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 đảm bảo mô hình có thể triển khai 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.

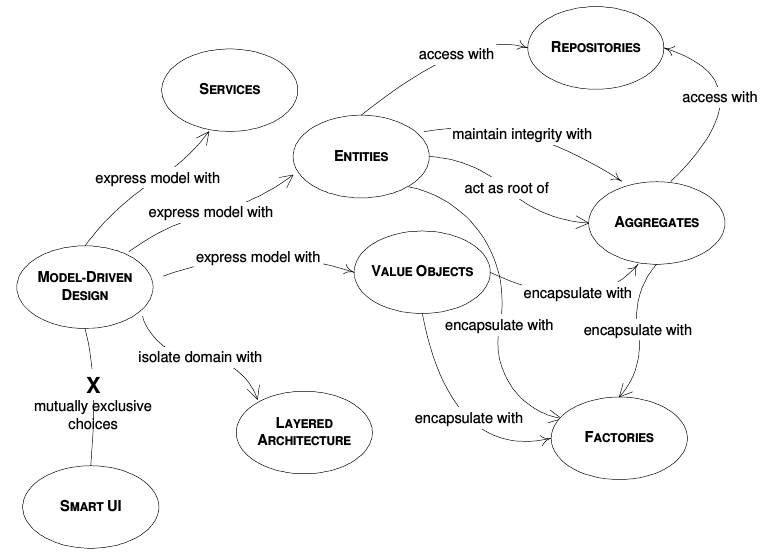
Lập trình hướng đối tượng là phù hợp để cài đặt model vì chúng cùng có chung mô hình. Lập trình hướng đối tượng cung cấp các class của đối tượng và liên kết giữa các class, các biểu hiện của đối tượng, cùng thông điệp giữa chúng. Ngôn ngữ OOP khiến chúng có khả năng tạo ánh xạ trực tiếp giữa đối tượng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

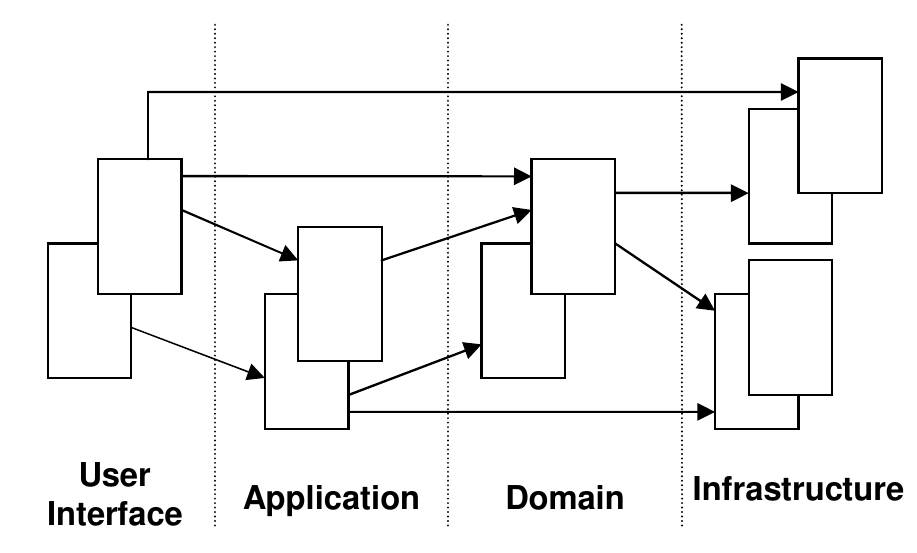
Ở một vài lĩnh vực đặc biêt, như là Toán học, có thể dễ dàng được mô hình hóa và cài đặt theo ngôn ngữ thủ tục vì nhiều lý thuyết toán học được giải quyết đơn giản bằng các lời gọi hàm và cấu trúc dữ liệu vì nó chủ yếu là về các tính toán. Lĩnh vực phức tạp hơn không chỉ là một bộ các khái niệm trừu 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 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) (Giao diện người dùng) Chịu trách nhiệm trình bày thông tin tới người sử dụng và thông dịch lệnh của người dùng.

Application Layer (Lớp Chương trình) 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 (Lớp Domain) 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 (Lớp hạ tầng) 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.

Sau đây là ví dụ về sự tương tác điển hình giữa các lớp application, domain và infrastructure. 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 tầng infrastructure.

Thực thể

Trong các đối tượng của một phần mềm, có một nhóm các đối tượng có định danh riêng, những định danh - tên gọi này của chúng được giữ nguyên xuyên suốt trạng thái hoạt động của phần mềm. Đối với những đối tượng này thì các thuộc tính của chúng có giá trị như thế nào không quan trọng bằng việc chúng tồn tại liên tục và xuyên suốt quá trình của hệ thống, thậm chí là sau cả khi đó. Chúng được gọi tên là những Thực thể - Entity.

Các ngôn ngữ lập trình thường lưu giữ các đối tượng trong bộ nhớ và chúng được gắn với một tham chiếu hoặc một địa chỉ nhớ cho từng đối tượng. Tham chiếu trong vùng nhớ này có đặc điểm là sẽ không thay đổi trong một khoảng thời gian nhất định khi chạy chương trình, tuy nhiên không có gì đảm bảo là nó sẽ luôn như vậy mãi. Thực tế là ngược lại, các đối tượng liên tục được đọc vào và ghi ra khỏi bộ nhớ, chúng có thể được đóng gói lại và gửi qua mạng và được tái tạo lại ở 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 mãn, cả hệ thống có thể sẽ gặp lỗi.

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

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

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.

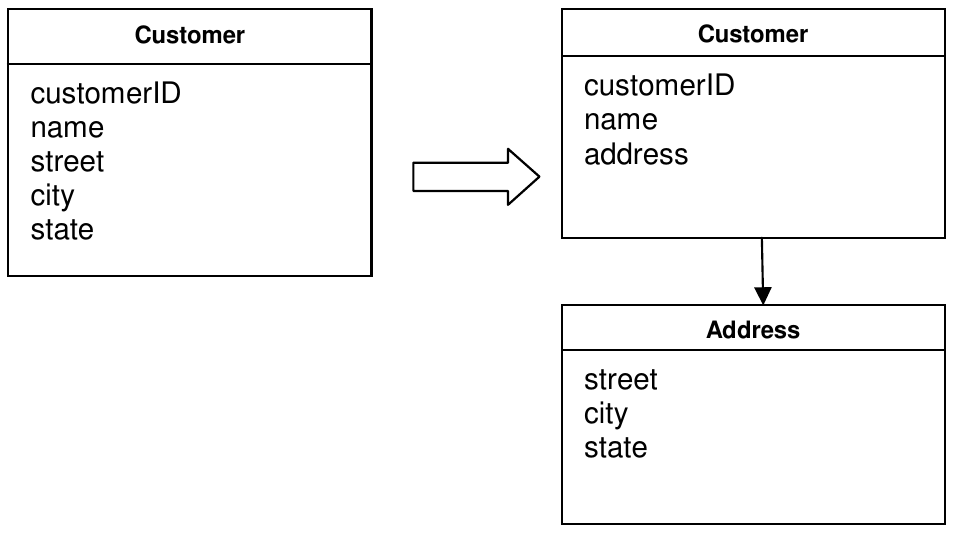
Vì những đặc điểm này nên ta cần phân biệt rõ Entity Object và Value Object. Để đảm bảo tính đồng nhất ta sẽ không gắn mọi đối tượng thành thực thể. Thay vào đó ta chỉ gắn thực thể cho những đối tượng nào phù hợp nhất với các đặc tính của thực thể. Các đối tượng còn lại sẽ là Value Object. (Chúng ta 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 nên tạo một đối tượng mới chứ không nên 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. Hãy thử tưởng tượng về 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 chuyển 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, cách 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 nếu 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 đối tượng, có thể sẽ vẫn còn rất nhiều các mối quan hệ còn lại. Lấy ví dụ một hệ thống ngân hàng lưu giữ và xử lý thông tin của khách hàng. Những dữ liệu này bao gồm các thông tin cá nhân của khách hàng như tên, địa chỉ, số điện thoại, nghề nghiệp và thông tin tài khoản: số tài khoản, số dư, các thao tác đã thực hiện. Khi hệ thống lưu trữ hoặc xóa hoàn toàn thông tin của một khách hàng thì nó cần đảm bảo rằng tất cả các tham chiếu đã được loại bỏ. Nếu như có nhiều các đối tượng lưu trữ các tham chiếu như vậy thì việc trên là rất khó khăn. Ngoài ra nữa, khi thay đổi thông tin của một khách hàng, hệ thống cần đảm bảo rằng dữ liệu đã được cập nhật trong toàn bộ hệ thống và tính toàn vẹn dữ liệu không bị xâm phạm. Thông thường yêu cầu này được đảm 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, tất cả những gì chúng có thể làm đó là thay đổi gốc một cách trực tiếp, hoặc yêu cầu gốc thực hiện một hành động nào đó. 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.

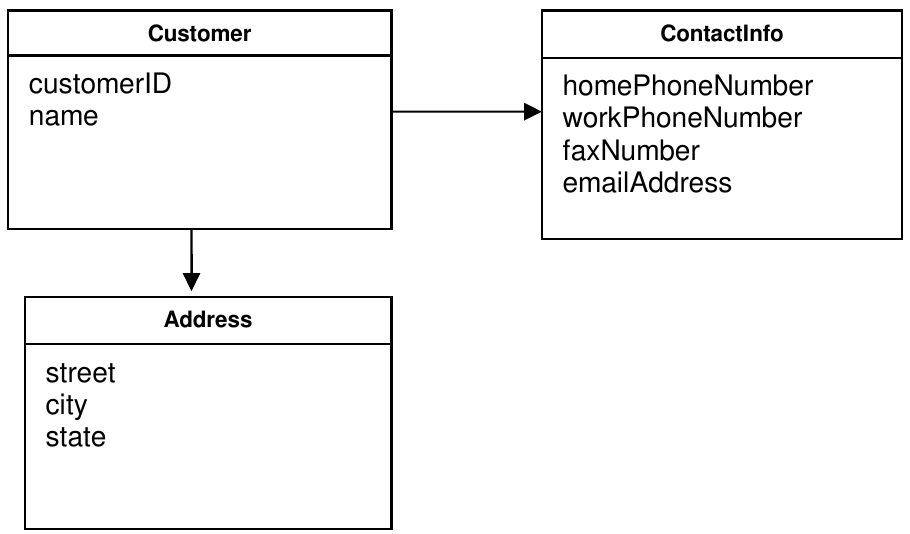
Nếu như các đối tượng của Aggregate được lưu trong CSDL thì chỉ nên cho phép truy vấn trực tiếp lấy gốc của aggregate, các đối tượng nội tại còn lại nên được truy cập thông qua các mối quan hệ bên trong.

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 aggregate (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[[1]](#footnote-0) 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 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.

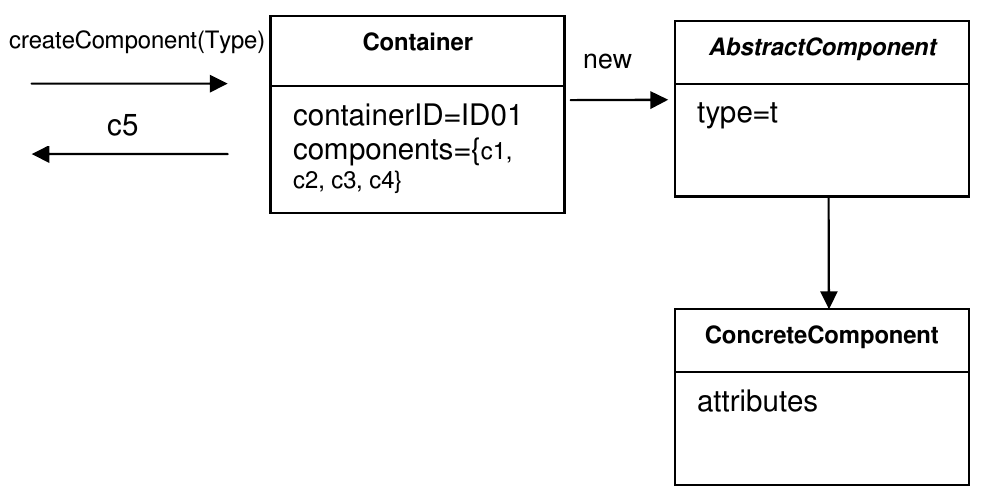
Do đó, ta cần đưa ra một khái niệm mới. Khái niệm này giúp việc gộp quy trình tạo ra đối tượng phức tạp và được gọi là Factory. Factory được dùng để gộp kiến thức cần cho việc tạo đối tượng, và chúng đặc biệt hữu dụng cho Aggregate. Khi gốc của Aggregate được tạo ra, mọi đối tượng chứa trong Aggregate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

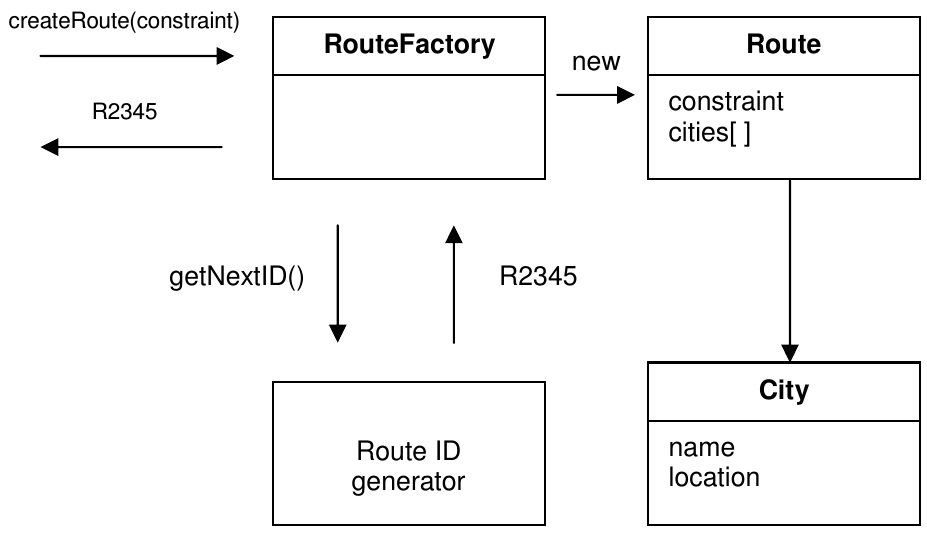
Có nhiều design pattern có thể dùng để thực thi Factory. Cuốn *Design Pattern* của *Gamma et al* mô tả chi tiết những design pattern này và chia pattern này thành 2 loại Factory Phương pháp và Factory Trừu tượng. Chúng ta không thể hiện những pattern này dưới góc độ thiết kế mà thể hiện chúng từ mô hình 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 phải có cùng kiểu. 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 thường là các đối tượng không thay đổi được, 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. Sự khởi dựng 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 đề hoàn nguyên đối tượng đã tồn tại trước đó, có thể trong CSDL. Đưa các Thực Thể trở lại bộ nhớ từ nơi lưu trữ của chúng tại CSDL có liên quan đến một quá trình hoàn toàn khác so với việc tạo ra một cái mới. Một sự khác biệt rõ ràng là các đối tượng mới không cần một định danh mới. Chúng đã có một cái. Sự vi phạm của invariant được xử lý khác nhau. Khi một đối tượng mới được tạo ra từ đầu, mọi vi phạm của invariant đều kết thúc trong một ngoại lệ. 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.

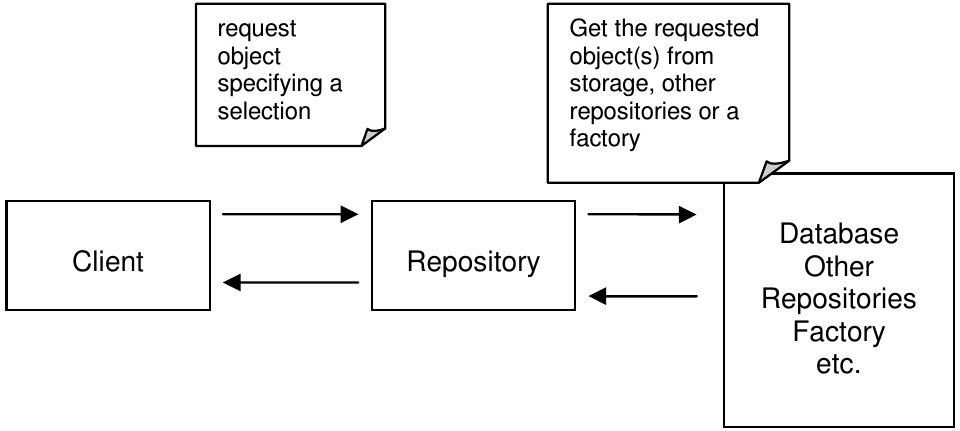
CSDL là một phần của lớp infrastructure. Một giải pháp *bần cùng* là cho client nhận biết chi tiết cách cần thiết để truy cập CSDL. Ví dụ như, client có thể tạo truy vấn SQL để lấy những dữ liệu cần thiết. Câu truy vấn dữ liệu có thể trả về một bộ bản ghi, thậm chí phơi bày các chi tiết bên trong nó. 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.

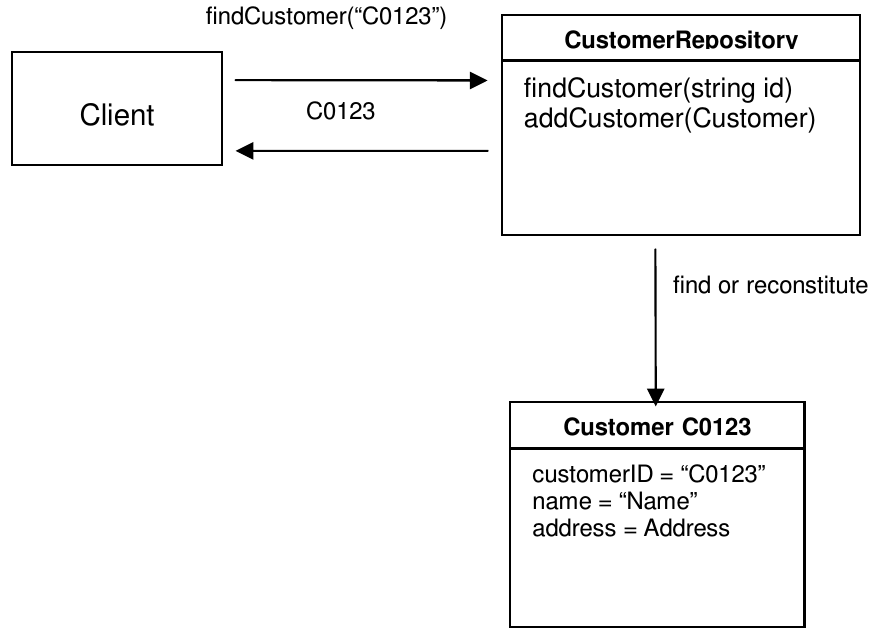
Repository có thể cũng chứa một Strategy. Nó có thể truy cập một bộ nhớ lưu trữ hoặc cách khác dựa trên Strategy chỉ định. Nó cũng có thể sử dụng các loại bộ nhớ khác nhau cho các loại khác nhau của các đối tượng. Hiệu quả chung là đối tượng domain được tách rời khỏi các nhu cầu lưu trữ đối tượng và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



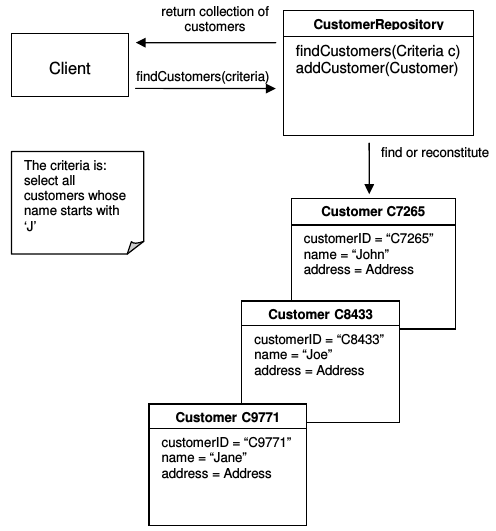
Với mỗi loại đối tượng cần thiết để truy cập toàn cục, tạo một đối tượng có thể cung cấp một tập hợp trong bộ nhớ ảo của tất cả các đối tượng trong các loại đó. Cài đặt truy cập thông qua một Interface được biết ở mức toàn cục. Cung cấp các phương thức để thêm hoặc xóa đối tượng, nhằm đóng gói việc thêm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

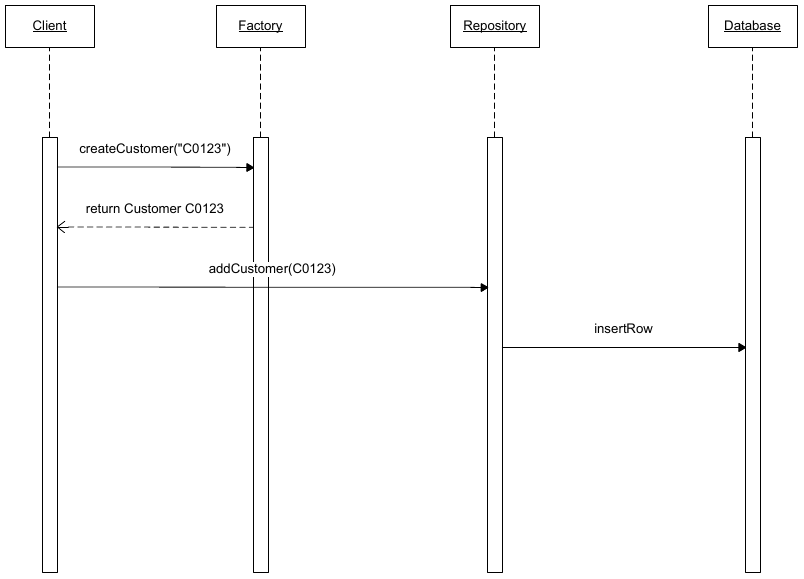
Việc cài đặt một repository có thể rất giống infrastructure, nhưng repository interface sẽ chỉ là một model thuần túy.



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.

1. a client object [↑](#footnote-ref-0)