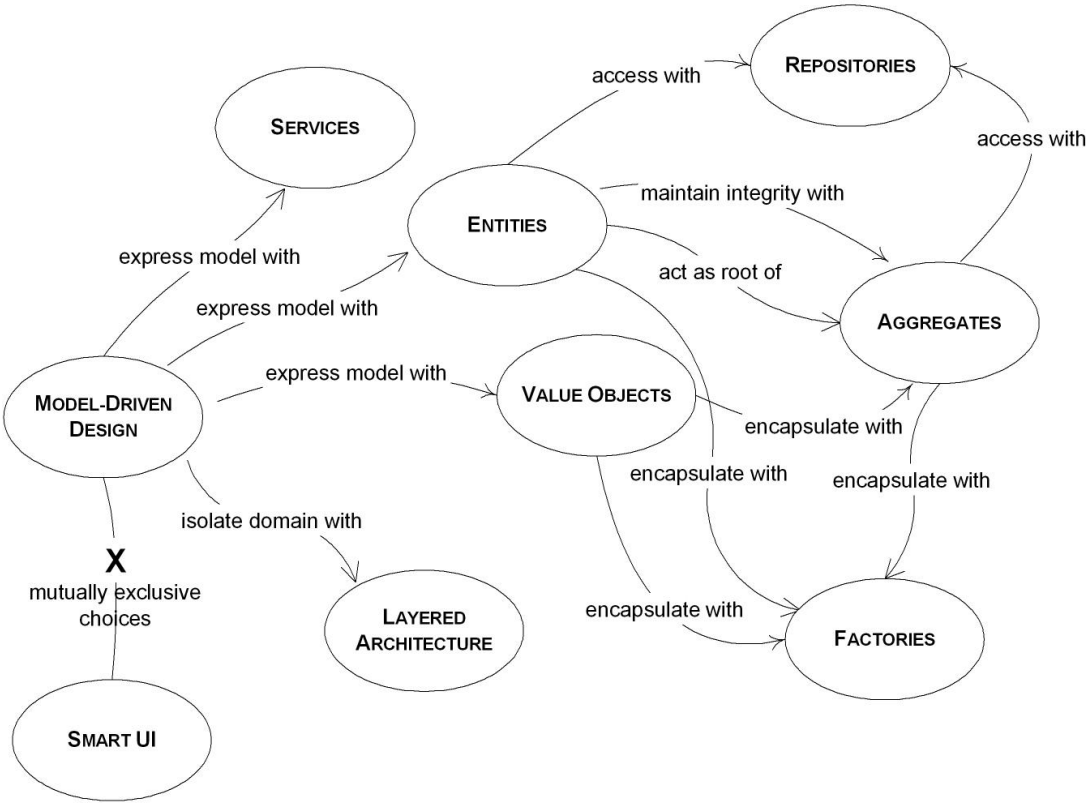
Domain-Driven Design *Quickly*

(Phiên bản tiếng Việt)



by Abel Avram & Floyd Marinescu

edited by: Dan Bergh Johnsson, Vladimir Gitlevich

InfoQ *Enterprise Software Development Series*

Domain-Driven Design

Quickly

(Phiên bản tiếng Việt)

C4Media, Publisher of InfoQ.com Enterprise Software Development Community

Part of the InfoQ Enterprise Software Development series of books.

For information or ordering of this or other InfoQ books, please contact
books@c4media.com.

Production Credits:
DDD Summary by: Abel Avram
Managing Editor: Floyd Marinescu
Cover art: Gene Steffanson
Composition: Laura Brown and Melissa Tessier
Special thanks to Eric Evans.

Library of Congress Cataloging-in-Publication Data:

ISBN: 978-1-4116-0925-9

Printed in the United States of America

10 9 8 7 6 5 3 2 1

MỤC LỤC

Lời tựa: Vì sao cần "DDD Quickly"?

Lần đầu tiên tôi nghe nói về Domain Driven Design (Thiết kế hướng lĩnh vực) và gặp Eric Evans tại một cuộc gặp nhỏ bởi những nhà thiết kế được tổ chức bởi Bruce Eckel vào mùa hè năm 2005. Rất nhiều người tôi kính phục cũng tham gia hội thảo này, bao gồm: Martin Fowler, Rod Johnson, Cameron Purdy, Randy Stafford và Gregor Hohpe.

Cả nhóm này khá ấn tượng với tầm nhìn của Domain Driven Design và đều hứng thú muốn tìm hiểu thêm về nó. Tôi cũng có cảm tưởng rằng mọi người đều muốn những khái niệm (về DDD) này trở thành chủ đạo. Tôi để ý cách Eric dùng mô hình lĩnh vực để thảo luận giải pháp cho một số thách thức kỹ thuật mà nhóm đã thảo luận, cách Eric nhấn mạnh lĩnh vực nghiệp vụ chứ không phải những chủ đề đặc thù kỹ thuật. Qua đó, tôi hiểu rằng tầm nhìn của Eric chính là cái mà cộng đồng cần.

Cộng đồng phát triển doanh nghiệp của chúng tôi, đặc biệt là cộng đồng phát triển web, bị hư hỏng bởi sự thổi phồng dẫn tới sự mất định hướng phát triển phần mềm hướng đối tượng đúng đắn. Trong cộng đồng Java, việc mô hình hóa lĩnh vực tốt bị lạc hướng bởi sự thổi phòng của EJB và mô hình container/component vào những năng 1999 - 2004. May mắn thay, sự dịch chuyển công nghệ và những kinh nghiệm chung của cộng đồng phát triển phần mềm đã hướng chúng tôi quay về với thực trạng hướng đối tượng truyền thống. Tuy nhiên, cộng động vẫn thiếu một tầm nhìn rõ ràng về cách áp dụng hướng đối tượng ở mức độ doanh nghiệp. Đây là lý do tại sao tôi nghĩ DDD trở nên quan trọng.

Điều không may mắn là, ngoài nhóm nhỏ gồm những nhà kiến trúc giàu kinh nghiệm, tôi nhận ra rằng chỉ một số rất ít biết tới DDD và đây chính là lý do InfoQ được ủy quyền viết cuốn sách này. Kỳ vọng của tôi là xuất bản một cuốn tổng hợp và giới thiệu, ngắn, đọc nhanh đối với nền tảng của DDD và cho phép tải về không hạn chế trên InfoQ, trong một định dạng nhỏ, in bỏ túi. Nếu làm được điều đó, tầm nhìn của chúng ta sẽ trở thành chính thống.

Cuốn sách này không giới thiệu bất kỳ khái niệm mới nào mà chỉ cố gắng tổng kết, chắt lọc những điều cơ bản như DDD là gì, viết lại hầu hết những chủ đề trong sách gốc của Eric Evans cũng như các nguồn khác đã được xuất bản như *Applying DDD* của Jimmy Nilsson và nhiều trao đổi về DDD trên các diễn đàn. Cuốn sách này đem tới cho bạn giáo trình *"vỡ lòng* về DDD cơ bản. Nó không nhắm tới mục đích thay thế rất nhiều ví dụ và trường hợp có trong cuốn sách của Eric hay những ví dụ thực hành có trong cuốn sách của Jimmy. Tôi đặc biệt khuyến nghị bạn đọc cả 2 cuốn sách tuyệt vời đó. Ngoài ra, nếu bạn đồng ý rằng cộng đồng cần DDD như là một phần của sự đồng thuận nhóm, hãy truyền bá cuốn sách này và tác phẩm của Eric thật rộng rãi..

Floyd Marinescu

Co-founder & Chief Editor của InfoQ.com

Lời nói đầu

Phần mềm là một dụng cụ được tạo ra để giúp chúng ta xử lý sự phức tạp trong cuộc sống hiện đại. Phần mềm thường được hiểu như là một điểm cuối, và thường là một điểm cuối rất thực tế và có thật. Ví dụ, chúng ta dùng phần mềm điều khiển không lưu và phần mềm này liên quan trực tiếp tới thế giới xung quanh ta. Chúng ta muốn bay từ một điểm tới một điểm khác bằng máy móc phức tạp, do đó chúng ta tạo phần mềm điều hành lịch trình bay của hàng ngàn máy bay vận hành liên tục, bất kỳ lúc nào trong không trung.

Phần mềm phải là thực tế và hữu dụng; nếu không chúng ta đã không đầu tư thời gian và nguồn lực để tạo ra nó. Điều này làm cho việc phát triển phần mềm liên kết vô cùng chặt chẽ với một số khía cạnh cụ thể trong cuộc sống của chúng ta. Một gói phần mềm không thể tách rời trong thực thể toàn bộ, là *lĩnh vực* mà nó làm ra để hỗ trợ chúng ta quản lý. Không những thế, phần mềm gắn chặt vào *lĩnh vực* đó.

Thiết kế phần mềm là một nghệ thuật. Giống như bất kỳ loại hình nghệ thuật nào khác, nó không thể dạy và học như một môn khoa học chính xác theo cách dạy định lý và công thức. Chúng ta có thể khám phá các nguyên tắc và kỹ thuật hữu dụng để áp dụng trong suốt quá trình tạo ra phần mềm, nhưng có lẽ chúng ta không thể vẽ ra một lộ trình chính xác và nếu theo đúng lộ trình đó chúng ta sẽ tạo ra một module mã nguồn thỏa mãn nhu cầu. Cũng giống như một bức tranh hay một tòa nhà, sản phẩm phần mềm hòa quyện cùng với người thiết kế và phát triển nó và sự quyến rũ, tinh tế (hoặc thiếu sự tinh tế) của những người đóng góp cho việc tạo ra phần mềm.

Có nhiều cách tiếp cận thiết kế phần mềm. Trong 20 năm gần đây, ngành công nghiệp phần mềm đã biết và dùng một số phương pháp để tạo ra sản phẩm. Mỗi cách tiếp cận đó đều có ưu thế và nhược điểm riêng. Mục đích của cuốn sách này là tập trung vào phương pháp thiết kế đã phát triển và tiến hóa trong hai thập kỉ vừa qua nhưng tập trung hơn vào các phương pháp tinh túy hơn trong vài năm gần đây: Thiết kế hướng lĩnh vực. Eric Evans là người cống hiến lớn cho chủ đề này bằng việc viết ra một cuốn sách tích tụ kiến thức về thiết kế hướng lĩnh vực. Chi tiết hơn về những chủ đề trình bày ở đây, chúng tôi khuyến nghị độc giả đọc các cuốn sách sau.

"Domain-Driven Design: Tackling Complexity in the Heart of Software", nhà xuất bản Addison-Wesley, ISBN: 0-321-12521-5.

Nhiều chủ đề sâu hơn được trao đổi trong nhóm

http://groups.yahoo.com/group/domaindrivendesign

Cuốn sách này chỉ là phần giới thiệu các điều cụ thể đó với mục đích giúp bạn hiểu nhanh kiến thức cơ sở chứ không hiểu chi tiết về Thiết kế hướng lĩnh vực. Chúng tôi muốn bạn thưởng thức một phong vị thiết kế phần mềm ngon miệng với những nguyên tắc và hướng dẫn được dùng trong giới thiết kế hướng lĩnh vực.

1

Domain-Driven Design là gì?

Phát triển phần mềm là qui trình tự động hóa được áp dụng thường xuyên nhất trên thế giới, nó cũng là sự cung cấp giải pháp cho những bài toán nghiệp vụ thực tế; Qui trình nghiệp vụ được tự động hóa hoặc những bài toán thực tế mà ở đó phần mềm là một lĩnh vực của nghiệp vụ. Chúng ta phải hiểu từ đầu rằng phần mềm bắt nguồn và liên quan rất sâu tới domain này.

Phần mềm được làm từ mã nguồn. Chúng ta bị cám dỗ cho việc dành rất nhiều thời gian với mã nguồn và nhìn phần mềm như các đối tượng và method đơn giản.

Hãy cùng xem xét ví dụ sản xuất xe ô tô. Những người công nhân liên quan đến việc sản xuất ô tô có thể chuyên biệt hóa việc sản xuất linh kiện ô tô, nhưng khi làm vậy, họ có góc nhìn hạn chế về qui trình sản xuất cả chiếc xe ô tô. Họ coi ô tô là một tập hợp khổng lồ những linh kiện và phải ráp chúng với nhau; thực ra ô tô phức tạp hơn thế nhiều. Một chiếc xe tốt bắt đầu từ một tầm nhìn. Họ viết đặc tả thật cẩn thận. Tiếp theo là thiết kế. Rất, rất nhiều thiết kế. Sau nhiều tháng, có thể thậm chí vài năm, họ thiết kế, thay đổi, cải tiến cho tới khi thiết kế trở nên hoàn hảo, cho tới khi nó thỏa mãn tầm nhìn đặt ra ban đầu. Qui trình thiết kế có thể không luôn làm trên giấy. Nhiều phần thiết kế bao gồm việc mô hình hóa, kiểm thử dưới điều kiện cụ thể để xem xe ô tô hoạt động hay không. Sau đó, bản thiết kế thay đổi theo kết quả kiểm thử. Cuối cùng, chiếc xe được đưa vào sản xuất, họ sản xuất linh kiện và ráp chúng vào với nhau.

Việc phát triển phần mềm cũng tương tự như vậy. Chúng ta không thể ngồi yên đó và gõ code. Chúng ta có thể làm vậy nhưng nếu làm vậy chỉ hiệu quả cho những trường hợp rất đơn giản. Chúng ta không thể tạo ra phần mềm phức tạp theo cách đó.

Để tạo ta một phần mềm tốt, bạn cần hiểu về phần mềm đó. Bạn không thể làm ra hệ thống phần mềm ngân hàng nếu trừ khi bạn có hiểu biết tương đối tốt về mảng ngân hàng và những điều liên quan. Nghĩa là, để làm phần mềm tốt, bạn 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. *Đồng chí* 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 cấu phần. Lập trình viên? Quên chuyện đó đi. 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 chúng ta thường xuất phát: Lĩnh vực (nghiệp vụ).

Khi chúng ta bắt đầu một dự án phần mềm, ta nên tập trung vào domain và hoạt động trong nó. Mục đích của cả phần mềm là để đề cao một domain cụ thể. Để làm được điều đó, phần mềm cần hài hòa với domain mà nó tạo nên. Nếu không nó sẽ gây nên sự căng thẳng với domain, dẫn tới hoạt động sai, phá hoại hay thậm chí rối loạn không kiểm soát được.

Vậy làm thế nào để làm ra phần mềm hài hòa với domain? Cách tốt nhất để làm việc đó là hãy làm cho phần mềm *phản chiếu* tới domain. Phần mềm cần hợp nhất với những khái niệm và thành phần cốt lõi của domain, và hiện thực hóa một cách chính xác quan hệ giữa chúng. Phần mềm phải mô hình hóa được domain.

Một số người không có kiến thức về banking có thể học bằng cách đọc thật nhiều mã nguồn về mô hình domain. Điều này là cần thiết. Phần mềm không bắt nguồn sâu từ gốc của domain sẽ phải thay đổi nhiều theo thời gian.

Rồi, chúng ta bắt đầu từ domain. Tiếp theo là gì? Một domain là cái gì đó có thật trên thế giới này. Nó không đơn thuần là nhìn vào đó và gõ ra từ bàn phím rồi trở thành mã nguồn. Chúng ta cần tạo một mô hình trừu tượng hóa cho domain.Chúng ta tìm hiểu nhiều về domain trong trao đối với chuyên gia lĩnh vực. Những kiến thức thô (về nghiệp vụ) này không dễ dàng chuyển hóa thành cấu trúc phần mềm trừ khi chúng ta xây dựng mô hình trừu tượng của nó với kế hoạch chi tiết trong đầu. Ban đầu, kế hoạch đó luôn không đầy đủ. Theo thời gian, càng làm việc với nó, chúng ta càng làm cho nó tốt hơn, trở nên rõ ràng hơn. Vậy "*trừu tượng hóa*" là gì? Nó là một mô hình, một mô hình của domain. Theo Eric Evans, một mô hình domain không phải là một giản đồ cụ thể; quan trọng là ý tưởng mà giản đồ đó muốn truyền đạt. Quan trọng không phải là kiến thức trong đầu của chuyên gia ngành; nó là sự trừu tượng hóa của cả nhóm, được tổ chức chặt chẽ của những kiến thức đó. Một giản đồ cần thể hiện và đối thoại với mô hình. Đây có thể là mã nguồn được viết cẩn thận hay một câu văn.

Mô hình là sự thể hiện nội bộ của chúng ta về domain cần xem xét, vốn rất cần thiết trong suốt qui trình thiết kế và phát triển. Trong suốt qui trình thiết kế, chúng ta liên tưởng tới và đề cập tới mô hình này. Thế giới chung quanh ta quá phức tạp để nhập vào đầu rồi xử lý. Ngay cả domain cụ thể cũng có thể vượt quá khả năng xử lý của bộ não con người trong một thời điểm. Chúng ta cần tổ chức thông tin, hệ thống hóa nó, chia nó thành những phần nhỏ hơn, nhóm nó thành những mô-đun theo logic, xử lý từng cái một. Đôi khi, chúng ta cần loại ra một số phần của domain. Một domain chứa quá nhiều thông tin có thể gói gọn trong 1 mô hình. Những cái thừa đó không cần xem xét tới. Đây là một thử thách nội tại của nó. Vậy, chúng ta bỏ gì và giữ gì? Đây là một phần của thiết kế, của quá trình tạo nên phần mềm. Phần mềm ngân hàng đơn thuần chỉ giữ địa chỉ khách hàng nhưng không cần quan tâm tới màu mắt của khách hàng. Một số ví dụ khác có thể phức tạp hơn ví dụ quá hiển nhiên này.

Mô hình là phần cần thiết của thiết kế phần mềm. Chúng ta cần nó để đối phó với sự phức tạp. Mọi qui trình tư duy về domain được cô đọng trong mô hình này. Chúng ta phải tự hiểu ra điều đấy. Sẽ không có ý nghĩa nếu nó chỉ được viết ở đấy, nhưng không ai hiểu, đúng không? Sử dụng mô hình này, chúng ta cần trao đổi với chuyên gia ngành, người thiết kế và kỹ sư. Một mô hình là cần thiết cho một phần mềm, nhưng chúng ta cần tạo ra cách thể hiện chúng, trao đổi với người khác.

Chúng ta không cô độc trong qui trình này và do đó, chúng ta cần chia sẻ kiến thức, thông tin, những gì chúng ta biết một cách chính xác, đầy đủ, rõ ràng. Có nhiều cách khác nhau để đạt được điều này. Một là dùng hình: giản đồ, use case, hình vẽ, ảnh chụp... Một cách khác là viết. Chúng ta viết ra tầm nhìn của domain. Cách khác nữa là ngôn ngữ. chúng ta có thể và nên thế, tạo ra một ngôn ngữ để giao tiếp về những vấn đề cụ thể của domain. Chúng ta sẽ đi vào chi tiết sau nhưng ở đây cần nhấn mạnh điểm chính là; Chúng ta cần trao đổi thông qua mô hình.

Khi đã có mô hình, chúng ta bắt đầu *thiết kế mã nguồn*. Điều này hơi khác với thiết kế phần mềm. Thiết kế phần mềm tương tự như kiến trúc một ngôi nhà, nó là bức trang tổng thể. Còn thiết kế mã nguồn là làm việc với những gì chi tiết như địa điểm sơn tường. Thiết kế mã nguồn cũng rất quan trọng, nhưng nó không phải là phần cơ sở như thiết kế phần mềm. Lỗi thiết kế code thường dễ sửa nhưng lỗi thiết kế phần mềm thường rất tốn kém. Cũng giống như việc ta chuyển bức tranh từ phải qua trái, đặt đồ khác vào bên phải. Tuy vậy, sản phẩm cuối không thể tốt nếu không có thiết kế mã nguồn tốt. Lúc này, mẫu thiết kế mã nguồn trở nên tiện dụng và áp dụng khi cần. Kỹ thuật lập trình tốt giúp tạo mã nguồn sạch và dễ bảo trì.

Có nhiều cách tiếp cận với thiết kế phần mềm. Một là phương pháp thiết kế thác nước. Đây là phương pháp gồm nhiều bước. Chuyên gia nghiệp vụ lên yêu cầu qua việc trao đổi với chuyên gia phân tích nghiệp vụ. Người phân tích (nghiệp vụ) tạo mô hình dựa trên yêu cầu, chuyển cho lập trình viên và viết mã dựa trên những tài liệu phân tích nghiệp vụ họ nhận được. Luồng kiến thức ở đây là một chiều. Dù đây là cách tiếp cận truyền thống của thiết kế phần mềm, nó đã đạt được một mức độ thành công nhất định trong nhiều năm dù có nhiều vấn đề và hạn chế. Vấn đề chính ở mô hình này là không có phản hồi giữa *chuyên gia nghiệp vụ* với *chuyên gia phân tích* cũng như giữa chuyên gia phân tích và lập trình viên.

Một cách tiếp cận khác là phương pháp luận Agile như Extreme Programming (XP). Những phương pháp luận này là cuộc vận động tập thể chống lại cách tiếp cận thác nước vốn có nhược điểm là đưa ra thiết kế trước (khi lập trình), trước khi có thay đổi yêu cầu. Thực sự là rất khó để tạo ra mô hình phủ mọi khía cạnh của domain ngay từ ban đầu. Điều này đòi hỏi suy nghĩ và thường bạn không thể nhìn thấy hết các vấn đề liên quan ngay từ ban đầu, hay bạn không thể nhìn thấy một số hiệu ứng phụ tiêu cực hoặc lỗi trong thiết kế. Một vấn đề mà Agile muốn giải quyết được gọi là "*analysis paralysis*" - là tình huống thành viên nhóm sợ quyết định sửa thiết kế và do đó họ không có tiến độ. Agile khuyến khích nhận thức về tầm quan trọng của quyết định thiết kế và họ chống lại *thiết-kế-trước*. Thay vào đó, họ cố gắng thực thi một cách mềm dẻo qua việc phát triển lặp có sự tham gia của những người liên quan về nghiệp vụ một cách liên tục, refactor liên tục. Qua đó, nhóm phát triển học được domain của khách hàng và tạo ra phần mềm tốt hơn, phù hợp với nhu cầu của khách hàng.

Phương pháp Agile cũng có vấn đề và hạn chế riêng của nó; nó chủ trương sự đơn giản nhưng mọi người lại có cách nhìn riêng, mỗi phần có ý nghĩa riêng. Tương tự, việc refactor liên tục của lập trình viên khi không có nguyên tắc thiết kế vững chắc sẽ tạo ra mã nguồn khó hiểu và khó thay đổi. Tuy cách tiếp cận thác nước có thể dẫn tới tình trạng tập trung quá nhiều vào mặt kỹ thuật, sự sợ hãi đụng tới kỹ thuật có thể dẫn tới sự sợ hãi khác: Sợ đụng sâu, toàn diện vào thiết kế.

Cuốn sách này trình bày những nguyên tắc của thiết kế hướng lĩnh vực. Nếu được áp dụng sẽ tăng khả năng thiết kế cho mô hình và thực thi những vấn đề phức tạp trong domain theo cách dễ bảo trì. Thiết kế hướng lĩnh vực kết hợp thiết kế và phát triển, chỉ ra rằng thiết kế và phát triển có thể *cộng dồn* để tạo nên giải pháp tốt hơn. Thiết kế tốt tạo đà cho sự phát triển, phản hồi từ qui trình phát triển sẽ cải tiến thiết kế.

Xây dựng Kiến thức về Domain

Chúng ta hãy xem xét ví dụ trong dự án hệ thống điều khiển lịch bay và xem kiến thức domain được xây dựng thế nào.

Có hàng ngàn máy bay di chuyển trong không gian vào một thời điểm bất kỳ. Chúng di chuyển theo quỹ đạo riêng, tới đích của chúng và quan trọng là chúng không được để máy bay đâm vào nhau. Chúng ta không thể xây dựng được cả hệ thống điều khiển giao thông, nhưng một phần nhỏ của hệ thống theo dõi bay thì có thể. Dự án được đề xuất là hệ thống theo dõi mọi chuyến bay trong một khu vực nhất định, xác định xem chuyến bay theo đúng lộ trình của nó hay không, liệu có xảy ra va chạm hay không.

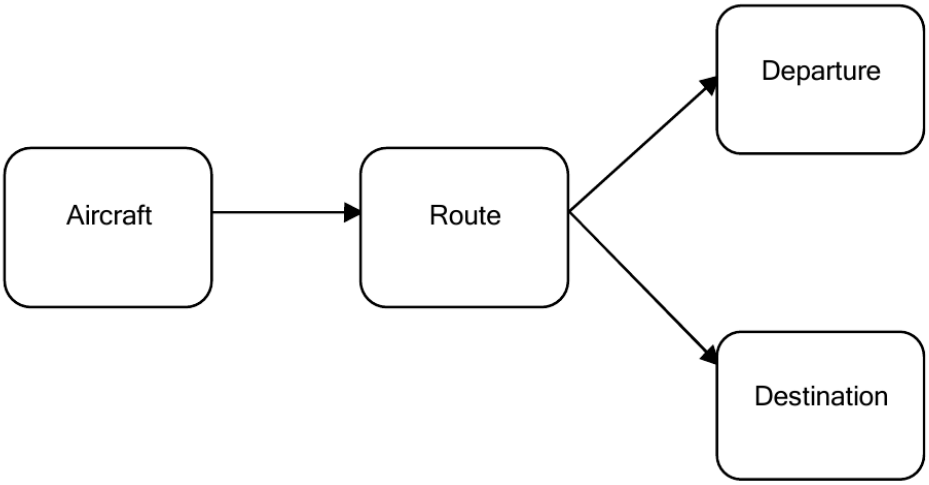
Chúng ta bắt đầu phát triển phần mềm này từ đâu? Trong phần trước chúng ta nói rằng đầu tiên cần hiểu domain, trường hợp này là theo dõi không lưu. Người theo dõi không lưu là chuyên gia trong mảng này. Tuy nhiên, người theo dõi không lưu không phải là người thiết kế hay chuyên gia phần mềm. Bạn không thể hy vọng họ viết cho bạn mô tả vấn đề về domain của họ.

Người theo dõi không lưu có kiến thực rộng về domain của họ, nhưng để mô hình hóa được, bạn cần chắt lọc thông tin và tổng quát hóa nó. Khi bạn nói chuyện với họ, bạn sẽ nghe nhiều các từ như "*cất cánh*", "*hạ cánh*", "*máy bay trên không trung*" và rủi ro va chạm rồi việc máy bay chờ để được hạ cánh, vv... Để có một thứ tự nhịp nhàng những thông tin hỗn loạn này, chúng ta cần bắt đầu từ đâu đó.

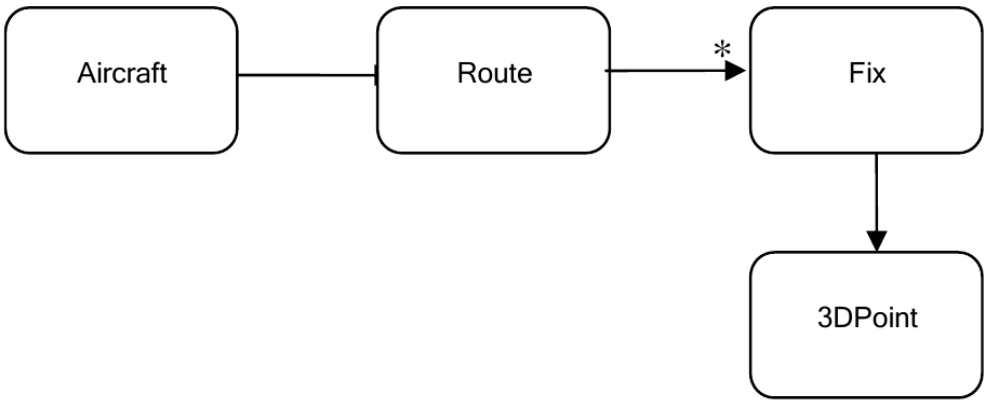
Người điều khiển và bạn đồng ý rằng mỗi máy bay đều có điểm xuất phát và điểm hạ cánh. Như vậy chúng ta có "*máy bay*", "*xuất phát*" và "*điểm đích*", như hình dưới đây.

Ổn hơn chưa? Máy bay cất cánh từ một nơi vào đó và đáp xuống ở chỗ khác. Những gì xảy ra trên không trung? Lộ trình bay theo đường nào? Thực ra chúng ta cần biết hơn về những gì xảy ra trên không trung. Người điều khiển nói rằng mỗi máy bay đều được gắn với một lịch trình bay - mô tả toàn bộ lộ trình trên không. Khi nghe thấy "*lộ trình bay*", bạn liên tưởng tới con đường mà máy bay bay trên không khi máy bay bay. Sau một hồi trao đổi, bạn biết thêm một từ thú vị: *route*. Dễ nhận thấy rằng, đập ngay vào sự chú ý của bạn, rất tốt. "*Route*" bao gồm những khái niệm quan

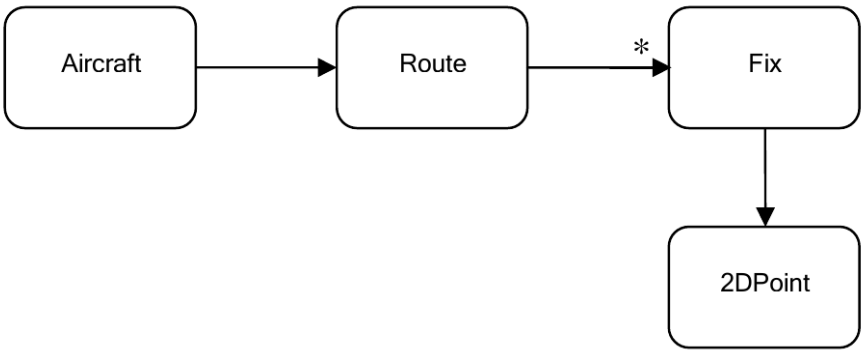
trọng về *flight travel*. Đó là những gì máy bay làm khi nó đang bay, nó đi theo một route. Hiển nhiên là điểm khởi hành và điểm đích của máy bay là điểm đầu và điểm cuối của route đó. Do đó, thay vì gắn máy bay với điểm "*khởi hành*" và điểm "*đích*", sẽ là tự nhiên hơn nếu ta gắn máy bay với route, ở đó route tương ứng với điểm xuất phát và điểmđích.



Tiếp tục trao đổi với người điều khiển về route của máy bay, bạn khám phá ra rằng route được tạo từ nhiều đoạn, tạo thành một đường gấp khúc từ khởi hành tới đích. Đường gấp khúc này được giả định là đi qua một số điểm được chỉ định trước. Do đó, một route được coi là một chuỗi các *fix* liên tiếp. Vào lúc này, bạn không nhìn thấy xuất phát và đích như là điểm kết thúc của route, mà là hai *fix* khác nhau. Đây có thể là cách mà người điều khiển nhìn *route*, nhưng nó có ích cho việc trừu tượng hóa sau này. Kết quả dựa trên thay đổi đã khám phá như sau:



Giản đồ này thể hiện thêm một nhân tố: Mỗi *fix* là một điểm trong không gian theo một *route*, được thể hiện bằng một điểm ba chiều. Nhưng khi bạn nói chuyện với người điều khiển thì lại hiểu ra rằng anh ta không nhìn điểm theo cách đó. Thực ra anh ta nhìn *route* như là một ánh xạ trên mặt đất của chuyến bay. Các *fix* chỉ là các điểm trên mặt đất được xác định duy nhất qua vĩ độ và kinh độ. Như vậy, giản đồ đúng như sau:



Điều gì xảy ra ở đây? Bạn và chuyên gia nghiệp vụ nói chuyện và trao đổi kiến thức. Bạn đặt câu hỏi, anh ta trả lời. Khi họ làm việc đó tức là họ đào sâu hơn những khái niệm về mảng không lưu. Những khái niệm này xuất hiện một cách không chau chuốt, không tổ chức nhưng nó cần thiết cho việc hiểu domain. Bạn cần hiểu họ càng nhiều càng tốt từ chuyên gia về domain. Bằng cách đặt câu hỏi đúng, xử lý thông tin đúng cách, bạn và chuyên gia dần sẽ vẽ ra domain, một mô hình domain. Cái nhìn này không hoàn hảo hay không đúng, nhưng bạn cần bắt đầu từ đó. Hãy tìm cách hiểu những khái niệm cần thiết cho domain.

Giản đồ là phần quan trọng của thiết kế. Thông thường, những trao đổi giữa người kiến trúc phần mềm hay lập trình viên với chuyên gia domain thường dài. Chuyên gia phần mềm muốn trích xuất kiến thức từ chuyên gia domain và họ muốn chuyển kiến thức đó thành hình dạng có ích. Ở một thời điểm nào đó, họ có thể cần tạo prototype để xem mọi thứ hoạt động ra sao. Khi làm việc đó họ có thể tìm ra những vấn đề của mô hình hay trong cách tiếp cận của họ và nếu cần, họ thay đổi mô hình. Sự liên lạc giữa hai bên không phải là một chiều, từ chuyên gia domain tới người thiết kế phần mềm và tới lập trình viên. Có phản hồi giữa các bên và phản hồi đó giúp tạo nên một mô hình tốt hơn, trong sáng hơn và hiểu đúng hơn về domain. Chuyên gia domain hiểu lĩnh vực của họ cũng như kỹ năng ở mảng đó nhưng họ không biết cách tổ chức kiến thức của họ theo cách cụ thể, hữu ích cho người phát triển để thực thi một hệ thống phần mềm. Tư duy phân tích phần mềm của người thiết kế phần mềm đóng vai trò chính trong trao đổi với chuyên gia domain và điều đó cũng giúp cho việc xây dựng một cấu trúc cho việc trao đổi về sau như ta sẽ thấy trong chương tiếp theo. Chúng ta, những chuyên gia phần mềm (kiến trúc sư phần mềm và lập trình viên) và các chuyên gia domain cùng tạo nên mô hình cho domain và mô hình đó là nơi chuyên môn của cả hai bên giao thoa. Qui trình này có vẻ mất thời gian nhưng vì mục đích cuối cùng của phần mềm là giải quyết vấn đề nghiệp vụ trong thực tế, qui trình đó phải uyển chuyển theo domain.

Ngôn ngữ Chung

Sự cần thiết của một ngôn ngữ chung

Trong chương trước chúng 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 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 đời 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, persistence, 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.

Trong ví dụ 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, chúng ta xây dựng mô hình, chúng ta phải trao đổi, giao tiếp ý tưởng về mô hình, về những thành phần liên quan đến mô hình, cách chúng 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 chúng 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 ngườicó 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 thấp trong mã nguồn hay trong văn bản viết.

Khi trao đổi, chúng 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.

Chúng 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 chúng ta cần nói chung một ngôn ngữ khi chúng ta gặp và trao đổi về mô hình, qua đó định 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 mìn 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 chắt lọc. Chúng 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 chúng 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.

Tạo nên Ngôn ngữ chung

Chúng ta xây dựng một ngôn ngữ thế nào? Đây là một đoạn hội thoại giả định giữa lập trình viên và chuyên gia domain trong dự án theo dõi không lưu. Hãy để ý những từ in đậm.

Developer: Chúng ta cần theo dõi không lưu. Nên bắt đầu từ đâu nhỉ?

Expert: Hãy bắt đầu từ cơ bản. Mọi traffic đều được tạo nên bởi planes. Mỗi plan cất cánh từ điểm xuất phát và hạ cánh ở điểm đích.

Developer: Như vậy dễ quá. Khi nó bay, plan cần chọn đường đi trong không trung theo các mà phi công muốn đúng không? Họ tùy ý quyết định đường đi tới điểm đích đúng không nhỉ?

Expert: Không phải thế. Phi công nhận được thông tin route và phải theo route đó. Họ cần phải bám theo route càng sát càng tốt.

Developer: Tôi nghĩ là chúng ta đang dùng hệ quy chiếu Đề-các và một chuỗi điểm 3D. Route là một path trong không trung. Route đơn thuần là một chuỗi điểm 3D.

Expert: Tôi không nghĩ vậy. Tôi nhìn route theo cách khác. Route thực ra là ánh xạ trên mặt đất của đường mà máy bay bay. Route đi theo một chuỗi các điểm được định nghĩa bởi vĩ độ và kinh độ.

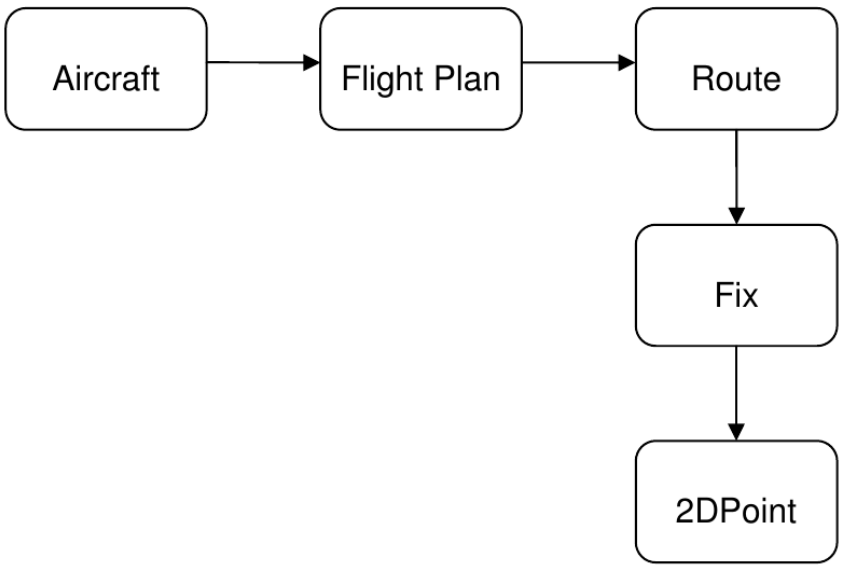
Developer: OK, vậy ta hãy gọi các điểm này là fix vì chúng là các điểm cố định trên mặt đấy. Và chúng ta dùng chuỗi điểm 2D để mô tả path. À, tiện nói thêm, điểm xuất phát và điểm đích cũng chỉ là những fix. Chúng ta nên xem xét riêng từng khái niệm. Route đạt tới đích khi nó vượt qua mọi fix. Máy bay phải bay theo route, nhưng không có nghĩa là nó muốn bay ở cao độ nào cũng được?

Expert: Không. Cao độ của máy bay ở thời điểm nào đó được thiết lập trong flight plan.

Developer: Flight plan là cái gì vậy?

Expert: Trước khi rời sân bay, phi công nhận được flight plan chi tiết, trong đó có mọi thông tin về chuyến bay: route, cao độ, tốc độ, kiểu máy bay và thậm chí thông tin của một số thành viên tổ lái.

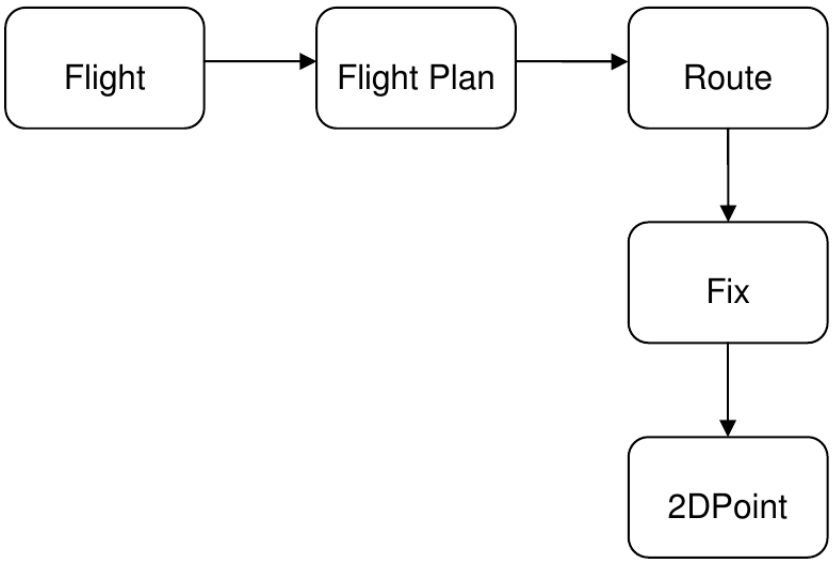
Developer: Có vẻ flight plan rất quan trọng đối với tôi. Hãy đưa nó vào mô hình



Developer: Tốt hơn rồi. Khi nhìn vào hình này, tôi nhận ra một số điểm. Khi chúng ta theo dõi không lưu, thực ra chúng ta không cần để ý tới bản thân máy bay xem nó màu trắng hay xanh, là Boeing hay Airbus. Cái chúng ta cần quan tâm là flight. Đây là cái chúng ta cần theo dõi và đo. Tôi nghĩ ta nên hiệu chỉnh mô hình chính xác hơn một chút.

Hãy để các nhóm trao đổi về domain theo dõi không lưu và mô hình sơ bộ, họ từ từ tạo ra ngôn ngữ từ những từ in đậm. Chúng ta cũng để ý được sự thay đổi trong mô hình!

Tuy nhiên, trong thực tế trao đổi dài hơn thế này, và người ta thường nói một cách gián tiếp hoặc đi rất sâu vào chi tiết, hay chọn sai khái niệm; điều này có thể gây khó khăn cho việc tạo nên ngôn ngữ. Để chỉ ra những khó khăn đó, thành viên nhóm cần ý thức rằng việc tạo ra ngôn ngữ chung luôn quan trọng, cần tập trung để ý và họ phải dùng ngôn ngữ bất kỳ khi nào cần thiết. Chúng ta nên dùng jargon riêng trong những phiên thế này ít nhất có thể, thay vào đó chúng ta dùng Ngôn ngữ chung để giao tiếp trở nên rõ ràng và chính xác.



Lập trình viên được khuyến nghị thực thi những khái niệm chính trong mô hình và mã nguồn. Một lớp viết cho Route và một lớp khác viết cho Fix. Lớp Fix có thể kế thừa lớp 2DPoint, hay chứa 2DPoint như là một thuộc tính chính. điều này phụ thuộc vào các yếu tố khác mà chúng ta sẽ đề cập sau này. bằng việc tạo ra lớp cho những khái niệm mô hình tương ứng, chúng ta ánh xạn giữa mô hình và mã nguồn, giữa ngôn ngữ và mã nguồn. Điều này hữu dụng vì nó làm cho mã nguồn trở nên dễ đọc hơn, làm cho mã nguồn có thể tạo nên mô hình. việc có mã nguồn mô tả mô hình sẽ được bù đắp cho dự án về sau này khi mô hình phình to, và khi thay đổi mã nguồn dẫn tới những hệ quả không mong muốn nếu mã nguồn không được thiết kế đúng đắn.

Chúng ta đã thấy cách ngôn ngữ được chia sẻ trong cả nhóm, và cũng thấy nó giúp xây dựng kiến thức cũng như tạo nên mô hình như thế nào. Chúng ta nên dùng ngôn ngữ như thế nào? Chỉ bằng lời nói? Chúng ta đã dùng biểu đồ. Còn gì khác nữa không? Bằng cách viết?

Một số người có thể nói rằng UML là đủ tốt để xây dựng mô hình. UML là công cụ tuyệt vờ để viết ra những khái niệm chính như lớp cũng như mô tả quan hệ giữa chúng. Bạn có thể vẽ phác họa bốn hay năm lớp viết ra tên chúng, chỉ ra quan hệ giữa chúng. UML rất dễ dùng để mô tả những gì bạn nghĩ, một ý tưởng bằng hình rất dễ hiểu. Một người chia sẻ tức thì tầm nhìn chung về một chủ đề nào đó, và việc trao đổi trở nên đơn giản hơn nếu dựa trên UML. Khi ý tưởng mới xuất hiện, giản đồ được cập nhật theo những thay đổi có tính khái niệm.

Giản đồ UMLrất hữu dụng khi số các phần tử ít. Nhưng UML phình như nấm sau mưa. Bạn sẽ làm gì nếu có hàng trăm lớp trong một file dài lê thê? Chuyên gia phần mềm đọc UML như thế rất khó. Họ không hiểu khi UML to hay ngay cả với dự án cỡ trung bình.

Ngoài ra, UML rất tốt khi mô tả lớp, các thuộc tính và quan hệ giữa chúng. Tuy vậy, ứng xử của lớp và ràng buộc không dễ mô tả. Vì những lý do đó chúng ta thêm giải thích bằng lời vào UML. Do UML không thể chuyển tải được hai mặt quan trọng của mô hình: Ý nghĩa của khái niệm nó trình bày và việc đối tượng phải làm gì. Việc này không ảnh hưởng gì do chúng ta có thể thêm những công cụ giao tiếp để giải quyết.

Chúng ta có thể dùng tài liệu. Một cách giao tiếp khôn ngoan với mô hình là tạo những giản đổ nhỏ chứa những tập nhỏ của mô hình. Những giản đồ này chứa nhiều lớp và quan hệ giữa chúng. Làm vậy ta đã chuyển tại được phần lớn những khái niệm liên quan. Chúng ta cũng thể thêm giải thích chữ vào giản độ. Văn bản sẽ giải thích hành vi và ràng buộc vốn không thể chuyển tải được bằng giản độ. Mỗi phần nhỏ giải thích một khía cạnh quan trọng của domain, nó chỉ ra phần cần làm sáng tỏ của domain.

Những tài liệu có thể vẽ bằng tay vì nó chỉ truyền đạt "tinh thần" một cách tạm thời và có thể thay đổi trong tương lai gần do mô hình thay đổi nhiều lần từ khi bắt đầu dự án tới khi mô hình đạt tới trạng thái ổn định.

Chúng ta bị cám dỗ tạo ra một giản đồ lớn trải trên nhiều mô hình. Tuy nhiên, trong đa số trường hợp, giản đồ không thể gộp cùng nhau. Hơn nữa, ngay cả khi chúng ta gộp được, giản đồ gộp sẽ rất lộn xộn. Do đó, việc tạo một tập hợp nhiều giản đồ nhỏ vẫn tốt hơn.

Hãy thận trọng với những tài liệu dài. Ta vừa tốn thời gian tạo ra chúng và những tài liệu đó trở nên lỗi thời trước khi chúng được hoàn thiện. Tài liệu phải đồng bộ với mô hình. Tài liệu cũ, dùng ngôn ngữ sai, không phản ánh đúng mô hình là không hữu ích. Hãy tránh điều này nếu có thể.

Chúng ta cũng có thể giao tiếp bằng mã nguồn. Cách tiếp cận này được khởi xướng bởi cộng đồng XP. Mã nguồn tốt có tính giao tiếp cao. Cho dù hành vi được mô tả bởi method là rõ ràng nhưng tên method có đủ rõ ràng bằng phần thân của nó không? Việc kiểm thử tự nói lên kết quả nhưng tên biến và cấu trúc mã nguồn tổng thể thì sao? Chúng diễn tả được mọi ý một cách rõ ràng không? Chức năng mà mã nguồn thực thi làm đúng việc, nhưng nó có làm việc đó đúng không. Việc viết một mô hình trong mã nguồn là rất khó.

Có nhiều cách khác để trao đổi trong quá trình thiết kế. Cuốn sách này không nhằm mục đích trình bày hết mọi khía cạnh đó. Một điều rất rõ ràng là: nhóm thiết kế (bao gồm người thiết kế phần mềm, lập trình viên và chuyên gia domain) cần một ngôn ngữ thống nhất cho hành động của họ và giúp họ tạo ra một mô hình thể hiện mô hình đó bằng mã nguồn.

3

THIẾT KẾ HƯỚNG MÔ HÌNH

Các chương 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. Chúng 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.

Nó xảy ra khi mà các nhà phân tích phần mềm làm việc với chuyên gia domain trong vòng nhiều tháng, khám phá ra những yêu tố căn bản của domain, nhấn mạnh quan hệ giữa chúng, và tạo ra một model đúng đắn, có thể nắm bắt chính xác domain. Sau đó, các mô hình được chuyển cho các lập trình viên. Các lập trình viên này có thể nhìn vào model và phát hiện ra rằng một số khái niệm hoặc các mối quan hệ được tìm thấy trong nó không thể được thể hiện đúng trong code. Vậy nên họ sử dụng model theo cảm hứng, nhưng họ tao ra thiết kế của riêng họ, gồm ý tưởng từ model, và thêm vào đó một số thứ của họ. Quá trình phát triển mới tiếp tục, và các class được thêm vào code, mở rộng sự chia rẽ giữa model ban đầu và lần triển khai trước đó. Có một kết quả tốt sẽ không được đảm bảo. Các lập trình viên có thể lấy về một sản phẩm để làm việc cùng nhau, nhưng nó lại đứng trước những thử thách về thời gian? Liệu nó có dễ dàng mở rộng? Và liệu rằng nó có thể dễ dàng bảo trì?

Bất kì domain nào cũng có thể được thể hiện bằng nhiều model, và bất kỳ model nào cũng có thể được biểu thị bằng nhiều cách khác nhau trong code. Đối với từng vấn đề cụ thể có thể có nhiều hơn một giải pháp. Vậy lựa chọn của chúng ta là gì? Dù có một model được phân tích đúng đắn cũng không có nghĩa là model ấy sẽ được biểu hiện thẳng vào trong code. Hoặc là thực hiện nó có thể sẽ phá vỡ một số nguyên tắc thiết kế phần mềm, một điều không nên. Điều quan trọng là chọn một model chính xác và có thể dễ dàng để đưa vào trong code. Câu hỏi cơ bản ở đây là: làm thế nào để chúng ta tiếp cận quá trình chuyển đổi từ model cho đến code?

Một trong những kỹ thuật thiết kế được đề nghị được gọi với cái tên là analysis model, đây được xem như một sự tách biệt với thiết kế code đồng thời thường được thực hiện bởi những người khác nhau. Analysis model là kết quả của quá trình phân tích domain, kết quả trong một model không chứa sự cân nhắc để phần mềm sử dụng cho cài đặt. Một mô hình được sử dụng để hiểu rõ domain. Một lượng kiến thức được xây dựng, và dẫn tới một model có thể được phân tích chính xác. Phần

mềm không được xét đến trong giai đoạn này vì nó được coi là một yếu tố khó hiểu. Model này ảnh hưởng đến các lập trình viên trong đó có việc phải thiết kế chúng. Vì mô hình này không được xây dựng dựa trên ý thức về các nguyên tắc thiết kế, nó có thể không phục vụ mục đích này tốt. Những lập trình viên sẽ phải thay đổi nó, hoặc tạo thiết kế khác. Và đó không còn là ánh xạ giữa model và code nữa. Kết quả là các phân tích model bị bỏ rơi ngay sau khi mã hóa bắt đầu.

Một trong những vấn đề chính của phương pháp này là người phân tích không thể thấy trước một số khiếm khuyết trong model của họ, và tất cả những sự phức tạp của domain. Người phân tích có thể quá sa đà vào một số thành phần của model mà bỏ qua các phần khác. Chi tiết rất quan trọng được phát hiện trong quá trình thiết kế và thực hiện. Một mô hình đáng tin cậy với domain có thể trở thành vấn đề nghiêm trọng tới việc lưu trữ đối tượng, hoặc thực hiện hành vi không thể chấp nhận được.

Các lập trình viên sẽ bị buộc phải thực hiện những quyết định riêng của họ, và nó sẽ làm thay đổi thiết kế đối với các model, cái mà không được xét tới khi các model này được tạo ra nhằm giải quyết một vấn đề trong thực tế. Họ tạo ra một thiết kế trượt ra khỏi các model và làm cho ít liên quan đến chúng.

Nếu người phân tích làm việc một cách độc lập thì cuối cùng họ sẽ tạo ra một model. Khi những model này được gửi tới những người thiết kế, vài hiểu biết về domain của người phân tích sẽ bị mất đi. Trong lúc thể hiện model bằng các diagram hoặc cách viết ra, rất có thể là các nhà thiết kế sẽ không nắm bắt được toàn bộ ý nghĩa của model, hoặc các mối quan hệ giữa một số đối tượng, cũng như hành vi giữa chúng. Có những chi tiết của một model mà trong đó không hề dễ dàng trình bày đầy đủ bằng một diagram, hay thậm chí bằng văn bản. Các lập trình viên sẽ có một khoảng thời gian khó khăn để tạo ra chúng (trong code). Trong một số trường hợp, họ sẽ tạo ra vài giả định cho các hành vi được mong đợi, và một trong số chúng có thể khiến họ trở nên sai lầm, kết quả là chức năng của chương trình hoạt động không chính xác.

Người phân tích có các cuộc *họp kín* của riêng của họ nơi mà có nhiều cuộc tranh luận được diễn ra về domain, và ở đó có rất nhiều kiến thức được chia sẻ. Họ tạo ra một model mà được cho là chứa đựng tất cả thông tin được cô đọng, và các lập trình viên phải tiêu hóa toàn bộ số tài liệu họ được giao. Nó sẽ thu được nhiều hiệu quả hơn nếu các lập trình viên có thể tham gia các cuộc họp cùng người phân tích và do đó đạt được một cái nhìn rõ ràng và đầy đủ hơn về domain và các model trước khi họ bắt đầu thiết kế code.

Một cách tiếp cận tốt hơn là làm quá trình mô hình hóa domain và design liên quan chặt chẽ với nhau. Model nên được xây dựng với nhiều quan điểm dành cho phần mềm và sự thiết kế kỹ lưỡng. Lập trình viên nên tham gia vào quá trình mô hình hóa. Mục đích chính là chọn ra một model có thể thích hợp với phần mềm, do đó quá trình thiết kế phải minh bạch và dựa trên model. Sự liên quan chặt chẽ giữa code và model nằm dưới nó sẽ khiến code có nghĩa và gắn với model.

Để các lập trình viên tham gia cung cấp phản hồi. Nó khiến model chắc chắn có thể hiện thực ở trong phần mềm. Nếu có điều đó gì không đúng, nó sẽ được xác định ở giai đoạn đầu, và các vấn đề có thể dễ dàng sửa chữa hơn.

Người viết code cũng cần phải hiểu rõ về model và có trách nhiệm với tính nhất quán của nó. Họ nên nhận ra rằng sự thay đổi mã đồng nghĩa với một sự thay đổi model; bằng không họ sẽ refactor lại code đến điểm mà nó không còn thể hiện model gốc nữa. Nếu người phân tích bị tách ra từ quá trình cài đặt, anh ta sẽ sớm mất đi mối quan tâm của mình về những vấn đề hạn chế được đưa ra bởi quá trình phát triển. Kết quả là model trong đó trở nên không thực tế.

Mọi vị trí làm về kỹ thuật đều phải có đóng góp vào model bằng cách dành thời gian để tiếp xúc với code, bất kể vị trí anh ấy hay cô ấy là gì trong dự án. Bất cứ ai chịu trách nhiệm về việc thay đổi code đều phải học cách trình bày một model thông qua code. Mỗi lập trình viên đều phải được tham gia một mức độ nào đó về cuộc thảo luận của model và có liên lạc với *chuyên gia ngành.* Những người tham gia đóng góp theo nhiều cách khác nhau cần có ý thức đóng góp trao đổi tích cực cho người viết code về ý tưởng model thông qua Ngôn ngữ Chung.

Nếu thiết kế, hoặc một thành phần nào của nó, không được ánh xạ sang mô hình domain thì model đó trở nên ít giá trị, và tính đúng đắn của phần mềm là đáng nghi ngờ. Đồng thời, ánh xạ phức tạp giữa model và thiết kế chức năng khó cho việc hiểu, trong thực tế, cũng không khả thi để duy trì những thay đổi về thiết kế. Việc chia có tính mở giữa phân tích và thiết kế sao cho nhận thức thu được của mỗi hoạt động là không ăn nhập với phần còn lại.

Thiết kế một thành phần nào đó của hệ thống phần mềm để phản ánh domain model theo cách rất trực quan, vậy nên phải lập ánh xạ rõ ràng. Xem lại các model và sửa đổi chúng sao cho dễ dàng cài đặt hơn, thậm chí là cả khi bạn tìm cách khiến chúng phản ánh cái nhìn sâu sắc về domain. Yêu cầu với một model là phải phục vụ tốt cả hai mục đích trên, đồng thời cũng hỗ hợ Ngôn ngữ chung một cách trôi chảy.

Rút ra từ mô hình là thuật ngữ được sử dụng cho thiết kế và sự phân công nhiệm vụ cơ bản. Đoạn code trở thành sự diễn tả của mô hình, vậy một sự thay đổi về code có thể dẫn tới thay đổi về model. Ảnh hưởng của nó cần được lan tỏa ra các hoạt động còn lại của dự án cho phù hợp.

Để khiến việc cài đặt model trở nên chặt chẽ thường cần yêu cầu một công cụ và ngôn ngữ phát triển phần mềm hỗ trợ việc mô hình hóa, giống như là lập trình hướng đối tượng.

Lập trình hướng đối tượng là phù hợp để cài đặt model vì chúng cùng có chung mô hình. Lập trình hướng đối tượng cung cấp các class của đối tượng và liên kết giữa các class, các biểu hiện của đối tượng, cùng thông điệp giữa chúng. Ngôn ngữ OOP khiến chúng có khả năng tạo ánh xạ trực tiếp giữa đối tượng model và quan hệ và gắn kế của chúng.

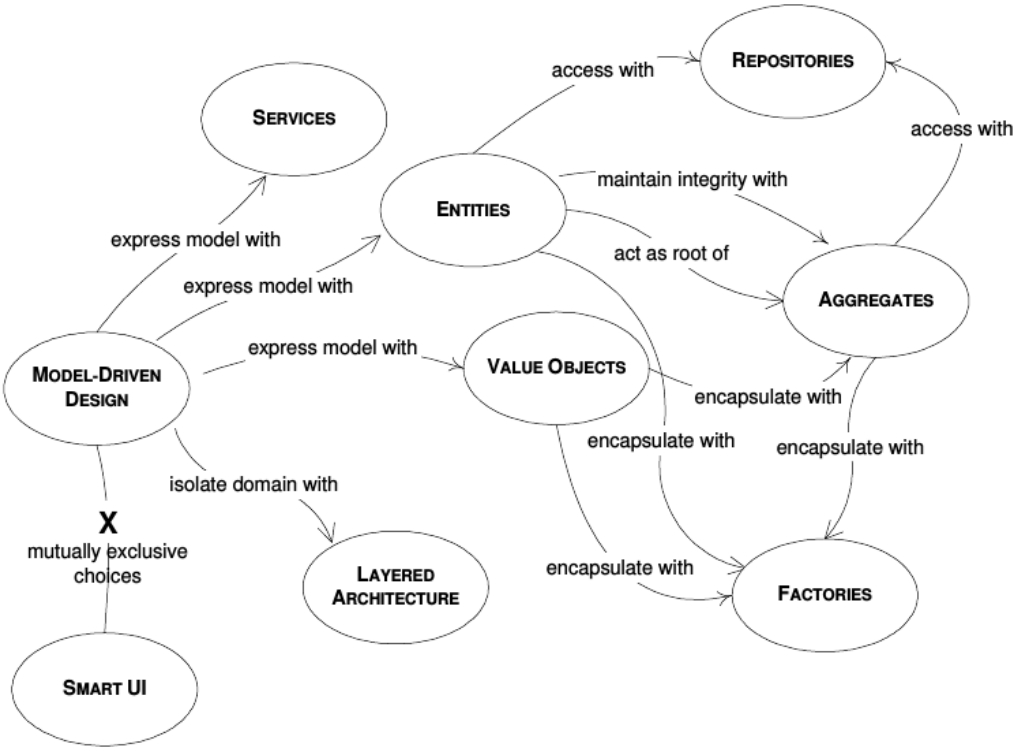
Ngôn ngữ thủ tục cung cấp sự hỗ trợ hạn chế cho thiết kế hướng nghiệp vụ. Các ngôn ngữ như thế không cung cấp đủ các cấu trúc cần thiết để hiện thực các thành phần chính của một model. Một số người nói rằng OOP có thể được thực hiện với một ngôn ngữ thủ tục như C, và trên thực tế, một số chức năng có thể được sao chép theo cách đó. Đối tượng cần được mô phỏng như cấu trúc dữ liệu. Cấu trúc như vậy không có các hành vi của các đối tượng, và có thêm một số chức năng riêng biệt. Ý nghĩa của dữ liệu đó chỉ tồn tại trong đầu của lập trình viên, vì đoạn code của chính nó không đủ rõ nghĩa. Một chương trình được viết bằng ngôn ngữ thủ tục thường được coi như là một tập hợp các chức năng, một hàm gọi tới các hàm khác, và làm việc cùng nhau để đạt được một kết quả nhất

định. Một chương trình như vậy không hề dễ dàng đóng gói các khái niệm kết nối, làm cho ánh xạ giữa domain và mã nguồn khó được nhận ra.

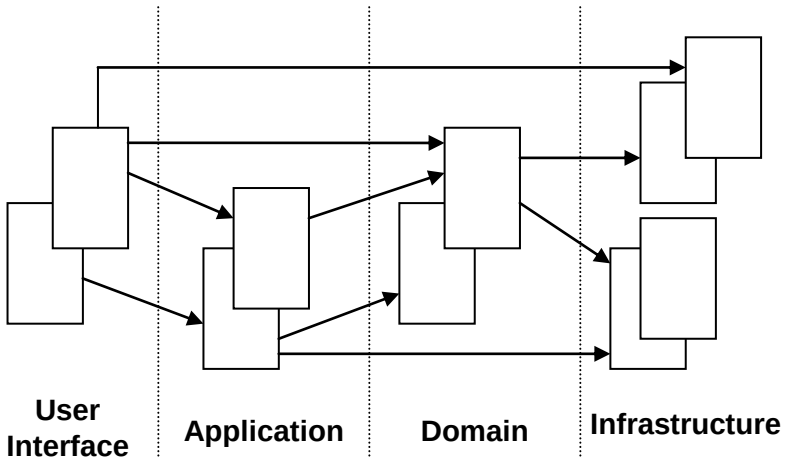
Ở một vài lĩnh vực đặc biệt, như là Toán học, có thể dễ dàng được mô hình hóa và cài đặt theo ngôn ngữ thủ tục vì nhiều lý thuyết toán học được giải quyết đơn giản bằng cách gọi hàm và cấu trúc dữ liệu vì nó chủ yếu là về các tính toán. Lĩnh vực phức tạp hơn không chỉ là một bộ các khái niệm trừu tượng liên quan đến tính toán, và không thể giảm bớt tập hợp thuật toán, vậy ngôn ngữ thủ tục không đạt được của nhiệm vụ thể hiện các model tương ứng. Vì lý do đó, ngôn ngữ lập trình thủ tục không được khuyến cáo sử dụng cho thiết kế hướng lĩnh vực.

Các Khối Ghép Của Một Thiết kế Hướng Lĩnh vực

Phần sau đây của chương này sẽ giới thiệu 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.



Kiến trúc phân lớp



Khi chúng 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, vân vân...

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, vân vân, 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):

User Interface (Presentation Layer)[1]	Chịu trách nhiệm trình bày thông tin tới người sử dụng và thông dịch lệnh của người dùng.
Application Layer[2]	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.
Domain Layer[3]	Tầng này chứa thông tin về các lĩnh vực. Đây là trái tim của nghiệp vụ phần mềm. Trạng thái của đối tượng nghiệp vụ được giữ tại đây. Persistence của các đối tượng nghiệp vụ và trạng thái của chúng có thể được ủy quyền cho tầng Infrastructure.
Infrastructure Layer[4]	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, vv...

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 vướng víu 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.

Thực thể

Trong các đối tượng của một phần mềm, có một nhóm các đối tượng có định danhriêng, 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

[1] Giao diện người dùng
[2] Lớp Chương trình
[3] Lớp domain
[4] Lớp hạ tầng

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

Cuối cùng chúng 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 chúng 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 mã, 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.

Value Object[5]

Ở phần trước chúng ta đã đề cập đến các thực thể và tầm quan trọng của việc xác định chúng trong giai đoạn đầu của việc thiết kế. Vậy nếu thực thể là những đối tượng cần thiết trong mô hình domain thì liệu chúng ta có nên đặt tất cả các đối tượng đều là các thực thể không? Và mọi đối tượng có cần phải có định danh riêng hay không?

Chúng ta có thể theo thói quen đặt tất cả các đối tượng là các thực thể. Ta có thể truy vết các thực thể. Nhưng việc truy vết và tạo các định danh có giá của nó. Chúng ta phải chắc rằng mỗi thể hiện có một định danh duy nhất, mà việc truy vết định danh thì không phải là đơn giản. Chúng ta cần cẩn thận trong việc xác định định danh, bởi nếu ta quyết định sai thì có thể dẫn tới 2 hay nhiều đối tượng khác nhau mà có chung định danh. Vấn đề khác của hành động đặt tất cả các đối tượng thành thực thể đó là hiệu năng. Vì khi đó phải có từng thể hiện cho mọi đối tượng. Ví dụ nếu Customer là một đối tượng thực thể, thì mỗi thể hiện của đối tượng này, đại diện cho một vị khách cụ thể của ngân hàng, không thể được sử dụng lại cho bất cứ thao tác tài khoản nào của khách hàng khác. Kéo theo đó là ta phải tạo từng thể hiện cho mọi khách hàng trên hệ thống. Việc này sẽ ảnh huởng tới hiệu năng của hệ thống khi ta có tời hàng ngàn thể hiện.

Lấy ví dụ một ứng dụng vẽ. Trên một trang giấy, người sử dụng có thể vẽ các điểm hay đường với độ dày, kiểu dáng và màu sắc bất kỳ. Tiện nhất là ta sẽ tạo một lớp đặt tên là Point, và chương trình của ta sẽ tạo một thể hiện của lớp này với từng điểm trên trang giấy. Mỗi điểm đó chưa đựng 2 thuộc tính là tọa độ (x, y). Nhưng câu hỏi là có cần thiết để xem mỗi điểm đó có một định danh? Nó có tính liên tục không, khi mà ta chỉ cần quan tâm tới tọa độ khi dùng các đối tượng Point?

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

[5] Đối tượng giá trị

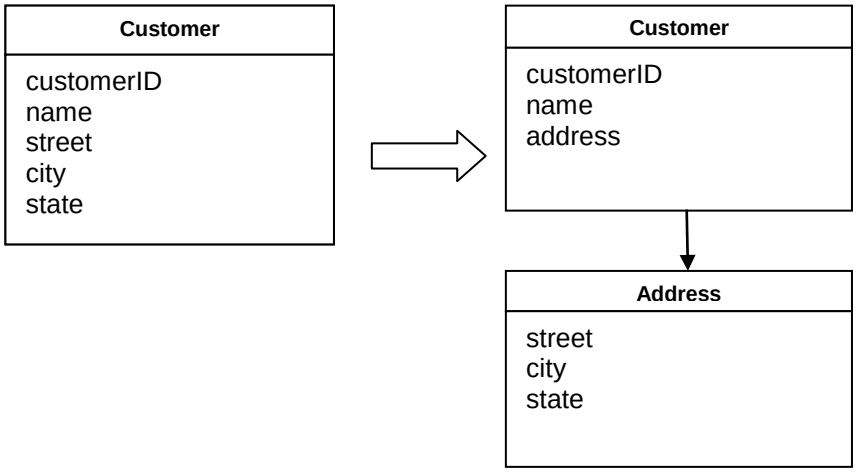
Vì những đặc điểm này nên ta cần phân biệt rõ Entity Object và Value Object. Để đảm bảo tính đồng nhất ta sẽ không gắn mọi đối tượng thành thực thể. Thay vào đó ta chỉ gắn thực thể cho những đối tượng nào phù hợp nhất với các đặc tính của thực thể. Các đối tượng còn lại sẽ là Value Object. (Chúng ta sẽ có các loại đối tượng khác ở chương sau, nhưng giờ ta sẽ giả định chỉ có Entity Object và Value Object). 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, và *garbage collector* sẽ lo phần dọn dẹp các đối tượng không được tham chiếu tới bởi bất cứ đối tượng nào khác. 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.



Dịch Vụ

Khi phân tích domain để xác định những đối tượng chính có trong mô hình chúng 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 gắn 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 chúng ta đang sử dụng Lập trình hướng đối tượng nên chúng 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 chúng 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 gắn 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 và 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 chúng 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 chúng ta 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 chúng 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 chúng 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.

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. Chúng 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.

Aggregate

Ba pattern cuối của chương này sẽ đề cập đến một vấn đề khác khi mô hình hóa: 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 chúng ta sẽ đề cập đến ba 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 chúng 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 chúng 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, chúng 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 chúng 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 chúng ta có thể giản lược nó nếu như chỉ cần biết mỗi chiếc xe đều có một động cơ.

Sau khi chúng ta đã tối giản và đơn giản hóa các mối quan hệ giữa các đói 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 bảm 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, chúng 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 chúng 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, chúng 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. Một Aggregate được phân tách với phần còn lại của hệ thống, ngăn cách giữa các đối tượng nội tại và các đối tượng ở ngoài. Mỗi Aggregate có một "*gốc*", đó là một Thực thể và cũng là đối tượng duy nhất có thể truy cập từ phía ngoài của Aggregate. Gốc của Aggregate có thể chứa những tham chiếu đến các đối tượng khác trong Aggregate, và những đối tượng trong này có thể chứa tham chiếu đến nhau, nhưng các

đối tượng ở ngoài chỉ có thể tham chiếu đến gốc. Nếu như trong Aggregate có những Thực thể khác thì định danh của chúng là nội tại, chỉ mang ý nghĩa trong aggregate.

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 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 chúng 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 chúng 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 chúng ta không được phép cấp 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.

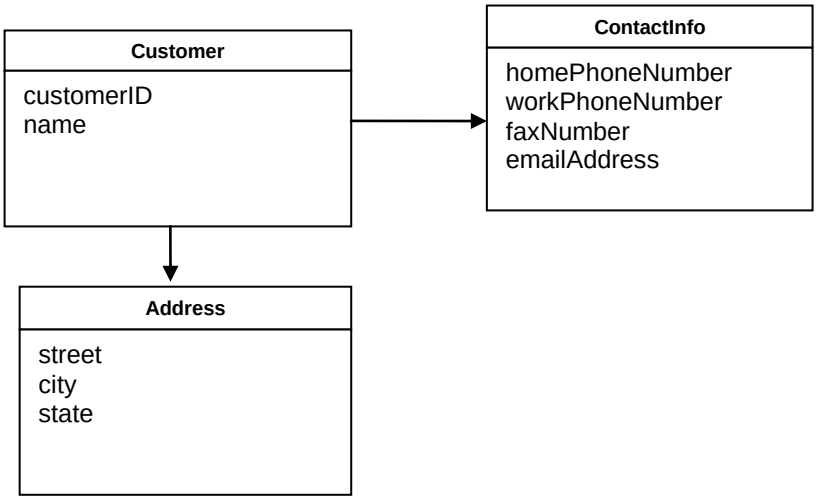
Nếu như các đối tượng của Aggregate được lưu trong CSDL thì chỉ nên cho phép truy vấn trực tiếp lấy gốc của aggregate, các đối tượng nội tại còn lại nên được truy cập thông qua các mối quan hệ bên trong.

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.



Factory

Các Thực thể và đối tượng tập hợp có thể lớn và phức tạp - quá phức tạp để khởi 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 chúng ta được cung cấp nhựa, cao su, kim loại và chúng 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.

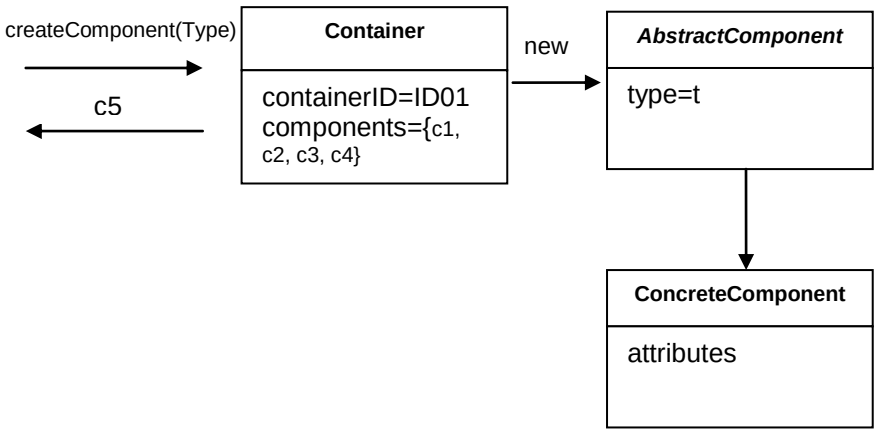
Do đó, ta cần đưa ra một khái niệm mới. Khái niệm này giúp việc gộp quy trình tạo ra đối tượng phức tạp và được gọi là Factory. Factory được dùng để gộp kiến thức cần cho việc tạo đối tượng, và chúng đặc biệt hữu dụng cho Aggregate. Khi gốc của Aggregate được tạo ra, mọi đối tượng chứa trong Aggregate 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 ra ngoại 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 chiếm 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 tốt bất biến của nó.

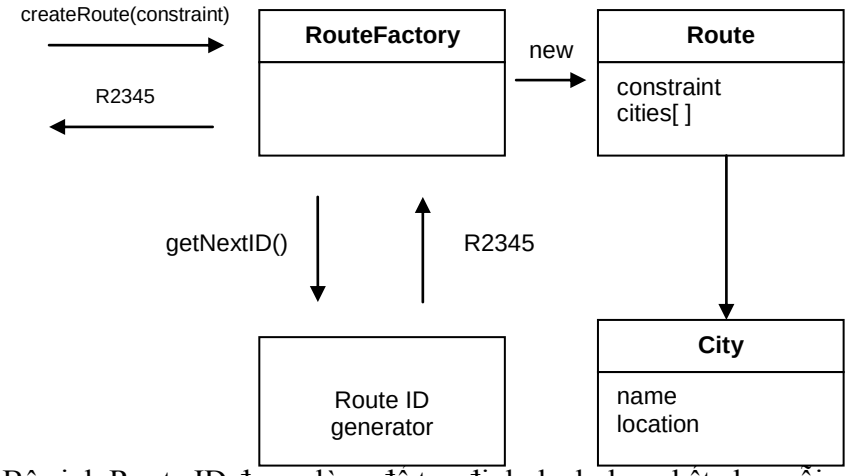
Có nhiều design pattern có thể dùng để thực thi Factory. Cuốn *Design Pattern* của *Gamma et al* mô tả chi tiết những design pattern này và chia pattern này thành 2 loại Factory Phương pháp và Factory Trừu tượng. Chúng ta không thể hiện những pattern này dưới góc độ thiết kế mà thể hiện chúng từ mô hình 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.



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 ẩn 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 site chạy 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.



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, chúng 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, chúng 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. Bạn 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 để 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. Chúng 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.

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 cự đến thiết kế.

CSDL là một phần của lớp hạ tầng. Một giải pháp *bần 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.

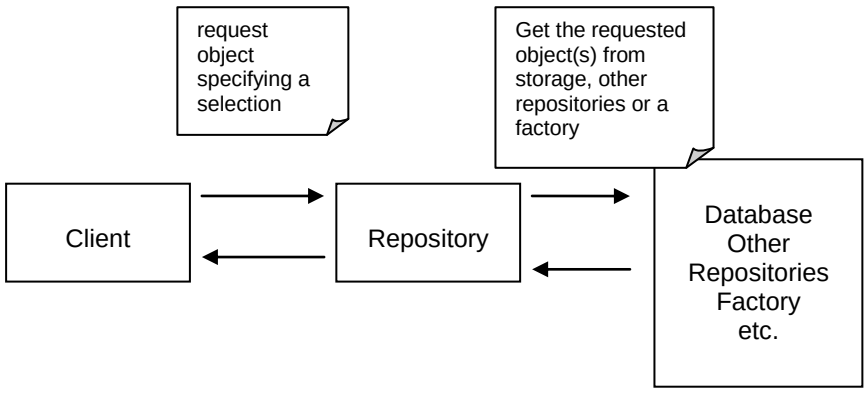
Một client cần một công cụ thiết thực thu thập tham chiếu tới đối tượng domain có từ trước. Nếu lớp infrastructure làm cho nó dễ dàng để làm như vậy, các lập trình viên của phía client có thể

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 thuần túy 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.

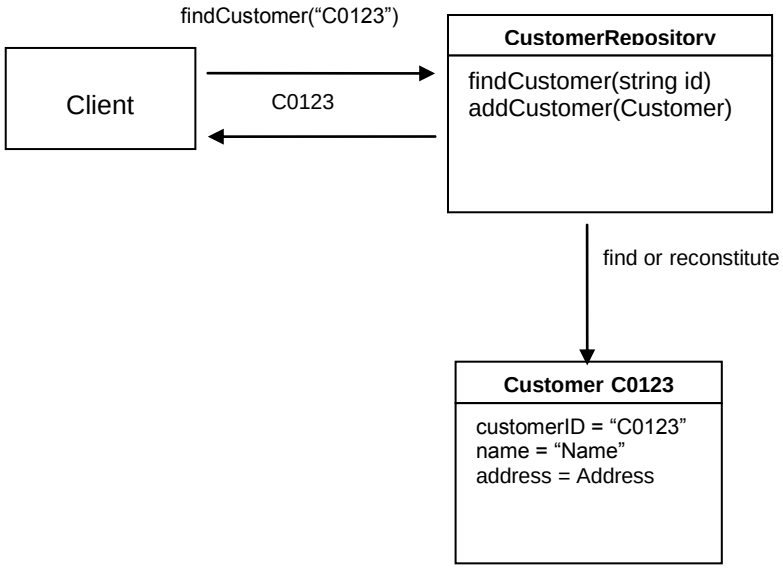


Với mỗi loại đối tượng cần thiết để truy cập toàn cục, tạo một đối tượng có thể cung cấp một tập hợp trong bộ nhớ ảo của tất cả các đối tượng trong các loại đó. Cài đặt truy cập thông qua một Interface được biết ở mức toàn cục. Cung cấp các phương thức để thêm hoặc xóa đối tượng, nhằm đóng gói việc thêm 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 kĩ thuật truy vấn. Chỉ cung cấp

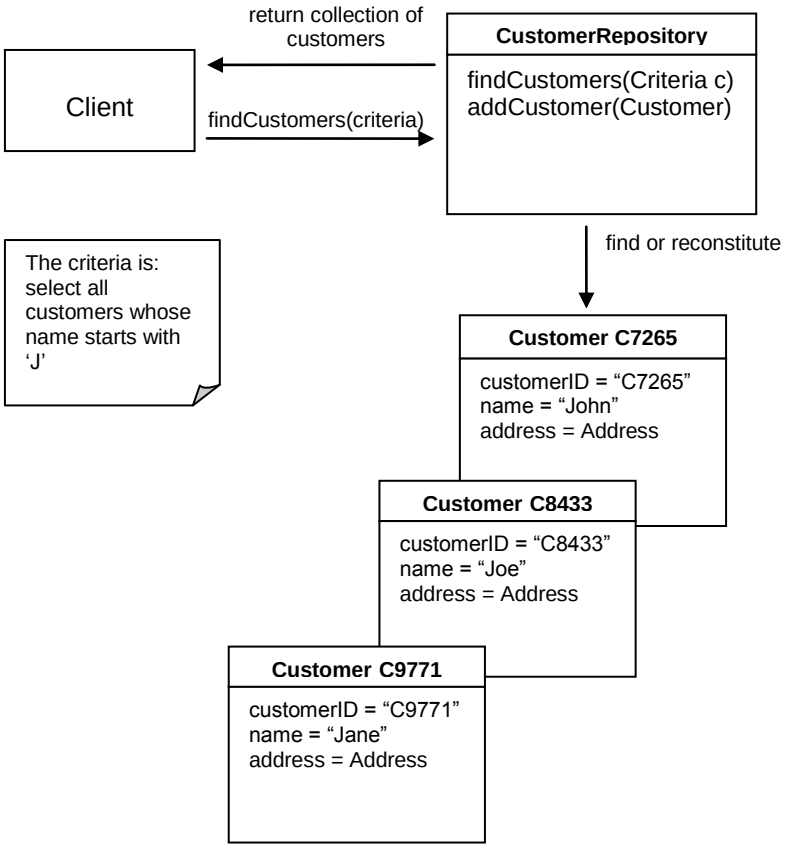
các repository cho Aggregate root mà thực sự cần một truy cập trực tiếp. Giữ client tập trung vào model, ủy quyền tất cả việc lưu trữ đối tượng và truy cập chúng cho một Repository.

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 trí 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.

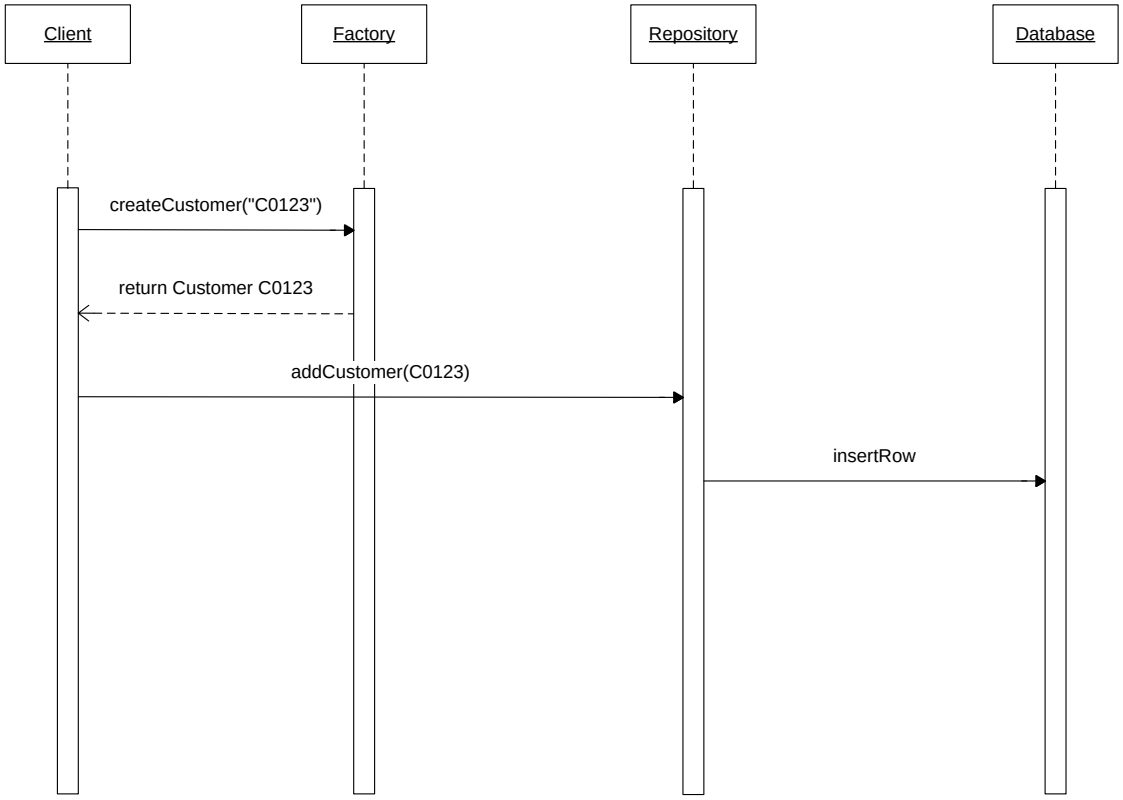
Việc cài đặt một repository có thể rất giống infrastructure, nhưng repository interface sẽ chỉ là một model thuần túy.



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. Chúng 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.

Một cái Nhìn Sâu hơn về Tái cấu

Tái cấu trúc liên tục

Chúng ta đã nói về lĩnh vực, và tầm quan trọng của việc tạo ra một mô hình biểu diễn lĩnh vực. Chúng ta đã đưa ra một số hướng dẫn về các kỹ thuật được sử dụng để tạo ra một mô hình hữu ích. Mô hình này có được liên kết chặt chẽ với các lĩnh vực phái sinh. Chúng ta cũng đã nói rằng thiết kế mã nguồn đã được thực hiện quanh mô hình trên, và tự bản thân mô hình nên được cải tiến dựa trên thiết kế. Thiết kế không theo mô hình có thể dẫn đến việc phần mềm không phù hợp với lĩnh vực liên quan, và có thể hoạt động không tốt. Mô hình không có phản hồi từ thiết kế nhưng không có người phát triển tham gia dẫn tới hậu quả là mô hình không được nắm bắt rõ bởi những người thực thi nó và có thể không phù hợp với các công nghệ được sử dụng.

Trong khi thiết kế và quá trình phát triển, chúng ta cần phải nhìn lại mã nguồn. Đó là thời gian cho tái cấu trúc. Tái cấu trúc là quá trình thiết kế lại mã nguồn, làm cho nó tốt hơn mà không thay đổi chương trình ứng dụng. Tái cấu trúc thường được thực hiện trong phạm vi nhỏ, các bước kiểm soát được để không phá vỡ chức năng hoặc tạo thêm lỗi. Mục đích của tái cấu trúc là làm cho mã nguồn tốt hơn. Các phương pháp Kiểm thử Tự động đóng vai trò quan trọng để đảm bảo chất lượng của tái cấu trúc.

Có rất nhiều cách để tái cấu trúc. Thậm chí có những mẫu tái cấu trúc. Những mẫu như vậy tự động hóa quá trình tái cấu trúc. Có những công cụ được xây dựng trên mô hình như vậy giúp người phát triển dễ dàng hơn nhiều so với trước đây. Kiểu tái cấu trúc này tương tác nhiều hơn với mã nguồn và chất lượng mã nguồn.

Còn có loại tái cấu trúc khác liên quan đến lĩnh vực và mô hình liên quan. Đôi khi cần có cái nhìn mới và rõ ràng hơn về lĩnh vực, hoặc một mối quan hệ giữa hai yếu tố được phát hiện. Tất cả điều đó nên được bao gồm trong các thiết kế thông qua tái cấu trúc. Việc tái cấu trúc rất quan trọng, nó giúp có mã nguồn dễ đọc và dễ hiểu. Bằng việc đọc mã nguồn, ai cũng có thể hiểu chức năng của đoạn mã nguồn đó là gì, và lý do tại sao phải có nó. Chỉ khi đó mã nguồn mới thực sự nắm bắt được bản chất của mô hình.

Tái cấu trúc dựa trên mẫu, có thể được tổ chức và cấu trúc. Tái cấu trúc sâu hơn không thể được thực hiện theo cách như trên. Chúng ta không thể tạo ra các mẫu cho nó. Sự phức tạp của một mô

hình và sự đa dạng của chúng không cung cấp cho chúng ta khả năng tiếp cận mô hình một cách cơ học. Một mô hình tốt là kết quả của suy nghĩ sâu sắc, hiểu biết, kinh nghiệm và sự tinh tế.

Một trong những điều đầu tiên chúng ta được dạy về mô hình hóa là đọc các đặc tả kinh doanh và tìm kiếm các danh từ và động từ. Các *danh từ được chuyển thành class*, trong khi các *động từ trở thành method*. Đây cách làm đơn giản, và sẽ dẫn đến một mô hình hời hợt. Vì tất cả mô hình thiếu chiều sâu từ ban đầu, nên chúng ta cần tái cấu trúc lại mô hình để có cái nhìn sâu sắc hơn.

Các thiết kế phải linh hoạt. Một thiết kế cứng nhắc không có khả năng tái cấu trúc. Mã nguồn không được xây dựng linh hoạt là những đoạn mã nguồn khó bạn sẽ phải mất nhiều thời gian để vật lộn, sửa đổi nó.

Sử dụng các nguyên tắc cơ bản cùng với ngôn ngữ thống nhất giúp quá trình phát triển rõ ràng hơn, giúp ta tập trung hơn vào tìm mô hình tối ưu, nắm bắt tinh tế hơn những vấn đề của lĩnh vực, hướng tới thiết kế thực tế hơn. Một mô hình nắm bắt được những điều cần thiết từ hiện tượng bề ngoài là một mô hình toàn diện. Điều này sẽ làm cho các phần mềm phù hợp hơn với cách chuyên gia lĩnh vực tư duy và đáp ứng tốt hơn nhu cầu của người dùng.

Theo truyền thống, tái cấu trúc được mô tả trong việc biến đổi mã nguồn dưới góc độ kỹ thuật. Tái cấu trúc cũng có thể được thúc đẩy bởi cách nhìn sâu sắc về lĩnh vực, mô hình hoặc biểu hiện của nó trong mã nguồn.

Các mô hình lĩnh vực phức tạp hiếm được phát triển ngoại trừ thông qua một quá trình lặp đi lặp lại của tái cấu trúc, trong đó có sự tham gia chặt chẽ của các chuyên gia lĩnh vực với các nhà phát triển quan tâm tìm hiểu về lĩnh vực đó.

Giới thiệu các Khái niệm

Tái cấu trúc được thực hiện từng bước nhỏ. Kết quả này cũng là một loạt các cải tiến nhỏ. Có những lúc rất nhiều thay đổi nhỏ tạo ra một sự thay đổi lớn mang tính đột phá.

Chúng ta bắt đầu với một mô hình thô. Sau đó tinh chỉnh nó và thiết kế dựa trên kiến thức sâu sắc hơn về lĩnh vực và các mối quan tâm. Chúng ta thêm khái niệm mới và trừu tượng hóa mới. Các

thiết kế này sau đó được tái cấu trúc. Mỗi bước tinh chỉnh làm cho thiết kế rõ ràng hơn. Điều này về sau tạo ra cơ sở cho đột phá.

Một bước đột phá thường liên quan đến một sự thay đổi trong suy nghĩ, trong cách chúng ta nhìn thấy các mô hình. Tuy là một bước tiến dài trong dự án, nhưng nó cũng có một số nhược điểm. Một bước đột phá có thể gồm nhiều sự tái cấu trúc. Điều đó có nghĩa là ta cần thời gian và nguồn lực - những thứ mà chúng ta dường như không bao giờ có đủ. Nó cũng tiềm tàng nguy cơ lỗi do nhiều tái cấu trúc có thể dẫn tới những thay đổi hành vi trong ứng dụng.

Để đạt được một bước đột phá, chúng ta cần phải đưa ra các khái niệm rõ ràng. Khi chúng ta nói chuyện với các chuyên gia lĩnh vực, chúng ta trao đổi rất nhiều ý tưởng và kiến thức. Một số khái niệm tạo nên Ngôn ngữ chung, nhưng một số còn chưa được chú ý ngay từ đầu. Chúng là những khái niệm ẩn, được sử dụng để giải thích các khái niệm đã có trong mô hình. Trong quá trình tinh chỉnh thiết kế, một số những khái niệm ẩn bắt đầu được chú ý. Chúng ta phát hiện ra rằng một số trong đó đóng vai trò quan trọng trong việc thiết kế ở thời điểm chúng ta nên làm cho các khái niệm tương ứng trở nên rõ ràng hơn. Chúng ta nên tạo các lớp và các mối quan hệ với giữa chúng. Khi điều đó xảy ra, chúng ta có thể có cơ hội của một đột phá.

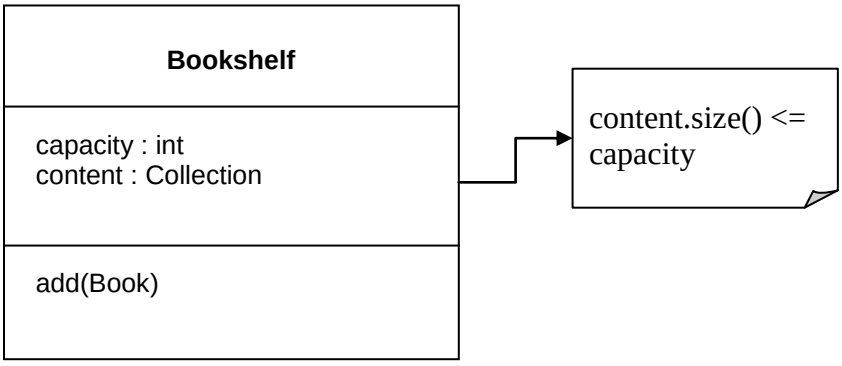
Không nên có khái niệm ẩn như vậy. Nếu chúng là những khái niệm lĩnh vực, chúng nên có mặt trong mô hình và thiết kế. Làm thế nào để chúng ta nhận ra chúng? Cách đầu tiên để khám phá các khái niệm ẩn là lắng nghe ngôn ngữ. Các ngôn ngữ chúng ta đang sử dụng trong quá trình mô hình hóa và thiết kế có chứa rất nhiều thông tin về lĩnh vực. Lúc đầu có thể không quá nhiều, hoặc một số thông tin có thể không được sử dụng một cách chính xác. Một số khái niệm có thể không được hiểu đầy đủ, hoặc thậm chí hoàn toàn hiểu sai. Đây là một phần của việc học một lĩnh vực mới. Nhưng khi chúng ta xây dựng Ngôn ngữ chung, các khái niệm chính được định nghĩa dần. Đó là nơi mà chúng ta nên bắt đầu tìm các khái niệm ẩn.

Đôi khi các phần của thiết kế có thể không rõ ràng. Có các mối quan hệ khiến cách tính toán khó mà theo được, hoặc các thủ tục phức tạp, khó hiểu. Đây là sự lúng túng trong việc thiết kế. Đây là một nơi tốt để tìm các khái niệm ẩn. Có lẽ là một cái gì đó là đang thiếu. Nếu một khái niệm quan trọng chưa tồn tại, những người khác sẽ phải thay thế chức năng của nó. Điều này sẽ làm một số đối tượng trở nên cồng kềnh ảnh hưởng tới sự rõ ràng của việc thiết kế. Hãy thử tìm khái niệm còn thiếu. Nếu tìm thấy, hãy làm cho nó rõ ràng hơn. Tái cấu trúc các thiết kế để làm cho nó đơn giản hơn.

Khi xây dựng kiến thức, ta có thể gặp mâu thuẫn. Các chuyên gia lĩnh vực đôi khi có cái nhìn khác nhau về cùng một lĩnh vực. Một đặc tả có thể có mâu thuẫn với một đặc tả khác. Một số trong những mâu thuẫn này không thực sự mâu thuẫn, nó có thể là những cách nhìn khác nhau về sự vật hiện tượng, hoặc đơn giản là thiếu chính xác trong lời giải thích. Chúng ta nên cố gắng hoà giải mâu thuẫn. Đôi khi điều này làm sáng tỏ những khái niệm quan trọng. Thậm chí nếu nó không, nó vẫn rất quan trọng để giữ cho mọi thứ rõ ràng.

Một cách rõ ràng về việc tạo ra khái niệm mô hình là sử dụng tài liệu về lĩnh vực. Có những cuốn sách được viết trên hầu như bất kỳ chủ đề nào. Chúng chứa rất nhiều kiến thức về các lĩnh vực tương ứng. Những cuốn sách thường không chứa mô hình cho các lĩnh vực mà chúng nói đến. Các thông tin đó cần phải được xử lý, tinh chế. Tuy nhiên, các thông tin tìm thấy trong cuốn sách có giá trị, và cung cấp một cái nhìn sâu sắc về lĩnh vực.

Có khái niệm khác đó là rất hữu ích khi thực hiện rõ ràng: Ràng buộc (Constraint), Qui trình (Process) và Đặc tả (Specification). Một ràng buộc là một cách đơn giản để thể hiện cái gì đó bất biến. Dù xảy ra với dữ liệu đối tượng, ràng buộc được bảo toàn. Điều này được thực hiện bằng cách đặt logic không thay đổi vào ràng buộc. Sau đây là một ví dụ đơn giản để giải thích các khái niệm nói trên.



Chúng ta có thể thêm sách vào giá sách, nhưng không bao giờ vượt quá khả năng chứa của nó. Điều này có thể được xem như là một phần của hành vi Bookshelf, như trong đoạn mã Java dưới đây.

```
public class Bookshelf {
	private int capacity = 20;
	private Collection content;
```

```
	public void add(Book book) {
		if (content.size() + 1 <= capacity) {
			content.add(book);
		} else {
			throw new IllegalOperationException("The bookshelf has reached its
limit.");
		}
	}
}
```

Chúng ta có thể tái cấu trúc, tạo các ràng buộc trong một method riêng biệt:

```
public class Bookshelf {
	private int capacity = 20;
	private Collection content;
	public void add(Book book) {
		if (isSpaceAvailable()) {
			content.add(book);
		} else {
			throw new IllegalOperationException("The bookshelf has reached its
			limit.");
		}
	}
	private boolean isSpaceAvailable() {
		return content.size() < capacity;
	}
}
```

Đặt ràng buộc thành một method riêng biệt có lợi thế là làm cho nó rõ ràng. Nó rất dễ dàng để đọc và mọi người sẽ nhận thấy rằng method add() chịu ràng buộc này. Ngoài ra còn có chỗ thêm logic cho các method nếu các ràng buộc trở nên phức tạp hơn.

Qui trình này thường được thể hiện trong mã nguồn với các thủ tục. Chúng ta sẽ không sử dụng cách tiếp cận thủ tục khi có ngôn ngữ hướng đối tượng, vì vậy chúng ta cần xác định đối tượng và các hành vi liên quan. Cách tốt nhất để thực hiện quá trình này là sử dụng Service. Nếu có những cách khác nhau để thực hiện quá trình này, sau đó chúng ta có thể đóng gói các thuật toán trong một đối tượng và sử dụng Strategy. Không phải tất cả các quá trình cần phải được làm rõ nếu như Ngôn ngữ chung không đề cập cụ thể quá trình tương ứng.

Phương pháp cuối cùng để làm cho khái niệm rõ ràng là sử dụng Specification. Nói đơn giản, đặc tả được sử dụng để kiểm tra một đối tượng để xem nếu nó thỏa mãn một tiêu chuẩn nhất định.

Các lớp domain chứa các quy tắc kinh doanh được áp dụng cho các thực thể và giá trị. Những quy tắc thường được kết hợp vào các đối tượng. Một số các quy tắc chỉ là một tập hợp các câu hỏi "có" hoặc "không". Quy định như vậy có thể được thể hiện qua một loạt các phép toán logic với các giá

trị Boolean, và kết quả cuối cùng cũng là một Boolean. Một ví dụ là bài kiểm tra thực hiện trên một đối tượng Customer để xem nó có đủ điều kiện vay một khoản tín dụng nào đó. Quy tắc có thể được thể hiện như là một method với tên isEligible(), và có thể được gắn vào đối tượng Customer. Nhưng nguyên tắc này không phải là một phương pháp đơn giản chỉ hoạt động đúng trên dữ liệu Customer. Đánh giá các quy tắc liên quan đến việc xác minh các thông tin của khách hàng, kiểm tra để xem nếu anh ta đã trả nợ của mình trong quá khứ hay chưa, kiểm tra số dư tài khoản... Các qui trình kinh doanh này có thể là lớn và phức tạp, làm các đối tượng "phình to" đến mức nó không có còn phục vụ mục đích ban đầu nữa. Tại thời điểm này, chúng ta có thể muốn để di chuyển toàn bộ qui trình lên mức ứng dụng, vì có vẻ như nó vượt quá mức lĩnh vực. Trên thực tế, đó là lúc cần tái cấu trúc.

Các quy tắc nên được đóng gói vào một đối tượng mà sau trở thành đặc tả của khách hàng, và nên được giữ trong lớp domain. Các đối tượng mới sẽ chứa một loạt các phương pháp Boolean để kiểm tra xem nếu một đối tượng Customer phải hội đủ điều kiện cho vay tín dụng hay không. Mỗi phương pháp đóng vai trò của một kiểm tra nhỏ, và tất cả các phương pháp kết hợp cung cấp cho câu trả lời cho câu hỏi ban đầu. Nếu các quy tắc kinh doanh không bao gồm trong một đặc tả, mã nguồn tương ứng sẽ không nhất quán.

Các đặc tả được sử dụng để kiểm tra đối tượng xem nếu chúng đáp ứng cho một số nhu cầu, hoặc sẵn sàng cho một số mục đích nào đó. Nó cũng có thể được sử dụng để chọn một đối tượng nhất định từ một tập hợp, hoặc như là một điều kiện trong việc tạo ra một đối tượng.

Thông thường, một Specification kiểm tra tính hợp lệ của một luật, và sau đó các Specification được kết hợp lại để biểu diễn luật tổng hợp:

```
Customer customer = customerRepository.findCustomer(customerIdentity);

…

Specification customerEligibleForRefund = new Specification(

new CustomerPaidHisDebtsInThePast(),

new CustomerHasNoOutstandingBalances());

if (customerEligibleForRefund.isSatisfiedBy(customer) {
      refundService.issueRefundTo(customer);

}
```

Kiểm tra các luật đơn giản hơn rõ ràng là đơn giản hơn, và mã nguồn đã tự diễn giải khi khách hàng hội tụ đủ điều kiện để được hoàn tiền.

Duy trì Tính Toàn vẹn của Mô hình

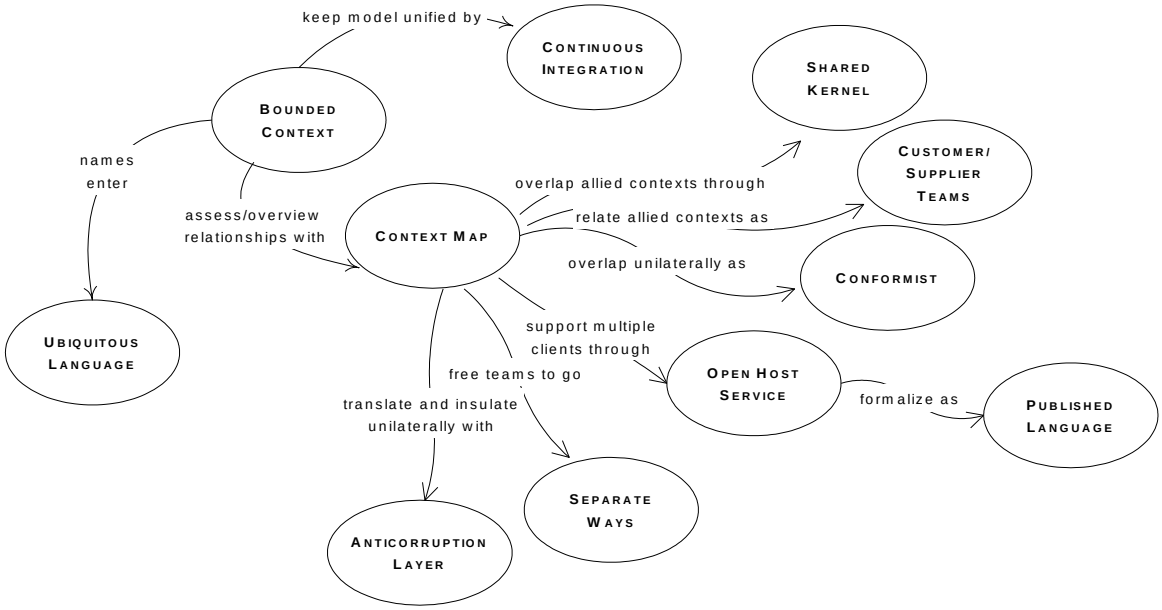
Chương này đề cập tới những dự án lớn, đòi hỏi sự tham gia của nhiều nhóm. Chúng ta gặp nhiều loại thách thức khác nhau khi có nhiều nhóm, được quản lý và phối hợp theo cách khác nhau. Những dự án cho doanh nghiệp thường là những dự án lớn, đòi hỏi nhiều loại công nghệ và nguồn lực. Thiết kế của những dự án ấy vẫn thường dựa trên mô hình domain và chúng ta cần đo đạc thích hợp để đảm bảo sự thành công của dự án.

Khi nhiều nhóm làm cùng một dự án, họ viết mã nguồn song song, mỗi nhóm được chỉ định làm một phần của mô hình. Những phần này không độc lập, nhưng ít liên hệ chéo. Họ đều bắt đầu từ một mô hình lớn và chia nhỏ ra để thực thi. Giả sử rằng một nhóm nào đó đã tạo ra một mô-đun và họ mở cho các nhóm khác dùng. Một lập trình viên từ nhóm khác dùng mô-đun này và phát hiện ra rằng mô-đun nó thiếu một số chức năng anh ta cần. Anh ta thêm những chức năng nó và check-in mã nguồn để mọi người đều có thể dùng. Điều anh ta không ngờ đến là việc anh ta làm thay đổi cả mô hình, và có thể rằng thay đổi này sẽ làm hỏng tính năng của ứng dụng. Điều này rất dễ xảy ra, không ai có thời gian để hiểu cả mô hình. Mọi người đều có sân chơi riêng của mình và những "*sân riêng*" này không đủ rõ với người khác.

Sẽ là dễ dàng nếu ta bắt đầu từ một mô hình tốt và sau đó biến thành các mô hình không nhất quán. Yêu cầu đầu tiên của mô hình là phải nhất quán, với những điều khoản không bất biến và không có mâu thuẫn. Sự nhất quán nội tại của một mô hình được gọi là *sự thống nhất*. Một dự án doanh nghiệp có thể có một mô hình phủ mọi domain của doanh nghiệp, không có mâu thuẫn và không có điều khoản chồng chéo. Một mô hình doanh nghiệp thống nhất lý tưởng không dễ đạt được và đôi khi không đáng giá để làm thử. Những dự án như thế cần nỗ lực tổng hợp của nhiều nhóm. Các nhóm này cần độ độc lập cao trong quy trình phát triển vì họ không có thời gian gặp nhau thường xuyên để trao đổi về thiết kế. Việc điều phối những nhóm như thế là một nhiệm vụ khó khăn. Họ có thể thuộc về nhiều phòng ban khác nhau với cách quản lý khác nhau. Khi thiết kế của mô hình tiến hóa từng phần một cách độc lập, chúng ta đối mặt với rủi ro mất tính nhất quán của mô hình. Duy trì tính nhất quán của mô hình bằng việc cố gắng giữ một mô hình thống nhất lớn cho toàn dự án doanh nghiệp sẽ không thể hoạt động được. Giải pháp không dễ vì nó đi ngược với tất cả những gì chúng ta đã tìm hiểu tới nay. Thay vì giữ một mô hình lớn nhưng sẽ bị phân nhỏ về sau này, chúng ta nên chia một cách có ý thức mô hình thành nhiều mô hình khác. Những mô hình này sẽ tích hợp và tiến hóa độc lập nếu

chúng tuân thủ giao ước của chúng. Mỗi mô hình cần có một ranh giới rõ ràng, và quan hệ giữa các mô hình phải được định nghĩa chính xác.

Chúng ta sẽ trình bày một số kỹ thuật dùng để duy trì tính nhất quán của mô hình. Hình sau đây là những kỹ thuật và quan hệ giữa chúng.



Ngữ cảnh Giới hạn

Mỗi mô hình đều có một ngữ cảnh tương ứng với nó. Chúng ta làm việc với một mô hình, ngữ cảnh là ẩn. Chúng ta không cần định nghĩa chúng. Khi chúng 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 chúng ta làm việc với một ứng dụng doanh nghiệp lớn, chúng 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 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.

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 chúng ta hạn chế trong một mảng cụ thể. Định nghĩ một cách hiển minh ngữ cảnh trong ngữ cảnh giới hạn. Hãy xác định hiển minh 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. Môt ngữ cảnh giới hạn cung cấp khung logic bên trong của mô hình. Mô-đun tổ chức các thành phần của mô hình, đo đó ngữ cảnh giới hạn chứa mô-đun.

Khi nhiều nhóm làm việc trên cùng một mô hình, chúng ta phải cẩn thận sao cho chúng không dẫm chân lên nhau. Chúng ta phải luôn ý thức tới sự thay đổi có thể ảnh hưởng đến chức năng. Khi dùng nhiều mô hình, mọi người phải làm việc tự do trên mảng của họ. Chúng ta đều biết hạn chế trong mô hình của chúng ta và không bước ra ngoài giới hạn. Chúng ta phải đảm bảo giữ được mô hình tinh khiết, nhất quán và thống nhất. Mỗi mô hình cần hỗ trợ việc tái cấu trúc dễ hơn, không gây ảnh hưởng tới mô hình khác. Thiết kế có thể được làm mịn và chắt lọc để đạt được sự tinh khiết tối đa.

Việc có nhiều mô hình có cái giá phải trả. Chúng 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. Chúng ta không thể chuyển đối tượng bất kỳ giữa các mô hình khác nhau và chúng 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 chúng 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 chúng 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 bố sự kiện khi có đơn đặt hàng vì ai đó cần phải gửi thư có hạng mục đã được yêu cầu. Chúng 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, chúng ta bắt đầu bằng một mô hình phủ cả domain thương mại điện tử. Chúng ta muốn làm thế vì sau chúng, chúng ta cũng sẽ cần làm một chương trình lớn. Tuy nhiên, chúng 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 chúng ta chỉ cần đảm bảo giao diện giữa chúng hoạt động tốt.

Tích hợp Liên tục

Khi một Ngữ cảnh Giới hạn đã được định nghĩa, chúng ta cần đảm bảo rằng nó luôn mới và hoạt động như kỳ vọng. Khi nhiều người cùng làm việc trên cùng một Ngữ cảnh Giới hạn, mô hình có khuynh hướng bị phân mảnh. Nhóm càng lớn, vấn đề càng lớn. Nhóm nhỏ ba, bốn người cũng có thể gặp vấn đề nghiêm trọng.

Tuy nhiên, nếu ta chia nhỏ hệ thống thành những ngữ cảnh rất-rất nhỏ thì thực ra, ta đánh mất mức độ giá trị về tích hợp và tính tương liên.

Ngay cả khi nhóm làm việc cùng trên một Ngữ cảnh Giới hạn thì vẫn có thể có lỗi. Chúng ta cần giao tiếp trong nội bộ nhóm để đảm bảo rằng mọi người đều hiểu vai trò của từng phần tử trong mô hình. Nếu có người không hiểu quan hệ giữa các đối tượng, họ có thể sửa mã nguồn làm nó mâu thuẫn với mô hình ban đầu. Lỗi dễ xảy ra và chúng ta không thể đảm bảo 100% tập trung vào sự tinh khiết của mô hình. Một thành viên của nhóm có thể thêm mã nguồn trùng với mã đã có mà không biết, hay có thể lặp lại mã nguồn mà không hề thay đổi mã hiện tại vì sợ rằng có thể làm hỏng chức năng đang đang có.

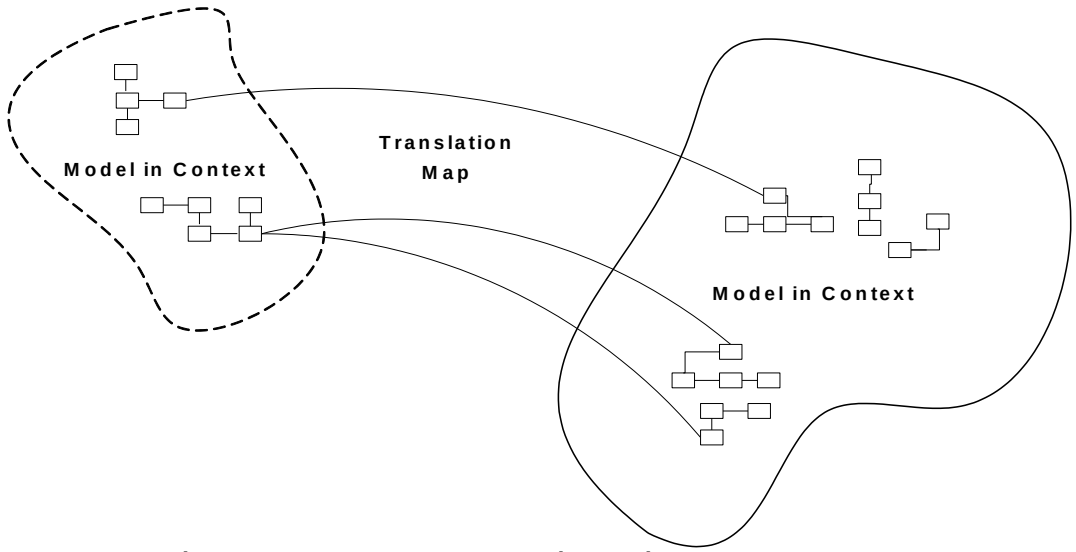
Không thể định nghĩa đầy đủ một mô hình ngay từ đầu. Mô hình được tạo ra, tiến hóa liên tục dựa trên thực tế của domain và phản hồi từ quy trình phát triển. Điều này nghĩa là khái niệm mới có thể xuất hiện trong mô hình, phần tử mới được thêm vào mã nguồn. Mọi nhu cầu kiểu này được tích hợp vào một mô hình thống nhất, được thực thi tương ứng trong mã nguồn. Đó là lý do tại sao Tích hợp Liên tục là quy trình cần thiết trong Ngữ cảnh Giới hạn. Chúng ta cần quy trình tích hợp để đảm bảo rằng mọi phần tử được thêm vào một cách hài hòa trong toàn bộ phần còn lại của mô hình và được thực thi đúng trong mã nguồn. Chúng ta cần có một thủ tục khi merge mã nguồn. Merge mã nguồn càng sớm càng tốt. Với một nhóm nhỏ, nên merge hàng ngày. Chúng ta cũng cần xây dựng quy

trình. Mã nguồn đã được merge cần được build tự động và được kiểm thử. Một yêu cầu cần thiết khác là kiểm thử tự động. Nếu nhóm có công cụ kiểm thử và tạo ra một bộ kiểm thử thì việc kiểm thử có thể chạy mỗi khi build để để đưa ra mọi cảnh báo khi có lỗi. Sửa mã nguồn để fix lỗi đã được báo cáo là dễ vì chúng ta phát hiện sớm. Sau đó chúng ta tiếp tục quy trình với các bước: merge, build, kiểm thử tiếp.

Tích hợp liên tục dựa trên khái niệm tích hợp của mô hình, tìm cách thực thi những chỗ được kiểm thử. Bất kỳ sự không nhất quán nào trong mô hình có thể được phát hiện trong quá trình thực thi. Tích hợp liên tục áp dụng vào một Ngữ cảnh Giới hạn và nó được dùng để đối phó với quan hệ giữa các Ngữ cảnh xung quanh.

Ngữ cảnh Ánh xạ

Một phần mềm doanh nghiệp có nhiều mô hình, mỗi mô hình có Ngữ cảnh giới hạn riêng. Bạn nên dùng ngữ cảnh như là cơ sở để tổ chức nhóm. Người trong cùng nhóm sẽ giao tiếp dễ hơn, tích hợp công việc và thực thi công việc dễ hơn. Khi mọi nhóm làm việc với mô hình của riêng họ, mọi người cần hiểu bức tranh tổng thể. Một Ánh xạ Ngữ cảnh mô tả khái quát các Ngữ cảnh Giới hạn và quan hệ giữa chúng. Một Ánh xạ Ngữ cảnh có thể là một giản đồ như hình dưới đây, hoặc là tài liệu viết. Mức độ chi tiết của nó tùy trường hợp. Điều quan trọng là mọi người làm việc trên cùng một dự án phải hiểu và cùng chia sẻ bức tranh này.

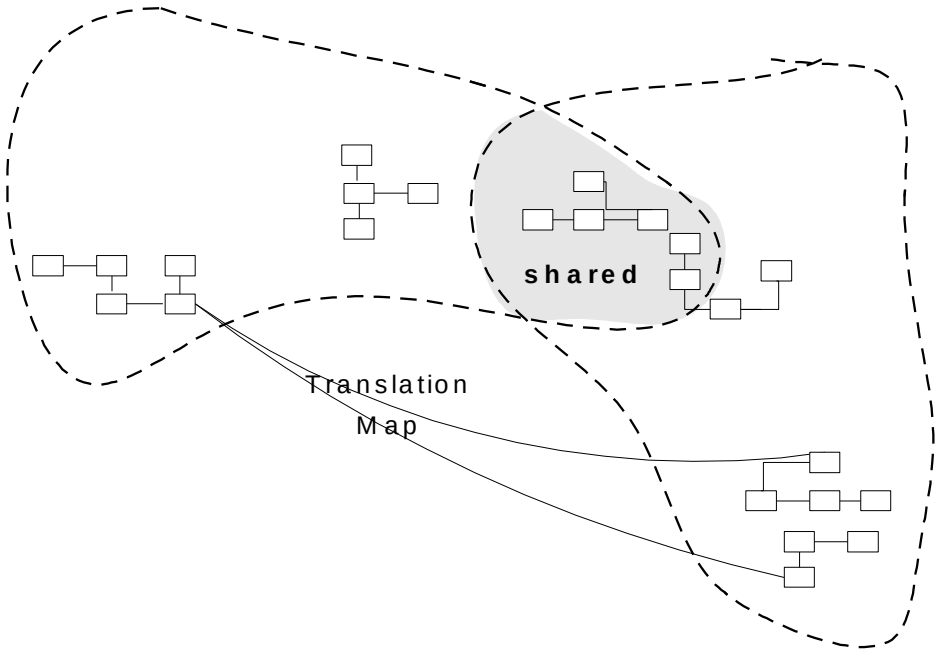


Sẽ là không đủ nếu phải tách riêng mô hình thống nhất. Chúng được tích hợp lại vì chức năng của mỗi mô hình chỉ là một phần của hệ thống. Cuối cùng mỗi mảnh ghép đó gộp lại và cả hệ thống phải chạy đúng. Nếu ngữ cảnh không được định nghĩa rõ ràng, có thể là chúng sẽ dẫm chân lên nhau. Nếu quan hệ giữa các ngữ cảnh không được chỉ ra, hệ thống có thể sẽ không chạy khi tích hợp.

Mỗi Ngữ cảnh Giới hạn cần có tên, và tên này là một phần của Ngôn ngữ Chung. Nó sẽ giúp nhiều cho nhóm trao đổi về cả hệ thống. Mọi người cần biết ranh giới giữa các ngữ cảnh và ánh xạ giữa ngữ cảnh và mã nguồn. Một thực hành chung thường thấy là định nghĩa ngữ cảnh, tạo mô-đun cho từng ngữ cảnh, dùng quy định đặt tên để thể hiện rằng mô-đun nào thuộc ngữ cảnh nào.

Trong những trang tiếp theo chúng ta sẽ nói về sự tương tác giữa nhiều ngữ cảnh. Chúng ta trình bày một chuỗi các mẫu có thể dùng để tạo Ánh xạ ngữ cảnh khi ngữ cảnh có vài trò rõ ràng và quan hệ chúng đã được chỉ ra. Nhân chung và Khách hàng - Nhà cung cấp là các mẫu có độ tương tác cao giữa các ngữ cảnh. *Separate Ways* là một mẫu dùng khi chúng ta muốn ngữ cảnh có tính độc lập cao và tiến hóa riêng biệt. Có hai mẫu khác liên quan đến sự tương tác giữa hệ thống và một hệ thống cũ hoặc một hệ thống ngoài là Open Host Service và Lớp Chống đổ vỡ.

Nhân Chung



Khi việc tích hợp chức năng bị hạn chế, chi phí cho Tích hợp Liên tục có thể tăng cao. Điều này đặc biệt đúng khi nhóm không đủ kỹ năng hay yếu tố chính trị của tổ chức để duy trì tích hợp liên tục, hoặc khi cỡ của một nhóm quá lớn và không thể hành động được. Do đó việc phân chia Ngữ cảnh Giới hạn lúc này mới tách và phân chia thành nhiều nhóm.

Nhiều nhóm không gắn kết cùng làm trên một phần mềm có liên kết chặt chẽ có thể chạy độc lập, nhưng kết quả họ làm ra có thể không ăn khớp. Họ sẽ cần nhiều thời gian để dịch

lớpvà chỉnh lại hơn là dành thời gian cho việc Tích hợp Liên tục ngay từ ban đầu, tức là trong lúc họ làm việc ấy, họ tốn công vào việc người khác đã làm và đánh vật với lợi ích của Ngôn ngữ Chung.

Do đó, hãy chỉ định một vài tập con của mô hình domain mà các nhóm đồng ý chia sẻ. Tất nhiên, việc này bao gồm, đi cùng với tập con của mô hình, là tập con của mã nguồn và thiết kế CSDL đi cùng với phần tương ứng của mô hình. Rõ ràng, việc làm này chia sẻ những cái có trạng thái đặc biệt và không thể được thay đổi nếu không có sự tham vấn của nhóm khác.

Hãy tích hợp một hệ thống chức năng thường xuyên, nhưng ít thường xuyên hơn nhịp của Tích hợp Liên tục trong nội bộ nhóm. Trong những phiên tích hợp này, hãy chạy kiểm thử cho cả hai nhóm.

Mục đích của Nhân Chung là giảm sự lặp lại, nhưng vẫn giữ riêng hai ngữ cảnh riêng biệt. Phát triển trên một Nhân Chung đòi hỏi độ chú ý cao hơn. Cả hai nhóm có thể thay đổi mã nguồn của nhân, và họ phải tích hợp các thay đổi. Nếu các nhóm chia riêng bản copy của mã nguồn nhân, họ phải merge mã nguồn lại càng sớm càng tốt, ít nhất là một tuần một lần. Một bộ kiểm thử là cần thiết để mọi thay đổi tới nhân cần được báo cho các nhóm khác để họ biết khi có chức năng mới.

Khách hàng - Nhà cung cấp[6]

Nhiều trường hợp khi hai hệ thống con có quan hệ đặc biệt: Một hệ thống phụ thuộc rất nhiều vào hệ thống còn lại. Ngữ cảnh khi hai hệ thống con tồn tại độc lập, và kết quả xử lý của một hệ thống được truyền đến hệ thống kia. Chúng không có Nhân chung vì chúng có thể không thể có nhân chung đúng khái niệm, hoặc không thể tạo được mã nguồn chia sẻ chung cho hai hệ thống con vì lý do kỹ thuật nào đó. Trường hợp này, ta gọi hai hệ thống con là có quan hệ Khách hàng - Nhà cung cấp.

Chúng ta hãy quay lại ví dụ trước. Ở đó chúng ta đã nói về mô hình liên quan tới chương trình thương mại điện tử bao gồm chức năng báo cáo và truyền thông điệp. Chúng ta đã nói rằng sẽ tốt hơn nếu tạo riêng mô hình cho những ngữ cảnh như thế này vì một mô hình đơn lẻ sẽ trở thành một nút thắt cổ chai liên tục và là nguyên nhân cãi vã trong quy trình phát triển. Giả sử rằng chúng ta thống nhất có nhiều mô hình, khi đó quan hệ giữa hệ thống con web shopping và hệ thống báo cáo là gì? Nhân chung không phải là giải pháp tốt cho trường hợp này. Hệ thống con chắc sẽ dùng kỹ thuật thực thi khác. Cách thứ nhất là chỉ dùng trình duyệt, cách thứ hai có thể là dùng chương trình giàu GUI. Ngay cả khi chương trình báo cáo xong, sử dụng giao diện web thì khái niệm chính của mô hình tương

[6] Customer-Supplier

ứng cũng khác. Ở đây có thể có sự chồng chéo nhưng không đủ kết luận gì về Nhân chung. Do đó chúng ta chọn một con đường khác. Mặt khác, hệ thống con e-shopping không phụ thuộc vào phần báo cáo. Người dùng chương trình e-shopping là khách hàng web, họ xem sản phẩm trên web và đặt hàng. Mọi dữ liệu khách hàng, hàng và đơn đặt hàng đều đặt trong CSDL. Chỉ có vậy. Chương trình e-shopping không cần quan tâm tới gì xảy ra với dữ liệu tương ứng. Trong khi đó, chương trình báo cáo lại rất quan tâm và cần dữ liệu lưu bởi chương trình e-shopping. Cũng cần một số thông tin bổ sung để tạo ra báo cáo. Khách hàng cho hàng vào rỏ, hủy hàng đó trước khi "check out" (mua hàng). Khách hàng có thể xem một vài liên kết khác. những thông tin này không có ý nghĩa với chương trình e-shopping, nhưng nó có thể có ý nghĩa với chương trình báo cáo. Tiếp theo đó, hệ thống con supplier cần thực thi một số đặc tả cần cho hệ thống con đó. Đây là một kết nối giữa hai hệ thống.

Một yêu cầu khác liên quan là dữ liệu được sử dụng, nói chính xác hơn là schema. Cả hai chương trình đều thay đổi cùng một CSDL. Nếu hệ thống cho e-shopping chỉ là một hệ thống con duy nhất truy cập tới CSDL, thì schema của cở sở dữ liệu có thể thay đổi bất kỳ lúc nào để phù hợp yêu cầu. Nhưng hệ thống con báo cáo cũng cần truy cập vào CSDL, do đó cần sự ổn định của schema. Không có trường hợp schema CSDL không thay đổi bất kỳ gì trong quá trình phát triển. Nó không thể hiện một vấn đề của chương trình e-shopping, mà nó cũng là vấn đề của cả hệ thống con báo cáo. Hai nhóm cần trao đổi về việc làm chung trên CSDL, quyết định khi nào cần thay đổi. Điều này là một hạn chế của hệ thống báo cáo bởi vì nhóm có thể nhanh chóng cần thay đổi để phát triển tiếp thay vì đợi chương trình e-shopping. Nếu nhóm e-shopping phủ quyết, họ có thể sẽ gây ảnh hưởng tới sự thay đổi của CSDL, gây ảnh hưởng tới hoạt động của nhóm Báo cáo. Nếu nhóm e-shopping hoạt động độc lập, họ sẽ phát vỡ cam kết, sớm hay muộn và thay đổi một số cái nhóm Báo cáo không muốn. Mô hình này chạy tốt khi nhóm có chung quản lý - giúp quá trình đưa ra quyết định dễ hơn, tạo ra sự hài hòa.

Khi đối mặt với những tình huống như vậy, chúng ta phải hành động ngay. Nhóm Báo cáo cần đóng vai trò customer (khách hàng) và nhóm e-shopping cần đóng vai trò supplier (nhà cung cấp). Hai nhóm này cần gặp thường xuyên hay khi có đề nghị, hoặc chat giữa khách hàng và nhà cung cấp. Nhóm khách hàng cần đưa ra yêu cầu, nhóm supplier cần lên kế hoạch tương ứng. Dù cuối cùng mọi yêu cầu của nhóm khách hàng phải được thoả mãn hết, kế hoạch thực hiện việc đó nằm ở nhóm Cung cấp. Nếu một yêu cầu nào đó được coi là quan trọng, yêu cầu đó cần được thực thi sớm hơn và các yêu cầu khác có thể tạm dừng. Nhóm khách hàng cũng cần cung cấp kiến thức và đầu vào cho nhóm Cung cấp. Quy trình này là một chiều nhưng đôi khi ta cần làm vậy.

Giao diện giữa hai hệ thống con cần được định nghĩa chính xác. Cần tạo một bộ kiểm thử tính phù hợp và có thể chạy nó bất kỳ lúc nào để đảm bảo rằng yêu cầu của giao diện được tôn trọng. Nhóm cung cấp cần làm việc nỗ lực để đảm bảo thiết kế của họ đủ an toàn và bộ kiểm thử đưa ra cảnh báo khi có vấn đề xảy ra.

Hãy thiết lập mối quan hệ khách hàng/người cung cấp rõ ràng giữa hai nhóm. Trong phiên lên kế hoạch, hãy để nhóm khách hàng đóng vai khách hàng với nhóm nhà cung cấp. Đàm phán và lên kế hoạch công việc với các yêu cầu của khách hàng sao cho mọi người hiểu cam kết và kế hoạch.

Hãy phát triển bộ kiểm thử chấp nhận chung. Bộ kiểm thử này kiểm tra tính đúng đắn của giao diện. Thêm những trường hợp kiểm thử này vào bộ kiểm thử của nhóm cung cấp và cho nó chạy dưới dạng một phần của quá trình tích hợp tự động. Việc kiểm thử này sẽ đảm bảo rằng nhóm cung cấp có thể thay đổi mà không sợ gây ra hiệu ứng phụ với chương trình của nhóm khách hàng.

Chủ nghĩa Thủ cựu[7]

Quan hệ khách hàng - nhà cung cấp có tính quan trọng sống còn khi cả hai nhóm đều cần trong quan hệ. Khách hàng rất phụ thuộc vào nhà cung cấp, trong khi nhà cung cấp thì không phụ thuộc vào khách hàng. Nếu cần quản lý để quan hệ này hoạt động tốt, nhà cung cấp cần để ý, lắng nghe yêu cầu của khách hàng. Nếu người quản lý chưa quyết định rõ ràng phân định giữa hai nhóm, hoặc quản lý kém, hay không có quản lý, nhóm cung cấp sẽ dần để ý tới mô hình và thiết kế và ít quan tâm đến việc hỗ trợ khách hàng. Họ đều có áp lực hạn chót. Ngay cả khi họ đều là người tốt, sẵn sàng hỗ trợ nhóm khác, áp lực về thời gian sẽ trả lời, kết quả là nhóm khách hàng bị sẽ chịu thiệt hại. Điều này cũng xảy ra khi các nhóm thuộc các công ty khác nhau. Giao tiếp khó, công ty của nhóm người cung cấp không quan tâm và đầu tư nhiều thời gian vào quan hệ này. Họ sẽ chỉ hỗ trợ kiểu nhỏ giọt, hay từ chối không hợp tác. Kết quả là nhóm khách hàng chỉ còn mỗi họ, tự vận động làm tốt nhất với mô hình và thiết kế riêng của chính họ.

Khi hai nhóm phát triển có quan hệ khách hàng - người cung cấp, nhóm nhà cung cấp không có động lực cung cấp những gì nhóm khách hàng cần, nhóm khách hàng đợi chờ trong vô vọng. Lòng vị tha có thể tạo nên động lực cho lập trình viên phía nhà cung cấp và họ hứa, nhưng có thể họ không thực hiện được. Lòng tin về những ý định tốt này làm cho nhóm khách hàng lên kế hoạch dựa trên những chức năng không bao giờ tồn tại. Dự án khách hàng bị dừng cho đến khi nhóm nhận ra rằng họ phải học cách sống với những gì đang có. Một giao diện được may đo cho nhóm khách hàng không nằm trong danh sách công việc.

Nhóm khách hàng có một số lựa chọn. Lựa chọn rõ ràng nhất là tách khỏi nhóm nhà cung cấp và tự làm hết. Chúng ta sẽ xem xét ý này trong pattern "Separate Way". Thỉnh thoảng lợi ích đem lại từ hệ thống con của nhóm cung cấp nhỏ tới mức không đáng để nhờ. Và sẽ đơn giản hơn nếu tạo một mô hình riêng rẽ, tự thiết kế mà không cần để ý tới mô hình của nhóm cung cấp. Tuy vậy, đây không phải là cách chúng ta luôn được chọn.

[7] Conformist

Thỉnh thoảng, mô hình của nhóm cung cấp có một số giá trị và chúng ta phải duy trì kết nối với họ. Nhưng vì nhóm cung cấp không thể hỗ trợ nhóm khách hàng, nhóm khách hàng cần tìm cách tự bảo vệ khỏi sự thay đổi của nhóm cung cấp. Nhóm khách hàng phải tự thực thi lớp chuyển đổi kết nối hai ngữ cảnh. Lý do có thể khác là mô hình của nhóm cung cấp kém tới mức không thể thay đổi. Ngữ cảnh của nhóm khách hàng vẫn có thể dùng nó, nhưng điều đó không làm cho họ tự bảo vệ được thông qua việc sử dụng Lớp Chống đổ vỡ - một pattern chúng ta sắp đề cập.

Nếu khách hàng phải sử dụng mô hình của nhóm cung cấp, và nếu họ làm tốt, chúng ta cần tuân thủ chặt chẽ. Nhóm khách hàng có thể bám vào mô hình của nhóm cung cấp, hoàn toàn lấy đó làm chuẩn. Điều này rất giống với mẫu Nhân Chung, nhưng sự khác nhau ở đây là quan trọng. Nhóm khách hàng không thể thay đổi nhân. họ chỉ có thể dùng nó như một phần trong mô hình của họ, và họ có thể xây dựng dựa trên mã nguồn đã được cung cấp. Có nhiều trường hợp như vậy khi ta không có giải pháp khả dĩ. Khi ai đó cung cấp một cấu phần lớn cùng với giao diện đi kèm, chúng ta có thể xây dựng mô hình chứa cấu phần tương ứng giống như cách coi đó là mô hình của chúng ta. Nếu cấu phần chỉ là một giao diện nhỏ, sẽ tốt hơn nếu chỉ đơn giản tạo một bộ nối cho nó và dịch mô hình của nhóm khách hàng và nhóm nhà cung cấp. Bằng cách này, chúng ta đã cô lập mô hình của mình và có thể phát triển nó với độ độc lập cao.

Lớp chống Đổ vỡ[8]

Chúng ta thường xuyên gặp tình huống viết một chương trình mới tương tác với phần mềm cũ hoặc phần mềm độc lập. Đây là một thử thách khác với người vẽ mô hình domain. Nhiều phần mềm cũ không được xây dựng với kỹ thuật mô hình domain và bản thân mô hình của họ lộn xộn, khó hiểu và khó làm. Ngay cả khi những phần mềm ngoài như vậy tốt, mô hình của phần mềm cũ có thể không hữu dụng cho chúng ta nhiều lắm vì các mô hình thường là khác nhau. Dù sao, sẽ có một mức độ tích hợp nào đó giữa mô hình của chúng ta và mô hình của phần mềm cũ vì chúng ta nhận được yêu cầu phải sử dụng phần mềm cũ.

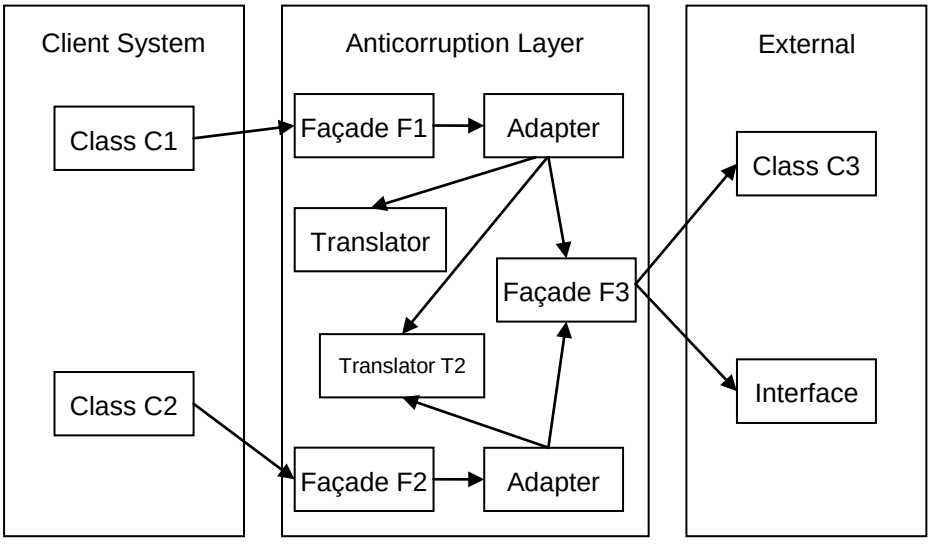
Có nhiều cách để hệ thống khách của chúng ta tương tác với hệ thống ngoài. Một trong những cách đó là kết nối mạng. Cả hai chương trình cần dùng chung giao thức mạng, và client cần tuân theo giao diện được cung cấp bởi hệ thống ngoài. Một phương pháp khác là tương tác với CSDL. Hệ thống ngoài tác động và lưu dữ liệu vào CSDL. Giả định rằng hệ thống client cũng truy cập vào cùng CSDL. Trong cả hai trường hợp chúng ta đều làm việc với dữ liệu nguyên thủy không chứa thông tin nào về mô hình. Chúng ta không thể lấy mọi dữ liệu từ CSDL và đối xử nó như là dữ liệu nguyên thủy. Có nhiều semantic ẩn

[8] Anticorruption Layer

trong dữ liệu. CSDL dạng quan hệ chứa dữ liệu nguyên thủy liên quan tới dữ liệu nguyên thủy khác như một mạng lưới quan hệ chằng chịt. Semantic dữ liệu là rất quan trọng và cần được xem xét. Chương trình client không thể truy cập CSDL và viết vào nó nếu không hiểu ý nghĩa của dữ liệu đang dùng. Chúng ta thấy một phần của mô hình ngoài được phản ánh vào CSDL và đi vào mô hình của chúng ta.

Có rủi ro cho mô hình ngoài khi thay đổi mô hình của client nếu ta cho phép điều đó xảy ra. Chúng ta không thể bỏ qua sự tương tác với mô hình ngoài, nhưng chúng ta cần cẩn thận và cô lập hóa mô hình của chúng ta khỏi mô hình đó. Chúng ta cần xây dựng một Lớp chống lại sự đổ vỡ đứng giữa mô hình client và mô hình của hệ thống ngoài; nhưng không giống như một yếu tố ngoài. Nó hoạt động với khái niệm và hành động tương tự như mô hình của chúng ta. Tuy nhiên Lớp chống đổ vỡ nói chuyện với mô hình ngoài sử dụng ngôn ngữ ngoài, không phải ngôn ngữ client. Lớp này hoạt động như là trình dịch giữa hai domain và ngôn ngữ. Thành quả to lớn là mô hình client vẫn giữ được sự trong sạch và nhất quán mà không bị làm bẩn bởi mô hình ngoài.

Vậy chúng ta thực thi Lớp chống đổ vỡ thế nào? Một giải pháp tốt là coi lớp như là một Dịch vụ từ mô hình client. Dịch vụ rất dễ dùng vì nó trừu tượng hóa hệ thống khác và cho phép chúng ta dùng các thuật ngữ riêng của chúng ta. Dịch vụ không cần dịch, do đó mô hình của chúng ta giữ được sự cách biệt. Về cách thực thi thực tế, Dịch vụ được làm dưới dạng Façade (xem cuốn *Design Pattern* của Gamma et al. 1995) Bên cạnh đó, Lớp chống đổ vỡ cần một Bộ nối. Bộ nối này cho phép bạn biến đổi interface của một lớp thành một thứ khác mà client hiểu. Trong trường hợp của chúng ta, Bộ nối không cần wrap một lớp vì công việc của nó là dịch giữa hai hệ thống.



Lớp chống đổ vỡ có thể chứa nhiều Dịch vụ. Với mỗi dịch vụ có một Façade tương ứng và với mỗi Façade chúng ta thêm một Bộ nối. Chúng ta không nên dùng một Bộ nối duy nhất cho mọi Dịch vụ vì nếu làm vậy chúng ta sẽ làm "loạn" mọi thứ lên do các chức năng bị trộn lẫn.

Chúng ta vẫn phải thêm một cấu phần. Bộ nối đảm nhận nhiệm vụ wrap hành vi của hệ thống ngoài. Chúng ta cũng cần biến đổi đối tượng và dữ liệu. Việc này có thể được thực hiện bằng trình dịch.

Đó có thể làm một đối tượng đơn giản, ít chắc năng, phục vụ nhu cầu dịch dữ liệu cơ bản. Nếu hệ thống ngoài có giao diện phức tạp, chúng ta có thể cần thêm Façade giữa Bộ nối và giao diện đó. Làm như vậy sẽ đơn giản hóa giao thức của Bộ nối và phân tách nó với hệ thống khác.

Separate Ways

Chúng ta đã tìm hiểu nhiều cách để tích hợp hệ thống con, làm cho chúng hoạt động cùng nhau sao cho mô hình và thiết kế hoạt động tốt. Điều này đòi hỏi công sức và sự thỏa hiệp. Những nhóm làm việc với hệ thống con tương ứng cần dành một lượng thời gian đáng kể để củng cố mối quan hệ giữa các hệ thống con. Chúng có thể cần merge liên tục vào mã nguồn, tiến hành kiểm thử để đảm bảo không phần nào không chạy. Thỉnh thoảng, một trong các nhóm cần dành thời gian đáng kể để thực thi một số yêu cầu mà nhóm khác cần. Chúng ta cần sự thỏa hiệp ở đây. Một mặt ta cần phát triển độc lập, chọn khái niệm và liên kết tự do, mặt khác cần đảm bảo mô hình của bạn khớp với framework của hệ thống khác. Chúng ta cũng cần thêm lớp đặc biệt đóng vai trò trình dịch giữa hai hệ thống con. Trong đa số trường hợp, chúng ta phải làm vậy, nhưng nhiều khi chúng ta chúng ta có thể chọn cách khác. Chúng ta phải đánh giá sự thuận tiện của việc tích hợp và dùng nó chỉ khi thực sự có giá trị. Nếu chúng ta kết luận rằng, việc tích hợp đem lại nhiều vấn đề hơn giá trị của nó, chúng ta có thể dùng mẫu Separate Way.

Mẫu Separate Way chỉ ra trường hợp khi một chương trình doanh nghiệp được tạo nên bởi nhiều chương trình nhỏ có ít hoặc không hề có sự liên quan về mô hình. Có một tập yêu cầu, và từ phương diện người dùng đây là một chương trình, nhưng từ góc nhìn mô hình và thiết kế thì nó cỏ thể thành có mô hình độc lập và thực thi riêng biệt. Chúng ta nhìn vào yêu cầu và xem liệu có thể chia thành hai hay nhiều tập không hề có phần chung hay không? Nếu chúng ta có thể chia như vậy nghĩa là chúng ta có thể phân chia Ngữ cảnh giới hạn và làm mô hình hóa độc lập. Điều này thuận tiện ở chỗ chúng ta được tự do lựa chọn công nghệ dùng trong thực thi. Chương chình chúng ta tạo ra chia sẻ một phần GUI chung nhỏ và hoạt động như là một portal với liên kết hay nút được truy cập từ mỗi chương trình. Đây là sự tích hợp nhỏ trong đó việc chúng ta phải làm chỉ là tổ chức lại chương trình chứ không phải mô hình phía sau chúng.

Trước khi làm Separate Way chúng ta cần đảm bảo rằng chúng ta không quay lại câu chuyện hệ thống đã tích hợp. Mô hình được phát triển độc lập rất khó tích hợp. Chúng có rất ít sự liên quan và không đáng được thích hợp.

Dịch vụ Host mở[9]

Khi chúng ta tích hợp hai hệ thống con, chúng ta thường tạo một lớp phiên dịch giữa chúng. Lớp này đóng vai trò như một miếng đệm giữa hệ thống con client và hệ thông con ngoài chúng ta sẽ tích hợp tới. Lớp này có thể là nhất quán, phụ thuộc vào độ phức tạp của mối quan hệ và cách hệ thống con ngoài được thiết kế. Nếu hệ thống con ngoài không được dùng bởi một hệ thống con client bởi nhiều hệ thống con thì chúng ta cần tạo lớp phiên dịch cho tất cả chúng. Tất cả những lớp này lặp lại công việc phiên dịch giống nhau, và chứa mã nguồn tương tự nhau.

Khi một hệ thống con phải tích hợp với nhiều hệ thống khác, việc tùy biến một trình phiên dịch cho mỗi chúng có thể làm cả nhóm sa lầy. Chúng ta phải bào trì ngày càng nhiều, phải lo lắng nhiều và nhiều hơn nữa khi có thay đổi. Giải pháp là xem hệ thống con ngoài như một bên cung cấp dịch vụ. Nếu chúng ta wrap một tập các dịch vụ xung quanh chúng, và bắt mọi hệ thống con khác phải truy cập qua những Dịch vụ này thì chúng ta không cần lớp phiên dịch nữa. Khó khăn là mỗi hệ thống con có thể cần phải tương tác theo một cách riêng với hệ thống con ngoài, và tạo nên một tập liên kết các Dịch vụ có thể gây ra vấn đề.

Định nghĩa một giao thức cho phép truy cập tới hệ thống con của bạn như là một tập các Dịch vụ. Mở một giao thức sao cho bất kỳ ai muốn tích hợp tới đều có thể dùng nó. Tăng cường và mở rộng giao thức để đối phó với yêu cầu tích hợp mới, trừ khi một nhóm lẻ có yêu cầu riêng. Sau đó, dùng một trình phiên dịch chỉ-dùng-một-lần để bổ sung cho giao thức với trường hợp đặc biệt đó sao cho giao thức chia sẻ trở nên đơn giản và gắn kết.

Chưng cất[10]

Chưng cất là quá trình phân chia những thực thể tạo nên một thể hỗn hợp. Mục đích việc chưng cất là trích xuất ra một thực thể cụ thể từ thể hỗn hợp. Trong quá trình chưng cất, chúng ta có thể tạo ra sản phẩm phụ (có thể thú vị).

[9] Open Host Service
[10] Distillation

Một domain lớn kéo theo một mô hình lớn ngay cả khi chúng ta đã làm mịn và tạo nhiều trừu tượng. Nó có thể trở nên lớn sau khi tái cấu trúc. Trong những trường hợp như vậy, ta cần chưng cất. Ý tưởng định nghĩa một Domain Cốt lõi thể hiện sự thiết yếu của domain. Sản phẩm trung gian của quá trình chưng cất là domain con chung chứa những phần còn lại của domain.

Khi thiết kế hệ thống lớn, sẽ có nhiều cấu phần đi kèm. Những cấu phần này đều phức tạp và cần cho sự thành công đối với mô hình domain, với tài sản kinh doanh thực có thể bị ẩn và bỏ qua.

Khi làm việc với mô hình lớn, chúng ta nên thử chia những khái niệm chủ yếu khỏi những mô hình chung. Trong phần đầu, chúng ta đã đưa ra ví dụ về hệ thống theo dõi không lưu. Chúng ta nói rằng Lộ trình Bay chứa Route đã được thiết kế mà máy bay phải bay theo. Route thể hiện một khái niệm trong hệ thống. Thực ra, khái niệm này là chung và không phải là thiết yếu. Khái niệm Route được dùng trong nhiều domain và ta có thể thiết một mô hình chung có thể dùng mô tả nó. Sự cần thiết của việc theo dõi không lưu nằm ở chỗ khác. Hệ thống theo dõi biết route mà máy bay phải theo, nhưng nó cũng nhận đầu vào từ một mạng lưới ra-đa theo dõi máy bay trong không trung. Dữ liệu này chỉ ra đường đi thực tế của máy bay và nó khác với lộ trình đã mô tả ở trên. Hệ thống sẽ tính quỹ đạo của máy bay dựa trên những tham số bay hiện tại cũng như đặc tính của thời tiết. Quỹ đạo là một đường 4 chiều mô tả đầy đủ route mà máy bay bay. Ta có thể tính quỹ đạo với độ dài vài phút hay vài tiếng.

Mỗi công thức tính này hỗ trợ quy trình tạo ra quyết định. Mục đích tổng thể của việc tính quỹ đạo máy bay là để xem máy bay có thể bị đâm vào máy bay khác hay không. Trong vùng lân cận của sân bay, trong khi cất cánh, nhiều máy bay bay lòng vòng quanh đầu trời. Nếu một máy bay trượt ra khỏi route đã dự định, có khả năng máy bay sẽ đâm nhau. Hệ thống theo dõi không lưu sẽ tính quỹ đạo của máy bay và đưa ra cảnh báo nếu có thể xảy ra va chạm. Khi máy bay xa nhau hơn, quỹ đạo được tính xa hơn, dài hơn theo thời gian và ta có nhiều thời gian phản ứng hơn. Mô hình tổng hợp quỹ đạo máy bay từ những dữ liệu đã có là linh hồn của hệ thống nghiệp vụ này. Ta có thể coi là nó là domain cốt lõi. Mô hình tính đường đi là domain chung.

Domain Cốt lõi của một hệ thống phụ thuộc vào cách chúng ta xem xét nó. Một hệ thống Tính đường đi đơn giản sẽ nhìn Route và những gì phụ thuộc vào nó là trung tâm của thiết kế. Hệ thống theo dõi không lưu sẽ con Route là domain con chung. Domain Cốt lõi của một chương trình có thể là một domain con chung của một hệ thống khác. Việc xác định chính xác (Domain) Cốt lõi là rất quan trọng, nó xác định mối quan hệ của nó với các thành phần khác trong mô hình.

Hãy đào sâu tìm hiểu kỹ mô hình. Tìm ra Domain Cốt lõi và tìm cách phân biệt dễ dàng nó với mô hình và mã nguồn có tính hỗ trợ cho phần còn lại. Nhấn mạnh khái niệm chuyên biệt và có giá trị nhất. Ngoài ra, hãy làm cho (Domain) Cốt lõi càng nhỏ càng tốt.

Hãy để những siêu sao của nhóm làm việc với Domain Cốt lõi, tuyển những nhân tài như thế. Dành công sức vào (Domain) Cốt lõi để tìm ra những mô hình sâu và xây dựng thiết kế uyển chuyển - đủ để thoả mãn tầm nhìn của hệ thống. Bảo vệ nguồn đầu tư trong các phần khác và xem chúng hỗ trợ (Domain) Cốt lõi thế nào.

Việc chỉ định những lập trình viên tốt nhất làm việc với Domain Cốt lõi là quan trọng. Lập trình viên thường thích công nghệ, học ngôn ngữ mới nhất, họ hướng tới hạ tầng hơn là logic nghiệp vụ. Logic nghiệp vụ của domain có thể không làm cho họ hứng thú và kết quả đem lại sẽ không tốt. Cuối cùng, điểm nhấn trong việc tìm hiểu đặc tả của quỹ đạo máy bay là gì? Khi dự án hoàn thành, mọi kiến thức trở thành dĩ vãng và ít lợi ích. Tuy vậy logic nghiệp vụ của domain là linh hồn của nó. Sai lầm trong thiết kế và thực thi của phần cốt lõi có thể dẫn tới sự sụp đổ của dự án. Nếu Logic nghiệp vụ cốt lõi không hoàn thành nhiệm vụ của nó, những nỗ lực và thành quả kỹ thuật trở nên vô nghĩa.

Một Domain Cốt lỗi không phải luôn được tạo ở bước cuối cùng. Quy trình làm mịn và tái cấu trúc liên tục là cần thiết trước khi phần Lõi tiến hóa và trở nên rõ ràng. Chúng ta cần tăng cường Lõi như là phần chính của thiết kế, loại bỏ những ranh giới. Chúng ta cũng cần xem xét lại các thành phần khác của mô hình trong quan hệ với Lõi mới. Chúng có thể cần tái cấu trúc hay cần thay đổi một số chức năng.

Một số phần của mô hình làm tăng độ phức tạp nhưng không tạo ra hay đề cập tới mảng kiến thức đặt thù nào. Loại bỏ tất cả những gì thừa, làm cho Domain Lõi trở nên khó phân biệt và khó hiểu. Mô hình bị mắc kẹt bởi những nguyên tắc mà mọi người biết không thuộc về chuyên môn chính cần tập trung. Tuy nhiên, những mô hình này vẫn cần cho tổng thể để đảm bảo hệ thống được thể hiện dưới một mô hình trọn vẹn.

Xác định domain con tương liên không tạo động lực cho dự án của bạn. Chỉ ra mô hình chung của những domain con này và đặt chúng vào bộ mô-đun chung. Không đặt nghiệp vụ chuyên môn vào những mô-đun đó.

Sau khi chúng ta đã phân chia những mô hình đó, hãy phát triển chúng với độ ưu tiên thấp hơn Domain Lõi, tránh đưa lập trình viên chính làm việc với chúng (vì họ sẽ không thu được nhiều kiến thức domain từ đó). Ngoài ra, cũng cần xem các giải pháp *off-the-shelf* hay mô hình công khai cho những domain con chung này.

Mọi domain dùng khái niệm được dùng bởi domain khác. Tiền hay những khái niệm liên quan như tiền tệ hay tỉ giá cần đưa vào hệ thống khác. Đồ thị là một khác niệm khác được dùng rộng rãi và phức tạp nhưng có thể được dùng ở nhiều chương trình khác nhau.

Có nhiều cách để thực thi Domain con chung:

1. **Giải pháp Off-the-shelf.** Lợi thế là cả giải pháp đã được làm bởi người khác. Chúng ta cần học nó và giải pháp này vậy kéo theo sự phụ thuộc. Nếu mã nguồn nhiều lỗi, bạn cần chờ cho tới khi chúng được sửa. Bạn cũng cần dùng một số trình biên dịch và phiên bản thư viện nhất định. Việc tích hợp không dễ dàng so với việc tự phát triển hệ thống bằng nội lực.

2. **Khoán ngoài**. Việc thiết kế và thực thi được giao cho một nhóm khác, có thể là công ty khác. Cách này cho phép bạn tập trung vào Domain cốt lõi, giảm tải vì không phải xử lý những domain khác. Sự bất tiện là tích hợp mã nguồn đã khoán ngoài. Giao diện dùng để giao tiếp với domain con cần được định nghĩa và trao đổi với nhóm khác.

3. **Mô hình Có sẵn**. Một giải pháp dễ dàng là dùng một mô hình có sẵn. Có một vài cuốn sách đã công bố về mô hình phân tích, và chúng có thể dùng như một cách để truyền cảm hứng cho domain con. Có thể chúng ta không copy nguyên xi được các mẫu nhưng chúng ta có thể tái sử dụng chúng mà không cần thay đổi nhiều.

4. **Thực thi Nội bộ**[11]. Giải pháp này có thuận lợi là mức độ tích hợp đạt được tốt nhất. Nó có thể tốn công hơn, bao gồm cả chi phí bảo trì.

[11] In-House Implementation

6

Tới ngày hôm nay, DDD vẫn quan trọng: Phỏng vấn Eric Evans

InfoQ.com phỏng vấn Eric Evans (người sáng lập của Domain Driven Design) (về DDD) trong bối cảnh hiện đại:

Tại sao DDD vẫn quan trọng?
Về cơ bản, DDD là nguyên tắc mà chúng ta nên tập trung vào những vấn đề cốt lõi của lĩnh vực (domain) người dùng mà chúng ta tham gia vào, đó là phần tốt nhất của tâm trí chúng ta nên được dành cho sự hiểu biết về lĩnh vực đó, và cộng tác với các chuyên gia lĩnh vực đó để nhào nặn nó thành một hình thức khái niệm mà chúng ta có thể sử dụng để xây dựng phần mềm mạnh mẽ, linh hoạt.

Đây là một nguyên tắc rằng sẽ không bao giờ lỗi mốt. Nó áp dụng bất cứ khi nào chúng ta làm việc trong một lĩnh vực phức tạp.

Xu hướng lâu dài là hướng tới việc áp dụng phần mềm vào các vấn đề lớn hơn, phức tạp hơn và ngày càng sâu hơn vào các lĩnh vực liên quan. Dường như với tôi, xu hướng này đã bị gián đoạn trong một vài năm trong giai đoạn các website bùng nổ. Sự chú ý đã được chuyển đi từ logic phong phú và các giải pháp sâu vì có rất nhiều giá trị trong việc đẩy dữ liệu lên web với thao tác rất đơn giản. Trước đây, có rất nhiều điều để để làm, các nỗ lực phát triển được áp dụng khá tốt.

Nhưng bây giờ mức độ cơ bản của việc sử dụng web đã được sử dụng đại trà, và các dự án đang bắt đầu có nhiều tham vọng hơn nữa về logic kinh doanh.

Rất gần đây, nền tảng phát triển web đã bắt đầu trưởng thành, đủ để phát triển web nhanh chóng cho DDD, và có một số xu hướng tích cực. Ví dụ, SOA, khi nó được sử dụng tốt, cung cấp cho chúng ta một phương pháp rất hữu ích để phân chia lĩnh vực.

Trong khi đó, quy trình Agile đã có đủ ảnh hưởng mà hầu hết các dự án hiện nay có ít nhất ở việc làm việc định kỳ và chặt chẽ với các đối tác kinh doanh, ứng dụng tích hợp liên tục, và làm việc trong một môi trường truyền thông cao.

Vì vậy, DDD có vẻ như ngày càng quan trọng trong tương lai gần, và một số nền tảng đã được xây dựng sẵn.

Nền tảng công nghệ (Java, .NET, Ruby...) được liên tục phát triển. Làm thế nào Domain Driven Design tận dụng tốt các nền tảng này?

Trong thực tế, công nghệ và quy trình mới cần phải được đánh giá xem liệu chúng có hỗ trợ nhóm tập trung vào lĩnh vực của họ, chứ không phải để làm họ sao nhãng. DDD không phải dành riêng cho một nền tảng công nghệ nào, mà là một số nền tảng cho diễn đạt tốt hơn trong việc tạo logic kinh doanh. Liên quan tới ý thứ hai, những năm gần đây đã xuất hiện nhiều hướng đi đầy triển vọng, đặc biệt là sau năm 1990.

Java đã được lựa chọn mặc định trong vài năm qua, nó là ngôn ngữ hướng đối tượng đặc trưng. Nó có thu gom bộ nhớ, trong đó, trong thực tế, hóa ra là điều cần thiết. (C++ ngược lại với Java ở chỗ nó đòi hỏi quá nhiều sự chú ý đến các chi tiết ở mức độ thấp) Cú pháp Java vẫn phức tạp, nhưng mã nguồn các đối tượng dạng POJO vẫn dễ dàng đọc. Ngoài ra, một số đặc tả trong Java 5 giúp việc đọc/hiểu mã nguồn tốt hơn.

Nhưng sau khi J2EE đầu tiên xuất hiện, nó hoàn toàn chôn vùi khả năng diễn đạt dưới núi mã nguồn khổng lồ. Việc áp dụng các quy ước trước đây như EJB home, get/set accessor, tiền tố cho tất cả các biến... chỉ làm mã nguồn càng khó đọc hơn. Những quy ước quá cồng kềnh làm giảm hiệu suất làm việc của đội ngũ phát triển khi áp dụng. Và rất khó khăn để thay đổi các đối tượng, một khi các mã nguồn được tạo ra, vì thế mọi người chỉ không thay đổi chúng nhiều. Đây là một nền tảng không tối ưu cho việc mô hình hoá domain.

Kết hợp với sự cấp thiết để phát triển giao diện Web qua HTTP và HTML (vốn không được thiết kế cho mục đích đó) khi sử dụng công cụ thô sơ thế hệ đầu tiên. Cũng trong thời gian đó, việc tạo và duy trì một giao diện người dùng tinh tế trở nên khó khăn, khiến cho việc thiết kế các chức năng phức tạp bên trong không được chú ý nhiều lắm. Trớ trêu thay, ngay lúc công nghệ hướng đối tượng trở thành trào lưu, việc mô hình hóa và thiết kế gặp phải trở ngại lớn.

Tình hình cũng tương tự như trong nền tảng .Net, với một số vấn đề đang được xử lý tốt hơn một chút, và số khác thì tồi tệ hơn.

Đó là một khoảng thời gian đáng quên, nhưng xu hướng này đã thay đổi trong bốn năm qua. Đầu tiên, nhìn vào Java, cộng đồng xì xào về cách sử dụng các framework có chọn lọc, và một loạt các framework mới (chủ yếu là mã nguồn mở) đang từng bước được cải thiện. Framework như Hibernate và Spring xử lý công việc cụ thể mà J2EE đã cố gắng giải quyết, nhưng theo một cách nhẹ nhàng hơn nhiều. Phương pháp tiếp cận như AJAX để giải quyết vấn đề giao diện người dùng theo một cách dễ dàng hơn. Và các dự án trở nên thông minh hơn khi chọn lọc các yếu tố của J2EE cung cấp cho họ giá trị, đồng thời trộn với các yếu tố mới. Thuật ngữ POJO được sinh ra trong thời gian này.

Kết quả là việc xây dựng hệ thống dễ dàng hơn, và một sự cải thiện rõ rệt trong việc tách logic kinh doanh ra khỏi phần còn lại của hệ thống, do đó nó có thể được viết lại với các POJO. Điều này không tự động tạo ra một thiết kế hướng lĩnh vực, nhưng nó tạo ra một cơ hội thực tế.

Đó là thế giới Java. Sau đó, bạn có các ngôn ngữ khác như như Ruby. Ruby có một cú pháp rất dễ hiểu, và ở mức độ cơ bản này nó phải là một ngôn ngữ rất tốt cho DDD (mặc dù tôi đã không nghe nói về nhiều sử dụng thực tế của nó trong những những ứng dụng như thế). Rails đã tạo ra rất nhiều hứng thú vì nó cuối cùng dường như làm cho việc tạo ra các Web UI dễ dàng khi các vấn đề về UI đã trở lại trong đầu những năm 1990, trước cả Web. Ngay cả bây giờ cũng vậy, khả năng này đã phần lớn được áp dụng để xây dựng một số lượng lớn các ứng dụng Web không có nhiều kiến thức lĩnh vực phía sau họ. Nhưng hy vọng của tôi là, như phần xây dựng giao diện người dùng của vấn đề dễ hơn, người ta sẽ thấy điều này như một cơ hội để tập trung nhiều hơn sự chú ý của họ vào lĩnh vực. Nếu sử dụng Ruby theo hướng đó, tôi nghĩ rằng nó có thể cung cấp một nền tảng tuyệt vời cho DDD. (Một vài phần liên quan tới cơ sở hạ tầng có thể sẽ phải được thêm vào)

Thêm vào đó là những nỗ lực trong ngôn ngữ miền cụ thể, mà tôi đã tin tưởng lâu nay có thể là bước lớn tiếp theo cho DDD. Cho đến nay, chúng ta vẫn chưa có một công cụ thực sự mang lại cho chúng ta những gì chúng ta cần. Nhưng mọi người đang thử nghiệm nhiều hơn bao giờ hết trong lĩnh vực này, và điều đó làm cho tôi hy vọng.

Ngay bây giờ, như tôi có thể nói, hầu hết mọi người cố gắng áp dụng DDD làm việc với Java hay .Net, hay một chút trong Smalltalk. Vì vậy, xu hướng tích cực trong thế giới Java sẽ có hiệu quả ngay lập tức.

Những gì đã và đang diễn ra trong cộng đồng DDD kể từ khi ông viết cuốn sách của mình?

Một điều kích thích tôi là khi mọi người dùng những nguyên tắc mà tôi đã nói trong cuốn sách đó và sử dụng chúng theo những cách tôi không bao giờ ngờ tới. Một ví dụ là việc sử dụng các thiết

kế chiến lược tại Statoil, các công ty dầu khí quốc gia Na Uy. Các kiến trúc sư có viết một báo cáo kinh nghiệm về nó. (Bạn có thể đọc tại http://domaindrivendesign.org/articles/)

Trong số những thứ khác, họ đã lập bản đồ ngữ cảnh và áp dụng nó để đánh giá các phần mềm trong xây dựng quyết định mua hay tự xây dựng.

Như một ví dụ khá khác, một số người trong chúng ta đã đang khám phá một số vấn đề bằng cách phát triển một thư viện Java cho các lĩnh vực khác nhau, có thể tham khảo tại:

http://timeandmoney.domainlanguage.com

Chúng tôi đang tìm kiếm, ví dụ, làm thế nào đến nay chúng ta có thể đưa ra các ý tưởng về một ngôn ngữ lĩnh vực cụ thể, trong khi vẫn sử dụng các đối tượng trong Java.

Tôi luôn trân trọng khi mọi người liên hệ với tôi để cho tôi biết về những gì họ đang làm.

Ông có lời khuyên nào cho những người đang cố gắng tìm hiểu DDD??

Đọc sách của tôi! ;-) Ngoài ra, hãy thử sử dụng timeandmoney vào dự án của bạn. Một trong những mục tiêu ban đầu của chúng tôi là cung cấp các ví dụ tốt mà mọi người có thể học hỏi bằng cách sử dụng nó.

Một điều cần lưu ý là DDD là những gì nhóm làm, vì vậy bạn có thể được coi như một nhà truyền giáo. Thực tế, bạn có thể thấy cần một dự án mà họ đang nỗ lực làm điều này.

Hãy ghi nhớ một số điều sau trong thực tế:

1) Luôn lập trình. Người làm mô hình cần phải lập trình.
2) Tập trung vào các tình huống cụ thể. Tư duy trừu tượng phải được đặt trong các trường hợp cụ thể.
3) Đừng cố gắng áp dụng DDD vào tất cả mọi thứ. Vẽ ánh xạ ngữ cảnh và quyết định xem nơi nào bạn sẽ dùng DDD và nơi nào không.
4) Thử nghiệm nhiều và hy vọng sẽ có rất nhiều sai lầm. Mô hình hóa là một quá trình sáng tạo.

Về Eric Evans

Eric Evans là tác giả cuốn *"Domain-Driven Design: Tackling Complexity in Software", Addison-Wesley 2004.*

Kể từ đầu những năm 1990, ông đã làm việc trong nhiều dự án phát triển hệ thống kinh doanh lớn với các đối tượng với nhiều cách tiếp cận khác nhau và có nhiều kết quả khác nhau. Cuốn sách tổng hợp những kinh nghiệm đó. Nó trình bày một hệ thống mô hình hóa và kỹ thuật thiết kế mà các nhóm đã thành công trong việc sử dụng để gắn các hệ thống phần mềm phức tạp với nhu cầu kinh doanh và để giữ cho các dự án linh hoạt như các hệ thống lớn.

Hiện tại Eric phụ trách *"Domain Language"* - một nhóm tư vấn và đào tạo các nhóm áp dụng thiết kế hướng lĩnh vực, giúp họ làm cho việc phát triển hiệu quả hơn và có giá trị hơn cho doanh nghiệp của họ.

Danh sách Thuật ngữ

No	Tiếng Anh	Tiếng Việt
1	Domain Driven Design	Thiết kế hướng lĩnh vực
2	Domain	(không dịch)
3	Abstraction	Trừu tượng
4	Domain expert	Chuyên gia lĩnh vực, chuyên gia domain
5	Design pattern	Mẫu thiết kế
6	Developer	Lập trình viên, người phát triển
7	Ubiquitous language	Ngôn ngữ chung
8	Entity	Thực thể
9	Identity	Định danh - tên
10	Bounded context	Ngữ cảnh giới hạn
11	Persistence	<để nguyên không dịch>
12	Refactor	(không dịch)
13	Coherency	Tính tương liên
14	Merge	<không dịch>, (merge) mã nguồn, gộp
15	Context map	Ánh xạ ngữ cảnh
16	Shared Kernel	Nhân chung
17	Distillation	Chưng cất
18	Generic subdomain	Domain con chung
19	Generic model	Mô hình chung
20	Deep model	Mô hình sâu
21	Supple design	Thiết kế uyển chuyển
22	Cohesive	Tương liên
23	Off-the-shelf	<để nguyên>
24	Published models	Mô hình công khai
25	Analysis pattern	<không dịch>, hoặc "mô hình phân tích"
26	Database	CSDL, cơ sở dữ liệu
27	Cohesion	Tính tương liên

Về bản dịch tiếng Việt “DDD Quickly”

Nhóm dịch và hiệu đính "DDD Quickly" Tiếng Việt

Lê Đức Anh
Võ Đức Phương
Nguyễn Huy Tuấn
Nguyễn Thanh Tùng
Nguyễn Vũ Hưng

Mọi ý kiến về bản dịch xin liên hệ nhóm: https://www.facebook.com/DDD-Quickly-Tiếng-Việt-1522596288051952/

Lời cảm ơn

Chúng tôi xin chân thành cảm ơn công ty Septeni Technology đã tạo điều kiện để bản dịch hoàn tất.
Chúng tôi cũng xin gửi lời cảm ơn tới cộng động AgileVietnam và IT Leader Club đã hỗ trợ việc review và hiệu đính bản dịch này.

Phiên bản

Phiên bản	*Nội dung thay đổi*	*Ngày thay đổi*	*Tác giả*
1.0.1	Sửa lại hình trang bìa với độ phân giải cao hơn (để in)	2015/12/08	Nguyễn Vũ Hưng
1.0	Hoàn thiện dịch và review phiên bản DDD Quickly tiếng Việt	2015/11/26	Nhóm dịch DDD Quickly tiếng Việt (xem trang trên)