**Tìm hiểu Domain Driven Design**

# **Domain Driven Design là gì?**

Có lẽ chúng ta đã quen với cách tiếp cận Data Driven Design. Nghĩa là đầu tiên chúng ta đọc specification và tìm hiểu chức năng, rồi chia nhỏ các task cần phải làm. Trong phần lớn các project, việc này có mục đích là ước lượng thời gian có thể bàn giao và lập kế hoạch cho các task. Vậy trình tự công việc sẽ là ước lượng thời gian, chia các task nhỏ cho các thành viên trong team, thiết kế databse, code.

Với Data Driven Design chúng ta vẫn có thể hoàn thành project, tuy nhiên chúng ta chưa thực hiện tốt trong việc bảo trì và mở rộng project.

Trong các ứng dụng có rất nhiều phần code không liên quan tới Business Logic như truy cập file, mạng và database, các phần code này được gọi là plumping code. Plumping code được nhúng vào Business Object và nhiều Business Logic cũng được nhúng vào behavior của UI hay script của database. Chỉ một thay đổi nhỏ ở tầng UI cũng có thể dẫn tới việc thay đổi tầng logic và ngược lại khi thay đổi một business rule của ứng dụng đòi hỏi chúng ta phải quan tâm đến từng chi tiết nhỏ phía UI cũng như Database để đáp ứng được sự thay đổi này.

Chúng ta sẽ không nhìn thấy vấn đề này trong các ứng dụng nhỏ. Ở các ứng dụng cỡ vừa thì vấn đề này sẽ thấy rõ. Còn đối với ứng dụng lớn thì nó sẽ trở thành 1 vấn đề nghiêm trọng. Data Driven Design sẽ không giúp chúng ta thiết kế hướng đối tượng chính xác. Giải pháp ở đây sẽ là Domain Driven Design (DDD).

DDD là một cách tiếp cận trong việc xây dựng phần mềm phức tạp, sự phức tạp ở đây là những logic của lĩnh vực kinh doanh (domain business).

# **Giải thích Domain**

Định nghĩa dễ hiểu nhất về domain ở đây là, ví dụ trong một e-commerce system thì product, order, customer... và các hiểu biết như customer có thể order một product, một order có thể được thanh toán bởi một customer chính là thuộc về domain.

Những phần về kỹ thuật bao gồm xử lý request, form, connect database, lưu trữ thông tin nằm ngoài domain.

# **Ubiquitous Language (ngôn ngữ chung)**

Là ngôn ngữ liên quan tới domain được sử dụng bởi tất cả các thành viên trong nhóm phát triển (developer, domain expert, customer).

Đây là ngôn ngữ giải quyết được vấn đề là domain expert thường không hiểu được ngôn ngữ kỹ thuật, còn nhà phát triển phần mềm thường không hiểu được business logic. Nên sẽ phải cần một ngôn ngữ chung để domain expert và nhà phát triển phần mềm có thể hiểu một sản phẩm theo một cách như nhau.

Một nguyên tắc cốt lõi của thiết kế hướng lĩnh vực là sử dụng ngôn ngữ dựa trên mô hình. Vì mô hình là xuất phát điểm chung, là nơi ở đó phần mềm "gặp" domain, việc sử dụng nó là nền tảng cho ngôn ngữ là hợp lý.

# **Thiết kế hướng mô hình**

Các nhà phân tích kiến trúc hệ thống khi làm việc với các chuyên gia nghiệp vụ (Domain Experts) sẽ xây dựng mô hình, trao đổi với các Domain Expert để xác định mô hình là phù hợp. Sau đó các nhà phân tích kiến trúc hệ thống sẽ trình bày và truyền đạt lại với nhà phát triển, lập trình viên để đảm bảo cho họ cũng hiểu được mô hình đó. Lập trình viên khi code cũng phải thể hiện được mô hình qua code, nếu có bất kỳ điều gì không ổn hoặc bất hợp lý và cần phải sửa đổi lại mô hình thì phải thông báo cho nhóm phát triển để có được sự đồng thuận, hoặc lớn hơn phải được review và đồng thuận từ Domain Experts.

## **Kiến trúc phân lớp.**

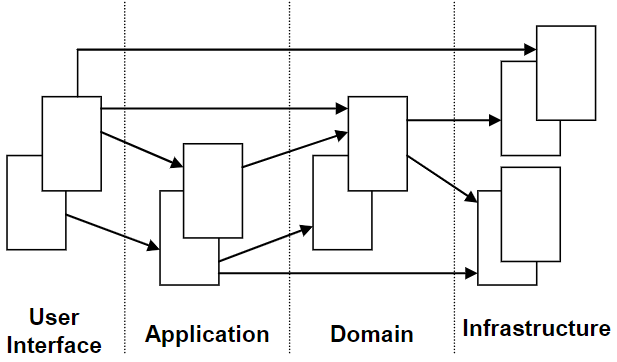
Về mặt kiến trúc DDD đề ra việc phân chia làm 4 tầng logic như sau:

+ User Interface: Chịu trách nhiệm trình bày thông tin tới người sử dụng và thông dịch lệnh của người dùng.

+ Application Layer: Tầng này được thiết kế khá mỏng (ít xử lý logic) phối hợp các hoạt động của ứng dụng. Nó không chứa logic nghiệp vụ. Nó không lưu giữ trạng thái của các đối tượng nghiệp vụ nhưng nó có thể giữ trạng thái một tiến trình của ứng dụng. Chúng ta có thể hình dung phần này gần giống với các Controller trong mô hình MVC chỉ làm nhiệm vụ chuyển tiếp các task đến nơi cần xử lý.

+ Domain Layer: Tầng này chứa thông tin về các lĩnh vực. Đây là trái tim của nghiệp vụ phần mềm. Trạng thái của đối tượng nghiệp vụ được giữ tại đây. Persistence của các đối tượng nghiệp vụ và trạng thái của chúng có thể được ủy quyền cho tầng Infrastructure.

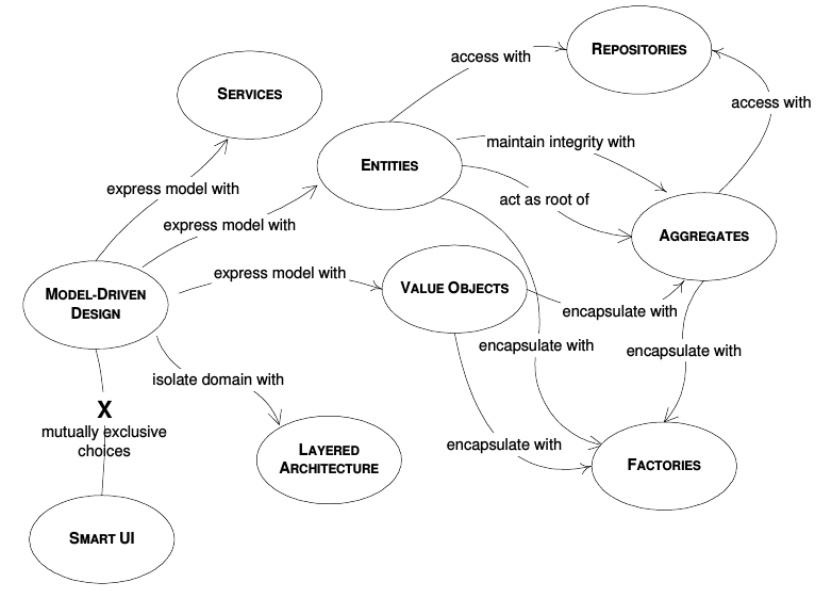
+ Infrastructure Layer: Tầng này đóng vai trò như một thư viện hỗ trợ cho tất cả các tầng còn lại. Nó cung cấp thông tin liên lạc giữa các lớp, cung cấp chức năng lưu trữ các đối tượng nghiệp vụ, chứa các thư viện hỗ trợ cho tầng giao diện người dùng...



Việc thiết lập và giữ vững các quy tắc giao tiếp giữa các lớp, phân chia trách nhiệm hợp lý là điều kiền cần thiết đảm bảo kiến trúc hệ thống là trách nhiệm cực quan trọng của lập trình viên.

## **Các khối ghép của một thiết kế hướng lĩnh vực**

Phần này sẽ giới thiệu các khuôn mẫu quan trọng nhất được sử dụng trong DDD. Mục đích của những khuôn mẫunày là để trình bày một số yếu tố chính của mô hình hóa hướng đối tượng và thiết kế phần mềm từ quan điểm của DDD. Sơ đồ sau đây là một biểu đồ của các khuôn mẫu sẽ được trình bày và các mối quan hệ giữa chúng.



## **Entity**

Entity (Thực thể) dùng để biểu thị các khái niệm mà sự tồn tại của nó liên tục xuyên suốt, dù các thuộc tính có thay đổi.

Lấy ví dụ, để tạo một lớp Person chứa thông tin về một người chúng ta có thể tạo Person với các trường như: tên, ngày sinh, nơi sinh v.v... Những thuộc tính này có thể coi là định danh của một người không? Tên thì không phải vì có thể có trường hợp trùng tên nhau, ngày sinh cũng không phải là định danh vì trong một ngày có rất nhiều người sinh ra, và nơi sinh cũng vậy. Một đối tượng cần phải được phân biệt với những đối tượng khác cho dù chúng có chung thuộc tính đi chăng nữa. Việc nhầm lẫn về định danh giữa các đối tượng có thể gây lỗi dữ liệu nghiêm trọng.

Vì vậy 1 Entity trong phần mềm cần phải tạo 1 định danh. Ví dụ: Đối với một tài khoản ngân hàng thì số tài khoản là đủ để tạo định danh.

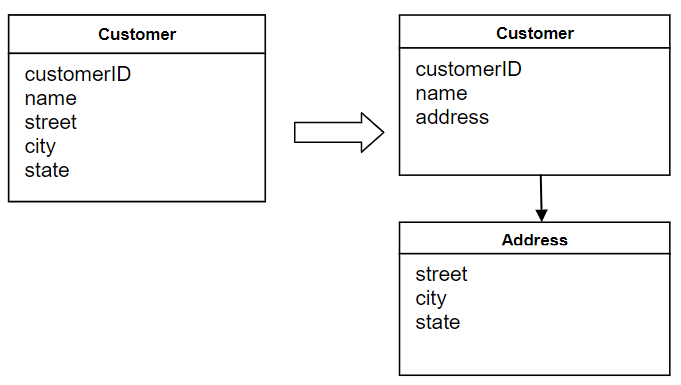
## **Value Object**

Có những lúc mà ta cần có các thuộc tính của một phần tử trong domain. Khi đó ta không quan tâm đó là đối tượng nào, mà chỉ quan tâm thuộc tính nó có. Một đối tượng mà được dùng để mô tả các khía cạnh cố định của một Domain, và không có định danh, được gọi tên là Value Object.

Cần phân biệt giữa Entity Object và Value Object. Để đảm bảo tính đồng nhất ta sẽ không gắn mọi đối tượng thành thực thể. Thay vào đó ta chỉ gắn thực thể cho những đối tượng nào phù hợp nhất với các đặc tính của thực thể. Các đối tượng còn lại sẽ là Value Object.

Nhờ không có định danh, Value Object có thể được tạo và hủy dễ dàng. Lập trình viên không cần quan tâm tạo định danh.

Ví dụ: Một khách hàng bao gồm họ tên, tên đường phố, thành phố và bang. Và thông tin địa chỉ nên được đặt trong một đối tượng, và đối tượng khách hàng sẽ chứa tham chiếu tới đối tượng địa chỉ đó. Tên đường, thành phố, bang nên hình thành đối tượng Địa chỉ của riêng nó, vì chúng thuộc cùng khái niệm với nhau, thay vì đặt chúng là các thuộc tính rời của đối tượng Khách hàng, như hình vẽ dưới đây.



## **Service**

Ví dụ như việc chuyển tiền từ một tài khoản này sang một tài khoản khác, chức năng này nên đặt ở tài khoản gửi hay tài khoản nhận? Trong trường hợp này cả hai đều không phù hợp. Vì vậy, cách tốt nhất là khai báo chúng như là một service.

Một service nhiệm vụ của nó là cung cấp chức năng cho domain. Service có vai trò cực kỳ quan trọng trong domain. Chúng thể hiện cho những hành vi quan trọng trong domain nên không thể bỏ qua chúng hoặc gắn vào các Entity hay Value Object.. Nếu gán các chức năng này vào Entity hoặc Value Object sẽ làm mất ý nghĩa của đối tượng đó.

Một service không nên bao gồm các thao tác vốn thuộc về các đối tượng của domain. Không phải cứ có bất kỳ thao tác nào thì chúng ta cũng tạo ra service. Chỉ khi một thao tác đóng một vai trò quan trọng trong domain ta mới cần tạo một service để thực hiện. Service có ba đặc điểm:

+ Các thao tác của một Dịch vụ khó có thể gán cho một Thực thể hay Đối tượng giá trị nào.

+ Các thao tác này tham chiếu đến các đối tượng khác của domain.

+ Thao tác này không mang trạng thái (stateless).

## **Module**

Với một hệ thống lớn, việc chia một mô hình lớn thành nhiều module là việc cần thiết. Các Entity, Value Object, Service, Module là những thành phần chính tạo nên mô hình.

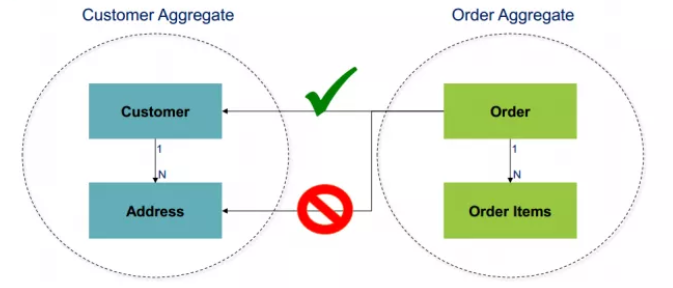
Module được dùng rộng rãi trong hầu hết các dự án. Sẽ dễ dàng hơn để hình dung mô hình lớn nếu ta nhìn vào những module chứa trong mô hình đó, sau đó là quan hệ giữa các module.

Tên module cũng là một phần trong Ubiquitous Language (ngôn ngữ chung).

## **Aggregate**aggregate.jpg

Một Aggregate là một nhóm các đối tượng, nhóm này có thể được xem như là một đơn vị thống nhất đối với các thay đổi dữ liệu. Một Aggregate được phân tách với phần còn lại của hệ thống, ngăn cách giữa các đối tượng nội tại và các đối tượng ở ngoài. Mỗi Aggregate có một "gốc" (Aggregate root), đó là một Entity và cũng là đối tượng duy nhất có thể truy cập từ phía ngoài của Aggregate. Gốc của Aggregate có thể chứa những tham chiếu đến các đối tượng khác trong Aggregate, và những đối tượng trong này có thể chứa tham chiếu đến nhau, nhưng các đối tượng ở ngoài chỉ có thể tham chiếu đến gốc của Aggreagte. Nếu như trong Aggregate có những Entity khác thì định danh của chúng là nội tại, chỉ mang ý nghĩa trong Aggregate.

Các đối tượng khác chỉ có thể tham chiếu đến Aggregate Root. Tất cả các thay đổi của Aggregate chỉ thực hiện được thông qua Aggregate Root. Nếu như Aggregate Root bị xóa và loại bỏ khỏi bộ nhớ thì những đối tượng khác trong Aggregate cũng sẽ bị xóa, vì không còn đối tượng nào chứa tham chiếu đến chúng.



## **Factory**

Khi việc khởi tạo Entity và Aggregate trở nên phức tạp, thì chúng ta nên chuyển giao việc khởi tạo này cho Factory.

Factory được dùng để gộp kiến thức cần cho việc tạo đối tượng, và chúng đặc biệt hữu dụng cho Aggregate. Khi gốc của Aggregate được tạo ra, mọi đối tượng chứa trong Aggregate đó cũng được tạo ra cùng cùng những yếu tố bất biến.

Đây là cách áp dụng Design Pattern “Factory” khá nổi tiếng và phổ biến vào kiến trúc chung của DDD.

## **Repository**

Repository là kho chứa cho bạn lấy ra hay lưu lại các aggregate. Các đối tượng domain sẽ không cần phải xử lý với infrastructure để lấy những tham chiếu cần thiết tới các đối tượng khác của domain. Chúng sẽ chỉ lấy nó từ Repository.

Repository hoạt động như một nơi lưu trữ các đối tượng cho việc truy xuất đối tượng toàn cục.

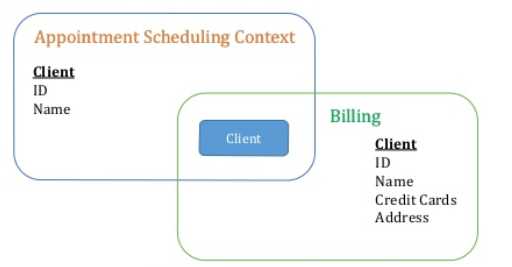
Có một mối quan hệ giữa Factory và Repository. Chúng đều là một pattern của thiết kế hướng lĩnh vực, và chúng cùng giúp ta quản lý vòng đời của các domain đối tượng. Trong khi Factory liên quan tới việc khởi tạo đối tượng, thì Repository lo liệu các đối tượng đã tồn tại.

# **Bounded Context (Phạm vi ngữ cảnh)**

Phạm vi ngữ cảnh có thể được xem như một ứng dụng thu nhỏ, chứa những model của chính nó và mã nguồn cùng cơ chế lưu trữ (persistence) riêng. Bên trong phạm vi ngữ cảnh cần có sự thống nhất hợp lý. Mỗi bounded contexts nên độc lập với bất kỳ bounded contexts khác.

Chia một ứng dụng lớn trong những bối cảnh hợp lý khác nhau sẽ cho chúng ta khả năng module hóa hệ thống, sẽ giúp ta phân tách mối quan tâm khác nhau vào những phần riêng biệt và làm cho các ứng dụng dễ dàng để quản lý và nâng cấp.

Ví dụ: cùng là 1 Client nhưng ở 2 ngữ cảnh khác nhau thì thiết kế cũng sẽ khác nhau.



# **Microservices and Domain Driven Design**

Trong microservices, service được xây dựng chỉ để phục vụ cho một công việc. Các service hoàn toàn tách biệt với nhau. DDD có thể giúp chúng ta giữ được scope của 1 service nhỏ thông qua khái niệm được gọi là “Bounded Context”.

DDD giúp bạn biết được Domain và các Subdomain thông qua việc giao tiếp với các domain expert. Khi mà hiểu rõ về Domain và các Subdomain, bạn sẽ biết cách tất cả Subdomain tương tác với nhau. Điều này giúp bạn trong việc thiết kế microservices architecture và triển khai. Tất cả điều trên đều phụ thuộc vào kiến thức bạn có về Domain mà bạn đang làm.

DDD chỉ nên được áp dụng chỉ khi bạn đang triển khai microservices phức tạp với các tính năng quan trọng.