Tạp_Chí
{LậpTrình ():}

# Tạp chí LẬP TRÌNH

tapchilaptrinh.vn

Số phát hành Vol.6 | 08 2020

# DESIGN PATTERN

Vỡ lòng về bộ nguyên tắc thiết kế SOLID

Giải thích mô hình MVC thông qua ... cốc trà đá

Factory Method trong thực tiễn

# MỤC LỤC



VOL. 6

# LỜI MỞ ĐẦU

*Quý bạn đọc thân mến*,

Trong thế giới của các lập trình viên, có những thứ đã trở thành kinh điển, chẳng hạn như những thuật toán kinh điển, những kiến trúc kinh điển, những công nghệ kinh điển, những giao thức kinh điển,... Design Pattern cũng có thể được xếp vào một trong những mảng kinh điển mà các lập trình viên già rơ thường bàn luận đến. Ấn phẩm Tạp chí Lập trình lần này sẽ đề cập đến chủ đề Design Pattern với mong muốn cung cấp cho các lập trình viên một cái "cớ" để một lần nữa tìm hiểu thêm về một mảng rất quan trọng này.

Design Pattern không phải một chủ đề dễ dàng đối với những người mới chân ướt chân ráo làm lập trình. Để có thể hiểu được cặn kẽ và áp dụng được các Design Pattern trong các giải pháp của mình, một lập trình viên cần phải có hiểu biết và kỹ năng tốt về những nền tảng của lập trình hướng đối tượng và thiết kế hướng đối tượng. Mặc dù vậy, ấn phẩm lần này cũng có một số bài viết có hơi hướng giới thiệu dành cho những người mới lần đầu tiếp xúc với khái niệm Design Pattern.

Thế giới của Design Pattern nói riêng và của Phân tích và Thiết kế Hướng đối tượng (OOAD) nói chung đều rất rộng lớn và luôn luôn thay đổi. Khó để có thể nhanh chóng làm chủ được thế giới đó và luôn ở trạng thái cập nhật nếu chúng ta không dành nhiều thời gian và nỗ lực để tìm hiểu, nghiên cứu và trau dồi những mảng kiến thức và kỹ năng trong thế giới đó. Ấn phẩm lần này cũng kỳ vòng sẽ giúp mọi người một lần nữa tích lũy thêm được những trải nghiệm với Design Pattern.

Ban biên tập cũng rất mong muốn nhận được sự đóng góp nội dung của các lập trình viên về chủ đề này để có thể chia sẻ trên website của Tạp chí Lập trình hoặc đưa vào các ấn phẩm tiếp theo trong tương lai.

Cảm ơn và chúc quý bạn đọc nhiều tiến bộ.

---

**Ban biên tập Tạp chí Lập trình**


# VỠ LÒNG VỀ BỘ
# NGUYÊN TẮC THIẾT KẾ SOLID

**Nguyễn Bình Sơn**

## Kiến trúc

Hôm nay phần mềm của bạn vẫn phục vụ tốt với bộ tính năng mà nó cung cấp, ngày mai người dùng có thể sẽ yêu cầu bạn sản xuất tính năng mới, và họ sẽ luôn làm thế. Bạn đang sản xuất dở phần mềm, hôm nay bạn có 50% tính năng và ngày mai sẽ làm 51%. Chúng ta gọi công cuộc thêm tính năng mới này là "mở rộng" phần mềm.

Khi viết phần mềm, chúng ta luôn phải quan tâm đến việc duy trì khả năng mở rộng của phần mềm. Dân sản xuất phần mềm truyền nhau hài ngôn sau để nói về việc mất khả năng mở rộng:

> Chúng ta sẽ dành 10% thời gian đầu tiên của dự án để phát triển 90% tính năng, và chúng ta sẽ dùng 90% thời gian sau đó để phát triển 10% tính năng cuối cùng (mà không biết có xong không).

Khả năng mở rộng phần mềm cũng giống như khả năng xây thêm tầng cho một ngôi nhà. Khả năng đó trước hết đến từ hai thứ: vật liệu tốt, kiến trúc tốt, và nền móng tốt. Đối ứng với phần mềm thì vật liệu tốt chính là mã sạch và kiến trúc tốt là thiết kế.

## Nguyên tắc thiết kế

Mã sạch cũng giống như gạch tốt. Dĩ nhiên gạch tốt là một trong những điều đầu tiên chúng ta cần quan tâm đến khi xây dựng một ngôi nhà. Nhưng để ngôi nhà được vững chắc và duy trì được khả năng sử dụng cao khi xây dựng thêm các tầng mới thì chỉ gạch tốt là chưa đủ. Chúng ta cần đến thiết kế. Và như thế chúng ta tìm đến các nguyên tắc SOLID.

Nếu như các nguyên tắc Mã sạch hướng dẫn chúng ta viết nên các hàm và các class tốt, các nguyên tắc SOLID chỉ ra cách đặt các hàm và cấu trúc dữ liệu và các class, cũng như cách các class nội kết với nhau. Dụng ngôn "class" không có nghĩa rằng SOLID chỉ áp dụng cho phần mềm hướng đối tượng. Class chỉ thuần túy là một tập các hàm và các dữ liệu được nhóm lại với nhau. Bất kỳ phần mềm nào cũng có những nhóm như thế, cho dù chúng có được gọi là class hay không. Và SOLID nhắm đến những nhóm này.

Mục tiêu của SOLID là giúp tạo ra những cấu trúc phần mềm cấp trung mang tính uyển chuyển, dễ hiểu, và có khả năng dùng làm cơ sở để tạo thành những component có thể dùng chung cho nhiều hệ thống phần mềm. "Cấp trung" nói lên thực tế rằng chúng ta dùng đến SOLID khi tập trung vào cách kết cấu các khối và cấu phần trong phần mềm, thay vì đi vào chi tiết mã lệnh.

## SOLID

Các nguyên tắc SOLID không được hình thành ngay một lúc mà trải qua một lịch sử dài. Trong quá trình tìm kiếm các nguyên tắc cốt lõi khi thiết kế các cấu trúc cấp trung, các thợ cả trong nghành thủ công phần mềm đã tạo ra nhiều bộ nguyên tắc khác nhau. Trong số có tới nay còn lại năm nguyên tắc dễ hiểu, thanh nhã, và đủ tốt; mà nếu sắp xếp theo đúng thứ tự thì chúng ta sẽ có tính từ solid (rắn chắc).

Sơ bộ, SOLID gồm 5 nguyên tắc như sau:

1. **Single Responsibility Principle** – nguyên tắc này yêu cầu mỗi khối phần mềm chỉ có một lý do để thay đổi.
2. **Open-Closed Principle** – phần mềm luôn phải ở trạng thái sao cho có thể thay đổi hành vi của chúng bằng cách thêm mã mới thay vì sửa mã cũ.
3. **Liskov Substitution Principle** – phần mềm luôn phải ở trạng thái mà mỗi thành phần đều có thể thay thế mà không ảnh hưởng đến hành vi của nó.
4. **Interface Segregation Principle** – tránh tạo ra quan hệ phụ thuộc với những thứ không dùng đến
5. **Dependency Inversion Principle** – mã triển khai chính sách cấp cao không được phụ thuộc vào triển khi chi tiết ở mức thấp. Thay vì thế chi tiết nên phụ thuộc vào chính sách.

Các nguyên tắc này đã được mô tả trong rất nhiều ấn phẩm trong suốt nhiều năm. Bài viết này sẽ đi vào ý nghĩa thiết kế đằng sau chúng, thay vì tiếp tục các tranh luận chi tiết trong khái niệm.

## Nguyên tắc Đơn Trách Nhiệm

Đơn Trách Nhiệm, hay Trách Nhiệm Duy Nhất, có lẽ là nguyên tắc được nghe đến nhiều nhất và bị hiểu sai nhiều nhất trong số các nguyên tắc SOLID. Cứ mười anh lập trình viên thì phải đến hơn chín anh cho rằng nguyên tắc này phát biểu điều gì đó liên quan đến mỗi hàm (hay tệ hơn – mỗi class???) chỉ được làm một việc.

Thật ra không thể trách được, quả thật là có một nguyên tắc như thế. Một hàm chỉ được phép làm một và chỉ một, việc. Chúng ta áp dụng nguyên tắc đó khi thực hiện tái cấu trúc những hàm lớn thành những hàm nhỏ hơn; chúng ta áp dụng nguyên tắc đó lại mức thấp của công việc viết mã. Nhưng đó không phải là một trong các nguyên tắc SOLID, càng không phải Nguyên Tắc Đơn Trách nhiệm.

Trong lịch sử, Nguyên Tắc Đơn Trách Nhiệm từng được phát biểu như sau:

> Một module chỉ được có một và chỉ một lý do để thay đổi.

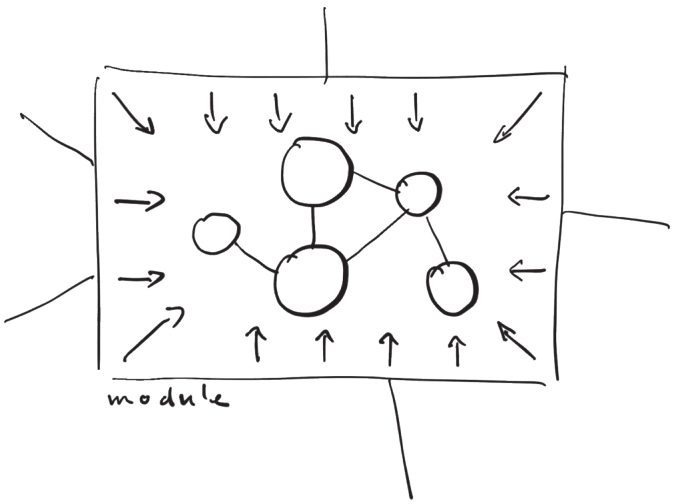
Các sản phẩm phần mềm thay đổi là để đáp ứng người dùng và các bên liên quan. Họ là các lý do để thay đổi mà nguyên tắc nói tới. Thế nên Đơn Trách Nhiệm có thể được phát biểu lại như sau:

> Một module chỉ phải chịu trách nhiệm thay đổi trước một và chỉ một người dùng hay bên liên quan.

Những từ "người dùng" và "bên liên quan" không thực sự đúng cho lắm. Đôi khi có nhiều hơn một người dùng hay bên liên quan muốn hệ thống thay đổi nhưng theo cùng một cách. Cái chúng ta thực sự muốn nói đến là một nhóm — một hay nhiều người ra yêu cầu thay đổi. Chúng ta gọi nhóm đó là một tác nhân. Theo đó, định nghĩa cuối cùng của Nguyên Tắc Đơn Trách Nhiệm sẽ là:

> Một mô-đun chỉ phải chịu trách nhiệm trước một, và chỉ một, tác nhân.

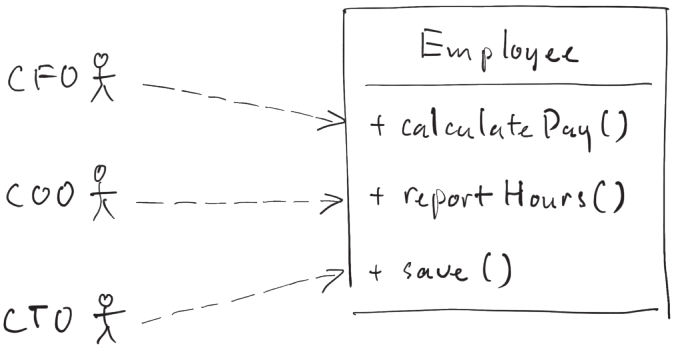
Tiếp theo là đến ý nghĩa của khái niệm mô-đun. Định nghĩa đơn giản nhất của mô-đun là một tập tin mã nguồn. Đa phần chúng ta có thể dùng định nghĩa đó. Có vài ngôn ngữ và môi trường phát triển đặc thù không sử dụng tập tin để chứa mã nguồn của chúng. Tuy nhiên dù trong bất kỳ môi trường nào cũng luôn tồn tại các nhóm cấu trúc dữ liệu và hàm tụ về một hướng, mà chúng ta gọi là sự ngưng tụ, sự ngưng tụ cố kết các thành phần rời rạc lại với nhau và tạo thành mô-đun. Nói tóm lại, mô-đun là một tập hợp ngưng tụ các hàm và cấu trúc dữ liệu.




Giờ chúng ta sẽ làm sáng tỏ Nguyên Tắc Đơn Trách Nhiệm thông qua một vài ví dụ về sự vi phạm nó.

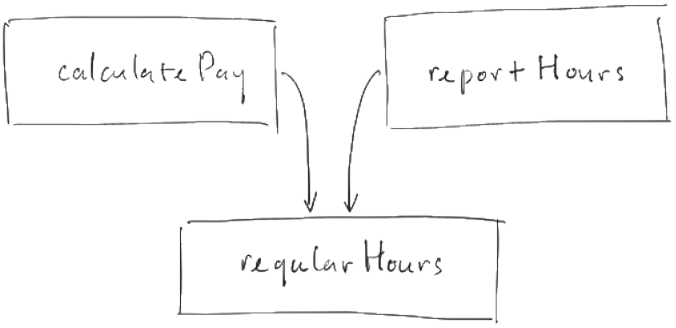
## Triệu chứng: xung đột nghiệp vụ

Trong ví dụ dưới đây có lớp `Employee` của một chương trình tính lương. Nó có các phương thức `calculatePay()` được sử dụng bởi phòng kế toán để tính lương, phục vụ giám đốc tài chính; `reportHours()` được sử dụng bởi phòng nhân sự để tính ngày công, phục vụ giám đốc nhân sự; và `save()` được sử dụng bởi các quản trị viên cơ sở dữ liệu để lưu thông tin nhân viên, phục vụ giám đốc công nghệ. Các giám đốc rõ là các tác nhân khác nhau, và lớp này đã vi phạm *Nguyên Tắc Đơn Trách Nhiệm*.




Bằng việc ngưng tụ cả ba phương thức vào cùng một lớp, các nhà phát triển đã ràng buộc ba tác nhân khác nhau với nhau. Một yêu cầu thay đổi nào đó từ phía COO sẽ có thể gây ảnh hưởng tới nghiệp vụ của CTO.

Lấy ví dụ, `calculatePay()` và `reportHours` sử dụng cùng một công thức để quy số giờ làm việc thành giờ làm việc tiêu chuẩn (những giờ làm việc quá giờ sẽ có hệ số cao hơn 1 chẳng hạn). Và bởi khử mã lặp nên các nhà phát triển đã đặt công thức này vào một hàm dùng chung tên là `regularHous():`




Rồi một ngày, CFO quyết định rằng công thức cần phải thay đổi một chút, nhưng đội ngũ của COO thì không có nhu cầu với thay đổi này, do họ dùng con số giờ làm việc tiêu chuẩn cho những mục đích khác nhau.

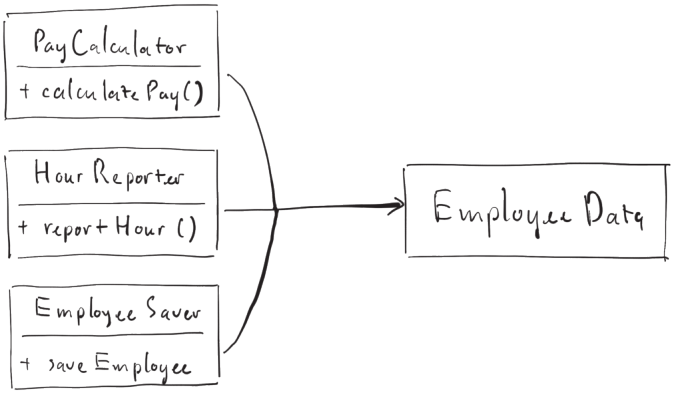
Người lập trình viên được giao nhiệm vụ triển khai thay đổi không nhận ra rằng `regularHours` cũng được sử dụng bởi `reportHours`, anh ta đã thực hiện sửa đổi, kiểm thử cẩn thận, đội ngũ của CFO đã kiểm tra, chức năng hoạt động như mong muốn, được nghiệm thu và được đưa vào thực tế.

Đội ngũ của COO phải rất lâu sau mới nhận ra rằng những con số mà họ dựa vào để báo cáo và ra quyết định đang có vấn đề. Và tới khi đó thì hậu quả đã là rất nhiều tài nguyên và nỗ lực đã bị lãng phí.

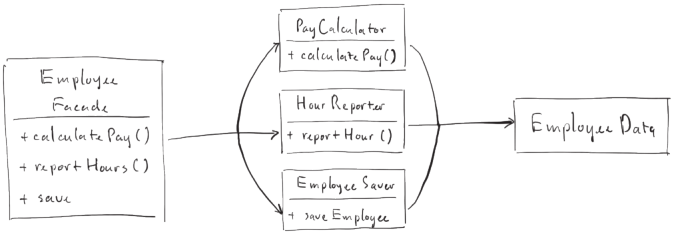
Bất kỳ ai có thâmtrong nghành cũng đều đã thấy những chuyện tương tự. Chúng xảy ra bởi vì chúng ta đã đặt những mã nguồn phụ thuộc vào những tác nhân khác nhau lại gần với nhau quá. Nguyên Tắc Đơn Trách Nhiệm dặn chúng ta đặt chúng xa ra.

## Giải pháp

Có nhiều giải pháp khác nhau cho vấn đề này. Mỗi giải pháp lại đặt các hàm vào các lớp khác nhau. Có lẽ cách dễ nhận thấy nhất đó là tách dữ liệu khỏi các hàm, và đặt các hàm chức năng vào trong những lớp chỉ chứa đủ mã nguồn cần thiết để chức đó hoạt động. Các lớp mang chức năng không biết về sự tồn tại của nhau, do đó tránh được bất kỳ sự xung đột nghiệp vụ nào.

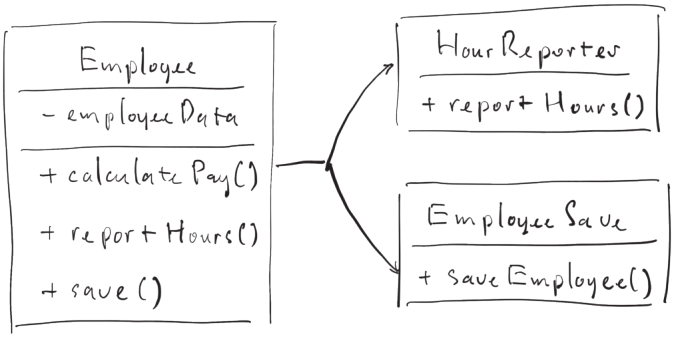



Nhược điểm của giải pháp này chính là việc lập trình viên sẽ phải quan tâm đến những ba lớp khác nhau. Cách giải quyết là sử dụng mẫu thiết kế Facade:




Lớp `EmployeeFacade` chứa rất ít mã. Nó chỉ chịu trách nhiệm khởi tạo và ủy thác công việc cho các lớp mang chức năng.

Một số nhà phát triển thích giữ những nghiệp vụ quan trọng nhất được gần với dữ liệu hơn. Điều này có thể được thực hiện bằng cách giữ phương thức quan trọng nhất trong lớp `Employee` ban đầu, và dùng lớp này làm facade cho các hàm chức năng nhỏ hơn.




Bạn có thể sẽ cảm thấy muốn chối bỏ các giải pháp trên bởi việc mỗi lớp chỉ chứa một hàm trông không được tự nhiên lắm. Thực tế điều này hiếm khi xảy ra. Mỗi lớp luôn chứa cả các hàm riêng tư, chẳng hạn để phục vụ cho chức năng tính lương, tính giờ làm hay lưu tồn dữ liệu, và số lượng của chúng thường không ít.

# Nguyên Tắc Đóng/Mở

Nguyên Tắc Đóng/Mở được phát biểu như sau:

> Một tạo tác phần mềm nên mở cửa cho sự mở rộng nhưng đóng lại với những sửa đổi.

Nói cách khác, một tạo phẩm phần mềm phải hành xử theo lối có khả năng mở rộng mà không phải sửa đổi tạo phẩm đó. Đây, cơ bản là lý do khiến chúng ta nghiên cứu về kiến trúc phần mềm. Nếu chỉ một vài mở rộng đơn giản theo yêu cầu cũng kéo theo những thay đổi lớn trên phần mềm, thì rõ ràng các kiến trúc sư của hệ thống phần mềm đó đã trên bờ vực mất kiểm soát kiến trúc đến nơi.

Hầu hết người học thiết kế phần mềm đều coi Nguyên Tắc Đóng/Mở như một nguyên tắc hướng dẫn khi thiết kế các lớp và mô-đun. Nhưng không chỉ thế, nguyên tắc này còn phát huy ý nghĩa của nó khi chúng ta thiết kế các cấu phần của hệ thống phần mềm.

**Ví dụ: Hệ thống báo cáo tài chính**

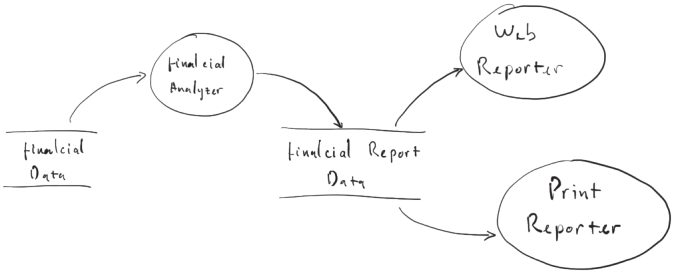
Hãy tưởng tượng chúng ta có một hệ thống hiển thị thông tin tài chính tổng hợp lên trang web. Danh mục thông tin rất dài, và có thể cuộn để xem. Các số âm được hiển thị bằng màu đỏ.

Bây giờ các bên liên quan yêu cầu một khối thông tin tương tự nhưng ở định dạng sẵn sàng để in lên máy in đen trắng. Danh mục được phân trang, và các số âm được bao quanh bởi cặp dấu ngoặc đơn.

Rõ ràng phải viết thêm mã mới, nhưng bao nhiêu mã cũ sẽ phải thay đổi? Hệ thống phần mềm được thiết kế tốt sẽ giảm số lượng mã cũ phải sửa xuống tối thiểu. Lý tưởng nhất là không có.

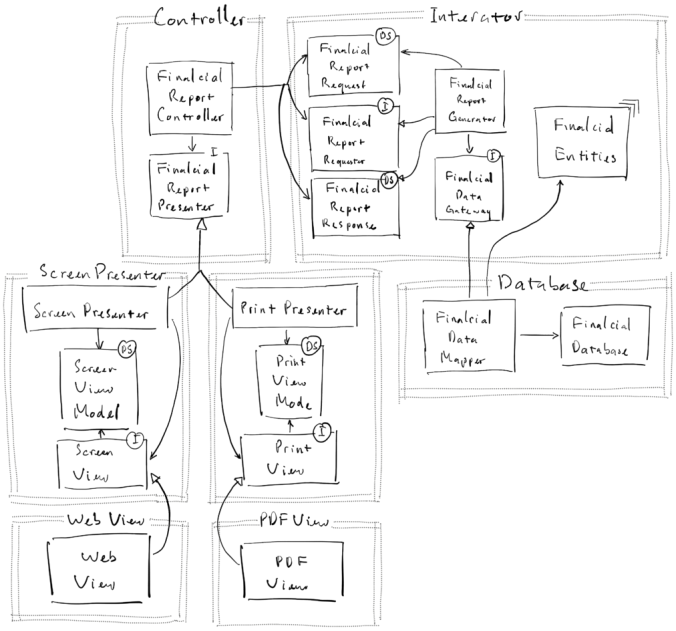
Làm thế nào? Bước đầu tiên là tổ chức phân tách những cấu phần chịu trách nhiệm bởi những tác nhân khác nhau, sau đó tổ chức các mối phụ thuộc giữa các cấu phần đó một cách hợp lý.

Với Nguyên Tắc Đơn Trách Nhiệm, chúng ta có được cái nhìn tổng thể về luồng dữ liệu như dưới đây. Thủ tục Phân tích Tài chính sẽ tạo ra Dữ liệu Báo cáo, thứ sau đó được định dạng cho phù hợp bởi hai Trình tạo Báo cáo.



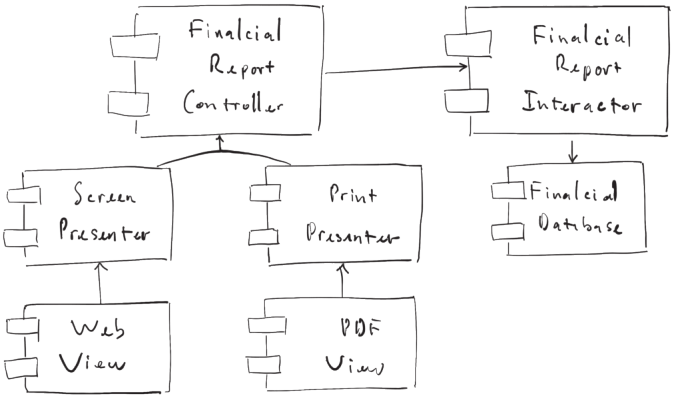

Bước tiếp theo là tổ chức các mối phụ thuộc trong mã nguồn sao cho các thay đổi từ một trong các trách nhiệm không kéo theo thay đổi trên các trách nhiệm còn lại. Đồng thời cũng phải đảm bảo rằng hành vi có thể được mở rộng mà không cần phải sửa đổi mã nguồn của hành vi cũ.

Chúng ta tổ chức các tiến trình vào các lớp và phân bổ các lớp đó vào các cấu phần, được thể hiện bằng các khu vực với đường bao kép, như trong hình dưới đây. Các cấu phần bao gồm Controller, Interator, Database, các Presenter, và các View.




Trong sơ đồ, các ký hiệu `I` là các interface, `DS` là các cấu trúc dữ liệu, các mối quan hệ dùng đến được thể hiện bởi các mũi tên hở, và các quan hệ thi hành hay kế thừa được thể hiện bởi các mũi tên kín.

Hãy để ý vào các mũi tên cắt ngang qua đường biên giữa các cấu phần, chúng là những mũi tên đơn hướng. Những mũi tên hướng về những cấu phần mà chúng ta muốn bảo vệ khỏi sửa đổi.




Nếu cấu phần A cần được bảo vệ khỏi những thay đổi từ cấu phần B, thì cấu phần B nên phụ thuộc vào cấu phần A.

Chúng ta muốn bảo vệ Controller khỏi những thay đổi của các Presenter. Chúng ta muốn bảo vệ các Presenter khỏi những thay đổi trong các View. Chúng ta muốn bảo vệ Interactor khỏi những thay đổi ở — bất cứ nơi nào khác. Các thay đổi trên View, Presenter, Controller, hay Database sẽ không làm thay đổi Interactor.

Tại sao Interactor lại được lưu tâm như vậy? Interactor chứa những quy tắc nghiệp vụ, chứa các chính sách ở cấp cao nhất của ứng dụng. Interactor là trung tâm và tất cả các cấu phần khác là thiết bị ngoại vi.

Controller là ngoại vi của Interactor, nhưng là trung tâm đối với Presenter; và tương tự như thế, Presenter là trung tâm của các View. Chúng ta gọi đây là hệ thống cấp bậc bảo vệ. View là cấu phần ở cấp thấp nhất, vì vậy chúng ít được bảo vệ nhất, Presenter có cấp cao hơn View nhưng thấp hơn so với Controller và Interactor.

Đó la sự hoạt động của Nguyên Tắc Đóng/Mở ở mức độ kiến trúc. Người kiến trúc sư thực hiện phân tách các chức năng bằng cách đặt ra những câu hỏi như thế nào, tại sao, bao giờ các thay đổi sẽ xuất hiện, sau đó tổ chức các chức năng riêng biệt vào một hệ thống phân cấp. Các cấu phần nằm ở mức cao hơn trong hệ thống được bảo vệ khỏi những thay đổi nằm ở cấu phần thấp hơn.

### Hướng điều khiển

Đừng hoảng hốt trước mức độ chi tiết quá mức của thiết kế ở trên, hầu hết các sự phức tạp trong sơ đồ là để đảm bảo rằng các mối phụ thuộc băng qua đường biên giữa các cấu phần được chỉ theo đúng hướng.

**Ví dụ:**

`<I>FinancialDataGateway` tồn tại giữa `FinancialReportGenerator` và `FinancialDataMapper` là để để đảo ngược sự phụ thuộc mà lẽ ra đã chỉ từ hướng Interactor đến Database. Tương tự với `<I>FinancialReportPresenter` và hai giao diện View.

### Ẩn giấu thông tin

Giao diện `<I>FinancialReportRequester` phục vụ một cho mục đích khác. Nó ở đó để đảm bảo `FinancialReportController` không bị biết quá nhiều về cấu trúc bên trong của Interactor. Nếu không có nó, Controller sẽ có quá nhiều mối phụ thuộc vào `FinancialEntities`.

Các phụ thuộc bắc cầu vi phạm nguyên tắc chung rằng các thực thể phần mềm không nên phụ thuộc vào những thứ chúng không trực tiếp sử dụng. Chúng ta sẽ gặp lại nguyên tắc đó khi nói về Nguyên Tắc Phân Tách Giao Diện.

Vì vậy, tuy nói rằng ưu tiên hàng đầu của chúng ta là bảo vệ Interactor khỏi các thay đổi của Controller, nhưng chúng ta cũng muốn bảo vệ Controller khỏi các thay đổi của Interactor bằng cách ẩn đi cấu trúc nội bộ của Interactor.

### Kết luận

Nguyên Tắc Đóng/Mở là một trong những lực lượng ngầm lèo lái kiến trúc của hệ thống phần mềm. Mục tiêu là làm cho hệ thống dễ dàng mở rộng mà không phải gây ra những thay đổi có tác động lớn. Mục tiêu này được thực hiện bằng cách quy hoạch hệ thống thành các cấu phần và phân bố các cấu phần đó vào một hệ thống có thứ bậc để bảo vệ các cấu phần cấp cao khỏi những thay đổi trong các cấu phần cấp thấp hơn.

## Nguyên Tắc Thay Thế Liskov

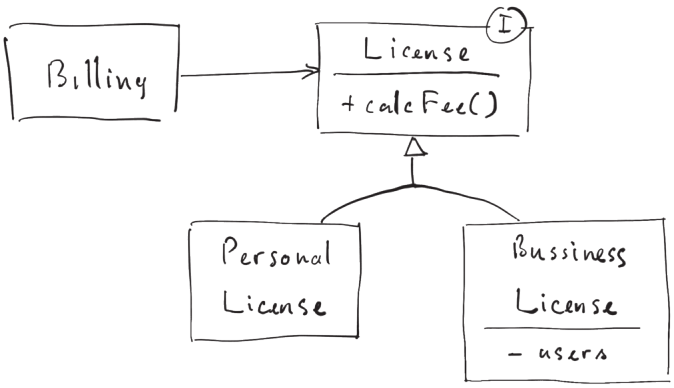
Năm 1988, Barbara Liskov, để định nghĩa một kiểu con, đã viết như sau, trong paper Thông báo SIGPLAN 23, 5 — Trừu tượng hóa và phân cấp dữ liệu (tháng 5 năm 1988).

> Điều muốn có ở đây là một cái gì đó giống như thuộc tính thay thế như sau đây: Nếu với mỗi đối tượng o1 có kiểu S tồn tại một đối tượng o2 có kiểu T sao cho với tất cả các chương trình P được xác định theo T thì hành vi của P không thay đổi khi lấy o1 thay thế cho o2, thì S là một kiểu con của T.

Ý tưởng này được gọi là Nguyên Tắc Thay Thế Liskov, để hiểu nó chúng ta hãy xem xét một số ví dụ.

### Dẫn dắt quan hệ kế thừa

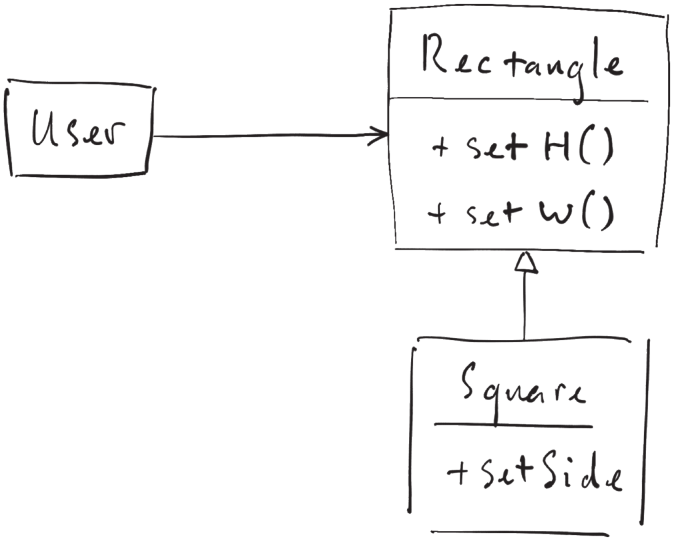
Tưởng tượng rằng chúng ta có một lớp có tên là `License` như trong hình dưới đây. Lớp này có một phương thức có tên là `calcFee()`, được gọi bởi ứng dụng Billing. `License` có hai lớp "con": `PersonalLicense` và `BusinessLicense`. Chúng dùng các thuật toán khác nhau để tính phí.




Thiết kế này phù hợp với Nguyên Tắc Thay Thế Liskov vì hành vi của ứng dụng Billing không phụ thuộc, theo bất kỳ cách nào, vào bất kỳ kiểu nào trong hai kiểu con mà nó sử dụng. Cả hai kiểu con đều có thể thay thế cho kiểu License.

## Vấn đề Hình vuông/Hình chữ nhật

Pha kinh điển về vi phạm Nguyên Tắc Thay Thế Liskov là vấn đề hình vuông/hình chữ nhật nổi tiếng (hoặc khét tiếng, tùy theo quan điểm của bạn), được thể hiện trong hình dưới đây.




Trong ví dụ này, Square không nên là một kiểu con của Rectangle vì width và và height của Rectangle có thể thay đổi độc lập; ngược lại, width và height của Square phải thay đổi cùng nhau. Nếu User tin rằng đối tượng đang dùng là một Rectangle, họ có thể sẽ rối tinh lên. Đoạn mã sau cho thấy tại sao:

```
Rectangle r = …
r.setW(5);
r.setH(2);
assert(r.area() == 10);
```

Nếu r ở trên là một Square thì phép assert sẽ nắm chắc thất bại. Với các sắp đặt quan hệ kế thừa như hiện tại, cách duy nhất để khử sự vi phạm Nguyên Tắc Thay Thế Liskov là thêm các cơ chế để User có thể phát hiện xem r có phải là một Square hay không. Điều này sẽ khiến hành vi của User bị phụ thuộc vào kiểu dữ liệu mà nó sử dụng, và mất khả năng thay thế sang các kiểu dữ liệu khác.

## Tác động mở rộng lên kiến trúc

Nguyên Tắc Thay Thế không chỉ như một hướng dẫn để dẫn dắt các mối quan hệ kế thừa. Nó còn có ảnh hưởng thấy rõ đến kiến trúc của hệ thống trên khía cạnh các giao diện và các triển khai của chúng.

Các giao diện có thể nằm ở nhiều dạng thức. Có thể là một giao diện Java, được triển khai bởi một số lớp; hay dưới dạng một lớp Ruby với khung các phương thức; hoặc cũng có thể là một tập hợp các dịch vụ đáp ứng cùng một giao diện REST... Dù là ở tình huống nào, Nguyên Tắc Thay Thế Liskov được áp dụng vì có những người dùng cần đến những giao diện được thiết kế tốt cũng như triển khai của chúng.

Cách tốt nhất để hiểu Nguyên Tắc Thay Thế Liskov từ quan điểm kiến trúc là xem xét những gì xảy ra với kiến trúc của một hệ thống khi nguyên tắc bị vi phạm.

## Dịch vụ gọi xe

Giả sử chúng ta đang xây dựng một cổng tích hợp cho nhiều dịch vụ gọi xe công cộng. Khách hàng sử dụng cổng của chúng ta để tìm phương tiện đi lại phù hợp nhất, không kể là từ công ty nào. Khi khách đã có quyết định, chúng ta sẽ chuyển hướng yêu cầu của họ đến tài xế thông qua một dịch vụ RESTful.

Bây giờ giả sử rằng URI của dịch vụ chuyển tiếp được đặt trong cơ sở dữ liệu các tài xế. Một khi hệ thống của chúng ta chọn được trình tài xế phù hợp với yêu cầu của người dùng, nó sẽ lấy URI từ cơ sở dữ liệu của tài xế và sau đó thực hiện chuyển tiếp.

Giả sử tài xế Bob tại công ty Purple Cab có URI như sau:

```
purplecab.com/do/Bob
```

Hệ thống của chúng ta sẽ nối thông tin từ khách hàng vào URI này và PUT:

```
purplecab.com/do/Bob
    /pickupAddress/24 Maple St.
    /pickupTime/153
    /destination/ORD
```

Dễ thấy rằng như vậy, tất cả các dịch vụ chuyển tiếp, bất kể công ty nào, đều sẽ có giao diện REST giống nhau. Chúng đều cần các tham số pickupAddress, pickupTime và destination.

Bây giờ, giả sử lập trình viên của công ty taxi Acme không đọc kỹ tài liệu. Họ viết tắt tham số destination. Acme là khách hàng lớn nhất của chúng ta và bà ngoại của CEO Acme là vợ mới của CEO của chúng ta, blah blah đại loại thế. Vậy là chúng ta phải thay đổi hệ thống của mình.

Rõ ràng là chúng ta cần thêm một trường hợp ngoại lệ. Yêu cầu chuyển tiếp đến trình điều khiển Acme sẽ phải sử dụng một bộ quy tắc khác với các trình điều khiển khác.

Và thế là một chỉ lệnh rẽ nhánh xuất hiện.

```
if (driver.getDispatchUri().
startsWith("acme.com")) {
 // ...
```

Không một kiến trúc sư hệ thống nào xứng đáng với số muối mình từng ăn vào sẽ cho phép một chỉ lệnh như vậy tồn tại trong hệ thống. Sự tồn tại của từ acme đặt mã nguồn trước vô số lỗi nguy hiểm và bí ẩn, đấy là chưa kể tới các rủi ro an ninh.

Thử nghĩ điều gì sẽ xảy ra nếu Acme mua về Purple Cab, thống nhất tất cả các hệ thống, nhưng vẫn duy trì thương hiệu và website riêng biệt? Chẳng lẽ lại phải phải bổ sung thêm một chỉ lệnh rẽ nhánh khác?

Kiến trúc sư của chúng ta sẽ tìm phương án cách ly hệ thống khỏi các vấn đề như thế này, bằng cách tạo ra một loại mô-đun-kiến-tạo-yêu cầu-chuyển-tiếp, được điều khiển bởi một cơ sở dữ liệu có khả năng cấu hình trông như dưới đây:

```
URI          Dispatch Format
---
Acme.com     /pickupAddress/%s/pickup
             Time/%s/dest/%s
*.*          /pickupAddress/%s/pickup
             Time/%s/destination/%s
```

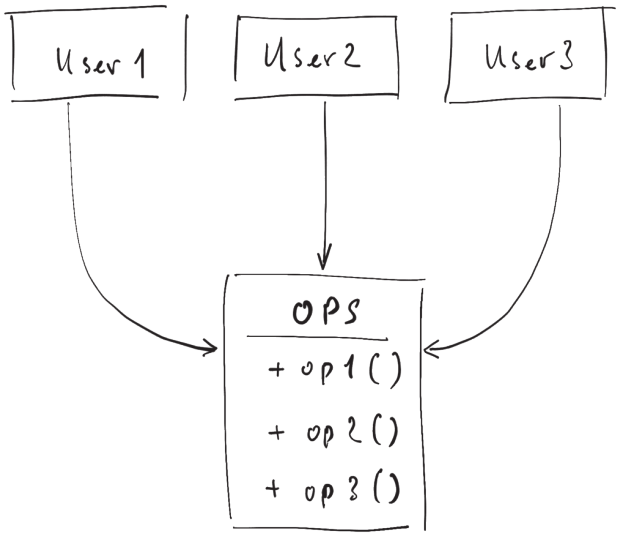
Và thế là anh ta phải đối phó thêm với một lượng lớn thấy rõ các cơ chế và mô-đun mới. Chỉ vì có một dịch vụ con không tuân thủ nguyên tắc thay thế.

### Kết luận

Nguyên Tắc Thay Thế Liskov có thể, và nên được áp dụng tại mức kiến trúc. Một vi phạm đơn giản về khả năng thay thế có thể khiến kiến trúc của hệ thống bị xâm nhiễm bởi một lượng lớn các cơ chế bổ sung.

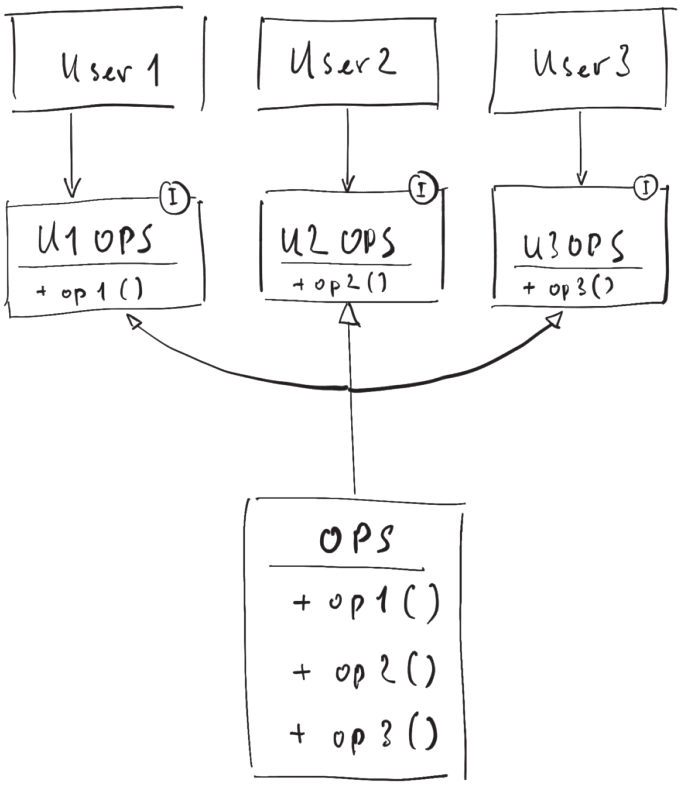
## Nguyên Tắc Phân Tách Giao Diện

Nguyên Tắc Phân Tách Giao Diện được lấy tên từ sơ đồ dưới đây:




Trong tình huống trên, có một số người dùng sử dụng đến các hoạt động của OPS. Giả sử rằng User1 sử dụng đến op1, User2 sử dụng op2 và User3 sử dụng op3.

Bạn cứ thử nghĩ mà xem, thế rồi mã nguồn của User1 sẽ phụ thuộc vào op2 và op3, mặc dù nó không gọi chúng. Vấn đề này có thể giải quyết bằng cách tách các hoạt động của OPS vào các giao diện riêng như dưới đây:




Một trường hợp khác, tương tự, nhưng ở cấp độ cao hơn, khi hệ thống của bạn phụ thuộc vào một framework F, và F thì bị ràng buộc với một cơ sở dữ liệu D. Điều này đã khiến cho hệ thống của bạn bị ràng buộc với D, mặc dù có khi bạn chẳng hề dùng đến.


Những phụ thuộc như thế có thể gây ra điều gì xấu? Mã nguồn bị buộc phải biên dịch lại khi các mối phụ thuộc có thay đổi là điều thứ nhất. Thứ hai, một khi các phụ thuộc có vấn đề, vấn đề đó có thể gây ra lỗi ở mô-đun của bạn, mặc dù bạn không dùng đến bất kỳ tính năng nào ở đó.

## Kết luận

Bài học rút ra ở đây là những mối phụ thuộc không cần thiết có thể gây ra những rắc rối không ngờ tới. Hãy tránh những mối phụ thuộc đó.

# Nguyên Tắc Đảo Ngược Phụ Thuộc

Nguyên Tắc Đảo Ngược Phụ Thuộc cho chúng ta biết rằng các hệ thống linh hoạt nhất là những hệ thống trong đó các phụ thuộc mã nguồn chỉ đề cập đến trừu tượng, chứ không phải các cụ thể hóa.

Trong những ngôn-ngữ-định-kiểu-tĩnh, Java chẳng hạn, phát biểu trên có nghĩa là các chỉ lệnh use, import, hay include chỉ nên tham chiếu đến các mô-đun chứa giao diện, lớp trừu tượng hay các định nghĩa trừu tượng khác. Không nên tham chiếu đến bất kỳ thứ gì cụ thể.

Quy tắc tương tự cũng được áp dụng cho các ngôn-ngữ-định-kiểu-động, như Ruby hay Python. Các phụ thuộc mã nguồn không nên tham chiếu đến các mô-đun cụ thể. Tuy nhiên trong các ngôn ngữ này, việc phân định rõ một mô-đun có được gọi là cụ thể hay không khó khăn hơn đôi chút. Mô-đun cụ thể là những mô-đun mà các hàm của nó đã được triển khai.

Rõ ràng, coi phát biểu trên là một quy tắc là phi thực tế. Các hệ thống phần mềm phải phụ thuộc vào nhiều công cụ cụ thể. Ví dụ, lớp String trong Java là một lớp cụ thể, và sẽ không tố chút nào nếu cố ép String trở thành lớp trừu tượng. Không thể tránh phụ thuộc mã nguồn vào lớp java.lang.string, và không nên tránh.

Khi cân nhắc cẩn thận, lớp String rất ổn định. Nó rất hiếm khi thay đổi và các thay đổi đó luôn được kiểm soát chặt chẽ. Lập trình viên và các kiến trúc sư không phải lo lắng về những thay đổi thường xuyên và thất thường của String.

Với những lý do tương tự, chúng ta nên bỏ qua những hoạt động ổn định của hệ điều hành cũng như các công cụ nền tảng khi nói đến Nguyên Tắc Đảo Ngược Phụ Thuộc. Chúng ta chấp nhận những phụ thuộc đó bởi chúng ta có thể tin tưởng chúng sẽ không thay đổi.

Thứ cần lưu tâm là những triển khai cụ thể trong hệ thống của chúng ta. Đó là những mô-đun mà chúng ta đang tích cực phát triển và đang trải qua thay đổi thường xuyên.

## Những thành phần trừu tượng ổn định

Mỗi thay đổi trên một giao diện trừu tượng luôn kéo theo thay đổi trên những triển khai cụ thể của nó. Ngược lại, các thay đổi của triển khai cụ thể không phải lúc nào cũng (thậm chí thường là không) yêu cầu thay đổi giao diện mà chúng triển khai. Do đó, các giao diện ít biến đổi hơn so với các triển khai.

Các nhà thiết kế phần mềm và kiến trúc sư giỏi sẽ cố gắng để giảm thiểu sự biến đổi của các giao diện. Họ sẽ cố gắng tìm cách thêm chức năng vào triển khai mà không cần phải thay đổi giao diện. Điều này nằm trong mọi bài học vỡ lòng về thiết kế.

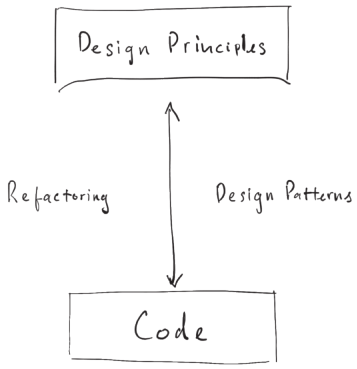
Như vậy là muốn kiến trúc phần mềm được ổn định thì cần tránh đi sự phụ thuộc vào những gì cụ thể và dễ thay đổi. Điều này kéo theo một danh sách các hành dụng khi viết mã:

- Không tham chiếu các lớp cụ thể dễ thay đổi. Thay vào đó, tham chiếu đến các giao diện trừu tượng. Quy tắc này áp dụng cho tất các các ngôn ngữ, cho dù được định kiểu tĩnh hay động. Nó cũng đặt ra những hạn chế khắt khe cho thao tác tạo ra các đối tượng và thường bắt buộc sử dụng mẫu thiết kế Abstract Factory.
- Không kế thừa các lớp cụ thể dễ thay đổi. Đây thật ra là một hệ quả của quy tắc trên, nhưng được viết ra tường minh để nhấn mạnh. Trong các ngôn ngữ định kiểu tĩnh, kế thừa là mạnh mẽ và cứng nhắc nhất trong số tất cả các mối quan hệ trong mã nguồn; do đó nó nên được sử dụng thật cẩn thận. Trong các ngôn ngữ định kiểu động, sự kế thừa ít gặp vấn đề hơn, nhưng nó vẫn là một sự phụ thuộc, và sử dụng phụ thuộc một cách cẩn trọng luôn là một hành động khôn ngoan.
- Không ghi đè các hàm cụ thể. Các hàm cụ thể thường mang theo các phụ thuộc mã nguồn. Khi ghi đè một hàm, chúng ta cũng kế thừa luôn các mối phụ thuộc của nó. Nếu muốn quản lý được các phụ thuộc của hàm, chúng ta nên khiến hàm trở thành hàm trừu tượng và sau đó kiến tạo nhiều triển khai khác nhau.
- Không bao nhắc đến tên của bất kỳ thứ gì cụ thể và dễ thay đổi. Đây thực ra là hồi quy của chính nguyên tắc đầu tiên.

## Kết luận

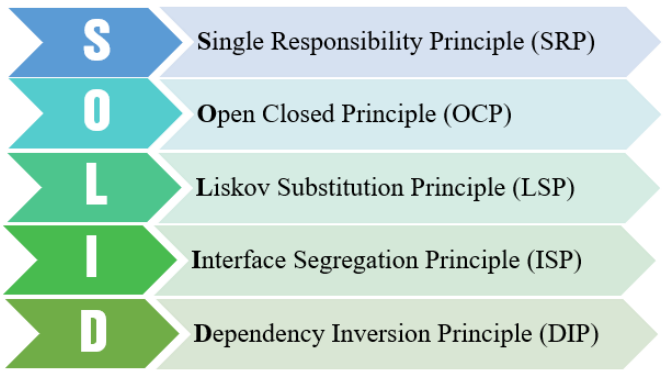
Chúng ta sẽ dành 10% thời gian đầu tiên của dự án để viết nên 90% tính năng. Sau đó chúng ta sẽ dành 90% thời gian còn lại để viết 10% tính năng cuối cùng (mà không biết có xong không).

Các nguyên tắc SOLID không phải là những hướng dẫn thực hành, chúng cũng không được viết ra bởi tính quân phiệt hay kể cả đồng thuận. SOLID xuất phát từ các vấn đề thực tế, rằng sự vi phạm chúng sẽ ngay lập tức dẫn đến những hệ quả rất xấu cho phần mềm. Sự vi phạm các nguyên tắc thiết kế sẽ khiến cho phần mềm ngày càng trở nên khó bổ sung tính năng mới hơn.




Khoảng cách giữa mã nguồn và các nguyên tắc thiết kế được lấp đầy thông qua tái cấu trúc, hoặc áp dụng các mẫu thiết kế, hoặc cả hai. Tái cấu trúc giúp phát hiện mà gỡ bỏ những mã có "mùi" – mà thường là dấu hiệu của một bộ phận kiến trúc không tốt. Trong khi đó các mẫu thiết kế đưa ra những giải pháp hoàn thiện và ổn định để tạo nên các thiết kế cục bộ mà thỏa mãn các nguyên tắc thiết kế.

*Nguồn: https://nguyenbinhson.com*



# DESIGN PATTERN LÀ GÌ VÀ NHỮNG ĐIỀU KHÔNG THỂ BỎ QUA

**Hoàng Mạnh Linh**

Các chuyên gia phần mềm có thể đã quá quen thuộc với thuật ngữ "Design Pattern – Mẫu thiết kế" (sau đây viết tắt là DP), nhưng thực tế cũng còn khá nhiều nhiều người không biết Design Pattern là gì. Do đó, người ta thường không thấy được các DP có giá trị như thế nào và lợi ích nó mang lại cho quá trình phát triển phần mềm, đặc biệt là trong các lĩnh vực bảo trì và tái sử dụng mã ra sao. Hãy cùng tóm tắt các tính năng nổi bật của một DP điển hình và đi đến một định nghĩa để bạn sẽ biết DP là gì và bạn có thể mong đợi những gì khi kết hợp DP vào thiết kế của bạn.

## Design Pattern là gì?

Design Pattern (DP) được định nghĩa là các giải pháp đã được thử nghiệm để giúp giải quyết các vấn đề về thiết kế.

Các DP có nguồn gốc từ Christopher Alexander, một kỹ sư xây dựng đã đúc rút qua kinh nghiệm trong việc giải các vấn đề thiết kế vì chúng liên quan đến các tòa nhà và thị trấn. Alexander nhận ra rằng các cấu trúc thiết kế nhất định, khi được sử dụng hết lần này đến lần khác đều sẽ dẫn đến hiệu quả mong muốn. Ông đã ghi lại và công bố rộng rãi về kinh nghiệm này để những người khác có thể hưởng lợi.

Khoảng năm 1994, các chuyên gia phần mềm đầu tiên đã bắt đầu kết hợp các nguyên tắc của Alexandre vào việc tạo ra tài liệu mẫu thiết kế ban đầu như một hướng dẫn cho các nhà phát triển mới. Sự việc này sau đó được lan rộng và đạt đến đỉnh cao, đưa ra 23 mẫu dựa trên kinh nghiệm của các tác giả tại thời điểm đó. Những mẫu này được chọn vì chúng đại diện cho các giải pháp cho các vấn đề phổ biến trong phát triển phần mềm.

Tóm lại, Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, được các nhà nghiên cứu đúc kết và tạo ra các mẫu thiết kế chuẩn. Và DP không phải là một ngôn ngữ lập trình cụ thể nào cả, nó có thể sử dụng được trong hầu hết các ngôn lập trình có hỗ trợ OOP hiện nay.

## Điều gì khiến Design Pattern trở nên quan trọng?

Như đã nói, DP đang được ứng dụng trong hầu hết các ngôn ngữ lập trình có hỗ trợ OOP hiện nay. Tại sao nó lại phổ biến đến vậy?

### Design Pattern giúp ích trong "giao tiếp", học tập và có cái nhìn sâu sắc hơn

Trong thập kỷ qua, các mẫu thiết kế đã trở thành một phần không thể thiếu trong kho kiến thức của các nhà phát triển. Điều này thực sự giúp ích trong "giao tiếp", ở đây nhắc đến việc giao tiếp qua những dòng code. Người ta có thể dễ dàng nói với một nhà phát triển khác trong nhóm, rằng "tôi sử dụng DP Command ở đây" và nhà phát triển khác không chỉ có thể hiểu về cách thiết

- Trong thập kỷ qua, các mẫu thiết kế đã trở thành một phần không thể thiếu trong kho kiến thức của các nhà phát triển. Điều này thực sự giúp ích trong "giao tiếp", ở đây nhắc đến việc giao tiếp qua những dòng code. Người ta có thể dễ dàng nói với một nhà phát triển khác trong nhóm, rằng "tôi sử dụng DP Command ở đây" và nhà phát triển khác không chỉ có thể hiểu về cách thiết kế mà còn có thể dễ dàng tìm ra lý do tại sao người kia lại thiết kế như vậy.

- Mẫu thiết kế thực sự giúp ích trong học tập. Đặc biệt. khi bạn là người mới trong một dự án. Bạn không tốn quá nhiều công sức để nghĩ về cách những người cũ đang làm với từng dòng code, thay vào đó bạn có thể hòa nhập nhanh hơn. Thêm vào đó, việc có một sự chỉ dẫn chuẩn mực (việc đi theo các DP) sẽ giúp bạn làm "đúng" được nhiều hơn.
- Ngoài ra, điều này giúp cung cấp cho các nhà phát triển có cái nhìn sâu sắc hơn về các phần của ứng dụng mà họ sử dụng của bên thứ 3 thiết kế.

### Design Pattern giúp các nhà phát triển giảm đi sự khó khăn trong quá trình thiết kế hệ thống

- Phân tách hệ thống thành các đối tượng: Phần khó của Thiết kế hướng đối tượng là tìm ra các đối tượng phù hợp và phân tích để bóc tách một hệ thống. Trong mỗi bài toán, người thiết kế cần phải suy nghĩ về tính đóng gói, độ chi tiết, tính phụ thuộc, linh hoạt, hiệu suất, khả năng mở rộng, tái sử dụng và nhiều hơn thế nữa. Việc phải làm hài hòa tất cả những tính chất trên gây nhiều khó khăn trong quá trình phân tích, bóc tách một vấn đề. DP thực sự giúp giảm bớt sự trừu tượng, khiến việc thiết kế trở nên "giảm phần nào" sự khó khăn.
- Chỉ ra rõ ràng việc triển khai một đối tượng: Chúng ta nên triển khai một đối tượng như thế nào? Với một Interface có thể có nhiều lớp cụ thể có thể triển khai nó, mỗi lớp có thể có các cách triển khai rất khác nhau. Các mẫu thiết kế cung cấp những hướng dẫn tổng quát để có thể dẫn đến một bản Code thực sự tốt.
- Đảm bảo cơ chế tái sử dụng: Khi nào nên sử dụng tính Kế thừa, khi nào sử dụng Kỹ thuật Tổng hợp (Composition), khi nào nên sử dụng các kiểu tham số? Làm thế nào để đưa ra được quyết định thiết kế đúng trong trường hợp này? Một lập trình viên, khi cố gắng thiết kế để đảm bảo có thể tái sử dụng, có khả năng duy trì của mã nguồn, họ gặp phải rất nhiều câu hỏi như trên. Việc có hiểu biết về các mẫu thiết kế có thể thực sự có ích khi đưa ra các quyết định như vậy.

### Việc phát triển (mở rộng, bảo trì) hệ thống trở nên dễ dàng hơn

Mọi thứ đều có thể thay đổi, và việc phát triển phần mềm cũng không nằm ngoài quy luật trên, thậm chí việc thay đổi trong công nghệ còn diễn ra nhanh hơn nữa. Các thay đổi làm cho hệ thống phình to do các tính năng mới được thêm vào và bài toán hiệu năng cần được tối ưu.

Vậy làm thế nào để xây dựng phần mềm mà ảnh hưởng của những thay đổi này là nhỏ nhất? Việc hiểu được code (có thể được viết bởi người khác) đã khó và thay đổi code cũ mà không phát sinh các lỗi mới hoặc các bugs ko mong muốn lại càng khó khăn hơn.

Sẽ không có một kĩ thuật thần kỳ nào, nhưng việc dùng DP sẽ cung cấp các mẫu thiết kế có thể áp dụng vào thiết kế của bạn và giải quyết các vấn đề chung. Chúng không phải thư viện hay module. Chúng là những hướng dẫn để bạn tích hợp vào thiết kế để tạo nên các hệ thống hướng đối tượng linh hoạt và dễ bảo trì.

## Để học được Design Pattern, bạn cần gì?

Cần khẳng định điều đầu tiên là DP không dành cho những bạn mới bắt đầu tìm hiểu về lập trình. Điều này không có nghĩa nếu bạn là người mới, bạn không được động vào DP, mà "dành cho" là khái niệm mà khi bạn tìm hiểu sâu, biết vận dụng nó trong các thiết kế của riêng mình.

Để tìm hiểu và học được Design Pattern thì bạn phải nắm chắc được kiến thức OOP đặc biệt là về abstract class, interface và static.

## Các loại Design Pattern

Cơ bản, DP được chia làm 3 dạng chính, tổng cộng 32 mẫu designs:

> **Note:**
>
> Các DP đánh dấu (*) là các DP quan trọng (theo các nguồn tham khảo)

### Creational Pattern (nhóm khởi tạo):

Phục vụ trong việc khởi tạo đối tượng. Nhóm này gồm 9 mẫu design là:

- Abstract Factory. + Prototype
- *Builder. + Simple Factory
- *Factory Method. + *Singleton
- Multiton. + Static Factory
- Pool.

### Structural (nhóm cấu trúc):

Giúp thiết lập, định nghĩa quan hệ giữa các đối tượng. Nhóm này gồm 11 mẫu design là:

- *Adapter/ Wrapper. + *Facade
- Bridge. + Fluent Interface
- Composite. + Flyweight
- Data Mapper. + Registry
- Decorator. + Proxy
- Dependency Injection.

### Behavioral patterns (nhóm ứng xử):

Tập trung thực hiện các hành vi của đối tượng. Gồm 12 mẫu design là

- Chain Of Responsibilities. + *Observer
- Command. + Specification
- Iterator. + *State
- Mediator. + *Strategy
- Memento. + Template Method
- Null Object. + Visitor

## Bạn có hứng thú với Design Pattern?

Với nhà thiết kế, để có được một thiết kế tốt nói chung và hiểu, vận dụng được Design Patterns nói riêng, họ sẽ cần thực tiễn công việc để có thể đi sâu vào các vấn đề và hiểu thông qua các ví dụ code cụ thể hơn là việc học lý thuyết.

Design Pattern đang dần trở nên vô cùng quan trọng. Nhưng việc học, hiểu và thành thạo được những mẫu thiết kế trên không phải là điều dễ dàng.

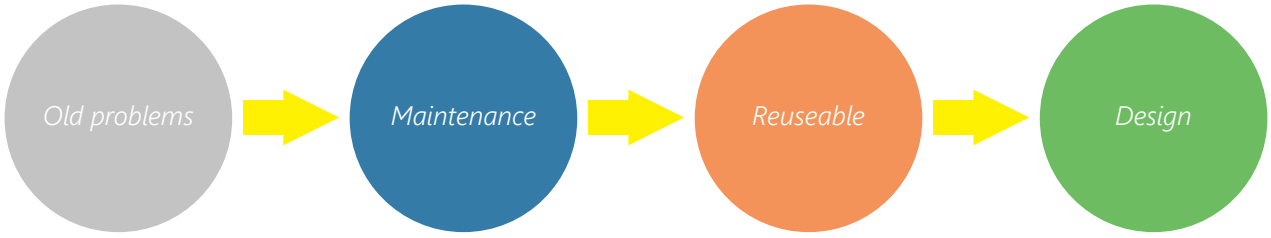
Nhưng trên con đường thành công, người có được sự đột phá sẽ trở nên khác biệt. Vậy tại sao không thử sức mình với Design Pattern, với việc nghiên cứu, nắm chắc lý thuyết, cùng với sự kiên trì thực hành, việc thành thạo được các mẫu thiết kế trên để vận dụng trong thiết kế của riêng bạn? Điều đó sẽ là điểm cộng vô cùng lớn cho bạn trong mắt những nhà tuyển dụng.

Vậy có nên thử?

### Tham khảo:


- https://blogs.agilefaqs.com
- https://viblo.asia
- https://toidicode.com
- https://medium.com


## What is Design Pattern?



# GIẢI THÍCH MÔ HÌNH MVC THÔNG QUA ... CỐC TRÀ ĐÁ

**Nguyễn Khắc Thế**

## Nếu bạn từng đi uống trà đá, thì bạn đã hiểu được MVC rồi

Model - View - Controller (MVC) là một mô hình thiết kế web hiện đại đã trở nên quá đỗi quen thuộc. Cứ thử bước vào một phòng làm việc của các lập trình viên xem, bạn sẽ bị "dội bom" bởi hàng đống thứ khủng khiếp mà bạn có khi còn chưa từng nghe tên như là Ruby on Rails, Angular hay Django.

Nhìn rộng hơn, logic của mô hình MVC được sử dụng để mô tả quá trình làm web của đại đa số các ngôn ngữ như là PHP, Ruby, Python hay JavaScript.

Nhiều ông lập trình web mà cứ theo kiểu nhìn đời qua kẽ lá. Cứ thử để mấy ông lập trình viên khác nhìn vào code của họ xem, họ sẽ cho ngay một bài thuyết trình dài cả tiếng toàn những giáo điều quen thuộc.

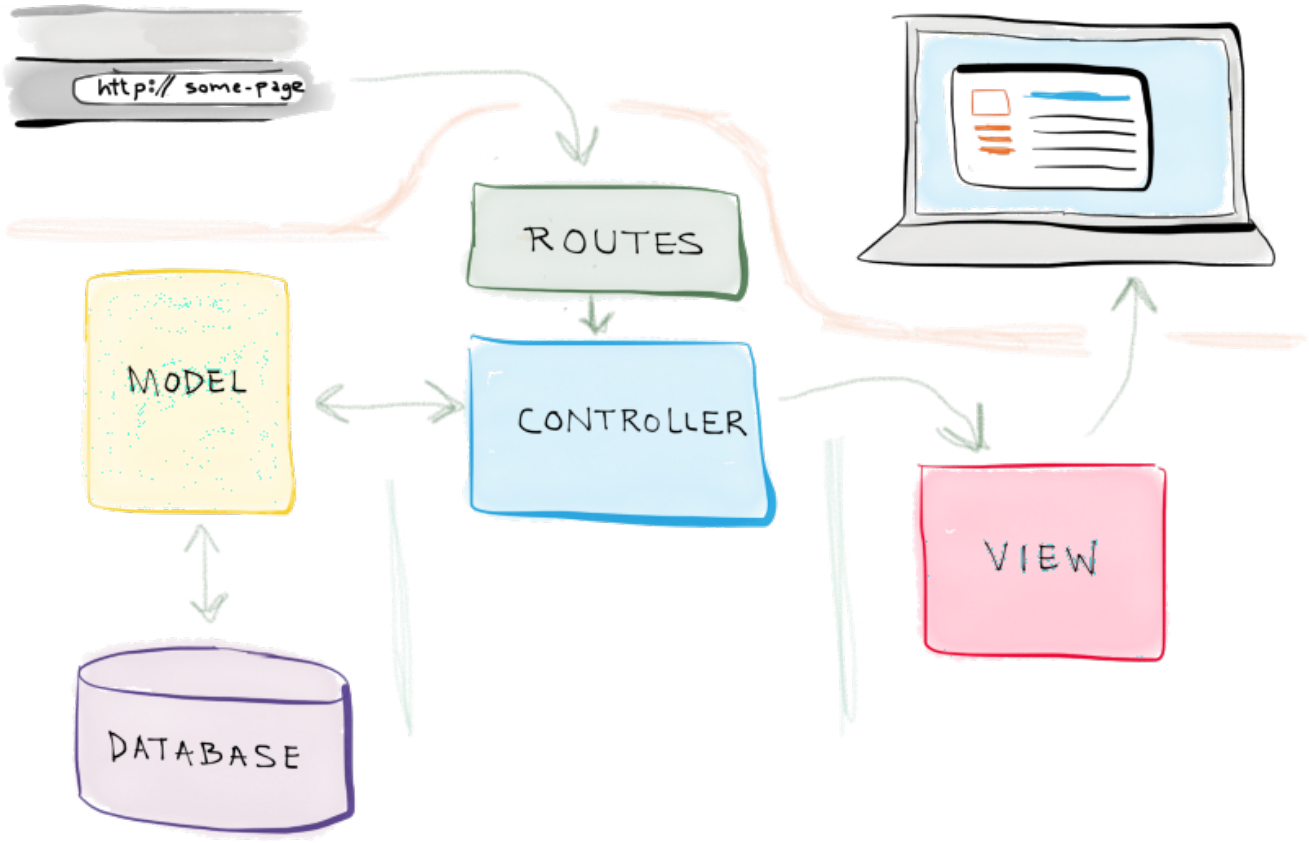
Mà rõ ràng là mấy thứ đấy không thực tế. Trong khi mô hình MVC hiện tại hoàn toàn có thể được giải thích dễ dàng bằng hình thức trà đá vỉa hè. Vậy nên, nếu bạn từng đi ngồi trà đá với bạn bè, thì bạn có thể hiểu được cấu trúc của các ứng dụng web hiện đại rồi đấy.

## Mô hình MVC là gì?

MVC gồm 3 thành phần bao gồm:

- M là Model: một cấu trúc dữ liệu chắc chắn, chuẩn bị dữ liệu để cung cấp cho Controller
- V là View: nơi hiển thị dữ liệu cho người dùng theo cách mà người dùng có thể dễ dàng hiểu và tương tác được
- C là Controller: nhận về lệnh từ người dùng, gửi lệnh đến Model để lấy hay cập nhật dữ liệu, rồi truyền lệnh đến View để cập nhật giao diện hiển thị cho đúng với dữ liệu đã cập nhật

Trên sơ đồ thì nó sẽ như thế này:




Ôi giời lằng nhằng phức tạp quá. Làm cốc trà đá cho hạ nhiệt

Bạn bước vào quán trà đá. Giữa mùa hè, quán nào quán nấy nóng nực đông đúc. Bạn luồn lách qua đám đông để gọi cho bằng được bà chủ quán "cho cháu một cốc trà đá cô ơi". Lúc này, bạn là "người dùng" và "cốc trà đá" là "yêu cầu từ phía người dùng". Đối với bạn, cốc trà đá là thứ đồ uống ưa thích, mát lạnh, quá phù hợp để xua tan cái nóng thủ đô 40 độ C. Bà chủ quán gật đầu "ô kê em nhớ". Đối với bà chủ quán, cốc trà đá không chỉ ngon lành mát lạnh, mà còn là một mớ quy trình các bước:

1. Lấy cái cốc
2. Cho đá vào
3. Cho trà vào
4. Cho thêm nước lọc cho trà loãng bớt
5. Khuấy đều lên
6. Đưa cốc trà cho bạn
7. Thanh toán

Bộ não của bà chủ quán lúc này đóng vai trò Controller. Kể từ thời điểm bạn nói "một cốc trà đá" bằng tiếng Việt và bà chủ quán hiểu được, công việc bắt đầu. Trà đá, nước mía hay cocktail thì cũng như nhau, nhưng nguyên liệu thì hoàn toàn khác biệt. Bà chủ quán chỉ có thể sử dụng những công cụ và nguyên liệu của quán, và những công cụ đó sẽ đóng vai trò Model, bao gồm:

- Đôi tay của bà chủ
- Các nguyên liệu pha chế (trà, nước ...)
- Đá lạnh
- Bia, nước ngọt, thuốc lá...
- Chanh, sấu, gừng...
- Các ly cốc để đựng đồ uống

Những nguyên liệu và công cụ này, thông qua một loạt các bước, đã trở thành cốc trà đá mát lạnh đến tay bạn. Cốc trà đá lúc này đóng vai trò "View". "View" được làm nên từ những công cụ, nguyên liệu trong "Model", được chế biến và bàn giao tới tay bạn thông qua "Controller" (chính là bộ não của bà chủ quán)

## Bài học rút ra là gì?

- 1 cốc không đủ, bạn muốn gọi cốc nữa? Rõ ràng là bạn không thể hét to vào cái cốc đã hết (chính là "View") được, bạn phải gọi bà chủ quán "Controller".
- Thời gian từ lúc bà chủ quán nhận được yêu cầu tới khi làm xong phải tối thiểu nhất có thể. Đó chính là "skinny controller", có thể hiểu là "controller" nên chứa tối thiểu lượng logic cần xử lý và được quản lý lượng model nhiều nhất có thể. Một bà chủ quán "thiện nghệ" không chỉ ghi nhớ chính xác cách làm, mà còn chuẩn bị đầy đủ công cụ và nguyên liệu để không mất nhiều thời gian tìm kiếm và chế biến.
- Vậy nếu bà chủ quán đưa hết nguyên liệu cho bạn rồi bảo bạn tự pha? Chẳng ai làm thế cả. Do vậy, bạn cần phải để việc xử lý logic trên model nhiều nhất có thể, và tối giản hóa view. Nói cách khác, được phục vụ tận miệng thì vẫn thích hơn là phải đi pha.
- Nếu bạn gọi 1 lon bia thì sao? Bà chủ chắc chẳng phải làm gì nhiều, bật nắp lon bia rồi đưa bạn là xong. Nhưng mà rõ ràng bạn vẫn phải gọi bà chủ quán, vì lon bia không thể tự nhảy ra trước mặt bạn được.

## Quay trở lại vấn đề lập trình web

Giờ thử xem một ứng dụng web hiện đại có quy trình thế nào nhé

- Người dùng tạo ra một yêu cầu thông qua đường dẫn, ví dụ /home
- Controller nhận yêu cầu và đưa ra một mệnh lệnh để xử lý yêu cầu đó. Nếu lệnh thực thi với phần View thì cập nhật lại màn hình hiển thị, với Model thì để trình diễn logic. Giả sử yêu cầu của người dùng có yếu tố logic
- Model thực thi phần logic lấy từ một cơ sở dữ liệu nào đó và gửi trả lại phản hồi theo hướng dẫn từ Controller
- Controller truyền dữ liệu này ra phần view và cập nhật lại giao diện cho người dùng.
- Mọi yêu cầu đều phải đi qua Controller trước khi chuyển hóa thành lệnh thực thi cho View hay Model.

## Tổng kết

Bất cứ khi nào bạn học một framework lập trình web mới, bạn sẽ gặp mô hình MVC. Nói cách khác, một khi bạn đã lập trình dựa trên MVC thì không cần phải ngán bất cứ framework mới nào cả.

*Nguồn: https://www.codementor.io/@kevinkononenko/model-view-controller-mvc-explained-through-ordering-drinks-at-the-bar-i7nupj4oe*

# SỬ DỤNG **OBSERVER PATTERN** TRONG JAVASCRIPT

**Đặng Huy Hòa**

Với chủ đề "Design Patterns" của Tạp Chí Lập Trình Vol.6, tác giả muốn giới thiệu với các bạn về Observer pattern. Đây 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

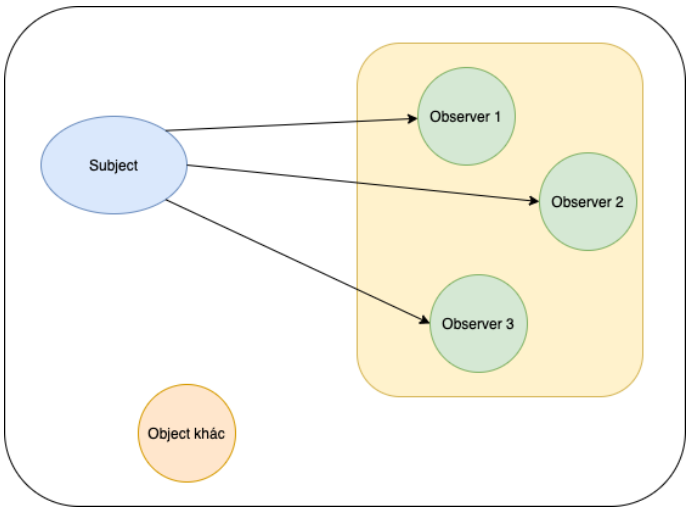
## Giới thiệu

### 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.

### 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 |

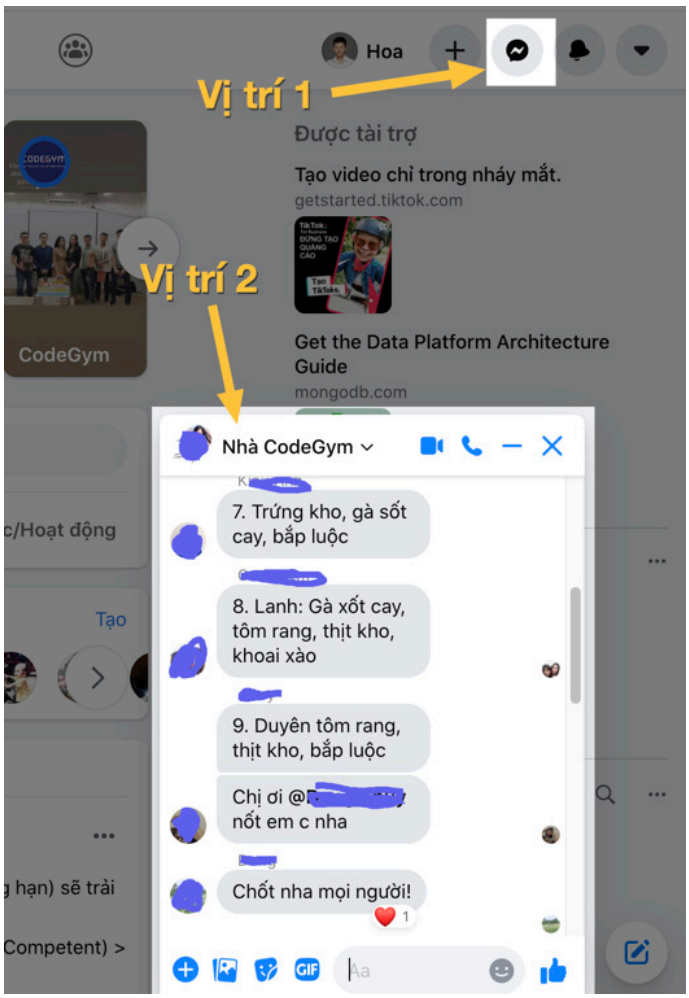
Với ngôn ngữ JavaScript, mặc dù một số từ khoá theo mô hình OOP (như class, extends) vẫn được hỗ trợ, nhưng lại thiếu cơ chế abstraction như abtract class, interface. Vậy nên, việc triển khai Observer pattern sẽ hơi khác so với các hướng dẫn dành cho OOP. Phần triển khai cụ thể sẽ được trình bày chi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

Tác giả đánh dấu hai vị trí cần lưu ý:

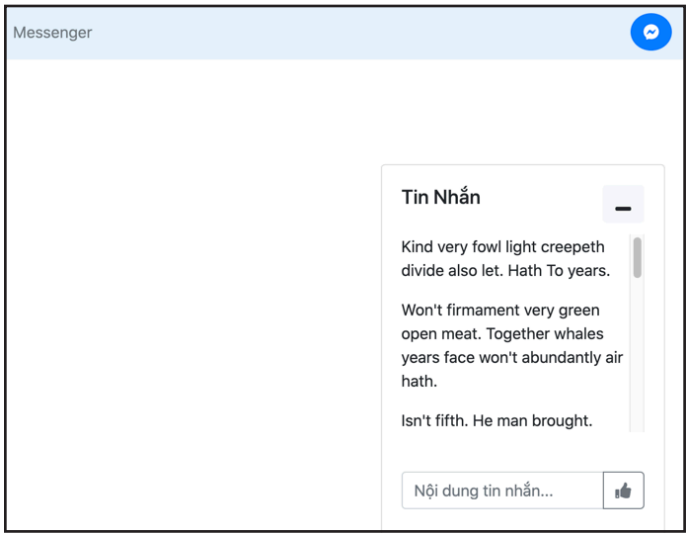
- Vị trí thứ nhất là thông báo số tin nhắn mới từ Messenger.
- Vị trí thứ hai là cửa sổ chat Messenger khi người dùng lướt Newsfeed. Nếu ở dạng rút gọn (minimize) thì vị trí này sẽ hiển thị số tin mới chưa được đọc (giống vị trí thứ nhất).




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**
>
> 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(message));
}
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.

# REPOSITORY DESIGN PATTERN
# HOÀN THIỆN TRONG LARAVEL

**Dịch: Dương Nhật Minh**

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.

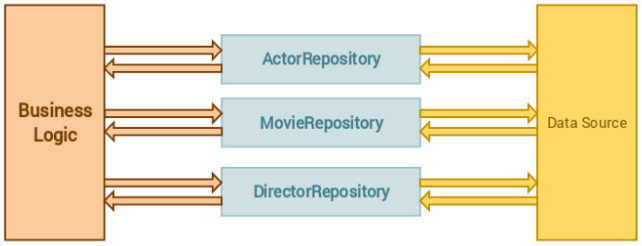
## Hãy bắt tay vào việc code bây giờ

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'. $u
         er->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?

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

# DESIGN PATTERNS TRONG PHP: DECORATOR (VỚI LARAVEL)

**Dịch: Đinh Thị Hoài Thu**

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 Wikipedia

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 = 1440; # 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 Posts-
RepositoryInterface
{
    protected $model;
    public function __construct(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\
PostsRepository;
use Illuminate\Support\ServiceProvider;
class AppServiceProvider extends Ser-
viceProvider
{
    public function register()
    {
      $this->app->bind
      (PostsRepositoryInterface::class,
      PostsRepository::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 1 `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 = 1440; # 1 day
    public function __construct(Cache
    anager $cache, PostsRepository
    $repo) {
      $this->repo = $repo;
      $this->cache = $cache;
    }
    public function get() {
      return $this->cache->
      remeber('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\PostsReposi-
toryInterface;
use App\Repositories\Posts\PostsCacheRe-
pository;
use Illuminate\Support\ServiceProvider;
class  AppServiceProvider  extends  Ser-
viceProvider
{
    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

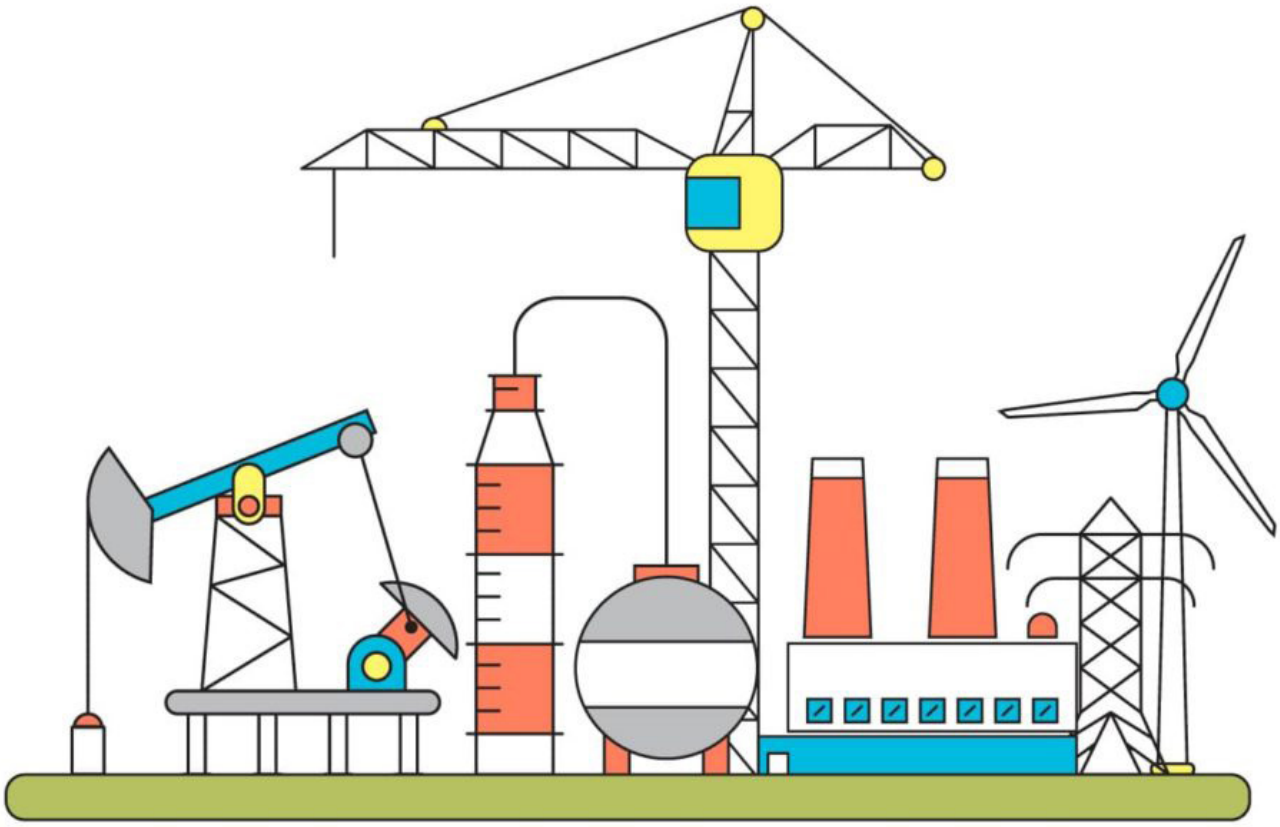
## Kết luận

- 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.

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

# FACTORY METHOD TRONG THỰC TIỄN



**Nguyễn Khắc Thế**

## 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

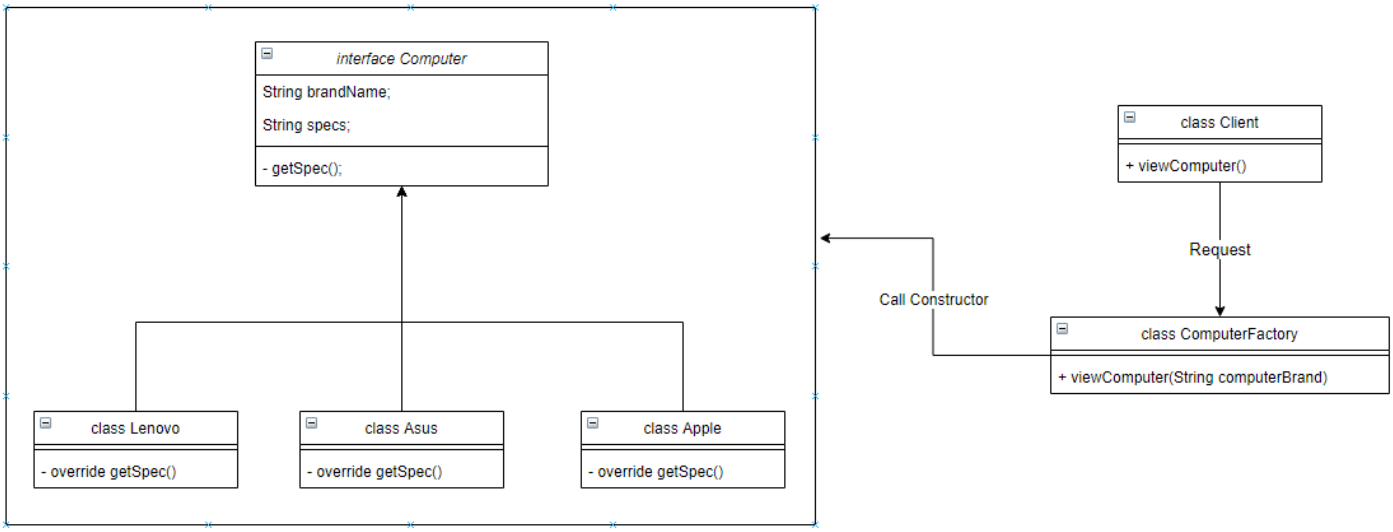
Xét theo ví dụ ở đầu bài viết, Factory pattern sẽ có cấu trúc dạng như sau:

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 Computer-Factory và truyền vào đó một tham số computer-Brand, 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(lenovoCompu
        er.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 coputerBrand) {
      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.