

**BỘ GIÁO DỤC VÀ ĐÀO TẠO**  
**TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG**



ISO 9001:2008

**ĐỖ VĂN TUYỀN**

**LUẬN VĂN THẠC SĨ**  
**NGÀNH HỆ THỐNG THÔNG TIN**

**Hải Phòng – 2016**

BỘ GIÁO DỤC VÀ ĐÀO TẠO  
TRƯỜNG ĐẠI HỌC DÂN LẬP HẢI PHÒNG

ĐỖ VĂN TUYẾN

CHIẾN LƯỢC THIẾT KẾ LĨNH VỰC VÀ ỨNG  
DỤNG PHẦN MỀM QUẢN LÝ NGƯỜI DÙNG  
TẬP TRUNG

LUẬN VĂN THẠC SĨ  
NGÀNH CÔNG NGHỆ THÔNG TIN

CHUYÊN NGÀNH: HỆ THỐNG THÔNG TIN

MÃ SỐ: 60 48 01 04

NGƯỜI HƯỚNG DẪN KHOA HỌC:

TS. LÊ VĂN PHÙNG

## **LỜI CẢM ƠN**

Trước tiên tôi xin được bày tỏ sự trân trọng và lòng biết ơn đối với TS. Lê Văn Phùng. Trong thời gian học tập và làm luận văn tốt nghiệp, thầy đã dành nhiều thời gian quý báu, tận tình chỉ bảo và hướng dẫn tôi trong việc nghiên cứu, thực hiện luận văn.

Tôi xin được cảm ơn các GS, TS, các thầy cô giáo đã giảng dạy tôi trong quá trình học tập và làm luận văn. Các thầy cô đã giúp tôi hiểu sâu sắc và thấu đáo hơn lĩnh vực mà mình nghiên cứu để có thể vận dụng các kiến thức đó một cách hiệu quả nhất vào trong công tác của mình.

Xin cảm ơn các bạn bè, đồng nghiệp và nhất là các thành viên trong gia đình đã tạo mọi điều kiện tốt nhất, giúp đỡ, động viên, ủng hộ và cổ vũ tôi trong suốt quá trình học tập và nghiên cứu để hoàn thành tốt bản luận văn tốt nghiệp này.

**Tác giả**

**Đỗ Văn Tuyên**

## **LỜI CAM ĐOAN**

Tôi xin cam đoan rằng, đây là công trình nghiên cứu của tôi trong đó có sự giúp đỡ rất lớn của thầy hướng dẫn và các đồng nghiệp ở cơ quan. Các nội dung nghiên cứu và kết quả trong đề tài này là hoàn toàn trung thực.

Trong luận văn, tôi có tham khảo đến một số tài liệu của một số tác giả đã được liệt kê tại phần Tài liệu tham khảo ở cuối luận văn.

*Hải Phòng, ngày 10 tháng 12 năm 2016*

**Tác giả**

**Đỗ Văn Tuyên**

## MỤC LỤC

LỜI CẢM ƠN.....	i
LỜI CAM ĐOAN .....	iv
MỤC LỤC .....	v
DANH MỤC TỪ VIẾT TẮT .....	vii
Danh mục hình.....	vii
PHẦN MỞ ĐẦU.....	x
<b>Lý do chọn đề tài.....</b>	<b>x</b>
<b>Mục đích nghiên cứu .....</b>	<b>xi</b>
<b>Đối tượng và phạm vi nghiên cứu .....</b>	<b>xi</b>
<b>Phương pháp nghiên cứu .....</b>	<b>xi</b>
<b>Những nội dung chính của luận văn .....</b>	<b>xi</b>
Chương 1.....	1
Tổng quan về các tiến trình phát triển phần mềm.....	1
và các chiến lược thiết kế.....	1
<b>1.1. Tổng quan về các tiến trình phát triển phần mềm và kỹ nghệ phần mềm hướng đối tượng.....</b>	<b>1</b>
1.1.1. Tiến trình phát triển phần mềm .....	1
1.1.2. Kỹ nghệ phần mềm hướng đối tượng.....	11
<b>1.2. Các cách tiếp cận thiết kế phần mềm.....</b>	<b>16</b>
<b>1.3. Một số chiến lược hiện đại để thiết kế phần mềm .....</b>	<b>18</b>
1.3.1.Thiết kế phần mềm hướng mô hình.....	18
1.3.2. Thiết kế phần mềm hướng dữ liệu.....	19
1.3.3. Thiết kế phần mềm hướng Trách nhiệm .....	23
1.3.4. Thiết kế phần mềm hướng kiểm thử .....	26

1.3.5. Thiết kế phần mềm hướng lĩnh vực.....	33
<b>KẾT LUẬN CHƯƠNG .....</b>	<b>33</b>
Chương 2.....	35
Chiến lược thiết kế phần mềm hướng lĩnh vực.....	35
<b>2.1. Cách tiếp cận hướng lĩnh vực trong tiến trình phát triển phần mềm ....</b>	<b>35</b>
2.1.1. Khái niệm về thiết kế hướng lĩnh vực .....	35
2.1.2. Tìm hiểu về lĩnh vực.....	36
2.1.3. Ngôn ngữ chung .....	38
<b>2.2. Các đặc trưng thiết kế phần mềm hướng lĩnh vực .....</b>	<b>40</b>
2.2.1 Thực thể.....	43
2.2.2 Đối tượng giá trị .....	45
2.2.2 Dịch vụ .....	47
2.2.3 Mô-đun .....	50
<b>2.3. Các mô hình trong chiến lược thiết kế phần mềm hướng lĩnh vực.....</b>	<b>52</b>
2.3.1 Aggregates and Aggregate Roots .....	53
2.3.2 Factory.....	56
2.3.3. Repository.....	60
2.3.4 Bounded Contexts .....	65
<b>2.4. Quy trình phân tích và thiết kế phần mềm hướng lĩnh vực .....</b>	<b>67</b>
Chương 3: Ứng dụng chiến lược thiết kế hướng lĩnh vực trong việc xây dựng phần mềm quản lý tài khoản tập trung theo hướng dịch vụ microservice.....	69
<b>3.1 Mô tả bài toán quản lý tài khoản dùng chung tại trường ĐHDL Hải Phòng .....</b>	<b>69</b>
<i>Đề xuất giải pháp cho các vấn đề đặt ra: .....</i>	<i>70</i>
<b>3.2 Tìm hiểu kiến trúc Microservices.....</b>	<b>70</b>

3.3 Tìm hiểu mô hình Publisher – Subscriber Event.....	75
3.4 Phân tích và thiết kế yêu cầu phần mềm hướng lĩnh vực .....	76
3.5. Cài đặt và đánh giá phần mềm thử nghiệm .....	87
Đánh giá và kết luận .....	94
TÀI LIỆU THAM KHẢO .....	95

## DANH MỤC TỪ VIẾT TẮT

DDD	Domain Driven Design
RAD	Rapid Application Development
PO	People-oriented
MDD	Model Driven Design
MDA	Model Driven Achitecture
CSDL	Cơ sở dữ liệu
TDD	Test-Driven Development
BDD	Behavior-Driven Development
AMS	Account Management System

## Danh mục hình

Hình 1- 1 Quá trình phát triển phần mềm .....	1
Hình 1- 2 Mô hình thác nước .....	2
Hình 1- 3 Mô hình chữ V .....	3
Hình 1- 4 Mô hình xoắn ốc .....	4
Hình 1- 5 Mô hình tiếp cận lặp .....	5
Hình 1- 6 Mô hình tăng trưởng .....	6
Hình 1- 7 Mô hình phát triển ứng dụng nhanh.....	7
Hình 1- 8 Mô hình Agile.....	9
Hình 1- 9 Mô hình SCRUM.....	10
Hình 2- 1. Mô hình ngôn ngữ chung.....	38
Hình 2- 2 Kiến trúc phân lớp .....	41

Hình 2- 3 Mô hình 3 lớp.....	43
Hình 2- 4 Value Object .....	47
Hình 2- 5 Những mẫu sử dụng trong DDD.....	52
Hình 2- 6 Aggregate root .....	55
Hình 2- 7 Factory .....	58
Hình 2- 8 Repository .....	63
Hình 2- 9 Cài đặt repository.....	64
Hình 2- 10 Quy trình thiết phát triển phần mềm theo hướng DDD.....	68
Hình 3- 1Quy trình phát triển TDDđề cập vấn đề khó khăn trong việc hiểu rõ yêu cầu chức năng trước khi viết kịch bản kiểm thử.....	27
Hình 3- 2 TDD trong Agile framework phác họa bởi Mohammad Sami .....	29
Hình 3- 3 Mô hình BDD – TDD trong Agile mô phỏng bởi Paul Littlebury .....	30
Hình 3- 4 Factory .....	59
Hình 3- 5 Quá trình phát triển các mô hình ứng dụng phần mềm của nhà trường ...	69
Hình 3- 6 Microservices của một công ty điều hành taxi kiểu Uber, Hailo [13].....	71
Hình 3- 7Mô hình Publisher – Subscriber Events.....	75
Hình 3- 8 Mô hình kiến trúc liên lạc .....	75
Hình 3- 9 Usecase của hệ thống.....	77
Hình 3- 10Mô hình kiến trúc của hệ thống hướng Microservice.....	79
Hình 3- 11 Mô hình DDD .....	79
Hình 3- 12 DDD của dịch vụ Profile .....	80
Hình 3- 13Profile Usecase .....	81
Hình 3- 14DDD Account .....	82
Hình 3- 15 Account Usecase.....	82
Hình 3- 16 Tạo tài khoản người dùng .....	83
Hình 3- 17Mô hình DDD của dịch vụ Authenticate .....	84
Hình 3- 18Authenticate Usecase.....	84
Hình 3- 19Mô hình DDD của dịch vụ ApplicationRole .....	85
Hình 3- 20 Mô hình DDD của ApplicationRole .....	85



Hình 3- 21 Register Account.....	87
Hình 3- 22 Change password .....	87
Hình 3- 23 Xóa một Profile.....	88
Hình 3- 24 Các Envets .....	89
Hình 3- 25 Cấu trúc thư mục code chương trình .....	90
Hình 3- 26 Danh sách các Model.....	91

# PHẦN MỞ ĐẦU

## Lý do chọn đề tài

Gần đây các tổ chức, doanh nghiệp, nhóm phát triển phần mềm thường chọn *Domain Driven Design* (DDD) làm phương pháp chính trong việc thiết kế phần mềm. Khác với các phương pháp thiết kế phần mềm truyền thống, DDD tập trung vào việc hiểu vấn đề khách hàng cần giải quyết. Nó đặt yêu cầu của khách hàng vào trung tâm của quá trình thiết kế phần mềm. Theo quan điểm đó, nhóm phát triển tiến hành trao đổi với khách hàng để tìm hiểu về lĩnh vực (domain) hoạt động, các quy trình nghiệp vụ và vấn đề mà họ đang gặp phải. Mô hình DDD được hình thành xoay quanh các đối tượng và nghiệp vụ nhằm giải quyết các vấn đề của khách hàng.

Thông qua mô hình DDD, một ngôn ngữ chung (*Ubiquitous language*) được thiết lập cho mọi đối tượng tham gia vào phát triển phần mềm: nhóm thiết kế, nhóm lập trình, nhóm kiểm thử và cả khách hàng. Phương pháp thiết kế tiếp cận theo lĩnh vực làm đơn giản hóa các bài toán có nghiệp vụ lớn và phức tạp, đồng thời cung cấp cái nhìn sâu vào hành vi nghiệp vụ trong một cách như nhau để dễ hiểu hơn cho cả nhân viên nghiệp vụ và kỹ thuật khi phát triển phần mềm.

Khi thiết kế các hệ thống lớn, số lượng người dùng lớn có nhiều chức năng, nghiệp vụ phức tạp thì module quản lý người dùng là nền tảng vì cung cấp khả năng quản lý toàn bộ người dùng mà các modul được phát triển sau đều phải sử dụng. Đối với một hệ thống phức tạp, có tính thay đổi nhanh, vòng đời ngắn, nhóm phát triển không thể dự đoán trước mọi yêu cầu mong muốn của khách hàng. Liệu việc thiết kế phần mềm theo hướng DDD có thể giải quyết được vấn đề này?. Khả năng thích ứng, linh hoạt của phần mềm theo DDD trước những thay đổi, những yêu cầu mới của khách hàng sẽ như thế nào và các bước nào sẽ phải triển khai khi xây dựng ngôn ngữ dùng chung cho nhóm phát triển đối với một phần mềm cụ thể?

Đó cũng là lý do mà tôi chọn đề tài “*Chiến lược thiết kế lĩnh vực và ứng dụng phần mềm quản lý người dùng tập trung.*” nhằm tìm hiểu, giải quyết và trả lời những câu hỏi được nêu ở trên.

## **Mục đích nghiên cứu**

Nghiên cứu bản chất của chiến lược hướng lĩnh vực, khả năng ứng dụng của nó trong việc phát triển phần mềm quản lý người dùng tập trung tại trường Đại học dân lập Hải Phòng.

## **Đối tượng và phạm vi nghiên cứu**

Đối tượng nghiên cứu là chiến lược thiết kế hướng lĩnh vực (DDD). Phạm vi nghiên cứu là trong kỹ nghệ phát triển phần mềm và ứng dụng trong môi trường trường đại học.

## **Phương pháp nghiên cứu**

Sưu tập tổng hợp lý thuyết về: Tiến trình phát triển phần mềm, chiến lược thiết kế phần mềm theo DDD

Thử nghiệm: Xây dựng phần mềm quản lý người dùng tập trung theo mô hình DDD. Phân tích, so sánh định tính với các chiến lược thiết kế khác

## **Những nội dung chính của luận văn**

Bố cục của luận văn gồm có 3 chương:

*Chương 1:* Tổng quan về các tiến trình phát triển phần mềm và các chiến lược thiết kế: tiến trình phát triển phần mềm, kỹ nghệ phần mềm hướng đối tượng, chiến lược thiết kế phần mềm, một số chiến lược thiết kế phần mềm phổ biến.

*Chương 2:* Chiến lược thiết kế phần mềm hướng lĩnh vực: cách tiếp cận hướng lĩnh vực trong tiến trình phát triển phần mềm, các đặc trưng thiết kế phần mềm hướng lĩnh vực, các mô hình trong chiến lược thiết kế phần mềm hướng lĩnh vực, quy trình phân tích và thiết kế phần mềm hướng lĩnh vực.

*Chương 3:* Ứng dụng chiến lược thiết kế hướng lĩnh vực trong việc xây dựng phần mềm quản lý tài khoản dùng chung: mô tả bài toán quản lý tài khoản dùng chung tại trường ĐHDL Hải Phòng, phân tích và thiết kế yêu cầu phần mềm hướng lĩnh vực, một số giao diện tiêu biểu của phần mềm, cài đặt và đánh giá phần mềm thử nghiệm, đồng thời đưa ra những vấn đề nghiên cứu tiếp theo cho tương lai.

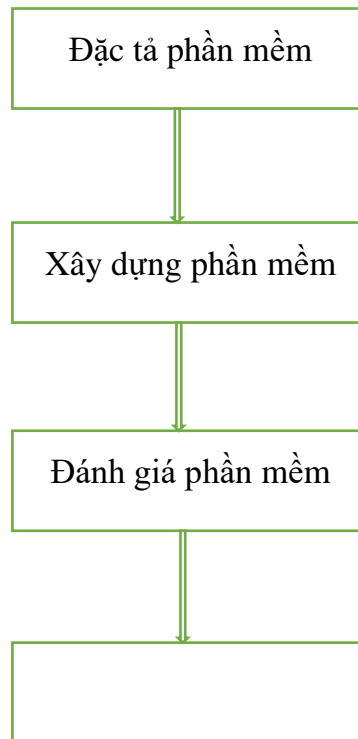
# Chương 1

## Tổng quan về các tiến trình phát triển phần mềm và các chiến lược thiết kế

### 1.1. Tổng quan về các tiến trình phát triển phần mềm và kỹ nghệ phần mềm hướng đối tượng

#### 1.1.1. Tiến trình phát triển phần mềm

Một tiến trình phát triển phần mềm là một tập của các hoạt động cần thiết (đặc tả, xây dựng, đánh giá, tiến hóa) để chuyển các yêu cầu của người dùng thành một hệ thống phần mềm đáp ứng được các yêu cầu đặt ra.



Hình 1- 1 Quá trình phát triển phần mềm

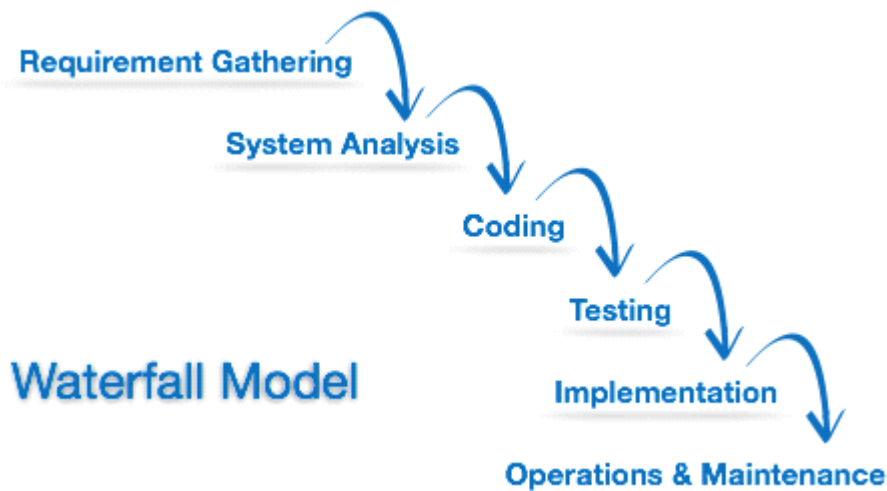
Vòng đời phát triển của phần mềm được chia thành 4 pha [4]:

- Đặc tả phần mềm: Định nghĩa được các chức năng, điều kiện hoạt động của phần mềm.
- Phát triển phần mềm: Là quá trình xây dựng các đặc tả.

- Đánh giá phần mềm: Phần mềm phải được đánh giá để chắc chắn rằng ít nhất có thể thực hiện những gì mà tài liệu đặc tả yêu cầu.
- Tiến hóa phần mềm: Đây là quá trình hoàn thiện các chức năng cũng như giao diện để ngày càng hoàn thiện phần mềm cũng như các yêu cầu đưa ra từ phía khách hàng.

### Các mô hình phát triển trong dự án phần mềm [12]:

a) Waterfall model- Mô hình thác nước:



Hình 1- 2Mô hình thác nước

#### Mô tả:

- Mô hình thác nước là mô hình áp dụng theo tính tuần tự của các giai đoạn phát triển phần mềm.
- Có nghĩa là: giai đoạn sau chỉ được thực hiện tiếp khi giai đoạn trước đã kết thúc.
- Không được quay lại giai đoạn trước để xử lý các thay đổi trong yêu cầu
- Đây được coi là mô hình phát triển phần mềm đầu tiên.

#### Áp dụng:

- Thường được áp dụng cho các dự án không thường xuyên bị thay đổi về yêu cầu.

#### Đặc điểm:

#### + Ưu điểm:

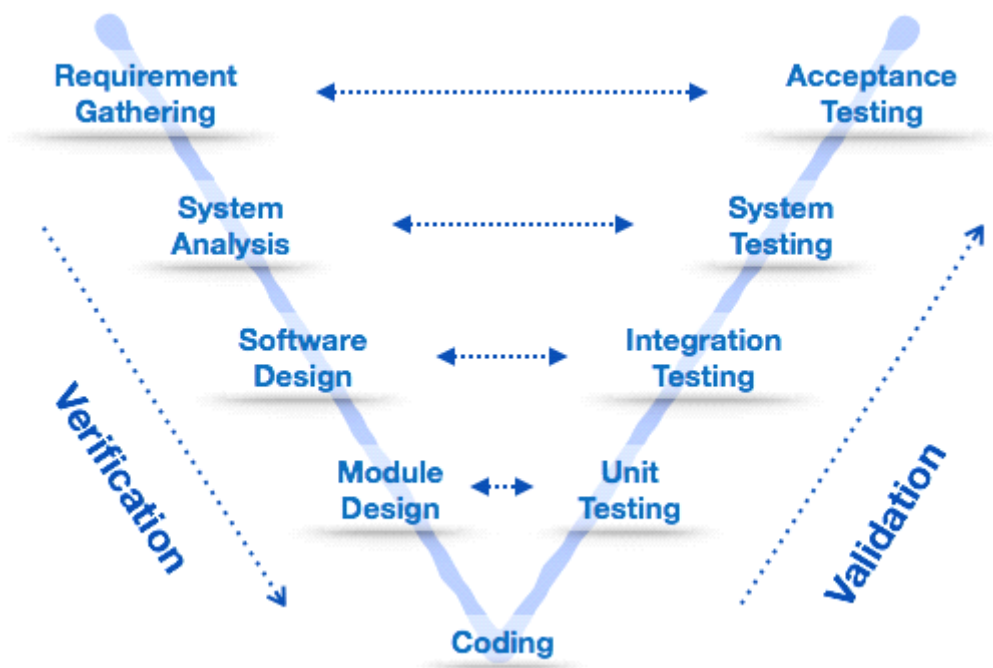
- o Dễ sử dụng, dễ tiếp cận

- Các giai đoạn và hoạt động được xác định rõ ràng
- Xác nhận ở từng giai đoạn, đảm bảo phát hiện sớm các lỗi

**+ Nhược điểm:**

- Rất khó để quay lại giai đoạn nào khi nó đã kết thúc
- Ít tính linh hoạt và phạm vi điều chỉnh của nó khá là khó khăn, tốn kém.

b) V- Shaped Model- Mô hình chữ V



Hình 1- 3 Mô hình chữ V

**Mô tả:**

- Đây là mô hình mở rộng từ mô hình thác nước
- Thay vì di chuyển xuống theo tuần tự các bước thì quy trình sẽ đi theo hình chữ V

**Áp dụng:**

- Yêu cầu phần mềm phải xác định rõ ràng
- Công nghệ phần mềm và các công cụ phải được tìm hiểu kĩ

**Đặc điểm:**

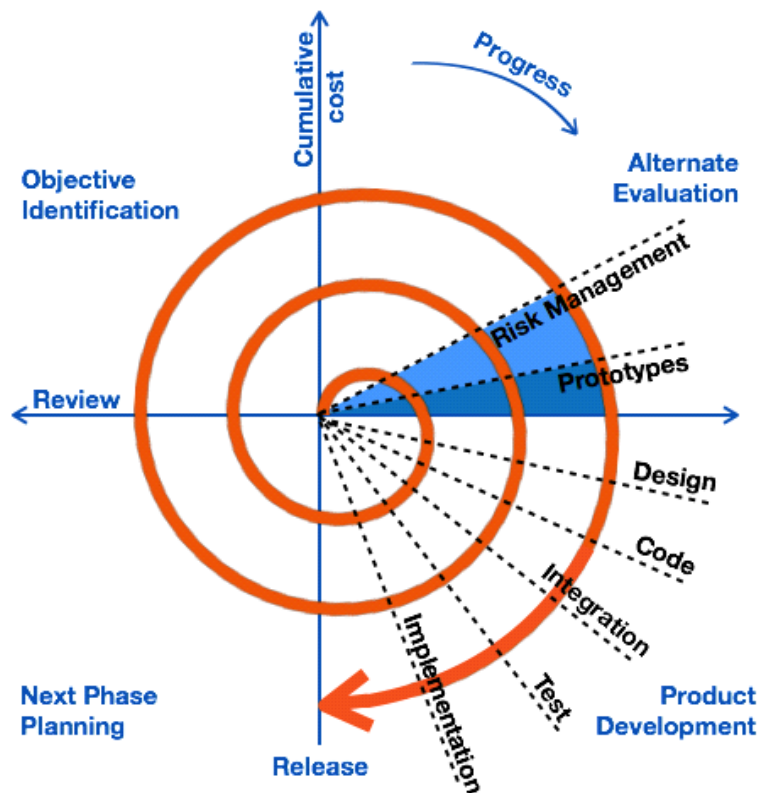
**+ Ưu điểm:**

- Đơn giản dễ sử dụng
- Phân phối cụ thể theo mỗi giai đoạn
- Thực hiện verification và validation sớm trong mỗi giai đoạn phát triển

**+ Nhược điểm:**

- Phạm vi điều chỉnh khá là khó khăn và tốn kém.

c) Spiral Model – Mô hình xoắn ốc



Hình 1- 4 Mô hình xoắn ốc

**Mô tả:**

- Là mô hình kết hợp giữa các tính năng của mô hình prototyping và mô hình thác nước.
- Mô hình xoắn ốc được ưa chuộng cho các dự án lớn, đắt tiền và phức tạp.
- Mô hình này sử dụng nhiều những giai đoạn tương tự như mô hình thác nước, về thứ tự, plan, đánh giá rủi ro,

**Áp dụng:**

- Thường được sử dụng cho các ứng dụng lớn và các hệ thống được xây dựng theo các giai đoạn nhỏ hoặc theo các phân đoạn

**Đặc điểm:**

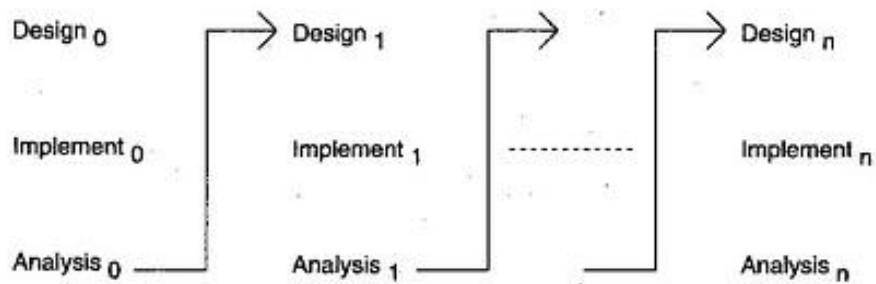
**+ Ưu điểm:**

- Estimates (i.e. budget, schedule, etc.) trở nên thực tế hơn như là một quy trình làm việc, bởi vì những vấn đề quan trọng đã được phát hiện sớm hơn.
- Có sự tham gia sớm của developers
- Quản lý rủi ro và phát triển hệ thống theo phase

**+ Nhược điểm:**

- Chi phí cao và thời gian dài để có sản phẩm cuối cùng
- Phải có kỹ năng tốt để đánh giá rủi ro và giả định.

d) Iterative Model- Mô hình tiếp cận lặp



*Hình 1- 5 Mô hình tiếp cận lặp*

- Một mô hình được lặp đi lặp lại từ khi start cho đến khi làm đầy đủ spec
- Thay vì phát triển phần mềm từ spec đặc tả rồi mới bắt đầu thực thi thì mô hình này có thể review dần dần để đi đến yêu cầu cuối cùng.
- Quy trình phát triển được lặp đi lặp lại cho mỗi một version của sản phẩm trong mỗi chu kỳ.



- **Áp dụng:**

- Yêu cầu của hệ thống đã hoàn chỉnh, được xác định rõ ràng và dễ hiểu
- Yêu cầu chính cần được xác định và một số chi tiết có thể được đổi mới theo thời gian

**Đặc điểm**

+ **Ưu điểm:**

- o Xây dựng và hoàn thiện các bước sản phẩm theo từng bước
- o Nhận được phản hồi của người sử dụng từ những bản phác thảo
- o Thời gian làm tài liệu sẽ ít hơn so với thời gian thiết kế

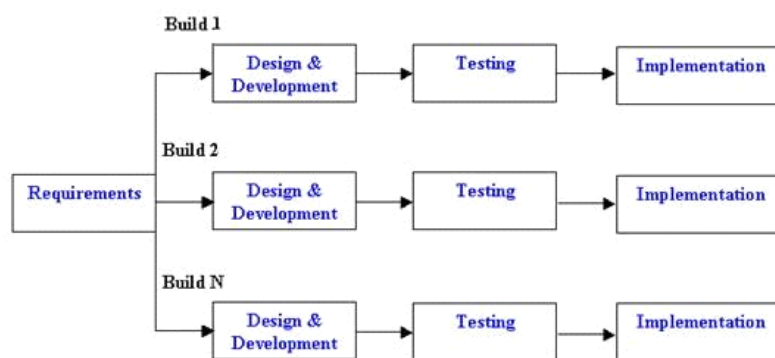
+ **Nhược điểm:**

- o Mỗi giai đoạn lặp lại thì cứng nhắc
- o Tôn kiến trúc hệ thống hoặc thiết kế các vấn đề có thể phát sinh nhưng không phải tất cả đều xảy ra trong toàn bộ vòng đời.

e) Incremental Model – Mô hình tăng trưởng



Ví dụ:



Incremental Life Cycle Model

Hình 1- 6 Mô hình tăng trưởng

**Mô tả:**

- Trong mô hình này thì spec được chia thành nhiều phần.
- Chu kỳ được chia thành các module nhỏ, dễ quản lý.

- Mỗi môđun sẽ đi qua các yêu cầu về thiết kế, thực hiện, ... như 1 vòng đời phát triển thông thường.

**Áp dụng:**

- Áp dụng cho những dự án có yêu cầu đã được mô tả, định nghĩa và hiểu một cách rõ ràng
- Có nhu cầu về sản phẩm sớm

**Đặc điểm:**

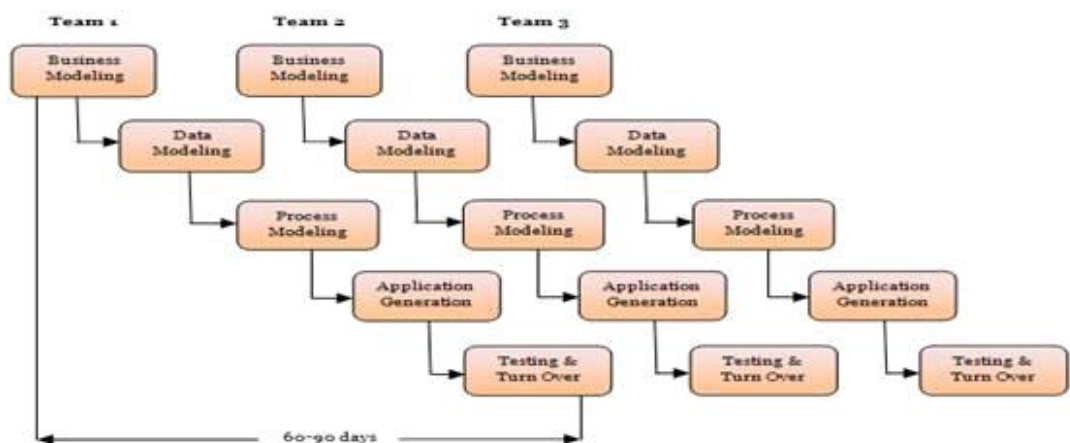
**+ Ưu điểm:**

- o Phần mềm làm việc một cách nhanh chóng trong suốt vòng đời phát triển
- o Mô hình này linh hoạt hơn, ít tốn kém hơn để thay đổi phạm vi và yêu cầu
- o Dễ dàng hơn trong việc kiểm tra và sửa lỗi với sự lặp lại nhỏ hơn

**+ Nhược điểm:**

- o Cần lập plan và thiết kế tốt
- o Cần một định nghĩa rõ ràng và đầy đủ của toàn bộ hệ thống trước khi nó có thể được chia nhỏ và được xây dựng từng bước
- o Tổng chi phí là cao hơn so với thác nước.

f) RAD Model (Rapid Application Development)



Hình 1- 7 Mô hình phát triển ứng dụng nhanh

**Mô tả:**

- Là một dạng của incremental model

- Trong mô hình RAD các thành phần hoặc chức năng được phát triển song song như thể chúng là các dự án nhỏ
- Việc phát triển này theo thời gian nhất định, cung cấp và lắp ráp thành một nguyên mẫu làm việc
- Điều này có thể nhanh chóng đưa ra một cái gì đó cho khách hàng để xem và sử dụng và cung cấp thông tin phản hồi liên quan đến việc cung cấp và yêu cầu của họ.

**Áp dụng:**

- RAD nên được sử dụng khi có nhu cầu để tạo ra một hệ thống có Modularized trong khoảng thời gian 2-3 tháng.
- Nên được sử dụng khi đã có sẵn designer cho model và chi phí cao

**Đặc điểm:**

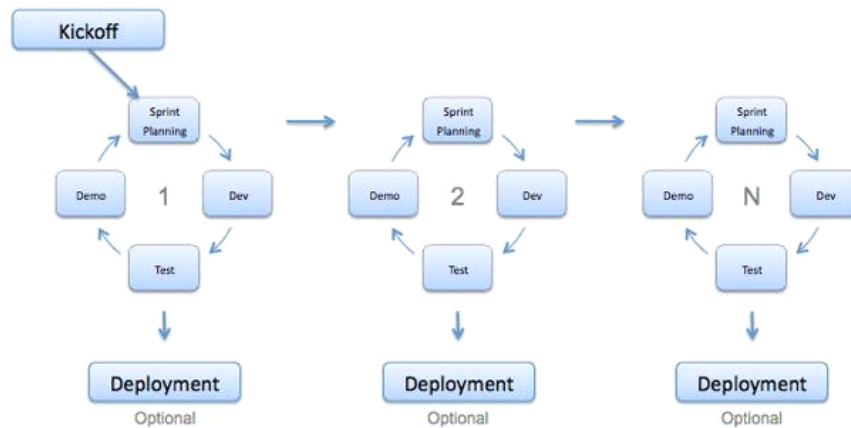
**+ Ưu điểm:**

- Giảm thời gian phát triển.
- Tăng khả năng tái sử dụng của các thành phần
- Đưa ra đánh giá ban đầu nhanh chóng
- Khuyến khích khách hàng đưa ra phản hồi

**+ Nhược điểm:**

- Cần có một team giỏi để xác định yêu cầu phần mềm
- Chỉ những hệ thống có module mới sử dụng được mô hình này
- Yêu cầu về dev/ design phải có nhiều kinh nghiệm
- Phụ thuộc rất nhiều vào kỹ năng model

g) Agile Model



Hình 1- 8Mô hình Agile

**Mô tả:**

- Dựa trên mô hình iterative and incremental
- Các yêu cầu và giải pháp phát triển dựa trên sự kết hợp của các function

**Áp dụng:**

- Nó có thể được sử dụng với bất kỳ loại hình dự án nào, nhưng nó cần sự tham gia và tính tương tác của khách hàng. Ngoài ra, nó có thể được sử dụng khi khách hàng yêu cầu chức năng sẵn sàng trong khoảng thời gian ngắn ( 3 tuần)

**Đặc điểm:**

**+ Ưu điểm:**

- Giảm thời gian cần thiết để tận dụng một số tính năng của hệ thống
- Kết quả cuối cùng là phần mềm chất lượng cao trong thời gian ít nhất có thể và sự hài lòng của khách hàng

**Nhược điểm:**

- Phụ thuộc vào kỹ năng của người phát triển phần mềm Scalability
- Tài liệu được thực hiện ở giai đoạn sau
- Cần một team có kinh nghiệm.

h) Scrum Model



Hình 1- 9 Mô hình SCRUM

**Mô tả:**

Là một quy trình phát triển phần mềm theo mô hình linh hoạt (agile). Với nguyên tắc chủ đạo là chia nhỏ phần mềm cần sản xuất ra thành các phần nhỏ để phát triển (các phần nhỏ này phải độc lập và Release được), lấy ý kiến khách hàng và thay đổi cho phù hợp ngay trong quá trình phát triển để đảm bảo sản phẩm release đáp ứng những gì khách hàng mong muốn. Scrum chia dự án thành các vòng lặp phát triển gọi là các sprint. Mỗi sprint thường mất 2- 4 tuần (30 ngày) để hoàn thành. Nó rất phù hợp cho những dự án có nhiều sự thay đổi và yêu cầu tốc độ cao. Ngoài ra Scrum hoạt động dựa trên ba giá trị cốt lõi, còn gọi là “Ba chân của Scrum” bao gồm Minh bạch, Thanh tra và Thích nghi.

**Đặc điểm:**

- Scrum (hay agile nói chung) được xếp vào nhóm “Feature-driven development”. Sản phẩm được phát triển theo tính năng, chứ không phát triển sản phẩm theo kiến trúc hệ thống.
- Scrum khác với các mô hình Agile ở chỗ nó là mô hình hướng khách hàng (Customer oriented), vai trò của khách hàng trong việc đánh giá sản phẩm rất quan trọng. Chỉ sau mỗi sprint (2-4 tuần) khách hàng sẽ thấy được sự thay đổi của sản phẩm của mình qua đó đưa ra phản hồi sớm để định hướng. → Thích ứng nhanh với sự thay đổi yêu cầu

- Scrum giảm thiểu tài nguyên dành cho việc quản lý mà tập trung nhiều hơn cho những công việc liên quan trực tiếp đến việc làm ra sản phẩm. Bằng cách giảm vai trò quản lý (PM) bằng cách đẩy việc quản lý tới từng người,
- Giảm thời gian dành cho việc viết tài liệu bằng cách tăng thời gian trao đổi trực tiếp. Thông thường khi estimate công việc, thì team estimate cả thời gian dành cho communication để hoàn thành task đó nữa.
- Tập trung vào sản phẩm, sản phẩm mới là đích cuối cùng chứ không phải qui trình.

#### + Ưu điểm:

- Một người có thể làm nhiều việc ví dụ như dev có thể test
- Phát hiện lỗi sớm hơn rất nhiều so với các phương pháp truyền thống
- Khách hàng nhanh chóng thấy được sản phẩm qua đó đưa ra phản hồi sớm.
- Có khả năng áp dụng được cho những dự án mà yêu cầu khách hàng không rõ ràng ngay từ đầu.

#### + Nhược điểm:

- Trình độ của nhóm là có một kỹ năng nhất định
- Phải có sự hiểu biết về mô hình agile .
- Khó khăn trong việc xác định ngân sách và thời gian.
- Luôn nghe ý kiến phản hồi từ khách hàng và thay đổi theo nên thời gian sẽ kéo dài khi có quá nhiều yêu cầu thay đổi từ khách hàng.
- Vai trò của PO rất quan trọng, PO là người định hướng sản phẩm. Nếu PO làm không tốt sẽ ảnh hưởng đến kết quả chung

### 1.1.2. Kỹ nghệ phần mềm hướng đối tượng

Để xây dựng được những hệ thống phần mềm đáp ứng được những yêu cầu chất lượng, ta cần phải:

- Có một qui trình phát triển phần mềm thống nhất gồm các bước thực hiện rõ ràng, mà sau mỗi bước đều phải có các sản phẩm cụ thể.
- Có các phương pháp và kỹ nghệ phù hợp với những pha thực hiện phát triển phần mềm.

- Có công cụ để làm ra sản phẩm phần mềm theo yêu cầu.

Quá trình phát triển một sản phẩm là quá trình định nghĩa ai làm cái gì, làm khi nào và như thế nào. Quá trình xây dựng một sản phẩm phần mềm hoặc nâng cấp một sản phẩm đã có được gọi là *quá trình phát triển phần mềm*.

Hệ thống phần mềm được kiến tạo là sản phẩm của một loạt các hoạt động sáng tạo và có quá trình phát triển. Quá trình phát triển những phần mềm phức tạp phải có nhiều người tham gia thực hiện. Trước hết đó là *khách hàng* và những *nhà đầu tư*, đó là những người đưa ra vấn đề cần giải quyết trên máy tính. Những người phát triển hệ thống gồm các nhà phân tích, thiết kế và lập trình làm nhiệm vụ phân tích các yêu cầu của khách hàng, thiết kế các thành phần của hệ thống và thực thi cài đặt chúng. Sau đó phần mềm được kiểm thử và triển khai ứng dụng để thực thi những vấn đề mà người dùng yêu cầu.

Quá trình phát triển phần mềm được xác định thông qua các hoạt động cần thực hiện để chuyển đổi các yêu cầu của khách hàng thành hệ thống phần mềm. Có năm bước chính cần thực hiện trong quá trình phát triển phần mềm:

- Xác định các yêu cầu
- Phân tích hệ thống
- Thiết kế hệ thống
- Lập trình và kiểm tra hệ thống
- Vận hành và bảo trì hệ thống.

Có thể thực hiện các bước trên theo nhiều phương pháp khác nhau, theo đó số các bước đề xuất của các phương pháp cũng có thể khác nhau.

#### **a) Xác định các yêu cầu hệ thống**

Pha xác định các yêu cầu của hệ thống gồm 2 giai đoạn:

- + Xây dựng mô hình nghiệp vụ
- + Xác định yêu cầu

#### **• Xây dựng mô hình nghiệp vụ**

Việc đi tới nắm bắt rành mạch, rõ ràng về hệ thống cần tin học hóa cần xuất phát từ việc tìm hiểu và nghiên cứu về hệ thống thực. Công việc tìm hiểu này được tiến hành bằng cách tìm hiểu hoạt động nghiệp vụ và xây dựng các mô

hình miền và mô hình nghiệp vụ để nắm bắt được toàn bộ các vấn đề liên quan đến nghiệp vụ của hệ thống. Việc tìm hiểu quy trình nghiệp vụ của hệ thống được tiến hành qua các bước sau:

- Tìm hiểu bối cảnh hệ thống
- Nắm bắt những yêu cầu bổ sung, các yêu cầu phi chức năng.

- **Xác định yêu cầu**

Từ những yêu cầu của khách hàng, ta xác định được mục tiêu của hệ thống cần thực hiện. Thường đó là các yêu cầu chức năng về những gì mà hệ thống phải thực hiện, nhưng chưa cần chỉ ra các chức năng đó thực hiện như thế nào. Việc xác định đúng và đầy đủ yêu cầu bài toán là nhiệm vụ hết sức quan trọng, nó làm cơ sở cho tất cả các bước tiếp theo trong dự án phát triển phần mềm. Muốn vậy, thì phải thực hiện đặc tả chi tiết các yêu cầu của hệ thống. Ngôn ngữ mô hình hóa UML(*Unified Modeling Language*) cũng cung cấp biểu đồ ca sử dụng để đặc tả các yêu cầu của hệ thống.

Các bước trong giai đoạn này gồm các bước sau:

- + Tìm các tác nhân và các ca sử dụng
- + Sắp xếp thứ tự ưu tiên các ca sử dụng
- + Mô tả chi tiết một ca sử dụng
- + Tạo một bản mẫu giao diện người dùng
- + Tái cấu trúc mô hình ca sử dụng

- b) Phân tích hệ thống**

Phân tích hướng đối tượng là một giai đoạn của quá trình phát triển phần mềm, trong đó mô hình khái niệm được mô tả chính xác, xúc tích thông qua các đối tượng thực và các khái niệm của bài toán ứng dụng. Giai đoạn này tập trung vào việc tìm kiếm các đối tượng, khái niệm trong lĩnh vực bài toán và xác định mối quan hệ của chúng trong hệ thống. Nhiệm vụ của người phân tích là nghiên cứu kỹ các yêu cầu của hệ thống và phân tích các thành phần của hệ thống cùng các mối quan hệ của chúng.

Kết quả của chính của pha phân tích hệ thống hướng đối tượng là sơ đồ lớp (sơ lược), sơ đồ trạng thái, sơ đồ trình tự, sơ đồ hành động.



Giai đoạn phân tích hệ thống bao gồm các giai đoạn chính sau:

- Phân tích kiến trúc
- Phân tích một ca sử dụng
- Phân tích một lớp
- Phân tích một gói

### c) **Thiết kế hệ thống**

Dựa vào các đặc tả yêu cầu và các kết quả phân tích thể hiện ở các biểu đồ để thực hiện thiết kế hệ thống. Thiết kế hướng đối tượng là một giai đoạn trong quá trình phát triển phần mềm, trong đó hệ thống được tổ chức thành tập các đối tượng tương tác với nhau và mô tả được các hệ thống thực thi nhiệm vụ của bài toán ứng dụng.

Nhiệm vụ chính của giai đoạn này là xây dựng các thiết kế chi tiết mô tả các thành phần của hệ thống ở mức cao hơn khâu phân tích để phục vụ cho việc cài đặt. Nghĩa là các lớp đối tượng được định nghĩa chi tiết gồm đầy đủ các thuộc tính, thao tác phục vụ cho việc cài đặt. Đồng thời đưa ra được kiến trúc hệ thống để đảm bảo cho hệ thống có thể thay đổi, có tính mở, dễ bảo trì, thân thiện với người dùng,... Nghĩa là các tổ chức các lớp thành các hệ thống con theo một kiến trúc phù hợp với nhu cầu phát triển công nghệ và xu hướng phát triển của lĩnh vực ứng dụng.

Những kết quả được thể hiện trong các biểu đồ: biểu đồ lớp ( chi tiết), biểu đồ công tác thành phần và biểu đồ triển khai.

Pha thiết kế hệ thống này gồm các giai đoạn chính sau:

- Thiết kế kiến trúc
- Thiết kế một ca sử dụng
- Thiết kế một hệ thống con

### d) **Lập trình và kiểm tra vận hành**

Giai đoạn xây dựng phần mềm có thể được hiện sử dụng kỹ thuật lập trình hướng đối tượng. Đó là phương thức thực hiện thiết kế đối tượng qua việc sử dụng một ngôn ngữ lập trình hỗ trợ các tính năng hướng đối tượng. Một vài ngôn ngữ hướng đối tượng thường được nhắc tới như C++,

Java, ... Kết quả chung của giai đoạn này là một loạt các code chạy được, nó chỉ đưa vào sử dụng sau khi đã trải qua nhiều vòng thử nghiệm khác nhau.

Trong giai đoạn này, mỗi thành phần đã được thiết kế sẽ được lập trình thành những mô-đun chương trình (các chương trình con). Mỗi mô-đun này sẽ được kiểm chứng hoặc thử nghiệm theo các tài liệu đặc tả của giai đoạn thiết kế. Sau đó, các mô-đun chương trình đã được kiểm tra sẽ được tích hợp với nhau thành hệ thống tổng thể và được kiểm tra xem có đáp ứng các yêu cầu của khách hàng. Kết quả của giai đoạn này là phần mềm cần được xây dựng.

#### e) Vận hành và bảo trì

Giai đoạn này bắt đầu bằng việc cài đặt hệ thống phần mềm trong môi trường sử dụng của khách hàng sau khi sản phẩm đã được giao cho họ. Hệ thống sẽ hoạt động, cung cấp các thông tin, xử lý các yêu cầu và thực hiện những gì đã được thiết kế.

Tuy nhiên vấn đề bảo trì phần mềm hoàn toàn khác với bảo trì của phần mềm cứng. Như đã phân tích ở trên, bảo trì phần mềm là đảm bảo cho hệ thống, hoạt động đáp ứng được các yêu cầu của người sử dụng, của khách hàng. Mà các yêu cầu này thay đổi hệ thống sao cho nó phù hợp với các yêu cầu hiện tại của họ, thậm chí có những thay đổi chưa phát hiện được trong các pha phân tích, thiết kế. Nghĩa là hệ thống phần mềm phải được nâng cấp, hoàn thiện liên tục và chi phí cho công tác bảo trì là khá tốn kém. Thông thường, có hai loại nâng cấp:

- *Nâng cấp hiệu quả của hệ thống*: Bao gồm những thay đổi mà khách hàng cho là sẽ cải thiện hiệu quả công việc của hệ thống, như bổ sung thêm các chức năng hay giảm thời gian xử lý, trả lời của hệ thống,...
- Đảm bảo sự thích nghi đối với sự thay đổi của môi trường của hệ thống hay sự thay đổi cho phù hợp với những thay đổi của chính sách, qui chế mới ban hành của Chính phủ.

## 1.2. Các cách tiếp cận thiết kế phần mềm

Do các hệ phần mềm lớn là phức tạp nên người ta thường dùng các cách tiếp cận khác nhau trong việc thiết kế các phần khác nhau của một hệ thống. Chẳng có một cách tiếp cận nào tốt nhất chung cho các dự án. Hai cách tiếp cận thiết kế hiện đang được dùng rộng rãi trong việc phát triển phần mềm đó là cách tiếp cận theo hướng chức năng và hướng đối tượng. Mỗi cách tiếp cận đều có ưu và nhược điểm riêng phụ thuộc vào ứng dụng phát triển và nhóm phát triển phần mềm.

Cách tiếp cận hướng chức năng hay hướng đối tượng là bổ sung và hỗ trợ cho nhau chứ không phải là đối kháng nhau. Kỹ sư phần mềm sẽ chọn cách tiếp cận thích hợp nhất cho từng giai đoạn thiết kế.

Cách tiếp cận *Hướng chức năng (Function-oriented (structured) approach)*: Theo hướng chức năng, hệ thống được thiết kế theo quan điểm chức năng, bắt đầu ở mức cao nhất, sau đó tinh chế dần dần để thành thiết kế chi tiết hơn.

Thiết kế hướng chức năng là một cách tiếp cận thiết kế phần mềm trong đó bản thiết kế được phân giải thành một bộ các đơn thể tác động lẫn nhau, mà mỗi đơn thể có một chức năng được xác định rõ ràng. Các chức năng có các trạng thái cục bộ nhưng chúng chia sẻ với nhau trạng thái hệ thống, trạng thái của hệ thống là tập trung và mọi chức năng đều có thể truy cập được.

Cách tiếp cận *hướng đối tượng (Object oriented approach)*: Hệ thống được nhìn nhận như một bộ các đối tượng (*chứ không phải là bộ các chức năng*). Hệ thống được phân tán, mỗi đối tượng có những thông tin trạng thái riêng của nó. Đối tượng là một bộ các thuộc tính xác định trạng thái của đối tượng đó và các phép toán của nó. Nó được thừa kế từ một vài lớp đối tượng lớp cao hơn, sao cho dễ định nghĩa nó chỉ cần nêu đủ các khác nhau giữa nó và các lớp cao hơn nó. Che dấu thông tin là chiến lược thiết kế dấu càng nhiều thông tin trong các thành phần càng hay. Cái đó ngầm hiểu rằng việc kết hợp điều khiển logic và cấu trúc dữ liệu được thực hiện trong thiết kế càng chậm càng tốt. Liên lạc thông qua các thông tin trạng thái dùng chung (các biến tổng thể) là ít nhất, nhờ vậy khả năng hiểu là được tăng lên. Thiết kế là tương đối dễ thay đổi vì sự thay đổi một thành phần không thể không dự kiến các hiệu ứng phụ trên các thành phần khác. Cách tiếp cận hướng đối tượng là dựa

trên việc che dấu thông tin, nhìn hệ phần mềm như là một bộ các đối tượng tương tác với nhau chứ không phải là một bộ các chức năng như cách tiếp cận chức năng. Các đối tượng này có một trạng thái được che dấu và các phép toán trên các trạng thái đó. Thiết kế biểu thị các dịch vụ được yêu cầu và được cung cấp bởi các đối tượng có tương tác với nó.

*Cách tiếp cận hướng đối tượng có ba đặc trưng:*

- + Vùng dữ liệu dùng chung là bị loại bỏ. Các đối tượng liên lạc với nhau bằng cách trao đổi thông báo chứ không phải bằng các biến dùng chung.
- + Các đối tượng là các thực thể độc lập mà chúng sẵn sàng được thay đổi vì rằng tất cả các trạng thái và các thông tin biểu diễn là chỉ ảnh hưởng trong phạm vi chính đối tượng đó thôi. Các thay đổi về biểu diễn thông tin có thể được thực hiện không cần sự tham khảo tới các đối tượng hệ thống khác.
- + Các đối tượng có thể phân tán và có thể hành động tuần tự hoặc song song. Không cần có quyết định về tính song song ngay từ một giai đoạn sớm của quá trình thiết kế.

*Các ưu điểm:*

- + Dễ bảo trì vì các đối tượng là độc lập. Các đối tượng có thể hiểu và cải biên như là một thực thể độc lập. Thay đổi trong thực hiện một đối tượng hoặc thêm các dịch vụ sẽ không làm ảnh hưởng tới các đối tượng hệ thống khác.
- + Các đối tượng là các thành phần dùng lại được thích hợp (do tính độc lập của chúng). Một thiết kế có thể dùng lại được các đối tượng đã được thiết kế trong các bản thiết kế trước đó.
- + Đối với một vài lớp hệ thống, có một phản ánh rõ ràng giữa các thực thể có thực (chẳng hạn như các thành phần phần cứng) với các đối tượng điều khiển nó trong hệ thống. Điều này cải thiện được tính dễ hiểu của thiết kế.

*Nhược điểm:*

Sự tường minh các đối tượng hệ thống thích hợp là khó khăn. Cách nhìn tự nhiên nhiều hệ thống là cách nhìn chức năng và việc thích nghi với cách nhìn hướng đối tượng đôi khi là khó khăn. Cách tiếp cận kế hướng đối tượng vẫn còn là tương đối chưa chín muồi và đang thay đổi mau chóng.

### **1.3. Một số chiến lược hiện đại để thiết kế phần mềm**

#### **1.3.1. Thiết kế phần mềm hướng mô hình**

Phát triển phần mềm truyền thống ngày càng phải đối mặt với nhiều khó khăn như vấn đề quy trình phát triển, vấn đề khả chuyển, vấn đề tính liên tác hay vấn đề làm tài liệu và bảo trì. Chính vì vậy, một xu hướng phát triển phần mềm mới nhằm khắc phục những khó khăn này là kỹ nghệ phát triển phần mềm hướng mô hình (Model Driven Design– MDD).

Hiện nay, các kết quả đạt được của MDD chủ yếu dựa trên kiến trúc phần mềm hướng mô hình (Model Driven Architecture- MDA). Tuy nhiên, trong quá trình chuyển đổi mô hình vẫn còn một số vấn đề tồn tại. Để giải quyết những vấn đề này, trong khung làm việc chuyển đổi mô hình, người ta tập trung vào hai kỹ thuật chính là áp dụng ngôn ngữ chuyển đổi mô hình và áp dụng mẫu thiết kế.

Khung làm việc MDA dựa trên nền tảng các chuẩn UML, MOF, XMI và xoay quanh các ý tưởng chính là mô hình độc lập nền (PIM), mô hình cụ thể nền (PSM) và sự chuyển đổi giữa chúng. Có thể nói, phát triển phần mềm hướng mô hình nói chung và kiến trúc phần mềm hướng mô hình nói riêng hứa hẹn một bước tiến mới trong phát triển phần mềm giúp quá trình phát triển tập trung nhiều hơn vào mô hình, nâng cao chất lượng sản phẩm và năng suất làm việc.

Trong quá trình phát triển phần mềm hướng mô hình, có hai vấn đề nổi lên là việc chuyển đổi mô hình và việc biểu diễn mẫu thiết kế:

##### **1. Vấn đề chuyển đổi mô hình**

Chuyển đổi mô hình là trái tim của kỹ nghệ phần mềm hướng mô hình. Một ví dụ điển hình là các mô hình ở mức trừu tượng cao được chuyển đổi sang các mô hình cụ thể gần với nền phát triển. Tuy nhiên, còn có rất các dạng chuyển đổi khác được áp dụng trong quá trình phát triển phần mềm theo hướng mô hình.

##### **2. Vấn đề biểu diễn mẫu thiết kế**

Mẫu thiết kế là một định dạng chung của các thiết kế có thể tái sử dụng. Một mẫu thiết kế mô tả họ các giải pháp cho một lớp của các vấn đề thiết kế lặp lại. Tuy nhiên, các thông tin không tổng quát của các giải pháp mẫu mô tả làm hạn chế khả năng sử dụng mẫu, như là việc ứng dụng mẫu vào sự phát triển các công cụ hỗ trợ

cách sử dụng mẫu thiết kế trong phát triển phần mềm. Trong đó có một hướng đi mới trong phát triển phần mềm đó là phát triển phần mềm hướng mô hình yêu cầu một sự đặc tả chính xác mẫu thiết kế. Có các phương pháp biểu diễn mẫu theo hai hướng chủ yếu là mô tả mẫu theo nguyên mẫu và đặc tả mẫu theo hướng tự động hóa bằng ngôn ngữ đặc tả mẫu.

Để hiện thực hóa quy trình phát triển phần mềm hướng mô hình, các công cụ phát triển phải hỗ trợ tự động hóa sự chuyển đổi mô hình. Các công cụ này không chỉ cần phải cung cấp khả năng áp dụng những sự chuyển đổi được định nghĩa trước mà còn phải cung cấp một ngôn ngữ cho phép người dùng định nghĩa sự chuyển đổi mô hình và thực thi chúng theo các yêu cầu riêng. Nói cách khác, những cài đặt cho ngôn ngữ chuyển đổi không chỉ hỗ trợ phát triển phần mềm hướng mô hình mà còn phải nâng cao năng suất và chất lượng của sự phát triển.

### **1.3.2. Thiết kế phần mềm hướng dữ liệu**

Ta biết rằng, cấu trúc dữ liệu có tác động quan trọng tới độ phức tạp và tính hiệu quả của thuật toán được thiết kế để xử lý thông tin.

Khi thiết kế phần mềm tiến hoá, một trường phái cho rằng: Việc xác định cấu trúc dữ liệu cố hữu (đối với hệ thống dựa trên máy tính) là điều sống còn, còn cấu trúc của dữ liệu (cái vào và cái ra) có thể được dùng để đưa ra cấu trúc (và một số chi tiết) về chương trình. Trong lĩnh vực ứng dụng một cấu trúc thông tin có cấp bậc, phân biệt là tồn tại. Dữ liệu vào, thông tin ghi nhớ bên trong (như CSDL) và dữ liệu ra có thể có một cấu trúc duy nhất. Thiết kế hướng cấu trúc dữ liệu dùng những cấu trúc này làm nền tảng cho việc phát triển phần mềm.

Cấu trúc dữ liệu phản ánh thiết kế của các khía cạnh cấu trúc và thủ tục của phần mềm. Trong thực tế, cấu trúc thông tin là điều báo trước cho cấu trúc chương trình. Thiết kế hướng cấu trúc dữ liệu biến đổi một biểu diễn của cấu trúc dữ liệu thành biểu diễn của phần mềm. Giống như các kỹ thuật luồng dữ liệu, người phát triển thiết kế hướng cấu trúc dữ liệu xác định ra một tập các thủ tục ánh xạ có dùng cấu trúc (dữ liệu) thông tin như hướng dẫn.

Các đóng góp của thiết kế hướng cấu trúc dữ liệu có thể tìm thấy trong những thảo luận về nền tảng của cấu trúc dữ liệu, thuật toán máy tính, cấu trúc điều khiển

và dữ liệu, và khái niệm về trừu tượng dữ liệu. Những xử lý thực chứng hơn về thiết kế phần mềm và mối quan hệ của nó với cấu trúc dữ liệu đã được Jackson, Warnier và Orr đề nghị. Lập trình có cấu trúc Jackson (JSP), một phương pháp thiết kế phần mềm được sử dụng rộng rãi, lấy quan điểm là sự song song giữa cấu trúc của dữ liệu vào và dữ liệu ra (báo cáo) sẽ đảm bảo chất lượng thiết kế. Những mở rộng gần đây hơn thành phương pháp luận, gọi là phát triển hệ thống Jackson, tập trung vào việc xác định các thực thể thông tin và những hành động được áp dụng lên chúng và hoàn toàn tương tự trong một số khía cạnh của cách tiếp cận thiết kế hướng sự vật đã được mô tả. Jackson nhấn mạnh về mặt thực hành, phát triển thực chứng để biến đổi dữ liệu thành cấu trúc chương trình. Xây dựng logic chương trình (LPC), được J.D.Warnier phát triển, đưa ra một phương pháp chặt chẽ cho thiết kế phần mềm. Rút ra từ mối quan hệ giữa cấu trúc dữ liệu và cấu trúc thủ tục, Warnier đã phát triển một tập các kỹ thuật thực hiện ánh xạ từ cấu trúc dữ liệu vào/ra sang biểu diễn thủ tục chi tiết cho phần mềm. Phát triển hệ thống có cấu trúc dữ liệu (DSSD), cũng còn được gọi là phương pháp luận Warnier Orr, là một mở rộng của LCP và bổ sung thêm các khả năng phân tích cũng như thiết kế mạnh. Cách tiếp cận DSSD đưa ra một phương pháp và nhiều thủ tục để suy ra cấu trúc dữ liệu, cấu trúc chương trình, và thiết kế thủ tục chi tiết cho các thành phần chương trình (mô đun). Bên cạnh đó, DSSD cung cấp một kỹ pháp làm cho người thiết kế có khả năng xem xét luồng dữ liệu giữa nơi phát và nơi nhận thông tin và đi qua các tiến trình biến đổi thông tin. Một kỹ thuật gọi là xây dựng logic phần mềm là đại biểu cho việc tổng hợp của các cách tiếp cận thiết kế hướng luồng dữ liệu và cấu trúc dữ liệu. Những người phát triển phương pháp này cho rằng thiết kế logic có thể được mô tả tường minh nếu phần mềm được xét như một hệ thống các tập dữ liệu và các phép biến đổi dữ liệu.

Thiết kế hướng cấu trúc dữ liệu có thể được áp dụng thành công trong các ứng dụng có cấu trúc thông tin cấp bậc, được xác định rõ. Các thí dụ điển hình bao gồm: ứng dụng hệ thống thông tin kinh doanh, cái vào và cái ra có cấu trúc phân biệt (như tệp vào, báo cáo ra); việc dùng CSDL cấp bậc là thông dụng, các ứng dụng hệ thống. Cấu trúc dữ liệu cho hệ điều hành có chứa nhiều bảng, tệp và danh sách có

cấu trúc xác định rõ, các ứng dụng CAD/CAE/CIM. Các hệ thống thiết kế, kỹ nghệ và chế tạo có máy tính trợ giúp đòi hỏi các cấu trúc dữ liệu phức tạp để ghi nhớ, chuyển dịch và xử lý thông tin.

Cả hai cách thiết kế hướng cấu trúc dữ liệu và luồng dữ liệu đều bắt đầu từ cách phân tích để đặt nền tảng cho các bước thiết kế tiếp; cả hai đều hướng theo thông tin; cả hai đều định biến đổi thông tin thành biểu diễn phần mềm; cả hai đều dựa trên các khái niệm suy diễn tách biệt về thiết kế tốt. Thiết kế hướng cấu trúc dữ liệu không dùng biểu đồ luồng dữ liệu một cách tường minh. Do đó, các phân loại phép biến đổi và luồng giao tác ít có liên can tới phương pháp thiết kế hướng cấu trúc dữ liệu. Điều quan trọng hơn là mục tiêu cuối cùng của phương pháp hướng cấu trúc dữ liệu là tạo ra một mô tả thủ tục cho phần mềm. Khái niệm về cấu trúc mô đun chương trình không được xem xét một cách tường minh. Các mô đun được coi như các thứ phẩm của thủ tục và triết lý về sự độc lập của mô đun cũng ít được nhấn mạnh tới. Thiết kế hướng cấu trúc dữ liệu dùng biểu đồ phân cấp để biểu diễn cấu trúc thông tin.

Thiết kế hướng cấu trúc dữ liệu và thiết kế hướng sự vật (OOD) đều tập trung vào các sự vật thế giới thực và biểu diễn của chúng trong hệ thống dựa trên phần mềm nên có những điểm tương đồng quan trọng giữa hai phương pháp thiết kế. Thiết kế hướng cấu trúc dữ liệu và OOD cả hai đều hướng thông tin; cả hai đều dùng một biểu diễn dữ liệu làm cơ sở cho việc phát triển mô biểu diễn chương trình; cả hai có khái niệm riêng của chúng (được suy diễn độc lập) về thiết kế tốt. Cấp bậc dữ liệu (được dùng trong các phương pháp hướng cấu trúc dữ liệu) là tương tự với cấp bậc lớp được dùng trong OOD. Cả hai đều áp dụng các trừu tượng dữ liệu và mỗi phương pháp đều coi các thao tác biến đổi dữ liệu là thứ yếu so với chính khoản mục dữ liệu. Sự khác biệt chủ yếu giữa OOD và phương pháp thiết kế hướng cấu trúc dữ liệu ở trong định nghĩa về sự vật. Trong OOD, một sự vật bao bọc cả dữ liệu và tiến trình. Các phương pháp thiết kế hướng cấu trúc dữ liệu chọn con đường qui ước nhiều hơn một sự vật (sự vật dữ liệu) chỉ là dữ liệu. Mặc dầu không có biểu diễn trực tiếp cho kế thừa, truyền thông báo, hay bao bọc trong phương pháp thiết



kế hướng cấu trúc dữ liệu, các khái niệm này vẫn cứ tự biểu lộ tinh vi trong trực cảm thiết kế được mô tả trong chương trình

*Thiết kế hướng dữ liệu* (Data driven design) là kết quả của phương pháp thiết kế kiểu dữ liệu trừu tượng ứng với đối tượng lập trình. Sự thích nghi là đơn giản bởi vì các lớp khá giống nhau.

Từ một quan điểm hoàn toàn thực tế trên, đối tượng đóng gói hành vi (thực hiện trách nhiệm của một đối tượng) và cấu trúc (các đối tượng khác nhận biết trực tiếp đối tượng đó). Điều này cũng tương tự như định nghĩa của một kiểu dữ liệu trừu tượng.

Trước khi mô tả thiết kế hướng dữ liệu, ta hãy xem xét việc thiết kế kiểu dữ liệu trừu tượng.

Một kiểu dữ liệu trừu tượng là đóng gói dữ liệu và các thuật toán hoạt động trên dữ liệu đó. Các kiểu dữ liệu trừu tượng được thiết kế bằng cách hỏi các câu hỏi:

- Kiểu này gồm loại dữ liệu gì ?
- Các thuật toán nào có thể được áp dụng cho dữ liệu này?

Trọng tâm chính của những câu hỏi này là để xác định những dữ liệu đang được trình diễn trong hệ thống. Điều này có thể được thực hiện ban đầu bằng cách xác định các dữ liệu cần thiết của chương trình (có lẽ chỉ một phần của nó). Những thông tin này sau đó có thể được nhóm lại thành các loại sử dụng sự gắn kết như một hướng dẫn. (Gắn kết, chẳng hạn như cho một nhóm các dữ liệu, là một độ đo về sự liên quan chặt đến mức nào giữa các bộ phận của nhóm). Cuối cùng, việc xác định các thuật toán kết hợp với những loại dữ liệu thường dẫn đến việc phát hiện các loại yêu cầu khác.

Trong thiết kế hướng dữ liệu, các đối tượng được thiết kế bằng cách hỏi các câu hỏi:

- 1- đối tượng này đại diện cho cấu trúc nào?
- 2- hoạt động nào có thể được thực hiện bởi đối tượng này?

Một lần nữa, tiêu điểm chính lại nhằm vào cấu trúc dữ liệu được đại diện trong hệ thống.

Lợi ích chính của cách tiếp cận hướng dữ liệu là một quá trình quen thuộc cho các lập trình viên có kinh nghiệm với ngôn ngữ thủ tục truyền thống. Nó là tương đối dễ dàng cho các lập trình như vậy để thích nghi với kinh nghiệm trước đây của mình để thiết kế hệ thống hướng đối tượng.

### **1.3.3. Thiết kế phần mềm hướng Trách nhiệm**

Các phương pháp của một nhà thiết kế có ảnh hưởng sâu sắc đến mức độ mà đóng gói được thể hiện trong một thiết kế. Khi mô tả các phương pháp tiếp cận hướng dữ liệu để thiết kế và lý do tại sao nó không thành công để tối đa hóa đóng gói. Người ta đã dùng phương pháp thiết kế thay thế, gọi là trách nhiệm điều khiển để thiết kế với mục đích đạt được mức độ cao hơn về đóng gói.

Mục đích của thiết kế hướng trách nhiệm là để nâng cao đóng gói. Mô hình client/server cũng có mục đích như vậy nhưng ở mức thấp.

#### Mô hình Client-Server

Mô hình client /server là một mô tả sự tương tác giữa hai thực thể: các máy khách và máy chủ. Một khách hàng đưa ra yêu cầu của máy chủ để thực hiện dịch vụ. Một Server cung cấp một bộ các dịch vụ theo yêu cầu. Những cách thức trong đó khách hàng có thể tương tác với máy chủ được mô tả bằng một hợp đồng: một danh sách các yêu cầu mà có thể được thực hiện của máy chủ của khách hàng. Cả hai đều phải thực hiện hợp đồng: khách hàng bằng cách làm cho chỉ có những yêu cầu đó quy định cụ thể. và các máy chủ bằng cách đáp ứng những yêu cầu đó.

Trong lập trình hướng đối tượng, cả hai máy khách và máy chủ là một trong hai lớp học hoặc các trường hợp của các lớp. Bất kỳ đối tượng có thể hoạt động như hoặc là một khách hàng hoặc một máy chủ tại bất kỳ thời điểm nào.

Ưu điểm của mô hình client/server là nó tập trung vào những gì các máy chủ không cho các khách hàng, chứ không phải là làm thế nào các máy chủ hiện nó. Việc thực hiện các máy chủ được đóng gói - cách khóa không cho phép khách hàng xâm nhập.

Một lớp có thể có ba loại khác nhau của khách hàng:

- 1- khách hàng bên ngoài
- 2- khách hàng lớp con

### 3- chính lớp đó.

Mỗi một loại khách hàng được mô tả:

#### *1-Khách hàng bên ngoài*

Một khách hàng bên ngoài của một lớp là một đối tượng gửi tin nhắn đến một thể hiện của lớp hoặc các lớp học chính nó. Người nhận được xem như là một máy chủ, và người gửi sẽ được xem như là một khách hàng. Làm sao thông điệp được trả lời không phải là quan trọng cho người gửi. Tập hợp các thông điệp mà một đối tượng đáp ứng, bao gồm cả các loại tham số và kiểu trả về, định nghĩa hợp đồng giữa khách hàng và máy chủ.

#### *2-Khách hàng lớp con*

Một khách hàng lớp con của một lớp học bất kỳ lớp kế thừa từ các lớp. Các lớp cha được xem như là một hệ thống thoát nước; các phân lớp được xem như là một khách hàng. Các lớp con không nên quan tâm đến việc thực hiện bất kỳ hành vi mà nó thừa kế. Tập hợp các tin nhắn được thừa kế bởi các phân lớp định nghĩa hợp đồng giữa khách hàng và máy chủ.

Trong hầu hết các ngôn ngữ hướng đối tượng, các lớp con kế thừa không chỉ là hành vi mà còn cấu trúc được định nghĩa bởi cha của chúng. Thật là không may, vì nó có xu hướng khuyến khích các lập trình vi phạm đóng gói để tăng số lượng mã tái sử dụng.

#### *3-Bản thân khách hàng*

Đối với mục đích tối đa hóa đóng gói, các lớp cũng nên được xem như là khách hàng của mình. Số lượng mã dựa trực tiếp vào cấu trúc của lớp nên được giảm thiểu. Nói cách khác, cấu trúc của lớp nên được gói gọn trong số minimum của phương pháp. Đây là sự tối thiểu hóa sự lệ thuộc của một đối tượng trên cấu trúc riêng của mình, cho phép thay đổi cấu trúc đó được thực hiện càng minh bạch càng tốt.

#### Định nghĩa thiết kế hướng Trách nhiệm

Thiết kế hướng trách nhiệm là được thể hiện theo tinh thần từ mô hình client/server. Nó tập trung vào các hợp đồng bằng cách hỏi:

- Những hành động nào bắt đối tượng này chịu trách nhiệm?

- Những thông tin nào bất đối tượng này phải chia sẻ ?

Một điểm quan trọng là thông tin được chia sẻ bởi một đối tượng có thể hoặc không thể là một phần của cấu trúc của đối tượng đó. Điều này có nghĩa là các đối tượng có thể tính toán các thông tin, hoặc nó có thể ủy thác yêu cầu thông tin cho đối tượng khác.

Bản chất của thiết kế hướng đối tượng nằm trong cam kết ràng buộc vào cuối của chi tiết thực hiện các yêu cầu, và tập trung vào lúc đầu và giữa về các hành vi cần thiết để nhận ra những khả năng để đáp ứng yêu cầu quy định (Wirfs-Brock & Wilkerson, 1989). Một trong những nguyên lý của phương pháp hướng đối tượng là khả năng của mình để giúp quản lý sự phức tạp của sự phát triển hệ thống lớn. Giải thích về mục đích, cấu trúc và hành vi của các hệ thống thông minh là tương ứng phức tạp và đòi hỏi các phương pháp và các công cụ tương tự để giúp phơi bày những lý do cơ bản như thế nào và tại sao họ làm việc theo cách mà họ làm.

Theo Wirfs-Brock và Wilkerson (1989) trách nhiệm của các đối tượng bao gồm cả những hành động mà họ thực hiện hoặc các thông tin mà họ cung cấp. Một điểm quan trọng là một trách nhiệm không nhất thiết phải thực hiện hoặc thực hiện hoàn toàn trong các đối tượng được gắn thẻ với trách nhiệm. Các đối tượng có thể thực hiện trách nhiệm của mình bằng cách ủy quyền toàn bộ hoặc một phần của việc thực hiện cho các đối tượng khác, cho các hệ thống, cho các nguồn dữ liệu, hoặc ngay cả cho con người. Điều này cho thấy một cách tiếp cận để giải thích dựa trên phân chia chức năng và nhiệm vụ của các hành vi đến các yếu tố mô hình nguyên tử do đó trách nhiệm giải trình phân tán có thể bị xử lý tại địa phương đến mức tối đa có thể.

#### Điểm mạnh của thiết kế hướng Trách nhiệm

Đóng gói được thỏa hiệp khi các chi tiết cấu trúc của một đối tượng trở thành một phần của giao diện cho đối tượng đó. Điều này chỉ có thể xảy ra nếu các nhà thiết kế sử dụng kiến thức của những chi tiết về cấu trúc, thiết kế Trách nhiệm hướng tối đa hóa đóng gói khi các nhà thiết kế cố tình bỏ qua thông tin này.

Tính đa hình làm tăng khả năng đóng gói, bởi vì các khách hàng của một đối tượng không cần phải biết cách các đối tượng thực hiện các dịch vụ được yêu cầu,

mà cũng không cần biết lớp nào đã đáp ứng yêu cầu đó. Phương pháp tiếp cận hướng trách nhiệm có thể giúp một nhà thiết kế xác định các giao thức chuẩn (tên thông điệp) bằng cách khuyến khích các nhà thiết kế tập trung vào trách nhiệm thực hiện một cách độc lập. Điều này đã làm tăng tính đa hình.

Việc thiết kế các lớp mà không cần biết cấu trúc của chúng đã khuyến khích việc thiết kế hệ thống phân cấp thừa kế, vì chỉ biết kiểu của lớp đó. Nếu phân cấp thừa kế kéo theo một kiểu phân cấp, thì kiểu của các lớp là một kiểu con của lớp cha của lớp đó. Điều này có hai lợi thế:

1- Nó cải thiện đóng gói đối với khách hàng ở lớp con, đảm bảo rằng tất cả các hành vi thừa kế là một phần của hợp đồng của lớp con.

2-Nó làm cho các lớp trừu tượng dễ nhận diện. Một khó khăn trong việc xác định các lớp trừu tượng là xác định những phần nào của giao thức trong lớp hiện tại là những phần của kiểu trong các lớp đó, và những phần nào là những chi tiết thực hiện. Bởi vì các giao thức của một lớp chỉ bao gồm những thông điệp nào hình thành nên kiểu của lớp đó, do vậy sẽ loại trừ được khó khăn này.

#### **1.3.4. Thiết kế phần mềm hướng kiểm thử**

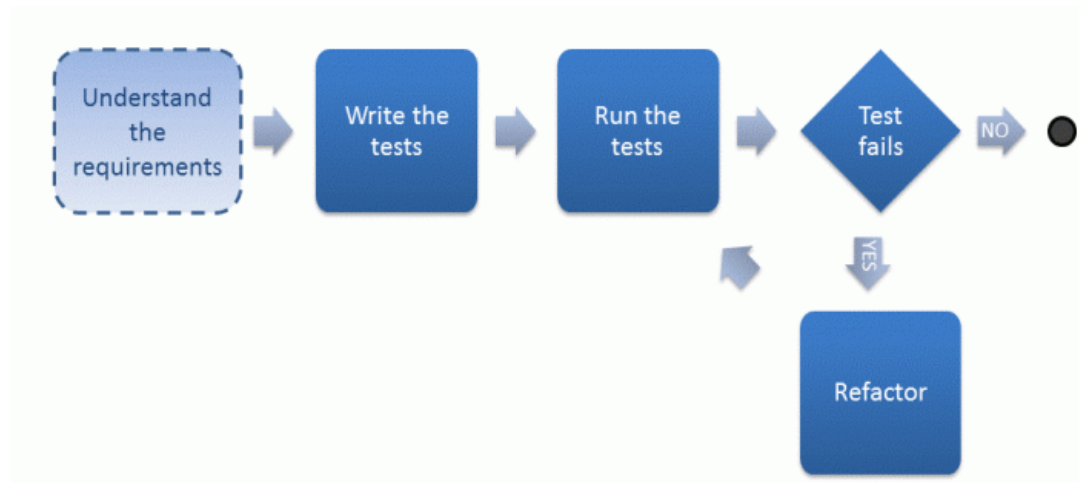
“*Test-Driven Development*” có thể hiểu là mô hình phát triển với trọng tâm hướng về việc kiểm thử. TDD được xây dựng theo hai tiêu chí: Test-First (kiểm thử trước) và Refactoring (điều chỉnh mã nguồn). Quy trình phát triển TDD khi một yêu cầu phần mềm (requirement) được đặt ra như sau:

Người phát triển soạn thảo kịch bản kiểm thử (test case) cho yêu cầu đó trước tiên và chạy thử kịch bản đó lần đầu tiên. Hiển nhiên, việc chạy thử sẽ đưa ra 1 kết quả thất bại vì hiện tại chức năng đó chưa được xây dựng (và thông qua kết quả đó, ta cũng kiểm tra được là kịch bản kiểm thử đó được viết đúng).

Theo đó, dựa vào mong muốn (expectation) của kịch bản kia, người phát triển sẽ xây dựng một lượng mã nguồn (source code) vừa đủ để lần chạy thử 2 của kịch bản đó thành công.

Nếu trong lần chạy thử 2 vẫn đưa ra kết quả thất bại, điều đó có nghĩa là thiết kế chưa ổn và người phát triển lại chỉnh sửa mã nguồn và chạy lại kịch bản đến khi thành công.

Khi kịch bản kiểm thử được chạy thành công, người phát triển tiến hành chuẩn hóa đoạn mã nguồn (base-line code) và tiếp tục hồi quy với kịch bản kiểm thử tiếp theo. Việc chuẩn hóa bao gồm thêm các chú giải, loại bỏ các dư thừa, tối ưu các biến,...



Hình 3- 1 Quy trình phát triển TDD để cập vấn đề khó khăn trong việc hiểu rõ yêu cầu chức năng trước khi viết kịch bản kiểm thử

Thông qua quy trình TDD như trên, có thể dễ dàng nhận ra là thứ tự các bước xây dựng 1 tính năng phần mềm gần như đã được đảo ngược so với 1 quy trình truyền thống. Vậy điều này giúp ích gì và có gì hay hơn?

Đối với mô hình thác nước (Waterfall Model): việc phân tích các yêu cầu (requirements) thường được tiến hành bởi nhà phát triển một cách chuyên hóa và khi đến giai đoạn xây dựng (implementing phase) thì đa phần các nhà phát triển tiếp xúc với các yêu cầu phần mềm dưới dạng các bản thiết kế. Họ chỉ quan tâm đến đầu vào, đầu ra (Input, Output) của tính năng mình xây dựng mà thiếu đi cái nhìn thực tiễn từ góc nhìn người dùng (end-users). Một hệ quả tất yếu là lỗi phần mềm đến từ việc sản phẩm không tiện dụng với người dùng.

Cùng với Agile, việc ứng dụng TDD góp phần làm gần khoảng cách giữa đội ngũ thiết kế phần mềm và sản phẩm thực tiễn, tối ưu quy trình. Cụ thể như sau: Thông qua kịch bản kiểm thử, nhà phát triển có cái nhìn trực quan về sản phẩm ngay trước khi xây dựng mã nguồn. Sản phẩm họ tạo ra chính xác và gần gũi người dùng hơn.

Phần mã nguồn được thêm vào chỉ vừa đủ để chạy thành công kịch bản kiểm thử, hạn chế dư thừa và qua đó hạn chế khả năng xảy ra lỗi trên những phần dư thừa.

Bảo đảm mã nguồn luôn phản ánh đúng và vừa đủ yêu cầu phần mềm, hạn chế được công sức tối ưu mã nguồn về sau.

Trong quá trình hình thành, TDD có liên quan mật thiết đến khái niệm “*Test-First Programming*” trong mô hình eXtreme Programming “XP” thuần túy Agile – thịnh hành từ năm 1999. Tuy nhiên, bằng việc ứng dụng đa dạng và linh hoạt, TDD cũng có những đặc điểm và tùy biến của riêng nó.

Ở đây, phương thức phát triển phần mềm linh hoạt (Agile Software Development), gọi tắt là “Agile”, đã trở nên phổ biến trong ngành phát triển phần mềm là một triết lý cùng với nhóm các phương pháp và phương pháp luận phát triển phần mềm dựa trên các nguyên tắc phát triển phân đoạn lặp (iterative) và tăng trưởng (incremental) trong đó các yêu cầu và giải pháp tiến hóa thông qua sự liên kết cộng tác giữa các nhóm tự quản và liên chức năng. Agile là cách thức làm phần mềm linh hoạt để làm sao đưa sản phẩm đến tay người dùng càng nhanh càng tốt càng sớm càng tốt và được xem như là sự cải tiến và phổ biến vượt trội so với những mô hình cũ như mô hình “Thác nước (waterfall)” hay “CMMI” (theo khảo sát của hãng nghiên cứu thị trường Forrester).

Agile thường sử dụng cách lập kế hoạch thích ứng (adaptive planning), việc phát triển và chuyển giao theo hướng tiến hóa, sử dụng các khung thời gian ngắn và linh hoạt để dễ dàng phản hồi lại với các thay đổi trong quá trình phát triển.

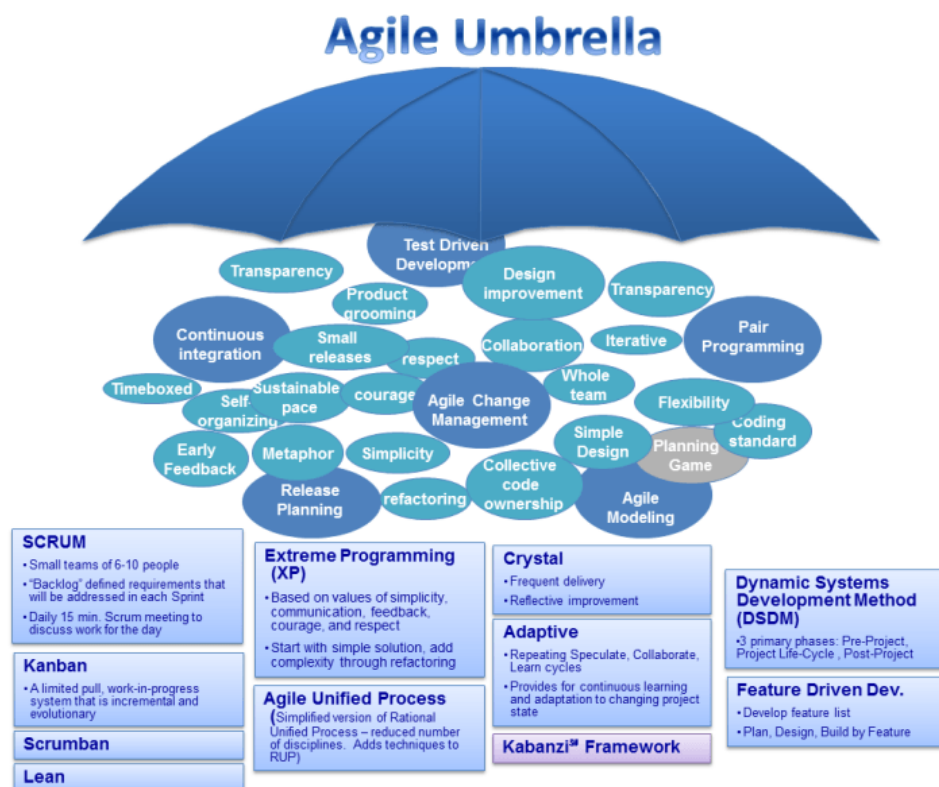
Với những phương thức tổ chức và triển khai mới lạ, năng động và linh hoạt, Agile đã thu hút sự quan tâm lớn và trở thành sự lựa chọn hàng đầu của cộng đồng làm phần mềm. Triết lý Agile đã vượt xa khỏi khu vực truyền thống của mình là phát triển phần mềm để đóng góp sự thay đổi trong cách thức làm việc, quản lý, sản xuất ở các ngành khác như sản xuất, dịch vụ bán hàng, quảng cáo, giáo dục,...

Thuật ngữ “Agile” chính thức được sử dụng rộng rãi theo cách thống nhất kể từ khi “Tuyên ngôn Agile” (Agile manifesto) được giới thiệu ra công chúng năm 2001. Tuyên ngôn của Agile được xem là cốt lõi là ngôi sao dẫn đường trong Agile.

Nó giúp các nhà phát triển có được gợi ý trong thực hành và vận dụng các phương pháp Agile trong thực tiễn. Theo tuyên ngôn, Agile hoạt động dựa trên 4 tôn chỉ:

- 1-Cá nhân và sự tương tác hơn là quy trình và công cụ;
- 2-Phần mềm chạy tốt hơn là tài liệu đầy đủ;
- 3-Cộng tác với khách hàng hơn là đàm phán hợp đồng;
- 4-Phản hồi với các thay đổi hơn là bám sát kế hoạch.

TDD đáp ứng “Tuyên ngôn về Agile” khi bản thân quy trình TDD thúc đẩy tính thực tiễn của sản phẩm, tương tác với người dùng. Để phát huy tối đa những lợi ích mà TDD mang lại, độ lớn của 1 đơn vị tính năng phần mềm (unit of function) cần đủ nhỏ để kịch bản kiểm thử dễ dàng được xây dựng và đọc hiểu, công sức debug kịch bản kiểm thử khi chạy thất bại cũng giảm thiểu hơn.



Hình 3- 2 TDD trong Agile framework phác họa bởi Mohammad Sami

### 1.3.5. Thiết kế phần mềm hướng hành vi

Trong mô hình TDD nhiệm vụ kiểm thử do nhà phát triển đảm nhiệm và vai trò chuyên hóa của người kiểm thử gần như không còn nữa. Để làm tốt công việc, xuyên suốt chu trình, người phát triển phải chú ý thêm những vấn đề thuần túy của



kiểm thử (test) như: Cái gì cần test và cái gì không?, Viết bao nhiêu kịch bản là đủ?, Làm sao để hiểu là test đó thất bại?, Bắt đầu test từ đâu?,...

Để giải quyết vấn đề phát sinh mà vẫn tận dụng triệt để lợi ích mà TDD mang lại, Dan North phát triển một mô hình mới với tên gọi: Behavior-Driven Development – BDD (hoặc ta có thể hiểu là Acceptance Test-Driven Development – ATDD). Trong đó, một vai trò mới trong việc thiết kế kiểm thử (Test Design) được đặt ra:

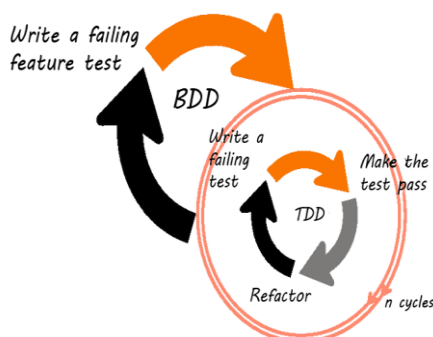
Thay vì chờ đợi sản phẩm hoàn thành và kiểm thử, người tester/analyst tham gia vào quá trình xây dựng mã nguồn với vai trò phân tích và xây dựng hệ thống kịch bản kiểm thử dưới góc độ ngôn ngữ tự nhiên dễ hiểu từ các yêu cầu (requirement).

Đồng thời, họ giúp đỡ người phát triển trong việc giải thích và đưa ra các phương án xây dựng mã nguồn mang tính thực tiễn với người dùng ngay trước khi bắt tay xây dựng.

Người phát triển liên hệ mật thiết với người tester và xây dựng mã nguồn với những phương án mà người kiểm thử cung cấp theo mô hình TDD.

Kịch bản kiểm thử được phân chia làm 2 lớp: Lớp chấp nhận (feature/acceptance test) và Lớp đơn vị (unit test).

Theo đó, kịch bản kiểm thử lớp đơn vị mang thuần tính thiết kế và phục vụ cho việc kiểm thử đơn vị (Unit test) còn kịch bản kiểm thử lớp chấp nhận có thể được tái sử dụng cho quá trình kiểm thử hồi quy (Regression Test) về sau



Hình 3- 3 Mô hình BDD – TDD trong Agile mô phỏng bởi Paul Littlebury

Từ mô hình trên ta dễ dàng nhìn nhận được sự ưu việt BDD mang lại đặc biệt là trong các dự án phần mềm lớn và phức tạp, khi cả hai khía cạnh phân hóa vai

trò và chất lượng phải đi đôi. Ngoài ra, việc chạy kịch bản kiểm thử và xử lý sớm các vấn đề thiết kế ngay trong khâu xây dựng giúp giảm thiểu tối đa chi phí và công sức sửa chữa lỗi.

Trong khi khái niệm BDD mang tính lý thuyết, việc ứng dụng của nó lại đặt nặng sự thực nghiệm. Để phát huy lợi ích về thời gian trong việc xây dựng kịch bản kiểm thử, ngôn ngữ và cách truyền tải là một thử thách khi phải đáp ứng khả năng đọc hiểu từ cả hai khía cạnh: tự nhiên và thiết kế. Bằng sự vay mượn từ ngôn ngữ viết User Story, ngôn ngữ Gherkin được phát triển để phục vụ nhu cầu đó với cấu trúc đơn giản, hướng đối tượng và tương đồng cho mọi kịch bản: Given – When – Then.

**Ví dụ:**

Given: Là một sinh viên,

When: Tôi đăng ký một khóa học thành công

Then: Tôi phải thấy tên khóa học hiện ra trong danh sách môn học của tôi.

Nói gọn lại, BDD (Behaviour Driven Development) là một quá trình phát triển phần mềm dựa trên phương pháp Agile (phát triển phần mềm linh hoạt); nó là sự mở rộng của TDD (Test driven development). Thay vì tập trung vào phát triển phần mềm theo hướng kiểm thử, BDD tập trung vào phát triển phần mềm theo hướng hành vi.

BDD thêm vào những phương pháp sau:

-Áp dụng kỹ thuật “5 Why” vào mỗi vấn đề của người sử dụng để biết được giá trị kinh doanh của mỗi người sử dụng.

-“Tư duy từ ngoài vào”, tức là chỉ cài đặt những hành vi mang lại giá trị kinh doanh để giảm thiểu lãng phí.

-Mô tả hành vi theo một loại ngôn ngữ mà cả chuyên gia nghiệp vụ, kiểm thử viên và nhà phát triển có thể giao tiếp được với nhau.

Dựa vào yêu cầu các kịch bản kiểm thử (Scenarios) sẽ được viết trước dưới dạng ngôn ngữ tự nhiên và dễ hiểu nhất sau đó mới thực hiện cài đặt mã nguồn. Những kịch bản kiểm thử này được viết dưới dạng các tệp đặc trưng (feature file) và đòi hỏi sự cộng tác từ tất cả các thành viên tham gia dự án hay các bên liên quan (stakeholder).

BDD được viết dưới dạng plain text language gọi là Gherkin. Những lợi ích khi sử dụng BDD bao gồm:

– Giúp xác định đúng yêu cầu của khách hàng: tài liệu được viết dưới dạng ngôn ngữ tự nhiên, bất kỳ đối tượng nào cũng có thể hiểu được. Khi đọc tài liệu này, khách hàng có thể dễ dàng nhận biết được lập trình viên có hiểu đúng yêu cầu của họ không và có phản hồi kịp thời.

– Là tài liệu sống của dự án: tài liệu này luôn được cập nhật khi có bất kỳ sự thay đổi nào nên tất cả các thành viên sẽ không bị miss thông tin khi phát triển hệ thống

– Nâng cao chất lượng phần mềm, tạo ra sản phẩm hữu ích: vì phát triển phần mềm theo hướng hành vi nên có thể focus vào việc tạo ra sản phẩm đúng với yêu cầu của khách hàng nhưng vẫn hữu ích cho người dùng.

Như vậy, *Behaviour Driven Development* là một quá trình phát triển phần mềm dựa trên thử nghiệm hướng phát triển.

BDD quy định rằng các người phát triển và người sở hữu sản phẩm cần hợp tác và xác định hành vi của người sử dụng. TDD với việc cộng gộp vai trò cả của kiểm thử chấp nhận (acceptance test) vào người phát triển dẫn tới phát sinh vấn đề quá tải cho người phát triển. Do đó, BDD sinh ra, hướng tới các kiểm thử đặc trưng (feature test) mà người thực hiện là các người kiểm thử chấp nhận (Acceptance Tester). Thay vì chờ đợi sản phẩm hoàn thành và kiểm thử, người kiểm thử hay người phân tích tham gia vào quá trình xây dựng mã nguồn với vai trò phân tích và xây dựng hệ thống kịch bản kiểm thử dưới góc độ ngôn ngữ tự nhiên để hiểu từ các yêu cầu (requirement).

Đồng thời, họ giúp đỡ người phát triển trong việc giải thích và đưa ra các phương án xây dựng mã nguồn mang tính thực tiễn với người dùng ngay trước khi bắt tay xây dựng. Người phát triển liên hệ mật thiết với người kiểm thử và xây dựng mã nguồn với những phương án mà người kiểm thử cung cấp theo mô hình TDD.

Kịch bản kiểm thử được phân chia làm 2 lớp:

1-Kiểm thử chấp nhận (*feature/acceptance test*)

2- Kiểm thử đơn vị (unit test) .

Theo đó, kịch bản kiểm thử đơn vị mang thuần tính thiết kế và phục vụ cho việc kiểm thử đơn vị (Unit test) còn kịch bản kiểm thử lớp chấp nhận có thể được tái sử dụng cho quá trình kiểm thử hồi quy về sau (Regression Test)

Một ứng dụng phổ biến của BDD là cố gắng kế thừa TDD bằng cách tập trung chủ yếu vào việc tạo ra nhiều phép thử của các kiểm thử chấp nhận hoặc chạy thử các đặc tả kỹ thuật. Mỗi đặc tả kỹ thuật tương ứng với đầu vào chu kỳ phát triển và mô tả, từ quan điểm của người dùng từng bước từng bước một, làm thế nào hệ thống hoạt động chính xác. Với một lần viết kịch bản nhưng các người phát triển có thể sử dụng đặc điểm kỹ thuật và quá trình TDD sẵn có của họ để triển khai phát triển mã trình đáp ứng đúng kịch bản.

### **1.3.5. Thiết kế phần mềm hướng lĩnh vực**

Thiết kế hướng lĩnh vực (*Domain Driven Design- DDD*) được giới thiệu lần đầu tiên bởi Eric Evans, trong cuốn sách của ông với tựa đề: *Domain Driven Design Tackling Complexity in the Heart of Software*. DDD là cách tiếp cận đến phát triển phần mềm cho phép các đội quản lý cấu trúc và bảo trì phần mềm trong những lĩnh vực có độ phức tạp lớn. Đây là nội dung quan trọng mà luận văn hướng tới đi sâu nghiên cứu và ứng dụng. Chiến lược thiết kế phần mềm hướng lĩnh vực sẽ được trình bày chi tiết ngay trong chương sau.

## **KẾT LUẬN CHƯƠNG**

Một tiến trình phát triển phần mềm là một tập của các hoạt động cần thiết (đặt tả, xây dựng, đánh giá, tiến hóa) để chuyển các yêu cầu của người dùng thành một hệ thống phần mềm đáp ứng được các yêu cầu đặt ra.

Có năm bước chính cần thực hiện trong quá trình phát triển phần mềm:

- Xác định các yêu cầu
- Phân tích hệ thống
- Thiết kế hệ thống
- Lập trình và kiểm tra hệ thống
- Vận hành và bảo trì hệ thống.

Do các hệ phần mềm lớn là phức tạp nên người ta thường dùng các cách tiếp cận khác nhau trong việc thiết kế các phần khác nhau của một hệ thống. Chẳng có một cách tiếp cận nào tốt nhất chung cho các dự án. Hai cách tiếp cận thiết kế hiện đang được dùng rộng rãi trong việc phát triển phần mềm đó là cách tiếp cận theo hướng chức năng và hướng đối tượng. Mỗi cách tiếp cận đều có ưu và nhược điểm riêng phụ thuộc vào ứng dụng phát triển và nhóm phát triển phần mềm. Chúng bổ sung và hỗ trợ cho nhau chứ không đối kháng nhau.

Có một số chiến lược hiện đại để thiết kế phần mềm như thiết kế phần mềm theo hướng mô hình, hướng dữ liệu, hướng trách nhiệm, hướng kiểm thử, hướng hành vi, hướng lĩnh vực. Mỗi hướng có những đặc thù riêng và có những lợi thế riêng, trong đó hướng lĩnh vực là trọng tâm nghiên cứu của luận văn.

## Chương 2

### Chiến lược thiết kế phần mềm hướng lĩnh vực

#### 2.1. Cách tiếp cận hướng lĩnh vực trong tiến trình phát triển phần mềm

##### 2.1.1. Khái niệm về thiết kế hướng lĩnh vực

Với cách tiếp cận truyền thống khi xây dựng một ứng dụng. Đầu tiên ta đọc các yêu cầu đặc tả và tìm hiểu các chức năng, sau đó tiến hành chia nhỏ các công việc. Trong phần lớn trường hợp, việc này nhằm mục đích ước lượng thời gian và lên kế hoạch thực hiện cho các công việc. Vậy trình tự công việc sẽ là ước lượng thời gian, chia việc cho các thành viên trong team, thiết kế CSDL, cuối cùng là bắt tay và code. Đây là cách thiết kế hướng dữ liệu hay còn gọi là Data Driven Design [1].

Vậy vấn đề với cách tiếp cận là gì? Lâu nay ta vẫn tiếp cận theo cách này và vẫn làm tốt ? Câu trả lời là Đúng và Sai. Đúng ở chỗ ta vẫn làm tốt trong việc bàn giao Project, Sai ở chỗ ta chưa thực hiện tốt trong việc bảo trì và mở rộng project.

Trong các ứng dụng điển hình có rất nhiều phần code xử lý các công việc không liên quan đến Logic nghiệp vụ như truy cập file, mạng hay database, các thành phần này thường được gọi là plumping code (*điền code*) và được nhúng trực tiếp vào trong Business Object và nhiều Business Logic cũng được nhúng vào thao tác của giao diện Widget hay script của database, điều này thường xảy ra vì nó làm ta phát triển ứng dụng một cách nhanh chóng và dễ dàng. Việc này dẫn đến phần lớn thời gian phát triển của Developer là dành cho việc viết các plumping code thay vì viết Business Logic thực sự, nó làm cho thiết kế của ta bị mất đi tính hướng đối tượng trong thực tế.

Ngoài ra khi logic nghiệp vụ trộn lẫn với lớp khác khiến cho việc đọc hiểu và suy nghĩ về logic của ứng dụng trở nên khó khăn hơn đối với những người ngoài. Chỉ một thay đổi nhỏ ở tầng UI (*User interface*) cũng có thể dẫn tới việc thay đổi tầng logic và ngược lại khi thay đổi một business rule của ứng dụng đòi hỏi ta phải

quan tâm đến từng chi tiết nhỏ phía UI cũng như database để đáp ứng được sự thay đổi này.

Trong những ứng dụng nhỏ thì vấn đề này ta không nhìn thấy. Ở các ứng dụng cỡ vừa thì thấy vấn đề này đã tồn tại và bắt đầu dẫn đến tình trạng phá vỡ các thiết kế chuẩn. Đối với các ứng dụng lớn thì nó trở thành vấn đề nghiêm trọng, cách tiếp cận trên sẽ không thể cho ta đưa ra một thiết kế hướng đối tượng chính xác. Giải pháp ở đây chính là Domain Driven Design.

Vậy DDD là gì ? DDD không liên quan gì đến công nghệ hay framework là những thứ thuộc về tầng vật lý mà nó là một khái niệm thuộc về tầng logic khi ta xây dựng một hệ thống phần mềm. Cụ thể hơn nó là mẫu thiết kế (design pattern) và hơn nữa đây là design pattern ở cấp độ kiến trúc của hệ thống, ta cần phân biệt rõ điều này để phân biệt các design pattern và ở cấp độ class. Nó cung cấp một cấu trúc thực hành và các thuật ngữ cho việc ra quyết định thiết kế nhằm tập trung và tăng tốc các dự án phần mềm trong các lĩnh vực phức tạp.

### **2.1.2. Tìm hiểu về lĩnh vực**

Để tạo một phần mềm tốt, cần hiểu về phần mềm đó. Ta không thể làm ra hệ thống phần mềm ngân hàng nếu trừ khi ta có hiểu biết tương đối tốt về mảng ngân hàng và những điều liên quan. Nghĩa là, để làm phần mềm tốt, ta cần hiểu lĩnh vực ngân hàng. Liệu có thể làm được phần mềm ngân hàng phức tạp dù không có hiểu biết nghiệp vụ tốt? Không thể. Không bao giờ. Ai hiểu về banking? Người thiết kế phần mềm? Không. Người này chỉ tới ngân hàng để gửi tiền và rút tiền khi cần. Người phân tích phần mềm? Cũng không hẳn. Anh ta chỉ biết phân tích một chủ đề cụ thể khi anh ta có đầy đủ tất cả các phần. Lập trình viên? Vậy là ai? Nhân viên ngân hàng, hiển nhiên. Hiểu nhất về hệ thống ngân hàng là những người ở trong đó, những chuyên gia của họ. Họ hiểu mọi thứ chi tiết, cái hay dở, mọi vấn đề có thể và mọi quy định. Đây là nơi ta thường xuất phát: Lĩnh vực (domain).

Giả sử cần xây dựng một hệ thống phần mềm quản lý bệnh viện. Rõ ràng là cần phải làm việc với đội ngũ bác sĩ, y tá (chính là các chuyên gia trong lĩnh vực này - domain expert) để xây dựng kiến thức về domain. Qua nói chuyện, trao đổi kiến thức, đặt câu hỏi và trả lời. Cần hiểu rõ càng nhiều càng tốt về domain này. Bằng

cách đặt câu hỏi đúng, xử lý thông tin đúng cách, kỹ sư phần mềm và chuyên gia sẽ dần vẽ ra một domain, một mô hình domain (domain model). Kỹ sư phần mềm, kết hợp với domain expert cùng tạo nên một domain model và mô hình đó là nơi kiến thức chuyên môn của cả hai bên được kết hợp và tổng hợp lại.

Hãy xem xét tiếp ví dụ sau. Giả sử ta đang tham gia thiết kế một tòa nhà.

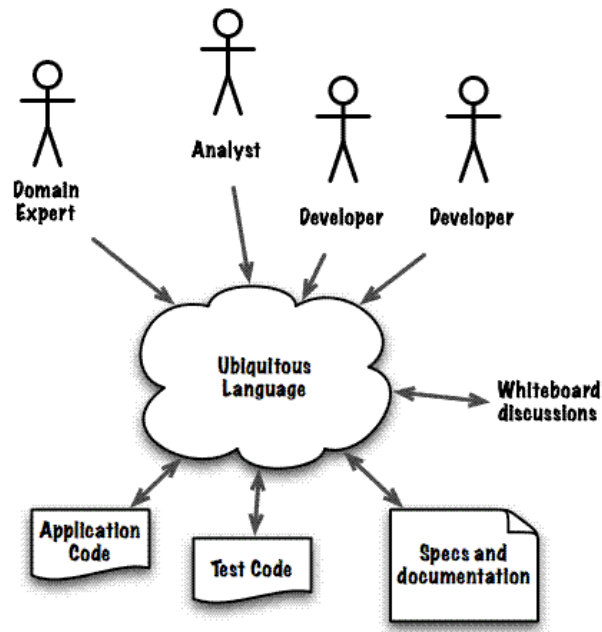
Yêu cầu là:

- + Xây dựng trên một diện tích đất cố định
- + Tòa nhà cao 6 tầng
- + Mỗi tầng có 4 căn hộ

Vậy domain ở đây là gì? Là công trình xây dựng chăng? Cũng có thể. Nhưng nếu xem công trình xây dựng là domain, thì thể ta đang bỏ qua một vài chi tiết trong yêu cầu. Công trình xây dựng đang thiết kế phải bao gồm thiết kế căn hộ cho người dân sinh sống. Vậy thuật ngữ "*công trình xây dựng*" có thể khiến ta bỏ lỡ chi tiết, thay vì đó ta có thể thu hẹp xuống thành "*chung cư*". Lúc này nếu nói với các kỹ sư về việc thiết kế, rõ ràng thuật ngữ "*chung cư*" sẽ dễ hiểu hơn là "*công trình xây dựng*" đơn thuần. Ta thấy đấy, chỉ một thay đổi nhỏ trong ngôn từ cũng có thể tạo nên sự khác biệt. Trở lại ví dụ phần mềm bệnh viện ở trên, người thiết kế phần mềm và các bác sĩ, y tá không thể nói cùng một ngôn ngữ được. Họ nói về từ ngữ chuyên môn, người thiết kế phần mềm nói bằng đối tượng, phương thức, quan hệ. Đó chính là lúc ta cần một ngôn ngữ chung để cả hai bên có thể làm việc với nhau dễ dàng hơn.



### 2.1.3. Ngôn ngữ chung



Hình 2- 1. Mô hình ngôn ngữ chung

Ta đã thấy sự cần thiết không thể phủ nhận của việc phát triển mô hình cho domain qua sự làm việc giữa chuyên gia phần mềm và chuyên gia domain; tuy nhiên, cách làm này ban đầu thường gặp những khó khăn về rào cản giao tiếp cơ bản. Lập trình viên chỉ nghĩ tới lớp, method, thuật toán, pattern và có khuynh hướng diễn tả mọi thứ đòi thường bằng những tạo tác lập trình. Họ muốn nhìn lớp đối tượng và tạo quan hệ mô hình giữa chúng. Họ nghĩ đến những thứ như kế thừa, đa hình, OOP... Và họ luôn nói theo cách đó. Điều này là dễ hiểu với lập trình viên. Lập trình viên vẫn chỉ là lập trình viên. Tuy vậy, chuyên gia domain thường không hiểu những khái niệm đó. Họ không có khái niệm gì về thư viện, framework phần mềm và trong nhiều trường hợp, thậm chí họ không hiểu về cơ sở dữ liệu. Tất cả những gì họ biết là chuyên ngành cụ thể của họ.

**Ví dụ:** Trong theo dõi không lưu, chuyên gia domain biết về máy bay, route, cao độ, vĩ độ, kinh độ, độ lệch của route chuẩn, về quỹ đạo của máy bay. Họ nói về những điều này bằng ngôn ngữ riêng của họ, đôi khi gây khó hiểu với người ngoài ngành. Để vượt qua sự khác nhau về cách giao tiếp, ta xây dựng mô hình, ta phải trao đổi, giao tiếp ý tưởng về mô hình, về những thành phần liên quan đến mô hình, cách ta liên kết chúng, chúng có liên quan với nhau hay không. Giao tiếp ở mức độ

này là tối quan trọng cho sự thành công của dự án. Nếu ai đó nói điều gì đó và người khác không hiểu, hoặc tệ hơn, hiểu sai, thì liệu ta có cơ hội tiếp tục dự án không?

Dự án sẽ gặp vấn đề nghiêm trọng nếu thành viên nhóm không chia chung ngôn ngữ khi trao đổi về domain. Chuyên gia domain dùng từ lóng của riêng họ trong khi thành viên kỹ thuật lại dùng ngôn ngữ riêng của họ để trao đổi về những thuật ngữ của domain trong thiết kế. Bộ thuật ngữ trong trao đổi hành ngày tách riêng với bộ thuật ngữ nhúng trong mã nguồn (là sản phẩm quan trọng cuối cùng của dự án phần mềm). Cùng một người có thể dùng khác ngôn ngữ khi nói hay viết do đó các thể hiện sắc sảo nhất của domain thường hiện ra và biến mất chóng vánh, không thể nhìn thấy trong mã nguồn hay trong văn bản viết.

Khi trao đổi, ta thường "*dịch*" sự hiểu về khái niệm nào đó. Lập trình viên thường diễn tả mẫu thiết kế bằng ngôn ngữ bình dân và thường là họ thất bại. Chuyên gia domain cố gắng hiểu những ý tưởng đó và thường tạo ra bộ jargon mới. Khi cuộc chiến đấu về ngôn ngữ kiểu này xảy ra, sẽ rất khó để có quy trình xây dựng kiến thức. Ta có khuynh hướng dùng "*phương ngữ*" riêng của mình trong phiên thiết kế, các "*phương ngữ*" này có thể trở thành ngôn ngữ chung vì không một ngôn ngữ nào thỏa mãn nhu cầu của tất cả mọi người.

Hiển nhiên ta cần nói chung một ngôn ngữ khi ta gặp và trao đổi về mô hình, qua nghĩa chúng. Vậy ngôn ngữ sẽ là gì? Là ngôn ngữ của lập trình viên? Là ngôn ngữ của chuyên gia domain? Hay một cái gì đó khác, ở giữa? 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. Hãy sử dụng mô hình như là xương sống của ngôn ngữ. Hãy yêu cầu nhóm sử dụng ngôn ngữ một cách nhất quán trong mọi trao đổi, bao gồm cả mã nguồn. Khi chia sẻ và làm mô hình, nhóm dùng ngôn ngữ nói, viết và giản đồ. Hãy đảm bảo rằng ngôn ngữ xuất hiện một cách nhất quán trong mọi hình thức trao đổi sử dụng trong nhóm; chính vì lý do này, ngôn ngữ này được gọi là Ngôn ngữ chung.

Ngôn ngữ chung kết nối mọi phần của thiết kế, tạo thành tiền hoạt động của nhóm thiết kế. Có thể mất hàng vài tuần hay thậm chí vài tháng để thiết kế của một dự án lớn ổn định được. Thành viên nhóm phát hiện yếu tố mới trong thiết kế cần hay cần xem xét để đưa vào thiết kế tổng thể. Những việc này không thể làm được nếu thiếu ngôn ngữ chung. Ngôn ngữ không xuất hiện một cách dễ dàng. Nó đòi hỏi nỗ lực và sự tập trung để đảm bảo rằng những thành phần chính của ngôn ngữ được chất lọc. Ta cần tìm ra những khái niệm chính, định nghĩa domain và thiết kế, tìm ra những thuật ngữ tương ứng và sử dụng chúng. Một số từ dễ nhìn ra, một số khó tìm ra hơn. Vượt qua những khó khăn bằng việc trình bày theo một cách khác mô tả mô hình khác. Sau đó *refactor* mã nguồn, đổi tên lớp, *method*, mô-đun để phù hợp với mô hình mới. Giải quyết sự nhầm lẫn về thuật ngữ trong trao đổi chính là cách ta làm để đi tới sự đồng thuận của những thuật ngữ tầm thường.

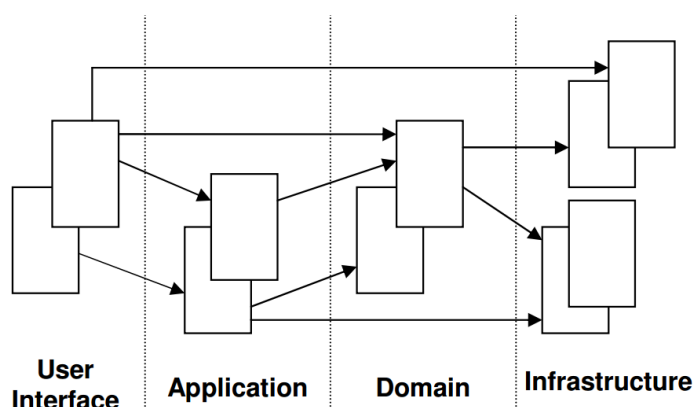
Xây dựng một ngôn ngữ theo cách đó có hiệu quả rõ ràng: Mô hình và ngôn ngữ gắn kết chặt hơn. Một thay đổi ngôn ngữ nên kéo theo sự thay đổi về mô hình. Chuyên gia domain cần phản đối những từ hoặc cấu trúc rắc rối hay không phù hợp cho việc hiểu domain. Nếu chuyên gia domain không hiểu điều gì đó trong mô hình hoặc ngôn ngữ thì chắc chắn có gì không ổn. Mặt khác, lập trình viên cần để tới sự không rõ ràng và tính không nhất quán vốn hay xảy ra trong thiết kế

## **2.2. Các đặc trưng thiết kế phần mềm hướng lĩnh vực**

Khi ta tạo ra một ứng dụng phần mềm, một lượng lớn thành phần của ứng dụng không liên quan trực tiếp đến nghiệp vụ, nhưng chúng là một phần của hạ tầng phần mềm hoặc phục vụ chính bản thân ứng dụng. Nó có khả năng và ổn cho phần nghiệp vụ của ứng dụng nhỏ so với các phần còn lại, vì một ứng dụng điển hình chứa rất nhiều đoạn mã liên quan đến truy cập CSDL, tệp hoặc mạng, giao diện người dùng,... Trong một ứng dụng hướng đối tượng thuần túy, giao diện người dùng, mã truy cập CSDL và các đoạn mã hỗ trợ khác thường được viết trực tiếp vào trong các đối tượng nghiệp vụ. Thêm vào đó, đối tượng nghiệp vụ này lại được nhúng vào trong các hành vi của giao diện người dùng và các kịch bản CSDL. Đôi khi điều này diễn ra bởi vì nó là cách dễ nhất để làm cho mọi việc trở nên nhanh chóng. Tuy nhiên, khi các đoạn code liên quan đến nghiệp vụ được trộn lẫn giữa các

tầng lại với nhau, nó trở nên vô cùng khó khăn cho việc đọc cũng như suy nghĩ về chúng. Các thay đổi ở giao diện người dùng cũng có thể thực sự thay đổi cả logic nghiệp vụ. Để thay đổi logic nghiệp vụ có thể yêu cầu tới truy vết tỉ mỉ các đoạn mã của giao diện người dùng, CSDL, hoặc các thành phần khác của chương trình. Mô hình phát triển hướng đối tượng trở nên phi thực tế. Kiểm thử tự động sẽ khó khăn. Với tất cả các công nghệ và logic liên quan trong từng hoạt động, chương trình cần được giữ rất đơn giản hoặc không thì sẽ biến chúng trở nên không thể hiểu được.

Do đó, hãy phân chia một chương trình phức tạp thành các LỚP. Phát triển một thiết kế cho mỗi LỚP để chúng trở nên gắn kết và chỉ phụ thuộc vào các tầng bên dưới. Tuân theo khuôn mẫu kiến trúc chuẩn để cung cấp liên kết lỏng lẻo tới các tầng phía trên. Tập trung các đoạn mã liên quan đến các đối tượng nghiệp vụ trong một lớp và cô lập chúng khỏi lớp giao diện người dùng, ứng dụng và infrastructure. Các đối tượng nghiệp vụ, được giải phóng khỏi việc hiển thị, lưu trữ chính chúng, hay quản lý các nhiệm vụ của ứng dụng, ... và có thể tập trung vào việc biểu hiện domain model. Điều này cho phép một model tiến hóa đủ phong phú và rõ ràng, nhằm nắm bắt kiến thức nghiệp vụ thiết yếu và áp dụng nó vào làm việc. Một giải pháp kiến trúc chung cho DDD chứa bốn lớp (trên lý thuyết):



Hình 2- 2 Kiến trúc phân lớp

- **Giao diện người dùng**(*User Interface - Presentation Layer*):Chịu trách nhiệm trình bày thông tin tới người sử dụng và thông dịch lệnh của người dùng.
- **Lớp chương trình**(*Application Layer*): Đây là một lớp mỏng phối hợp các hoạt động của ứng dụng. Nó không chứa logic nghiệp vụ. Nó không lưu giữ

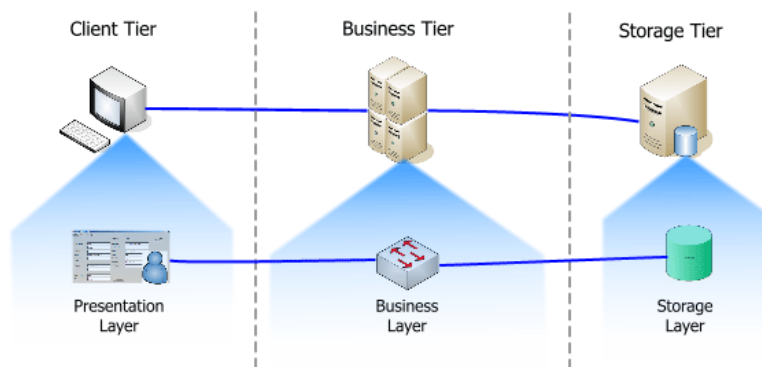
trạng thái của các đối tượng nghiệp vụ nhưng nó có thể giữ trạng thái của một tiến trình của ứng dụng.

- **Lớp Domain** (*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.
- **Lớp hạ tầng** (*Infrastructure Layer*): Lớp này đóng vai trò như một thư viện hỗ trợ cho tất cả các lớp còn lại. Nó cung cấp thông tin liên lạc giữa các lớp, cài đặt persistence cho đối tượng nghiệp vụ, đồng thời chứa các thư viện hỗ trợ cho tầng giao diện người dùng,...

Điều quan trọng là phải phân chia một ứng dụng ra nhiều lớp riêng biệt và thiết lập các quy tắc tương tác giữa các lớp với nhau. Nếu các đoạn code không được phân chia rõ ràng thành các lớp như trên, thì nó sẽ sớm trở nên vương vãi khiến vô cùng khó khăn cho việc quản lý các thay đổi. Một thay đổi nhỏ trong một phần của đoạn code có thể gây kết quả bất ngờ và không mong muốn trong các phần khác. Lớp domain nên tập trung vào những vấn đề nghiệp vụ cốt lõi. Nó không nên tham gia vào các hoạt động của infrastructure. Giao diện người dùng không nên kết nối quá chặt chẽ với các logic nghiệp vụ, cũng như không nên làm các nhiệm vụ mà thông thường thuộc về các lớp infrastructure. Một lớp application cần thiết trong nhiều trường hợp. Nó cần thiết để quản lý logic nghiệp vụ mà trong đó là giám sát và điều phối toàn bộ hoạt động của ứng dụng.

Chẳng hạn, một sự tương tác điển hình của các lớp application, domain và infrastructure có thể rất giống nhau. Người dùng muốn đặt một chặng bay và yêu cầu một service trong lớp application để làm như vậy. Tầng application sẽ tìm nạp các đối tượng nghiệp vụ có liên quan từ infrastructure và gọi các phương thức có liên quan từ chúng, ví dụ để việc kiểm tra security margin của các chuyến bay đã đặt khác. Một khi đối tượng nghiệp vụ đã vượt qua tất cả các kiểm tra và trạng thái của chúng được cập nhật sang “*đã chốt*”, dịch vụ chương trình sẽ gắn kết với đối tượng trong lớp hạ tầng.

Đến đây thì ta sẽ thấy kiến trúc của DDD tuy mới nhìn có vẻ lạ nhưng chỉ đơn giản là nó tùy biến lại mô hình kiến trúc 3 lớp (3-tier architecture) cho linh hoạt hơn. Tính linh hoạt này được tạo ra từ hệ quả của việc tái tổ chức lại các layer từ mô hình ba lớp, nó thể hiện ở data flow và control flow giữa 2 mô hình.



*Hình 2- 3 Mô hình 3 lớp*

Ta có thể thấy là trong mô hình 3 lớp thì tầng trên sẽ phụ thuộc trực tiếp vào tầng dưới nên không thể truy cập dữ liệu một cách trực tiếp từ tầng Presentation sang tầng Data Access Layer mà không thông qua tầng Business Layer. Còn mô hình DDD thì từ tầng User Interface nếu muốn lưu cái gì đó vào trong database chẳng hạn nó có thể gọi trực tiếp xuống tầng Infrastructure để làm được việc đó. Rõ ràng là trong kiến trúc DDD thì tính loose coupling được đảm bảo tốt hơn. Có thể hình dung một cách trực quan là mô hình 3 lớp giống như một ngôi nhà 3 tầng chỉ có cầu thang bộ và việc di chuyển giữa tầng một và tầng 3 cần đi qua sàn tầng 2, trong khi mô hình DDD giống như ngôi nhà 4 tầng có lắp thêm thang máy, ta có thể di chuyển đến các tầng khác nhau một cách tự do hơn.[8]

### **2.2.1 Thực thể**

Trong các đối tượng của một phần mềm, có một nhóm các đối tượng có định danh riêng, những định danh - tên gọi này của chúng được giữ nguyên xuyên suốt trạng thái hoạt động của phần mềm. Đối với những đối tượng này thì các thuộc tính của chúng có giá trị như thế nào không quan trọng bằng việc chúng tồn tại liên tục và xuyên suốt quá trình của hệ thống, thậm chí là sau cả khi đó. Chúng được gọi tên là những Thực thể - Entity. Các ngôn ngữ lập trình thường lưu giữ các đối tượng trong bộ nhớ và chúng được gắn với một tham chiếu hoặc một địa chỉ nhớ cho từng

đối tượng. Tham chiếu trong vùng nhớ này có đặc điểm là sẽ không thay đổi trong một khoảng thời gian nhất định khi chạy chương trình, tuy nhiên không có gì đảm bảo là nó sẽ luôn như vậy mãi. Thực tế là ngược lại, các đối tượng liên tục được đọc vào và ghi ra khỏi bộ nhớ, chúng có thể được đóng gói lại và gửi qua mạng và được tái tạo lại ở đâu kia, hoặc có thể chỉ đơn giản là chúng sẽ bị hủy. Và do vậy nên tham chiếu này chỉ đại diện cho một trạng thái nhất định của hệ thống và không thể là định danh mà ta đã nói ở trên được.

Lấy ví dụ một lớp chứa thông tin về thời tiết, ví dụ như nhiệt độ, khả năng có hai đối tượng có cùng một giá trị là hoàn toàn có thể xảy ra. Cả hai đối tượng này y hệ nhau và có thể sử dụng thay thế cho nhau, tuy nhiên chúng có tham chiếu khác nhau. Đây không phải là Thực thể.

Lấy một ví dụ khác, để tạo một lớp Person chứa thông tin về một người 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.

Cuối cùng ta hãy thử xem xét một hệ thống tài khoản ngân hàng. Mỗi tài khoản có một số tài khoản riêng và chúng có thể được xác định chính xác thông qua con số này. Số tài khoản được giữ nguyên trong suốt thời gian tồn tại của hệ thống, đảm bảo tính liên tục. Nó có thể được lưu như là một đối tượng trong bộ nhớ, hoặc có thể được xóa trong bộ nhớ và ghi ra cơ sở dữ liệu. Khi tài khoản bị đóng thì nó có thể được lưu trữ ra đâu đó và sẽ tiếp tục tồn tại chừng nào còn có nhu cầu sử dụng. Cho dù được lưu ở đâu thì con số này vẫn là không đổi. Do vậy, để có thể viết được một Thực thể trong phần mềm ta cần phải tạo một Định danh. Đối với một người đó có thể là một tổ hợp của các thông tin như: tên, ngày sinh, nơi sinh, tên bố mẹ, địa chỉ hiện tại v.v..., hay như ở Mỹ thì có thể chỉ cần mã số an sinh xã hội. Đối với một tài khoản ngân hàng thì số tài khoản là đủ để tạo định danh. Thông thường định danh là một hoặc một tổ hợp các thuộc tính của một đối tượng, chúng có thể

được tạo để lưu riêng cho việc định danh, hoặc thậm chí là hành vi. Điểm mấu chốt ở đây là hệ thống có thể phân biệt hai đối tượng với hai định danh khác nhau một cách dễ dàng, hay hai đối tượng chung định danh có thể coi là một. Nếu như điều kiện trên không được thỏa mãn, cả hệ thống có thể sẽ gặp lỗi.

Có nhiều cách để tạo một định danh duy nhất cho từng đối tượng. Định danh (từ nay ta gọi là ID) có thể được sinh tự động bởi một mô đun và sử dụng trong nội bộ phần mềm mà người sử dụng không cần phải biết. Đó có thể là một khóa chính trong cơ sở dữ liệu, điều này đảm bảo tính duy nhất của khóa. Mỗi khi đối tượng được lấy ra từ cơ sở dữ liệu, ID của đối tượng sẽ được đọc và tạo lại trong bộ nhớ. Trong trường hợp khác, ID của một sân bay lại được tạo bởi người dùng, mỗi sân bay sẽ có một ID chung được cả thế giới ghi nhận và được sử dụng bởi các công ty vận chuyển trên toàn thế giới trong lịch trình bay của họ. Một giải pháp khác là sử dụng luôn những thuộc tính của đối tượng để làm ID, nếu như vẫn chưa đủ thì có thể tạo thêm một thuộc tính khác cho việc đó.

Khi một đối tượng được xác định bởi định danh thay vì thuộc tính của nó, hãy làm rõ việc này trong định nghĩa model của đối tượng. Định nghĩa của một lớp nên chú trọng vào tính đơn giản, liên tục của chu kỳ tồn tại (*life cycle*) của chúng. Nên có một phương thức để phân biệt các đối tượng mà không phụ thuộc vào trạng thái hay lịch sử của chúng. Cần lưu ý kỹ các so sánh đối tượng dựa trên thuộc tính của chúng. Hãy định nghĩa một thao tác được đảm bảo sẽ có kết quả duy nhất cho từng đối tượng khác nhau (như gắn một ký hiệu đặc biệt cho chúng). Phương pháp này có thể xuất phát từ bên ngoài hoặc là một định danh tạo bởi hệ thống, miễn sao nó đảm bảo được tính duy nhất trong model. Model cần định nghĩa sao cho hai đối tượng ở hai nơi là một. Các thực thể là những đối tượng rất quan trọng của domain model và việc mô hình hóa quá trình nên lưu ý đến chúng ngay từ đầu. Việc xác định xem một đối tượng có phải là thực thể hay không cũng rất quan trọng.

### **2.2.2 Đối tượng giá trị**

Khi 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.



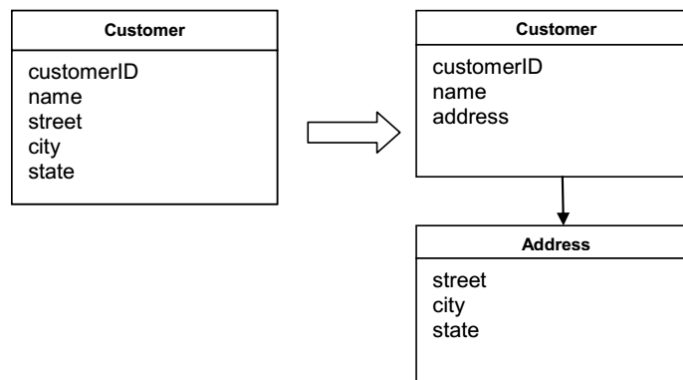
Vì những đặc điểm này nên ta cần phân biệt rõ Entity Object và Value Object. Để đảm bảo tính đồng nhất ta sẽ không gắn mọi đối tượng thành thực thể. Thay vào đó ta chỉ gắn thực thể cho những đối tượng nào phù hợp nhất với các đặc tính của thực thể. Các đối tượng còn lại sẽ là ValueObject. Điều này sẽ đơn giản hóa bản thiết kế và sẽ có các lợi ích về sau.

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. Điều này thực sự đơn giản hóa thiết kế rất nhiều.

Một điểm quan trọng là Value Object thì không khả chuyển. Chúng được tạo bởi các hàm constructor và không bao giờ được thay đổi trong vòng đời của mình. Khi bạn muốn đối tượng với giá trị khác, bạn tạo một đối tượng mới chứ không thay đổi giá trị của đối tượng cũ. Điều này ảnh hưởng rõ ràng tới thiết kế. Do không khả chuyển và không có định danh, Value Object có thể được dùng chung. Đó có thể là đòi hỏi cần thiết cho nhiều thiết kế. Các đối tượng không khả chuyển có thể chia sẻ là gợi ý tốt về hiệu năng. Chúng cũng thể hiện tính toàn vẹn, như toàn vẹn về dữ liệu. Tưởng tượng việc chia sẻ các đối tượng khả chuyển. Ví dụ một trong các thuộc tính của đối tượng đó là mã chuyến bay. Một khách hàng đặt vé cho một điểm đến. Một khách hàng khác đặt chuyến bay tương tự. Trong thời gian đó, người khách hàng thứ hai này đổi ý và chọn chuyến bay khác. Hệ thống khi đó đổi mã chuyến bay bởi vì mã chuyến bay "*khả chuyển*". Kết quả là mã chuyến bay của khách hàng đầu cũng bị đổi theo, dù không hề mong muốn.

Quy luật vàng: nếu Value Object có thể được chia sẻ, thì nó phải không khả chuyển. Value Object nên được thiết kế và duy trì đơn giản và mỏng. Khi một Value Object được truy xuất, ta có thể đơn giản là truyền giá trị, hoặc truyền một bản copy. Tạo một bản copy của một Value Object thì đơn giản và không tạo ra hệ quả gì. Và vì Value Object không có định danh, ta có thể tạo bao nhiêu bản copy cũng được và hủy các bản đó khi cần thiết. Value Object có thể chứa đựng các Value Object khác và nó thậm chí chứa đựng các tham chiếu tới các thực thể. Mặc dù Value Object được sử dụng đơn giản như chứa đựng các thuộc tính của một đối tượng Domain, nhưng không có nghĩa là nó nên chứa đựng nhiều thuộc tính. Các

thuộc tính nên được nhóm lại vào các đối tượng khác nhau. Các thuộc tính tạo nên một Value Object nên thuộc về một Value Object. 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.



Hình 2- 4 Value Object

### 2.2.2 Dịch vụ

Khi phân tích domain để xác định những đối tượng chính có trong mô hình ta sẽ gặp những thành phần không dễ để có thể gán chúng cho một đối tượng nhất định nào đó. Một đối tượng thông thường được xem là sẽ có những thuộc tính - những trạng thái nội tại - được quản lý bởi đối tượng đó, ngoài ra đối tượng còn có những hành vi. Khi thiết lập Ngôn ngữ chung, các khái niệm chính của domain sẽ được thể hiện trong Ngôn ngữ chung này, các danh từ sẽ là các đối tượng, các động từ đi kèm với danh từ đó sẽ thành các hành vi của đối tượng. Tuy nhiên có một số hành vi trong domain, những động từ trong Ngôn ngữ chung, lại có vẻ không thuộc về một đối tượng nhất định nào cả. Chúng thể hiện cho những hành vi quan trọng trong domain nên cũng không thể bỏ qua chúng hoặc gán vào các Thực thể hay Đối tượng giá trị. Việc thêm hành vi này vào một đối tượng sẽ làm mất ý nghĩa của đối tượng đó, gán cho chúng những chức năng vốn không thuộc về chúng. Dù sao đi nữa, vì ta đang sử dụng Lập trình hướng đối tượng nên ta vẫn cần phải có một đối tượng để chứa các hành vi này. Thông thường hành vi này hoạt động trên nhiều đối

tượng khác nhau, thậm chí là nhiều lớp đối tượng. 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.

Đối với những hành vi như vậy trong domain, các tốt nhất là khai báo chúng như là một Dịch vụ. Một Dịch vụ không có trạng thái nội tại và nhiệm vụ của nó đơn giản là cung cấp các chức năng cho domain. Dịch vụ có thể đóng một vai trò quan trọng trong domain, chúng có thể bao gồm các chức năng liên quan đến nhau để hỗ trợ cho các Thực thể và Đối tượng mang giá trị. Việc khai báo một Dịch vụ một cách tường minh là một dấu hiệu tốt của một thiết kế cho domain, vì điều này giúp phân biệt rõ các chức năng trong domain đó, giúp tách biệt rạch ròi khái niệm. Sẽ rất khó hiểu và nhập nhằng nếu gán các chức năng như vậy vào một Thực thể hay Đối tượng giá trị vì lúc đó ta sẽ không hiểu nhiệm vụ của các đối tượng này là gì nữa.

Dịch vụ đóng vai trò là một interface cung cấp các hành động. Chúng thường được sử dụng trong các framework lập trình, tuy nhiên chúng cũng có thể xuất hiện trong tầng domain. Khi nói về một dịch vụ người ta không quan tâm đến đối tượng thực hiện Dịch vụ đó, mà quan tâm tới những đối tượng được xử lý bởi dịch vụ. Theo cách hiểu này, Dịch vụ trở thành một điểm nối tiếp giữa nhiều đối tượng khác nhau. Đây chính là một lý do tại sao các Dịch vụ không nên tích hợp trong các đối tượng của domain. Làm như thế sẽ tạo ra các quan hệ giữa các đối tượng mang chức năng và đối tượng được xử lý, dẫn đến gán kết chặt chẽ giữa chúng. Đây là dấu hiệu của một thiết kế không tốt, mã nguồn chương trình sẽ trở nên rất khó đọc hiểu và quan trọng hơn là việc sửa đổi hành vi sẽ khó khăn hơn nhiều.

Một Dịch vụ không nên bao gồm các thao tác vốn thuộc về các đối tượng của domain. Sẽ là không nên cứ có bất kỳ thao tác nào ta cũng tạo một Dịch vụ cho chúng. Chỉ khi một thao tác đóng một vai trò quan trọng trong domain ta mới cần tạo một Dịch vụ để thực hiện. Dịch vụ có ba đặc điểm chính:

1. Các thao tác của một Dịch vụ khó có thể gán cho một Thực thể hay Đối tượng giá trị nào
2. Các thao tác này tham chiếu đến các đối tượng khác của domain

### 3. Thao tác này không mang trạng thái (stateless).

Khi một quá trình hay sự biến đổi quan trọng trong domain không thuộc về một Thực thể hay Đối tượng giá trị nào, việc thêm thao tác này vào một đối tượng/giao tiếp riêng sẽ tạo ra một Dịch vụ. Định nghĩa giao tiếp này theo Ngôn ngữ chung của mô hình, tên của thao tác cũng phải đặt theo một khái niệm trong Ngôn ngữ chung, ngoài ra cần đảm bảo Dịch vụ không chứa trạng thái.

Khi sử dụng Dịch vụ cần đảm bảo tầng domain được cách ly với các tầng khác. Sẽ rất dễ nhầm lẫn giữa các dịch vụ thuộc tầng domain và thuộc các tầng khác như infrastructure hay application nên cần cẩn thận giữ sự tách biệt của tầng domain.

Thông thường Dịch vụ của các tầng domain và application được viết dựa trên các Thực thể và Đối tượng giá trị nằm trong tầng domain, cung cấp thêm chức năng liên quan trực tiếp tới các đối tượng này. Để xác định xem một Dịch vụ thuộc về tầng nào không phải dễ dàng. Thông thường nếu Dịch vụ cung cấp các chức năng liên quan tới tầng application ví dụ như chuyển đổi đối tượng sang JSON, XML thì chúng nên nằm ở tầng này. Ngược lại nếu như chức năng của Dịch vụ liên quan tới tầng domain và chỉ riêng tầng domain, cung cấp các chức năng mà tầng domain cần thì Dịch vụ đó thuộc về tầng domain.

Hãy xem xét một ví dụ thực tế: một chương trình báo cáo trên web. Các báo cáo sử dụng các dữ liệu lưu trong CSDL và được sinh ra theo các mẫu có sẵn (template). Kết quả sau cùng là một trang HTML để hiển thị trên trình duyệt của người dùng.

Tầng UI nằm trong các trang web và cho phép người dùng có thể đăng nhập, lựa chọn các báo cáo họ cần và yêu cầu hiển thị. Tầng application là một tầng rất mỏng nằm giữa giao diện người dùng, tầng domain và tầng infrastructure. Tầng này tương tác với CSDL khi đăng nhập và với tầng domain khi cần tạo báo cáo. Tầng domain chứa các logic nghiệp vụ của domain, các đối tượng trực tiếp liên quan tới báo cáo. Hai trong số chúng là Report và Template, đó là những lớp đối tượng để tạo báo cáo. Tầng Infrastructure sẽ hỗ trợ truy vấn CSDL và file.

Khi cần một báo cáo, người dùng sẽ lựa chọn tên của báo cáo đó trong một danh sách các tên. Đây là reportID, một chuỗi. Một số các tham số khác cũng sẽ được truyền xuống, như các mục có trong báo cáo, khoảng thời gian báo cáo. Để đơn giản ta sẽ chỉ quan tâm đến reportID, tham số này sẽ được truyền xuống từ tầng application xuống tầng domain. Tầng domain giữ trách nhiệm tạo và trả về báo cáo theo tên được yêu cầu. Vì các báo cáo được sinh ra dựa trên các mẫu, chúng ta có thể sử dụng một Dịch vụ ở đây với nhiệm vụ lấy mẫu báo cáo tương ứng với reportID. Mẫu này có thể được lưu trong file hoặc trong CSDL. Sẽ không hợp lý lắm nếu như thao tác này nằm trong đối tượng Report hay Template. Và vì thế nên ta sẽ tạo ra một Dịch vụ riêng biệt với nhiệm vụ lấy về mẫu báo cáo dựa theo ID của báo cáo đó. Dịch vụ này sẽ được đặt ở tầng domain và sử dụng chức năng truy xuất file dưới tầng infrastructure để lấy về mẫu báo cáo được lưu trên đĩa.

### 2.2.3 Mô-đun

Với hệ thống lớn và phức tạp, mô hình thường có xu hướng phình càng ngày càng to. Khi mô hình phình tới một điểm ta khó nắm bắt được tổng thể, việc hiểu quan hệ và tương tác giữa các phần khác nhau trở nên khó khăn. Vì lý do đó, việc tổ chức mô hình thành các mô-đun là cần thiết. Mô-đun được dùng như một phương pháp để tổ chức các khái niệm và tác vụ liên quan nhằm giảm độ phức tạp. Mô-đun 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 mô-đun chứa trong mô hình đó, sau đó là quan hệ giữa các mô-đun. Khi đã hiểu tương tác giữa các mô-đun, ta có thể bắt đầu tìm hiểu phần chi tiết trong từng mô-đun. Đây là cách đơn giản và hiệu quả để quản lý sự phức tạp.

Một lý do khác cho việc dùng mô-đun liên quan tới chất lượng mã nguồn. Thực tế được chấp nhận rộng rãi là, phần mềm cần có độ tương liên cao và độ liên quan thấp. Sự tương liên bắt đầu từ mức lớp (class) và phương thức (method) và cũng có thể áp dụng ở mức mô-đun. Ta nên nhóm những lớp có mức độ liên quan cao thành một mô-đun để đạt được tính tương liên cao nhất có thể. Có nhiều loại tương liên. Hai loại tương liên hay dùng nhất là tương liên giao tiếp và tương

liên chức năng. Tương liên giao tiếp có được khi hai phần của mô-đun thao tác trên cùng dữ liệu. Nó có ý nghĩa là nhóm chúng lại vì giữa chúng có quan hệ mạnh. Tương liên chức năng có được khi mọi phần của mô-đun làm việc cùng nhau để thực hiện một tác vụ đã được định nghĩa rõ. Loại này được coi là tương liên tốt nhất.

Sử dụng mô-đun trong thiết kế là cách để tăng tính tương liên là giảm tính liên kết. Mô-đun cần được tạo nên bởi những thành phần thuộc về cùng một tương liên có tính chức năng hay logic. Mô-đun cần có giao diện được định nghĩa rõ để giao tiếp với các mô-đun khác. Thay vì gọi ba đối tượng của một mô-đun, sẽ tốt hơn nếu truy cập chúng qua một giao diện để giảm tính liên kết. Tính liên kết thấp làm giảm độ phức tạp và tăng khả năng duy trì. Việc hiểu cách hệ thống hoạt động sẽ dễ hơn khi có ít kết nối giữa các mô-đun thực hiện những tác vụ đã được định nghĩa rõ hơn là việc mọi mô-đun có rất nhiều kết nối với các mô-đun khác.

Việc chọn mô-đun nói lên câu chuyện của hệ thống và chứa tập có tính tương liên của các khái niệm. Điều này thường thu được bằng tính liên kết thấp giữa các mô-đun, nhưng nếu không nhìn vào cách mô hình ảnh hưởng tới các khái niệm, hoặc nhìn quá sâu vào vào những khái niệm có thể trở thành phần cơ bản của mô-đun có thể sẽ tìm ra những cách thức có ý nghĩa. Tìm ra sự liên kết thấp theo cách mà các khái niệm được hiểu và lý giải một cách độc lập với nhau. Làm mịn mô hình này cho tới khi các phần riêng rẽ tương ứng với khái niệm domain ở mức cao và mã nguồn tương ứng được phân rã thật tốt.

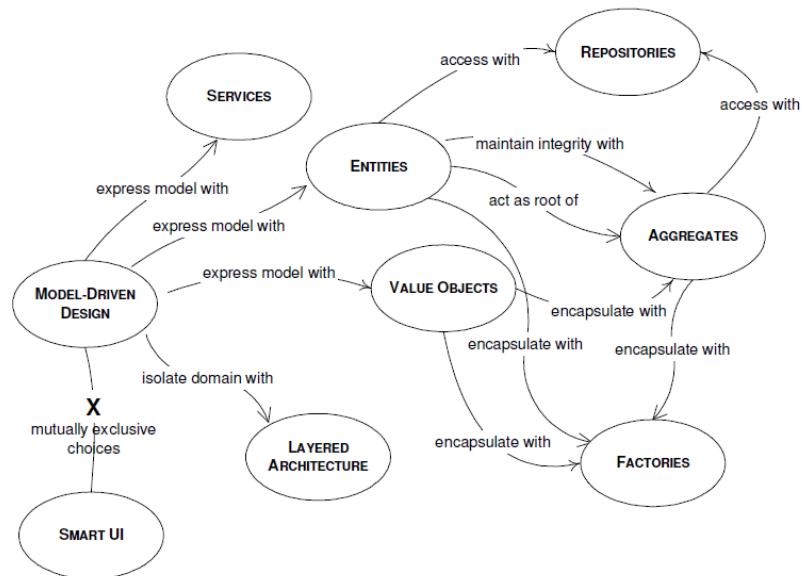
Hãy đặt tên mô-đun và lấy tên mô-đun đó là một phần của Ngôn ngữ Chung. Mô-đun và tên của nó cần thể hiện ý nghĩa sâu xa của domain.

Người thiết kế thường quen với việc tạo thiết kế ngay từ đầu. Và nó thường trở thành một phần của thiết kế. Sau khi vai trò của mô-đun được quyết định, nó thường ít thay đổi trong khi phần trong của mô-đun thì thường xuyên thay đổi. Người ta khuyến nghị rằng nên có sự linh hoạt ở mức độ nào đó, cho phép mô-đun được tiến hóa trong dự án chứ không giữ nguyên nó cố định. Thực tế là việc refactor một mô-đun thường tốn kém hơn việc refactor một lớp (class), nhưng khi ta

phát hiện lỗi trong thiết kế, tốt nhất là nên chỉ ra sự thay đổi của mô-đun và tìm ra phương án giải quyết.

### 2.3. Các mô hình trong chiến lược thiết kế phần mềm hướng lĩnh vực

Các *khuôn mẫu* quan trọng nhất được sử dụng trong DDD. Mục đích của những khuôn mẫu này 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.



Hình 2- 5 Những mẫu sử dụng trong DDD

Các phần trước đã nhấn mạnh tầm quan trọng của cách tiếp cận tới quy trình phát triển phần mềm tập trung vào lĩnh vực nghiệp vụ. Ta đã biết nó có ý nghĩa then chốt để tạo ra một model có gốc rễ là domain. Ngôn ngữ chung nên được thực hiện đầy đủ trong suốt quá trình mô hình hóa nhằm thúc đẩy giao tiếp giữa các chuyên gia phần mềm với chuyên gia domain và khám phá ra các khái niệm chính của domain nên được sử dụng trong model. Mục đích của quá trình mô hình hóa là nhằm tạo ra một model tốt. Bước tiếp theo là hiện thực nó trong code. Đây là một giai đoạn quan trọng không kém của quá trình phát triển phần mềm. Sau khi tạo ra một mô hình tuyệt vời, nhưng không chuyển thể chúng thành code đúng cách thì sẽ chỉ kết thúc bằng phần mềm có vấn đề về chất lượng.

### 2.3.1 Aggregates and Aggregate Roots

Khi quản lý vòng đời của các đối tượng trong domain. Trong suốt vòng đời của mình các đối tượng trải qua những trạng thái khác nhau, chúng được tạo ra, lưu trong bộ nhớ, sử dụng trong tính toán và bị hủy. Một số đối tượng sẽ được lưu vào trong hệ thống lưu giữ ngoài như CSDL để có thể đọc ra sau này, hoặc chỉ để lưu trữ. Một số thời điểm khác chúng có thể sẽ được xóa hoàn toàn khỏi hệ thống, kể cả trong CSDL cũng như lưu trữ ngoài.

Việc quản lý vòng đời các đối tượng trong domain không hề đơn giản, nếu như làm không đúng sẽ có thể gây ảnh hưởng đến việc mô hình hóa domain. Sau đây ta sẽ đề cập đến ba mẫu (*pattern*) thường dùng để hỗ trợ giải quyết vấn đề này. Aggregate (tập hợp) là *pattern* để định nghĩa việc sở hữu đối tượng và phân cách giữa chúng. Factory và Repository là hai *pattern* khác giúp quản lý việc tạo và lưu trữ đối tượng. Trước hết ta sẽ nói đến Aggregate.

Một mô hình có thể được tạo thành từ nhiều đối tượng trong domain. Cho dù có cẩn thận trong việc thiết kế như thế nào thì ta cũng không thể tránh được việc sẽ có nhiều mối quan hệ chằng chịt giữa các đối tượng, tạo thành một lưới các quan hệ. Có thể có nhiều kiểu quan hệ khác nhau, với mỗi kiểu quan hệ giữa các mô hình cần có một cơ chế phần mềm để thực thi nó. Các mối quan hệ sẽ được thể hiện trong mã nguồn phần mềm và trong nhiều trường hợp trong cả CSDL nữa. Quan hệ một-một giữa khách hàng và một tài khoản ngân hàng của anh ta sẽ được thể hiện như là một tham chiếu giữa hai đối tượng, hay một mối quan hệ giữa hai bảng trong CSDL, một bảng chứa khách hàng và một bảng chứa tài khoản.

Những khó khăn trong việc mô hình hóa không chỉ là đảm bảo cho chúng chứa đầy đủ thông tin, mà còn làm sao để cho chúng đơn giản và dễ hiểu nhất có thể. Trong hầu hết trường hợp việc bỏ bớt các quan hệ hay đơn giản hóa chúng sẽ đem lại lợi ích cho việc quản lý dự án. Tất nhiên là trừ trường hợp những mối quan hệ này quan trọng trong việc hiểu domain.

Mối quan hệ một-nhiều có độ phức tạp hơn so với mối quan hệ một-một vì có liên quan đến nhiều đối tượng một lúc. Mối quan hệ này có thể được đơn giản hóa bằng cách biến đổi thành mối quan hệ giữa một đối tượng và một tập hợp các

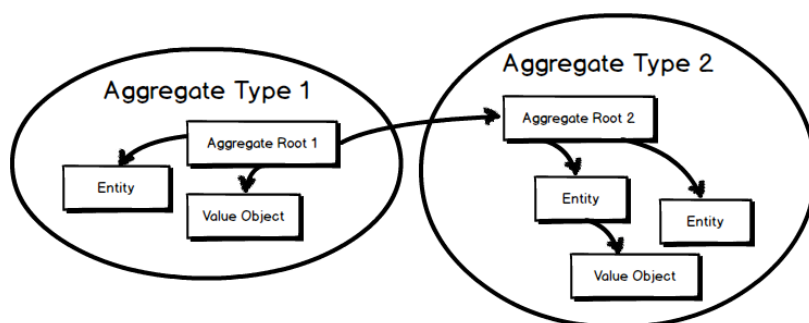


đối tượng khác, tuy nhiên không phải lúc nào cũng có thể thực hiện được điều này. Ngoài ra còn có mối quan hệ nhiều-nhiều và phần lớn trong số chúng là mối quan hệ qua lại. Điều này khiến cho độ phức tạp tăng lên rất nhiều và việc quản lý vòng đời của các đối tượng trong mối quan hệ rất khó khăn. Số quan hệ nên được tối giản càng nhỏ càng tốt. Trước hết, những mối quan hệ không phải là tối quan trọng cho mô hình nên được loại bỏ. Chúng có thể tồn tại trong domain nhưng nếu như không cần thiết trong mô hình thì nên bỏ chúng. Thứ hai, để hạn chế việc tăng số lượng quan hệ theo theo cấp số nhân, ta nên sử dụng các ràng buộc. Nếu có rất nhiều các đối tượng thỏa mãn một mối quan hệ, có thể chỉ cần một đối tượng trong số đó nếu như ta đặt một ràng buộc lên mối quan hệ. Thứ ba, trong nhiều trường hợp những mối quan hệ qua lại có thể được chuyển thành những mối quan hệ một chiều. Mỗi chiếc xe đều có động cơ và mỗi động cơ đều thuộc về một chiếc xe. Đây là một mối quan hệ qua lại, tuy nhiên ta có thể giản lược nó nếu như chỉ cần biết mỗi chiếc xe đều có một động cơ.

Sau khi ta đã tối giản và đơn giản hóa các mối quan hệ giữa các đối tượng, có thể sẽ vẫn còn rất nhiều các mối quan hệ còn lại. Lấy ví dụ một hệ thống ngân hàng lưu giữ và xử lý thông tin của khách hàng. Những dữ liệu này bao gồm các thông tin cá nhân của khách hàng như tên, địa chỉ, số điện thoại, nghề nghiệp và thông tin tài khoản: số tài khoản, số dư, các thao tác đã thực hiện. Khi hệ thống lưu trữ hoặc xóa hoàn toàn thông tin của một khách hàng thì nó cần đảm bảo rằng tất cả các tham chiếu đã được loại bỏ. Nếu như có nhiều các đối tượng lưu trữ các tham chiếu như vậy thì việc trên là rất khó khăn. Ngoài ra nữa, khi thay đổi thông tin của một khách hàng, hệ thống cần đảm bảo rằng dữ liệu đã được cập nhật trong toàn bộ hệ thống và tính toàn vẹn dữ liệu không bị xâm phạm. Thông thường yêu cầu này được đảm bảo ở dưới tầng CSDL bằng các Giao dịch. Tuy nhiên nếu như mô hình không được thiết kế cẩn thận thì có thể sẽ gây ra nghẽn ở tầng CSDL gây suy giảm hiệu năng hệ thống. Mặc dù các giao dịch đóng vai trò rất quan trọng trong CSDL, việc quản lý tính toàn vẹn dữ liệu trực tiếp trong mô hình sẽ đem lại nhiều lợi ích. Ngoài tính toàn vẹn dữ liệu, ta cũng cần đảm bảo các invariant của dữ liệu. Các Invariant là những luật, ràng buộc yêu cầu phải được thỏa mãn mỗi khi dữ liệu thay

đổi. Điều này rất khó thực hiện khi có nhiều đối tượng cùng lưu tham chiếu để sửa đổi dữ liệu. Việc đảm bảo tính thống nhất sau các thay đổi của một đối tượng có nhiều mối quan hệ không phải là dễ dàng. Các Invariant không chỉ ảnh hưởng đến từng đối tượng riêng rẽ mà thường là tập hợp các đối tượng. Nếu ta sử dụng khóa quá mức sẽ khiến cho các thao tác chồng chéo lên nhau và hệ thống không thể sử dụng được. Do vậy, ta cần sử dụng Aggregate. 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.

Vậy bằng cách nào mà Aggregate có thể đảm bảo được tính toàn vẹn và các ràng buộc của dữ liệu? Vì các đối tượng khác chỉ có thể tham chiếu đến gốc của

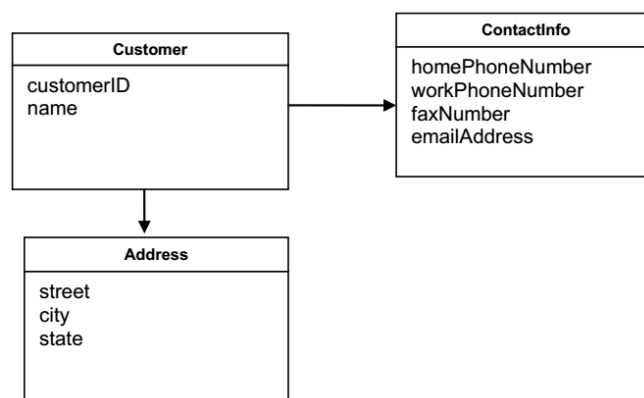


Hình 2- 6 Aggregate root

Aggregate, chúng không thể thay đổi trực tiếp đến các đối tượng nằm bên trong mà chỉ có thể thay đổi thông qua gốc, hoặc là thay đổi gốc Aggregate trực tiếp. Tất cả các thay đổi của Aggregate sẽ thực hiện thông qua gốc của chúng và ta có thể quản lý được những thay đổi này, so với khi thiết kế cho phép các đối tượng bên ngoài truy cập trực tiếp vào các đối tượng bên trong thì việc đảm bảo các invariant sẽ đơn giản hơn nhiều khi ta phải thêm các logic vào các đối tượng ở ngoài để thực hiện. Nếu như gốc của Aggregate 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.

Nhưng việc chỉ cho phép truy cập thông qua gốc không có nghĩa là ta không được phép cập các tham chiếu tạm thời của các đối tượng nội tại trong aggregate cho các đối tượng bên ngoài, miễn là các tham chiếu này được xóa ngay sau khi thao tác hoàn thành. Một trong những cách thực hiện điều này là sao chép các Value

Object cho các đối tượng ngoài. Việc thay đổi các đối tượng này sẽ không gây ảnh hưởng gì đến tính toàn vẹn của aggregate cả. Nếu như các đối tượng của Aggregate được lưu trong CSDL thì chỉ nên cho phép truy vấn trực tiếp lấy gốc của aggregate, các đối tượng nội tại còn lại nên được truy cập thông qua các mối quan hệ bên trong. Các đối tượng nội tại của Aggregate có thể lưu tham chiếu đến gốc của các Aggregate khác. Thực thể gốc có một định danh trong toàn hệ thống và giữ trách nhiệm đảm bảo các ràng buộc. Các thực thể bên trong có định danh nội bộ. Gộp các Thực thể và các Value Object thành những Aggregate và tạo các đường biên giữa chúng. Lựa chọn một Thực thể làm gốc cho một Aggregate và quản lý truy cập tới các đối tượng trong đường biên thông qua gốc. Chỉ cho phép các đối tượng bên ngoài lưu tham chiếu đến gốc. Các tham chiếu tạm thời tới các đối tượng nội bộ có thể được chép ra ngoài để sử dụng cho từng thao tác một. Vì gốc quản lý truy cập nên gốc phải biết đến mọi thay đổi nội tại của Aggregate. Cách thiết kế này giúp đảm bảo các ràng buộc trên các đối tượng của Aggregate cũng như toàn bộ Aggregate. Dưới đây là một sơ đồ của một ví dụ cho Aggregate. Đối tượng khách hàng là gốc của Aggregate, các đối tượng khác là nội tại. Nếu như một đối tượng bên ngoài cần địa chỉ thì có thể cho tham chiếu tới một bản sao của đối tượng này.



### 2.3.2 Factory

Các Thực thể và đối tượng tập hợp có thể lớn và phức tạp - quá phức tạp để tạo trong constructor của thực thể gốc (*root entity*). Trong thực tế, việc cố gắng tạo ra một Aggregate phức tạp trong hàm constructor là trái với những gì thường xảy ra về mặt domain, nơi mà những thứ thường được tạo ra bởi thứ khác (như các

thiết bị điện tử tạo từ các vi mạch). Nó giống như việc có chiếc máy in lại tự tạo ra nó vậy.

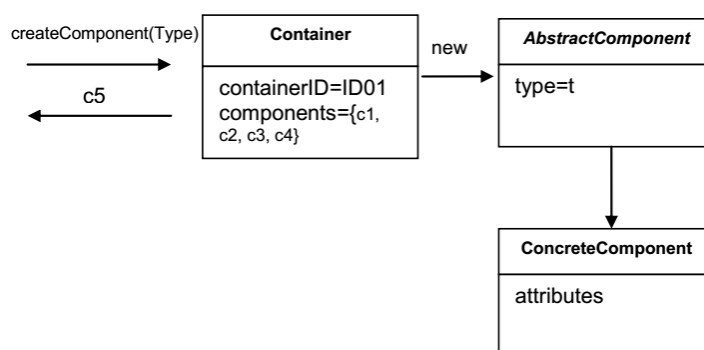
Khi một đối tượng khách thể muốn tạo ra đối tượng khác, nó gọi hàm khởi tạo của nó và có thể truyền thêm một vài tham số. Nhưng khi việc tạo đối tượng là một quá trình mệt nhọc, tạo ra đối tượng liên quan đến nhiều kiến thức về cấu trúc bên trong của đối tượng, về mối quan hệ giữa các đối tượng trong nó và các luật (*rule*) cho chúng. Điều này có nghĩa là mỗi đối tượng khách thể sẽ nắm giữ hiểu biết riêng về các đối tượng được sinh ra. Điều này phá vỡ sự đóng gói của đối tượng nghiệp vụ và của các Aggregate. Nếu các khách thể thuộc về tầng ứng dụng, một phần của tầng nghiệp vụ được chuyển ra ngoài, làm toàn bộ thiết kế bị lộn xộn.

Trong cuộc sống thực, giống như việc ta được cung cấp nhựa, cao su, kim loại và ta phải tự làm cái máy in. Điều đó thì không phải là không thể, nhưng nó có thực sự đáng để làm? Bản thân việc tạo một đối tượng có thể là thao tác chính của chính nó, nhưng những thao tác lắp ráp phức tạp không hợp với trách nhiệm của đối tượng được tạo ra. Việc tổ hợp những trách nhiệm ấy có thể tạo nên những thiết kế vụng về khó hiểu.

Do đó, ta cần đưa ra một khái niệm mới. Khái niệm này giúp việc gộp quy trình tạo ra đối tượng phức tạp và được gọi là 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.

Điều quan trọng là quá trình tạo ra (Factory) là atomic. Nếu kết quả tạo ra không phải là atomic thì có khả năng là quá trình sẽ chỉ tạo ra một số đối tượng nửa vời và chúng ở trạng thái không được định nghĩa. Điều này càng đúng hơn với Aggregate. Khi root được tạo ra, điều cần thiết là mọi đối tượng phải là bất biến cũng được tạo ra. Nếu không, những bất biến này không thể được đảm bảo. Với Đối tượng Giá trị bất biến, điều này nghĩa là mọi thuộc tính được khởi tạo với trạng thái hợp lệ của chúng. Nếu một đối tượng không thể được tạo ra một cách đúng đắn thì cần phải sinh rangeloại lệ và trả về giá trị không hợp lệ. Do đó, hãy

chuyển trách nhiệm của việc tạo instance cho đối tượng phức tạp và Aggregate tới một đối tượng riêng. Bản thân đối tượng này có thể không có trách nhiệm đối với mô hình domain nhưng vẫn là một phần của thiết kế domain. Hãy cung cấp interface gộp mọi phần rời rạc phức tạp và không yêu cầu client phải tham chiếu tới những lớp cụ thể của đối tượng đang được khởi tạo. Tạo toàn bộ Aggregate như là một đơn vị, đảm bảo những yếu tố bất biến của nó. Có nhiều design pattern có thể dùng để thực thi Factory. Có thể chia pattern này thành 2 loại Factory Phương pháp và Factory Trừu tượng. Ta không thể hiện những pattern này dưới góc độ thiết kế mà thể hiện chúng từ mô hình domain. Một Factory Phương pháp là một phương pháp đối tượng chứa và ẩn kiến thức cần thiết để tạo ra một đối tượng khác. Điều này rất hữu dụng khi client muốn tạo một đối tượng thuộc về một Aggregate. Giải pháp là thêm một method tới gốc của Aggregate. Method này sẽ đảm nhiệm việc tạo ra đối tượng, đảm bảo mọi điều kiện bất biến, trả về một tham chiếu tới đối tượng đó hoặc một bản copy tới nó.

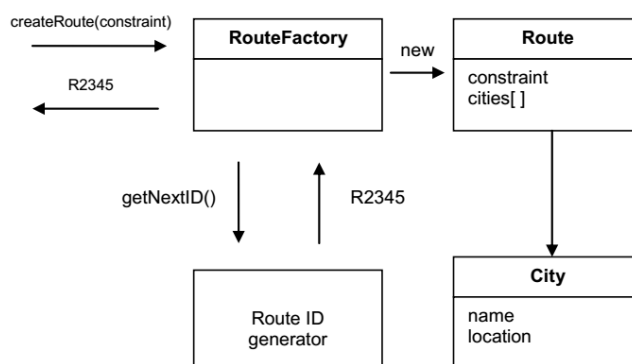


Hình 2- 7 Factory

Container chứa các thành phần và chúng thuộc một loại nào đó. Khi một thành phần được tạo ra, nó cần được tự động thuộc về một container. Việc này là cần thiết. Client gọi method createComponent(Type t) của container. Container khởi tạo một thành phần mới. Lớp cụ thể của thành phần này được xác định dựa trên loại của nó. Sau khi việc tạo thành công, thành phần được thêm vào tập các thành phần được chứa bởi container đó và trả về một copy cho client.

Với nhiều trường hợp, khi việc xây dựng một đối tượng phức tạp hơn, hoặc khi việc tạo đối tượng liên quan đến việc tạo một chuỗi các đối tượng. Ví dụ: việc tạo một Aggregate. Việc cần nhu cầu tạo nội bộ của một Aggregate có thể được

thực hiện trong một đối tượng Factory riêng dành riêng cho tác vụ này. Hãy xem ví dụ với mô-đun chương trình tính route có thể phải chạy qua của một ô tô từ điểm khởi hành tới điểm đích với một chuỗi các ràng buộc. Người dùng đăng nhập vào sitechạy chương trình và chỉ định ràng buộc: route nhanh nhất, route rẻ nhất. Route được ta có thể được chú thích bởi thông tin người dùng cần được lưu để họ có thể tham chiếu lại khi đăng nhập lần sau.



Hình 3- 4 Factory

Bộ sinh Route ID được dùng để tạo định danh duy nhất cho mỗi route, cái mà cần thiết cho một Thực Thể.

Khi tạo ra một Factory, ta buộc phải vi phạm tính đóng gói của đối tượng - là cái cần phải được thực hiện cẩn thận. Bất cứ khi nào một thứ gì đó thay đổi trong đối tượng có ảnh hưởng đến quy tắc khởi dựng hoặc trên một số invariant, ta cần chắc chắn rằng Factory cũng được cập nhật để hỗ trợ điều kiện mới. Các Factory có liên quan chặt chẽ đến các đối tượng mà chúng tạo ra. Đó là một điểm yếu, nhưng cũng có thể là một điểm mạnh. Một Aggregate chứa một loạt các đối tượng có liên quan chặt chẽ với nhau. Sự khởi dựng của gốc liên quan tới việc tạo ra các đối tượng khác nằm trong Aggregate. Đến đây đã có vài logic được đặt vào cùng nhau trong một Aggregate. Logic vốn không thuộc về bất kỳ đối tượng nào, vì chúng nói về sự khởi dựng của các đối tượng khác. Nó có vẻ để dùng cho một Factory đặc biệt được giao nhiệm vụ tạo ra toàn bộ Aggregate và nó sẽ chứa các quy tắc, ràng buộc với invariant mà buộc phải thực hiện cho Aggregate trở nên đúng đắn. Các đối tượng sẽ được giữ đơn giản và sẽ chỉ phục vụ mục đích cụ thể của chúng mà bỏ qua sự lộn xộn của logic khởi dựng phức tạp.

Factory cho Thực Thể và Factory cho Value Object là khác nhau. Giá trị của các đối tượng thường không thay đổi và tất cả các thuộc tính cần thiết phải được sinh ra tại thời điểm khởi tạo. Khi đối tượng đã được tạo ra, nó phải đúng đắn và là final. Nó sẽ không thay đổi. Các Thực Thể thì không bất biến. Chúng có thể được thay đổi sau này, bằng cách thiết lập một vài thuộc tính với việc đề cập đến tất cả các invariant mà cần được tôn trọng. Sự khác biệt khác đến từ thực tế rằng Entity cần được định danh, trong khi Value Object thì không. Đây là khi Factory không thực sự cần thiết và một constructor đơn giản là đủ. Sử dụng một constructor khi:

1. Việc khởi tạo không quá phức tạp
2. Việc tạo ra một đối tượng không liên quan đến việc tạo ra các đối tượng khác và toàn bộ thuộc tính cần thiết được truyền thông qua constructor
3. Client quan tâm đến việc cài đặt. Có thể dùng Strategy
4. Class là một loại. Không có sự phân cấp liên quan, vì vậy không cần phải lựa chọn giữa một danh sách triển khai cụ thể.

Một quan sát khác là Factory cần phải tạo ra các đối tượng từ đầu, hoặc phải được yêu cầu đề hoàn nguyên đối tượng đã tồn tại trước đó, có thể trong CSDL. Đưa các Thực Thể trở lại bộ nhớ từ nơi lưu trữ của chúng tại CSDL có liên quan đến một quá trình hoàn toàn khác so với việc tạo ra một cái mới. Một sự khác biệt rõ ràng là các đối tượng mới không cần một định danh mới. Chúng đã có một cái. Sự vi phạm của invariant được xử lý khác nhau. Khi một đối tượng mới được tạo ra từ đầu, mọi vi phạm của invariant đều kết thúc trong một ngoại lệ. Ta không thử thực hiện việc này với đối tượng được tái tạo từ CSDL. Các đối tượng cần phải được sửa chữa bằng cách nào đó, để chúng có thể hữu dụng, bằng không sẽ có mất mát dữ liệu.

### **2.3.3. Repository**

Trong thiết kế hướng lĩnh vực, đối tượng có một vòng đời bắt đầu từ khi khởi tạo và kết thúc khi chúng bị xóa hoặc lưu trữ. Một constructor hoặc một Factory sẽ lo việc khởi tạo đối tượng. Toàn bộ mục đích của việc tạo ra các đối tượng là sử dụng chúng. Trong một ngôn ngữ hướng đối tượng, người ta phải giữ một tham

chiếu đến một đối tượng để có thể sử dụng nó. Để có một tham chiếu như vậy, client hoặc là phải tạo ra các đối tượng hoặc có được nó từ nơi khác, bằng cách duyệt qua các liên kết đã có. Ví dụ như, để có được một Value Object của một Aggregate, client cần yêu cầu nó từ gốc của Aggregate. Vấn đề là bây giờ client cần có một tham chiếu tới Aggregate root. Đối với các ứng dụng lớn, điều này trở thành một vấn đề vì người ta phải đảm bảo client luôn luôn có một tham chiếu đến đối tượng cần thiết, hoặc tới chỗ nào có tham chiếu tới đối tượng tương ứng. Sử dụng một quy định như vậy trong thiết kế sẽ buộc các đối tượng giữ một loạt các tham chiếu mà có thể nó không cần. Nó tăng các ghép nối, tạo một loạt các liên kết không thực sự cần thiết.

Để sử dụng một đối tượng, đối tượng đó cần được tạo ra trước. Nếu đối tượng đó là root của một Aggregate, thì nó phải là một Thực thể và rất có thể là nó sẽ được lưu trữ trong CSDL hoặc một dạng lưu trữ khác. Nếu nó là Value Object, nó có thể lấy được từ Thực thể bằng cách duyệt từ một liên kết. Nó chỉ ra rằng một thỏa thuận tuyệt vời của các đối tượng có thể được lấy trực tiếp từ CSDL. Nó giải quyết vấn đề của việc lấy tham chiếu của đối tượng. Khi client muốn sử dụng một đối tượng, chúng truy cập CSDL, lấy đối tượng từ đó và dùng chúng. Đó dường như là giải pháp đơn giản và nhanh nhất, tuy nhiên nó có tác động tiêu cực đến thiết kế. CSDL là một phần của lớp hạ tầng. Một giải pháp bản cùng là cho client nhận biết chi tiết cách cần thiết để truy cập CSDL. Ví dụ như, client có thể tạo truy vấn SQL để lấy những dữ liệu cần thiết. Câu truy vấn dữ liệu có thể trả về một bộ bản ghi, thậm chí phơi bày các chi tiết bên trong nó. Khi nhiều client có thể tạo đối tượng trực tiếp từ CSDL, nó chỉ ra rằng code như vậy sẽ được nằm rải rác khắp toàn bộ domain. Vào thời điểm đó các domain model sẽ trở nên bị tổn thương. Nó sẽ phải xử lý rất nhiều chi tiết của tầng infrastructure thay vì chỉ làm việc với các khái niệm domain. Điều gì sẽ xảy ra nếu một quyết định tạo ra để thay đổi CSDL bên dưới? Tất cả các code nằm rải rác cần được thay đổi để có thể truy cập hệ thống lưu trữ mới. Khi client truy cập CSDL trực tiếp, nó có thể sẽ khôi phục một đối tượng nằm trong một Aggregate. Nó phá vỡ sự đóng gói của Aggregate với một hậu quả không lường trước. Một client cần một công cụ thiết thực thu thập tham chiếu tới đối tượng

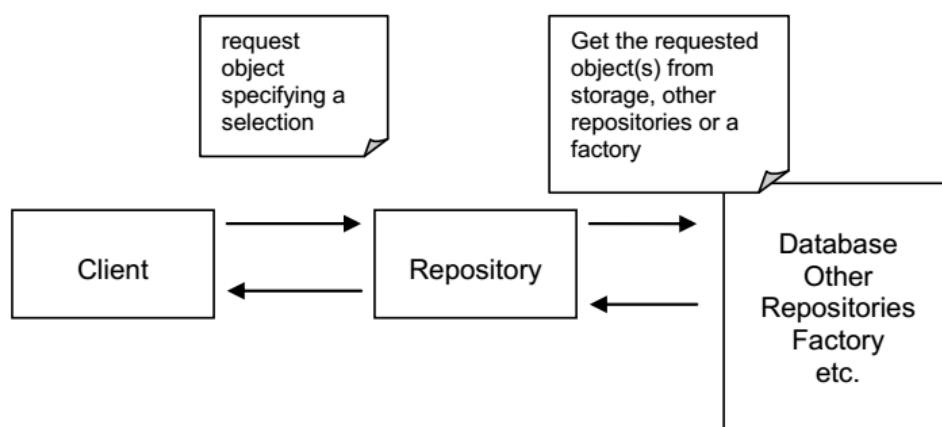


domain có từ trước. Nếu lớp infrastructure làm cho nó dễ dàng để làm như vậy, các lập trình viên của phía client có thể thêm nhiều liên kết có thể duyệt hơn, làm lộn xộn model. Mặt khác, họ có thể sử dụng các truy vấn để lấy dữ liệu chính xác mà họ cần từ các CSDL, hoặc lấy một vài đối tượng riêng biệt hơn là phải qua các Aggregate root. Nghiệp vụ domain chuyển vào trong truy vấn và code phía client, thực thể cùng Value Object trở thành các thùng chứa dữ liệu đơn thuần. Sự phức tạp kỹ thuật của việc áp dụng hầu hết việc truy cập CSDL vào lớp Hạ tầng sẽ làm mã client phình to, dẫn các lập trình viên tới việc giảm chất lượng tầng domain, khiến cho model không còn thích hợp. Ảnh hưởng chung là việc tập trung vào domain bị mất đi và thiết kế bị tổn hại.

Vì thế, ta sử dụng một Repository, mục đích của nó là để đóng gói tất cả các logic cần thiết để thu được các tham chiếu đối tượng. 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 và model lấy lại được sự rõ ràng và tập trung.

Repository có thể lưu trữ các tham chiếu tới một vài đối tượng. Khi một đối tượng được khởi tạo, nó có thể được lưu lại trong Repository và được lấy ra từ đây để có thể sử dụng sau này. Nếu phía client yêu cầu đối tượng từ Repository và Repository không chứa chúng, nó có thể sẽ được lấy từ bộ nhớ. Dù bằng cách nào, các Repository hoạt động như một nơi lưu trữ các đối tượng cho việc truy xuất đối tượng toàn cục.

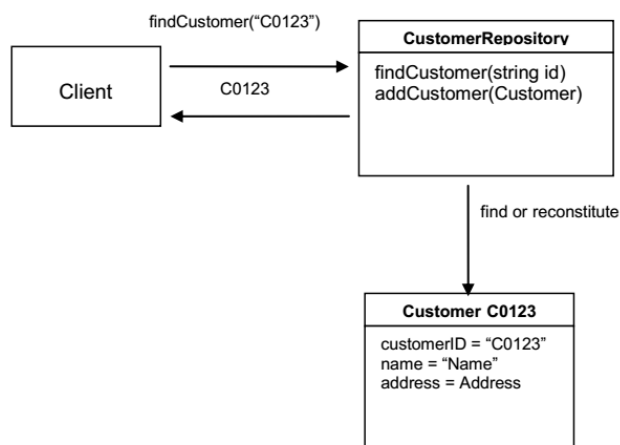
Repository có thể cũng chứa một Strategy. Nó có thể truy cập một bộ nhớ lưu trữ hoặc cách khác dựa trên Strategy chỉ định. Nó cũng có thể sử dụng các loại bộ nhớ khác nhau cho các loại khác nhau của các đối tượng. Hiệu quả chung là đối tượng domain được tách rời khỏi các nhu cầu lưu trữ đối tượng và các tham chiếu của chúng và truy cập lưu trữ phía dưới tầng infrastructure.



*Hình 2- 8 Repository*

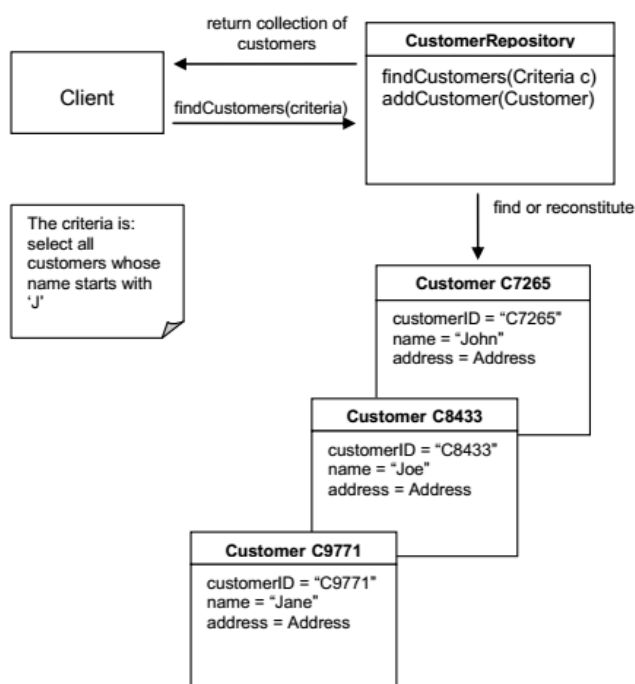
Với mỗi loại đối tượng cần thiết để truy cập toàn cục, tạo một đối tượng có thể cung cấp một tập hợp trong bộ nhớ ảo của tất cả các đối tượng trong các loại đó. Cài đặt truy cập thông qua một Interface được biết ở mức toàn cục. Cung cấp các phương thức để thêm hoặc xóa đối tượng, nhằm đóng gói việc thêm và xóa thật sự của dữ liệu trong bộ lưu trữ dữ liệu. Cung cấp phương thức lấy đối tượng dựa trên một vài tiêu chí và trả về toàn bộ đối tượng đã được khởi tạo hoặc tập hợp của đối tượng có thuộc tính trùng lặp với tiêu chí, qua đó đóng gói được bộ lưu trữ thật sự và kỹ thuật truy vấn. Chỉ cung cấp các repository cho Aggregate root mà thực sự cần một truy cập trực tiếp. Giữ client tập trung vào model, ủy quyền tất cả việc lưu trữ đối tượng và truy cập chúng cho một Repository.

Một Repository có thể chứa thông tin chi tiết sử dụng cho việc truy cập tới infrastructure, nhưng nó chỉ nên là một interface đơn giản. Một Repository nên có một bộ tập hợp các phương thức dùng cho lấy đối tượng. Client gọi một phương thức như vậy và truyền một hoặc nhiều tham số đại diện cho các tiêu chí dùng để lấy đối tượng hoặc bộ tập hợp đối tượng trùng với tiêu chí. Một Thực Thể có thể dễ dàng nhận diện bởi ID của chúng. Các tiêu chí lựa chọn khác có thể được tạo thành từ một tập hợp các thuộc tính của đối tượng. Repository sẽ so sánh tất cả đối tượng trái với tiêu chí và trả lại tập hợp thỏa mãn tiêu chí. Repository interface có thể chứa phương thức được sử dụng cho thực hiện vài tính toán bổ sung giống như số của đối tượng của một kiểu nhất định. Việc cài đặt một repository có thể rất giống infrastructure, nhưng repository interface sẽ chỉ là một model thuần túy.



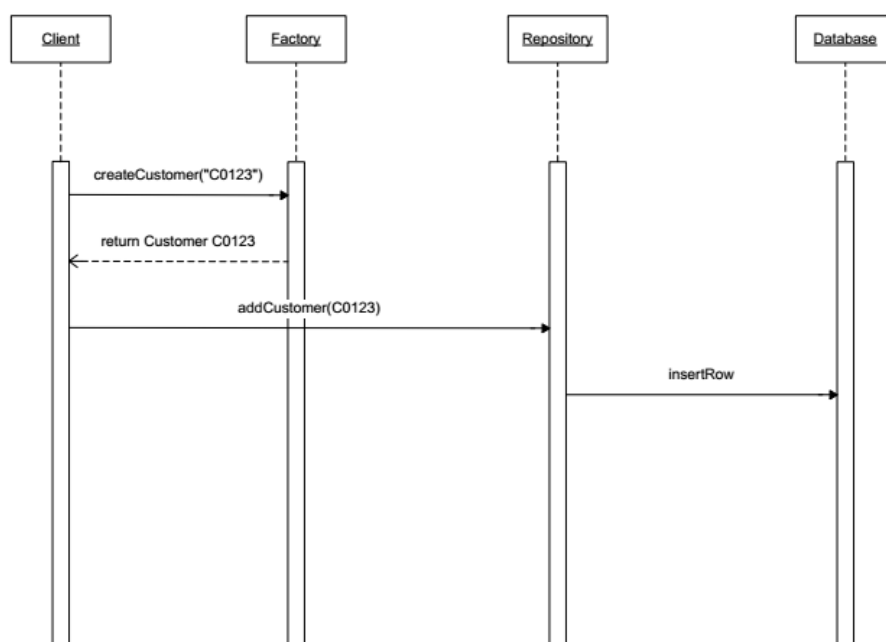
Hình 2- 9 Cài đặt repository

Các lựa chọn khác là để xác định một tiêu chí lựa chọn như là một Đặc tả. Đặc tả cho phép định nghĩa một tiêu chí phức tạp hơn, chẳng hạn như phía dưới đây:



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. Repository có thể cache đối tượng cục bộ, nhưng hầu hết chúng cần lấy đối tượng từ một bộ lưu trữ lâu dài nào đó. Đối tượng được tạo ra bằng cách sử dụng một constructor hoặc được trả về bởi một Factory khởi tạo. Với lý do đó, Repository có thể xem như là một Factory vì chúng có tạo ra đối tượng.

Nó không khởi tạo ngay từ đầu, nhưng nó là một sự hoàn nguyên của đối tượng đã tồn tại. Ta không nên trộn một Repository với một Factory. Factory nên để tạo mới đối tượng, trong khi đó Repository nên tìm các đối tượng đã được khởi tạo. Khi một đối tượng mới được thêm vào Repository, chúng nên được khởi tạo bằng Factory trước đó và sau đó chúng nên được trao lại cho Repository nhằm lưu trữ chúng giống với ví dụ dưới đây.



Một cách khác để lưu ý là các Factory là "*domain thuần khiết*", nhưng các Repository có thể chứa các liên kết tới infrastructure, như là database.

### 2.3.4 Bounded Contexts

Mỗi mô hình đều có một ngữ cảnh tương ứng với nó. Ta làm việc với một mô hình, ngữ cảnh là ẩn. Ta không cần định nghĩa chúng. Khi ta tạo một chương trình tương tác với phần mềm khác, ví dụ một ứng dụng cũ, hiển nhiên là ứng dụng mới có mô hình và ngữ cảnh riêng của nó và chúng được chia riêng với mô hình và ngữ cảnh cũ. Chúng không thể gộp, trộn lẫn và không bị hiểu lầm. Tuy nhiên, khi ta làm việc với một ứng dụng doanh nghiệp lớn, ta cần định nghĩa ngữ cảnh cho từng mô hình ta tạo ra.

Dự án lớn nào cũng có nhiều mô hình. Tuy vậy, mã nguồn dựa trên các mô hình riêng biệt lại được gộp lại và phần mềm trở nên nhiều lỗi, kém tin tưởng và

khó hiểu. Giao tiếp giữa các thành viên trong nhóm trở nên rối. Thường thì việc quyết định mô hình nào áp dụng cho ngữ cảnh nào là không rõ ràng.

Không có một công thức nào để chia mọi mô hình to thành nhỏ thành nhỏ hơn. Hãy thử đặt những thành phần có liên quan vào mô hình theo những khái niệm một cách tự nhiên. Một mô hình cần đủ nhỏ để nó phù hợp với một nhóm. Sự phối hợp và trao đổi nhóm sẽ ý tưởng chính cho việc định nghĩa một mô hình là vẽ ra ranh giới giữa các ngữ cảnh, sau đó cố gắng giữ các mô hình có được sự thống nhất càng cao càng tốt. Việc giữ một mô hình "*thuần khiết*" khi nó mở rộng ra cả dự án doanh nghiệp là rất khó, nhưng dễ dàng hơn khi ta hạn chế trong một mảng cụ thể. Định nghĩa một cách hiển minh ngữ cảnh trong ngữ cảnh giới hạn. Hãy xác định ranh giới giữa các điều khoản của tổ chức nhóm, dùng nó trong một phần cụ thể của mô hình, thể hiện nó một cách vật lý bằng, chẳng hạn như schema của mã nguồn. Duy trì mô hình này tương thích chặt chẽ với các giới hạn nó. Tuy vậy, đừng để mất tập trung hay rối loạn vì những vấn đề bên ngoài.

Một ngữ cảnh giới hạn không phải là một mô-đun. Một ngữ cảnh giới hạn cung cấp khung logic bên trong của mô hình. Mô-đun tổ chức các thành phần của mô hình, do đó ngữ cảnh giới hạn chứa mô-đun thông suốt và hoàn chỉnh, giúp cho lập trình viên làm việc trên cùng một mô hình. Ngữ cảnh của mô hình là tập các điều kiện cần được áp dụng để đảm bảo những điều khoản của mô hình có ý nghĩa cụ thể.

Việc có nhiều mô hình có cái giá phải trả. Ta cần định nghĩa ranh giới và quan hệ giữa các mô hình khác nhau. Điều này đòi hỏi nhiều công sức và có thể cần sự "*phiên dịch*" giữa các mô hình khác nhau. Ta không thể chuyển đổi tương đương bất kỳ giữa các mô hình khác nhau và ta không thể gọi tự do giữa các mô hình như là không có ranh giới nào. Đây không phải là một công việc quá khó và lợi ích nó đem lại đáng được đầu tư.

Ví dụ ta muốn tạo ra một phần mềm thương mại điện tử để bán phần mềm trên Internet. Phần mềm này cho phép khách hàng đăng ký và ta thu thập dữ liệu cá nhân, bao gồm số thẻ tín dụng. Dữ liệu được lưu trong CSDL quan hệ. Khách hàng có thể đăng nhập, duyệt trang web để tìm hàng và đặt hàng. Chương trình cần công

bổ sự kiện khi có đơn đặt hàng vì ai đó cần phải gửi thư có hạng mục đã được yêu cầu. Ta cần xây dựng giao diện báo cáo dùng cho việc tạo báo cáo để có thể theo dõi trạng thái hàng còn, khách hàng muốn mua gì, họ không thích gì... Ban đầu, ta bắt đầu bằng một mô hình phủ cả domain thương mại điện tử. Ta muốn làm thế vì sau chúng, ta cũng sẽ cần làm một chương trình lớn. Tuy nhiên, ta xem xét công việc này cẩn thận và phát hiện ra rằng chương trình e-shop thực ra không liên quan đến chức năng báo cáo. Chúng quan tâm đến cái khác, vận hành theo một cách khác và thậm chí, không cần dùng công nghệ khác. Điểm chung duy nhất ở đây là khách hàng và dữ liệu hàng hóa được lưu trong CSDL mà cả hai đều truy cập vào. Cách tiếp cận được khuyến nghị ở đây là tạo mô hình riêng cho mỗi domain, một domain cho e-commerce, một cho phần báo cáo. Chúng có thể tiến hóa tự do không cần quan tâm tới domain khác và có thể trở thành những chương trình độc lập. Có trường hợp là chương trình báo cáo cần một số dữ liệu đặc thù mà chương trình e-commerce lưu trong CSDL dù cả hai phát triển độc lập. Một hệ thống thông điệp cần để báo cáo cho người quản lý kho về đơn đặt hàng để họ có thể gửi mail về hàng đã được mua. Người gửi mail sẽ dùng chương trình và cung cấp cho họ thông tin chi tiết về hàng đã mua, số lượng, địa chỉ người mua cũng như yêu cầu giao hàng. Việc có mô hình e-shop phủ cả hai domain là không cần thiết. Sẽ đơn giản hơn nhiều cho chương trình e-shop chỉ gửi Đối tượng Giá trị chứa thông tin mua tới kho bằng phương thức thông điệp phi đồng bộ. Rõ ràng là có hai mô hình có thể phát triển độc lập và ta chỉ cần đảm bảo giao diện giữa chúng hoạt động tốt.

#### **2.4. Quy trình phân tích và thiết kế phần mềm hướng lĩnh vực**

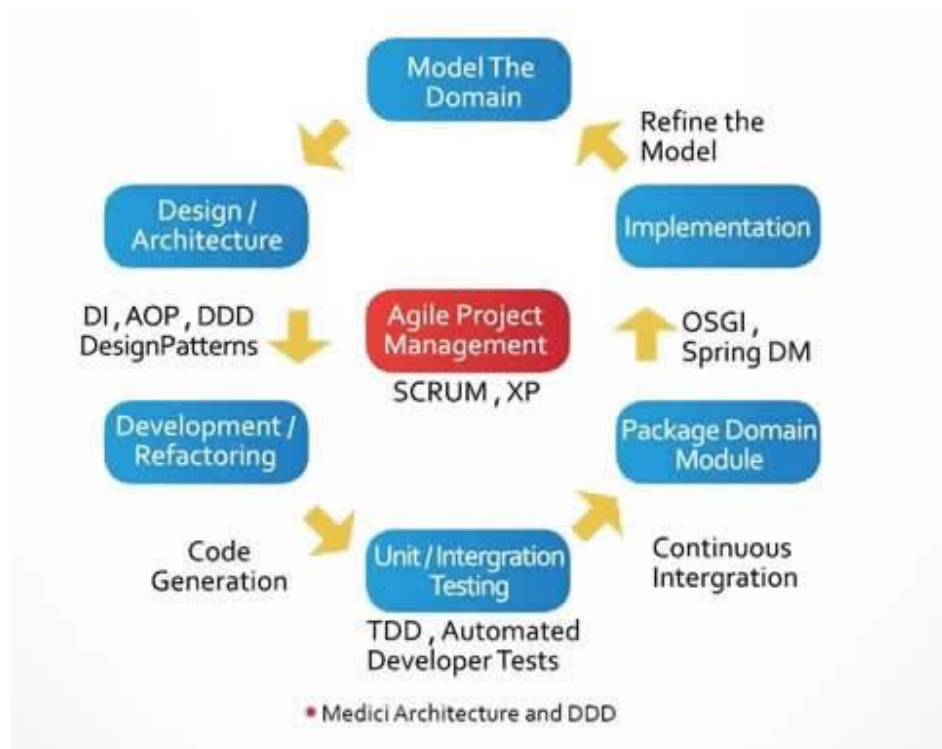
Một dự án theo mô hình DDD gồm các bước như sau:

- Mô hình và quy trình hóa tài liệu nghiệp vụ
- Lựa chọn một quy trình nghiệp vụ chuẩn và làm việc với các chuyên gia về kiểm chứng các tài liệu nghiệp vụ thông qua việc sử dụng ngôn ngữ dùng chung.
- Xác định tất cả các dịch vụ được yêu cầu cho quy trình nghiệp vụ đó. Những dịch vụ có thể là các bước đơn lẻ hoặc là nhiều bước không có Work - Flow.

- Định danh, xác định trạng thái, hành vi của các đối tượng được sử dụng bởi các dịch vụ đã được xác định ở bước trước

Điều quan trọng là giữ mô hình ở mức độ cao ban đầu tập trung vào các thành phần cốt lõi của lĩnh vực nghiệp vụ. Từ một dự án quản lý chuẩn, một thực tế khi thực hiện một dự án DDD bao gồm các giai đoạn tương tự như bất kỳ dự án phát triển phần mềm khác. Các pha bao gồm:

- Mô hình hóa lĩnh vực
- Thiết kế
- Phát triển
- Unit and Integration Testing
- Tinh chỉnh và cấu trúc lại mô hình nghiệp vụ dựa vào việc thiết kế và phát triển (tích hợp liên tục các định nghĩa mô hình). Lặp lại các bước trên bằng cách cập nhật lại các mô hình nghiệp vụ



Hình 2- 10 Quy trình thiết phát triển phần mềm theo hướng DDD

## **Chương 3: Ứng dụng chiến lược thiết kế hướng lĩnh vực trong việc xây dựng phần mềm quản lý tài khoản tập trung theo hướng dịch vụ microservice**

### **3.1 Mô tả bài toán quản lý tài khoản dùng chung tại trường ĐHDL Hải Phòng**

Trường Đại học Dân lập Hải Phòng được thành lập vào ngày 24 tháng 09 năm 1997 GS.TS.NGƯT Trần Hữu Nghị làm hiệu trưởng. Cho đến nay Nhà trường đã đào tạo ra hàng nghìn cử nhân, kỹ sư cho cả nước. Với cơ sở vật chất khang trang hiện đại. Nhà trường luôn chủ động tin học hóa các nghiệp vụ đào tạo chính của nhà trường. Trường ĐHDL Hải Phòng là một trong số ít trường áp dụng thành công bài toán quản lý đào tạo tín chỉ một cách triệt để. Sau 20 năm phát triển nhà trường đã không ngừng đầu tư và phát triển các ứng dụng phần mềm vào công tác quản lý đào tạo có những sản phẩm mua của các công ty bên ngoài, có những sản phẩm do nhà trường xây dựng mỗi một phần mềm lại có hệ thống tài khoản dùng riêng. Nhiều phần mềm không còn có thể đáp ứng được với những yêu cầu thực tế của Nhà trường. Mỗi một sinh viên có tới 4 đến 5 tài khoản khác nhau dẫn đến gây rất nhiều khó khăn cho sinh viên. Bên cạnh đó các ứng dụng phát triển sau không thể kế thừa hoặc tận dụng lại được các ứng dụng đã có trước nên việc xây dựng phần mềm mất nhiều thời gian, công sức hơn.



*Hình 3- 5 Quá trình phát triển các mô hình ứng dụng phần mềm của nhà trường*

Trước đây các phần mềm phát triển và chạy độc lập nhau thành các nguyên khối. Do đó việc trao đổi thông tin cũng như mở rộng ứng dụng là không thể. Dữ liệu ngày càng nhiều dẫn đến hệ thống chạy chậm và không linh hoạt. Sau đó ứng dụng mô hình SOA (Services Orient Architecture) đối với các ứng dụng. Tuy nhiên với mô hình này các dịch vụ khó có thể mở rộng.



### ***Đề xuất giải pháp cho các vấn đề đặt ra:***

- Áp dụng chiến lược thiết kế hướng miền cho việc phân tích thiết kế các bài toán đào tạo của nhà trường
- Tìm hiểu và phát triển các ứng dụng của nhà trường tiếp cận theo mô hình Microservice (Microservice có thể từ một ứng dụng chia nhỏ thành các service nhỏ dễ dàng mở rộng thêm các dịch vụ một cách nhanh chóng. Các dịch vụ hoạt động một cách độc lập). Mô hình Microservice thay thế cho các phần mềm xây dựng theo mô hình nguyên khối (monolithic) hay mô hình kiến trúc hướng dịch vụ SOA (Services Orient Architecture).
- Xây dựng dịch vụ quản lý tài khoản tập trung cho tất cả các đối tượng (sinh viên, giảng viên, cán bộ, nhân viên..) sử dụng các dịch vụ của nhà trường. Chỉ cần sử dụng một tài khoản có thể đăng nhập vào các hệ thống đã đăng ký với các quyền quy định. Dịch vụ này được sử dụng cho các dịch vụ khác trong hệ thống. Dịch vụ được xây dựng trên nền tảng Webbase dễ dàng cho việc bảo trì, bảo dưỡng và triển khai, hỗ trợ cả các thiết bị mobile.

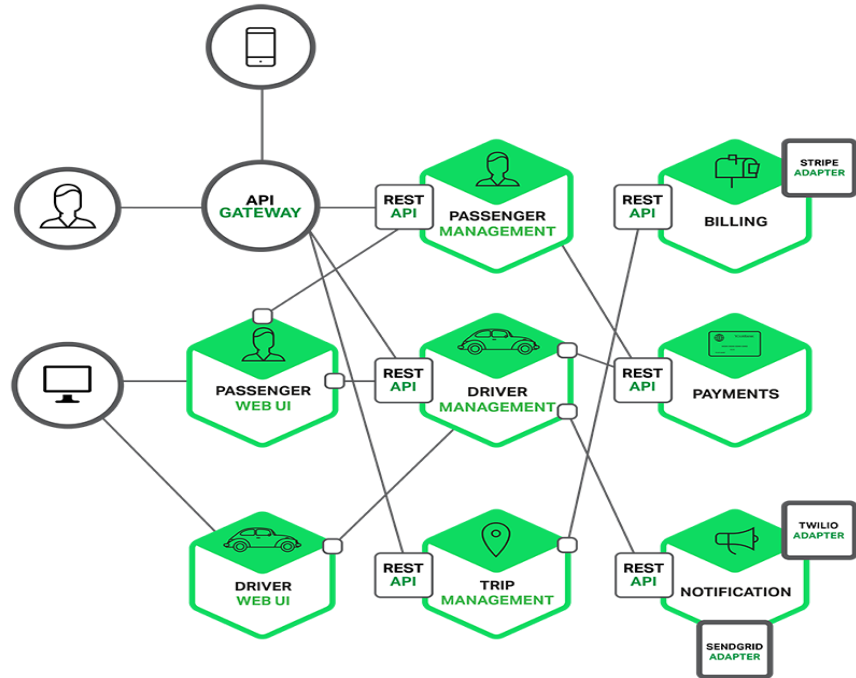
### **3.2 Tìm hiểu kiến trúc Microservices**

Ứng dụng quản lý tài khoản tập trung được xây dựng theo hướng Microservices. Microservices là kiến trúc dịch vụ siêu nhỏ “*Microservice Architecture*” phát triển nhanh chóng trong nhiều năm gần đây nhằm mô tả cách thiết kế phần mềm ứng dụng mà các dịch vụ có thể triển khai một cách độc lập. Mặc dù không có định nghĩa rõ ràng về kiểu kiến trúc này, ta vẫn có thể kể đến rất nhiều đặc tính chung của các tổ chức, phạm vi nghiệp vụ, tính chất quản lý phân tán cũng như dữ liệu khi nói đến microservices.

Nhiều tập đoàn như Amazon, eBay, Netflix đã giải quyết vấn đề ứng dụng một khối bằng kiến trúc microservices (*nhiều dịch vụ nhỏ*). Ý tưởng là chia nhỏ ứng dụng lớn ra thành các dịch vụ nhỏ kết nối với nhau.

Mỗi dịch vụ nhỏ thực hiện một tập các chức năng chuyên biệt như quản lý đơn hàng, quản lý khách hàng. Mỗi dịch vụ là một ứng dụng nhỏ có kiến trúc đa diện lõi là business logic kết nối ra các adapter khác nhau. Một số dịch vụ nhỏ lộ ra giao tiếp lập trình API cho dịch vụ nhỏ khác hay ứng dụng client gọi tới. Khi

vận hành, mỗi dịch vụ nhỏ được chạy trong một máy ảo (virtual machine) hoặc Docker container (ảo hóa tầng ứng dụng).



Hình 3- 6 Microservices của một công ty điều hành taxi kiểu Uber, Hailo [13]

Mỗi vùng chức năng giờ được thực thi bởi một dịch vụ nhỏ. Ứng dụng web cũng có thể chia nhỏ hơn chuyên cho từng đối tượng người dùng (một cho hành khách taxi, một cho tài xế). Thiết kế giao diện cho từng đối tượng người dùng giúp tối ưu trải nghiệm tốt hơn, tốc độ nhanh hơn, dễ tương thích hơn trong khi chức năng tối giản hơn.

Mỗi dịch vụ đằng sau (back end service) lộ ra REST API (hiện nay còn nhiều lựa chọn khác như Google Protobuf, Apache Thrift, Apache Avro tốn ít băng thông hơn REST JSON)

Các dịch vụ sẽ gọi / sử dụng API cung cấp bởi dịch vụ khác. Ví dụ dịch vụ quản lý taxi sử dụng Notification Server để chủ động báo tài xế đang rảnh đón khách hàng tiềm năng. Phần giao diện (UI services) sẽ gọi đến các dịch vụ khác để lấy dữ liệu hiển thị. Hiện nay, pattern reactive cho phép dịch vụ có thể thông báo hoặc chủ động gửi dữ liệu mới để giao diện cập nhật. Đặc điểm của kết nối giữa các dịch vụ có thể là:

- Synchronous (đồng bộ - gọi xong chờ)

- Asynchronous (bất đồng bộ - gọi xong chạy tiếp. Khi có kết quả thì xử lý),

Cách gọi:

- REST (tập lệnh gửi qua HTTP để truy vấn, thao tác dữ liệu. Kiểu dữ liệu XML, JSON, JSONb)

- RPC (remote procedure call - lệnh gọi từ xa. Kiểu dữ liệu binary, Thrift, Protobuf, Avro)

- SOAP (Simple Object Access Protocol)

Một số dịch API REST có thể lộ ra cho thiết bị di động của hành khách và tài xế kết nối. Ứng dụng của người dùng cuối sẽ không được kết nối trực tiếp vào dịch vụ đằng sau. Thay vào đó có một cổng API (API gateway) đứng giữa. Cổng API có một số nhiệm vụ như phân tải, lưu tạm (cache), kiểm tra quyền truy cập, đo và theo dõi (API metering and monitoring).

Kiến trúc microservices ảnh hưởng lớn đến quan hệ ứng dụng và cơ sở dữ liệu. Thay vì dùng chung một cơ sở dữ liệu giữa các dịch vụ, mỗi dịch vụ sẽ có CSDL riêng. Cách này đi ngược lại tập quán tập trung hóa cơ sở dữ liệu. Hệ quả là sẽ có dư thừa dữ liệu, cơ chế foreign key ràng buộc quan hệ dữ liệu không thể áp dụng với bảng ở 2 cơ sở dữ liệu tách biệt. Thiết kế này sẽ gây sốc vì trước đây ta đã quá quen với mô hình client - server, ở đó cơ sở dữ liệu luôn là một trung tâm, tập hợp mọi bảng. [13]

#### **Ưu điểm của Microservices:**

- Giảm thiểu sự gia tăng phức tạp rối rắm hệ thống lớn.
- Chia nhỏ ứng dụng một khối công kênh thành các dịch vụ nhỏ dễ quản lý, bảo trì nâng cấp, tự do chọn, nâng cấp công nghệ mới.
- Mỗi dịch vụ nhỏ sẽ định ra ranh giới rõ ràng dưới dạng RPC hay API hướng thông điệp.
- Microservice thúc đẩy tách rạch rời các khối chức năng (*loose coupling - high cohesion*), điều rất khó thực hiện với ứng dụng một khối. Nếu muốn loose coupling - high cohesion trong ứng dụng một khối, sẽ phải thiết kế theo Design Pattern (Gang Of Four) và liên tục tái cấu trúc (refactor)

- Mỗi dịch vụ nhỏ sẽ phát triển dễ hơn, nhanh hơn, dễ viết mã kiểm thử tự động.
- Một số dịch vụ có thể thuê ngoài phát triển mà vẫn bảo mật hệ thống - mã nguồn phần dịch vụ còn lại. Đội phát triển có nhiều lựa chọn công nghệ mới, framework, CSDL mới, đa dạng để nâng cấp từng dịch vụ nhỏ, chọn môi trường tối ưu nhất để chạy. Các dịch vụ có thể bật tắt để kiểm nghiệm so sánh A|B, tăng tốc quá trình cải tiến giao diện. Triển khai đều đặn khả thi với microservice. Dịch vụ nhỏ đóng gói trong Docker container có thể chuyển từ môi trường phát triển sang môi trường chạy thật không phải cấu hình thủ công lại, không phải copy file quá lớn.

#### **Nhược điểm của microservices:**

Nhược điểm đầu tiên của microservices cũng chính từ tên gọi của nó. Microservice nhấn mạnh kích thước nhỏ gọn của dịch vụ. Một số lập trình đề xuất dịch vụ siêu nhỏ cỡ dưới 100 dòng code. Chia quá nhiều sẽ dẫn đến manh mún, vụn vặt, khó kiểm soát. Việc lưu dữ liệu cục bộ bên trong những dịch vụ quá nhỏ sẽ khiến dữ liệu phân tán quá mức cần thiết. Nhược điểm tiếp của microservice đến từ đặc điểm hệ thống phân tán (distributed system):

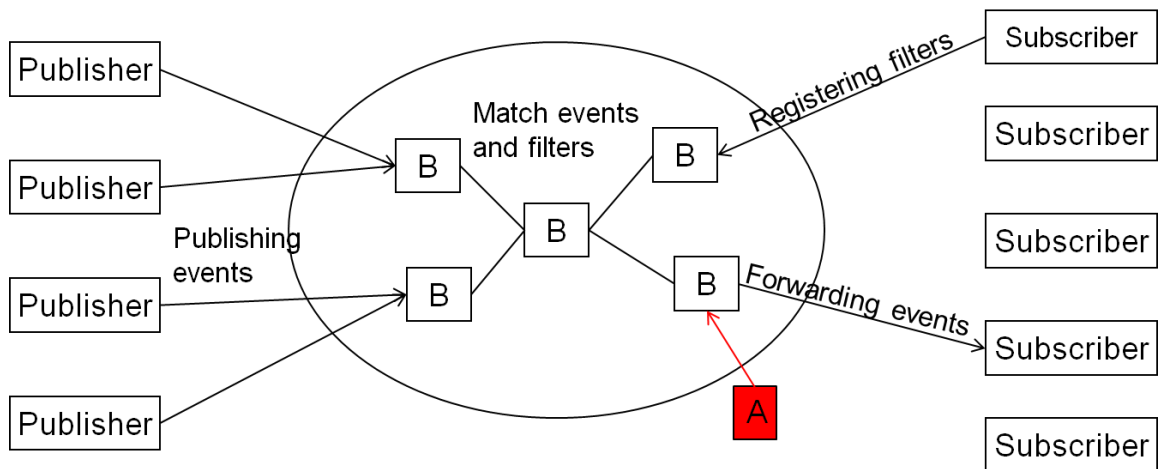
- Phải xử lý sự cố khi kết nối chậm, lỗi khi thông điệp không gửi được hoặc thông điệp gửi đến nhiều đích đến vào các thời điểm khác nhau.
- Đảm bảo giao dịch phân tán (distributed transaction) cập nhật dữ liệu đúng đắn (all or none) vào nhiều dịch vụ nhỏ khác nhau khó hơn rất nhiều, đôi khi là không thể so với đảm bảo giao dịch cập nhật vào nhiều bảng trong một cơ sở dữ liệu trung tâm.
- Theo nguyên tắc CAP (CAP theorem) thì giao dịch phân tán sẽ không thể thỏa mãn cả 3 điều kiện: consistency (dữ liệu ở điểm khác nhau trong mạng phải giống nhau), availability (yêu cầu gửi đi phải có phúc đáp), partition tolerance (hệ thống vẫn hoạt động được ngay cả khi mạng bị lỗi). Những công nghệ cơ sở dữ liệu phi quan hệ (NoSQL) hay môi giới thông điệp (message broker) tốt nhất hiện nay cũng chưa vượt qua nguyên tắc CAP.

- Kiểm thử tự động một dịch vụ trong kiến trúc microservices đôi khi yêu cầu phải chạy cả các dịch vụ nhỏ khác mà nó phụ thuộc. Do đó khi phân rã ứng dụng một khối thành microservices cần luôn kiểm tra mức độ ràng buộc giữa các dịch vụ mềm dẻo hơn hay cứng nhắc - lệ thuộc hơn. Nếu ràng buộc ít đi, lỏng lẻo hơn, bạn đi đúng hướng và ngược lại.
- Nếu các dịch vụ nhỏ thiết kế phụ thuộc vào nhau theo chuỗi. A gọi B, B gọi C, C gọi D. Nếu một mắt xích có giao tiếp API thay đổi, liệu các mắt xích khác có phải thay đổi theo không? Nếu có thì việc bảo trì, kiểm thử sẽ phức tạp tương tự ứng dụng một khối. Thiết kế dịch vụ tốt sẽ giảm tối đa ảnh hưởng lan truyền đến các dịch vụ khác.
- Triển khai dịch vụ microservices nếu làm thủ công theo cách đã làm với ứng dụng một khối phức tạp hơn rất nhiều. Ứng dụng một khối bổ sung các server mới giống hệt nhau đằng sau bộ cần bằng tại. Trong khi ở kiến trúc microservice, các dịch vụ nhỏ nằm trên nhiều máy ảo hay Docker container khác nhau, hoặc một dịch vụ có nhiều thực thể phân tán ra nhiều. Trong dịch vụ đám mây, các máy ảo, docker container, thực thể có thể linh động bật tắt, dịch chuyển. Vậy cần thiết phải có một cơ chế phát hiện dịch vụ (service discovery mechanism) để cập nhật tự động địa chỉ IP và cổng, mô tả, phiên bản của mỗi dịch vụ.

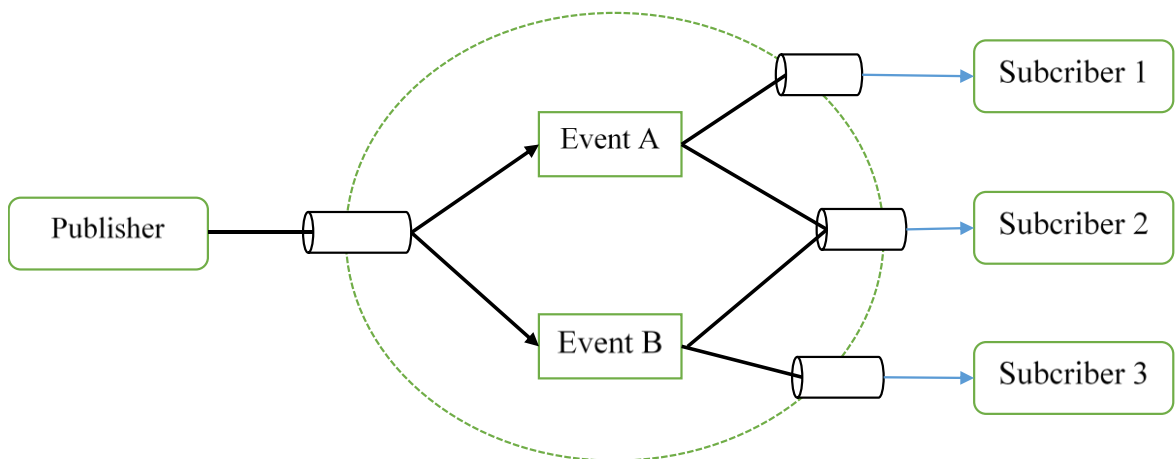
## **Kết luận**

Kiến trúc một khối sẽ hữu hiệu đối với ứng dụng đơn giản, ít chức năng. Nó bộc lộ nhiều nhược điểm khi ứng dụng phát triển lớn nhiều chức năng. Kiến trúc microservices chia nhỏ kiến trúc một khối ra các dịch vụ nhỏ. Microservices sẽ hiệu quả, phù hợp cho những ứng dụng phức tạp, liên tục phát triển nếu được thiết kế đúng và tận dụng các công nghệ quản lý, vận hành tự động.

### 3.3 Tìm hiểu mô hình Publisher – Subscriber Event



Hình 3- 7 Mô hình Publisher – Subscriber Events



Hình 3- 8 Mô hình kiến trúc liên lạc

Khi ứng dụng được chia nhỏ ra thành các Service độc lập theo kiến trúc Microservices. Để đảm bảo trao đổi dữ liệu cần có một cơ chế thực hiện để các service ít phụ thuộc nhau. Trong ứng dụng này tác giả xin được sử dụng cơ chế mô hình Publisher – Subscriber. Service này có thể đẩy (Pub) các sự kiện vào bộ đệm và các Service khác có thể lắng nghe (listener) và lấy về (Sub).

Hệ thống được xây dựng tự động, là các service có thể hoàn toàn đăng ký các Publisher hay các Subscriber sự kiện.

Có thể thỏa mãn cả 3 điều kiện: consistency (dữ liệu ở điểm khác nhau trong mạng phải giống nhau), availability (yêu cầu gửi đi phải có phúc đáp), partition tolerance (hệ thống vẫn hoạt động được ngay cả khi mạng bị lỗi).

### **3.4 Phân tích và thiết kế yêu cầu phần mềm hướng lĩnh vực**

#### **a) Mục đích**

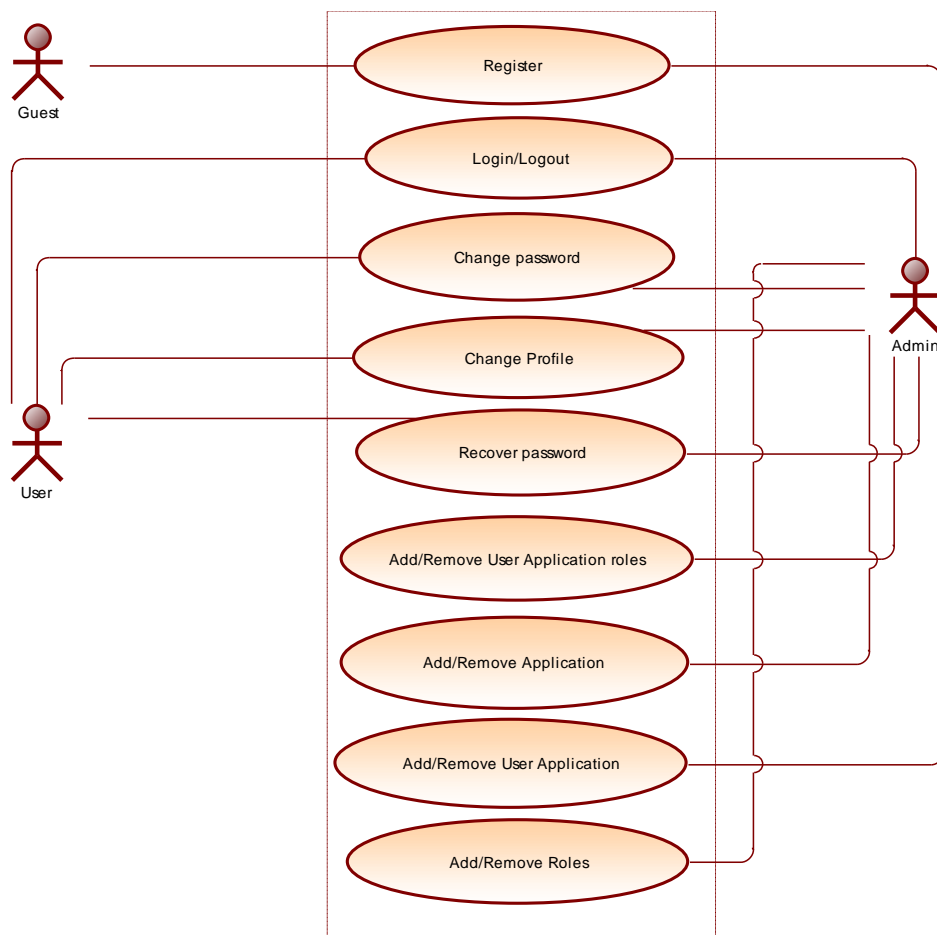
Xây dựng, thiết kế hệ thống quản lý tài khoản tập trung. Với một tài khoản duy nhất người dùng có thể sử dụng nhiều ứng dụng. Đồng thời giúp cho việc xây dựng, tích hợp các dịch vụ mới vào hệ thống một cách nhanh chóng, linh hoạt mà không phải quan tâm về việc quản lý tài khoản, hay phân quyền cho người dùng.

#### **b) Các định nghĩa, mô tả**

1. Một người dùng chưa có tài khoản khi vào ứng dụng thì phải tạo tài khoản
2. Người dùng khi tạo một tài khoản mới phải điền các thông tin cơ bản như: FirstName, LastName, Email, password. Hệ thống AMS sẽ xác định tài khoản được tạo cho ứng dụng nào và gửi vào mail cho người dùng đường dẫn (URL) để kích hoạt tài khoản. Sau khi kích hoạt thành công người dùng có thể truy cập được vào ứng dụng mà hệ thống cho phép.
3. Người dùng khi đã có tài khoản trên AMS, khi vào ứng dụng → đăng nhập → Hệ thống AMS xác định tài khoản và ứng dụng yêu cầu xác thực. Nếu có thì AMS sẽ gửi lại App một khóa (token) cho phép tài khoản truy cập vào ứng dụng → App sẽ kiểm tra Token. Nếu hợp lệ sẽ xác định các quyền của tài khoản thao tác trên ứng dụng đó. Không hợp lệ sẽ không cho phép truy cập vào ứng dụng.

4. Khi người dùng muốn thay đổi một số thông tin cá nhân. Người dùng sẽ đăng nhập → bổ sung và thay đổi một số thông tin.
5. Phân quyền cho người dùng trên AMS. Trên AMS tài khoản Admin sẽ có quyền quản lý các người dùng và phân quyền.
6. Phân quyền cho tài khoản người dùng trên các ứng dụng. Admin ứng dụng đăng nhập vào chức năng quản lý người dùng để: Tìm kiếm, đưa danh sách hoặc tài khoản người dùng vào ứng dụng, tạo nhóm, phân quyền nhóm, phân quyền, khóa tài khoản, loại bỏ tài khoản ra khỏi ứng dụng.

**c) Các yêu cầu nghiệp vụ**



*Hình 3- 9 Usecase của hệ thống*

**d) Nghiệp vụ, chức năng (Product Backlog)**

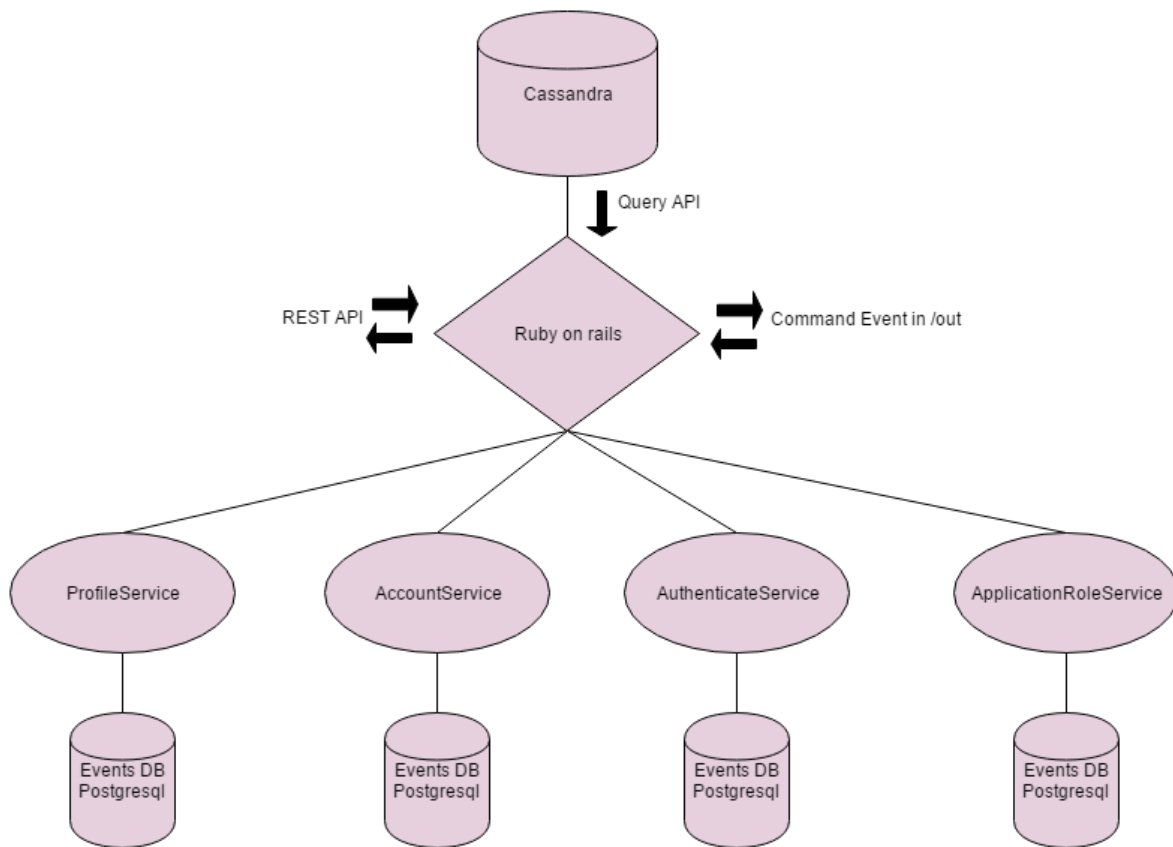
- Là khách thì có các chức năng (Guest functions):
  - o Có thể đăng ký tài khoản (Register). Khách khi vào ứng dụng nếu chưa có tài khoản thì có thể đăng ký. Khi đăng ký cần phải điền các thông tin



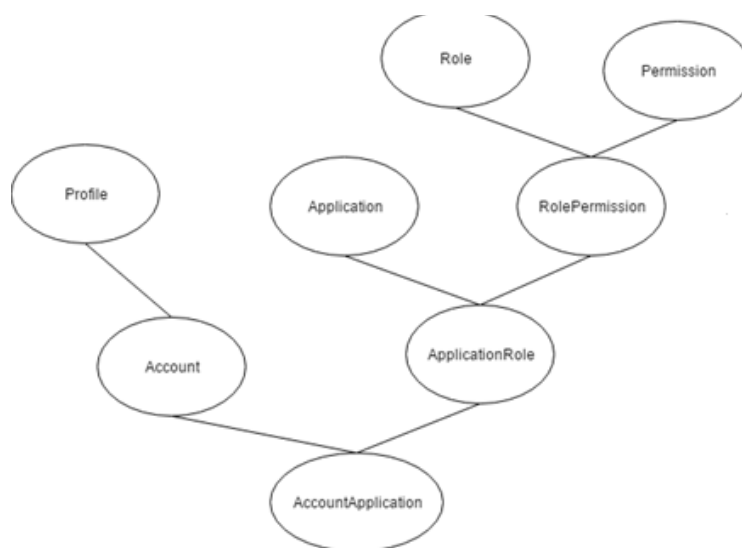
như: User\_name, First\_name, Last\_name, Email, Password. Nếu tài khoản đã tồn tại thì thông báo yêu cầu tạo tài khoản khác.

- Là người dùng thì có các chức năng (User functions)
  - Login/LogOut: Khi có tài khoản người dùng có thể login vào ứng dụng thông qua Username và password, hay có thể logout.
  - Thay đổi Profile (Change profile): User login đăng nhập vào ứng dụng thay đổi các thông tin cá nhân của mình như: FirstName, LastName, Email, Password.
  - Khôi phục mật khẩu (Recover password) khi bị mất: Khi người dùng quên mật khẩu có thể sử dụng chức năng này để lấy lại mật khẩu. Hệ thống sẽ gửi vào email đường link thay đổi mật khẩu.
- Là Admin thì có các chức năng (Admin function)
  - Tạo người dùng (Create User): Chỉ có Admin mới có thể tạo tài khoản người dùng. Các thông tin gồm: Username, Firstname, Lastname, Email, Password
  - Tìm kiếm người dùng: Tìm kiếm theo Username, email...
  - Thêm, xóa, sửa người dùng theo ứng dụng: Admin thêm, xóa, sửa User cho Application.
  - Gán, gỡ quyền cho người dùng: Admin có thể gán, gỡ quyền của User ứng dụng.
  - Thêm, xóa, sửa ứng dụng (application): Admin có thể khai báo thêm, xóa, sửa một ứng dụng
  - Thêm, xóa, sửa nhóm (role): Admin có thể khai báo thêm, xóa, sửa quyền
  - Khóa tài khoản người dùng: Admin có thể khóa, mở khóa tài khoản người dùng cho ứng dụng.
  - Theo dõi nhật ký sử dụng: Admin có thể theo dõi nhật ký sử dụng của người dùng.

e) Mô hình kiến trúc của hệ thống hướng Microservices



Hình 3- 10 Mô hình kiến trúc của hệ thống hướng Microservice



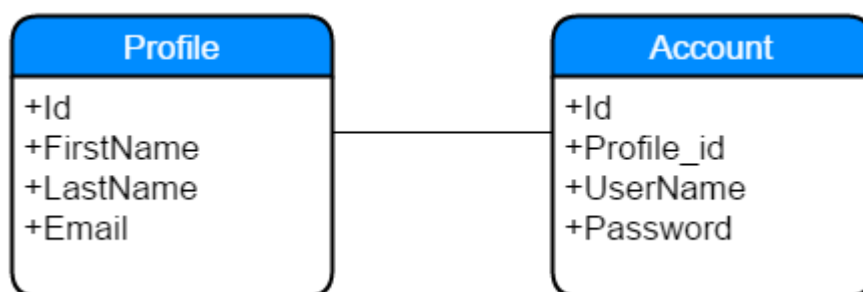
Hình 3- 11 Mô hình DDD

- **Ruby on rails:** Hệ thống được xây dựng dựa trên nền tảng công nghệ Ruby on rails. Ruby On rails là một Framework cho phép phát triển ứng dụng Web gồm 2 phần cơ bản:
  - o Phần ngôn ngữ Ruby: *“Ruby là một ngôn lập trình mã nguồn mở, linh hoạt, với một sự nổi bật về sự đơn giản dễ dùng và hữu ích. Nó có cú pháp dễ đọc và dễ dàng để viết”.*
  - o Phần Framework Rails bao gồm nhiều thư viện liên kết.
- **Cassandra:** Cassandra là một quản trị hệ cơ sở dữ liệu phân tán mã nguồn mở được thiết kế để xử lý một khối lượng lớn dữ liệu giàn trải trên nhiều node mà vẫn đảm bảo tính sẵn sàng cao (Highly Availability), khả năng mở rộng hay thu giảm số node linh hoạt (Elastic Scalability) và chấp nhận một số lỗi (Fault Tolerant). Nó được phát triển bởi Facebook và vẫn còn tiếp tục phát triển và sử dụng cho mạng xã hội lớn nhất thời giới này. Năm 2008, Facebook chuyển nó cho cộng đồng mã nguồn mở và được Apache tiếp tục phát triển đến ngày hôm nay. Cassandra được coi là sự kết hợp của Amazon’s Dynamo và Google’s BigTable.

#### f) ProfileService

##### Profile Rules:

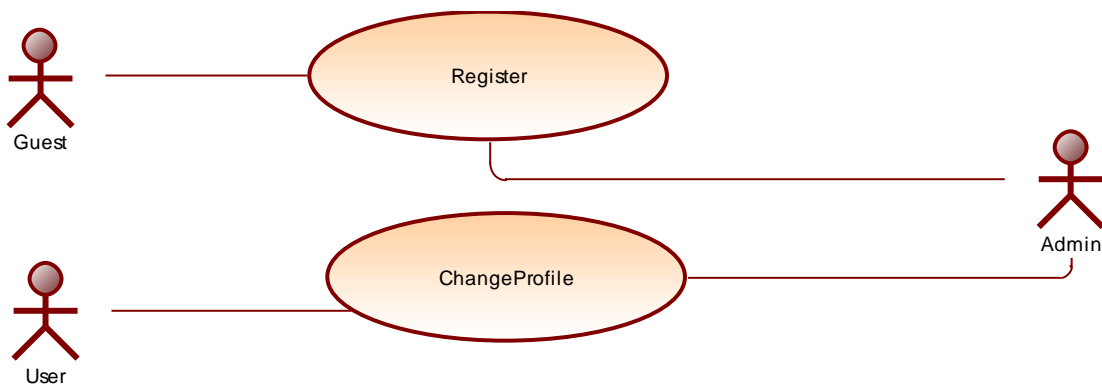
- + FirstName, LastName, Email không được để trống và độ dài không quá 256 ký tự
- + Email là duy nhất, độ dài không quá 256 kí tự.



Hình 3- 12 DDD của dịch vụ Profile

**Entity:** Profile

**Value Objects:** FirstName, LastName, Email



*Hình 3- 13Profile Usecase*

- Một Profile được đăng ký thông qua Guest, hoặc Admin.
- Một User được quyền sửa thông tin cá nhân của mình như thay đổi FirstName, LastName, Email...
- Admin có quyền chỉnh sửa thông tin cá nhân của User.

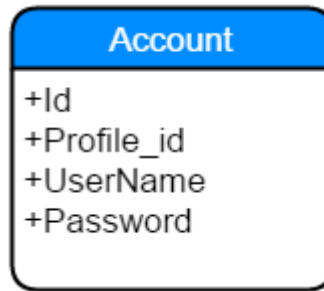
**Profile Command Events:**

- Đăng ký một tài khoản người dùng (RegisterAccount): Khi đăng ký một tài khoản sẽ xảy ra các Event sau:
  - o ProfileCreate:Guest/Admin({id:1,FirstName:'Do Van', LastName:'Hung',Email: 'hungdv@gmail.com'})
  - o AccountCreate:Guest/Admin({id,Profile\_id,UserName:'hungdv',password:'123654'})
- Thay đổi thông tin Profile (ChangeProfile): Khi một User/Admin muốn thay đổi thông tin cá nhân của một tài khoản. User chỉ được phép thay đổi thông tin của mình. Các Event xảy ra:
  - o ProfileUpdate:User/Admin({FirstName:'Nguyen Van', LastName: 'Hung', Email:hungdv@gmail.com})
- Xóa một Profile(Remove Profile). Admin có thể xóa một Profile người dùng:
 

Các Event xảy ra:

- ProfileRemove:Admin({id:1})
- AccountRemove:Admin({Profile\_id:1})
- AccountApplicationRoleRemove:Admin({Account\_id:})

**g) AccountService**



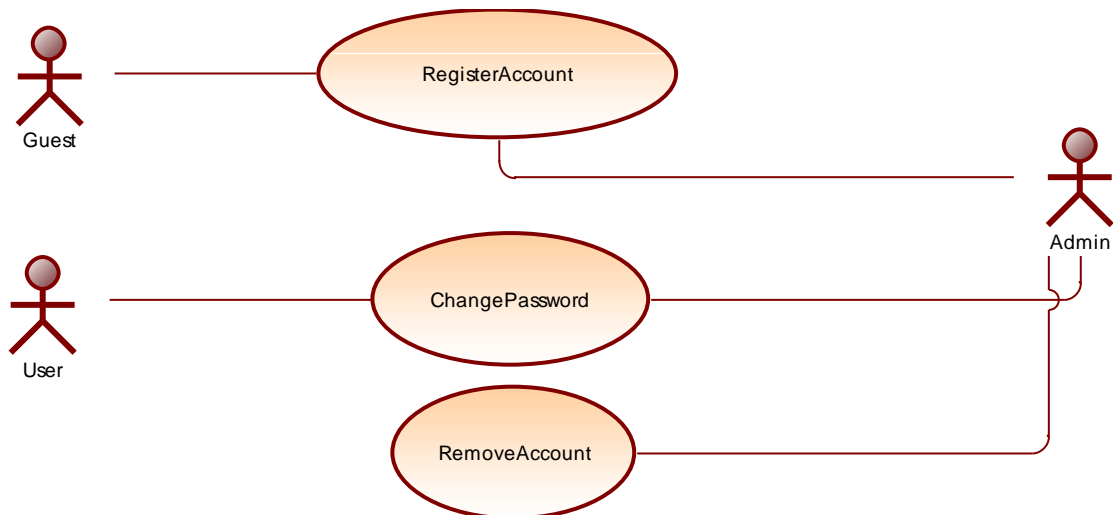
*Hình 3- 14DDD Account*

**Account Rules:**

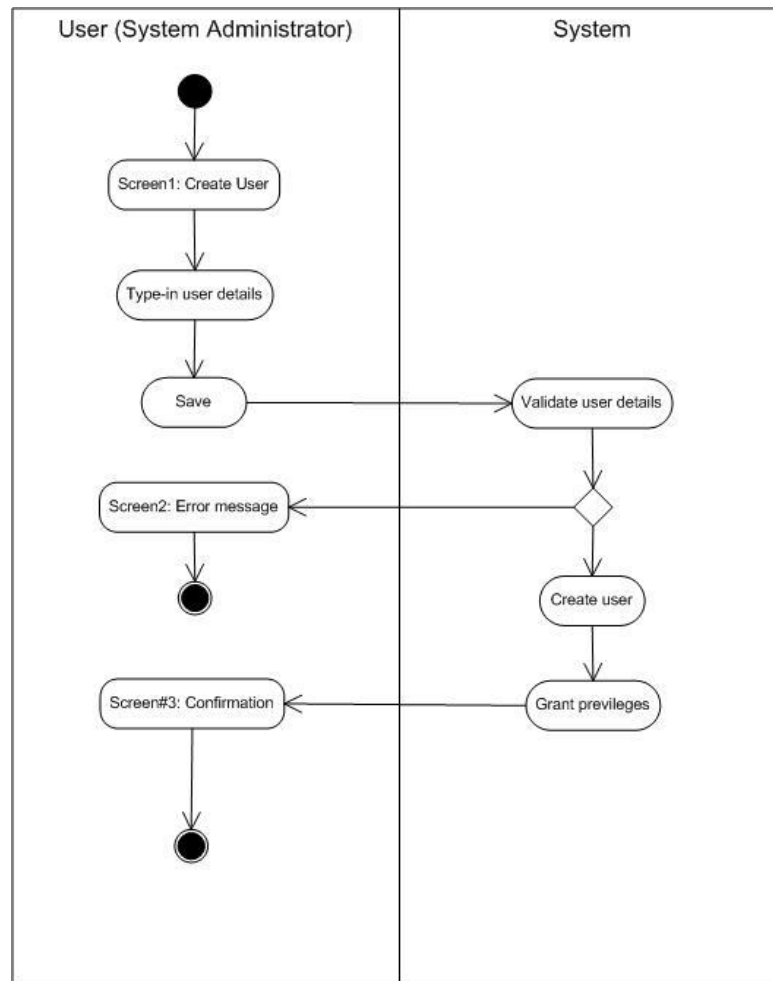
- + Một Username cần phải có một Profile.
- + UserName không được để trống và độ dài không quá 256 ký tự
- + Password không được phép trùng với Username, độ dài là 6 ký tự trở lên

**Entity:** Account

**Value Objects:** Password



*Hình 3- 15 Account Usecase*



Hình 3- 16 Tạo tài khoản người dùng

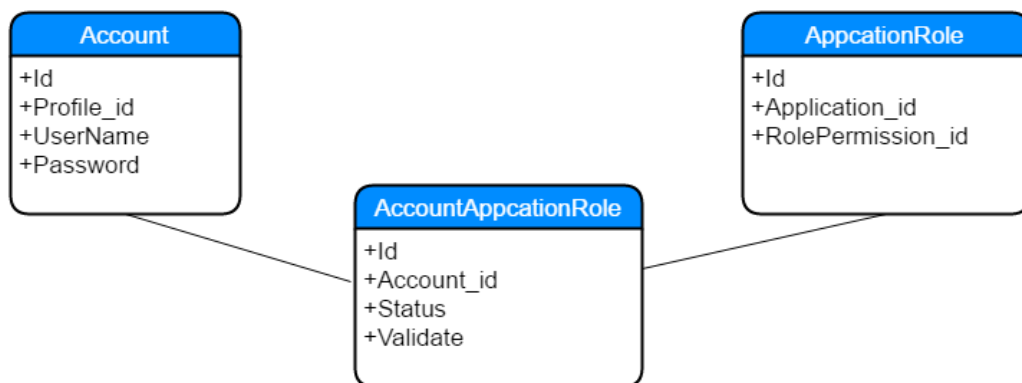
- Guest/Admin có thể đăng ký tài khoản người dùng.

#### Account Command Events:

- Đăng ký một tài khoản người dùng (RegisterAccount): Khi đăng ký một tài khoản sẽ xảy ra các Event sau:
  - o ProfileCreate:Guest/Admin({id:1,FirstName:'DoVan',LastName:'Hung', Email:'hungdv@gmail.com'})
  - o AccountCreate:Guest/Admin({id,Profile\_id,UserName:'hungdv',password:'123654'})
- Thay đổi mật khẩu người dùng(ChangePassword): User/Admin có quyền thay đổi mật khẩu. Các Event sau:
  - o AccountChangePassword:User/Admin({UserName:'hungdv', Password:New password })

- Xóa một tài khoản Account(Remove Account). Admin có thể xóa một tài khoản người dùng: Các Event xảy ra:
  - o ProfileRemove:Admin({id:1})
  - o AccountRemove:Admin({Profile\_id:1})
  - o AccountApplicationRole:Admin({ Account\_id:})

#### h) Authenticate Service

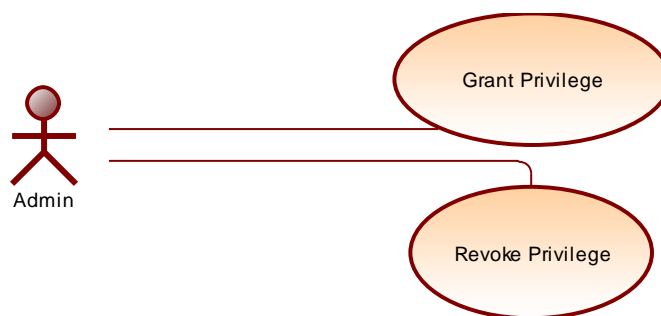


Hình 3- 17Mô hình DDD của dịch vụ Authenticate

#### Authenticate Rules:

- + Một tài khoản sẽ có nhiều quyền.
- + Mỗi quyền đều có Validate (Yes/No) được phép hay không.
- + Cần phải tồn tại người dùng và quyền của ứng

Value Objects: Status, Validate



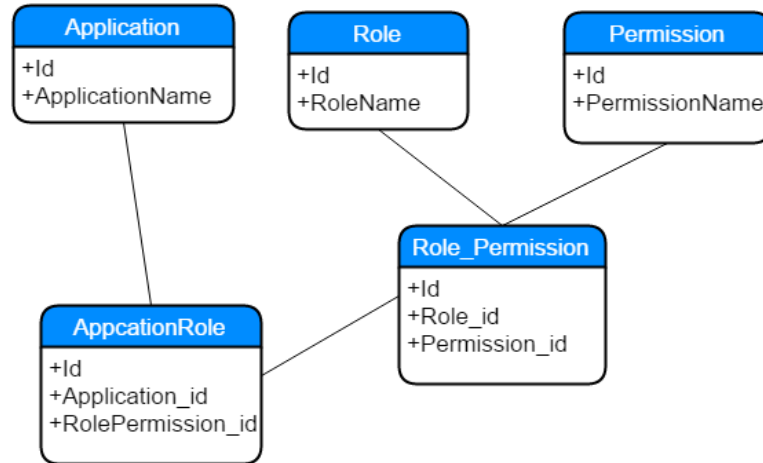
Hình 3- 18Authenticate Usecase

#### Authenticate Command Events:

- Khi Admin Thêm quyền cho người dùng (Gant privilege): Các Event xảy ra:
  - o AccountApplicationRoleGrantPrivilege:Admin({ Account\_id:1,AppliactionRole\_id:2})
- Khi Admin gỡ quyền của người dùng (Revoke privilege): Các Event xảy ra:

- AccountApplicationRoleRevokePrivilege:Admin({ Account\_id:1, ApplicationRole\_id:2})

**i) Application Role Service**



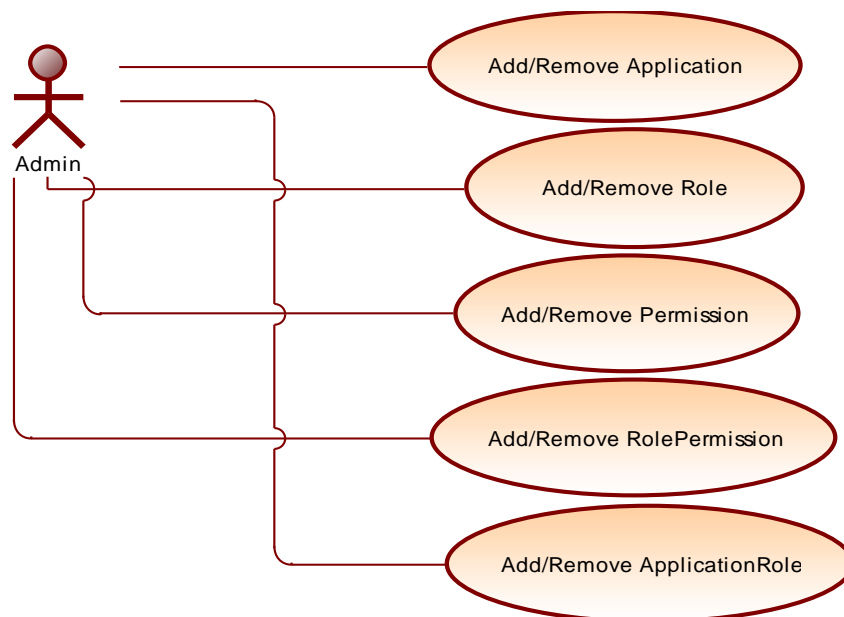
Hình 3- 19 Mô hình DDD của dịch vụ ApplicationRole

**Application Rules:**

+ Admin có quyền thực hiện các thao tác này

Entity: Application, Role, Permission

Value Objects: RolePermission, ApplicationRole



Hình 3- 20 Mô hình DDD của ApplicationRole

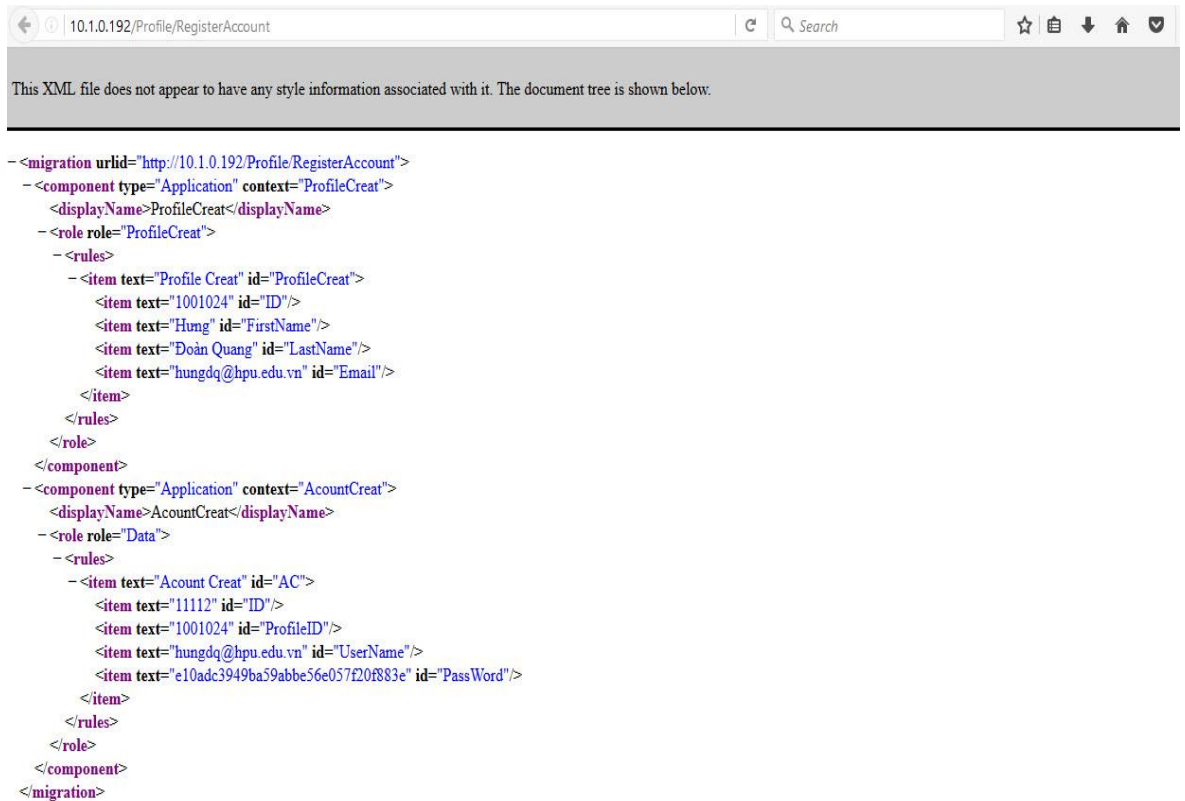


### **ApplicationRole Command Events:**

- Khi thêm một application (AddApplication). Các Event xảy ra:
  - o ApplicationCreate:Admin({id:1,Applicationname:'Account Management System'})
- Khi xóa một Application(RemoveApplication) các Event xảy ra:
  - o ApplicationRemove:Admin({id:1})
  - o ApplicationRoleRemove:Admin({Application\_id:1})
- Khi thêm một Role các Event xảy ra:
  - o RoleCreate:Admin({id:1, Rolename:'Student'})
- Xóa Role các Event xảy ra:
  - o RoleRemove:Admin({id:1})
  - o RolePermissionRemove:Admin({Role\_id:1})
- Thêm các quyền cho nhóm (AddRolePermission): Các Event:
  - o RolePermissionCreate:Admin({id:, Role\_id:, Permission\_id:})
- Xóa một nhóm quyền (RemoveRolePermission):Các Event xảy ra:
  - o RolePermissionRemove:Admin({id:})
- Thêm một nhóm quyền cho ứng dụng(AddApplicationRole): Các Event:
  - o ApplicationRoleCreate:Admin({id:,Application\_id:, RolePermission\_id:})
- Xóa một nhóm quyền của ứng dụng (RemoveApplicationRole):Các Event xảy ra:
  - o ApplicationRoleRemove:Admin({id:})

### 3.5. Cài đặt và đánh giá phần mềm thử nghiệm

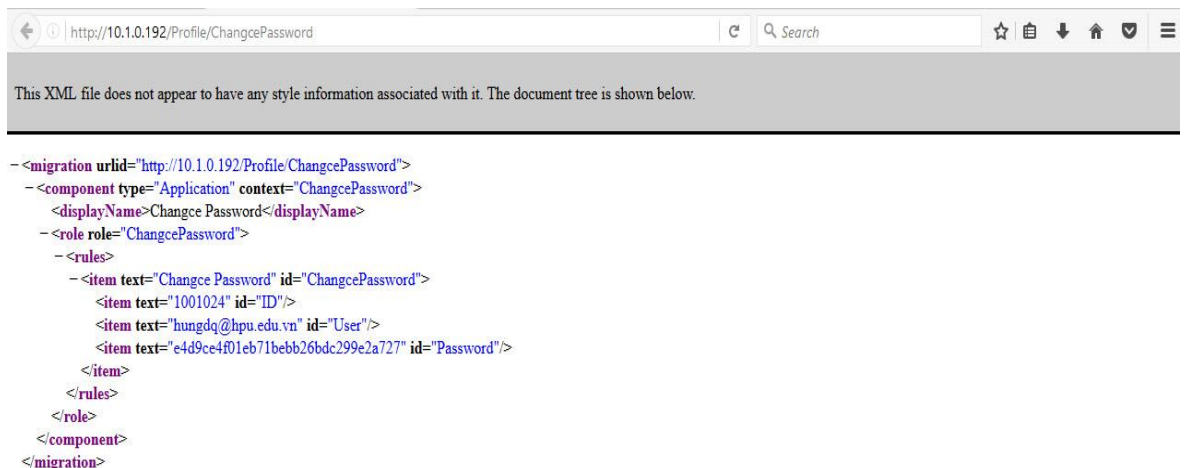
#### Command RegisterAccount.



```
-<migration urlid="http://10.1.0.192/Profile/RegisterAccount">
-<component type="Application" context="ProfileCreat">
  <displayName>ProfileCreat</displayName>
  -<role role="ProfileCreat">
    -<rules>
      -<item text="Profile Creat" id="ProfileCreat">
        <item text="1001024" id="ID"/>
        <item text="Hung" id="FirstName"/>
        <item text="Đoàn Quang" id="LastName"/>
        <item text="hungdq@hpu.edu.vn" id="Email"/>
      </item>
    </rules>
  </role>
</component>
-<component type="Application" context="AccountCreat">
  <displayName>AccountCreat</displayName>
  -<role role="Data">
    -<rules>
      -<item text="Account Creat" id="AC">
        <item text="11112" id="ID"/>
        <item text="1001024" id="ProfileID"/>
        <item text="hungdq@hpu.edu.vn" id="UserName"/>
        <item text="e10adc3949ba59abbe56e057f20f883e" id="PassWord"/>
      </item>
    </rules>
  </role>
</component>
</migration>
```

Hình 3- 21 Register Account

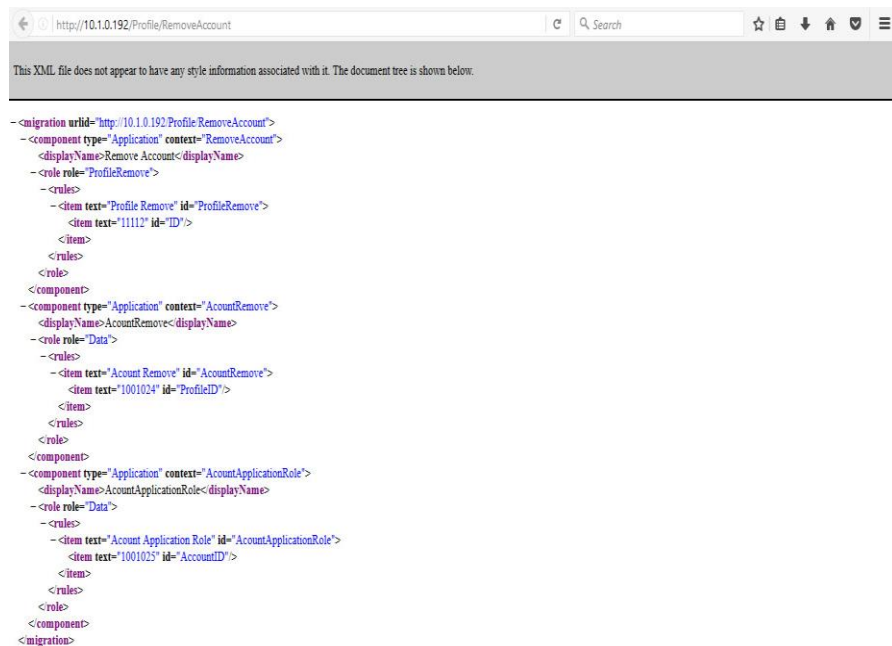
#### Command change password



```
-<migration urlid="http://10.1.0.192/Profile/ChangePassword">
-<component type="Application" context="ChangePassword">
  <displayName>Change Password</displayName>
  -<role role="ChangePassword">
    -<rules>
      -<item text="Change Password" id="ChangePassword">
        <item text="1001024" id="ID"/>
        <item text="hungdq@hpu.edu.vn" id="User"/>
        <item text="e4d9ce4f01eb71bebb26bdc299e2a727" id="Password"/>
      </item>
    </rules>
  </role>
</component>
</migration>
```

Hình 3- 22 Change password

#### Command ProfileRemove



The screenshot shows a web browser window with the address bar containing the URL `http://10.1.0.192/Profile/RemoveAccount`. Below the address bar, a message states: "This XML file does not appear to have any style information associated with it. The document tree is shown below:". The XML document tree is displayed as follows:

```
<-migration urlId="http://10.1.0.192/Profile/RemoveAccount">
  <-component type="Application" context="RemoveAccount">
    <displayName>Remove Account</displayName>
    <-rule role="ProfileRemove">
      <-rules>
        <-item text="Profile Remove" id="ProfileRemove">
          <item text="11112" id="ID"/>
        </item>
      </rules>
    </rule>
  </component>
  <-component type="Application" context="AccountRemove">
    <displayName>AccountRemove</displayName>
    <-rule role="Data">
      <-rules>
        <-item text="Account Remove" id="AccountRemove">
          <item text="1001024" id="ProfileID"/>
        </item>
      </rules>
    </rule>
  </component>
  <-component type="Application" context="AccountApplicationRole">
    <displayName>AccountApplicationRole</displayName>
    <-rule role="Data">
      <-rules>
        <-item text="Account Application Role" id="AccountApplicationRole">
          <item text="1001025" id="AccountID"/>
        </item>
      </rules>
    </rule>
  </component>
</migration>
```

*Hình 3- 23 Xóa một Profile*

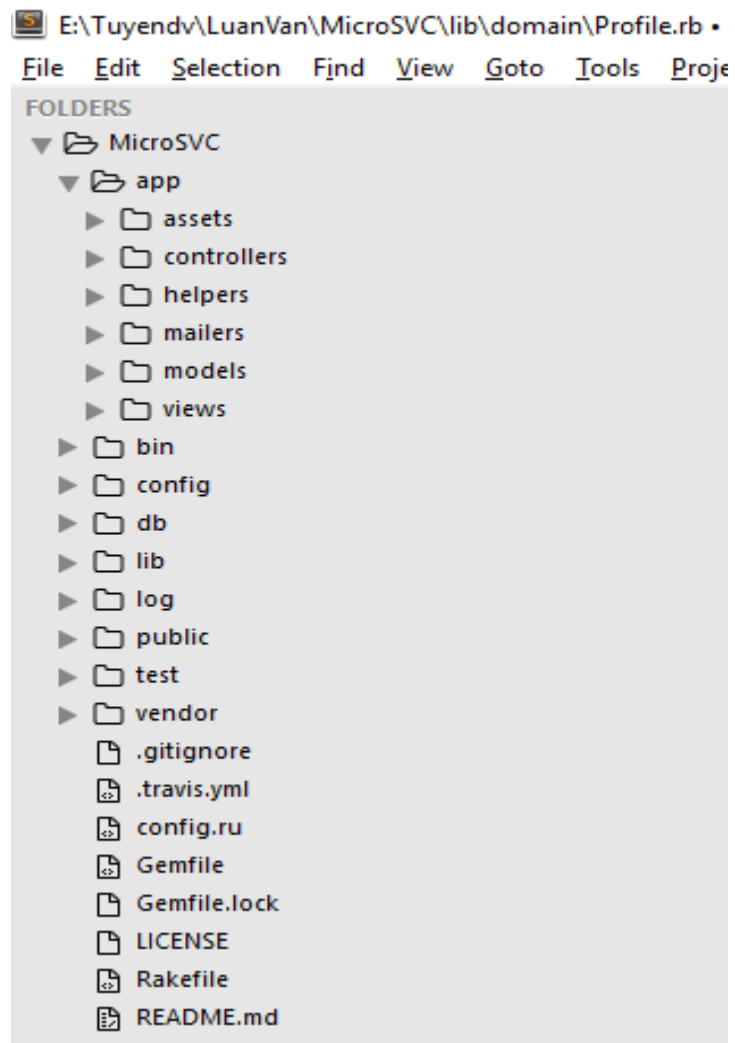
## Danh sách các Events

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
-<migration urlid="http://10.1.0.192/Profile/Event">
  -<component type="Application" context="Event">
    <displayName>Event</displayName>
    -<role role="Event">
      -<rules>
        -<item text="Event" id="ProfileCreat">
          <item text="1" id="ID"/>
          <item text="ProfileCreat" id="EventName"/>
          <item text="{1001024:Hung:Đoàn Quang:hungdq@hpu.edu.vn}" id="Data"/>
          <item text="Profile Creat" id="MetaData"/>
          <item text="2016-12-13 17:45:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="AccountCreat" id="AccountCreat">
          <item text="2" id="ID"/>
          <item text="AccountCreat" id="EventName"/>
          <item text="{11112:1001024:hungdq@hpu.edu.vn:'e10adc3949ba59abbe56e057f20f883e'}" id="Data"/>
          <item text="Account Creat" id="MetaData"/>
          <item text="2016-12-13 09:43:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="Event" id="AccountChangePassword">
          <item text="3" id="ID"/>
        </component>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="Event" id="AccountChangePassword">
          <item text="3" id="ID"/>
          <item text="AccountChangePassword" id="EventName"/>
          <item text="{1001024:hungdq@hpu.edu.vn:'e4d9ce4f01eb71bebb26bdc299e2a727'}" id="Data"/>
          <item text="Change Password" id="MetaData"/>
          <item text="2016-12-13 10:43:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="Event" id="ProfileRemove">
          <item text="4" id="ID"/>
          <item text="ProfileRemove" id="EventName"/>
          <item text="{1001024}" id="Data"/>
          <item text="Profile Remove" id="MetaData"/>
          <item text="2016-12-13 15:43:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="Event" id="AccountRemove">
          <item text="5" id="ID"/>
          <item text="AccountRemove" id="EventName"/>
          <item text="{1001024;}" id="Data"/>
          <item text="Account Remove" id="MetaData"/>
          <item text="2016-12-13 17:43:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
  -<component type="Application" context="Event">
    -<role role="Event">
      -<rules>
        -<item text="Event" id="AccountApplicationRole">
          <item text="5" id="ID"/>
          <item text="AccountApplicationRole" id="EventName"/>
          <item text="{1001024;}" id="Data"/>
          <item text="Account Application Role" id="MetaData"/>
          <item text="2016-12-13 17:43:49" id="CreatedAt"/>
        </item>
      </rules>
    </role>
  </component>
</migration>
```

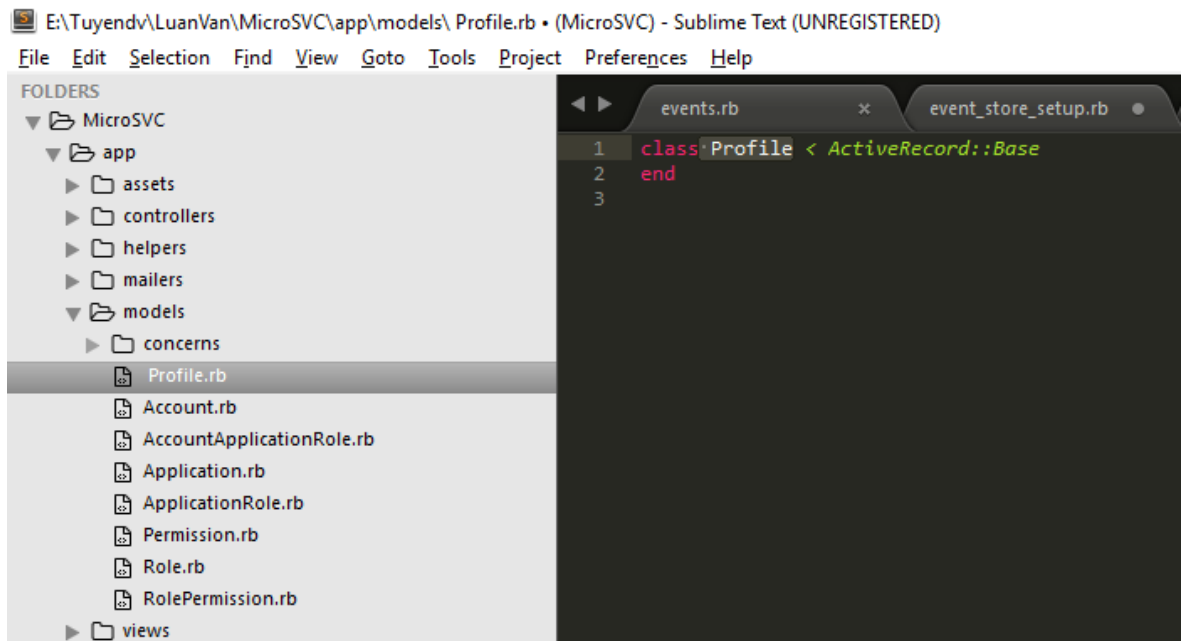
Hình 3- 24 Các Events

Cấu trúc thư mục:



Hình 3- 25 Cấu trúc thư mục code chương trình

## Models:



Hình 3- 26 Danh sách các Model

## Domain:

### + Profile.rb

```
module Domain
  class Profile
    include AggregateRoot
    AlreadySubmitted = Class.new(StandardError)
    ProfileExpired = Class.new(StandardError)
    def submit(Id, FirstName, LastName, UserName, LastName, Email, Password)
      raise AlreadySubmitted if state == :submitted
      raise ProfileExpired if state == :expired
      apply Events::ProfileSubmitted.new(data: {id: id, FirstName: FirstName, LastName: LastName,
      Email: Email, Password: Password})
    end
    def expire
      raise AlreadySubmitted unless state == :draft
      apply Events::ProfileExpired.new(data: {id: id})
    end
    def apply_Profile_submitted(event)
      @id = event.data[:id]
      @Firstname = event.data[:Firstname]
      @Lastname = event.data[:Lastname]
      @Email = event.data[:Email]
      @state = :submitted
    end
    def apply_Profile_expired(event)
      @state = :expired
    end
  end
end
```

```
end
```

#### + Events.rb

```
module Events
  ProfileCreate = Class.new(RailsEventStore::Event)
  AccountCreate = Class.new(RailsEventStore::Event)
  ProfileUpdate = Class.new(RailsEventStore::Event)
  ProfileRemove = Class.new(RailsEventStore::Event)
  AccountRemove = Class.new(RailsEventStore::Event)
  AccountChangePassword = Class.new(RailsEventStore::Event)
  AccountApplicationRoleRemove = Class.new(RailsEventStore::Event)
  AccountApplicationRoleGrantPrivilege = Class.new(RailsEventStore::Event)
  AccountApplicationRoleRevokePrivilege = Class.new(RailsEventStore::Event)
  ApplicationCreate = Class.new(RailsEventStore::Event)
  ApplicationRemove = Class.new(RailsEventStore::Event)
  ApplicationRoleRemove = Class.new(RailsEventStore::Event)
  RoleCreate = Class.new(RailsEventStore::Event)
  RoleRemove = Class.new(RailsEventStore::Event)
  RolePermissionCreate = Class.new(RailsEventStore::Event)
  RolePermissionRemove = Class.new(RailsEventStore::Event)
  ApplicationRoleCreate = Class.new(RailsEventStore::Event)
  ApplicationRoleRemove = Class.new(RailsEventStore::Event)
end
```

#### + Event\_store\_setup.rb

```
module EventStoreSetup
  def event_store
    @event_store ||= RailsEventStore::Client.new.tap do |es|
      es.subscribe(Denormalizers::ProfileCreate.new, [Events::ProfileCreate])
      es.subscribe(Denormalizers::AccountCreate.new, [Events::AccountCreate])
      es.subscribe(Denormalizers::ProfileUpdate.new, [Events::ProfileUpdate])
      es.subscribe(Denormalizers::ProfileRemove.new, [Events::ProfileRemove])
      es.subscribe(Denormalizers::AccountRemove.new, [Events::AccountRemove])
      es.subscribe(Denormalizers::AccountChangePassword.new,
        [Events::AccountChangePassword])
    end
  end
end
```

```
    es.subscribe(Denormalizers::AccountApplicationRoleRemove.new,  
[Events::AccountApplicationRoleRemove])  
    es.subscribe(Denormalizers::AccountApplicationRoleGrantPrivilege.new,  
[Events::AccountApplicationRoleGrantPrivilege])  
    es.subscribe(Denormalizers::AccountApplicationRoleRevokePrivilege.new,  
[Events::AccountApplicationRoleRevokePrivilege])  
    es.subscribe(Denormalizers::ApplicationCreate.new, [Events::ApplicationCreate])  
    es.subscribe(Denormalizers::ApplicationRemove.new, [Events::ApplicationRemove])  
    es.subscribe(Denormalizers::RoleCreate.new, [Events::RoleCreate])  
    es.subscribe(Denormalizers::RoleRemove.new, [Events::RoleRemove])  
    es.subscribe(Denormalizers::RolePermissionCreate.new, [Events::RolePermissionCreate])  
    es.subscribe(Denormalizers::RolePermissionRemove.new, [Events::RolePermissionRemove])  
    es.subscribe(Denormalizers::ApplicationRoleCreate.new, [Events::ApplicationRoleCreate])  
    es.subscribe(Denormalizers::ApplicationRoleRemove.new, [Events::ApplicationRoleRemove])  
end  
end
```



## **Đánh giá và kết luận**

Chiến lược phân tích thiết kế hệ thống phần mềm theo lĩnh vực là một hướng đi mới và gần đây được ứng dụng rộng rãi. Thông qua luận văn tôi muốn tập trung nghiên cứu, xây dựng và phát triển một hệ thống quản lý tài khoản tập trung AMS sử dụng phân tích theo lĩnh vực DDD, cùng với việc chia hệ thống thành các nhiều các dịch vụ theo hướng Microservice đảm bảo cho khả năng phát triển mở rộng lâu dài. Một số điểm chính luận văn đã đạt được cụ thể như sau:

- Nghiên cứu tổng quan các chiến lược phân tích thiết kế phần mềm hiện nay.
- Đề xuất về chiến lược thiết kế hướng lĩnh vực và một số quy trình phát triển phần mềm theo hướng lĩnh vực.
- Tìm hiểu mô hình phần mềm hướng dịch vụ Microservice, nền tảng framework Ruby on Rails.
- Áp dụng vào bài toán quản lý tài khoản tập trung của trường ĐH Dân lập Hải Phòng.
- Xây dựng phần mềm AMS trên nền Ruby on Rails

Một số hướng nghiên cứu tiếp theo có thể phát triển là:

- Phát triển tiếp các dịch vụ sử dụng AMS
- Hoàn thiện việc phân tích theo hướng lĩnh vực cho các dịch vụ quản lý trường học.

## TÀI LIỆU THAM KHẢO

### Tiếng Anh:

- [1] Eric Evans (2003), “ *Domain-Driven Design Tackling Complexity in the Heart of Software*”, Addison-Wesley; ISBN: 0-321-12521-5
- [2] Vaughn Vernon (2013), “*Implementing Domain-Driven Design*”, Addison-Wesley
- [3] Sam Ruby, Dave Thomas, David Heinemeier Hansson (2011), “*Agile Web Development with Rails*”, United States of America

### Tiếng Việt:

- [4] Lê Văn Phùng (2014), “*Kỹ nghệ phần mềm*”, tái bản lần 1 NXB Thông tin và truyền thông, Hà Nội.
- [5] Lê Văn Phùng, Lê Hương Giang (2013), “*Kỹ nghệ phần mềm nâng cao*”, NXB Thông tin và truyền thông, Hà Nội.
- [6] Lê Văn Phùng (2014), “*Các mô hình cơ bản trong phân tích và thiết kế hướng đối tượng*”, NXB Thông tin và truyền thông, Hà Nội.
- [7] Nguyễn Văn Vy, Nguyễn Việt Hà (2009), “*Giáo trình kỹ nghệ phần mềm*”, NXB Giáo dục Việt Nam

### Internet:

- [8] <https://viblo.asia/nghiadd/posts/mrDGMOExkzL>
- [9] [https://vi.wikipedia.org/wiki/Domain\\_driven\\_design](https://vi.wikipedia.org/wiki/Domain_driven_design)
- [10] <https://www.railstutorial.org/book>
- [11] [http://www.vi.w3eacademy.com/cassandra/cassandra\\_introduction.htm](http://www.vi.w3eacademy.com/cassandra/cassandra_introduction.htm)
- [12] <https://viblo.asia/nguyen.thi.phuong.mai/posts/15XRBVZeRqPe>
- [13] <https://techmaster.vn/posts/33594/gioi-thieu-ve-microservices>