**Clean Architecture**

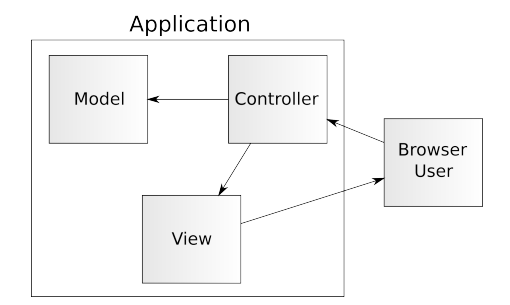
[Software](https://viblo.asia/tags/software)[Clean Architecture](https://viblo.asia/tags/cleanarchitecture)[Design Pattern](https://viblo.asia/tags/design-pattern)[Software Architecture](https://viblo.asia/tags/software-architecture)

**Hạn chế của MVC**

Để xây dựng một ứng dụng cần rất nhiều thành phần khác biệt kết hợp lại. Đó là HTML, CSS, JavaScript để trình diễn ứng dụng đến người dùng. Cũng có thể là tầng dữ liệu kiểu database, một API hay một file với kích thước lớn. Sau đó sẽ có các đoạn mã xử lý, được dùng để xác nhận request, xử lý và hiển thị dữ liệu cho người dùng. Điều cuối cùng chính là khía cạnh nghiệp vụ của ứng dụng. Các quy tắc bắt buộc ứng dụng phải tuân theo, các thành phần liên quan đến nhau như thế nào, và mối quan hệ giữa chúng ra sao.

Khi một lập trình viên muốn viết mã đẹp hơn, họ nhanh chóng tìm đến kiến trúc MVC. MVC tách biệt tầng database, tầng điều khiển và tầng giao diện. Mặc dù MVC cũng có một số yếu điểm, nhưng nó cung cấp một kiến trúc cho việc phân chia mã gọn gàng.

*MVC trong đồ thị*



Về cơ bản, controller là phần cơ sở phân tích request và xác định hành động. Người dùng bắt đầu một controller bằng cách gõ một đường link cụ thể trên trình duyệt, nó sau đó định tuyến đến một controller cụ thể được thiết kế với nhiệm vụ là xử lý request đó. Controller thao tác với một vài model để nhận dữ liệu. Cuối cùng, controller trả về một view đến người dùng.

**MVC không đủ tốt**

Kiến trúc MVC là một khởi đầu tuyệt vời để xây dựng một phần mềm mạnh mẽ và linh hoạt, nhưng nó không phải lúc nào cũng tốt. Với chỉ 3 layer để tổ chức tất cả các đoạn mã, developer thường tạo ra quá nhiều chi tiết trong một layer. Có vẻ hơi táo bạo khi nghĩ rằng mọi thứ nên được đặt trong 3 nhóm này, dù đó là model, view hay controller. View thường tránh khỏi việc bị chế biến vì vai trò của nó khá rõ ràng, trong khi controller trở nên quá tải với các business logic, hoặc là model chứa hết. Cộng đồng này có thời gian luôn tin tưởng câu thần chú: "fat model, skinny controller".

Câu thần chú "fat model, skinny controller" vẫn rất tốt, cho đến khi tầng model trở thành tầng trừu tượng và nhất quán dữ liệu, và phần business logic trở nên ghép nối chặt với data source. Nó dần trở thành một obese model, chứ không chỉ là fat nữa.

Thật vô nghĩa khi đặt đoạn mã cấu hình hoặc tương tác cơ sở dữ liệu trong tầng view, mọi thứ liên quan đến cơ sở dữ liệu, truy vấn nên được giải quyết trong model. Tuy nhiên, việc ghép nối chặt data model và data source rõ ràng có vấn đề.

Giả sử, trong vòng lặp đầu tiên của ứng dụng, chúng ta lưu và nhận dữ liệu từ hệ quản trị cơ sở dữ liệu. Sau đó, theo nhu cầu và số ứng dụng tăng lên, chúng ta quyết định xây dựng một API web service để quản lý tương tác của dữ liệu với tất cả các ứng dụng. Vì chúng ta ghép nối chặt business logic và data access với nhau trong model, rõ ràng rất khó để chuyển đổi sang API mà không động chạm nhiều đến code.

Thay vì đổi data source, chúng ta có thể tìm một thư viện trừu tượng hóa cơ sở dữ liệu. Có lẽ thư viện này đã được xây dựng một cách tối ưu cho truy vấn, lưu trữ tài nguyên, thực thi nhanh hơn và dễ viết hơn. Có nhiều lý do để đổi, tuy nhiên, nếu lúc đầu bạn đã đi theo con đường khớp cài đặt cơ sở dữ liệu với trình diễn dữ liệu, bạn có thể sẽ phải viết lại toàn bộ tầng model chỉ để đổi sang một thư viện tốt hơn.

Một model sạch, đẹp có thể viết như sau:

class User {

public $id;

public $alias;

public $fullName;

public $email;

}

Nếu chúng ta thêm phương thức vào class User này thì nó rất khó test và chuyển sang data source hoặc database abstract layer khác.

**Model layer vs Model class vs Entities**

Giải pháp cho vấn đề này là nhận ra sự khác biệt giữa tầng Model của mô hình MVC, và class Model. Cái chúng ta viết ở ví dụ trên là class Model, là biểu diễn dữ liệu. Đoạn mã chịu trách nhiệm nhất quán dữ liệu để lưu trữ cơ sở dữ liệu là một phần của tầng model, nhưng không nên là một phần của class Model. Chúng ta đang lẫn lộn chúng với nhau. Một bên là mô hình dữ liệu, một bên là thống nhất dữ liệu.

Từ bây giờ, chúng ta sẽ gọi class User như trên là entity. Entity đơn giản là một trạng thái của những thuộc tính đã được định danh duy nhất. Chẳng hạn, một Order, Customer, Product, là entity.

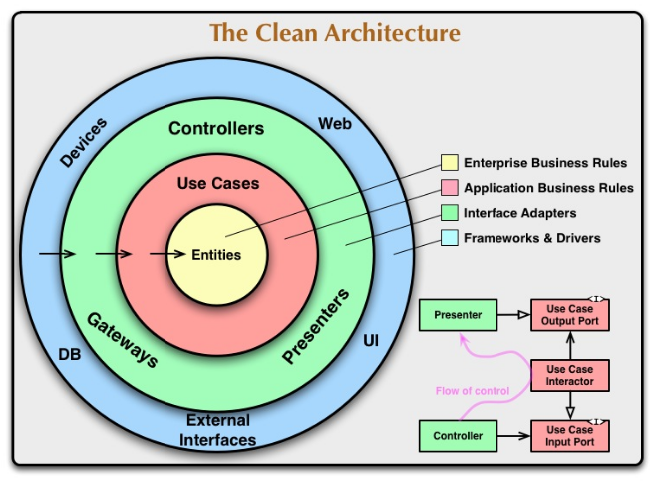
**Thêm layter cho tất cả mọi thứ**

Tất nhiên giải pháp đưa ra của chúng ta ở đây là thêm một layer đến mô hình MVC, lúc này trở thành EPVC, có nghĩa là Entity-Persistence-View-Controller. Tất nhiên không có nghĩa rằng mọi vấn đề đều được giải quyết đơn giản bằng cách quẳng hết các layer khác, ý ở đây là chia nhỏ phần biểu diễn dữ liệu với việc nhất quán dữ liệu vì chúng thực tế là hai thứ hoàn toàn khác nhau.

Điều này cho phép chúng ta chuyển database khỏi phần core của ứng dụng thành một nguồn riêng biệt. Entity trở thành core, dấn đến một cách nghĩ hoàn toàn khác biệt về ứng dụng phần mềm.

**Clean Architecture**

Vậy nếu MVC là chưa đủ thì giải pháp là gì, và cái gì mới là đủ?



Qua rất nhiều năm, chúng ta đã được nhìn thấy rất nhiều ý tưởng về kiến trúc hệ thống. Bao gồm:

* Kiến trúc hexagonal
* Kiến trúc Onion
* Kiến trúc screaming
* DCI
* BCE

Dù những kiến trúc này có biến đổi một chút về chi tiết, nhưng nhìn chung chúng tương tự nhau. Tất cả đều chung một mục đích là phân tách các mối quan hệ, bằng cách chia phần mềm vào các layer. Mỗi kiến trúc đều có ít nhất một layer cho quy tắc nghiệp vụ (business rule), và một layer cho giao diện.

Mỗi một kiến trúc đều sinh ra một hệ thống như sau:

1. Độc lập với framework: kiến trúc không phụ thuộc vào bất kỳ thư viện nào. Điều này cho phép bạn sử dụng framework như những công cụ, thay vì nhồi nhét hệ thống của bạn vào những hạn chế của chính nó.
2. Testable: những quy tắc nghiệp vụ có thể được kiểm chứng mà không cần UI, database, web server hoặc bất kỳ yếu tố bên ngoài nào cả.
3. Độc lập với giao diện người dùng: giao diện người dùng có thể thay đổi dễ dàng, mà không ảnh hưởng đến phần còn lại của hệ thống. Chẳng hạn, một giao diện web có thể được thay đổi thành giao diện dòng lệnh, mà không làm thay đổi business rule.
4. Độc lập với cơ sở dữ liệu: bạn có thể thay đổi từ Oracle hoặc Sql Server thành Mongo, BigTable, CouchDB hoặc bất kỳ hệ quản trị cơ sở dữ liệu nào. Business rule không bị ràng buộc bởi cơ sở dữ liệu.
5. Độc lập với các chương trình bên ngoài: Quy tắc nghiệp vụ không biết bất cứ điều gì về thế giới bên ngoài.

Kiến trúc Clean chính là nỗ lực để tích hợp tất cả những kiến trúc trên thành một khối ý tưởng duy nhất.

**Dependency Rule**

Những vòng tròn đồng tâm đại diện cho các khu vực khác nhau của phần mềm. Nhìn chung, càng đi sâu thì phần mềm càng ở mức độ cao hơn. Các vòng tròn bên ngoài là cơ chế. Các vòng tròn bên trong là điều khoản.

Quy tắc quan trọng để kiến trúc này hoạt động là Dependency Rule. Quy tắc này cho rằng sự phụ thuộc mã nguồn chỉ nên ảnh hưởng đến phần bên trong, tức là những thứ ở vòng tròn bên trong không thể biết về những thứ ở vòng tròn bên ngoài. Cụ thể, một định danh được khai báo ở vòng tròn bên ngoài không thể được gọi đến bởi đoạn mã ở vòng tròn bên trong. Bao gồm hàm, lớp, biến và các thực thể có thể gọi tên khác của phần mềm.

Tương tự như vậy, định dạng dữ liệu được sử dụng ở vòng tròn bên ngoài không nên được dùng bởi vòng tròn bên trong, đặc biệt nếu những định dạng đó được sinh ra bởi framework thuộc vòng tròn bên ngoài. Chúng ta không muốn bất cứ thứ gì ở vòng tròn bên ngoài ảnh hưởng đến vòng tròn bên trong.

**Entity (Thực thể)**

Các thực thể sẽ đóng gói các quy tắc nghiệp vụ. Một thực thể có thể là một đối tượng với phương thức, hoặc có thể là một tập các cấu trúc dữ liệu và hàm, miễn là nó có thể được sử dụng bởi nhiều ứng dụng khác nhau trong toàn bộ phần mềm.

Nếu bạn chỉ viết một ứng dụng đơn lẻ, thì những thực thể chính là các đối tượng nghiệp vụ của ứng dụng. Chúng đóng gói các quy tắc chung và cao cấp nhất. Chúng ít có khả năng thay đổi do ảnh hưởng từ những thay đổi bên ngoài. Chẳng hạn, bạn không thể mong những đối tượng này bị ảnh hưởng bởi một thay đổi đến từ việc phân trang hoặc bảo mật. Những hoạt động làm thay đổi một ứng dụng cụ thể không nên ảnh hưởng đến tầng thực thể.

**Use Cases**

Phần mềm trong những layer này chứa những quy tắc nghiệp vụ cụ thể. Nó đóng gói và thiết lập tất cả use case của hệ thống. Những use case này sắp xếp các luồng dữ liệu đến và đi từ thực thể, và chỉ đạo các thực thể này sử dụng các quy tắc nghiệp vụ để đạt được mục đích của use case.

Chúng ta không mong chờ việc thay đổi ở layer này ảnh hưởng đến các thực thể. Chúng ta cũng không muốn layer này bị ảnh hưởng bởi những thay đổi của các yếu tố bên ngoài như database, giao diện người dùng, hoặc bất kỳ framework nào. Layer này được phân tách khỏi những mối quan hệ như thế.

Tuy nhiên, chúng ta lại muốn những thay đổi trong việc hoạt động của ứng dụng sẽ ảnh hưởng đến use case, và cả phần mềm trong layer này. Nếu chi tiết của một use case thay đổi, thì một vài đoạn mã trong layer này cũng cần thay đổi tương ứng.

**Interface Adapter**

Phần mềm trong layer này là một bộ các adapter có nhiệm vụ chuyển dữ liệu từ định dạng thuận tiện nhất cho các use case và thực thể, thành định dạng thuận tiện nhất cho các chương trình bên ngoài như Database hoặc Web. Layer này chứa kiến trúc MVC. Presenters, Views và Controllers đều thuộc layer này. Những Model chỉ là cấu trúc dữ liệu được truyền từ Controller đến use case, và sau đó truyền từ use case đến Presenter và View.

Tương tự, dữ liệu được chuyển đổi, từ hình thức phù hợp cho entity và use case, thành hình thức phù hợp cho framework đang được sử dụng. Những đoạn mã ở trong vòng tròn này không biết gì về cơ sở dữ liệu. Nếu cơ sở dữ liệu là kiểu SQL, thì tất cả truy vấn SQL nên được giới hạn cho layer này, cụ thể là các phần của layer này mà phải làm việc với cơ sở dữ liệu.

**Framework và Driver**

Layer ngoài cùng là kết hợp các framework và công cụ như cơ sở dữ liệu, web framework,… Nói chung bạn không viết nhiều code trong layer này ngoại trừ các đoạn mã để liên lạc với các vòng tròn tiếp theo ở bên trong.

Layer này là nơi tập trung của các chi tiết. Web là một chi tiết. Cơ sở dữ liệu là một chi tiết. Chúng ta giữ những thứ này ở bên ngoài, nơi chúng khó có thể gây ảnh hưởng đến các phần ở vòng tròn bên trong.

**Chỉ 4 vòng tròn là đủ?**

Các vòng tròn là giản đồ. Bạn có thể thấy rằng bạn cần nhiều hơn 4 vòng tròn. Không có quy luật nào nói rằng bạn luôn phải có chỉ 4 vòng tròn. Tuy nhiên, Dependency Rule luôn được áp dụng. Sự phụ thuộc mã nguồn luôn hướng vào bên trong. Khi bạn di chuyển vào tâm vòng tròn, mức độ trừu tượng tăng lên. Vòng tròn ngoài cùng là các chi tiết cụ thể ở mức thấp nhất. Bạn càng di chuyển vào trong, phần mềm càng phát triển trừu tượng hơn, và đóng gói các điều khoản ở cấp cao hơn. Vòng tròn trong cùng chỉ chứa những gì chung nhất, khó có thể chia nhỏ được nữa.

Góc bên phải, phía dưới của sơ đồ là ví dụ cách chúng ta vượt qua những vòng tròn. Nó chỉ ra rằng, Controllers và Presenters liên lạc với Use Case trong layer kế tiếp. Chú ý dòng điều khiển, nó bắt đầu trong controller, di chuyển qua các use case và sau đó thực thi trong presenter.

Dữ liệu đi qua các vùng ranh giới là cấu trúc dữ liệu đơn giản. Chúng ta không muốn truyền Entity hoặc các bản ghi của cơ sở dữ liệu. Chúng ta cũng không muốn cấu trúc dữ liệu có bất kỳ phụ thuộc nào mà vi phạm Dependency Rule.

Lấy ví dụ, nhiều framework cơ sở dữ liệu trả về định dạng dữ liệu phù hợp cho một truy vấn. Chúng ta có thể gọi nó là một RowStructure. Rõ ràng ta không muốn truyền kiểu định dạng dữ liệu này vào các vòng bên trong vì nó vi phạm Dependency Rule, các vòng tròn bên trong có thể biết được đôi chút về vòng tròn bên ngoài. Chính vì thế, khi truyền dữ liệu vào vòng tròn bên trong, chúng luôn phải là định dạng phù hợp với vòng tròn đó.

**Kết luận**

Tuân theo các quy tắc đơn giản này không phải là một việc quá khó khăn, bằng việc phân tách phần mềm vào các layer, đồng thời ghi nhớ Dependency Rule, bạn sẽ xây dựng được một hệ thống về bản chất có thể test được, cùng với những lợi ích kèm theo như đã đề cập ở trên. Khi bất kỳ bộ phận bên ngoài của hệ thống trở nên lỗi thời, chẳng hạn như cơ sở dữ liệu, hoặc web framework, bạn hoàn toàn có thể thay thế chúng với một effort tối thiểu.

*Lược dịch* ([The Clearn Architecture - Uncle Bob - 2012](https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html))

*Note*: Ở đây mình chỉ lược dịch một chút về khái niệm Clean Architecture, do Robert C. Martin đề xuất năm 2012, giữa năm sau, khoảng tháng 6 năm 2017, quyển sách *Clean Architecture* của ông sẽ được xuất bản, trình bày và giải quyết đầy đủ mọi khía cạnh xoay quanh khái niệm này.