thái của Subject thay đổi, Subject sẽ thông báo đến các Observer thông qua cơ chế do Observer cung cấp. Pattern này nằm trong nhóm Behavioral (Hành vi) - nhóm giải quyết các vấn đề tương tác giữa các đối tượng với nhau.

Triển khai Observer khá đơn giản. Có 2 thành phần chính cần được xây dựng:

- **Subject:** Là chủ thể của trạng thái dữ liệu. Đối tượng này duy trì danh sách các đối tượng phụ thuộc khác (được gọi là Observers) và thông báo cho các đối tượng đó khi trạng thái dữ liệu thay đổi. Ví dụ: nhận tin nhắn mới từ máy chủ, người dùng click vào vị trí nào đó trên giao diện,...
- **Observer**: Thành phần nhận thông báo từ Subject và thực hiện những hành vi tương ứng.



Quan sát hình mình hoạ ở trên, các bạn có thể thấy một đối tượng Subject có mối quan hệ đến những đối tượng Observer trong vùng màu vàng. Khi Subject có sự thay đổi hoặc cập nhật, các Observer sẽ nhận được thông báo. Còn "Object khác" không phải là Observer nên sẽ không nhận được cập nhật từ Subject.

Ý nghĩa

Như đã trình bày, một Observer có thể được gán hoặc xoá khỏi danh sách trong Subject. Việc này giúp mã nguồn linh hoạt hơn khi xử lý các tình huống gửi thông báo. Đôi khi bạn sẽ gặp trường hợp như: có một thông báo được phép gởi đồng loạt cho tất cả observer, và thông báo khác thì chỉ gửi tới một số lượng hạn chế các observer liên quan.

Ví dụ tình huống sau:

- Khi bạn chuyển khoản thành công trên hệ thống Internet Banking, hệ thống có thể gửi thông báo đồng loạt đến các kênh:
    - Tin nhắn điện thoại (SMS)
    - Email cá nhân
    - Thông báo trên mobile app (Notification)
- Tuy nhiên, với chức năng xác thực để chuyển khoản, nếu ngân hàng mà bạn sử dụng có dùng giải pháp OTP (One - Time Password) thì tin nhắn chứa mã OTP sẽ chỉ gửi đến tin nhắn di động, mà không gửi qua email hoặc notification của mobile app.

Triển khai với JavaScript

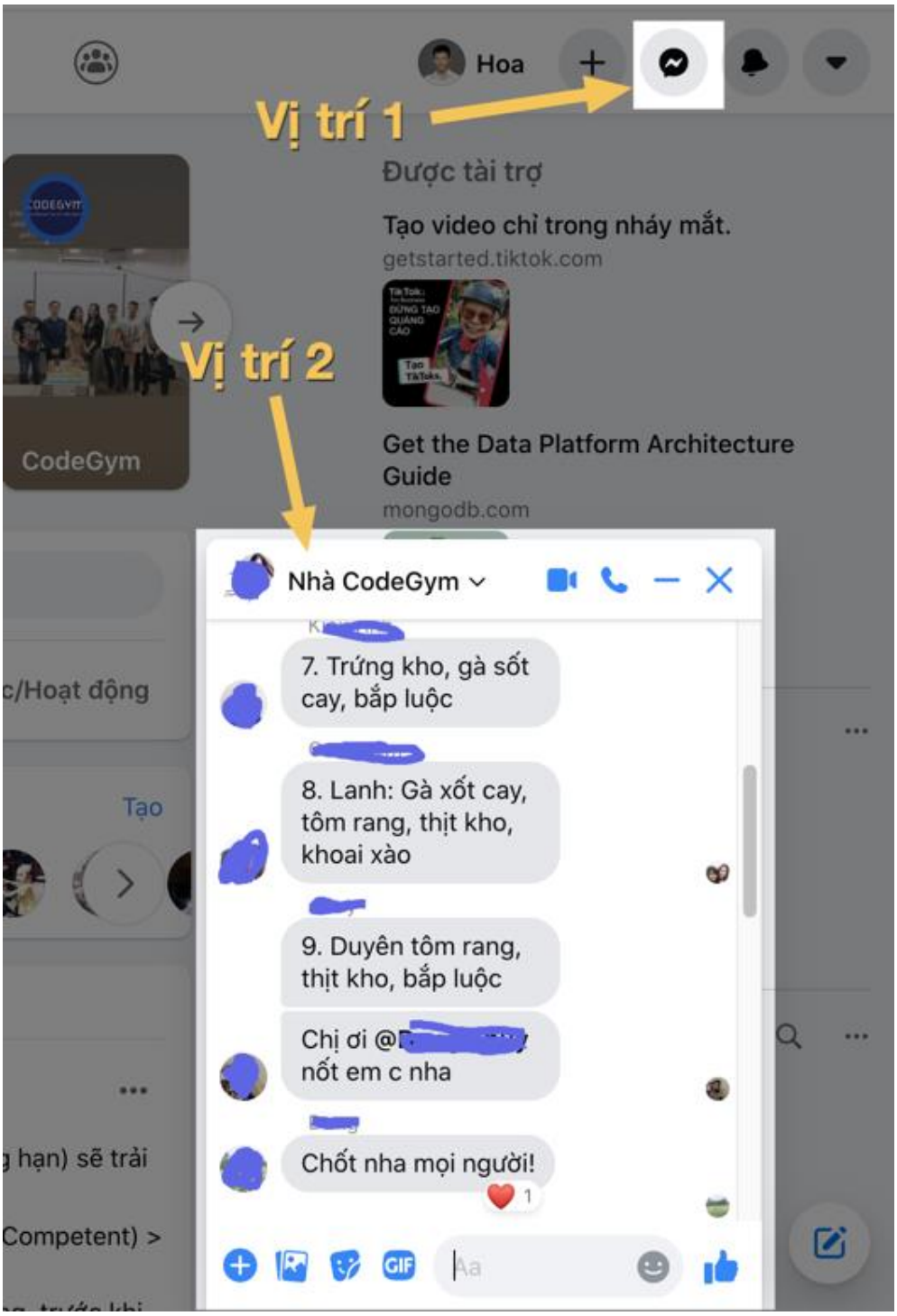
Trong thế giới các ngôn ngữ lập trình Hướng đối tượng (OOP) như Java, C#,... triển khai mẫu này cần có các Interface phục vụ mục đích giảm tính phụ thuộc giữa các thành phần trong ứng dụng:

Thành phần	Mô tả
Subject	Là một interface khai báo các phương thức quản lý danh sách các Observer
Observer	Là interface cung cấp cơ chế cập nhật trạng thái từ Subject
ConcreteSubject	Triển khai mã cụ thể cho interface Subject
ConcreteObserver	Triển khai mã cụ thể cho interface Observer

Với ngôn ngữ JavaScript, mặc dù một số từ khoá theo mô hình OOP (như class, extends) vẫn được hỗ trợ, nhưng lại thiếu cơ chế abstraction như `abtract class`, `interface`. Vậy nên, việc triển khai Observer pattern sẽ hơi khác so với các hướng dẫn dành cho OOP. Phần triển khai cụ thể sẽ được trình bày chi tiết hơn ở mục Giải pháp (Xây dựng demo).

Tình huống

Để giải thích pattern này dễ hiểu nhất có thể, tác giả mượn một tình huống khá quen thuộc trong ứng dụng mạng xã hội Facebook. Đó là cửa sổ chat Messenger khi người dùng lướt Newsfeed như hình ảnh dưới đây:



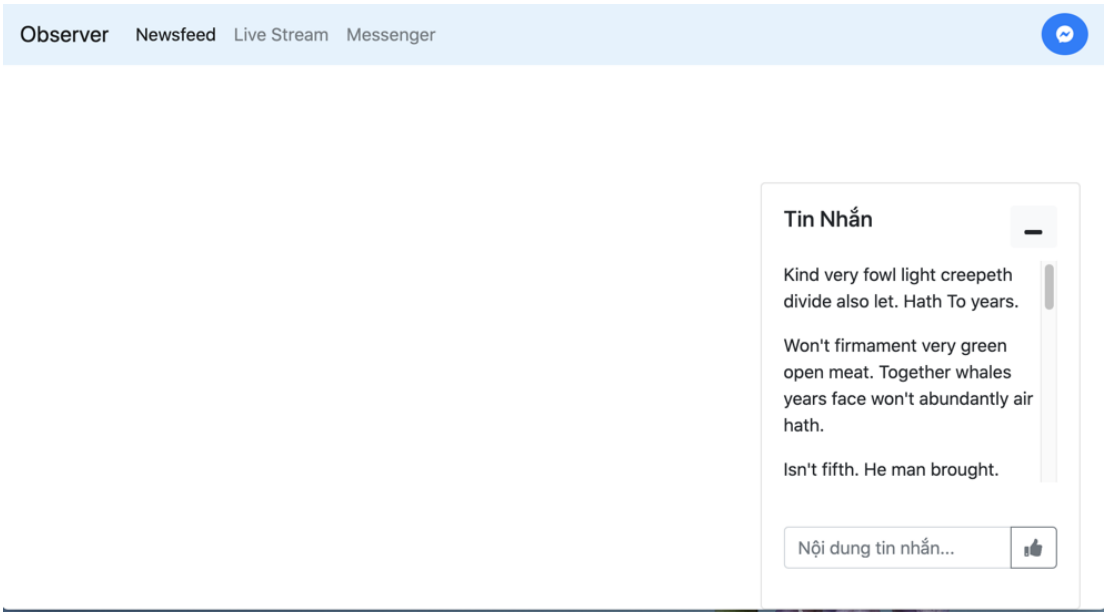
Ở hình trên, tác giả đánh dấu hai vị trí cần lưu ý:

- Vị trí thứ nhất là thông báo số tin nhắn mới từ Messenger.
- Vị trí thứ hai là cửa sổ chat Messenger khi người dùng lướt Newsfeed. Nếu ở dạng rút gọn (minimize) thì vị trí này sẽ hiển thị số tin mới chưa được đọc (giống vị trí thứ nhất).

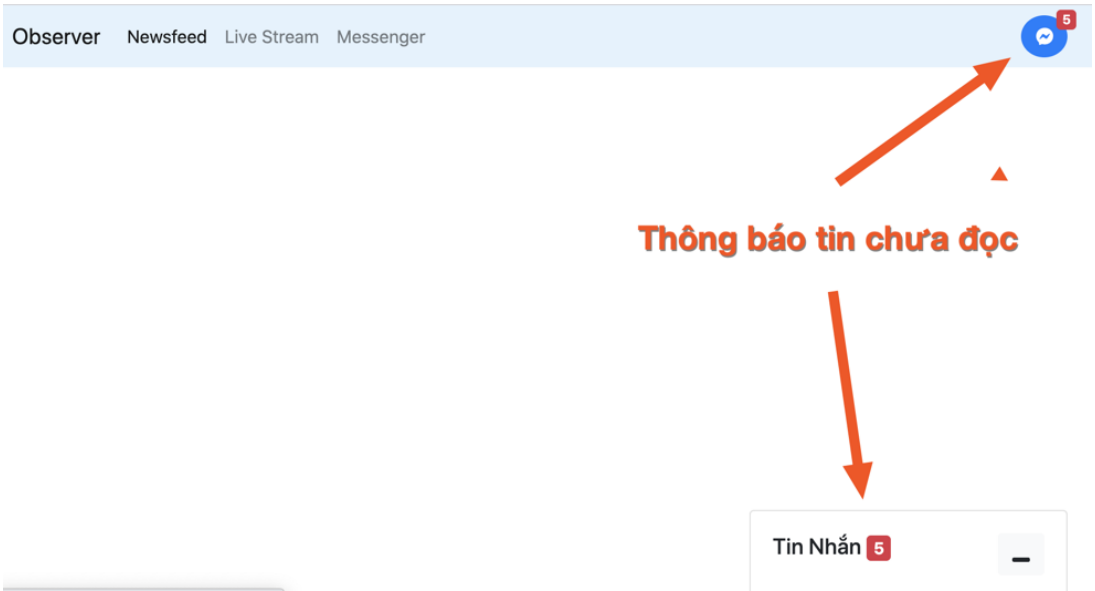
Khi người dùng đọc các tin mới, số tin nhắn ở cả hai vị trí đều được reset và ẩn đi. Trong tình huống người dùng đang lướt newsfeed và có tin nhắn mới, số tin nhắn mới sẽ được cập nhật ở cả hai vị trí trên.

Chúng ta bắt đầu xây dựng tính năng này.

Trước hết, cần có một layout đơn giản với Bootstrap để mô phỏng lại tính năng trên.



Thông báo khi thu gọn cửa sổ tin nhắn:



Mục tiêu của bài viết là demo được cách sử dụng Observer pattern, nên phần giao diện sẽ không được chú trọng. Bạn đọc có thể cải tiến thêm. Mã nguồn layout khá dài và chưa liên quan trực tiếp đến nội dung bài viết nên tác giả sẽ không mô tả tại đây. Các bạn có thể thao khảo mã nguồn đầy đủ ở gần cuối bài viết.

Giải pháp

Với tình huống trên, chúng ta có thể thấy rằng: hành vi cập nhật giao diện ở cả hai vị trí đều liên quan đến tình huống nhận tin nhắn mới từ máy chủ. Thoạt nhìn, tính năng có vẻ khá đơn giản. Nhưng nếu không phân tích kĩ một số góc cạnh khi thiết kế và viết mã, chúng ta có thể sẽ viết nên một khối mã nhập nhằng, khó đọc và khó bảo trì. Mẫu Observer phù hợp để giải quyết những tình huống như vậy.

Để triển khai mẫu này, chúng ta sẽ làm các thao tác như sau:

Bước 1 - Khai báo class Subject

```
class Subject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  unsubscribe(observer) {
    this.observers = this.observers.filter(subscriber => subscriber
!== observer);
  }

  notify(params) {
    this.observers.forEach(observer => observer.update(params));
  }
}
```

Giải thích mã: Class Subject ở trên có 3 phương thức:

- Phương thức subscribe được sử dụng để gán một hành vi, hoặc một thành phần observer vào.
- Phương thức unsubscribe được sử dụng để huỷ một hành vi, hoặc một thành phần observer khỏi danh sách.
- Phương thức notify có nhiệm vụ thông báo đến toàn bộ observer về trạng thái mới (hoặc sự kiện mới xảy ra).

Với ngôn ngữ lập trình JavaScript, chúng ta sử dụng kỹ thuật callback. Các observer sẽ lưu lại những function callback để có thể được thực hiện hành vi tương ứng khi sự kiện xảy ra.

Lưu ý thêm: Nếu bạn quen thuộc với những các ngôn ngữ lập trình hướng đối tượng, bạn có thể áp dụng cơ chế interface để xác định một đối tượng là Observer. Tuy nhiên, cơ chế này không tồn tại trong JavaScript nên đây là giải pháp gần gũi và dễ thực hiện nhất.

Bước 2 - Khai báo các Observer

Như đã giải thích ở bước 1, chúng ta sử dụng cơ chế callback. Vì vậy các function dưới đây sẽ được khai báo để đảm trách vai trò thực hiện những hành vi cụ thể khi sự kiện diễn ra. Với tình huống demo mà chúng ta đang thực hiện, mã nguồn có thể được viết như sau:

1. Các function tương tác với giao diện khi cập nhật dữ liệu, bao gồm tin nhắn mới và số tin nhắn chưa đọc:

```
function render_chat_message({ message }) {
  document.getElementById(chat_lines).appendChild(render_one_line(me
ssage));
}

function render_unread_count_header({ unread_state }) {
  document.getElementById(header_badge).innerText =
unread_state.get_string();
}

function render_unread_count_chatbox({ unread_state }) {
  document.getElementById(chat_box_badge).innerText =
unread_state.get_string();
}

function reset_unread_notification_ui() {
  document.getElementById(header_badge).innerText = "";
  document.getElementById(chat_box_badge).innerText = "";
}
```

2. Khai báo một class Observer để wrap các function này thành callback:

```
class Observer {
  constructor(callback) {
    this.callback = callback;
  }

  update(params) {
    this.callback(params);
  }
}

const chatbox_content_observer        = new
Observer(render_chat_message);
const chatbox_unread_notify_observer  = new
Observer(render_unread_count_chatbox);
const header_unread_notify_observer   = new
Observer(render_unread_count_header);
```

Bước 3 - Cập nhật danh sách Observer trong Subject

Chúng ta khởi tạo một đối tượng kiểu Subject, với tên là `message_receiver`. Sau đó, lần lượt gán ba observer được khai báo ở bước 2 vào `message_receiver`.

```
const message_receiver = new Subject();

message_receiver.subscribe(chatbox_content_observer);
message_receiver.subscribe(chatbox_unread_notify_observer);
message_receiver.subscribe(header_unread_notify_observer);
```

Mỗi khi sự kiện tin nhắn mới diễn ra, chúng ta sẽ gọi phương thức notify để thông báo đến ba observer đã gán.

```
const socket = io(chat_server_url);
socket.on(NEW_MESSAGE, (msg) => {
  unread_state.increase();
  message_receiver.notify(msg);
});
```

Chú thích: Trong demo này, tác giả sử dụng Socket.IO, một thư viện JavaScript giúp gửi và nhận thông điệp theo thời gian thực. Thư viện này sử dụng giao thức Web Socket để làm việc. Dòng mã `const socket = io(chat_server_url)` phục vụ tạo ra một kết nối đến Chat Server (mã nguồn sẽ được kèm trong link bên dưới).

Bước 4 - Bổ sung tình huống

Ở tình huống cửa sổ chat đang mở và được focus, các tin nhắn mới sẽ được hiển thị, người dùng đọc được nên ứng dụng không cần phải thông báo số tin nhắn chưa đọc nữa. Vì vậy, chúng ta sẽ xoá các observer liên quan đến thông báo số lượng tin chưa đọc khỏi subject. Đoạn mã này được viết lại như sau:

```
message_receiver.subscribe(chatbox_content_observer);

function hide_unread_notification() {
  message_receiver.unsubscribe(chatbox_unread_notify_observer);
  message_receiver.unsubscribe(header_unread_notify_observer);
}

function show_unread_notification() {
  message_receiver.subscribe(chatbox_unread_notify_observer);
  message_receiver.subscribe(header_unread_notify_observer);
}
```

Đồng thời, các hàm này được gọi ở sự kiện click vào button phóng to/thu gọn cửa sổ chat:

```
document.getElementById(chatbox_toggle).addEventListener('click', _
=> {
  chatbox_ui.toggle();
  unread_state.reset();
  if (chatbox_ui.is_maximize()) {
    hide_unread_notification();
    reset_unread_notification_ui();
  } else {
    show_unread_notification();
  }
});
```

Mã nguồn

Các bước ở trên chỉ là những đoạn mã rời rạc giúp người đọc hình dung được cách triển khai Observer pattern. Để ứng dụng demo có thể chạy được, mã nguồn cần được bổ sung hoàn chỉnh và cấu trúc tốt hơn.

Toàn bộ mã nguồn (bao gồm cả demo chat server và demo layout) được lưu trữ tại đây để các bạn có thể tham khảo và cải tiến: https://github.com/hoadh/tutorial-observer-pattern-javascript

Sau khi clone repository về, các bạn chạy lệnh sau để cài đặt các gói cần thiết cho demo server:

```
cd src/server
npm i
```

Khởi động server:

```
node server.js
```

Cuối cùng, mở file `index.html` (trong thư mục src/client) bằng trình duyệt để thực hiện tính năng được mô tả trong bài viết.

Kết bài

Hy vọng bài viết này có thể mang lại cho người đọc những kiến thức hữu ích về Observer pattern. Qua đó, có thể áp dụng được và nâng cao chất lượng công việc hằng ngày. Observer chỉ là một trong số rất nhiều design pattern mà chúng ta có thể áp dụng giúp nâng cao chất lượng mã nguồn. Tác giả gợi ý một số tài nguyên dưới đây để bạn có thể tìm hiểu bài bản hơn về chủ đề này:

- Sách "Head First Design Patterns" (Tác giả: Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra). Link: https://www.amazon.com/_/dp/0596007124 Sách có nhiều hình ảnh minh hoạ, giúp bạn dễ dàng hình dung được vấn đề mà pattern được đề cập giải quyết là gì, cũng như cách hoạt động của nó.
- Sách "Learning JavaScript Design Patterns" (Tác giả: Addy Osmani). Link: https://addyosmani.com/resources/essentialjsdesignpatterns/book/ Đây là quyển sách phù hợp cho các lập trình viên sử dụng ngôn ngữ JavaScript trong công việc hằng ngày.

Tác giả rất mong muốn nhận được phản hồi về nội dung bài viết, cũng như có được những thảo luận sâu hơn để nâng cấp kiến thức của bản thân. Chúc các bạn luôn tìm thấy được những niềm vui khi học và làm lập trình!

Bài 3. [Factory Method Pattern] Factory Method trong thực tiễn

Lời mở đầu

Giả sử như bạn cần mua một chiếc máy tính, nhưng bạn vẫn chưa quyết định được nên sử dụng máy tính của hãng nào. Ngoài kia bao la bạt ngàn những thương hiệu chất lượng cao như Apple, Lenovo, Asus, HP... Như vậy để có thể chọn được chiếc máy tính ưng ý, bạn sẽ có 2 cách sau:

- Đến showroom của từng hãng, rồi tham khảo từng máy. Bạn đi hết showroom của hãng này tới showroom của hãng khác để xem, rồi nhớ thông tin trong đầu để so sánh chiếc này hợp hơn, chiếc kia rẻ hơn ...
- Bạn đến một cửa hàng bày bán tất cả các loại laptop của tất cả các hãng, rồi bạn hỏi tư vấn là với số tiền này, bạn có thể chọn được laptop loại nào. Hay bạn đang muốn xem thử máy của hãng A, bạn nhờ tư vấn viên mang ra giúp bạn một chiếc. Rồi bạn băn khoăn một chiếc của hãng B, bạn lại nhờ tư vấn viên mang ra một chiếc khác.

Rõ ràng là cách thứ hai tiết kiệm thời gian và công sức cho bạn rất nhiều. Đây chính là cách mà mẫu thiết kế Factory hoạt động.

Mẫu thiết kế factory là gì?

Factory pattern là một mẫu thiết kế thuộc nhóm Khởi tạo. Pattern này sử dụng một interface hay một abstract class mà tất cả các lớp chúng ta cần khởi tạo đối tượng sẽ kế

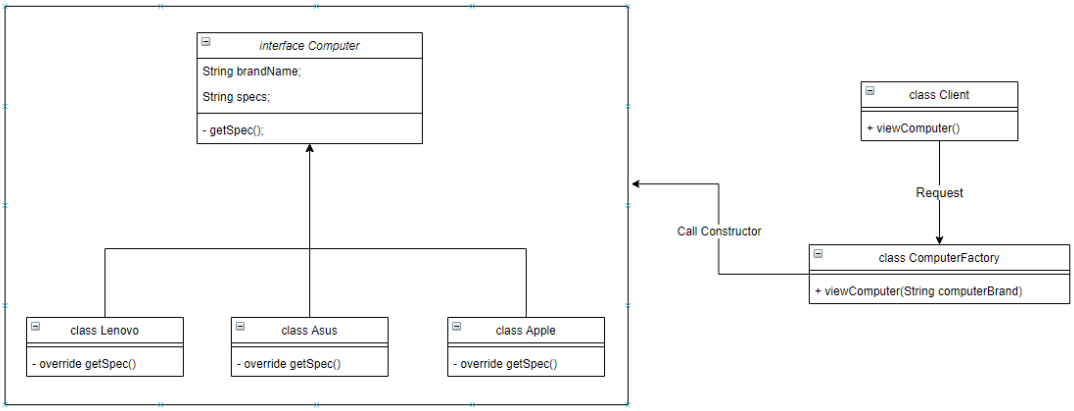
thừa. Factory sẽ định nghĩa việc khởi tạo đối tượng, nhưng đối tượng nào sẽ được tạo thì phụ thuộc vào các lớp con. Do vậy, pattern này còn được gọi với cái tên Virtual Constructor (phương thức khởi tạo ảo).

Factory pattern mang lại những tác dụng:

- Tạo ra một cách khởi tạo object mới
- Che giấu quá trình xử lý logic của phương thức khởi tạo
- Giảm sự phụ thuộc, dễ dàng mở rộng trong trường hợp chưa biết chắc số lượng đối tượng là đã đủ hay chưa. Trong trường hợp chúng ta có thêm lớp con kế thừa Factory, việc gọi đến virtual constructor vẫn không hề thay đổi.
- Giảm khả năng gây lỗi compile, trong trường hợp chúng ta cần tạo một đối tượng mà quên khai báo lớp, chúng ta cũng có thể xử lý lỗi trong Factory và khai báo lớp cho chúng sau.

Cấu trúc của Factory pattern

Xét theo ví dụ ở đầu bài viết, Factory pattern sẽ có cấu trúc dạng như sau:



Các lớp con Lenovo, Asus, HP đều override lại phương thức getSpec từ interface Computer. Phương thức `viewComputer()` của client sẽ gọi tới phương thức viewComputer của lớp ComputerFactory và truyền vào đó một tham số `computerBrand`, chính là tên của máy tính mà client muốn xem thêm, để tạo một đối tượng tương ứng. Đối tượng này sẽ được sử dụng để chạy phương thức view mà lớp con đã override lại.

Factory pattern trong Java

Trong ví dụ trên, nếu chúng ta làm theo cách 1, mã nguồn sẽ có dạng như sau:

```
```

```
public class Client {
      public void viewComputer() {
            public void ViewLenovo() {
                  Lenovo lenovoComputer = new Lenovo();
```

```
            System.out.println(lenovoComputer.getSpec());
                  }

                  public void ViewAsus() {
                        Asus asusComputer = new Asus();

            System.out.println(asusComputer.getSpec());
                  }

                  public void ViewHP() {
                        HP hpComputer = new HP();

            System.out.println(hpComputer.getSpec());
                  }
            }
      }
```

Như vậy, client sẽ phải gọi đến từng constructor cụ thể của từng lớp để tạo được đối tượng mong muốn.

Vậy khi áp dụng Factory pattern, mã nguồn sẽ thế nào?

Bước 1: Xây dựng lớp factory:

```
public class ComputerFactory {
      public void viewComputer(String computerBrand) {
            Computer computer;
            switch (computerBrand) {
                  case "Lenovo":
                        computer = new Lenovo();
                        break;
                  case "Asus":
                        computer = new Asus();
                        break;
                  case "HP":
                        computer = new HP();
                        break;
                  default:
                        System.out.println("Computer  brand  not
found");
                        break;
            }
            if (computer != null) {
                  System.out.println(computer.getSpec());
            }
      }
}
```

Bước 2: Từ lớp client, chúng ta gửi chỉ thị đến Factory:

```
public class Client {
      public void viewComputer() {
            ComputerFactory      computerFactory      =      new
ComputerFactory();
            computerFactory.viewComputer("Lenovo");
            computerFactory.viewComputer("HP");
            computerFactory.viewComputer("Asus");
            computerFactory.viewComputer("Dell");
      }
}
```

Như vậy, việc khởi tạo các đối tượng thuộc từng lớp kế thừa interface Computer đã bị ẩn đi đối với client. Mặt khác, khi chúng ta thêm mới lớp kế thừa Computer, chỉ có virtual constructor trong Factory cần cập nhật, thay vì phải thay đổi trên cả lớp Client.

Bài 4. [Decorator Pattern] Design Patterns trong PHP: Decorator (với Laravel)

Design patterns vô cùng quan trọng với mỗi lập trình viên. Nó giải quyết các vấn đề phổ biến trong mỗi dự án mà bạn xây dựng.

Định nghĩa về Decorator Pattern

Nó cho phép bạn thêm các hành vi vào một đối tượng mà không ảnh hưởng tới các đối tượng khác trong cùng một lớp. Giờ chúng ta sẽ xem điều này nghĩa là gì nhé

Theo Wiki

Decorator pattern là một mẫu thiết kế cho phép thêm chức năng vào một đối tượng riêng lẻ mà không ảnh hưởng đến hành vi của các đối tượng khác trong cùng một lớp

Đặt Vấn đề

Giả sử chúng ta có một model là Post

```
class Post extends Model {
    public function scopePublished($query) {
        return $query->where('published_at', '<=', 'NOW()');
    }
}
```

và trong PostsController chúng ta có method index như bên dưới

```
class PostsController extends Controller {
    public function index() {
        $posts = Post::published()->get();
        return $posts;
    }
}
```

Để lưu post vào cache và tránh truy vấn cơ sở dữ liệu mỗi khi cần liệt kê những post chúng ta có thể làm như sau

```
class PostsController extends Controller {
    public function index() {
        $minutes = 1400; # 1 day
        $posts = Cache::remember('posts', $minutes, function() {
            return Post::published()->get();
        });
        return $posts;
    }
}
```

Giờ chúng ta đặt thời gian cache cho post là 1 ngày. Nhưng nhìn vào đoạn mã này xem, controller biết quá nhiều thứ không cần thiết: nó biết bạn cache trong bao lâu và còn tự tạo ra đối tượng cache. Ngoài ra, thử hình dung nếu bạn triển khai cấu trúc tương tự cho các model khác như Tags (thẻ), Categories (các danh mục), Archives (tài nguyên lưu trữ khác) trong `HomePageController`. Quá nhiều thứ để đọc, quá khó khăn để bảo trì.

Repository

Trong hầu hết các trường hợp, Repository pattern được kết nối với Decorator pattern và bạn sẽ thấy như thế nào

Đầu tiên, giả sử phân tách bằng cách dùng Repository pattern, tạo `app/Repositories/Posts/PostsRepositoryInterface.php` với nội dung :

```
namespace App\Repositories\Posts;

interface PostsRepositoryInterface {
    public function get();

    public function find(int $id);
}
```

Tạo 1 `PostsRepository` trong cùng đường dẫn với nội dung sau

```
namespace App\Repositories\Posts;

use App\Post;

class PostsRepository implements PostsRepositoryInterface {
    protected $model;

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

```
    public function get() {
        return $this->model->published()->get();
    }

    public function find(int $id) {
        return $this->model->published()->find($id);
    }
}
```

PostsController và áp dụng những thay đổi như bên dưới

```
namespace App\Http\Controllers

use App\Repositories\Posts\PostsRepositoryInterface;
use Illuminate\Http\Request;

class PostsController extends Controller {
    public function index(PostsRepositoryInterface $postsRepo) {
        return $postsRepo->get();
    }
}
```

Controller trở nên “heathy" hơn và biết rõ ràng từng chi tiết để hoàn thành nhiệm vụ của nó.

Ở đây, chúng ta phụ thuộc vào Laravel's IOC để inject đối tượng cụ thể của interface Post để kết nối các Post

Tất cả những gì chúng ta cần làm là chỉ dẫn cho Laravel's IOC sẽ tạo lớp nào khi sử dụng giao diện

Trong `app/Providers/AppServiceProvider.php` thêm cách thức ràng buộc

```
namespace App\Providers;

use App\Repositories\Posts\PostsRepositoryInterface;
use App\Repositories\Posts\PostsCacheRepository;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider {
    public function register() {
        $this->app->bind(PostsRepositoryInterface::class,
PostsCacheRepository::class);
    }
}
```

Bây giờ, bất cứ khi nào chúng ta inject PostsRepositoryInterface laravel sẽ tạo 1 PostsRepository và thay đổi như thế.

Triển khai cache qua Decorator

Chúng ta thấy ngay từ đầu rằng, Decorator pattern cho phép chúng năng được thêm vào 1 đối tượng riêng lẻ mà không ảnh hưởng đến đối tượng khác cùng lớp

Ở đây cache là hành vi và đối tượng/lớp, là PostsRepository

Giả sử tạo PostsCacheRepository trong cùng một
`app/Repositories/Posts/PostsCacheRepository.php` với nội dung sau:

```
namespace App\Repositories\Posts;

use App\Post;
use Illuminate\Cache\CacheManager;

class PostsCacheRepository implements PostsRepositoryInterface {
    protected $repo;

    protected $cache;

    const TTL = 1400; # 1 day

    public function __construct(CacheManager $cache, PostsRepository
$repo) {
        $this->repo = $repo;
        $this->cache = $cache;
    }

    public function get() {
        return $this->cache->remember('posts', self::TTL, function()
{
            return $this->repo->get();
        });
    }


    public function find(int $id) {
        return $this->cache->remember('posts.'.$id, self::TTL,
function() {
            return $this->repo->find($id);
        });
    }
}
```

Trong lớp, chúng ta chấp nhận bộ đệm và PostsRepository rồi sử dụng lớp (Decorator) để thêm hành vi PostsReposiory

Chúng ta có thể sử dụng VD để gửi các yêu cầu HTTP từ một số service và sau đó chúng ta quay lại model nếu thất bại. Tôi tin rằng bạn đã nhận được lợi ích từ mô hình và biết cách dễ dàng để thêm các hành vi.

Điều cuối cùng là sửa đổi ràng buộc interface AppServiceProvider để tạo phiên bản PostCacheRepository thay vì PostsRepository

```
namespace App\Providers;

use App\Repositories\Posts\PostsRepositoryInterface;
use App\Repositories\Posts\PostCacheRepository;

use Illuminate\Support\ServiceProviders;

class AppServiceProvider extends ServiceProvider {
```

```
    public function register() {
        $this->app->bind(PostsRepositoryInterface::class,
PostsCacheRepository::class);
    }
}
```

Kiểm tra lại các tập tin của bạn lúc này sẽ thấy nó rất dễ đọc và bảo trì. Ngoài ra, nó có thể kiểm tra được và nếu tại một thời điểm nào đó, bạn quyết định di chuyển bỏ lớp bộ đệm bạn sẽ chỉ thay đổi ràng buộc trong AppServiceProvider và không có gì cần phải thay đổi thêm.

Kết luận

- Chúng ta đã học cách cache model với Decorator pattern
- Chúng ta đã chỉ ra cách Repository pattern được kết nối với Decorator Pattern
- Cách DependecyInjection và Laravel IOC làm cho cuộc sống của chúng ta dễ dàng
- Cách tràn đầy sức mạnh với các thành phần laravel

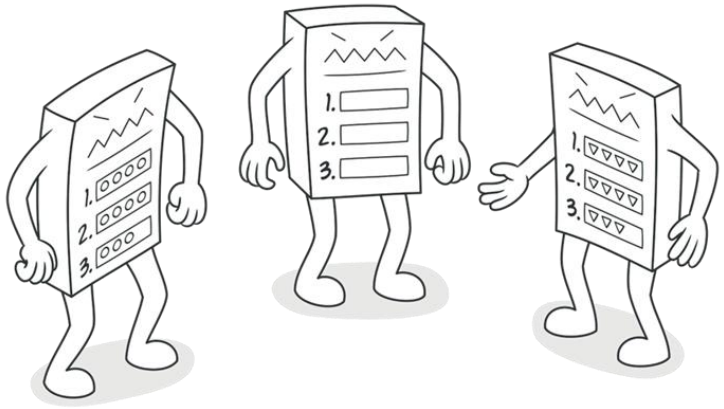
Hy vọng bạn thưởng thức bài viết và thấy được sức mạnh của các design patterns và cách nó giúp dự án của bạn dễ dàng bảo trì và quản lý.

Bài 5. [Template Method Pattern] Template Method Pattern

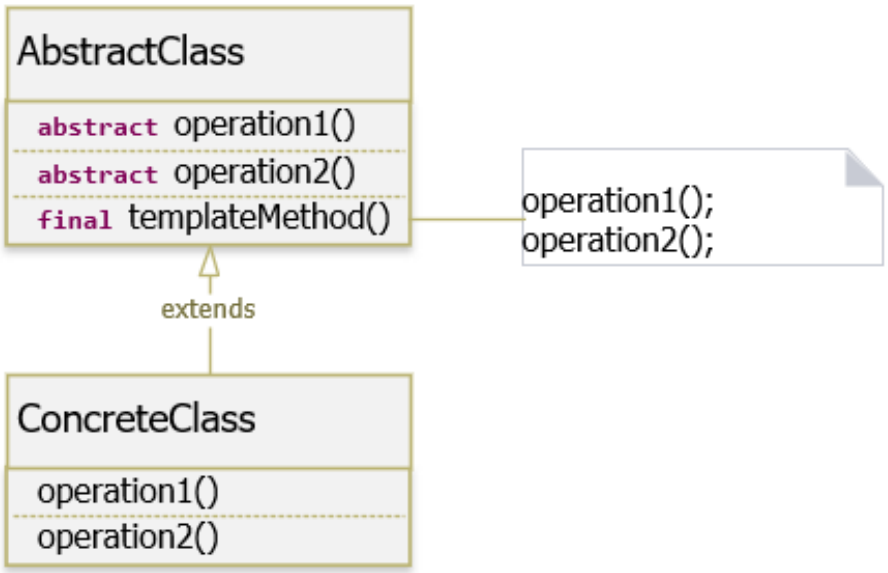
Tổng quan về Template Method

Là một pattern thuộc nhóm behavior - định nghĩa một bộ khung của một thuật toán trong một chức năng, chuyển giao việc thực hiện nó cho các lớp con. Template Method Pattern cho phép lớp con định nghĩa lại cách thực hiện của một thuật toán, mà không phải thay đổi cấu trúc thuật toán.

Điều này có nghĩa là Template method giúp cho chúng ta tạo nên một bộ khung (template) cho một vấn đề đang cần giải quyết. Trong đó các đối tượng cụ thể sẽ có cùng các bước thực hiện, nhưng trong mỗi bước thực hiện đó có thể khác nhau. Điều này sẽ tạo nên một cách thức truy cập giống nhau nhưng có hành động và kết quả khác nhau.



Các thành phần của một Template Method



Trong đó thì:

- **AbstractClass(Protocol)** : Định nghĩa các phương thức trừu tượng cho từng bước có thể được điều chỉnh bởi các lớp con. Cài đặt một phương thức duy nhất điều khiển thuật toán và gọi các bước riêng lẻ đã được cài đặt ở các lớp con.
- **ConcreteClass** : là một thuật toán cụ thể, cài đặt các phương thức của AbstractClass. Các thuật toán này ghi đè lên các phương thức trừu tượng để cung cấp các triển khai thực sự. Nó không thể ghi đè phương thức duy nhất đã được cài đặt ở AbstractClass (templateMethod).

Việc sử dụng template method có nhiều lợi ích trong việc lập trình

- Tái sử dụng code (reuse), tránh trùng lặp code (duplicate): đưa những phần trùng lặp vào lớp cha (abstract class).
- Cho phép người dùng override chỉ một số phần nhất định của thuật toán lớn, làm cho chúng ít bị ảnh hưởng hơn bởi những thay đổi xảy ra với các phần khác của thuật toán.

Vơi những lợi ích như thế thì template method rất hay được sử dụng khi:

- Khi có một thuật toán với nhiều bước và mong muốn cho phép tùy chỉnh chúng trong lớp con.
- Mong muốn chỉ có một triển khai phương thức trừu tượng duy nhất của một thuật toán.
- Mong muốn hành vi chung giữa các lớp con nên được đặt ở một lớp chung.
- Các lớp cha có thể gọi các hành vi trong các lớp con của chúng một cách thống nhất (step by step).

Cách sử dụng Template Method Pattern

Chúng ta sẽ đi vào một ví dụ cụ thể để hiểu hơn về pattern này.

Cấu trúc của một website thông thường gồm các phần header, footer, navigation (menu), body. Riêng phần body thường xuyên thay đổi, sẽ hiển thị riêng theo từng trang. Những phần khác ít khi thay đổi, trừ khi có yêu cầu đặt biệt. Thay vì phải viết tất cả các phần ở mỗi trang, chúng ta có thể gom chúng lại và đặt trong một template để tái sử dụng mà không duplicate code ở nhiều nơi.

Đầu tiên chúng ta sẽ đi vào khởi tạo một **AbstractProtocol** tên `PageTemplate`

```
protocol PageTemplate {
    func showPage()

    func showHeader()

    func showNavigation()

    func showFooter()

    func showBody()
}
```

viết extension triển khai các phương thức chung cho các lớp con

```
extension PageTemplate {

    func showPage() {
        showHeader()
        showNavigation()
        showBody()
        showFooter()
    }
```

```
    func showHeader() {
        print("<header />\n")
    }

    func showNavigation() {
        print("<nav />\n")
    }

    func showFooter() {
        print("<footer />\n")
    }
}
```

Tiến hành khởi tạo các `concreteClass adopt` **PageTemplate** và khởi tạo **Client** để thử template method.

```
class HomePage: PageTemplate {

    func showBody() {
        print("Content of home page page\n")
    }
}

class DetailPage: PageTemplate {

    func showBody() {
        print("Content of detail\n")
    }
}

class ContactPage: PageTemplate {

    func showNavigation() {
        // Just do nothing
        // Because we don't want to show navigation bar on contact
page
    }

    func showBody() {
        print("Content of contact page\n")
    }
}

class Client {
    static func clientCode(use object: PageTemplate) {
        object.showPage()
    }
}
```

Tiến hành test kết quả

```
    Client.clientCode(use: HomePage())
    Client.clientCode(use: DetailPage())
    Client.clientCode(use: ContactPage())
kết quả nhận được như sau:
<header />

<nav />
```

```
Content of home page page

<footer />

<header />

<nav />

Content of detail

<footer />

<header />

Content of contact page

<footer />
```

Thông qua ví dụ trên chúng ta cũng hiểu cách thức hoạt động của template method như thế nào. Giờ chúng ta đi đến ứng dụng của pattern này được áp dụng như thế nào trong IOS.

Ứng dụng của Template Method trong lập trình IOS

Template Method thể hiện rõ ràng nhất trong **Overridden Framework Methods** và **Document Architecture in OS X** Cụ thể là:

- Template Method là một thiết kế cơ bản của Cocoa và của các object-oriented frameworks nói chung. cho phép các thành phần tùy chỉnh của chương trình móc vào một thuật toán, nhưng các thành phần framework xác định thời điểm và cách chúng thể hiện.
- Kiến trúc tài liệu được xác định bởi AppKit Framewoks là một thể hiện riêng biệt và quan trọng của thiết kế chung của các phương thức trong framework được ghi đè như là một sự điều chỉnh của template method. Các ứng dụng Cocoa có thể tạo và quản lý nhiều tài liệu, mỗi tài liệu trong cửa sổ riêng, hầu như luôn dựa trên kiến trúc tài liệu.

Bài 6. [Repository Pattern] Repository Pattern trong Laravel

Mở đầu

Design patterns là các giải pháp đã được tối ưu hóa, được tái sử dụng cho các vấn đề lập trình mà chúng ta gặp phải hàng ngày. Nó là một khuôn mẫu đã được suy nghĩ, giải quyết trong tình huống cụ thể rồi.

Repository Pattern là một mẫu thiết kế trong design pattern.

Repository Pattern là gì?

Repository Pattern là lớp trung gian giữa tầng Business Logic và Data Access, giúp cho việc truy cập dữ liệu chặt chẽ và bảo mật hơn. **Repository** đóng vai trò là một lớp kết nối giữa tầng Business và Model của ứng dụng. Hiểu đơn giản thì khi muốn truy xuất dữ liệu từ database, thay vì viết code xử lý trong controller thì ta tạo ra 1 thư mục là **Repository** rồi viết code xử lý vào đây. Sau đó chúng ta chỉ việc inject vào thông qua __construct.

Những lý do ta nên sử dụng mẫu Repository Pattern:

- Code dễ dàng maintain.
- Tăng tính bảo mật và rõ ràng cho code.
- Lỗi ít hơn.
- tránh việc lặp code.

Repository trong laravel

Để hiểu hơn về Repository pattern chúng ta sẽ xây dựng 1 controller trong laravel. Chúng ta sẽ có bảng post chứa thông tin: `id, title, content.` Với model như sau:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class post extends Model
{
    protected $table = "posts";

     protected $fillable = [
        'title', 'content','view'
    ];
}
```

Chúng ta sẽ tạo ra 1 PostController để có thể xử lý tác vụ cơ bản curd. Thông thường khi chưa áp dụng repository thì code của chúng ta sẽ như thế này:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Post;

class PostController extends Controller
{

    public function index()
    {
        $posts = Post::all();

        return view('home.posts', compact('posts'));
    }

   public function show($id)
   {
        $post = Post::findOrFail($id);

        return view('home.post', compact('post'));
   }

   public function store(Request $request)
   {
        $data = $request->all();

        //... Validation here

        $post = Post::createOrFail($data);

        return view('home.post', compact('post'));
   }

   public function update(Request $request, $id)
   {
        $data = $request->all();

        $post = Post::findOrFail($id);
        $post->update($data);


        return view('home.post', compact('post'));
   }

   public function destroy($id)
   {
        $post = Post::findOrFail($id);
        $post->delete();


        return view('home.post', compact('post'));
   }
}
```

Có vẻ nhìn cũng rất ổn đấy chứ. Tuy nhiên, trong project có thêm thằng `CategoryController` và bạn nhận ra rằng chúng ta phải viết lại code y hệt thằng `PostController` chỉ thay đổi mỗi models xử lý. Và bạn mới nghĩ rằng tại sao không thể dùng chung code xử lý, Từ đây thằng repository ra đời (tất nhiên đây chỉ là 1 ứng dụng của repository mình lấy ra để các bạn dễ tiếp cận).

Xây dựng Repository cơ bản

Ý tưởng: tạo ra 1 class chung đặt tên là `EloquentRepository` để xử lý tác vụ crud chung. Sau đó, chúng ta sẽ tạo ra các class con extend class `EloquentRepository` và truyền vào models tương ứng để xử lý.

Trước Tiên các bạn tạo ra một thư mục Repositories trong thư mục app để quản lý các Repository. Trong thư mục Repositories các bạn tạo ra 1 class EloquentRepository với nội dung như sau:

```
<?php

namespace App\Repositories;

abstract class EloquentRepository
{
    /**
     * @var \Illuminate\Database\Eloquent\Model
     */
    protected $_model;

    /**
     * EloquentRepository constructor.
     */
    public function __construct()
    {
        $this->setModel();
    }

    /**
     * get model
     * @return string
     */
    abstract public function getModel();

    /**
     * Set model
     */
    public function setModel()
    {
        $this->_model = app()->make(
            $this->getModel()
        );
    }

    /**
     * Get All
     * @return \Illuminate\Database\Eloquent\Collection|static[]
     */
    public function getAll()
```

```
{

    return $this->_model->all();
}

/**
 * Get one
 * @param $id
 * @return mixed
 */
public function find($id)
{
    $result = $this->_model->find($id);

    return $result;
}

/**
 * Create
 * @param array $attributes
 * @return mixed
 */
public function create(array $attributes)
{

    return $this->_model->create($attributes);
}

/**
 * Update
 * @param $id
 * @param array $attributes
 * @return bool|mixed
 */
public function update($id, array $attributes)
{
    $result = $this->find($id);
    if ($result) {
        $result->update($attributes);
        return $result;
    }

    return false;
}

/**
 * Delete
 *
 * @param $id
 * @return bool
 */
public function delete($id)
{
    $result = $this->find($id);
    if ($result) {
        $result->delete();

        return true;
    }
```

```
        return false;
    }

}
```

Ở trong thư mục Repositories chúng ta tạo 1 class post.

```
<?php
namespace App\Repositories\Post;

use App\Repositories\EloquentRepository;

class PostEloquentRepository extends EloquentRepository
{

    /**
     * get model
     * @return string
     */
    public function getModel()
    {
        return \App\Models\Post::class;
    }
}
```

Trong class PostEloquentRepository chúng ta sẽ khi đè hàm `getModel()` để thay vào models tương ứng. Ở đây là models Post. Tất nhiên là bạn cũng có thể viết thêm các method xử lý riêng của thằng post vào đây.

Quay trở lại thằng PostController. Việc cần làm là chúng ta sẽ inject thằng PostRepository vào `__construct()` của PostController. Rồi sử dụng nó thông qua biến `$postRepository` chúng ta tạo ra.

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Repositories\Post\PostRepository;

class PostController extends Controller
{
    /**
     * @var PostRepositoryInterface|\App\Repositories\Repository
     */
    protected $postRepository;

    public function __construct(PostRepository $postRepository)
    {
        $this->postRepository = $postRepository;
    }
    /**
     * Show all post
     *
     * @return \Illuminate\Http\Response
     */
```

```
public function index()
{
    $posts = $this->postRepository->getAll();

    return view('home.posts', compact('posts'));
}

/**
 * Show single post
 *
 * @param $id int Post ID
 * @return \Illuminate\Http\Response
 */
public function show($id)
{
    $post = $this->postRepository->find($id);

    return view('home.post', compact('post'));
}

/**
 * Create single post
 *
 * @param $request \Illuminate\Http\Request
 * @return \Illuminate\Http\Response
 */
public function store(Request $request)
{
    $data = $request->all();

    //... Validation here

    $post = $this->postRepository->create($data);

    return view('home.post', compact('post'));
}

/**
 * Update single post
 *
 * @param $request \Illuminate\Http\Request
 * @param $id int Post ID
 * @return \Illuminate\Http\Response
 */
public function update(Request $request, $id)
{
    $data = $request->all();

    //... Validation here

    $post = $this->postRepository->update($id, $data);

    return view('home.post', compact('post'));
}

/**
 * Delete single post
 *
 * @param $id int Post ID
 * @return \Illuminate\Http\Response
```

```
     */
    public function destroy($id)
    {
        $this->postRepository->delete($id);
        return view('home.post');
    }
}
```

Bạn có thể làm tương tự với thằng CategoryController. Tạo repository tương ứng và gọi sang. Thật đơn giản phải không nào.

Repository thông qua interface

Ở trên chúng ta đã áp dụng repository cơ bản rồi. Tuy nhiên, khi join vào 1 dự án lớn có nhiều người làm việc. Việc áp dụng interface để đảm bảo code chuẩn theo thiết kế cũng như dễ dàng thay đổi, maintain dễ dàng là điều cần thiết. Quay trở lại `class EloquentRepository`, Việc cần làm trước khi xây dựng `class EloquentRepository` đó là tạo ra 1 bản thiết kế. Đó là 1 interface mình đặt tên là `RepositoryInterface`

```
<?php

namespace App\Repositories;

interface RepositoryInterface
{
    /**
     * Get all
     * @return mixed
     */
    public function getAll();

    /**
     * Get one
     * @param $id
     * @return mixed
     */
    public function find($id);

    /**
     * Create
     * @param array $attributes
     * @return mixed
     */
    public function create(array $attributes);

    /**
     * Update
     * @param $id
     * @param array $attributes
     * @return mixed
     */
    public function update($id, array $attributes);

    /**
     * Delete
     * @param $id
```

```
     * @return mixed
     */
    public function delete($id);
}
```

Đây là khung của EloquentRepository. Trong class EloquentRepository chúng ta sẽ implements RepositoryInterface. Chỉnh lại code trong class EloquentRepository 1 tí nào.

```
<?php

namespace App\Repositories;

use App\Repositories\RepositoryInterface;

abstract class EloquentRepository implements RepositoryInterface
{
    /**
     * @var \Illuminate\Database\Eloquent\Model
     */
    protected $_model;

    /**
     * EloquentRepository constructor.
     */
    public function __construct()
    {
        $this->setModel();
    }

    /**
     * get model
     * @return string
     */
    abstract public function getModel();

    /**
     * Set model
     */
    public function setModel()
    {
        $this->_model = app()->make(
            $this->getModel()
        );
    }

    /**
     * Get All
     * @return \Illuminate\Database\Eloquent\Collection|static[]
     */
    public function getAll()
    {

        return $this->_model->all();
    }


    /**
     * Get one
```

```
     * @param $id
     * @return mixed
     */
    public function find($id)
    {
        $result = $this->_model->find($id);

        return $result;
    }

    /**
     * Create
     * @param array $attributes
     * @return mixed
     */
    public function create(array $attributes)
    {

        return $this->_model->create($attributes);
    }

    /**
     * Update
     * @param $id
     * @param array $attributes
     * @return bool|mixed
     */
    public function update($id, array $attributes)
    {
        $result = $this->find($id);
        if ($result) {
            $result->update($attributes);
            return $result;
        }


        return false;
    }

    /**
     * Delete
     *
     * @param $id
     * @return bool
     */
    public function delete($id)
    {
        $result = $this->find($id);
        if ($result) {
            $result->delete();


            return true;
        }

        return false;
    }

}
```

Chúng ta sẽ thiết kế thêm chức năng riêng cho class PostEloquentRepository. Ví dụ ở đây mình muốn thêm 1 method lấy 5 post nhiều lượt xem nhất trong vòng 1 tháng chẳng hạn. Chúng ta sẽ tạo ra 1 bản thiết kế cho PostEloquentRepository. Mình đặt tên là

```
<?php
namespace App\Repositories\Post;

interface PostRepositoryInterface
{
    /**
     * Get 5 posts hot in a month the last
     * @return mixed
     */
    public function getPostHost();
}
```

Ở bên PostEloquentRepository chúng ta sẽ

```
implements PostRepositoryInterface
<?php
namespace App\Repositories\Post;

use App\Repositories\EloquentRepository;
use Illuminate\Support\Carbon;

class PostEloquentRepository extends EloquentRepository implements
PostRepositoryInterface
{
    /**
     * get model
     * @return string
     */
    public function getModel()
    {
        return \App\Models\Post::class;
    }

    /**
     * Get 5 posts hot in a month the last
     * @return mixed
     */
    public function getPostHost()
    {
        return $this->_model::where('created_at', '>=',
Carbon::now()->subMonth())->orderBy('view', 'desc')->take(5)->get();
    };

}
```

Như đã nói ở trên, chúng ta sẽ làm việc thông qua interface. Mà các bạn đã biết chúng ta không thể sử dụng 1 đối tượng là interface. Do đó, chúng ta phải đăng kí với laravel để nó có thể hiểu rằng interface bạn sử dụng là của class cụ thể nào. Các bán sẽ mở tập tin `app/Providers/AppServiceProvider.php` và thêm vào `method register()` như sau:

```
public function register()
{
    $this->app->singleton(
        \App\Repositories\Post\PostRepositoryInterface::class,
        \App\Repositories\Post\PostEloquentRepository::class
    );
}
```

Bây giờ thì ta có thể sử dụng PostRepositoryInterface khi này nó sẽ hiểu là bạn sử dụng PostEloquentRepository. Bây giờ trong PostController thay vì inject PostEloquentRepository thì bạn sẽ inject PostRepositoryInterface.

```
  public function __construct(PostRepositoryInterface
$postRepository)
    {
        $this->postRepository = $postRepository;
    }
```

Kết luận

Việc áp dụng Repository design pattern cũng như các mẫu design pattern sẽ giúp các bạn tiết kiệm thời gian đồng thời cũng tăng hiệu suất, chất lượng code. Các mẫu design pattern không ràng buộc bởi ngôn ngữ lập trình dó đó nó có thể áp dụng ở mọi ngôn ngữ. Trên đây là bài tìm hiểu về Repository design pattern của minh. Cảm ơn các bạn đã theo dõi.

Bài 7. [Proxy Pattern] Proxy Pattern

Proxy cung cấp một class ảo đứng trước class thực sự mà chúng ta muốn làm việc để xử lí, tăng thêm tính bảo mật và cải thiện performance cho hệ thống khi xử lí dữ liệu liên quan đến class mà chúng ta làm việc

Khi nào nên sử dụng Proxy Pattern?

- Khi muốn thêm một số bước bảo mật, quản lí sự truy cập/truy xuất dữ liệu ra vào đối tượng
- Linh hoạt cách truy xuất/truy cập dữ liệu (Lazy Loading,...)

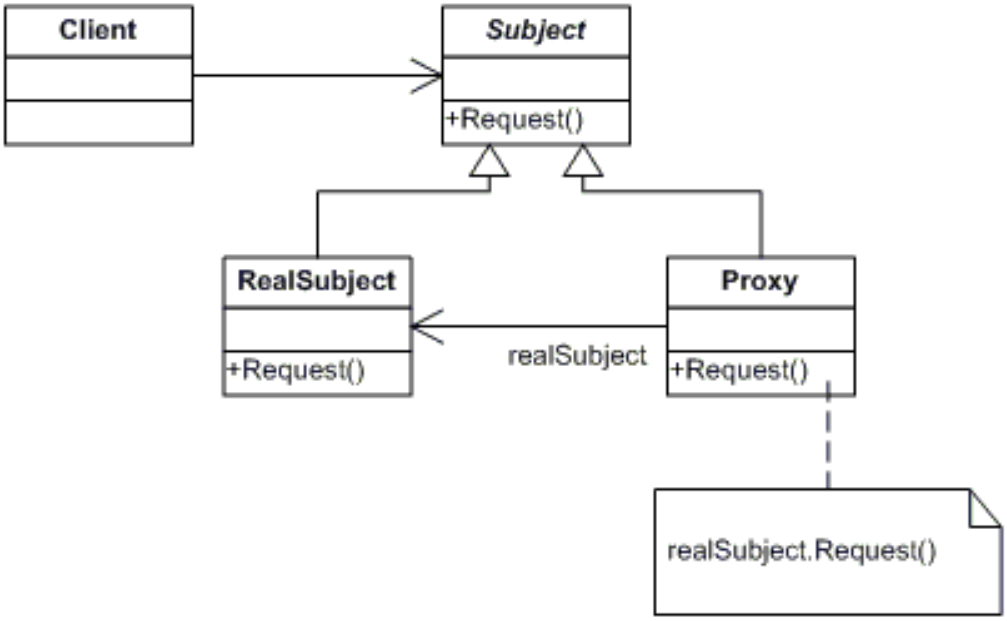
Các loại ứng dụng của Proxy Pattern

- **Remote Proxy**: Cung cấp một đại diện cho object nhưng nằm trong 1 địa chỉ khác
- **Virtual Proxy**: Áp dụng cho các dữ liệu tốn nhiều chi phí khởi tạo, loại proxy này cung cấp các phương pháp lấy dữ liệu linh hoạt, chỉ thực sự khởi tạo/truy

xuất dữ liệu khi được người dùng yêu cầu thay vì khởi tạo dữ liệu từ đầu. Một số ứng dụng như Lazy Loading,...

- **Protective Proxy**: Kiểm soát quyền truy cập vào object gốc. Loại proxy này tạo thêm các hàng rào kiểm soát nhằm đảo bảo bảo mật cho object
- **Smart Proxy**: Bổ sung thêm các hành động, phần mở rộng khi truy cập/truy xuất dữ liệu trong đối tượng

Cấu trúc



Các đối tượng tham gia vào Proxy Pattern:

- **Subject**: Interface giữ vai trò tạo xương sống cho **RealSubject** (đối tượng thực sự) và **Proxy**
- **RealSubject**: Đối tượng thực sự mà người dùng làm việc
- **Proxy**: Triển khai xương sống từ **Subject** và là class đứng trước (đại diện) **RealSubject** nhằm thực hiện các tiền/hậu xử lí, bảo mật cho **RealSubject**. Proxy duy trì một tham chiếu đến đối tượng của **RealSubject** nhằm truy cập/truy xuất dữ liệu

Ví dụ

- Tạo interface **IItem** và class **Item** triển khai interface đó
- **Item** là một class không tốn nhiều chi phí khởi tạo đối tượng nhưng giá trị **content** bên trong Item lại tốn nhiều chi phí khởi tạo (tưởng tượng rằng mỗi khi muốn lấy giá trị content thì phải request lên server để lấy giá trị, điều đó gây nên chi phí, thời gian kết nối đến server, có thể xem đối tượng IItem là `index/id` của giá trị content)

```
interface IItem {
```

```
    delaytoResponseContent(): Promise<number>;
    content: number;
}

class Item implements IItem {
    // Giá trị của content nằm trên server
    protected _content: number = Math.round(Math.random() * 10);
    get content() {
        return this._content;
    }
    set content(n: number) {
        this._content = n;
    }

    // Trả về giá trị của content mất nhiều chi phí
    async delaytoResponseContent() : Promise<number> {
        await this.sleep(1000);
        return this.content;
    }

    // Tạo chi phí kết nối
    sleep(ms: number) {
        return new Promise((resolve) =>
            setTimeout(resolve, ms)
        )
    }
}
```

- Tạo interface **Subject** và class **RealSubject**, **Proxy**

```
interface Subject {
    data: Array<IItem>;
    Request(index: number): any;
}

class RealSubject {
    constructor(private _data: Array<IItem>) { }
    get data() : Array<IItem> {
        return this._data;
    }

    Request(index: number) : Promise<number> {
        return this.data[index].delaytoResponseContent();
    }
}

class SubjectProxy implements Subject {
    // Khai báo cho đúng cấu trúc nhưng không sử dụng
    data!: Array<IItem>;
    // Duy trì một kết nối đến RealSubject để xử lí dữ liệu
    private realSubject!: RealSubject;


    // Kết nối đến host thông qua RealSubject
    constructor(data: Array<IItem>) {
        this.realSubject = new RealSubject(data);
    }

    // - Ghi đè lại phương thức Request
```

```
    // - Thay vì lấy về giá trị content mất nhiều chi phí
    //   thì ta chỉ lấy về index/id của giá trị đó
    // - Khi nào người dùng cần giá trị content thì mới
    //   sử dụng phương thức delaytoResponseContent để
    //   lấy giá trị content về
    Request(index: number) : IItem {
        return this.realSubject.data[index];
    }
}
```

- Tạo một host giả có chứa và **index Item**

```
// Tạo một host giả có chứa content và index Item
let data : Array<IItem> = [
    new Item(),
    new Item(),
    new Item(),
    new Item(),
    new Item(),
    new Item(),
];
```

- Khi không sử dụng Proxy. Giả sử đường truyền yếu chỉ có thể tại được 1 content 1 lần và ta phải tải hết dữ liệu về để hiển thị trong khi người dùng muốn xem dữ liệu thứ 4 trước tiên

```
let realSubject = new RealSubject(data);
(async () => {
    let wanted : number = 3;
    for(let i = 0; i < 5; i++) {
        await realSubject.Request(i).then(
            res => {
                if(i!=wanted)
                    console.log("Result: "+res)
                else
                    console.log("Wanted result: "+res)
            }
        );
    }
})();
```

- Kết quả khi không dùng Proxy

```
Result: 4
Result: 5
Result: 5
Wanted result: 4
Result: 9
```

- Khi sử dụng Proxy và ứng dụng Lazy Loading (Virtual Proxy). Dữ liệu cần thiết sẽ được tải trước, các dữ liệu còn lại sẽ được tải sau

```
let proxy = new SubjectProxy(data);
(async () => {
    let wanted : number = 3;
    let results : Array<IItem> = new Array<IItem>();
    for(let i = 0; i < 5; i++) {
        results.push(proxy.Request(i));
```

```
        if(i==wanted)
            results[i].delaytoResponseContent()
                .then(res => console.log("Wanted result via proxy:
"+res));
    }
    for(let i = 0; i < 5; i++)
        if(i!=wanted)
            results[i].delaytoResponseContent()
                .then(res => console.log("Result via proxy: "+res))
})();
```

- Kết quả khi sử dụng Proxy

```
Wanted result via proxy: 1
Result via proxy: 0
Result via proxy: 7
Result via proxy: 2
Result via proxy: 1
```

- Ngoài ra ta có thể cài đặt thêm các tính năng bảo mật, kiểm tra lỗi cho proxy để tăng tính bảo mật hơn cho hệ thống (Smart Proxy)

So sánh với pattern cùng loại (Structural Pattern)

Ta có thể thấy Proxy Pattern khá giống với **Adapter Patern** và **Decorator Pattern** nên ta cần phân biệt chúng:

- Khác với **Adapter Pattern**: Thông thường mẫu Adapter cung cấp một giao diện khác với đối tượng gốc, còn Proxy cung cấp cùng một giao diện giống như đối tượng gốc
- Khác với **Decorator Pattern**: Có thể cài đặt tương tự như Proxy, nhưng Decorator được dùng cho mục đích khác. Decorator bổ sung thêm nhiều nhiệm vụ cho một đối tượng nhưng ngược lại Proxy điều khiển truy cập đến một đối tượng. Proxy tuỳ biến theo nhiều cấp khác nhau mà có thể được cài đặt giống như một Decorator:
    - **Protection Proxy, Smart Proxy**: Có thể được cài đặt như một Decorator
    - **Remote Proxy**: Sẽ không tham chiếu trực tiếp đến đối tượng thực sự tham chiếu gián tiếp, giống như ID của host và địa chỉ trên host vậy
    - **Virtural Proxy**: Tham chiếu gián tiếp chẳng hạn như tên file, index và sẽ tham chiếu trực tiếp khi cần thiết

Bài 8. [Factory Pattern] Factory Pattern trong Laravel

Tản mạn về định nghĩa

In object-oriented programming (OOP), a factory is an object for creating other objects – formally a factory is a function or method that returns objects of a varying prototype or class[1] from some method call, which is assumed to be "new".

(Nguồn: Factory OOP (Wikipedia))

Riêng tên Pattern đã nói lên tất cả, Factory ở đây có thể hiểu nôm na là **nhà máy**. Theo định nghĩa từ Wikipedia, Factory chính là một đối tượng dùng để khởi tạo các đối tượng khác, thường thì factory sẽ là một hàm hay một phương thức trả về các đối tượng của một vài class khác nhau.

Factory Pattern thuộc vào nhóm khởi tạo (**Creational**), bao gồm các loại pattern:

1. **Simple Factory**: không được liệt vào dạng design pattern, nó chỉ được coi như một kỹ thuật dùng để **đóng gói** quá trình khởi tạo đối tượng.
2. **Factory Method**: định nghĩa ra một **giao diện** (interface) để khởi tạo đối tượng, tuy nhiên việc khởi tạo đối tượng từ class nào lại **được thực thi từ class triển khai giao diện đó**.
3. **Abstract Factory**: Cung cấp một giao diện dùng để khởi tạo một tập hợp các đối tượng có liên quan hoặc phụ thuộc với nhau mà không chỉ ra có những lớp cụ thể nào ở thời điểm thiết kế.

Như vậy tôi đã giới thiệu xong với các bạn về định nghĩa cũng như phân loại trong Factory Pattern. Đúng là lý thuyết vẫn mãi là lý thuyết (yaoming), vẫn luôn hàn lâm, nhiều thuật ngữ và khó có thể hiểu ngay được. Chính vì vậy, trong bài viết này tôi muốn có một cách tiếp cận khác giúp bạn dễ hiểu hơn về pattern này thông qua ví dụ thực tế (thay vì việc đưa ra định nghĩa, đưa ra sơ đồ, đưa ra triển khai, ... blah blah).

Vấn đề 1

Tôi và các bạn hãy cùng thử đóng vai trò là ông chủ trong bài viết này. Chúng ta sẽ thành lập một công ty phần mềm (lấy đại tên là F.C.U.K đi, trụ sở chính ở Hà Nội luôn cho máu), ngoài việc làm outsource để kiếm tiền nuôi công ty, chúng ta sẽ có một bộ phận **Training** những bạn sinh viên mới ra trường chẳng hạn để đưa vào các dự án thực tế. Và chúng ta sẽ lần lượt đi qua từng khó khăn để đưa ra một bộ máy hoạt động ổn nhất.

OK! Mọi việc đã rõ ràng và bây giờ chúng ta tập trung vào việc xây dựng bộ máy **Training** những bước đầu tiên. Trước mắt chúng ta sẽ đào tạo để thu được các **Developer** về mảng **PHP** và **Ruby**.

Chúng ta sẽ thiết kế một class DevelopersFactory (đây chính là bộ phận Training) và cung cấp một phương thức `produceDeveloper()`:

```
class DevelopersFactory {
    public function produceDeveloper($type) {
        switch ($type) {
            case 'Php':
                $developer = new PhpDeveloper();
                break;
            case 'Ruby':
                $developer = new RubyDeveloper();
                break;

            default:
                $developer = null;
                break;
        }

        $developer->training();
        $developer->deliver();

        return $developer;
    }
}
```

Như đã nói ở trên, ta cần "*sản xuất*" ra các ông Developer nên tôi đặt tên hàm là `produceDeveloper()` (okay), nhận đầu vào là tham số $type (có thể hiểu là mong muốn của các bạn sinh viên mới ra trường muốn dev Php hay Ruby).

Công việc cần làm là khởi tạo một ông `$developer`, sau đó huấn luyện (`training()`), và cuối cùng là gán vào các dự án thực tế `(deliver())`. Việc huấn luyện và bàn giao thế nào tạm cho qua ở thời điểm này.

Sau một thời gian chạy quy trình này mọi thứ êm xuôi, các ông Developer được huấn luyện xong vào dự án "cốt" ngon lành. Bài toán đặt ra với chúng ta là mở rộng thêm ngôn ngữ để training, ví dụ: **Android**. Việc này quá đơn giản, thêm 1 case nữa vào trong switch ... case là giải quyết vấn đề:

```
class DevelopersFactory {
    public function produceDeveloper($type) {
        switch ($type) {
            case 'Php':
                $developer = new PhpDeveloper();
                break;
            case 'Ruby':
                $developer = new RubyDeveloper();
                break;
            case 'Android':
                $developer = new AndroidDeveloper();
                break;

            default:
                $developer = null;
                break;
        }

        $developer->training();
```

```
        $developer->deliver();

        return $developer;
    }
}
```

Mọi việc tưởng chừng như dễ dàng nhưng lại có vấn đề ở đây, ta vừa thực hiện sửa trực tiếp vào hàm `produceDeveloper`() mỗi khi có thêm nội dung mới cần training. Công việc cần làm nặng nề hơn (ví dụ: thêm giáo án mới cho nội dung mới cần training). Đồng thời việc làm này cũng vị phạm nguyên tắc open/close:

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

Đây chính là lúc cần áp dụng kỹ thuật Simple Factory để giải quyết vấn đề!

Giải pháp: Simple Factory

Công việc của chúng ta đang bị dồn hết vào một bộ phận (chính là hàm `produceDeveloper`()). Các đối tượng được khởi tạo ngay trong bộ máy chính của chúng ta (DevelopersFactory).

Đã đến lúc tạo một class mới để đóng gói quá trình khởi tạo các đối tượng, ta gọi class đó là SimpleFactory. Bạn có thể hiểu là chúng ta tạo thêm một phòng ban mới hỗ trợ Training, chuyên làm công việc thu nhận hồ sơ, tâm sự ứng viên, định hướng, đưa ra plan và các yêu cầu cần training, ...

```
class SimpleFactory {
    public function createDeveloper($type) {
        switch ($type) {
            case 'Php':
                $developer = new PhpDeveloper();
                break;
            case 'Ruby':
                $developer = new RubyDeveloper();
                break;
            case 'Android':
                $developer = new AndroidDeveloper();
                break;

            default:
                $developer = null;
                break;
        }

        return $developer;
    }
}
```

Sau đó ta chỉnh sửa lại quy trình ở bộ phận chính:

```
class DevelopersFactory {
    public $simpleFactory;

    public function __construct() {
```

```
        $this->simpleFactory = new SimpleFactory();
    }

    public function produceDeveloper($type) {
        $developer = $this->simpleFactory->createDeveloper($type);

        $developer->training();
        $developer->deliver();

        return $developer;
    }
}
```

Vậy là nhờ có sự trợ giúp của phòng mới thêm (SimpleFactory), bộ phận chính của chúng ta (DevelopersFactory) không phải lo việc nói chuyện với ứng viên nữa mà chỉ nhận đầu vào từ phòng mới và training theo đúng plan và đúng ngôn ngữ ứng viên mong muốn hoặc được định hướng.

Thử demo với đoạn code sau:

```
$developersFactory = new DevelopersFactory();
$developersFactory->produceDeveloper('Php');
```

Kết quả:

```
Php Developer is trained.
Php Developer is delivered with 1000$.
```

Vấn đề 2

Tôi xin dành chút thời gian để giải thích kỹ hơn về các `class PhpDeveloper, RubyDeveloper` cũng như `AndroidDeveloper`. Thực ra thì chúng đều được kế thừa từ một lớp trừu tượng (**abstract**) Developer được định nghĩa sẵn các phương thức `training()` và `deliver()`. Hãy cùng xem thiết kế của lớp Developer:

```
abstract class Developer {
    public $type = '';
    public $price = 0;

    public function training() {
        echo $this->type . ' Developer is trained.<br/>';
    }

    public function deliver() {
        echo $this->type . ' Developer is delivered with ' . $this-
>price . '$.<br/>';
    }
}
Lớp PhpDeveloper, RubyDeveloper và AndroidDeveloper hết sức đơn
giản:
class PhpDeveloper extends Developer {
    public $type = 'Php';
    public $price = 1000;
}
```

```
class RubyDeveloper extends Developer {
    public $type = 'Ruby';
    public $price = 500;
}

class AndroidDeveloper extends Developer {
    public $type = 'Android';
    public $price = 150;
}
```

Lại tiếp tục câu chuyện tưởng tượng thì cứ coi như mỗi ông Developer sau khi được **training** sẽ ra ngoài outsource cho công ty với một giá nhất định được công ty trả.

Mọi thứ tạm thời lại tiếp tục ổn về mặt quy trình. Nhưng chúng ta không muốn dừng lại ở đây, chúng ta cần mở rộng quy mô và quyết định thành lập thêm hai chi nhánh ở *TP Hồ Chí Minh* và *Đà Nẵng* (honho). Trong tay ta đã có sẵn flow chạy ngon lành từ *Hà Nội* rồi nên việc làm hết sức đơn giản là setup và bê nguyên quy trình sang bên hai chi nhánh mới. Cụ thể việc triển khai:

Với chi nhánh **HCM**:

```
class HcmSimpleFactory {
    public function createDeveloper($type) {
        switch ($type) {
            case 'Php':
                $developer = new HcmPhpDeveloper();
                break;
            case 'Ruby':
                $developer = new HcmRubyDeveloper();
                break;
            case 'Android':
                $developer = new HcmAndroidDeveloper();
                break;

            default:
                $developer = null;
                break;
        }

        return $developer;
    }
}

class HcmDevelopersFactory {
    public $simpleFactory;

    public function __construct() {
        $this->simpleFactory = new HcmSimpleFactory();
    }

    public function produceDeveloper($type) {
        $developer = $this->simpleFactory->createDeveloper($type);

        $developer->training();
        $developer->deliver();

        return $developer;
```

```
    }
}
```

Sau một khoảng thời gian chạy thì phát sinh vấn đề về luồng hoạt động. Chi nhánh HCM thì bỏ hàm `training()` đưa thẳng vào dự án, chi nhánh Đà Nẵng thì bỏ hàm `deliver()` chỉ training xong để đấy. Hàm `produceDeveloper()` bị ảnh hưởng nặng nề và không theo quy chuẩn của chi nhánh Hà Nội. Rõ ràng nếu là chủ chúng ta không muốn điều này nên cần có biện pháp để ép quy trình về một flow chuẩn.

Giải pháp: Factory Method

Giải pháp đưa ra hết sức đơn giản, hàm `produceDeveloper()` không được phép thay đổi vì ảnh hưởng quy trình nên ta sẽ để hàm này cố định trong một `Abstract Class DevelopersFactory`, hàm này sử dụng kết quả của `một abstract method createDeveloper()` (trả về đối tượng developer cụ thể nào đó). Bằng cách này hàm `createDeveloper()` giúp chúng ta đóng gói quá trình khởi tạo đối tượng, và bởi vì đây là **abstract method** nên quá trình khởi tạo thế nào sẽ được thực thi ở lớp con kế thừa.

Có vẻ giải pháp khá hợp lý và ta sẽ triển khai như sau. Với quy trình chuẩn, lớp trừu tượng `DevelopersFactory` được thiết kế như sau:

```
abstract class DevelopersFactory {
    public function produceDeveloper($type) {
        $developer = $this->createDeveloper($type);

        $developer->training();
        $developer->deliver();

        return $developer;
    }

    abstract public function createDeveloper($type);
}
```

Với chi nhánh HCM và Đà Nẵng, chúng ta cần triển khai phương thức trừu tượng `createDeveloper()`:

```
class HcmDevelopersFactory extends DevelopersFactory {
    public function createDeveloper($type) {
        switch ($type) {
            case 'HcmPhp':
                $developer = new HcmPhpDeveloper();
                break;
            case 'HcmRuby':
                $developer = new HcmRubyDeveloper();
                break;
            case 'HcmAndroid':
                $developer = new HcmAndroidDeveloper();
                break;

            default:
                $developer = null;
                break;
        }
```

```
        return $developer;
    }
}
class DnDevelopersFactory extends DevelopersFactory {
    public function createDeveloper($type) {
        switch ($type) {
            case 'DnPhp':
                $developer = new DnPhpDeveloper();
                break;
            case 'DnRuby':
                $developer = new DnRubyDeveloper();
                break;
            case 'DnAndroid':
                $developer = new DnAndroidDeveloper();
                break;

            default:
                $developer = null;
                break;
        }

        return $developer;
    }
}
```

Vấn đề đã được giải quyết. Trong đoạn code trên ta có thể nói phương thức `createDeveloper()` trong lớp trừu tượng DevelopersFactory được gọi là Factory Method.

Thử demo với đoạn code sau:

```
$test = new HcmDevelopersFactory();
$test->produceDeveloper('HcmPhp');
```

Kết quả:

```
HCM Php Developer is trained.
HCM Php Developer is delivered with 950$.
```

Đúng như định nghĩa **Factory Method** pattern định nghĩa ra một giao diện (`createDeveloper()`) để khởi tạo đối tượng, tuy nhiên việc khởi tạo đối tượng thực sự như thế nào lại được triển khai ở các lớp con (`HcmDevelopersFactory`, `DnDevelopersFactory`).

Ưu điểm khi sử dụng Factory Method pattern

- Giúp cho code của chúng ta tuân thủ nguyên tắc DRY. Dù cho việc khởi tạo đối tượng phụ thuộc vào nhiều class tuy nhiên chúng ta cũng chỉ có một hàm để khởi tạo, việc thêm một số config khi khởi tạo cũng không làm code thay đổi quá nhiều chỗ.
- Tránh sự phân tán tư tưởng khi code. Cụ thể từ ví dụ của chúng ta, ở lớp DevelopersFactory và hàm `produceDeveloper()`, ta không cần quan

tâm xem ông Developer là PHP, Ruby hay Android. Ta chỉ cần biết là có 1 ông Developer mới, sau đó được training và đưa vào dự án cụ thể.

Vấn đề 3

Tiếp theo ta cùng đi sâu hơn vào quy trình **training** (cụ thể là phương thức `training()` trong Developer class). Để đảm bảo có một đầu ra tốt, chúng ta phải thắt chặt quy trình, giả sử để training một new dev đều phải qua một khóa training chung về *Git, basic, advanced, framework, ...*

Abstract Factory Pattern

Đầu tiên, chúng ta phải thiết kế một Abstract Factory class - khung quy trình chung cho việc training. Ta có thể hiểu rằng Abstract Factory bao gồm nhiều Factory Method. Cụ thể ta sẽ thiết kế một lớp abstract `TrainingComponentsFactory`:

```
class TrainingComponentsFactory {
    abstract function trainingGit();
    abstract function trainingBasic();
    abstract function trainingFramework();
}
Triển khai quy trình chung với chi nhánh HCM:
class HcmTrainingComponentsFactory extends TrainingComponentsFactory
{
    public function trainingGit() {
        return new HcmGitRequirement();
    }

    public function trainingBasic() {
        return new HcmBasicProject();
    }

    public function trainingFramework() {
        return new HcmFrameworkProject();
    }
}
```

Điều này dẫn tới lớp abstract Developer của chúng ta cũng phải thay đổi dựa trên yêu cầu training, cụ thể ta cần thêm các thuộc tính về *git*, *basic*, *framework* và phương thức training sẽ được chuyển về dạng **abstract** để các Developer từng mảng có những triển khai đặc biệt riêng.

Với lớp Developer ta thay đổi như sau:

```
abstract class Developer {
    public $type = '';
    public $price = 0;
    public $git = null;
    public $basic = null;
    public $framework = null;

    abstract function training();

    public function deliver() {
```

```
        echo $this->type . ' Developer is delivered with ' . $this-
>price . '$.<br/>';
    }
}
```

Đồng thời ta cũng phải thay đổi các lớp PhpDeveloper và Ruby Developer:

```
class PhpDeveloper extends Developer {
    public $type = 'Php';
    public $price = 1000;
    public $trainingComponentsFactory = null;

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

    public function training() {
        $this->git = $this->trainingComponentsFactory-
>trainingGit();
        $this->basic = $this->trainingComponentsFactory-
>trainingBasic();
        $this->framework = $this->trainingComponentsFactory-
>trainingFramework();
    }
}


class RubyDeveloper extends Developer {
    public $type = 'Ruby';
    public $price = 900;
    public $trainingComponentsFactory = null;

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

    public function training() {
        $this->git = $this->trainingComponentsFactory-
>trainingGit();
        $this->basic = $this->trainingComponentsFactory-
>trainingBasic();
    }
}
```

Nhìn qua thì ta có thể thấy là chỉ có PhpDeveloper cần training đủ 3 bước còn RubyDeveloper thì chỉ cần qua basic là có thể sử dụng được luôn. Cuối cùng ta cần thay đổi lớp DevelopersFactory bằng việc thêm các thành phần cần training (TrainingComponentsFactory) vào:

```
class HcmDevelopersFactory extends DevelopersFactory {
    public function createDeveloper($type) {
        $hcmTrainingComponentsFactory = new
HcmTrainingComponentsFactory();

        switch ($type) {
            case 'HcmPhp':
```

```
                $developer = new
HcmPhpDeveloper($hcmTrainingComponentsFactory);
                break;
            case 'HcmRuby':
                $developer = new
HcmRubyDeveloper($hcmTrainingComponentsFactory);
                break;

            default:
                $developer = null;
                break;
        }

        return $developer;
    }
}
```

Factory Pattern trong Laravel

Trong Laravel, chúng ta cũng thấy họ sử dụng Factory Method pattern trong việc quản lý khởi tạo các kết nối database. Cụ thể là trong Illuminate/Database/Connectors/ConnectionFactory.php:

```
class ConnectionFactory {
    ...
    /**
     * Create a new connection instance.
     *
     * @param  string   $driver
     * @param  \PDO     $connection
     * @param  string   $database
     * @param  string   $prefix
     * @param  array    $config
     * @return \Illuminate\Database\Connection
     *
     * @throws \InvalidArgumentException
     */
    protected function createConnection($driver, PDO $connection,
$database, $prefix = '', array $config = array())
    {
        if ($this->container->bound($key =
"db.connection.{$driver}"))
        {
            return $this->container->make($key, array($connection,
$database, $prefix, $config));
        }

        switch ($driver)
        {
            case 'mysql':
                return new MySqlConnection($connection, $database,
$prefix, $config);

            case 'pgsql':
                return new PostgresConnection($connection,
$database, $prefix, $config);

            case 'sqlite':
```

```
                return new SQLiteConnection($connection, $database,
$prefix, $config);

            case 'sqlsrv':
                return new SqlServerConnection($connection,
$database, $prefix, $config);
        }

        throw new \InvalidArgumentException("Unsupported driver
[$driver]");
    }
}
```

Thêm một ví dụ về việc sử dụng Factory pattern trong Laravel đó là việc validate dữ liệu. Chúng ta có thể định nghĩa nhiều luật (rules) validate dữ liệu thông qua Validation class. Để làm việc này chúng ta thường hay định nghĩa **validation rules** ở Model và gọi từ phía Controller.

Và bạn cũng có thể thấy rằng chúng ta hoàn toàn có thể đưa ra các custom rules và custom error message để validate dữ liệu đúng không? Hãy cùng xem class Illuminate/Validation/Factory.php

```
class Factory implements FactoryContract {
    ...
    /**
     * Create a new Validator instance.
     *
     * @param  array  $data
     * @param  array  $rules
     * @param  array  $messages
     * @param  array  $customAttributes
     * @return \Illuminate\Validation\Validator
     */
    public function make(array $data, array $rules, array $messages
= [], array $customAttributes = [])
    {
        // The presence verifier is responsible for checking the
unique and exists data
        // for the validator. It is behind an interface so that
multiple versions of
        // it may be written besides database. We'll inject it into
the validator.
        $validator = $this->resolve($data, $rules, $messages,
$customAttributes);
        if (! is_null($this->verifier)) {
            $validator->setPresenceVerifier($this->verifier);
        }
        // Next we'll set the IoC container instance of the
validator, which is used to
        // resolve out class based validator extensions. If it is
not set then these
        // types of extensions will not be possible on these
validation instances.
        if (! is_null($this->container)) {
            $validator->setContainer($this->container);
        }

        // Điều kỳ diệu nằm ở đây
```

```
        $this->addExtensions($validator);

        return $validator;
    }
}
```

Ta có thể thấy Validation Class được khởi tạo cùng với Translator class và một IoC Container. Với hàm `make()` mà tôi dẫn chứng ở trên trước khi trả về kết quả có gọi đến `$this->addExtentions($validator)`.

Phương thức `addExtentions()` được gọi để gắn thêm những rule custom của chúng ta.

Kết luận

- Trên đây tôi đã giới thiệu với các bạn về khái niệm cũng như đưa ra một bài toán cụ thể để hiểu hơn về Factory Pattern. Đồng thời cũng đưa ra ưu điểm khi sử dụng pattern này.
- Trong Laravel họ cũng triển khai Factory Pattern nhưng không theo một cách truyền thống mà có biến thể đi chút.

Bài 9. [Builder Pattern] Builder Pattern trong Laravel

Bài toán đặt ra

Khi khai báo một lớp (Class) nào đó chúng ta biết đến khái niệm constructor. Một thực thể (Instance) của lớp được tạo ra bao giờ cũng được gọi hàm *constructor* để khởi tạo các thành phần (hay có thể hiểu như các thuộc tính) ban đầu. Như vậy các đối tượng được sinh ra từ một lớp nào đó với cùng một constructor sẽ có các thể hiện giống nhau.

Vấn đề đặt ra khi ta phải làm việc với các đối tượng phức tạp:

- Được tạo ra từ nhiều thành phần nhỏ lắp ghép lại.
- Trong các thành phần nhỏ tạo nên đối tượng có những thành phần bắt buộc và có những thành phần không bắt buộc.

Để dễ tưởng tượng hơn chúng ta hãy cùng đến với một ví dụ cụ thể. Trong ứng dụng của tôi có quản lý đối tượng người dùng (user), tôi có một khai báo cho lớp **User** như sau:

```
class User {
      private $username; // required
      private $email;  // required
      private $birthday;  // optional
```

```
	private $description;  // optional

	function __construct($usernameParam, $emailParam,
$birthdayParam = "", $descriptionParam = "") {
		$this->username = $usernameParam;
		$this->email = $emailParam;
		$this->birthday = $birthdayParam;
		$this->description = $descriptionParam;
	}
}
```

Chúng ta có lớp User với các thuộc tính bắt buộc là **username** và **email**, đồng thời các thuộc tính không bắt buộc là **birthday** và **description**. Hàm constructor gồm bốn đầu vào và các giá trị *optional* sẽ có giá trị mặc định (nếu không được truyền giá trị).

Ta có thể nhận ra một số nhược điểm của cách làm này:

- Nếu có thêm nhiều thuộc tính thì **constructor** sẽ phải khai báo dài.
- Khi khởi tạo đối tượng từ lớp User chúng ta luôn phải để ý xem thứ tự khai báo các biến thế nào, giá trị nào bắt buộc phải truyền vào, giá trị nào không cần phải truyền và nếu không truyền thì giá trị mặc định là bao nhiêu.
- Khi thêm thuộc tính hoặc thứ tự thuộc tính trong hàm khởi tạo thay đổi sẽ gây không ít phiền hà.

Giải pháp

Đến đây ta có thể đưa ra giải pháp cho những nhược điểm vừa nêu ở trên như sau:

- Chỉ truyền những thuộc tính bắt buộc vào hàm khởi tạo.
- Các thuộc tính không bắt buộc cho phép khởi tạo thông qua setter.

Cụ thể cách triển khai như sau:

```
class User {
	private $username; // required
	private $email;  // required
	private $birthday;  // optional
	private $description;  // optional

	function __construct($usernameParam, $emailParam) {
		$this->username = $usernameParam;
		$this->email = $emailParam;
	}

	public function setBirthday($birthdayParam) {
		$this->birthday = $birthdayParam;
	}

	public function setDescription($descriptionParam) {
		$this->description = $descriptionParam;
	}
}
```

Tuy nhiên cách làm này vẫn tồn tại những nhược điểm:

- Khi số lượng thuộc tính không bắt buộc tăng lên ta sẽ phải triển khai nhiều setter.
- Việc sử dụng setter sẽ khiến cho trạng thái của đối tượng sau khi tạo ra biến đổi, khó kiểm soát (vì chúng ta có thể chủ động thay đổi giá trị).

Ý tưởng cải tiến

Ý tưởng đặt ra để khắc phục những nhược điểm nêu trên:

- Tách các xử lý phức tạo ra ngoài Constructor.
- Bàn giao công việc khởi tạo cho một đối tượng khác, chia việc khởi tạo các thuộc tính ra riêng rẽ sau đó sẽ lắp ghép lại để xây dựng nên đối tượng.

Đây chính là ý tưởng cơ bản của **Builder Pattern**.

Builder Pattern là gì?

Separate the construction of a complex object from its representation so that the same construction process can create different representations

- Là Pattern phục vụ cho việc khởi tạo các đối tượng (thuộc nhóm Creational)
- Tách rời quá trình tạo object với nội dung và cấu trúc bên trong của nó, nhờ vậy tương ứng với một quá trình tạo object có thể có nhiều cách tạo nhiều thể hiện khác nhau.
- Sử dụng khi quá trình khởi tạo object phức tạp.
- Sử dụng một đối tượng đóng vai trò là **Builder** để phục vụ cho việc khởi tạo đối tượng khác.

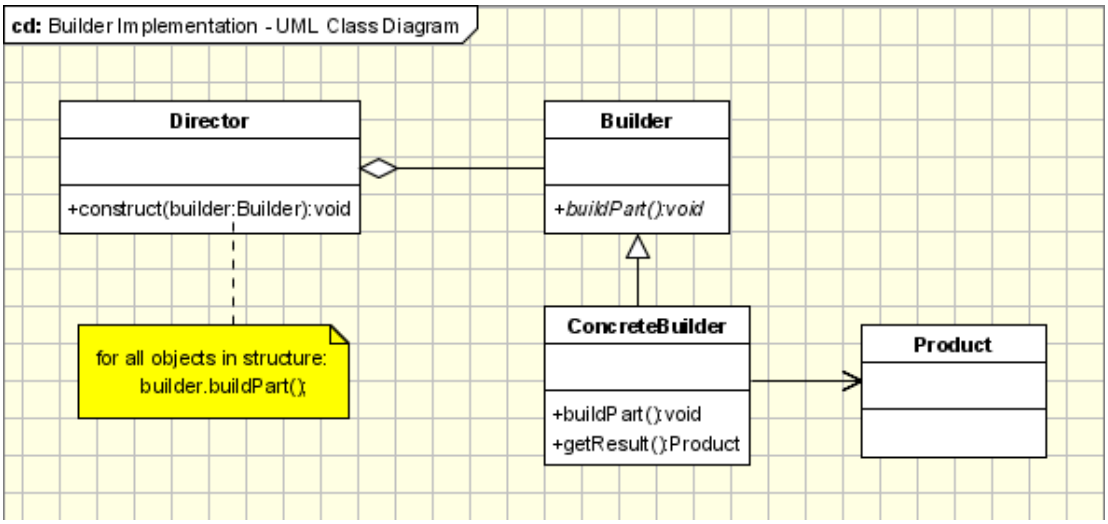
Áp dụng ý tưởng của Builder Pattern với ví dụ ban đầu, chúng ta sẽ triển khai khởi tạo một class UserBuilder làm nhiệm vụ khởi tạo đối tượng cho class User.

Mô hình Builder Pattern

Trước khi đi vào phân tích mô hình Builder Pattern, tôi xin đưa ra một bài toán để các bạn tiện hình dung. Chúng ta cần tạo ra một đối tượng cụ thể ví dụ là chiếc xe máy. Thành phần bao gồm: khung xe, lốp xe, động cơ, xích, phanh, ... việc tạo ra các thành phần không nhất thiết phải thực hiện đồng thời, cũng không nhất thiết phải có trình tự trước sau thế nào mà có thể được tạo ra độc lập bởi các cá nhân/tổ chức khác nhau.

Tuy nhiên để có một chiếc xe hoàn chỉnh chúng ta cần tập hợp lại các thành phần từ các nơi sản xuất riêng lẻ và lắp ráp lại thành một chiếc xe máy hoàn chỉnh. Việc làm này được thực hiện bởi một nhà máy sản xuất (VD: Honda Vietnam, Honda Laos, ...) và nhà máy sản xuất tôi đang nhắc đến đóng vai trò **Builder**.

Hãy cùng xem mô hình **Builder Pattern** dưới đây:



- **Builder**: định nghĩa một interface hoặc abstract class cho việc tạo ra các thành phần của đối tượng **Product**.
- **ConcreteBuilder**: lớp cài đặt chi tiết các thành phần được định nghĩa bởi **Builder**
    - Xây dựng và lắp ráp các thành phần của **Product** bằng cách thực thi **Builder**.
    - Định nghĩa và giữ liên kết các thành phần tạo ra.
    - Đưa ra interface để lấy **Product**.
- **Product**: đại diện cho đối tượng đang được xây dựng.
- **Director**: xây dựng đối tượng **Product** sử dụng **Builder** interface.

Builder (Manager) Pattern trong Laravel

Trong Laravel, Builder Pattern cũng được hiểu như là Manager Pattern. Ví dụ class AuthManager cần tạo ra một số thành phần cần được bảo mật để tái sử dụng với những thành phần lưu trữ như **cookie**, **session** hay **custom** (gọi là **driver**). Để giải quyết vấn đề này, AuthManager class sử dụng các hàm để lưu trữ như callCustomCreator() và getDrivers() từ class Manager.

Chúng ta hãy cùng xem Builder (Manager) Pattern thể hiện như thế nào thông qua file vendor/Illuminate/Support/Manager.php:

```
public function driver($driver = null)
{
      ...
}

protected function createDriver($driver)
{
    $method = 'create'.ucfirst($driver).'Driver';
    ...
```

```
}

protected function callCustomCreator($driver)
{
      return $this->customCreators[$driver]($this->app);
}

public function extend($driver, Closure $callback)
{
    $this->customCreators[$driver] = $callback;
    return $this;
}

public function getDrivers()
{
      return $this->drivers;
}

public function __call($method, $parameters)
{
      return call_user_func_array(array($this->driver(), $method),
$parameters);
}
```

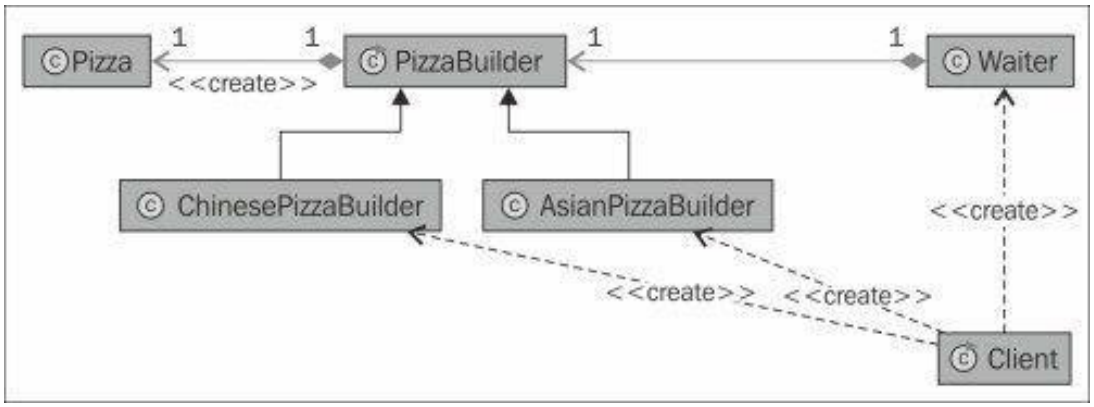
và vendor/Illuminate/Auth/AuthManager.php:

```
protected function createDriver($driver)
{
      ....
}

protected function callCustomCreator($driver)
{
      ...
}

public function createDatabaseDriver()
{
      ...
}

protected function createDatabaseProvider()
{
      ....
}

public function createEloquentDriver()
{
      ...
}

protected function createEloquentProvider()
{
      ...
}

public function getDefaultDriver()
{
      ...
}

public function setDefaultDriver($name)
```

```
{
        ...
}
```

Nhìn vào 2 file tóm tắt ở trên ta có thể thấy class AuthManager kế thừa từ class Manager. Như chúng ta đã biết Laravel cung cấp cơ chế basic auth, chúng ta lưu trữ thông tin xác thực trong **Database**. Đầu tiên, class AuthManager kiểm tra cấu hình lưu trữ thông tin xác thực có phải mặc định là database không thông qua phương thức getDefaultDriver(). Hàm này thực chất là sử dụng class Manager phục vụ cho các thao tác với Eloquent. Tất cả các tùy chọn cho **Database** và xác thực (**auth**) đều lấy được từ các file config (được đặt trong thư mục **config**).

Để hiểu hơn về Manager Pattern ta có thể xem qua sơ đồ ví dụ sau:



Client muốn đặt pizza: Asian pizza và/hoặc Chinese pizza. Chiếc pizza được yêu cầu từ class **Waiter**. class **PizzaBuilder** (trong trường hợp này có thể hiểu như class Manager) sẽ làm chiếc pizza theo đúng yêu cầu (trường hợp này là thông qua yêu cầu từ class AuthManager), sau đó sẽ chuyển chiếc pizza tới đúng nơi request thông qua **Waiter**.

Ngoài ra thì các bạn có thể đọc thêm code ở `vendor/Illuminate/Session/SessionManager.php` để tìm hiểu thêm về cách sử dụng `Manager Pattern` trong Laravel.

Bài 10. Hướng dẫn Java Design Pattern – Object Pool

Trong OOP, một class có thể có rất nhiều instance nhưng ngược lại Singleton là một dạng class mà chỉ hỗ trợ tối đa một instance duy nhất và một đối tượng khi đã được khởi tạo sẽ tồn tại suốt vòng đời chương trình. Trong một số trường hợp, chúng ta cần

khởi tạo và sử dụng một tập hợp các đối tượng. Khi với số lượng lớn các đối tượng giống nhau, thì việc khởi tạo nhiều lần sẽ gây lãng phí không cần thiết. Chúng ta cũng có thể sử dụng Prototype Pattern để cãi thiện performance bằng cách cloning object. Tuy nhiên, không phải lúc nào object cũng có thể được clone đầy đủ. Trong những trường hợp như vậy, chúng ta có thể dùng Object pool pattern.

Object Pool Pattern là gì?

Object Pool is a creational design pattern. Object Pool Pattern says that "to reuse the object that are expensive to create".

Object Pool Pattern là một trong những Creational pattern. Nó không nằm trong danh sách các Pattern được giới thiệu bởi GoF. Object Pool Pattern cung cấp một kỹ thuật để tái sử dụng objects thay vì khởi tạo không kiểm soát.

Ý tưởng của Object Pooling là: chúng ta dùng Object Pool Pattern quản lý một tập hợp các objects mà sẽ được tái sử dụng trong chương trình. Khi client cần sử dụng object, thay vì tạo ra một đối tượng mới thì client chỉ cần đơn giản yêu cầu Object pool lấy một đối tượng đã có sẵn trong object pool. Sau khi object được sử dụng nó sẽ không hủy mà sẽ được trả về pool cho client khác sử dụng. Nếu tất cả các object trong pool được sử dụng thì client phải chờ cho tới khi object được trả về pool.

Object pool thông thường hoạt động theo kiểu: tự tạo đối tượng mới nếu chưa có sẵn hoặc khởi tạo trước 1 object pool chứa một số đối tượng hạn chế trong đó.

Cài đặt Object Pool Pattern như thế nào?

Cài đặt

Một Object Pool Pattern bao gồm các thành phần cơ bản sau:

- **Client**: một class yêu cầu khởi tạo đối tượng PooledObject để sử dụng.
- **PooledObject**: một class mà tốn nhiều thời gian và chi phí để khởi tạo. Một class cần giới hạn số lượng đối tượng được khởi tạo trong ứng dụng.
- **ObjectPool**: đây là lớp quan trọng nhất trong Object Pool Pattern. Lớp này lưu giữ danh sách các PooledObject đã được khởi tạo, đang được sử dụng. Nó cung cấp các phương thức cho việc lấy đối tượng từ Pool và trả đối tượng sau khi sử dụng về Pool.

Ví dụ Object Pool thông qua ứng dụng Taxi

Một hãng taxi A chỉ hữu hạn N chiếc taxi, hãng taxi chịu trách nhiệm quản lý trạng thái các xe (đang rảnh hay đang chở khách), phân phối các xe đang rảnh đi đón khách, chăm sóc, kéo dài thời gian chờ đợi của khách hàng cho trong trường hợp tất cả các xe đều đang bận (để chờ một trong số các xe đó rảnh thì điều đi đón khách luôn), hủy khi việc chờ đợi của khách hàng là quá lâu.

Ta mô phỏng và thiết kế thành các lớp sau:

- **Taxi:** đại diện cho một chiếc taxi, là một class định nghĩa các thuộc tính và phương thức của một taxi.

- **TaxiPool**: Đại diện cho công ty taxi, có:
- **Phương thức getTaxi()**: để lấy về một thể hiện Taxi đang ở trạng thái rảnh, có thể throw ra một exception nếu chờ lâu mà không lấy được thể hiện.
- **Phương thức release()**: để trả thể hiện Taxi về Pool sau khi đã phục vụ xong.
- **Thuộc tính available**: lưu trữ danh sách Taxi rãnh, đang chờ phục vụ.
- **Thuộc tính inUse:** lưu trữ danh sách Taxi đang bận phục vụ.
- **ClientThread:** đại diện cho khách hàng sử dụng dịch vụ Taxi, mô phỏng việc gọi, chở và trả khách.

Trong đoạn code bên dưới, tôi sẽ cài đặt mô phỏng với TaxiPool quản lý được 4 taxi, cùng lúc có 8 cuộc gọi của khách hàng đến công ty để gọi xe, thời gian mỗi taxi đến địa điểm chở khách là 200ms, mỗi taxi chở khách trong khoảng thời gian từ 1000ms đến 1500ms (ngẫu nhiên), mỗi khách hàng chịu chờ tối đa 1200ms trước khi hủy.

Taxi

```
package com.gpcoder.patterns.creational.objecpool.taxi;

public class Taxi {

    private String name;

    public Taxi(String name) {
        super();
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return "Taxi [name=" + name + "]";
    }
}
```

TaxiPool

```
package com.gpcoder.patterns.creational.objecpool.taxi;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.atomic.AtomicBoolean;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Lazy pool
 *
 * @author gpcoder
```

```
 */
public class TaxiPool {

    private static final long EXPIRED_TIME_IN_MILISECOND = 1200; //
1.2s
    private static final int NUMBER_OF_TAXI = 4;

    private final List<Taxi> available =
Collections.synchronizedList(new ArrayList<>());
    private final List<Taxi> inUse =
Collections.synchronizedList(new ArrayList<>());

    private final AtomicInteger count = new AtomicInteger(0);
    private final AtomicBoolean waiting = new AtomicBoolean(false);

    public synchronized Taxi getTaxi() {
        if (!available.isEmpty()) {
            Taxi taxi = available.remove(0);
            inUse.add(taxi);
            return taxi;
        }
        if (count.get() == NUMBER_OF_TAXI) {
            this.waitingUntilTaxiAvailable();
            return this.getTaxi();
        }
        Taxi taxi = this.createTaxi();
        inUse.add(taxi);
        return taxi;
    }

    public synchronized void release(Taxi taxi) {
        inUse.remove(taxi);
        available.add(taxi);
        System.out.println(taxi.getName() + " is free");
    }

    private Taxi createTaxi() {
        waiting(200); // The time to create a taxi
        Taxi taxi = new Taxi("Taxi " + count.incrementAndGet());
        System.out.println(taxi.getName() + " is created");
        return taxi;
    }

    private void waitingUntilTaxiAvailable() {
        if (waiting.get()) {
            waiting.set(false);
            throw new TaxiNotFoundException("No taxi available");
        }
        waiting.set(true);
        waiting(EXPIRED_TIME_IN_MILISECOND);
    }

    private void waiting(long numberOfSecond) {
        try {
            TimeUnit.MILLISECONDS.sleep(numberOfSecond);
        } catch (InterruptedException e) {
            e.printStackTrace();
            Thread.currentThread().interrupt();
        }
    }
```

```
}
```

ClientThread

```
package com.gpcoder.patterns.creational.objecpool.taxi;

import java.util.Random;
import java.util.concurrent.TimeUnit;

public class ClientThread implements Runnable {

    private TaxiPool taxiPool;

    public ClientThread(TaxiPool taxiPool) {
        this.taxiPool = taxiPool;
    }

    @Override
    public void run() {
        takeATaxi();
    }

    private void takeATaxi() {
        try {
            System.out.println("New client: " +
Thread.currentThread().getName());
            Taxi taxi = taxiPool.getTaxi();

            TimeUnit.MILLISECONDS.sleep(randInt(1000, 1500));


            taxiPool.release(taxi);
            System.out.println("Served the client: " +
Thread.currentThread().getName());
        } catch (InterruptedException | TaxiNotFoundException e) {
            System.out.println(">>>Rejected the client: " +
Thread.currentThread().getName());
        }
    }

    public static int randInt(int min, int max) {
        return new Random().nextInt((max - min) + 1) + min;
    }
}
TaxiNotFoundException
package com.gpcoder.patterns.creational.objecpool.taxi;

public class TaxiNotFoundException extends RuntimeException {


    private static final long serialVersionUID = -
6670953536653728443L;

    public TaxiNotFoundException(String message) {
        System.out.println(message);
    }
}
```

TaxiApp

```
package com.gpcoder.patterns.creational.objecpool.taxi;

public class TaxiApp {

    public static final int NUM_OF_CLIENT = 8;

    public static void main(String[] args) {
        TaxiPool taxiPool = new TaxiPool();
        for (int i = 1; i <= NUM_OF_CLIENT; i++) {
            Runnable client = new ClientThread(taxiPool);
            Thread thread = new Thread(client);
            thread.start();
        }
    }
}
```

Kết quả thực thi chương trình

```
New client: Thread-0
New client: Thread-1
New client: Thread-2
New client: Thread-3
New client: Thread-4
New client: Thread-5
New client: Thread-6
New client: Thread-7
Taxi 1 is created
Taxi 2 is created
Taxi 3 is created
Taxi 4 is created
Taxi 1 is free
Served the client: Thread-1
Taxi 2 is free
Served the client: Thread-7
No taxi available
>>>Rejected the client: Thread-0
Taxi 3 is free
Served the client: Thread-6
Taxi 4 is free
Served the client: Thread-5
Taxi 2 is free
Served the client: Thread-4
Taxi 1 is free
Served the client: Thread-3
Taxi 3 is free
Served the client: Thread-2
```

Nhận xét

Ưu điểm của việc cài đặt Pool là việc tận dụng được các tài nguyên đã được cấp phát. Với ví dụ về taxi ở trên với 4 taxi, trong nhiều trường hợp vẫn có thể đáp ứng được nhiều hơn 4 yêu cầu cùng một lúc. Nó làm tăng hiệu năng hệ thống ở điểm không cần phải khởi tạo quá nhiều thể hiện (trong nhiều trường hợp việc khởi tạo này mấy nhiều thời gian), tận dụng được các tài nguyên đã được khởi tạo (tiết kiệm bộ nhớ, không mất thời gian hủy đối tượng).

Việc cài đặt Pool có thể linh động hơn nữa bằng cách đặt ra 2 giá trị N và M. Trong đó: N là số lượng thể hiện tối thiểu (trong những lúc rảnh rỗi), M là số thể hiện tối đa (lúc cần huy động nhiều thể hiện nhất mà phần cứng đáp ứng được). Sau khi qua trạng thái cần nhiều thể hiện, Pool có thể giải phóng bớt một số thể hiện không cần thiết.

Ví dụ Object Pool thông qua Connection Pooling

Khi làm việc với cơ sở dữ liệu hay cho những hệ thống tương đối lớn ở các công ty, thì vấn đề performance rất quan trọng. Nếu mỗi request đến chúng ta phải mở và đóng kết nối thủ công thì rất khó quản lý, điều quan trọng hơn nữa đó là cứ mỗi lần open và close connection mất khoảng từ 2-3s thì chắc chắn rằng hiệu năng hoạt động của ứng dụng web không tốt. Để giải quyết được vấn đề này, chúng ta sẽ dùng kỹ thuật connection pool để quản lý và chia sẻ số kết nối. Connection Pool cũng là một trong các ứng dụng của Object Pool Pattern.

Connection pooling là gì?

Connection pool (vùng kết nối) : là kỹ thuật cho phép tạo và duy trì 1 tập các kết nối dùng chung nhằm tăng hiệu suất cho các ứng dụng bằng cách sử dụng lại các kết nối khi có yêu cầu thay vì việc tạo kết nối mới.

Cách làm việc của Connection pooling?

Connection Pool Manager (CPM) là trình quản lý vùng kết nối, một khi ứng dụng được chạy thì Connection pool tạo ra một vùng kết nối, trong vùng kết nối đó có các kết nối do chúng ta tạo ra sẵn. Và như vậy, một khi có một request đến thì CPM kiểm tra xem có kết nối nào đang rỗi không? Nếu có nó sẽ dùng kết nối đó còn không thì nó sẽ đợi cho đến khi có kết nối nào đó rỗi hoặc kết nối khác bị timeout. Kết nối sau khi sử dụng sẽ không đóng lại ngay mà sẽ được trả về CPM để dùng lại khi được yêu cầu trong tương lai.

Ví dụ

Một connection pool có tối đa 10 connection trong pool. Bây giờ user kết nối tới database (DB), hệ thống sẽ kiểm tra trong connection pool có kết nối nào đang rảnh không?

Trường hợp chưa có kết nối nào trong connection pool hoặc tất cả các kết nối đều bận (đang được sử dụng bởi user khác) và số lượng connection trong connection < 10 thì sẽ tạo một connection mới tới DB để kết nối tới DB đồng thời kết nối đó sẽ được đưa vào connection pool.

Trường hợp tất cả các kết nối đang bận và số lượng connection trong connection pool = 10 thì người dùng phải đợi cho các user dùng xong để được dùng.

Sau khi một kết nối được tạo và sử dụng xong nó sẽ không đóng lại mà sẽ duy trì trong connection pool để dùng lại cho lần sau và chỉ thực sự bị đóng khi hết thời gian timeout (lâu quá không dùng đến nữa).

Chi tiết các bạn tham khảo thêm tại link sau: https://ejbvn.wordpress.com/category/week-2-entity-beans-and-message-driven-beans/day-09-using-jdbc-to-connect-to-a-database/

Source code về cách tạo Connection Pool các bạn tham khảo thêm tại đây: https://sourcemaking.com/design_patterns/object_pool/java

Ví dụ Object Pool thông qua Thread Pool

Thread Pool cũng là một trong các ứng dụng của Object Pool Pattern.

Tạo ra một Thread mới là một hoạt động tốn kém bởi vì nó đòi hỏi hệ điều hành cung cấp tài nguyên để có thể thực thi task (tác vụ). ThreadPool được dùng để giới hạn số lượng Thread được chạy bên trong ứng dụng của chúng ta trong cùng một thời điểm.

Thay vì tạo các luồng mới khi các task (nhiệm vụ) mới đến, một ThreadPool sẽ giữ một số luồng nhàn rỗi (no task) đã sẵn sàng để thực hiện tác vụ nếu cần. Sau khi một thread hoàn thành việc thực thi một tác vụ, nó sẽ không chết. Thay vào đó nó vẫn không hoạt động trong ThreadPool và chờ đợi được lựa chọn để thực hiện nhiệm vụ mới.

Chi tiết về Thread Pool các bạn có thể xem lại ở bài viết này: https://gpcoder.com/3548-huong-dan-tao-va-su-dung-threadpool-trong-java/

Một vài lưu ý khi triển khai Object Pool

Xác định số lượng tối đa các đối tượng được khởi tạo trong Pool?

Tùy vào ứng dụng, chúng ta cần xác định con số này sao cho hợp lý để đảm bảo không khởi tạo quá dư thừa đối tượng gây lãng phí tài nguyên, hay quá ít làm cho các ứng dụng client phải chờ lâu hay bị lỗi.

Thời gian timeout?

Để quản lý thời gian timeout bạn cần xác định:

Khi một đối tượng không được sử dụng trong một thời gian xác định có cần thiết hủy bỏ để giải phóng tài nguyên hay không? Chẳng hạn: nếu giới hạn số lượng tối thiểu là 4, số lượng tối đa là 100. Điều này có nghĩa là có ít nhất 4 đối tượng sẵn dùng trong Object Pool, tối đa là 100 đối tượng được tạo ra và được quản lý trong pool. Đối tượng không được sử dụng sau khoảng thời gian timeout, thì sẽ được hủy bỏ cho tới khi còn lại 4 đối tượng.

Khi một client giữ một object quá lâu mà không trả về object pool thì có cần thiết set timeout để trả về cho đối tượng khác sử dụng không? Chẳng hạn: một client1 cần sử dụng object trong khoảng thời gian 10 phút, một client2 cần sử dụng trong 20 giây. Khi client1 yêu cầu sử dụng trước, nếu không set timeout thì client2 phải chờ 10 phút mới được sử dụng trong 20 giây.

Khi một client chờ quá lâu thì sẽ xử lý như thế nào? Chờ đến khi có tài nguyên sử dụng hay sẽ throw ngoại lệ.

Làm gì khi Pool không chứa đối tượng nào?

Chúng ta có thể sử dụng một trong ba chiến lược để xử lý một yêu cầu từ client khi trong object pool không chứa đối tượng nào (rỗng):

- **Tạo mới:** khởi tạo thêm một đối tượng mới và trả về cho client nếu nó chưa vượt quá số lượng đối tượng được phép khởi tạo.
- **Chờ**: Trong một môi trường đa luồng, một object pool có thể block các yêu cầu từ client cho đến khi một luồng khác trả về một đối tượng có thể sử dụng vào object pool.
- **Trả lỗi:** không cung cấp một đối tượng và ngay lập tức trả lại lỗi cho client. Hoặc chờ một khoảng thời gian (timeout) và trả lại lỗi cho client.

Đảm bảo trạng thái của object không bị thay đổi khi trả về Object Pool?

Khi triển khai mô hình Object pool, chúng ta phải cẩn thận để đảm bảo rằng trạng thái của các đối tượng quay trở lại object pool phải được đặt ở trạng thái hợp lý cho việc sử dụng tiếp theo của đối tượng. Nếu không kiểm soát được điều này, đối tượng sẽ thường ở trong một số trạng thái mà chương trình client không mong đợi và có thể làm cho chương trình client lỗi (failed), không nhất quán, rò rỉ thông tin.

Lợi ích của Object Pool Pattern là gì?

- Tăng hiệu suất của ứng dụng.
- Hiệu quả trong một vài tình huống mà tốc độ khởi tạo một object là cao.
- Quản lý các kết nối và cung cấp một cách để tái sử dụng và chia sẻ chúng.
- Có thể giới hạn số lượng tối đa các đối tượng có thể được tạo ra.

Sử dụng Object Pool Pattern khi nào?

Objects pool được sử dụng khi:

- Khi cần tạo và hủy một số lượng lớn các đối tượng trong thời gian ngắn, liên tục.
- Khi cần sử dụng các object tương tự thay vì khởi tạo một object mới không có kiểm soát.
- Các đối tượng tốn nhiều chi phí để tạo ra.
- Khi có một số client cần cùng một tài nguyên tại các thời điểm khác nhau.

Một vài thư viện sử dụng Object Pool trong Java:

- Thread Pool
- Connection Pool: DBCP, c3p0, UCP, HikariCP, …
- HTTP Client connection pool, …

Như vậy là chúng ta đã đi qua một số Design Pattern về Creational pattern. Trong các bài viết tiếp theo chúng ta sẽ cùng tìm hiểu về Structuaral Pattern.

Bài 11. Singleton có thực sự dễ

Singleton: dùng hay không?

Nhân tiện mọi người có vẻ hào hứng về chủ đề nho nhỏ này và đang viết được, viết nôt kẻo ít hôm lại lười.

Trả lời ngắn gọn là dùng. Dùng chứ. Nhưng dùng thế nào cho đúng thì phải hiểu rõ một chút. Bài viết này cố gắng cung cấp nhiều góc nhìn để cân nhắc.

Vì sao ra đời?

Trong GoF, Singleton được đưa ra với mục đích sau: *"Ensure a class only has one instance, and provide a global point of access to it." – "Đảm bảo một class chỉ có duy nhất một instance, và cung cấp một điểm truy cập duy nhất trên toàn cục tới instance."*

Cái gì hay?

Hay thì rõ rồi, nó cung cấp ý tưởng của việc *một instance duy nhất,* điều mà chúng ta gặp trong nhiều bài toán thực tế: một ứng dụng được khởi tạo, một cấu hình hệ thống, một logger… Tôi cá là nhiều người không biết tới Singleton thì không biết giải quyết bài toán này thế nào.

- **Runtime:**
    - Không giống như static trong class, object và các giá trị trong đó chỉ được khởi tạo khi cần thiết. Memory và cả CPU đều được tiết kiệm.
    - Cho phép chủ động quản lý life cycle, giải phóng khi cần thiết.
- **Design:**
    - Abstract hơn sử dụng static trong class;
    - Có khả năng thừa kế;
    - Có thể kếp hợp với những design pattern khác.

Cái gì dở?

Dở thì cũng có, bởi vậy mới có nhiều tranh luận.

- **Tư tưởng**: Singleton trong GoF dở cơ bản về tư tưởng bởi nó giải quyết 2 bài toán khác nhau (dù có vẻ liên quan): "*Ensure a class only has one instance, **and** provide a global point of access to it."*
    - *Một class có duy nhất một instance;*
    - *Cung cấp một điểm truy cập duy nhất trên toàn cục tới instance.*
    - Tác giả đã vô tình kèm cả lời giải trong bài toán với giả định: *để có một điểm truy cập toàn cục duy nhất thì chỉ có duy nhất*

một instance được tạo ra từ một class. Bài toán "một điểm truy cập" có thể được giải quyết bởi Facade, Wrapper… không nhất thiết phải là Singleton.

- **Design:**
    - Coupling: Vì là global state nên các thành phần bị gắn chặt với nhau;
    - Khó / không thể viết test;
    - Viết dễ sai sót, để lại lỗ hổng (xem bài trước Singleton có thực sự dễ?).
- **Runtime:**
    - Không "thân thiện" với theading.

Nên dùng thế nào?

Như vậy, ta thấy rằng đa phần những thứ dở của design pattern này là ở tư tưởng *global state*. Vậy nên những ai yêu thích functional programming thì sẽ rất anti-pattern này. Cũng đúng thôi, GoF sinh ra cho OOP, không phải FP. Và thời đại của GoF (1994) cũng không quan tâm nhiều tới concurrency – bài toán trở thành rất cơ bản trong thời đại này. Bởi vậy, việc sử dụng Singleton có chút thay đổi. Có 3 điều cần chú ý:

- *Concurrency:* Global state là điều tệ hại cho concurrency. Hãy giảm thiểu tối đa nếu có thể. Global state bẻ cong cách suy nghĩ về luồng và gây mệt mỏi cho việc debug trong concurrency. Nếu bạn muốn thiết kế hệ thống tối ưu hiệu năng và concurrency thì không sử dụng Singleton cũng là một ý hay.
- *Memory:*
    - *Lưu cái gì?* Global state cũng là một ý hay vì khiến việc thiết kế và lập trình dễ dàng hơn, nó chỉ không hay khi bạn không cân nhắc tới nên lưu gì. Rất nhiều thứ có thể nhìn dưới góc độ *một instance* nếu chúng ta không có khả năng khái quát hoá. Logger là Singleton không? Hay có errorlog, accesslog? Database là single thì lưu cả database? Hay chỉ connection? Hay chỉ connectionString? *Lưu ít nhất có thể.*
    - *Lưu khi nào?* Nhiều người hay gắn Singleton với life cycle của cả ứng dụng, kèm theo việc lưu trữ nhiều, hoặc giữ strong reference dẫn đến GC không thể hoạt động; sớm muộn gì cũng gây ra memory leak. *Khởi tạo muộn nhất có thể, giải phóng sớm nhất có thể.*
- *Language*: Cần lưu ý cách sử dụng trong từng ngôn ngữ (xem bài trước Singleton có thực sự dễ? và Singleton:threading() in Java), mỗi ngôn ngữ khác nhau sẽ có vấn đề khác nhau. Dù design pattern là mức thiết kế song đừng mang nguyên cách cài đặt từ ngôn ngữ này sang ngôn ngữ khác, hãy nhìn vào diagram và đặc trưng ngôn ngữ.

Trên đây là một số góc nhìn, gợi ý để bạn dùng Singleton đúng hơn. Không có đúng hay sai khi dùng Singleton, dùng đúng hay không mới là vấn đề.

Bài 12. Xử lý theading khi cài đặt Singleton

singleton.threading() in java

Bài trước về Singleton có thực sự dễ, tôi nhận được vài comment rất chuẩn về cách cài đặt xử lý với theading. Vì bài trước tập trung nói về Singleton và các vấn đề có thể gặp phải với *reflection, threading, serializable*… nên tất cả những giải pháp đưa ra chỉ dừng ở mức ý tưởng không làm bạn đọc phân tâm. Bài này nói rõ hơn về xử lý theading khi cài đặt Singleton.

Tôi sẽ đi vào cài đặt "chuẩn" đã đưa cuối bài Singleton có thực sự dễ trước:

```
public class AppConfig {
    private static volatile AppConfig self;

    private AppConfig() {
        if (self != null) {
            throw new UnsupportedOperationException("Use
getInstance()");
        }
    }

    public static synchronized AppConfig getInstance() {
        if (self == null) {
            self = new AppConfig();
        }
        return self;
    }
}
```

Khi phân tích về threading trong bài trước bạn thấy tôi viết *"Vậy nên cần phải synchronized việc tạo object."*. Nếu để ý kỹ, có 2 phần thay đổi:

- *synchronized* được thêm vào method getInstance()
- *volatile* được thêm vào *self* (phần này được tôi lờ đi)

Có mấy vấn đề ở đây liên quan tới threading bạn nên biết.

synchronized cần phải tối thiểu

Cần phải khẳng định ngay rằng cách viết trên cho hiệu năng rất thấp. *synchronized* sẽ thực hiện việc lock method khiến các thread không thể invoke method song song, chúng buộc phải invoke tuần tự. Nếu có 100 thread gọi *getInstance()*, thread thứ 100 sẽ nhận được object AppConfig sau khi 99 thread trước đã hoàn thành. Sẽ tốt hơn nếu object AppConfig được tạo ra bởi thread đầu tiên, 99 thread còn lại có thể đồng thời được nhận lại object AppConfig.

Double check locking nên được sử dụng trong trường hợp này. *Double check locking* là một design pattern phổ biến trong bài toán về threading, kiểm tra lock trước khi thực sự lock method (theo nguyên lý *return as soon as possible*).

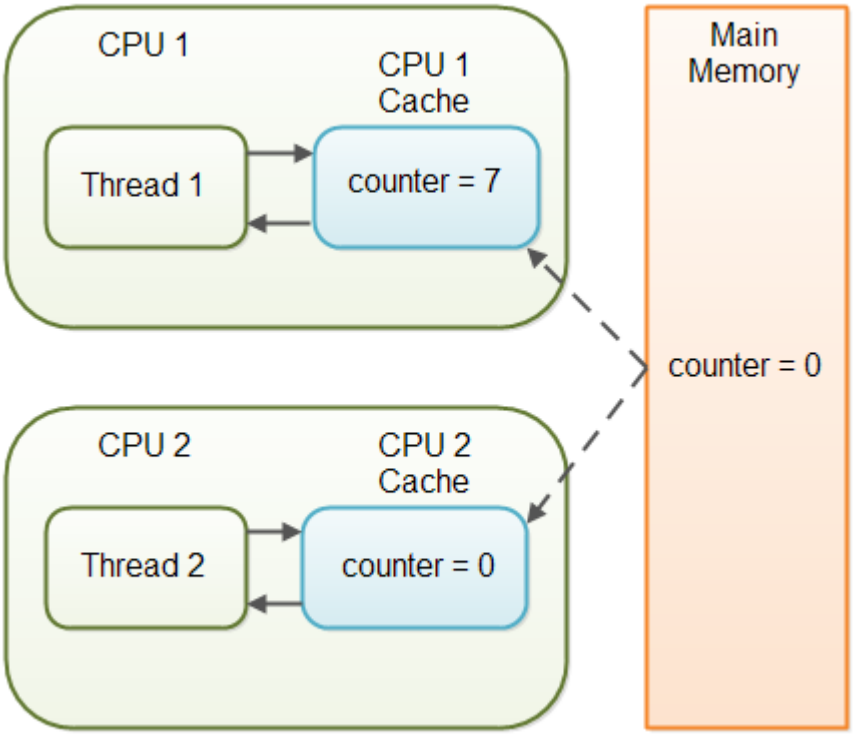
```
public static synchronized AppConfig getInstance() {
    if (self == null) {
        synchronized (AppConfig.class) {
            if (self == null) {
                self = new AppConfig();
            }
        }
    }
    return self;
}
```

if (self == null) được thực hiện 2 lần, nên pattern này được gọi là *double check*. Đừng bỏ câu lệnh thứ 2 nếu bạn không muốn một hệ quả tai hại (100 object được tạo ra… tuần tự, tại sao?). Đấy là lý do tôi muốn synchronized cả method để ai đọc cũng hiểu.

Chỉ lock những statement thực sự cần thiết và không nhiều hơn mức cần thiết là nguyên lý cơ bản của threading.

synchronized vẫn chưa đủ

Ngay cả khi bạn không bỏ đi câu lệnh *if (self == null)* nào, vẫn có thể có hàng chục object AppConfig được tạo ra.



Hãy nhớ, máy tính có đến mấy loại bộ nhớ, nên việc đảm bảo dữ liệu đồng nhất không dễ. Lan man 1 chút về kiến trúc máy tính qua hình trên:

- Main memory là RAM.
- CPU xử lý dữ liệu được nạp vào bộ nhớ của CPU, không phải RAM. Bởi vậy dữ liệu được sao chép theo trình tự RAM -> Lx ->… -> L2 -> L1 cache. Trong lúc đấy hàng triệu thứ đã xảy ra.
- Tưởng tượng, counter là self, khi 2 thread bắt đầu, self = null, được copy vào cache. Sau đó dù có *synchronized* thì 2 CPU vẫn lấy giá trị trên 2 cache độc lập để ra quyết định. Kể cả sau khi xử lý xong, giá trị vẫn được giữ trên cache đến khi thực sự cần đưa xuống RAM.

Vậy là ta phải dùng *volatile* để giá trị của biến luôn được tham chiếu tới RAM khi thay đổi. Vậy đây là giá trị duy nhất.

Còn cách nào khác không?

Nếu để ý, bạn thấy *volatile* trở nên vô dụng khi ta lock toàn bộ method. Tức là cách cài đặt của tôi ở cuối bài trước là rất ngớ ngẩn. Thật ra tôi cố tình viết vậy, ai không hiểu về threading thì vẫn có giải pháp chạy đúng; người hiểu biết về threading thì có hint đi tiếp.

Tôi không muốn đi sâu vào threading vì vấn đề này phụ thuộc ngôn ngữ, nền tảng. Hy vọng bạn hiểu rằng một design pattern đơn giản như Singleton cũng cần cài đặt cẩn thận và hiểu biết sâu sắc.

Kết

Có 2 vấn đề muôn thuở của lập trình: performance và memory. Bài này đề cập tới performance. Một số comment khác về việc không quản lý tốt life cycle khi sử dụng Singleton gây ra memory leak. Tôi sẽ cố gắng viết sau.

Và đừng hỏi tại sao lập trình lại phải biết kiến trúc máy tính. Bởi vì code không chạy trên máy tính thì chạy ở đâu? Nền tảng quan trọng lắm.

Tài nguyên tham khảo

https://viblo.asia/p/laravel-design-patterns-series-builder-manager-pattern-part-1-ZnbRlDJ3R2Xo

https://viblo.asia/p/laravel-design-patterns-series-factory-pattern-part-2

https://viblo.asia/p/tim-hieu-ve-proxy-pattern-strutural-pattern-4P856onOKY3

https://dev.to/ahmedash95/design-patterns-in-php-decorator-with-laravel-5hk6

https://itnext.io/repository-design-pattern-done-right-in-laravel-d177b5fa75d4

https://viblo.asia/p/design-pattern-template-method-pattern-QpmlervN5rd

https://viblo.asia/p/repository-pattern-trong-laravel-gGJ59jPaKX2

https://gurunh.com/2018/05/singleton-threading-in-java/

https://nguyenbinhson.com/2020/08/15/huong-dan-java-design-pattern-object-pool/

https://sourcemaking.com/design_patterns/object_pool

https://www.oodesign.com/object-pool-pattern.html

https://www.javatpoint.com/object-pool-pattern

https://ejbvn.wordpress.com/category/week-2-entity-beans-and-message-driven-beans/day-09-using-jdbc-to-connect-to-a-database/