**Chương 8** : Giới thiệu về mô hình miền.

Mô hình và trái tim của thiết kế tạo hình lẫn nhau

—Eric Evans

Cách đây 10 năm, một công ty tương đối trẻ và nhanh chóng thuê Dino cho đào tạo ASP.NET. Họ nói rằng họ chủ yếu muốn một số nội dung web nâng cao, nhưng họ cũng muốn tìm hiểu thêmVề những mô hình mới nổi giúp hình thành quan điểm khái niệm về lĩnh vực kinh doanh. Họ hỏi họ có thể dự trữ một ngày để có được một cái nhìn tổng quan về thiết kế hướng miền (DDD).

Những người có ít nhất một sự tò mò của DDD không nhiều vào thời điểm đó. May mắn thay, Andrea là một trong số họ và chúng tôi đã liên lạc. Vì vậy, Andrea đã cùng nhau đưa ra một mồi nhanh về DDD giúp Dino vô cùng, chuẩn bị thêm một ngày cho khách hàng. Nhân tiện, khách hàng đó bây giờ một nhà lãnh đạo toàn cầu trong ngành công nghiệp của mình, và chúng ta không thể không nghĩ rằng sự quan tâm của họ đối với CTO, DDD đã rất am hiểu và làm cho công ty có thể giải quyết hiệu quả sự phức tạp của một phát triển kinh doanh.

DDD ban đầu được thiết kế như là một cách tiếp cận để phát triển phần mềm bắt đầu từ một phổ biến ngôn ngữ, đi qua một bản đồ của các ngữ cảnh bị chặn, và kết thúc bằng một kiến trúc lớp được đề nghị tập trung vào một mô hình hướng đối tượng của lĩnh vực kinh doanh. Theo các nguyên tắc DDD, các bước phổ biến như yêu cầu hiểu biết, chuyển yêu cầu thành các thông số kỹ thuật, mã hóa thực tế và thử nghiệm đến một cách dễ dàng - hoặc ít nhất, đó là cách đơn giản nhất họ có thể đưa ra một mức độ khách quan cao phức tạp kinh doanh.

Chương này cung cấp cho bạn các lý do của DDD, nhấn mạnh vào canh chữ cái của lớp miền. Trong chương tiếp theo, chúng ta sẽ thấy DDD đang hoạt động được áp dụng cho một lĩnh vực kinh doanh cụ thể.

**Chuyển đổi dữ liệu sang hành vi**

Trong một thời gian dài, cách tiếp cận điển hình đối với sự phát triển đã ít nhiều như sau. Trước tiên, bạn thu thập yêu cầu và thực hiện một số phân tích để xác định cả các thực thể liên quan (ví dụ như khách hàng, đơn đặt hàng, sản phẩm) và các quy trình được triển khai. Thứ hai, được trang bị với sự hiểu biết này, bạn cố suy luận một mô hình dữ liệu vật lý (và chủ yếu quan hệ) có thể hỗ trợ các quy trình. Bạn đảm bảo mô hình dữ liệu phù hợp về mối quan hệ (các khoá chính, các ràng buộc, chuẩn, lập chỉ mục) và sau đó bắt đầu xây dựng các thành phần phần mềm đối với các bảng xác định các thực thể kinh doanh có liên quan nhất. Bạn cũng có thể dựa vào các tính năng cơ sở dữ liệu cụ thể như các thủ tục lưu trữ như là một cách để thực hiện hành vi trong khi vẫn giữ cơ cấu của cơ sở dữ liệu ẩn từ các cấp mã trên. Bước cuối cùng là tìm ra một mô hình thoải mái để đại diện cho dữ liệu và di chuyển nó lên đến lớp trình bày.

Cách tiếp cận này để mô hình hóa một lĩnh vực kinh doanh không phải là sai cho mỗi gia đình, và nó chỉ hoạt động. Gần như mọi nhà phát triển đều biết về nó; Đó là một thực hành trưởng thành và củng cố và, một lần nữa, nó chỉ hoạt động. Vậy tại sao lại nên thay đổi nó cho cái gì đó có một cái tên hấp dẫn hơn và kỳ lạ-DDD, nhưng cũng đại diện cho nhiều loại bước nhảy vào cái chưa biết?

**Lý do đằng sau các mô hình và tên miền**

Qua nhiều năm, hai người trong chúng ta đã nhiều lần chấp nhận thách thức giải thích các điểm cộng của một phương pháp tiếp cận dựa vào miền so với cách tiếp cận dựa vào cơ sở dữ liệu để thiết kế phần mềm. Và chúng ta thường thất bại. Tuy nhiên, hầu hết những lần đó, chúng tôi đã không thành công vì chúng tôi đã đặt toàn bộ điều này dưới dạng các đối tượng so với trình đọc dữ liệu.

**Nó không phải về việc sử dụng các đối tượng thay vì đọc dữ liệu**

Nhận thức rõ về những gì truy cập dữ liệu đã được trước khi .NET, chúng tôi nhận ra rằng sự sẵn có của ADO.NET là một mốc cho các nhà phát triển Microsoft Windows. ADO.NET là một lớp tương đối mỏng so với cơ sở hạ tầng của cơ sở dữ liệu quan hệ. Nhà cung cấp nguồn dữ liệu ngầm giấu các chi tiết có độ cằn cỗi của lưu trữ bên dưới và cho thấy các đối tượng .NET cho các con trỏ và các bộ kết quả. Mô hình làm việc rõ ràng và dễ dàng: bạn mở một kết nối, tạo một lệnh SQL, quyết định cách bạn muốn dữ liệu trở lại - hoặc là kết quả của một con trỏ chuột - hãy thực hiện lệnh và đóng kết nối. Tuyệt vời, các con trỏ ADO.NET cũng là các đối tượng hữu dụng và các tập kết quả thậm chí còn có thể tuần tự hóa được.

Trong ADO.NET, bạn phải đối mặt với một số rắc rối với việc chuyển đổi kiểu và đặt tên không mạnh như khi bạn làm việc với các lớp học. Ví dụ, để tham khảo cột bảng, bạn phải biết tên và tên là một chuỗi ký tự đơn giản. Bạn chỉ biết trong thời gian chạy nếu bạn đã nhập sai.

Thu hẹp khung nhìn của Microsoft .NET Framework, LINQ to SQL và Entity Framework được cung cấp là một thủ thuật nhanh chóng và dễ dàng để gói lên một cơ sở dữ liệu hiện tại thành một tập hợp các đối tượng dễ quản lý.

Để cắt ngắn một câu truyện ngắn, LINQ to SQL (bây giờ đã ngừng hoạt động) và Entity Framework bắt đầu với các công cụ Object / Relational Mapper (O / RM) với một số phép thuật khác để suy ra một mô hình đối tượng trong một cơ sở dữ liệu quan hệ mới hoặc hiện có.

Vâng, đây không phải là ý nghĩa của việc sử dụng một phương pháp tiếp cận dựa vào miền để thiết kế phần mềm.

Điều này chỉ đơn giản là viết một lớp truy cập dữ liệu bằng cách sử dụng các đối tượng tiện dụng thay vì các trình đọc dữ liệu ADO.NET hoặc DataSets. Và đó là một thành tựu lớn.

**Mô hình đối tượng liên tục không phải là mô hình miền**

Như chúng ta đã thảo luận trong Chương 5, "Khám phá kiến trúc miền", DDD là một cách tiếp cận không chỉ đòi hỏi phải viết mã truy cập dữ liệu bằng các đối tượng thay vì các bản ghi. DDD là chủ yếu về crunching kiến thức về miền và có ngôn ngữ phổ biến và bối cảnh bounded như là kết quả chính. Và hầu hết thời gian nhưng không nhất thiết mô hình này là một loại đặc biệt của một mô hình đối tượng và được gọi là "mô hình miền".

* Quan trọng là làm DDD thường được nhìn nhận đơn giản chỉ vì có một mô hình đối tượng với một số đặc điểm đặc biệt. Như chúng ta đã mô tả trong chương 5, DDD đặc biệt chú ý đến bối cảnh lập bản đồ và các mối quan hệ của họ, và sau đó là đưa ra cho mỗi ngữ cảnh một kiến trúc thích hợp. Kiến trúc mà Eric Evans đã đề nghị trong cuốn sách của ông là DomainDriven thiết kế: Khắc phục sự phức tạp trong trái tim của phần mềm (Prentice Hall, 2003) dựa trên trên một mô hình miền mô tả như là một mô hình đối tượng với các tính năng đặc biệt. Một vài thứ đã thay đổi từ đó. Ngày nay, việc lập bản đồ ngữ cảnh DDD là điều tối quan trọng và mô hình hóa tên miền thông qua các đối tượng chỉ là một trong những lựa chọn có thể - mặc dù là một trong những cách sử dụng rộng rãi nhất. Chúng ta sẽ đi với giả định rằng mô hình miền là một mô hình đối tượng trong chương này và kế tiếp.

Cuối cùng, một mô hình miền dựa trên đối tượng chỉ là một mô hình đối tượng dai dẳng, ngoại trừ nó được trang trí bởi một số đặc điểm rõ ràng là lạ: sự ưu tiên của các nhà máy trên các nhà thầu đơn giản, sử dụng phong phú các loại phức tạp, hạn chế sử dụng các kiểu nguyên thuỷ, Setters về tài sản, và phương pháp. Tất cả những đặc điểm rõ ràng kỳ lạ của tập các lớp học được gọi là "miền mô hình "tồn tại vì một lý do cụ thể.

**Mô hình là gì và tại sao bạn cần một**

Các chiếu Mercator là một đại diện đồ họa của bản đồ thế giới. Đặc điểm độc đáo của nó làm cho nó phù hợp với mục đích hải lý; Trên thực tế, phép chiếu Mercator là một sơ đồ tiêu chuẩn được sử dụng để xây dựng các khóa học hải lý. Seehttp: //en.wikipedia.org/wiki/Mercator\_projection để biết thêm chi tiết. Vì vậy,Mercator là một mô hình.

Liệu nó có phải là một mô hình thực sự phản ánh thế giới thực? Liệu nó có thực và lý tưởng?

Phép chiếu Mercator dựa trên thang đo tăng lên khi bạn tiến hành từ đường xích đạo lên và xuống đến cực. Gần cực, thang đo trở nên vô hạn. Mỗi hình vuông bạn nhìn thấy trên bản đồ đại diện cho bất kể kích thước-cùng một khu vực vật lý. Hiệu quả thuần của quy mô là một số khu vực trên thế giới dường như lớn hơn hoặc nhỏ hơn thực tế. Ví dụ: nếu bạn nhìn vào bố cục thế giới thông qua quan điểm của các chiếu Mercator, bạn có thể nói rằng Alaska là lớn như Brazil. Thực tế, thay vào đó, là khác nhau, như Brazil chiếm nhiều như ive lần diện tích Alaska.

Khi bạn vẽ một khóa học về phép chiếu thế giới Mercator, góc giữa đường và đường xích đạo vẫn không đổi trên tất cả các kinh tuyến. Điều này đã làm cho nó dễ dàng - đã trở lại trong các năm 1500 để đo các khóa học và vòng bi thông qua các thước đo góc.

Bản đồ Mercator là một mô hình làm biến dạng cả khu vực và khoảng cách; Về mặt này, nó không phù hợp để thể hiện trung thành bản đồ của thế giới như nó thực sự là. Bạn không muốn sử dụng nó thay vì, nói, Google Maps để đại diện cho một thành phố trên bản đồ. Mặt khác, nó là một mô hình tuyệt vời cho bối cảnh (bounded) nó được tạo ra cho bản đồ hải lý.

**Đó là tất cả về hành vi**

Mô hình miền nhằm mục đích thể hiện một cách trung thành nhất có thể các khái niệm kinh doanh cốt lõi cho ứng dụng để giải quyết. Mô hình tên miền là API của hoạt động kinh doanh cốt lõi và bạn muốn các lớp mô hình miền luôn luôn phù hợp với khái niệm kinh doanh mà chúng đại diện.

Khi thiết kế mô hình miền, bạn nên đảm bảo rằng không có cuộc gọi không chính xác đến API có thể có thể xác định rằng sau đó, bất kỳ đối tượng miền nào đang ở trong trạng thái không hợp lệ. Hạn chế về mô hình miền có mục đích duy nhất là tạo ra các lớp hoàn toàn phù hợp với các khái niệm kinh doanh cốt lõi và liên tục nhất quán.

Cần phải rõ ràng rằng để giữ liên tục phù hợp với doanh nghiệp, bạn nên tập trung thiết kế của mình vào hành vi nhiều hơn dữ liệu. Để làm cho DDD thành công, bạn phải hiểu cách hoạt động và làm cho nó với phần mềm. Phần mềm bạn viết chủ yếu là một mô hình miền.

**DDD luôn tốt cho tất cả mọi người**

Phần phân tích của DDD, như được giải thích trong Chương 5, là tốt cho bất kỳ dự án nào có tuổi thọ đáng kể. Đó là về phân tích cuối cùng, và nó bao gồm học các cơ học của tên miền và thể hiện rằng thông qua một từ vựng thích hợp. Nó không phải về mã hóa, và nó không phải về công nghệ.

Nó chỉ là những gì như là một kiến trúc sư bạn sẽ làm trong bất kỳ trường hợp phân tích và kiến trúc cấp cao nhất. Tuy nhiên, DDD thiết lập cách tiếp cận và giới thiệu khái niệm về ngôn ngữ phổ biến. Mô hình hoá miền là một khía cạnh khác của DDD gần với thiết kế phần mềm và mã hóa. Và các công thức DDD cho mô hình hóa miền có thể hoặc không thể hoạt động trong mọi trường hợp.

Cho đến một vài năm trước đây, DDD chỉ được coi là có khả năng đối với các hệ thống cực kỳ phức tạp được đặc trưng bởi một bộ quy tắc kinh doanh cực kỳ năng động. Lý do là chi phí thiết lập mô hình miền quá cao và dễ hấp thụ chỉ trong một dự án dài hạn. Các quyết định nội bộ kết quả từ kinh doanh ra khỏi cần phải có một mô hình bắt chước cấu trúc và hành vi của hệ thống và chi phí của việc đó. Xây dựng một mô hình đối tượng thành công bao gồm tất cả các mô hình có thể thực sự khó khăn, và nó đòi hỏi đúng kỹ năng và thái độ.

Đôi khi, bạn chỉ có thể quyết định rằng việc có một mô hình miền không phải là giá trị - và nó có thể là cả hai vì sự trở lại không được nhìn nhận là cao và bởi vì các chi phí, thay vào đó, được nhìn nhận là quá cao. Khi nói đến điều này, trong phân tích DDD của bạn, bạn chỉ cần chọn một kiến trúc hỗ trợ thay thế cho mỗi ngữ cảnh bị chặn.

**Cơ sở dữ liệu là cơ sở hạ tầng.**

Theo kinh nghiệm của chúng tôi, một trong những khía cạnh mà hầu hết mọi người lo ngại về DDD là bạn kết thúc với một mô hình có aura được agnostic cơ sở dữ liệu. Làm thế nào tôi có thể thiết kế một cách hiệu quả hệ thống của tôi và làm cho nó hoạt động như mong đợi - nhiều người nói - nếu tôi chuyển cơ sở dữ liệu xuống dưới cùng của mối quan tâm của tôi?

Có một quan niệm sai lầm rõ ràng ở đây.

Tất cả những gì DDD gợi ý là bạn tập trung vào các khái niệm kinh doanh và các quá trình của một bối cảnh bị giới hạn nhất định, tìm hiểu về chúng, và vẽ ra một hệ thống có thể mô phỏng các khái niệm và quy trình kinh doanh một cách trung thực. Hệ thống kết quả được tập trung vào một mô hình có chứa logic kinh doanh cốt lõi.

Miễn là bạn đang bận rộn thiết kế mô hình, cơ sở dữ liệu không phải là mối quan tâm chính của bạn. Nói một cách khác, DDD chỉ đơn giản nói rằng mô hình tên miền đến irst và tính kiên trì của tên miền được mô hình hóa đến sau đó. Nhưng nó chắc chắn sẽ đến. Một cách phổ biến để xây dựng khái niệm này là nói rằng cơ sở dữ liệu chỉ là một phần của lớp cơ sở hạ tầng, mặc dù phần nổi bật nhất.

**Một mô hình miền không có mối quan tâm bền bỉ**

Theo thiết kế, một mô hình miền không quan tâm đến tính bền bỉ. Mô hình này quan tâm đến các thực thể liên quan và các mối quan hệ, sự kiện của chúng và hành vi có thể quan sát được. Mô hình chỉ làm cho các thực thể như vậy thông qua các lớp học và phương pháp. Không có chồng chéo với cơ sở dữ liệu trong này.

Thuật ngữ kiên trì vô minh thường được trình bày như là một thuộc tính chính của một mô hình miền. Nhưng thuật ngữ này có nghĩa là gì?

Sự thiếu hiểu biết lâu dài có nghĩa là các lớp mô hình miền không nên được cung cấp các phương pháp để lưu hoặc thực hiện các thể hiện từ đĩa. Tương tự, các lớp trong mô hình miền không nên phơi bày các phương pháp đòi hỏi quyền truy cập vào lớp kiên trì để xác định những thứ như tổng số đơn hàng hoặc tình trạng của một khách hàng trong hệ thống khen thưởng của công ty.

**Một ứng dụng nên quan tâm đến sự bền bỉ**

Không có thứ như một hệ thống không cần sự kiên trì. Mô hình miền tốt hơn nếu nó bỏ qua mối quan tâm bền bỉ, nhưng cũng không thể đúng cho phần còn lại của ứng dụng. Tất cả các ứng dụng cần một số cơ sở hạ tầng để tồn tại và mối quan tâm xuyên suốt khác nhau như bảo mật, bộ nhớ đệm, và khai thác gỗ.

Ứng dụng DDD không phải là ngoại lệ.

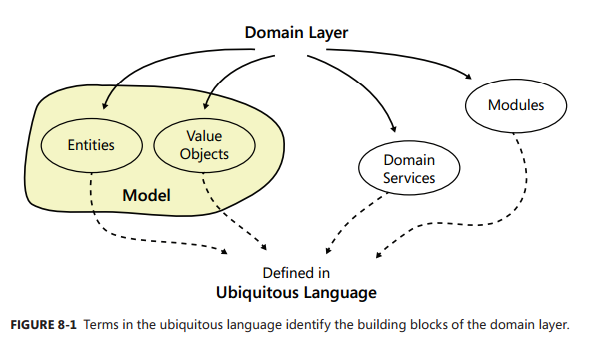
Hơn nữa, trong một kịch bản thực tế cơ sở dữ liệu hầu như luôn luôn là một hạn chế. Di chuyển dữ liệu vào cơ sở dữ liệu mới hiếm khi là một lựa chọn; Thường xuyên hơn, thay vào đó, bạn có một số hệ thống con kế thừa để tích hợp và giải quyết. Trong tình huống này, tập trung vào thiết kế dữ liệu thậm chí làm cho không có ý nghĩa. Bạn sẽ tập trung tốt hơn vào một mô hình định hướng kinh doanh mà có thể nối tiếp nhưng tách biệt một cách rõ ràng với lớp kiên trì. Đầu tiên bạn nhận được mô hình; Tiếp theo bạn ánh xạ nó vào một cơ sở dữ liệu mới hoặc hiện có.

Một mô hình miền được serializable nếu nó có thể được lưu vào cơ sở dữ liệu. Tính bền bỉ thường xảy ra thông qua các công cụ đối tượng / công cụ lập bản đồ quan hệ (O / RM). Nếu bạn sử dụng một công cụ O / RM để tồn tại, mô hình miền

Có thể phụ thuộc vào một số yêu cầu được đặt bởi công cụ O / RM. Một ví dụ điển hình là khi bạn cần cung cấp cho một lớp mô hình miền một hàm tạo tham số không cần thiết khác nếu không để cho Entity Framework (hoặc NHibernate) thực hiện đối tượng sau một truy vấn.

Cuối cùng, mô hình miền cần phải độc lập với các chi tiết triển khai hiệu quả, nhưng thường thì công nghệ O / RM có thể đặt ra một số hạn chế nhỏ đối với mô hình.

**Note** Thuật ngữ dịch vụ trong ngữ cảnh này chỉ đề cập đến các mảng logic kinh doanh và nó không có tham chiếu đến bất kỳ công nghệ nào (ví dụ Windows Communication Foundation), kiến trúc (kiến trúc hướng dịch vụ) và cơ sở hạ tầng web (dịch vụ web).

****

**Modules**

Khi bạn chuyển một mô hình miền thành phần mềm, bạn xác định một hoặc nhiều mô-đun. Các mô-đun cuối cùng chứa các đối tượng và phân vùng toàn bộ miền để tất cả các mối quan tâm bề mặt trong mô hình miền được rõ ràng và rõ ràng tách biệt. Về mặt cụ thể, một mô đun DDD chỉ giống như không gian tên .NET mà bạn sử dụng để tổ chức các lớp học trong một dự án thư viện lớp.

Trong .NET, bạn sử dụng không gian tên để nhóm các loại liên quan. Trong các giai đoạn tái cấu trúc, không gian tên NET có thể được đổi tên và thay đổi các loại của mỗi loại. Tương tự như vậy, mô-đun DDD có thể thay đổi theo thời gian để hiểu rõ hơn về sự hiểu biết hiện tại về lĩnh vực kinh doanh và các đường nét khái niệm thực tế của các bộ phận khác nhau của mô hình.

Nếu bạn nhận ra rằng, tại một số điểm, các đường nét khái niệm khác nhau tồn tại hơn những đường hiện có, bạn nên sửa đổi việc thực hiện mô hình để phản ánh điều đó. Khung khái niệm DDD và bê tông không gian tên NET nên luôn được giữ nguyên đồng bộ.

Trong một bối cảnh bị ràng buộc, một mô hình miền được tổ chức theo mô-đun. Trong một dự án .NET, một mô hình miền được tổ chức trong không gian tên, và các nhóm không gian tên có thể được cô lập trong các thư viện lớp khác biệt. Dưới đây là một bản tóm tắt của các cardinalities liên quan:

* + Một bối cảnh bị ràng buộc có một mô hình miền.
  + Một ngữ cảnh bị ràng buộc có thể có nhiều mô-đun.
  + Một mô hình miền có thể liên quan đến một số mô-đun.

**Value objects**

Mô-đun DDD chứa các đối tượng và đối tượng giá trị. Cuối cùng, cả hai được kết xuất dưới dạng các lớp .NET, nhưng các thực thể và các đối tượng giá trị đại diện cho các khái niệm khá khác nhau và dẫn đến các chi tiết thực hiện khác nhau.

Trong DDD, một đối tượng giá trị được thực hiện đầy đủ bởi các thuộc tính của nó. Các thuộc tính của một đối tượng giá trị không bao giờ thay đổi sau khi cá thể đã được tạo ra. Nếu nó thay đổi, đối tượng giá trị sẽ trở thành một thể hiện của một đối tượng giá trị khác được nhận dạng hoàn toàn bởi bộ sưu tập các thuộc tính mới. Trong .NET, các đối tượng giá trị DDD được gọi là các loại không thay đổi. Các loại Int32 và String là các loại bất biến phổ biến nhất trong .NET Framework

**Entities.**

Tất cả các đối tượng có các thuộc tính, nhưng không phải tất cả các đối tượng đều được nhận biết đầy đủ bởi bộ sưu tập các thuộc tính của chúng. Khi các thuộc tính không đủ để đảm bảo duy nhất, và khi tính duy nhất quan trọng đối với đối tượng cụ thể, bạn có các thực thể DDD. Nói một cách khác, nếu đối tượng cần một thuộc tính ID để theo dõi nó một cách độc đáo trong suốt ngữ cảnh cho toàn bộ vòng đời, đối tượng có một nhận dạng và được cho là một thực thể.

Trong cuốn sách của mình, Evans sử dụng một ví dụ từ một ứng dụng ngân hàng để minh họa sự khác nhau giữa các thực thể và các đối tượng giá trị và giải thích lý do tại sao cả hai khái niệm đều có liên quan đến mô hình miền. Làm thế nào bạn có thể xem xét hai tiền gửi của cùng một số tiền vào cùng một tài khoản trong cùng một ngày? Họ có cùng một hoạt động được thực hiện hai lần, hoặc là họ hoạt động riêng biệt mà mỗi xảy ra một lần?

Bạn có hai giao dịch ngân hàng riêng biệt và tên miền yêu cầu bạn phải đối xử độc lập với nhau. Bạn luôn có thể đề cập đến giao dịch sau đó và tên miền phải chỉ rõ điều gì cần thiết để nhận diện đối tượng một cách độc nhất. Chúng ta sẽ trở lại các thực thể và các đối tượng giá trị trong một khoảnh khắc và thảo luận chi tiết hơn những gì nó có ý nghĩa để tạo ra các lớp cho mỗi.

Các đối tượng giá trị chỉ là dữ liệu tổng hợp lại với nhau; Các thực thể thường được làm bằng dữ liệu và hành vi. Khi nói đến hành vi, tuy nhiên, điều quan trọng là phải phân biệt logic miền với logic ứng dụng. Domain logic đi trong lớp tên miền (mô hình hoặc dịch vụ); Việc thực hiện các trường hợp sử dụng đi vào lớp ứng dụng. Ví dụ, một thực thể Order thực chất là một tài liệu đưa ra cái gì khách hàng muốn mua. Logic liên kết với chính thực thể Đặt hàng có liên quan đến nội dung của tài liệu-thuế và tính chiết khấu, chi tiết đơn đặt hàng, hoặc có lẽ ngày ước tính thanh toán. Toàn bộ quá trình hoàn thiện, trạng thái theo dõi hoặc chỉ lập hóa đơn cho đơn đặt hàng nằm ngoài logic miền của đơn đặt hàng. Chúng chủ yếu sử dụng - trường hợp liên quan đến thực thể, và các dịch vụ có trách nhiệm thực hiện.

**Persistence of entities**

Một mô hình miền phải được liên tục, nhưng, như đề cập nhiều lần, nó không chăm sóc trực tiếp sự kiên trì của chính nó. Không có gì trong việc thực hiện mô hình miền đề cập đến việc tải và lưu hoạt động; Tuy nhiên, các hoạt động này là cần thiết để thực hiện các thể hiện của các thực thể miền và để thực hiện logic kinh doanh.

Một hương vị đặc biệt của các thành phần-kho-chăm sóc kiên trì thay mặt các thực thể.

Các kho lưu trữ thường được gọi từ bên ngoài mô hình miền - ví dụ: từ lớp ứng dụng - hoặc từ các thành phần khác của miền, chẳng hạn như dịch vụ miền. Hợp đồng cho các kho lưu trữ nằm trong lớp tên miền; Việc thực hiện các kho lưu trữ thuộc về, thay vào đó, với một lớp khác - tầng cơ sở.

Các tuyên bố rằng kho lưu trữ chăm sóc của các thực thể kiên trì là chính xác cho những gì chúng tôi đã thảo luận cho đến nay. Nó không chính xác ở góc độ rộng hơn bạn sẽ đạt được khi kết thúc chương. Điểm mấu chốt là các kho lưu trữ vẫn tồn tại tổng thể, một tập hợp con đặc biệt của các thực thể.

**Aggregates.**

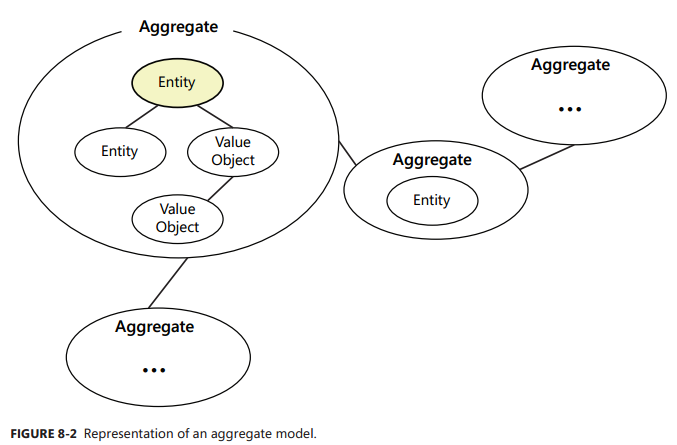
Khi bạn trải qua các trường hợp sử dụng được tìm thấy trong các yêu cầu và tìm ra mô hình miền của cho bối cảnh bị ràng buộc, nó không phải là bất thường mà bạn thấy một vài cá nhân được liên tục sử dụng và tham chiếu với nhau. Theo cách này, điều này có thể được xem như là sự xuất hiện của mã nguồn dữ liệu múi các đối tượng có liên quan đến logic đang được điều trị riêng lẻ thay vì được nhóm và xử lý một cách hợp lý như một đối tượng duy nhất.

Tổng hợp về cơ bản là một ranh giới nhất quán để nhóm và tách các thực thể trong mô hình. Thiết kế cốt liệu được lấy cảm hứng từ các giao dịch kinh doanh theo yêu cầu của hệ thống. Tính nhất quán thường là giao dịch, nhưng trong một số trường hợp, nó cũng có thể mang dạng nhất quán cuối cùng.

**Devising the aggregate model.**

Trong một mô hình miền, tập hợp nhiều thực thể trong một vùng chứa được gọi là tổng hợp. Để sử dụng các từ của Evans, "Tổng hợp là một nhóm các đối tượng liên quan mà chúng tôi coi như một đơn vị duy nhất với mục đích thay đổi dữ liệu."

Mô hình kết quả là kết hợp các tập hợp, các thực thể riêng lẻ, và các đối tượng giá trị của chúng, như trong Hình 8-2.



Các thực thể trong mô hình được chia thành các tập hợp. Hầu hết các tập hợp được thực hiện bởi nhiều thực thể ràng buộc nhau bởi mối quan hệ lẫn nhau giữa cha mẹ và con của họ. Các tập hợp khác chỉ được thực hiện bởi các thực thể riêng lẻ mà vẫn liên quan đến các tập hợp khác nhưng không có các thực thể con.

Mỗi tổng hợp đều có một ranh giới hợp lý chỉ đơn giản là xác định các thực thể nào nằm trong tổng thể. Quan trọng hơn, một tổng hợp có một thực thể gốc, được gọi là gốc tổng hợp. Cốt lõi tổng hợp là gốc của một đồ thị các đối tượng; Nó đóng gói các thực thể chứa và hành động như là proxy cho họ.

**Sử dụng bất biến nghiệp vụ để khám phá các tập hợp.**

Cốt liệu là một phần của thiết kế; Các kiến trúc sư và nhà phát triển sẽ tập hợp lại để hiểu rõ hơn về mã doanh nghiệp hiện tại. Trong một ngữ cảnh bị ràng buộc, các tập hợp đại diện cho các điều kiện bất biến trong mô hình miền. Một điều kiện bất biến về cơ bản là một quy tắc nghiệp vụ được yêu cầu phải được kiểm tra thường xuyên trong một mô hình miền. Nói một cách khác, tập hợp là một cách để đảm bảo tính nhất quán trong kinh doanh.

Nói chung, có hai loại tính nhất quán: tính nhất quán giao dịch và sự nhất quán cuối cùng. Tính nhất quán giao dịch đề cập đến tổng hợp để duy trì sự nhất quán sau mỗi giao dịch kinh doanh (tức là hoạt động) xảy ra trong miền. Sự nhất quán cuối cùng đề cập đến tính nhất quán, rằng tại một số điểm được đảm bảo sẽ được bảo toàn nhưng có thể không được đảm bảo sau mỗi lần giao dịch kinh doanh. Tổng hợp không hề quan tâm đến sự nhất quán và các hoạt động diễn ra bên ngoài ranh giới.

Khi suy nghĩ về tập hợp, bạn sẽ ind rằng xấp xỉ irsting thường đi kèm với tâm trí là một tổng hợp là một đồ thị của các đối tượng. Bạn có đơn đặt hàng có chứa các đơn hàng, chứa các sản phẩm, có chứa các loại, vân vân. Trong khi một tổng hợp có thể chứa và thường nó chứa một đồ thị của các đối tượng, cardinality của đồ thị thường được giới hạn trong hai hoặc ba đối tượng.

**Important** Trong mô hình miền được thiết kế tốt, chỉ có một tổng hợp được điều chỉnh trong mỗi giao dịch. Nếu tại một số điểm bạn ind rằng một trường hợp sử dụng nhất định dường như yêu cầu nhiều aggregate được điều chỉnh trong nhiều giao dịch, bạn có lẽ không phải trong một thiết kế tối ưu, mặc dù nó là chấp nhận được. Thay vào đó, khi trường hợp sử dụng dường như chỉ ra rằng nhiều uẩn nên được cập nhật trong một giao dịch, đó là dấu hiệu rõ ràng rằng cái gì đó sai ở đâu đó. Rất có thể, bạn đang bỏ lỡ bất biến và sự phân rã trong tổng hợp bạn tạo ra ít hơn lý tưởng cho việc kinh doanh.

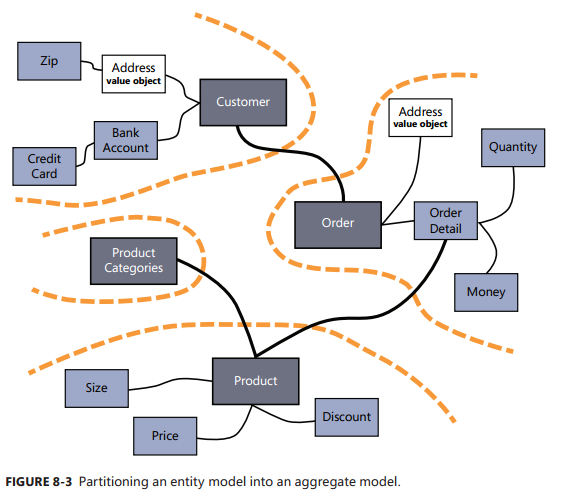
**Lợi ích của một mô hình tổng hợp**

Có một vài yếu tố quan trọng để tạo ra một mô hình tổng hợp trong phạm vi của một mô hình miền. Ở vị trí irst, việc thực hiện toàn bộ mô hình miền và logic kinh doanh của nó trở nên dễ dàng hơn bởi vì bạn làm việc với các đối tượng ít hơn và thô hơn và có ít mối quan hệ hơn.

Nếu điều này có vẻ mâu thuẫn với nguyên tắc SOLID (Trách nhiệm duy nhất, Mở / Đóng, Nguyên tắc của Liskov, Phân tách Giao diện, và Nguyên tắc đảo ngược) và có vẻ như nó có thể dẫn đến các đối tượng cồng kềnh, hãy xem xét những điều sau:

* + Mô hình tổng hợp là một nhóm hoàn toàn hợp lý. Lớp gốc tổng thể có thể trải qua một số thay đổi vì vai trò của nó, và những thay đổi này chủ yếu là đối phó với việc quản lý việc truy cập bên ngoài đối với các thực thể con.
  + Về mặt mã, các lớp thực thể vẫn là các lớp khác biệt và mỗi lớp vẫn giữ nguyên việc thực hiện. Tuy nhiên, bạn không muốn có nhà xây dựng công cộng trên các thực thể con, bởi vì những người có thể phá vỡ sự thống nhất của tổng hợp
  + Mô hình tổng hợp không nhất thiết phải giới thiệu các lớp mới mà nhóm mã của các lớp nhúng logic; Tuy nhiên, đối với tập hợp tiện lợi thường được mã hoá giống như các lớp học mới bao trùm toàn bộ đồ thị của các đối tượng con. Khi điều này xảy ra, đó là vì bạn muốn bảo vệ đồ thị càng nhiều càng tốt từ truy cập bên ngoài. Ranh giới của tổng hợp là rõ ràng trong mã trong trường hợp này.

Mô hình tổng hợp không nhất thiết phải giới thiệu các lớp mới mà nhóm mã của các lớp nhúng logic; Tuy nhiên, đối với tập hợp tiện lợi thường được mã hoá giống như các lớp học mới bao trùm toàn bộ đồ thị của các đối tượng con. Khi điều này xảy ra mô hình tổng hợp làm tăng mức độ trừu tượng và gói gọn nhiều thực thể trong một đơn vị duy nhất. Tần số cardinality của mô hình giảm và, cùng với đó, số lượng các mối quan hệ giữa các thực thể được thực thi trong mã cũng giảm. Hãy xem hình 8-3, mở rộng ý tưởng Hình 8-2 cho một kịch bản thực tế, bởi vì bạn muốn bảo vệ đồ thị càng nhiều càng tốt từ việc truy cập bên ngoài. Ranh giới của tổng hợp là rõ ràng trong mã trong trường hợp này.



Mô hình tổng hợp lớn khác là nó giúp ngăn chặn một mô hình kết hợp chặt chẽ, nơi mà mỗi thực thể có tiềm năng liên kết với một số người khác. Như bạn thấy trong hình 8-3, ranh giới có thể xác định nhiều thực thể được liên kết với gốc hoặc chỉ một thực thể là gốc của một cụm nút đơn.

Ví dụ kinh điển là tổng hợp bắt nguồn từ thực thể Đặt hàng trong một hệ thống thương mại điện tử. Một đơn đặt hàng được làm bằng một tập hợp các thực thể OrderItem. Cả hai khái niệm đều được biểu hiện thông qua các thực thể có nhận dạng và cuộc sống, nhưng OrderItem là một đứa trẻ của Order theo nghĩa là logic nghiệp vụ có thể không đòi hỏi phải truy cập vào một đơn đặt hàng bên ngoài ngữ cảnh của một đơn đặt hàng.

**Important** Rõ ràng, ranh giới thực sự của cốt liệu được xác định độc quyền bằng cách xem xét các yêu cầu và các quy tắc kinh doanh. Mặc dù, nói chung, một đơn đặt hàng dự kiến sẽ là con của một lệnh, có thể có một giả thuyết giả thuyết mà OrderItem tạo thành gốc của một tổng hợp khác biệt. Một ví dụ thậm chí tốt hơn là Địa chỉ, có thể là một đối tượng giá trị trong một số lĩnh vực và một thực thể ở những nơi khác, chẳng hạn như các công ty tiện ích (điện và điện thoại).

Một thực thể hoặc là một gốc tổng hợp hoặc được bao gồm trong một tập hợp. Một ví dụ điển hình là thực thể sản phẩm. Khi bạn nhìn vào đơn đặt hàng, hóa ra sản phẩm được tham chiếu trong các mục đặt hàng. Vì vậy, suy nghĩ là sản phẩm thuộc về Tổng hợp lệnh. Tuy nhiên, tiếp theo bạn có thể sử dụng các trường hợp sử dụng để xử lý các sản phẩm bên ngoài các đơn đặt hàng - ví dụ như trong danh mục sản phẩm. Điều này làm cho một tổng hợp bắt nguồn từ Sản phẩm. Tương tự với ProductCategories: miễn là các danh mục có thể thay đổi (ví dụ, xứng đáng với bảng của họ), bạn cần tổng hợp và nó có thể sẽ là một tổng hợp đơn.

Lời cuối cùng về ranh giới của một tổng hợp thuộc về ngôn ngữ phổ biến. Nếu ngôn ngữ nói rằng đơn đặt hàng chứa thông tin và số lượng sản phẩm, rõ ràng rằng Đơn hàng không tham chiếu sản phẩm nhưng một đối tượng giá trị thay thế.

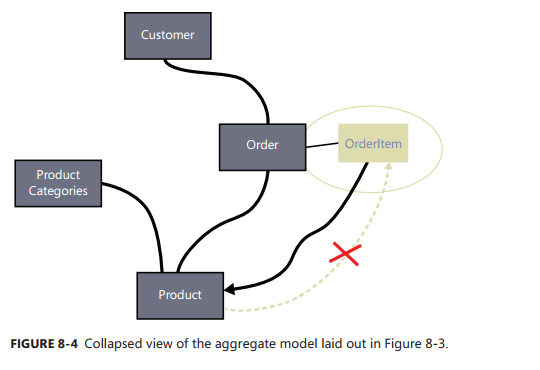
Điều gì về các đối tượng giá trị? Chúng là một phần không thể thiếu của các uẩn vì chúng được tham chiếu từ thực thể. Các đối tượng giá trị có thể được tham chiếu bởi nhiều thực thể, và sau đó, chúng có thể được sử dụng trong nhiều tập hợp.

Về mặt thiết kế, có thể sẽ rất thú vị khi xem xét khái niệm Địa chỉ. Chúng tôi trình bày Địa chỉ là một trong những ví dụ điển hình của một đối tượng giá trị. Về vấn đề này, một địa chỉ được sử dụng để đại diện cho nơi ở của khách hàng cũng như địa chỉ giao hàng của một đơn đặt hàng.

Giả sử bây giờ yêu cầu thay đổi hoặc sự hiểu biết của bạn về hệ thống được cải thiện. Kết quả là, bạn nghĩ tốt hơn nên cung cấp cho các địa chỉ một nhận dạng và lưu trữ chúng dưới dạng các thực thể riêng lẻ trên một bảng riêng biệt. Điều đó sẽ ngay lập tức được phản ánh trong mô hình tổng hợp vì bây giờ Địa chỉ trở thành gốc rễ của bản thân tổng thể; Có thể, đó là một tổng hợp đơn. Sau đó, hai mối quan hệ mới xuất hiện: Khách hàng Địa chỉ và Đặt hàng đến địa chỉ.

**Mối quan hệ giữa các cốt liệu**

Thực hiện mô hình tổng hợp đòi hỏi những thay đổi tối thiểu đối với các lớp thực thể liên quan nhưng nó có ảnh hưởng rất lớn đến phần còn lại của mô hình miền - dịch vụ và các kho lưu trữ nói riêng. Hình 8-4 chỉ ra cách sử dụng các aggregate làm giảm cardinality của các đối tượng có thể thực thi trong mã và giúp ngăn chặn một hệ thống bloated



Tổng hợp các lớp gốc ẩn lớp liên quan từ người gọi và yêu cầu người gọi tham chiếu đến chúng cho bất kỳ tương tác. Nói cách khác, một thực thể chỉ được phép tham chiếu một thực thể trong cùng một tập hợp hoặc gốc của một tập hợp khác. Đồng thời, một thực thể con như OrderItem trong hình 8-4 có thể giữ một tham chiếu đến một gốc tổng hợp khác.

**Marking aggregate roots.**

Một đối tượng gốc tổng thể là gốc của cụm các đối tượng liên kết tạo thành tổng hợp.

Một root tổng thể có khả năng hiển thị toàn cầu trong suốt mô hình miền và có thể được tham chiếu trực tiếp. Các thực thể bên trong tổng thể vẫn có bản sắc và cuộc sống, nhưng chúng không thể được tham chiếu trực tiếp từ bên ngoài tổng hợp.

Đây là nền tảng của mô hình tổng hợp.

Một đối tượng (ví dụ: một dịch vụ) có thể sử dụng, ví dụ, địa chỉ của khách hàng hoặc các mặt hàng của một đơn đặt hàng. Truy cập vào thông tin này đóng gói, tuy nhiên, phải luôn luôn xảy ra thông qua các gốc. Người gọi nên xem xét bất kỳ tài liệu tham khảo đến các đối tượng nội bộ mà họ nhận được như thoáng qua và tránh sử dụng chúng vượt quá phạm vi của một cuộc gọi duy nhất.

Đối tượng gốc gộp có các trách nhiệm liên quan. Đây là danh sách:

* + Tổng hợp gốc đảm bảo rằng chứa các đối tượng luôn ở trong trạng thái hợp lệ theo các quy tắc kinh doanh hiện hành (nhất quán). Ví dụ, nếu một quy tắc yêu cầu rằng không có chi tiết của đơn hàng có thể được cập nhật sau khi nó đã được vận chuyển, gốc phải làm cho cập nhật này không thể trong mã.
  + Sau đó, rễ tổng hợp chịu trách nhiệm cho sự bền bỉ của tất cả các vật thể đóng gói.
  + Sau đó, rễ tổng hợp chịu trách nhiệm cho việc cập nhật và xóa bỏ tầng thông qua các thực thể trong tổng thể.
  + Sau đó, một hoạt động truy vấn chỉ có thể lấy ra một gốc tổng hợp. Truy cập vào các đối tượng nội bộ phải luôn luôn xảy ra bằng cách điều hướng thông qua giao diện của root tổng hợp.

Cần phải rõ ràng rằng hầu hết các thay đổi mã cần thiết để hỗ trợ các quy tắc của một mô hình tổng thể xảy ra ở mức độ dịch vụ và kho. Trong điều khoản của mã, có nghĩa là gì để trở thành một tổng thể gốc?

Một root tổng hợp thường thực hiện một giao diện thường được gọi là IAggregateRoot, như được hiển thị ở đây:

public class Order : IAggregateRoot

{

...

}

Giao diện không yêu cầu nghiêm ngặt bất kỳ chức năng nào. Nó có thể dễ dàng là một giao diện đánh dấu đơn giản chỉ được sử dụng để làm sáng tỏ với thế giới bên ngoài rằng một thực thể thực sự là một gốc.

public interface IAggregateRoot

{

}

Một rễ tổng thể cũng có thể được thực hiện một cách phức tạp hơn, tổng thể, có thể làm cho việc triển khai mô hình mạnh mẽ hơn:

public interface IAggregateRoot {

bool CanBeSaved { get; }

}

Bằng cách làm cho thuộc tính mù quáng trở lại đúng, bạn sẽ trở lại với kịch bản khi IAggregateRoot chỉ là một điểm đánh dấu và giao diện không có thành viên. Bằng cách thêm một số logic trong getters, thay vào đó, bạn có thể xác nhận rằng trạng thái tổng hợp đã sẵn sàng để lưu hoặc xóa. Dưới đây là một cách để sử dụng biến thể của giao diện:

public class Order : IAggregateRoot

{

bool IAggregateRoot.CanBeSaved

{

get { return Validate(); }

}

...

}

Giao diện gốc gộp thường được sử dụng như một ràng buộc kiểu cho các lớp kho lưu trữ. Bằng cách này, trình biên dịch thực thi các quy tắc mà chỉ có tổng hợp rễ có kho và xử lý sự kiên trì.

**Note** Một quy tắc đơn giản để giúp các nhà phát triển có nền tảng cơ sở dữ liệu mạnh mẽ hướng tới bản thân họ trong thế giới DDD là như sau. Các thực thể là những thứ mà, trong một quan hệ mô hình, bạn muốn có một bảng riêng biệt. Các uẩn là sự kết hợp của các bảng liên kết với nhau bởi các mối quan hệ nước ngoài. Cuối cùng, một root tổng thể là một bảng duy nhất trong một biểu đồ của các mối quan hệ nước ngoài quan trọng chỉ chứa các tài liệu tham khảo ngoại khoá ra nước ngoài.

**Domain services**

Dịch vụ tên miền là các lớp có các phương pháp thực hiện logic miền không thuộc về một tổng hợp cụ thể và nhiều khả năng là nhiều thực thể. Dịch vụ tên miền phối hợp hoạt động của các tập hợp và các kho với mục đích thực hiện hành động kinh doanh. Trong một số trường hợp, dịch vụ miền có thể sử dụng dịch vụ từ cơ sở hạ tầng, chẳng hạn như khi gửi email hoặc tin nhắn văn bản là cần thiết.

Khi một lô logic kinh doanh nào đó không có trong bất kỳ tập hợp và tổng hợp hiện tại không thể được thiết kế lại theo cách mà hoạt động của nó, bạn thường nhìn vào sử dụng các dịch vụ tên miền. Vì vậy, trong một cách nào đó, các dịch vụ tên miền là phương thức cuối cùng cho logic mà không phải bất cứ nơi nào khác.

Tuy nhiên, các dịch vụ tên miền không phải là điều bình thường, các hạng mục công nhân trong một mô hình miền. Tác vụ họ thực hiện đến từ yêu cầu và được các chuyên gia về miền chấp thuận. Cuối cùng nhưng không kém phần quan trọng, tên được sử dụng trong các dịch vụ tên miền là một phần của ngôn ngữ phổ biến.

**Services as contracts**

Nó là phổ biến để đại diện cho dịch vụ tên miền thông qua các giao diện và các lớp thực hiện. Khi bạn sử dụng giao diện, hầu hết thời gian bạn dự định bảo mật testability hoặc tạo ra một điểm mở rộng trong phần mềm của bạn.

Đây không phải là trường hợp đối với các dịch vụ tên miền.

Giao diện của các dịch vụ tên miền đại diện cho một hợp đồng được viết bằng ngôn ngữ phổ biến và liệt kê các hành động có sẵn cho các dịch vụ ứng dụng để gọi.

**Cross-aggregate behavior**

Kịch bản phổ biến nhất cho các dịch vụ tên miền là thực hiện bất kỳ hành vi nào liên quan đến nhiều tập hợp, phương pháp truy cập cơ sở dữ liệu hoặc cả hai. Một ví dụ điển hình là một dịch vụ tên miền xác định liệu một khách hàng nhất định có đạt được trạng thái của khách hàng "vàng" hay không. Giả sử miền nói rằng khách hàng kiếm được trạng thái của khách hàng "vàng" sau khi vượt quá ngưỡng cho trước của các đơn đặt hàng trên một phạm vi sản phẩm đã chọn.

Làm thế nào bạn sẽ kiểm tra xem?

Bạn không thể có một cách hợp lý phương pháp IsGold trên tổng số Khách hàng tính trạng thái. Tính trạng thái đòi hỏi phải truy cập vào cơ sở dữ liệu - không phải là công việc thích hợp để tổng hợp. Một dịch vụ miền sau đó là hoàn hảo nó. Một dịch vụ trên thực tế có thể truy vấn các đơn đặt hàng và các sản phẩm cho khách hàng.

Cấu trúc của mô hình phần lớn ảnh hưởng đến số lượng và giao diện cho các dịch vụ miền.

Xem xét một kịch bản đặt chỗ. Bạn có một thành viên câu lạc bộ muốn đặt một tài nguyên-nói, một phòng họp. Hành động đặt phòng yêu cầu áp dụng một số bước logic kinh doanh, chẳng hạn như kiểm tra tính khả dụng và xử lý thanh toán. Có ít nhất hai cách bạn có thể làm điều này. Bạn có thể có một dịch vụ tên miền đặt chỗ đọc trạng thái tín dụng của thành viên và kiểm tra tính khả dụng của phòng hoặc bạn có thể có một thực thể Booking đề cập đến sự phụ thuộc của thành viên và phòng. Kho của thực thể Booking sau đó chịu trách nhiệm cho toàn bộ nhiệm vụ. Cả hai tùy chọn đều có giá trị như nhau bởi vì cả hai đều có hiệu quả mô hình miền.

**Note** Trong cuốn sách của mình, Evans sử dụng một ví dụ tương tự để minh họa trường hợp cho một dịch vụ miền. Ông đề cập đến dịch vụ chuyển khoản ngân hàng để làm ví dụ về dịch vụ tên miền hoạt động trên nhiều đơn vị (tài khoản ngân hàng).

**Repositories**

Các kho lưu trữ là loại dịch vụ tên miền phổ biến nhất và họ quan tâm đến sự tập trung dữ dội. Bạn sẽ có một kho trên mỗi tập hợp gốc: CustomerRepository, OrderRepository, v.v.

Một thực tiễn phổ biến là trích xuất một giao diện ra khỏi các lớp này và làm các giao diện trong cùng một thư viện lớp như mô hình lõi. Tuy nhiên, việc triển khai các giao diện kho lưu trữ thuộc về lớp cơ sở hạ tầng. Kho lưu trữ thường dựa trên một giao diện chung như IRepository:

public interface IRepository<T> where T:IAggregateRoot

{

// You can keep the interface a plain marker or

// you can have a few common methods here.

T Find (object id);

void Save (T item);

}

Các kho lưu trữ cụ thể sau đó được bắt nguồn từ giao diện cơ sở:

public interface IOrderRepository : IRepository<Order>

{

// The actual list of members is up to you

...

}

Lưu ý rằng không có cách nào để xây dựng một kho hoàn toàn đúng hay sai. Một kho lưu trữ tốt cũng có thể chỉ đơn giản là một lớp dựa trên giao diện với một danh sách các thành viên được bố trí khi bạn cần chúng. Bạn thậm chí có thể bỏ qua giao diện IAggregateRoot và chỉ cần thực hiện sự kiên trì cho toàn bộ đồ thị vì nó phải là:

public interface IOrderRepository

{

// The actual list of members is up to you

...

}

Các thành viên của một kho lưu trữ thực sự sẽ thực hiện truy cập dữ liệu - hoặc là truy vấn, cập nhật, hoặc chèn. Công nghệ được sử dụng để truy cập dữ liệu tùy thuộc vào bạn. Ngày nay, bạn sử dụng O / RM như Entity Framework, nhưng không có gì ngăn cản bạn sử dụng ADO.NET, các thủ tục được lưu trữ đồng thời, hoặc tại sao không phải là một cửa hàng NoSQL.

Một triển khai thú vị của giao diện IRepository <T> bao gồm việc đánh dấu giao diện với một lớp Hợp đồng mã để CLR có thể đưa một số mã ngôn ngữ bậc trung (Intermediate Language) vào bất kỳ triển khai các phương pháp IRepository:

[ContractClass(typeof(RepositoryContract<>))]

public interface IRepository<T> where T:IAggregateRoot

{

...

}

Và đây là việc thực hiện lớp hợp đồng:

[ContractClassFor(typeof(IRepository<>))]

sealed class RepositoryContract<T> : IRepository<T> where T : IAggregateRoot

{

public void Save(T item)

{

Contract.Requires<ArgumentNullException>(item != null, "item");

Contract.Requires<ArgumentException>(item.CanBeSaved);

throw new NotImplementedException();

}

}

Khi phương thức Save được triệu gọi, phương thức CanBeSaved trên aggregate được cung cấp được tự động gọi là chăm sóc CLR. Để đảm bảo sự nhất quán, kiểm tra CanBeSaved nên xảy ra ở cấp dịch vụ miền trước khi kho được gọi để tồn tại tổng hợp và trong kho chỉ trước khi lưu. Nhà phát triển có trách nhiệm viết mã này. Mánh lới quảng cáo hợp đồng đảm bảo rằng, bất kể điều gì xảy ra, CLR sẽ kiểm tra liệu tổng thể có thể được lưu lại hay không.

Giao diện của các kho lưu trữ thường được xây dựng trong lắp ráp, nơi bạn mã lớp miền, nhưng việc thực hiện rõ ràng thuộc về lớp cơ sở hạ tầng.

Việc lắp ráp có chứa việc triển khai các kho chứa có sự phụ thuộc trực tiếp vào O / RM như Entity Framework, NoSQL, hoặc cơ sở dữ liệu trong bộ nhớ; Các dịch vụ kiên trì bên ngoài; Và bất cứ nhà cung cấp dữ liệu nào bạn cần.

Để giải quyết các nghi ngờ còn lại, chúng tôi muốn chỉ ra rằng một kho lưu trữ là nơi bạn đối phó với chuỗi kết nối và sử dụng lệnh SQL.

**Important** Cho đến nay, chúng tôi đã đề cập nhiều lần đến sự bền bỉ, nhưng chúng tôi không bao giờ tập trung vào việc lưu trữ vật lý. Chúng ta sẽ trở lại khía cạnh này một cách chi tiết trong Chương 14, "Lớp bền bỉ,"

Được dành riêng cho lớp bền bỉ. Bây giờ, nó sufices để nói rằng một mô hình miền tồn tại để được bền bỉ; Để đạt được sự bền bỉ thành công, tuy nhiên, bạn có thể cần phải thực hiện một số nhượng bộ trong mô hình tên miền của bạn và thực hiện các thỏa hiệp khác.

**Note** Kho lưu trữ là một thuật ngữ khá cồng kềnh. Nếu bạn nhìn xung quanh, bạn có thể một vài conlicting deinitions cho nó và một số đề nghị thực hiện mạnh mẽ. Chúng tôi nghĩ

Cùng một thuật ngữ được dùng để chỉ các khái niệm khác nhau. Trong bối cảnh của DDD, một kho lưu trữ là những gì chúng ta đã mô tả trong chương này - lớp xử lý sự tồn tại thay mặt các thực thể, và lý tưởng là tổng hợp các rễ. Chúng ta sẽ nói thêm về điều này trong chương 14.

**Đó có phải là một dịch vụ hay là một sự tổng hợp?**

Như đã đề cập, chúng ta có xu hướng xem các dịch vụ miền là giải pháp sao lưu cho các phần của hành vi mà chúng ta không thể tự nhiên được vì bất cứ lý do gì. Tuy nhiên đôi khi chúng ta cũng băn khoăn liệu hành động kinh doanh đó thực sự không có trong mô hình miền hay liệu chúng ta không thể mô hình đúng.

Khi ngôn ngữ phổ biến có vẻ như là một hoạt động như là một khái niệm về lớp học irst, ví dụ, chuyển nhượng hoặc đặt phòng, chúng tôi tự hỏi liệu chúng tôi sẽ không được tốt hơn bằng cách sử dụng một thực thể liên tục mới thay vì một dịch vụ miền để orchestrate quá trình.

Cuối cùng, tuy nhiên, cả hai lựa chọn có thể được thực hiện để làm việc.

**Domain events**

Hãy tưởng tượng trường hợp sau: trong một ứng dụng cửa hàng trực tuyến, một đơn hàng được đặt và xử lý thành công bởi hệ thống. Khoản thanh toán đã được thực hiện OK, lệnh chuyển phát đã được chuyển đến và nhận được bởi công ty vận chuyển và đơn hàng đã được tạo và chèn vào hệ thống. Giờ thì sao? Hệ thống sẽ làm gì khi một lệnh được tạo ra? Hãy nói rằng ngôn ngữ phổ biến cho bạn biết chính xác những gì để làm, và đó là một số công việc T. Bạn sẽ thực hiện nhiệm vụ T ở đâu?

**Từ logic tuần tự đến các sự kiện**

Tùy chọn irst chỉ là nối đoạn mã cho nhiệm vụ T với phương thức dịch vụ miền thực hiện việc xử lý đơn đặt hàng. Đây là một số giả mã:

void Checkout(ShoppingCart cart)

{

// Proceed through the necessary steps

...

if (success)

{

// Execute task T

...

}

}

Mã này có một số vấn đề. Ở nơi irst, nó không phải là rất biểu cảm. Tất cả các bước tạo thành một thủ tục thanh toán nguyên khối end-end. Bên ngoài phương thức Checkout, không có khả năng hiển thị gì vào nội dung có thể xảy ra bên trong. Đây thực sự là một vấn đề? Vâng, nó phụ thuộc. Trong trường hợp này, ví dụ, nó có thể là một dấu hiệu của một số vi phạm của ngôn ngữ phổ biến.

Một vài dòng trước đó, chúng tôi đã viết phó từ "khi" bằng chữ nghiêng. Chúng tôi đã làm như vậy để gọi sự chú ý của bạn đến từ chuyên biệt. Bất cứ lúc nào bạn ind "khi" trong yêu cầu, bạn có thể có sự xuất hiện của một sự kiện có liên quan trong tên miền. Một sự kiện đăng ký sự xuất hiện của một thực tế trong một số bối cảnh; Từ "khi" trong yêu cầu chỉ ra rằng có một số lợi ích kinh doanh để biết khi một thực tế nhất định xảy ra.

Yêu cầu chỉ ra phải làm gì khi một sự kiện nhất định được quan sát. Nếu bạn đặt mã xử lý vào phương thức dịch vụ miền, bạn phải chạm vào phương thức dịch vụ miền trong trường hợp yêu cầu thay đổi hoặc cần phải thực hiện nhiều hành động hơn cho cùng một sự kiện. Sửa đổi một lớp dịch vụ miền không phải là nguy hiểm cho mỗi gia đình, nhưng nói chung, bất kỳ phương pháp tiếp cận nào khác làm giảm thiểu nguy cơ sửa đổi các lớp học hiện có cũng được hoan nghênh.

**Raising an event when something relevant occurs**

Sẽ không có gì tốt hơn nếu chúng ta chỉ tổ chức một sự kiện khi một thực tế có liên quan đến miền nhất định xảy ra? Sự kiện sẽ loại bỏ sự cần thiết phải có tất cả các mã xử lý ở một nơi duy nhất. Điều này mang lại hai lợi ích chính cho bảng:

* + Cho phép chúng tôi tự động xác định danh sách các trình điều khiển mà không cần chạm vào mã tạo sự kiện.
  + Cho phép chúng tôi có khả năng có nhiều nơi cùng xảy ra sự kiện tương tự. Điều này có nghĩa là các trình điều khiển sẽ chạy bất kể người gọi thực.

Hãy xem làm thế nào để xác định một sự kiện trong một mô hình miền.

**Formalizing a domain event**

Vào cuối ngày, một sự kiện miền là một lớp đơn giản đơn giản đại diện cho một sự xuất hiện thú vị trong miền. Ví dụ điển hình là khi một lệnh được tạo ra hoặc khi một khách hàng đạt đến trạng thái vàng theo chương trình lòng trung thành. Nếu bạn dành chút thời gian suy nghĩ về các sự kiện, bạn sẽ có rất nhiều ví dụ trong hầu hết bất kỳ mô hình miền nào bạn xem xét. Một sự kiện là khi hóa đơn được phát hành hoặc sửa đổi, khi một khoản nợ tín dụng được phát hành cho một hóa đơn đi, khi một khách hàng mới được đăng ký trong hệ thống, vân vân.

Phù hợp với nguyên tắc này, một sự kiện miền có thể chỉ đơn giản là một thành viên sự kiện được giao trong một lớp học. Tuy nhiên, trong kịch bản Mô hình Miền, các sự kiện thường được biểu diễn thông qua một lớp đặc trưng miền được đánh dấu bằng một giao diện đặc biệt.

public interface IDomainEvent

{

}

Here’s a sample class:

public class CustomerReachedGoldMemberStatus : IDomainEvent

{

public Customer Customer { get; set; }

}

**Note** Trong .NET, một sự kiện là một thể hiện của một lớp xuất phát từ EventArgs và nó chứa thông tin mô tả hoàn cảnh. Cuối cùng, IDomainEvent và một lớp cơ sở như EventArgs là hai cách khác nhau để tạo cho lớp sự kiện một vai trò.

Sử dụng EventArgs như là lớp cơ sở, cơ chế .NET của các sự kiện nâng cao có thể deinitely là một lựa chọn. Tuy nhiên, thực tế phổ biến trong mô hình miền để thực hiện công cụ sự kiện nội bộ, đánh dấu sự kiện thông qua giao diện và các thay đổi trạng thái tín hiệu đáng kể trong các thực thể miền bằng cơ chế xuất bản / đăng ký bên trong, như trong mã sau:

public class OrderRequestService

{

public void RegisterOrder(ShoppingCart cart)

{

// Create and persist the new order

...

// Check gold status for the customer who made the order

CheckGoldStatusForCustomer(cart.Customer);

}

public void CheckGoldStatusForCustomer(Customer customer)

{

// Get total amount of orders placed in current year

var totalOrdered = CalculateTotalOrdersForCustomer(customer, DateTime.Today.Year)

if (totalOrdered > 1000)

{

Bus.Raise(new CustomerReachedGoldMemberStatus() { Customer = customer });

}

}

}

**Handling domain events**

Nâng cao sự kiện chỉ là một phần của công việc. Bước tiếp theo bao gồm việc tìm ra hướng xử lý cho các sự kiện miền đó. Dưới đây là một sự thực hiện cơ bản của lớp học làm tăng các sự kiện:

public class Bus

{

private static IList<IHandler<IDomainEvent>> Handlers = new List<IHandler<IDomainEvent>>();

public static void Register(IHandler<IDomainEvent> handler)

{

if (handler != null)

Handlers.Add(handler);

}

public static void Raise<T>(T eventData) where T : IDomainEvent

{

foreach (var handler in Handlers)

{

if (handler.CanHandle(eventData))

handler.Handle(eventData);

}

}

}

Như bạn thấy, việc nâng cao một sự kiện thực sự có nghĩa là đi qua danh sách những người đăng ký đã biết và cho từng cơ hội để xử lý sự kiện. Bất kỳ trình xử lý đã đăng ký nào cũng có cơ hội xử lý một sự kiện của một loại nhất định.

Trình xử lý là một lớp nhỏ có chứa một số logic để chạy trong phản ứng với một sự kiện nhất định. Rõ ràng, bạn có thể có nhiều lớp để xử lý cùng một sự kiện miền. Điều này cho phép cho các thành phần của hành động, chẳng hạn như khi một sự kiện xảy ra, bạn irst làm công việc 1 và sau đó công việc 2. Dưới đây là một số mẫu mã cho một handler:

public class GoldStatusHandler : IHandler<IDomainEvent>

{

public void Handle(IDomainEvent eventData)

{

// Some synchronous task

...

return;

}

public bool CanHandle(IDomainEvent eventType)

{

return eventType is CustomerReachedGoldMemberStatus;

}

}

Lưu ý rằng nếu phần thân của phương thức Handle cần chứa một số thao tác async (ví dụ như gửi email), thay vì sử dụng các sự kiện, bạn có thể sử dụng các tin nhắn đơn giản trên một số hàng đợi. Hoặc, nếu có thể đưa ra bối cảnh, bạn chỉ có thể đặt một yêu cầu ire-and-forget cho hoạt động đồng bộ đến một thành phần khác được gọi ra từ bên trong phương pháp Handle.

**Cross-cutting concerns**

Trong một kiến trúc theo lớp, lớp cơ sở hạ tầng là nơi mà bạn có bất cứ điều gì liên quan đến công nghệ bê tông. Lớp chủ yếu bao gồm sự tồn tại dữ liệu thông qua các công cụ O / RM nhưng không giới hạn ở đó. Lớp cơ sở hạ tầng cũng là nơi dành cho các mối quan tâm xuyên suốt và bao gồm API cụ thể cho bảo mật (ví dụ: OAuth2), khai thác gỗ, truy tìm cũng như các khuôn khổ IoC.

Hãy nhanh chóng xem mối quan tâm xuyên suốt và sự kiên trì ảnh hưởng đến mô hình miền.

**Under the umbrella of a single transaction.**

Không chỉ bởi vì bạn làm mô hình miền thông minh và mát mẻ, bạn không cần phải bẩn tay của bạn với chuỗi kết nối cơ sở dữ liệu và những điều ưa thích như quy tắc thác, chìa khoá nước ngoài và thủ tục được lưu trữ. Như đã đề cập, tính kiên trì không phải là mối quan tâm bạn phải để lộ ra thông qua giao diện công khai của miền. Các hoạt động CRUD mặc dù xảy ra và được mã hóa chủ yếu thông qua các kho lưu trữ và kết hợp thông qua các dịch vụ miền và ở cấp cao hơn thông qua lớp ứng dụng.

Ngày nay, hầu hết các khía cạnh nâng cao của một giao diện CRUD được xử lý đầy đủ thông qua O / RM. Khi bạn sử dụng Entity Framework, bạn sẽ thao tác giao dịch trên một đơn vị công việc mà đối tượng ngữ cảnh khung của bạn sẽ quản lý một cách minh bạch cho bạn. Đơn vị công việc chỉ kéo dài một kết nối cơ sở dữ liệu duy nhất. Đối với các giao dịch multiconnection, bạn có thể sử dụng lớp TransactionScope. Thông thường, bạn có thể cần phải sử dụng lớp này từ lớp ứng dụng để may cùng nhau nhiều bước của một thao tác hợp lý.

Cuối cùng, bạn cô lập mã truy cập dữ liệu trong các kho và sử dụng các kho từ bên trong lớp ứng dụng hoặc các dịch vụ tên miền.

**Validation.**

Trong DDD, có hai tình cảm chính kể cả khi xác nhận là có liên quan. Hầu hết mọi người nói rằng một thực thể phải luôn ở trong một trạng thái hợp lệ, có nghĩa là nhà nước phải liên tục phù hợp với các quy tắc kinh doanh đã biết. Một số người khác cho rằng các thực thể nên được xem như là các đối tượng NET cho đến khi chúng được sử dụng theo một cách nào đó. Chỉ tại thời điểm này - thời gian chạy - sự xác nhận đó diễn ra và những sự không chắc chắn sẽ bị ném ra nếu có điều gì sai.

Vì DDD là mô hình hóa miền một cách trung thành, nên luôn phải đối phó với các thực thể hợp lệ - ngoại trừ việc tôn trọng các quy tắc xác nhận tất cả thời gian có thể là vấn đề khá phức tạp. Quy tắc xác nhận, trên thực tế, giống như bất biến; Do đó, các thực thể phải có trách nhiệm làm biến đổi bất biến trong cuộc đời của chúng. Thật không may, các điều kiện bất biến cho một thực thể thường được tính đến mô hình miền (chứ không phải lớp) và thường yêu cầu kiểm tra thông tin ngoài tầm với của một thực thể nhất định (ví dụ, các giá trị duy nhất của cơ sở dữ liệu).

Đây chính là điều làm cho một số tình huống được mong muốn để thực hiện nhượng bộ và thừa nhận các thực thể trong một trạng thái không hợp lệ mà một số các biến bất biến bị bỏ qua.

Trong bất kỳ lớp .NET nào, logic xác nhận có thể đơn giản như kiểm soát if-then-throw thấp, hoặc nó có thể đi qua một khuôn khổ xác nhận như các chú thích dữ liệu .NET hoặc Khối Xác nhận Thư viện Doanh nghiệp. Mặc dù chúng tôi ghi chú dữ liệu đó là tuyệt vời trong giao diện người dùng và lớp trình bày, nhưng chúng tôi vẫn giữ mục tiêu giữ mô hình miền càng gọn gàng càng tốt. Điều này có nghĩa là chúng ta chủ yếu sử dụng các câu lệnh if-then-throw hoặc .NET Code Contracts-chủ yếu là các điều kiện tiên quyết và hậu điều kiện.

Việc sử dụng đơn giản các điều kiện tiên quyết, tuy nhiên, có thể không đảm bảo rằng đối tượng đang ở trong một trạng thái hợp lệ. Bởi vì các biến thể bất biến áp dụng ở cấp mô hình miền, đơn vị một mình không thể thực tế được kiểm tra tính nhất quán của nó mọi lúc. Đây là nơi có thể tạo dịch vụ tên miền quảng cáo để thực hiện xác nhận đầy đủ.

**Important** Các lớp học trong mô hình tên miền nên kiểm tra bất cứ điều gì họ có thể kiểm tra và ném các ngoại lệ đặc biệt bất cứ khi nào phát hiện sai. Chúng tôi không mong đợi mô hình tên miền để được tha thứ: chúng tôi mong đợi nó để hét lên ngay khi cái gì đó đi ngược với mong đợi. Còn những thông báo lỗi thì sao? Một mô hình miền nên có tập hợp ngoại lệ riêng và phơi bày chúng với các lớp trên như lớp ứng dụng. Lớp ứng dụng sẽ quyết định xem ngoại lệ miền nên được bong bóng lên như là đối với lớp trình diễn, lùi lại, hoặc nuốt phải. Thông báo lỗi thực tế, thay vào đó, là đến lớp trình bày.

**Security.**

Chúng ta phân biệt hai khía cạnh của an ninh. Chúng ta phải làm gì với môi trường xung quanh ứng dụng và chạm vào những thứ như quyền truy cập kiểm soát truy cập, cross-site scripting, và xác thực người dùng. Không có khía cạnh nào-phù hợp trong bối cảnh chung của ứng dụng áp dụng cho cấp lớp miền.

Các khía cạnh thứ hai của an ninh đã làm với ủy quyền và đảm bảo rằng chỉ có người dùng được ủy quyền thực sự được phép thực hiện các hoạt động nhất định. Như chúng ta đã thấy, lớp ứng dụng chỉ cho phép một cuộc gọi được ủy quyền vượt qua được logic tên miền.

Có thể là khó khăn đối với dịch vụ tên miền OrderHistory để kiểm tra từ bên trong lớp tên miền cho dù người gọi đã được cấp quyền đúng. An ninh tại cửa khẩu là phương pháp tiếp cận ưa thích của chúng tôi, và chúng tôi thấy nó được thực hiện bởi lớp trình bày (nói bằng cách giấu liên kết và các nút) cùng với lớp ứng dụng.

**Note** An ninh tại cổng trong lớp trình bày có thể chấp nhận được trong các tình huống trong đó tính chi tiết của bảo mật phù hợp với trường hợp sử dụng. Ví dụ điển hình là khi người dùng trong một vai trò nhất định không được phép thực hiện một hành động nhất định. Bạn có thể đạt được điều đó bằng cách ẩn các phần tử giao diện người dùng. Nói chung, an ninh tại cổng có nghĩa là hạn chế bảo mật được áp dụng càng sớm càng tốt trong ngăn xếp cuộc gọi. Thông thường, nó đã được áp dụng trong các dịch vụ ứng dụng vào đầu các trường hợp sử dụng.

**Logging**

Điều cơ bản về đăng nhập là bạn quyết định đăng nhập và từ đâu. Nếu bạn quyết định quyết định đăng nhập từ bên trong mô hình miền, bạn cần một tham chiếu đến cơ sở hạ tầng khai thác gỗ ngay trong mô hình miền.

Tuy nhiên, bạn không muốn tạo ra một sự kết hợp chặt chẽ giữa mô hình miền và thành phần logger. Những gì bạn có thể làm để đăng nhập từ mô hình miền là tiêm một giao diện logger trong các lớp mô hình nhưng giữ cho việc thực hiện các logger trong dự án cơ sở hạ tầng.

Điều này cho biết, chúng tôi tin rằng trong một thực thi .NET, khai thác gỗ nên xảy ra bất cứ khi nào có thể thông qua các trình dò tìm được tích hợp, chẳng hạn như lớp EventLogTraceListener. Các lớp này được xây dựng trong .NET Framework và không yêu cầu các phụ thuộc bổ sung, chẳng hạn như log4net và Logging Application Block trong Thư viện Doanh nghiệp.

**Caching**

Giống như khai thác gỗ, việc thực hiện lý tưởng của bộ nhớ đệm và vị trí của mã tương đối phụ thuộc vào những gì bạn thực sự muốn bộ nhớ cache. Không giống như khai thác gỗ, bộ nhớ cache không thực sự có vẻ là một mối quan tâm của mô hình miền. Tuy nhiên, các dịch vụ miền và các kho lưu trữ có thể tận dụng bộ nhớ đệm để cải thiện hiệu suất và tiết kiệm một số lần đọc cơ sở dữ liệu hoặc gọi các dịch vụ từ xa.

Từ quan điểm thiết kế, tất cả những gì bạn làm là ẩn các thành phần phụ thuộc vào việc sử dụng bộ nhớ cache sau hợp đồng. Ví dụ, bạn deine giao diện công cộng của kho và sau đó sử dụng cơ sở dữ liệu caching bất cứ điều gì bạn thích trong việc thực hiện các kho. Điều đó giữ hệ thống lưu trữ hoàn toàn minh bạch cho người tiêu dùng của kho.

Đối với các lý do thử nghiệm và mở rộng, bạn cũng có thể tóm tắt API bộ nhớ cache cho một giao diện và triển khai hợp lệ trong kho lưu trữ hoặc dịch vụ tên miền.

**Important** Bộ nhớ đệm có nghĩa là phục vụ người dùng một số dữ liệu cũ. Mặc dù chúng ta nhận ra rằng rất ít ứng dụng không thể chấp nhận dữ liệu chỉ trong vài giây trở nên lỗi thời, nhưng nó không phải là sự phát triển nhóm người có thể quyết định liệu phục vụ dữ liệu cũ là phù hợp. Phần lớn hoạt động như sau: chuyên gia về tên miền chia sẻ kỳ vọng về hiệu suất của mình và nhóm phát triển chỉ ra các chi phí liên quan để đạt được điều đó, cho dù đó là chi phí của phần cứng hoặc bộ nhớ đệm. Các

Inal word thuộc về các chuyên gia về tên miền.

**Summary**

Mô hình miền cố gắng mô hình hóa một lĩnh vực kinh doanh dưới dạng các thực thể và các đối tượng giá trị và mối quan hệ, trạng thái và hành vi của chúng. Trong một mô hình miền, một thực thể là một đối tượng thích hợp trong không gian kinh doanh; Nó có một bản sắc và một cuộc sống. Một đối tượng giá trị, thay vào đó, chỉ đơn giản là một điều vô tri trong không gian kinh doanh. Một số thực thể sau đó được nhóm lại dưới dạng tập hợp lại vì lợi ích của mã và thiết kế

Trong chương này, chúng tôi tập trung chủ yếu vào nền tảng của mô hình Domain Model và xem xét các khía cạnh lý thuyết của nó. Trong chương tiếp theo, chúng ta sẽ tiến sâu hơn vào các khía cạnh triển khai. Trong các chương sau, chúng ta thảo luận về các kiến trúc hỗ trợ khác, như Phân chia Trách nhiệm Trật tự Trắc nghiệm / Truy vấn (CQRS) và Tìm nguồn sự kiện.

**Finishing with a smile**

Theo chúng tôi biết, không có luật pháp của Murphy cho các nhà phát triển. Tuy nhiên, chúng tôi đã đưa ra các ứng cử viên thú vị trong khi tìm kiếm các câu chuyện cười về phát triển và phần mềm hài hước trên Internet, và chúng tôi cũng tự phát minh ra một số:

* Nếu một nhà phát triển có thể sử dụng một API sai cách, ông sẽ
* Dữ liệu thực tế sẽ hiếm khi phản ánh các khái niệm đã được định trước của bạn về nó. Bạn thường sẽ khám phá ra điều này vào thời điểm không thích hợp nhất.
* Những người có ít kinh nghiệm thực tế nhất thường có những ý kiến mạnh mẽ nhất về cách mọi việc nên làm.
* Giải pháp cho một vấn đề thay đổi vấn đề.

Chương 9: Thực hiện mô hình miền

Câu nói ở đầu chương, bản chất của chúng ta có thể tóm tắt là "chỉ cho tôi đoạn code" (hay viết tắt là SMTC), là một câu ngắn gọn trong câu trả lời ngắn mà Torvalds đưa ra trên một danh sách gửi thư Linux cách đây nhiều năm. Bản gốc có tại đây: https://lkml.org/lkml/2000/8/25/132. Kể từ đó, SMTC đã được sử dụng thường xuyên để phá vỡ các cuộc thảo luận có tiềm năng vô tận mà là hướng tới phân tích / tê liệt.

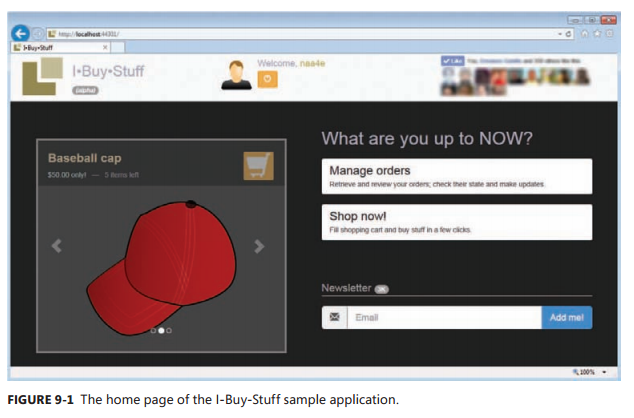
Mặc dù lý thuyết là vô giá, lý thuyết mà không có thực hành là rất ít sử dụng.

Chúng tôi đã thảo luận về thời gian về những gì sẽ là kịch bản tốt nhất để tập trung vào việc cho thấy tầm quan trọng của việc mô hình hóa lĩnh vực kinh doanh trong một kiến trúc lớp. Cuối cùng, chúng tôi đã đi cho một cửa hàng trực tuyến, bỏ qua nhiều kịch bản khác mà chúng tôi đã có cơ hội để làm việc trong quá khứ. Đặc biệt, Dino có những trải nghiệm thiết kế theo miền (DDD) thú vị trong bối cảnh hệ thống quản lý, đặt phòng đặt phòng, và các kịch bản về điểm số, trong khi đó Andrea đã hoạt động trong lĩnh vực ngân hàng, hoạch định tài nguyên doanh nghiệp (ERP), nhân sự (HR) , và xuất bản.

Cuối cùng, chúng tôi quyết định rằng một cửa hàng trực tuyến là một thứ dễ dàng nắm bắt cho mọi người, dễ dàng cho mô-đun hóa cho chúng tôi và dễ dàng xây dựng một con đường học tập hiệu quả. Vì vậy, trong chương này và các chương sau chúng ta sẽ xây dựng dự án "I-Buy-Stuff" và tạo ra ba nguyên mẫu khác nhau dựa trên các kiến trúc hỗ trợ khác nhau: mô hình miền, Command / Query Responsibility Segregation (CQRS) và sau đó là sự kiện tìm nguồn cung ứng. Và bây giờ, nếu không có chút náo nhiệt, hãy cho chúng tôi chỉ trình bày mã.

**The online store sample project**

Trong chương này, chúng ta sẽ sử dụng một mô hình miền làm kiến trúc hỗ trợ. Hình 9-1 trình bày trang chủ của ứng dụng mẫu đi kèm với chương này.



**Selected use-cases.**

Khách hàng gọi chúng tôi để giúp đỡ trong việc phát triển một cửa hàng trực tuyến. Họ muốn xây dựng một danh sách các khách hàng đã đăng ký trực tuyến và cho họ cơ hội để mua hàng, xem xét các đơn đặt hàng hiện có, nhận bản tin của trang web và tham gia chương trình khách hàng trung thành để được giảm giá.

Để đơn giản, nhưng không có tổn thất chung, chúng tôi đã xây dựng ứng dụng xung quanh các trường hợp sử dụng được liệt kê trong Bảng 9-1.

**Important** Ứng dụng mẫu chúng tôi trình bày là đơn giản, đủ để yêu cầu một ngữ cảnh bị ràng buộc. Tuy nhiên, kịch bản cửa hàng trực tuyến có thể phức tạp đến mức yêu cầu vì lợi ích của ứng dụng việc sử dụng một số bối cảnh bị giới hạn. Ví dụ, suy nghĩ về hệ thống Amazon - với quy mô và sự phức tạp của nó, không có cách nào nó có thể là một bối cảnh bị ràng buộc. Số lượng bối cảnh bị giới hạn phụ thuộc vào tính phức tạp của kịch bản, và sau đó là số lượng các trường hợp sử dụng, hiệu suất dự kiến và các yêu cầu chức năng và phi chức năng khác. Chúng ta sẽ quay trở lại điểm này trong phần sau của chương sau khi chúng ta giới thiệu dự án mẫu.

|  |  |
| --- | --- |
| **Use-case** | **Description** |
| Đăng ký vào trang web | Những lỗi của người dùng trong mẫu đơn và trở thành khách hàng chính thức của trang I-Buy-Stuff. |
| Đăng nhập để bắt đầu sử dụng trang web | Người dùng nhập thông tin đăng nhập và đăng nhập Thông tin xác thực có thể được nhập trực tiếp vào trang web hoặc thông qua một vài mạng xã hội. |
| Đăng ký bản tin | Người dùng thêm địa chỉ email để nhận bản tin. |
| Tìm đơn đặt hàng | Người dùng cho biết số lượng một trong những đơn đặt hàng của cô ấy và xem lại các chi tiết như ngày phân phối, mặt hàng và chi phí ước tính. |
| Đặt hàng | Người sử dụng gây ra một giỏ hàng với một danh sách các sản phẩm và sau đó tiến hành thanh toán và nhập chi tiết liên quan đến việc phân phối |

Tại irst, nó có thể không có vẻ như bất cứ điều gì ưa thích. Dường như chỉ là loại điều mà nhiều nhà phát triển làm hàng ngày, đã được làm trong nhiều ngày, hoặc có thể sẽ được làm ở một số điểm trong tương lai. Ứng dụng mẫu dường như thực sự gần với nhiều trình diễn mẫu bạn đã thấy xung quanh ASP.NET và Windows Presentation Foundation (WPF) qua nhiều năm. Đặc biệt, chúng tôi nhớ IbuySpy mẫu ứng dụng cổng thông tin được xây dựng cho ASP.NET trở lại trong những ngày đầu của Microsoft. NET Framework.

Thách thức cho thấy có rất nhiều việc phải làm trong lĩnh vực hiểu biết và làm cho lĩnh vực kinh doanh thậm chí trong một tình huống phổ biến luôn được xử lý thành công trong .NET với các phương pháp tiếp cận cơ sở dữ liệu.

**Selected approach.**

Một thủ thuật cũ nhưng vẫn hiệu quả để nhanh chóng hình thành một ý tưởng về một cuốn sách, hoặc thậm chí một ứng cử viên trong khi tiến hành một cuộc phỏng vấn việc làm, là kiểm tra những gì có thể phải nói về một chủ đề bạn biết rất rõ.

Một cửa hàng trực tuyến là điều mà mọi người hiểu được, và gần như mọi nhà phát triển có thể có một ý tưởng rõ ràng về cách tấn công nó. Trong chương này, chúng ta sẽ xây dựng nó bằng cách sử dụng phương pháp tiếp cận Mô hình Miền như được mô tả trong chương trước. Chúng ta sẽ định ra mô hình đối tượng trung thành đại diện cho lĩnh vực kinh doanh và gán cho mỗi thực thể một hành vi và logic miền riêng biệt từ các dịch vụ miền.

Sau khi phân tích các yêu cầu và xây dựng ngôn ngữ phổ biến, bạn nên có một ý tưởng rõ ràng về các thực thể bạn sẽ có. Được xây dựng như một dự án thư viện riêng biệt, mô hình miền chứa các thực thể sau: Quản trị viên, Khách hàng, Đặt hàng, OrderItem, Sản phẩm, Người đăng ký và FidelityCard. Bảng 9-2 tóm tắt nội dung mong đợi và hành vi của mỗi thực thể liên quan mà chúng tôi đã nhận dạng trong miền.

|  |  |
| --- | --- |
| **Entity** | **Description** |
| Admin | Đại diện cho một quản trị viên của trang web. Quản trị viên là người giữ tài khoản người dùng, người được quyền truy cập vào tất cả các tính năng của hệ thống. Cụ thể, Quản trị viên có thể cho phép giảm giá trên sản phẩm, định cấu hình tần suất của bản tin, tạo thống kê doanh thu, thêm thẻ tín dụng mới và các đối tác vận chuyển và hơn thế nữa |
| Customer | Đại diện cho người dùng của trang web. Trong thực thể này, bạn ind cả khía cạnh của một tài khoản người dùng cổ điển và một khách hàng cổ điển. Để trở thành một khách hàng, bạn cần có một ID và mật khẩu, và bạn cần có thẻ tín dụng và địa chỉ đã đăng ký. Một khách hàng được phân biệt theo giới tính và có thể có thẻ khách hàng và sau đó là mức thưởng (ví dụ: bạc, vàng, bạch kim). Đọc ghi chú tiếp theo để biết thêm chi tiết về khách hàng. |
| Order | Đại diện cho một đơn đặt hàng sau khi nó được tạo ra. Một đơn đặt hàng được tạo từ nội dung của giỏ hàng và bao gồm một danh sách các mặt hàng. Đơn đặt hàng có trạng thái thay đổi theo thời gian do hàng hoá đã đặt hàng được đóng gói và vận chuyển. Một lệnh biết tổng giá trị của nó. Thứ tự có ID của nó chỉ khi nó được tạo ra trong hệ thống. |
| Oderitem | Đại diện cho một dòng theo thứ tự và được làm bằng một lượng và một sản phẩm. Một bộ sưu tập các mặt hàng hình thành nội dung của giỏ hàng và được lushed vào một đơn đặt hàng. |
| Product | Đại diện cho một sản phẩm được bán thông qua trang web. Nó bao gồm các thuộc tính mô tả cộng với mức độ cổ phiếu hiện tại và độ trễ làm cho nó trở thành một sản phẩm đặc trưng, trong đó người mua kiếm được ưu đãi đặc biệt và giảm giá thêm. Các sản phẩm nổi bật được hiển thị trong băng chuyền trên trang chủ. Việc lựa chọn các sản phẩm đặc trưng được định hướng bởi các quy tắc kinh doanh. |
| Subscriber | Biểu thị một tổng hợp đơn giản chỉ bao gồm địa chỉ email mà bản tin sẽ được gửi tới. |
| FidelityCard | Biểu thị thẻ khách hàng trung thành do trang web đó phát hành. Nó có một số và tích lũy điểm và liên kết với khách hàng. Dựa trên các điểm, quy tắc kinh doanh xác định mức thưởng của người sử dụng (vàng, bạc, vân vân) Các điểm trên thẻ được cập nhật mỗi lần đặt hàng. Số tiền tích lũy được khi mua hàng được xác định theo các quy tắc kinh doanh. |

**Note** Trong ví dụ của chúng tôi, bất kỳ khách hàng nào của cửa hàng là một khách hàng đã đăng ký với một bản ghi trong hệ thống. Trong một kịch bản thực tế hơn, một chuyên gia về tên miền sẽ cho bạn xem xét

"Khách hàng" chỉ là người dùng thực sự đã mua một mặt hàng từ trang web ít nhất một lần. Thông thường, người dùng được phép ám ảnh giỏ hàng ngay cả khi ẩn danh, nhưng họ phải đăng ký và trở thành khách hàng-khi họ kiểm tra và thanh toán

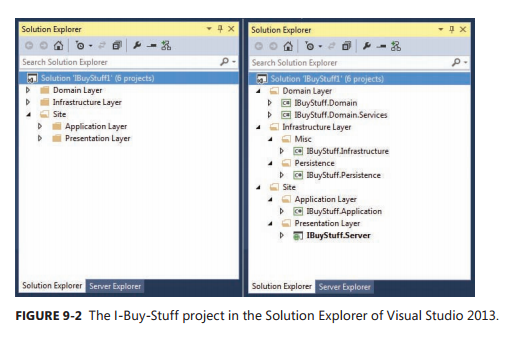
**Structure of the I-Buy-Stuff project**

Trong các chương trước, chúng tôi tập trung vào kiến trúc lớp và vai trò của nó trong DDD. Bây giờ là lúc để xem nó đang hoạt động. Và nó bắt đầu từ cấu trúc của giải pháp bạn xây dựng trong Microsoft Visual Studio 2013.

Hình 9-2 cung cấp hai lần xem của Solution Explorer.

Chế độ xem irst chỉ hiển thị các thư mục giải pháp.Như bạn thấy, có một lớp tên miền, lớp cơ sở hạ tầng và trang web. Ứng dụng I-Buy-Stuff, in

Thực tế, là một dự án trang web ASP.NET MVC trong thư mục Site.



Từ khung nhìn phía bên trái trong Hình 9-2, bạn cũng có thể nhận thấy sự tách biệt gọn gàng (lôgíc) giữa đầu cuối và đầu trước. Các lớp trình bày và ứng dụng là một phần của đầu cuối của hệ thống; Các miền và các lớp cơ sở hạ tầng là một phần của kết thúc trở lại. Phần cuối phía trước được sử dụng bởi nhiều giao diện người dùng khi bạn có trong cùng một ngữ cảnh bị chặn, cho dù giao diện người dùng chỉ đơn giản là một giao diện người dùng khác (web, WPF, web di động) hoặc chỉ là một ứng dụng riêng biệt với trường hợp sử dụng riêng của nó (Ví dụ: ứng dụng dành cho thiết bị di động hoặc trình khách WPF)

Chế độ xem hiển thị bên phải trong Hình 9-2 hiển thị các dự án đang được tạo ra. Ngoại trừ IBuyStuff.Server, là một dự án ASP.NET MVC 5, tất cả các dự án khác đều là các thư viện lớp NET.

Hình 9-2 trình bày các dự án và thư mục riêng biệt cho giao diện người dùng trực tuyến và lớp ứng dụng. Từ quan điểm khái niệm, tất cả đều tốt và thậm chí rất rõ ràng. Tuy nhiên, tách các lớp logic như vậy trong các dự án riêng biệt có thể tạo ra một số nếp nhăn trong việc quản lý hàng ngày của giải pháp. Ví dụ: hãy tưởng tượng bạn muốn thêm khung nhìn Razor mới vào dự án. Các CSHTML bao nhiêu đi trong dự án web, trong khi lớp mô hình xem thuộc về dự án ứng dụng. Cùng một điều xảy ra khi bạn thêm một

Bộ điều khiển mới-lớp điều khiển đi vào dự án web, trong khi lớp dịch vụ ứng dụng đi đến dự án ứng dụng. Điều này làm cho giàn giáo phức tạp hơn một chút so với lẽ phải. Cuối cùng, mặc dù sự tách biệt các mối quan tâm (SoC) là nền tảng vô tận của phần mềm, cách thức mà nó có thể được điều chỉnh để phục vụ các mục đích thực tiễn. Chúng tôi thường có toàn bộ nội dung của dự án ứng dụng mà bạn thấy trong ví dụ : dưới một thư mục Application bên trong dự án web. Và chúng tôi cũng làm như vậy nếu dự án là một ứng dụng Microsoft Windows hoặc Windows Phone.

**Selected technologies**

Các dự án sử dụng một loạt các công nghệ .NET, mà phần lớn đều được kết hợp thông qua các gói NuGet. Như đã đề cập, giao diện người dùng là một ứng dụng ASP.NET MVC 5 sử dụng ASP.NET Identity làm khuôn khổ cho việc xác thực người dùng. Bootstrap Twitter được sử dụng cho giao diện người dùng. Đối với chế độ xem trên thiết bị di động, chúng tôi sử dụng Ratchet2 cho giao diện người dùng và WURFL để thực hiện thiết bị phát hiện và chế độ hiển thị ASP.NET để xem định tuyến.

**Important** Ứng dụng I-Buy-Stuff là một loại codebase ứng dụng web phổ quát hỗ trợ nhiều chế độ xem cụ thể thiết bị. Không giống như cách sử dụng các truy vấn media theo kiểu kiểu xếp tầng (CSS) để điều chỉnh một mẩu nội dung cho nhiều thiết bị. Nó chỉ là về phục vụ nội dung web khác nhau được tạo ra từ một cơ sở mã duy nhất. Trang web vẫn đáp ứng nhưng thông minh hơn nhiều. Chúng ta sẽ trở lại điểm này trong chương nội dung của cuốn sách, nói về cơ sở hạ tầng.

Phần cuối của I-Buy-Stuff dựa trên Bộ quy tắc Thực thể Đầu tiên cho sự bền bỉ. Mã mẫu bạn sử dụng một trường hợp cục bộ của Microsoft SQL Server để tạo bảng trong thư mục App\_Data của trang web. Cuối cùng, tất cả các nhu cầu tiêm phụ thuộc vào giải pháp được quản lý thông qua Microsoft Patterns và Practices Unity.

**Note** Chúng tôi đã từng viết các nhà máy điều khiển riêng của chúng tôi trong ASP.NET MVC để cắm vào Unity hoặc các loại container Inversion of Control (IoC) khác. Ngày nay, đối với Unity liên quan, chúng tôi cắm nó qua gói Unity.Mvc NuGet.

**Bounded contexts for an online store**

Cửa hàng trực tuyến I-Buy-Stuff là một cách đơn giản và đơn giản, và các trường hợp kinh doanh mà nó phải đối mặt không đòi hỏi nhiều hơn là chỉ là một ngữ cảnh bị ràng buộc. Để cung cấp cho bạn một sự hiểu biết sâu rộng

Của DDD, nó có lẽ không phải là một ví dụ lý tưởng. Mặt khác, chúng tôi nghĩ rằng một ví dụ phức tạp hơn sẽ quá áp đảo.

Tuy nhiên, chúng tôi muốn dành một chút thời gian để thảo luận về kịch bản cửa hàng trực tuyến theo chiều sâu hơn vì nó có thể xuất hiện trong ngữ cảnh thực.

**Decomposing the business domain in contexts**

Trong một kịch bản đơn giản hợp lý, bạn có thể có ba bối cảnh bị giới hạn riêng biệt:

* + MembershipBC
  + OrderingBC
  + CatalogBC

**Thậm chí thêm nhiều giới hạn ngữ cảnh hơn**

Nếu sự phức tạp của cửa hàng tăng lên, bạn có thể cân nhắc thêm các ngữ cảnh bị giới hạn. Dưới đây là một số ví dụ về ngữ cảnh bạn có thể thêm:

* **ShippingBC:** Ngữ cảnh này có liên quan đến các miền kiểu mẫu để xử lý bất kì luận lý phức tạp nào. Ví dụ như I-Buy-Stuff của chúng tôi, được xem là một dịch vụ bên ngoài. Chúng ta sẽ không đi sâu vào những chi tiết của sự tương tác, nhưng chúng ta có thể cần một lớp đảm bảo rằng OrderingBC luôn luôn nhận data cùng một cách.
* **PricingBC:** Ngữ cảnh này trở nên cần thiết khi có giá trị bằng một hệ thống phức tạp dựa trên sự liên kết của người dùng và sản phẩm, chiết khấu cá nhân, kế hoạch thành thật, điểm thưởng,…
* **PurchaseBC:** Ngữ cảnh này liên quan đến miền hình mẫu cần thiết để quản lý hệ thống mua hàng được liên kết nhiều cổng thanh toán và có khả năng hỗ trợ các hình thức thanh toán khác, chẳng hạn như chuyển khoản ngân hàng và giao hàng tận nơi. Trong ví dụ I-Buy-Stuff của chúng tôi, đây được xem là một dịch vụ bên ngoài. Code mẫu giả định một cổng thanh toán duy nhất và cung cấp một mô hình giả cho việc thanh toán dưới dạng trang ASP.NET Web Forms.

Tóm lại, đối với các hệ thống tương đối đơn giản bạn có vài sự lựa chọn:

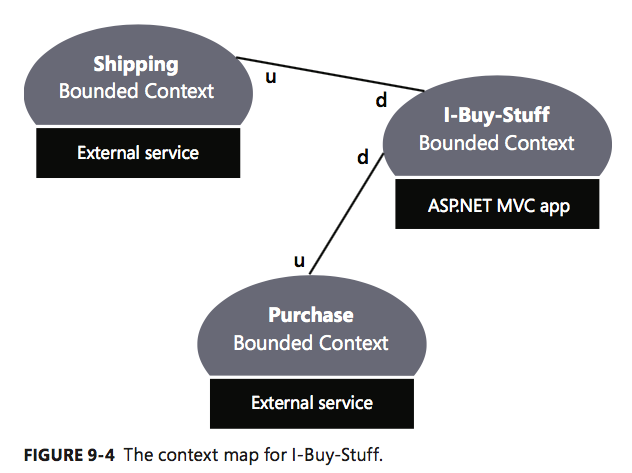
* Sử dụng một giới hạn ngữ cảnh lớn.
* Tách thành từng miền nhỏ, với dữ liệu trùng lặp, và có thể tương tác thông qua một thế thống publish/subscribe.
* Cô lập một vài giới hạn ngữ cảnh lớn hơn và phải xác thực chúng thông qua các dịch vụ web.

Không có quy tắc chung nào để chọn các sự lựa chọn ở trên. Tất cả chúng phụ thuộc và ngữ cảnh, kĩ năng và kinh nghiệm của bạn. Và, ở một mức độ tốt, nó cũng phụ thuộc vào tầm nhìn và sự hiểu biết về tên miền mà bạn đã đạt được. Trong một số trường hợp, ví dụ, nó có thể nhận ra rằng nhu cầu truy vấn có thể rất khác so với nhu cầu viết. Sau đó, nó có thể được ưu tiên lựa chọn hơn chỉ vì việc thiết kế các ngữ cảnh giới hạn riêng biệt cho việc truy vấn. đây là bản chất của CQRS, chúng ta sẽ thảo luận trong chương trình tiếp theo.

**Bản đồ ngữ cảnh của ứng dụng I-Buy-Stuff**

Hình 9-4 sẽ cho thấy bản đồ ngữ cảnh mà chúng ta sẽ có cho I-Buy-Stuff.

Như bạn thấy trên hình, I-Buy-Stuff là một ứng dụng ASP.NET MVC và sẽ thực hiện bằng kiến trúc Domain Model. Nó trở thành một bối cảnh rộng lớn hơn bởi vì nó có sự kết hợp các thành viên, thứ tự logic, và thậm chí là một phân đoạn rất đơn giản của ngữ cảnh mua hàng, chỉ là một giải thưởng vàng cho khách hàng.



Phần lớn logic trong kinh doanh nằm trong trang I-Buy-Stuff. Ngữ cảnh giới hạn việc Chuyển hàng và mua hàng đã được thực hiện(giả lập thực sự) như các dịch vụ bên ngoài. Cả hai dịch vụ bên ngoài cuối cùng là nguồn dẫn đến những trang web chính. Ít nhất trong bản demo, vì sự đơn giản, chúng ta không sử dụng một lớp chống hao mòn. Nếu chúng ta sử dụng nó, việc thực hiện có thể bao gồm một lớp phủ nhận thông tin thô từ các dịch vụ bên ngoài.

**Nghệ thuật vĩ đại của mô hình miền thực tế**

Về cơ bản chúng ta thấy trong phần mềm hiện đại là các nhà phát triển tập trung vào code hơn là mô hình. Cách tiếp cận này không phải là sai, nhưng nó rất khó để quản lý một cách phức tạp khi bạn phải đối mặt với nó.

Nếu bạn vui vẻ bỏ qua miền mô hình và bạn vẫn code thành công, liệu bạn có làm gì sai? Không hẳn thế, nhưng có thể bạn sẽ gặp rắc rối khi độ phức tạp lớn dẫn lên. Nó có thể là dự án tiếp theo hoặc cũng dự án này nhưng yêu cầu cứ liên tục đổi và lớn dần lên.

**Hành vi là một trò chơi thay đổi**

Một miền mô hình là 1 tập hợp các class với các thuộc tính và hàm. Cùng với nhau chúng hiển thị lên mô hình nghiệp vụ. Tuy nhiên, trong nhiều năm qua không ai thực sự nhìn vào cách để đào một môt hình như vậy ra khỏi bóng tối. Trong chương cuối, chúng tôi vạch ra những ý chính của một miền mô hình và tạo ra bộ sườn cho các class, cơ bản đó là lý thuyết. Khi đi đến việc thực hiện, có một số khí cạnh cần phải xem xét:

* Hiểu rõ và thực hiện những hành vi trở nên rất cần thiết cho sự thành công cuối cùng.
* Bạn cần phải kiên trì với mô hình, bạn sẽ sẵn sàng chấp nhận cho những sự thoả hiệp.
* Mô hình bạn làm việc là API cho lĩnh vực kinh doanh; nó phải được sắp xếp và kết nối được với bất kì người dùng nào.

Nghe có vẻ như khó, chúng tôi nghĩ rằng mọi người sẽ sẵn sàng mở rộng đường viền của “bức tranh” mà anh ta nhìn thấy sẽ cố gắng sử dụng miền mô hình trên một miền mà anh ta biết – ít nhất là một bài tập code kata.

Đây là linh hồn của I-Buy-Stuff

**Chú ý**: Giống như thế giới karate, một code kata là một bài tập làm mới các kiến thức cơ bản về kỹ năng của bạn. Trong khả năng phát triển phần mềm, nó có thể bao gồm viết lại một thuật toán được biết đến với một tâm trí tươi mới để xem liệu bạn có thể thực hiện cải tiến hay không. Để biết thêm chi tiết về tinh thần của code kata, xem ở đây http://codekata.com.

**Phối cảnh điển hình của một nhà phát triển .NET**

Hầu hết các nhà phát triển .NET đã phát triển theo hướng dẫn của mẫu thiết kế Bảng Mô-đun. Mô hình Bảng mô-đun cho thấy bạn tạo một mô hình dữ liệu quan hệ là lĩnh vực nghiệp vụ của nó. Tiếp theo, bạn xác định các bảng chính trong cơ sở dữ liệu và xây dựng một mô-đun cho mỗi bảng. Cuối cùng, một mô-đun bảng kết thúc chỉ là kho lưu trữ các phương thức thực hiện truy vấn và hành động lệnh đối với bảng (nói, Đơn hàng) và tất cả các bảng liên quan của nó (nói, OrderItems).

Các phương thức của lớp mô-đun bảng được gọi trực tiếp từ lớp hiển thị hoặc lớp ứng dụng, nếu bạn thực sự muốn có một trong số chúng. Hầu hết thời gian, mô hình back-end bạn nhận được có các khía cạnh sau:

* Kiến trúc hai lớp, với lớp trình bày gọi trực tiếp vào các kho lưu trữ hoặc các môđun bảng.
* Mô-đun bảng là các vùng chứa các hàm publuc không có trạng thái và tính biểu hiện giới hạn ở tên và danh sách tham số. Mở rộng và duy trì các lớp này có thể sớm trở thành một vấn đề. Cấu trúc của chúng rõ ràng là rõ ràng ngay từ đầu nhưng nhanh chóng trở nên không rõ ràng.
* Không rõ ràng các mô hình được sử dụng trong code. Bạn nên đi với các đối tượng DataSet? Bạn nên tự tạo các đối tượng truyền dữ liệu của riêng mình? Bạn có nên tận dụng Entity Framework và chỉ suy luận các lớp học mà phản ánh một-một-một cấu trúc của bảng?

Sự ra đời của Entity Framework khuyến khích sử dụng của các nhà thiết kế để suy luận cấu trúc cơ sở dữ liệu chỉ với một vài cú nhấp chuột bạn có thể tạo ra một mô hình đối tượng phản ánh cơ sở dữ liệu. Nó dễ và hiệu quả, và nó đã giải quyết được điểm cuối cùng trong việ có trước danh sách.

**Doc, mô hình của tôi bị thiếu máu**

Các chuyên gia DDD chỉ ra rằng không có mô hình nào cần phải bị thiếu máu. Vì thiếu máu phần mềm thường được đo lường bằng bao nhiêu hành vi bạn có trong các lớp học, trong một mô hình chỉ suy ra từ cơ sở dữ liệu bạn gần như không có hành vi. Vì vậy, mô hình của bạn là thiếu máu.

Bệnh thiếu máu phần mềm là bệnh nghiêm trọng?

Ở người, thiếu máu dẫn đến sự mệt mỏi và do thiếu các tế bào hồng cầu trong máu. Các tế bào hồng cầu có nhiệm vụ cung cấp oxy đến các mô cơ thể và mang lại sự sống và năng lượng. Tương tự, trong thiếu máu phần mềm biểu thị sự thiếu năng lượng để đối mặt với sự thay đổi, nhưng chúng tôi không gọi đó là bệnh. Một mô hình thiếu máu vẫn có thể hoạt động hiệu quả miễn là nhóm có khả năng quản lý sự phức tạp của miền và sự tiến hóa của nó.

Cuối cùng, một mô hình thiếu máu sẽ để lại thanh gươm Damocles treo trên code của bạn. Theo một câu chuyện kể về Hy Lạp cổ đại, Damocles ngồi trên ngai vàng với một thanh gươm khổng lồ treo lơ lửng trên người ông chỉ cầm bằng một sợi tóc duy nhất. Đây là phép ẩn dụ: tất cả đều tốt nếu thanh gươm vẫn tiếp tục treo.

Tập trung vào hành vi có hai mục đích chính: tạo ra các đối tượng có Interface Public gần các thực thể có thể quan sát được trong thế giới thực. Điều này cũng làm cho việc thực hiện mô hình hóa dễ dàng hơn theo tên và quy tắc của ngôn ngữ phổ thông.

**Vị trí của logic nghiệp vụ**

Từ quan điểm thực dụng hơn, tập trung vào hành vi chủ yếu là tìm một vị trí khác cho

Các phân đoạn của logic nghiệp vụ. Hãy xem xét lớp Hoá đơn:

class Invoice

{

public string Number {get; private set;}

public DateTime Date {get; private set;}

public Customer Customer {get; private set;}

public string PayableOrder {get; private set;}

public ICollection<InvoiceLine> Lines {get; private set;}

...

}

class không có hàm. Có sai không khi viết vậy? Là một phần của mô hình thiếu máu ?

Một mô hình đối tượng không phải là thiếu máu nếu nó chỉ thiếu phương pháp trên lớp. Một thực thể bị thiếu máu nếu có logic thuộc về nó được đặt bên ngoài lớp thực thể. Ví dụ: hãy xem xét vấn đề tính ngày ước tính thanh toán của một hoá đơn. Nhập dữ liệu bao gồm ngày của hóa đơn và các điều khoản thanh toán (tức thời, 30 ngày, 60 ngày hoặc bất kỳ điều gì).

Nếu bạn xác định phương pháp trên lớp Hoá đơn, nó sẽ kết thúc bằng câu lệnh sau đây:

DateTime EstimatedPayment()

Nếu không, nếu bạn muốn duy trì tất cả logic nghiệp vụ trong một lớp InvoiceModule đơn lẻ, nó có thể giống như thế này

DateTime EstimatedPayment(Invoice invoice)

Sau này nhận được thực thể để được điều tra như là lập luận duy nhất của nó. Đây là một cảnh báo có thể xảy ra rằng một phần logic thuộc về thực thể. cái "đúng" cho phương pháp này trong một thiết kế Mô hình Miền là ở đâu? Một quy tắc đơn giản nhưng hiệu quả để cơ giới hóa quyết định có thể được tóm tắt như sau:

* Nếu code trong phương thức chỉ xử lý với các thành viên của thực thể, nó có thể thuộc về thực thể.
* Nếu code cần truy cập các thực thể khác hoặc các đối tượng giá trị trong cùng một tổng hợp, nó có thể thuộc về tổng hợp gốc.
* Nếu code trong phương pháp cần truy vấn hoặc cập nhật lớp persistence hoặc, nói chung, cần có các tham chiếu bên ngoài ranh giới của thực thể (hoặc tổng hợp của nó), đó là một phương thức dịch vụ miền.

Nếu bạn bỏ qua hai quy tắc này và tập trung tất cả mọi thứ liên quan đến Hoá đơn trong một lớp dịch vụ duy nhất có kiểm tra tất cả mọi thứ, truy cập dữ liệu, logic kinh doanh thì bạn đang làm cho thực thể bị thiếu máu.

**Khán đài thực thể**

Một thực thể DDD là một lớp C# bằng phẳng và dự kiến sẽ chứa cả dữ liệu (thuộc tính) và hành vi (các hàm). Một thực thể có thể có các thuộc tính công khai, nhưng nó không phải là một vùng chứa dữ liệu đơn giản. Đây là các dấu hiệu của các thực thể DDD:

* được xác định rõ ràng
* Hành vi thể hiện qua các hàm, cả public và không
* Trạng thái được hiển thị thông qua thuộc tính chỉ đọc
* hạn chế sử dụng các loại dữ liệu thô, được thay thế bởi các đối tượng giá trị
* Một ưu tiên cho hàm fatory trên nhiều constructor

**Quản lý danh tính**

Đặc điểm chính của một thực thể là khả năng nhận diện. Nhận dạng là bất kỳ sự kết hợp của dữ liệu nhận dạng duy nhất một thể hiện của lớp thực thể. Theo cách nào đó, nhận dạng trong ngữ cảnh này có thể so sánh với một khoá chính cho các bản ghi của một bảng quan hệ.

Một danh tính phục vụ mục đích nhận dạng duy nhất một thực thể trong suốt thời gian sử dụng. Nhà phát triển sử dụng danh tính để lấy tham chiếu đến thực thể bất kỳ lúc nào. Trạng thái của một thực thể có thể thay đổi, nhưng danh tính thì không. Về mặt thực hiện, danh tính thường chỉ là một thuộc tính số nguyên như thuộc tính Id. Trong các trường hợp khác, nó có thể là một chuỗi số chẳng hạn như số An Sinh Xã Hội, ID VAT, tên người dùng hoặc kết hợp các thuộc tính.

Nhận dạng chỉ cần xác định điều gì làm cho hai thực thể cùng một thực thể. Một lớp thực thể, tuy nhiên, cũng cần thực hiện bình đẳng sử dụng các giá trị nhận dạng để xác định xem hai thực thể có giống nhau. Chúng ta hãy bắt đầu với một lớp Order như thể hiện ở đây:

public class Order

{

private readonly int \_id;

public Order(int id)

{

// Validate ID here

...

\_id = id;

}

public int Id

{

get {return \_id;}

}

...

}

Giả sử bây giờ bạn có hai trường hợp của lớp Order có tham chiếu đến cùng một ID:

var o1 = new Order(1);

var o2 = new Order(1);

Theo các hành vi cơ bản của một lớp. NET Framework, o1 và o2 là những trường hợp khác nhau. Mặc dù điều này là chính xác trong miền chung của .Net Framework, nhưng nó không phù hợp trong lĩnh vực nghiệp vụ bạn đang ở Trong lĩnh vực kinh doanh, những vấn đề ID của thứ tự: nếu ID phù hợp, hai đơn đặt hàng là như nhau . Tuy nhiên, những gì bạn nhận được ra khỏi hộp trong . NET và các khuôn khổ khác.

Để thực hiện một logic bình đẳng tùy chỉnh sử dụng các giá trị nhận dạng, bạn phải ghi đè bằng các phương thức Equals và GetHashCode rằng bất kỳ đối tượng .NET nào sẽ phơi bày. Đây là một ví dụ:

public class Order

{

private readonly int \_id;

...

public override bool Equals(object obj)

{

if (this == obj)

return true;

if (obj == null || GetType() != obj.GetType())

return false;

var other = (Order) obj;

// Your identity logic goes here.

// You may refactor this code to the method of an entity interface

return \_id == other.Id;

}

public override int GetHashCode()

{

return Id.GetHashCode();

}

}

**Quan trọng**: sự nhân dạng và sự bình đẳng là hai khái niệm riêng biệt, nhưng chúng có liên quan trong ngữ cảnh của một thực thể. Identity chỉ là một bộ sưu tập các giá trị làm cho một thực thể duy nhất (giống như một khoá chính). Bình đẳng là cơ sở hạ tầng lớp hỗ trợ các toán tử bình đẳng trong .NET Framework. Trong một thực thể, sự bình đẳng được kiểm tra bằng cách sử dụng các giá trị nhận dạng. Như bạn sẽ thấy trong chương sau, các đối tượng giá trị không có nhận dạng và không cần phải được truy lục sau để kiểm tra trạng thái của chúng, nhưng họ vẫn cần một số logic tùy chỉnh cho sự bình đẳng.

**Private Setters**

Một thực thể có cả hành vi (hàm) và trạng thái(thuộc tính). Tuy nhiên, trong thế giới thực, có thể các thuộc tính của một thực thể bị hạn chế chỉ là một vài giá trị. Một số sản phẩm, ví dụ, không phải là số nguyên vì nó không phải là số âm. Đồng thời, sử dụng các đối tượng unsigned cũng đáng tranh cãi, bởi vì nó có thể có nghĩa là bạn được phép đặt đơn đặt hàng cho hơn bốn tỷ mục.

Khi thiết kế các thực thể, bạn chỉ nên cung cấp các thuộc tính thực sự thể hiện trạng thái của thực thể và chỉ sử dụng các phương thức setter cho các thuộc tính có thể thay đổi. Ngoài ra, không để cho setters được mở cho bất kỳ giá trị mà các loại khai báo của thuộc tính. Người thiết lập mở có thể có các phản ứng phụ khó chịu, chẳng hạn như để các thực thể ở trạng thái tạm thời không hợp lệ.

Nói chung, cách làm việc tốt là để các setter là private và để lộ các hàm public để cho các nhà phát triển khác thay đổi trạng thái của thực thể. Tuy nhiên, khi bạn làm như vậy, chú ý đến quy ước đặt tên bạn sử dụng và đảm bảo nó phù hợp với ngôn ngữ phổ biến.

**Khuôn mẫu và hàm tạo**

Hàm tạo là cách thông thường để tạo các thể hiện của các lớp và có thể không cần các tham số. Trong mô hình miền, bạn nên tránh các lớp với các hàm khởi tạo không tham số. Ngoài ra, mỗi nhà xây dựng có thể trả về một thể hiện có trạng thái phù hợp với lĩnh vực kinh doanh. Đoạn mã sau đây có lẽ không phù hợp:

var request = new OrderRequest();

Yêu cầu đặt hàng mới được tạo ra không có tham chiếu đến ngày và khách hàng. Ngày có thể được đặt mặc định trong phần thân của hàm tạo, nhưng không phải là ID khách hàng. Sau đây là một chút tốt hơn:

var request = new OrderRequest(1234);

Có hai vấn đề ở đây. Trước tiên, khi nhìn vào code, người ta khó có thể đoán được điều gì đang xảy ra. Một thể hiện của OrderRequest đang được tạo ra, nhưng tại sao và sử dụng dữ liệu nào?ì 1234 là gì? Điều này dẫn đến vấn đề thứ hai: bạn đang vi phạm ngôn ngữ phổ thông trong ngữ cảnh bị ràng buộc. Ngôn ngữ có thể nói như sau: khách hàng có thể ra yêu cầu đặt hàng và được phép chỉ định ID mua hàng. Nếu đó là một trường hợp, đây là một cách tốt hơn để có được một trường hợp OrderRequest mới:

var request = OrderRequest.CreateForCustomer(1234);

hàm tạo là các hàm không tên và sử dụng chúng cũng có thể gây ra vấn đề đọc hiểu. Nhưng đó sẽ là những lo lắng ít nhất của bạn. Bằng cách sử dụng các hàm tạo public, bạn tự làm mình và các nhà phát triển sau bạn lý do thực sự tại sao các trường hợp của OrderRequest là cần thiết.

Hàm tạo là không thể tránh khỏi trong các ngôn ngữ lập trình hướng đối tượng; Chỉ cần làm nó private và phơi bày các hàm tạo static:

private OrderRequest() { ... }

public OrderRequest CreateForCustomer (int customerId)

{

var request = new OrderRequest();

...

return request;

}

Một kịch bản nâng cao hơn là khi bạn chỉ cần sắp xếp một dịch vụ riêng biệt cho một số thực thể và quản lý những thứ như đối tượng gộp lại bên trong.

**Chú ý:** Trong chương tiếp theo, bạn sẽ thấy rằng các hàm tạo thường được đánh dấu là protected thay vì private. Việc sử dụng các hàm tạo protected có liên quan đến sự bền bỉ và các công cụ O/RM. Khi một O/RM thể hiện một thể hiện thực thể và làm nó với nội dung cơ sở dữ liệu, nó thực sự trả về một thể hiện của một lớp được tạo ra động mà thừa kế từ lớp thực thể (ví dụ như một proxy). Điều này cho phép các công cụ O / RM gọi phương thức khởi tạo protected; Một constructor mặc định riêng - cũng như một lớp thực thể kín - sẽ làm cho sự kiên trì thông qua các công cụ O / RM khá khó hiểu.

**Value objects scaffolding**

Giống như một thực thể, một đối tượng giá trị là một lớp tổng hợp dữ liệu. Hành vi không liên quan đến các đối tượng giá trị; Các đối tượng có giá trị có thể vẫn có các hàm, nhưng những hàm này chủ yếu là hàm trợ giúp. Tuy nhiên, không giống các thực thể, các đối tượng giá trị không cần nhận dạng bởi vì chúng không có trạng thái biến và được xác định đầy đủ bởi dữ liệu của chúng.

**Cấu trúc dữ liệu**

Hầu hết thời gian, một đối tượng giá trị là một vùng dữ liệu thu gọn dữ liệu thông qua các trình getters public và chỉ nhận các giá trị thông qua constructor. Đây là một ví dụ:

public class Address

{

public Address(string street, string number, string city, string zip)

{

Street = street;

Number = number;

City = city;

Zip = zip;

}

public string Street { get; private set; }

public string Number { get; private set; }

public string City { get; private set; }

public string Zip { get; private set; }

// Equality members

...

}

Các hàm có thể được định nghĩa, nhưng chúng phải là các hàm thuần túy không làm thay đổi trạng thái. Ví dụ, bạn có thể có đối tượng giá trị sau để đại diện cho số điểm của một trận bóng rổ:

public class Score

{

public Score(int team1, int team2)

{

Team1 = team1;

Team2 = team2;

}

public int Team1 { get; private set; }

public int Team2 { get; private set; }

public bool IsLeading(int index)

{

if (index != 1 && index != 2)

throw new ArgumentException(“index”);

if (index == 1 && team1 > team2) return true;

if (index == 2 && team2 > team1) return true;

return false;

}

}

Hàm IsLeading không thay đổi trạng thái của đối tượng, nhưng sự hiện diện của nó sẽ làm cho nội dung của người điều khiển đối tượng giá trị sử dụng.

**Tính bình đẳng**

Sử dụng thuật ngữ .Net Framework, chúng ta có thể nói rằng một đối tượng giá trị là một kiểu bất biến. Một bất biến là bất kỳ loại nào không có trạng thái và yêu cầu các trường hợp mới mỗi khi các giá trị mới được yêu cầu. Như đã đề cập ở phần đầu của chương, kiểu String là ví dụ kinh điển của một loại Immu-table trong .NET Framework.

Bạn cần phải ghi đè cả phương thức Equals và GetHashCode để đảm bảo rằng trong các đối tượng mô hình được tạo ra từ cùng một dữ liệu được coi là các đối tượng bằng nhau. Dưới đây là cách kiểm tra xem liệu hai trường hợp Địa chỉ có giống nhau không:

public class Address

{

...

public override bool Equals(object obj)

{

if (this == obj)

return true;

if (obj == null || GetType() != obj.GetType())

return false;

var other = (Address) obj;

return string.Equals(Street, other.Street) &&

string.Equals(Number, other.Number) &&

string.Equals(City, other.City) &&

string.Equals(Zip, other.Zip);

}

public override int GetHashCode()

{

const int hashIndex = 307;

var result = (Street != null ? Street.GetHashCode() : 0);

result = (result \* hashIndex) ^ (Number != null ? Number.GetHashCode() : 0);

result = (result \* hashIndex) ^ (City != null ? City.GetHashCode() : 0);

result = (result \* hashIndex) ^ (Zip != null ? Zip.GetHashCode() : 0);

return result;

}

}

Logic bình đẳng tùy chỉnh là de facto một tính năng bắt buộc trong cả hai thực thể và các đối tượng giá trị.

**Mã băm và sự bình đẳng trong .NET Framework**

Hướng dẫn .NET Framework khuyến cáo mạnh mẽ rằng bất cứ khi nào bạn ghi đè Equals bạn cũng ghi đè cho GetHashCode. Nhưng tại sao vậy?

Hàm Equals luôn được sử dụng khi bình đẳng được kiểm tra một cách rõ ràng, chẳng hạn như khi các cuộc gọi mã của bạn Equals trực tiếp hoặc sử dụng toán tử ==. GetHashCode không tham gia vào quá trình đánh giá bình đẳng. Sau đó, việc thực hiện nó không ảnh hưởng đến việc đánh giá bình đẳng. Tuy nhiên, bạn sẽ có được một cảnh báo trình biên dịch nếu bạn ghi đè Equals nhưng không GetHashCode.

GetHashCode đi vào hoạt động chỉ khi các đối tượng của bạn sẽ được lưu trữ trong bảng băm

hoặc được sử dụng như các Dictionary. Trong trường hợp này, .NET Framework sử dụng giá trị được trả về bởi GetHashCode để index mục tương ứng trong bảng băm hoặc từ điển. Hai đối tượng đối tượng là cùng một đối tượng theo Equals nhưng trả lại các giá trị khác nhau theo GetHashCode sẽ không bao giờ được coi là bằng nhau. Do đó, các giá trị khác nhau sẽ được chọn từ bảng băm hoặc từ điển.

Khi GetHashCode trả về cùng một giá trị cho hai đối tượng, bạn sẽ bị va chạm và Equals được gọi để đảm bảo các đối tượng là như nhau. Mục tiêu thực hiện, tất nhiên, là để giảm thiểu số lượng va chạm bằng cách cho mỗi đối tượng với một sự kết hợp duy nhất của các giá trị một mã băm khác nhau.

Cuối cùng, lưu ý rằng mã băm không bao giờ nên thay đổi trong suốt cuộc đời của một đối tượng. để xảy ra, chỉ có bất động thuộc tính nên được sử dụng để tính toán mã. Quy tắc này hoàn toàn phù hợp với các nguyên tắc cho các thực thể và các đối tượng giá trị

**Operators overloading**

Công thức Equal không cho phép bạn tự động sử dụng toán tử == và! = Equality trên các đối tượng đã ghi đè. Trong hướng dẫn .NET Framework, Microsoft không khuyến khích bạn thực hiện quá tải các toán tử khi các thuộc tính có thể thay đổi được sử dụng để kiểm tra sự bình đẳng. Đây không phải là trường hợp đối với các thực thể và đối tượng giá trị. Vì vậy, người vận hành quá tải là hoàn toàn an toàn. Đây là một ví dụ:

public static bool operator ==(Customer c1, Customer c2)

{

// Both null or same instance

if (ReferenceEquals(c1, c2))

return true;

// Return false if one is null, but not both

if (((object)c1 == null) || ((object)c2 == null))

return false;

return c1.Equals(c2);

}

public static bool operator !=(Customer c1, Customer c2)

{

return !(c1 == c2);

}

Chúng ta thảo luận các thực thể và các đối tượng giá trị, khám phá sự khác biệt của chúng, và giờ đây hiểu được các sắc thái bình đẳng. Nhưng vào cuối ngày, khi nào bạn sẽ sử dụng các đối tượng giá trị trong một mô hình miền?

**Tính biểu cảm**

Như chúng ta thấy nó, lợi ích chính mà giá trị các đối tượng mang lại cho bảng là tính biểu cảm. Có hai khía cạnh của biểu cảm: thay thế các kiểu nguyên thủy và thay thế cụm dữ liệu.

Các kiểu nguyên thủy đôi khi quá cơ bản để được sử dụng rộng rãi trong một mô hình mà được khai báo đại diện cho một mảng nghiệp vụ thực tế. Các kiểu nguyên thủy là những gì chúng ta thực sự sử dụng khi chúng ta tồn tại các thực thể; Tuy nhiên, ở mức trừu tượng cao hơn của mô hình miền, các kiểu nguyên thủy thường là các yếu tố của sự không rõ ràng. Số nguyên biểu thị là gì? Là một lượng, một nhiệt độ, một lượng tiền, đo lường một? Nó có phải là một yếu tố của một phạm vi?

Trong thế giới thực, bạn có số lượng, nhiệt độ, số lượng-không phải nguyên số nguyên.

Trong khi bạn vẫn có thể sử dụng các số nguyên và cảm thấy vui vẻ, sử dụng các kiểu nguyên thủy buộc bạn phải xác nhận giá trị bất kỳ vị trí nào trong mô hình mà một số nguyên được sử dụng để có nghĩa số lượng, phép đo, phần tử của một dãy, vân vân. Không chỉ mã xác nhận của bạn có thể là rườm rà để viết, nhưng nó cũng có thể được nhân đôi bởi vì nó áp dụng cho setter chứ không phải là loại chính nó. Đây là cách bạn có thể mô hình hoá một số tiền hiệu quả hơn:

public sealed class Money

{

public Money(Currency currency, decimal amount)

{

Currency = currency;

Value = amount;

}

public Currency Currency { get; private set; }

public decimal Value { get; private set; }

// Add operator overloads

...

}

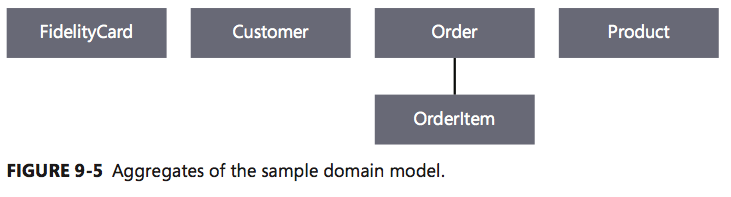
Đơn vị tiền tệ trong ví dụ này là một đối tượng giá trị được xây dựng xung quanh biểu tượng và tên.

Nói chung, bằng cách sử dụng các đối tượng giá trị, bạn đơn giản hóa mã và cũng tạo ra một mô hình gần với ngôn ngữ phổ biến hơn. Trong thuật ngữ tái cấu trúc, các cụm dữ liệu là mùi mã nổi tiếng. Một cụm dữ liệu là khi một nhóm các mục dữ liệu thường được sử dụng cùng nhau nhưng là các mặt hàng độc lập. Cuối cùng, một cụm dữ liệu chỉ là một tập hợp các giá trị nguyên thủy có thể dễ dàng biến thành một đối tượng mới.

Một ví dụ điển hình là tiền tệ và số tiền trong đối tượng giá trị tiền. Với ngôn ngữ phổ thông, bạn có thể có tiền, chứ không phải tiền tệ và số tiền; Bạn có địa chỉ, không phải các mục riêng lẻ như đường phố, thành phố và mã bưu điện. Đây là nơi giá trị các đối tượng nó.

**Xác thực tổng hợp**

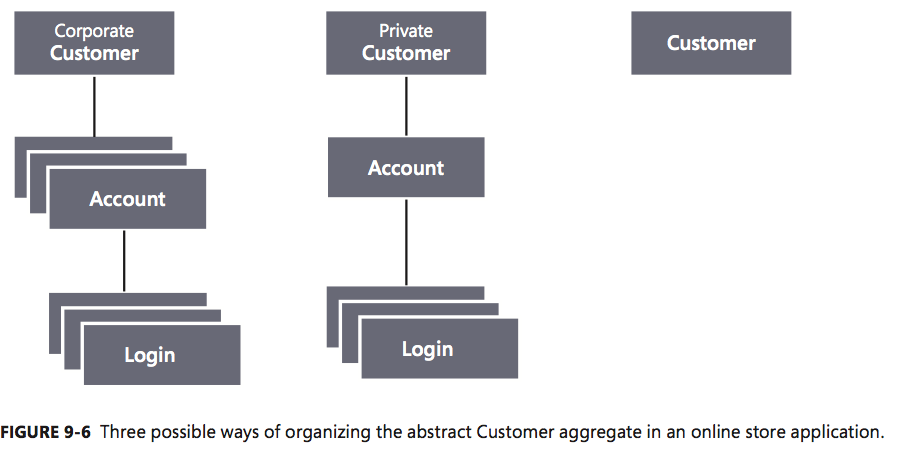
Bây giờ hãy tập trung vào các thực thể được liệt kê trong Bảng 9-2 và được sử dụng trong ứng dụng mẫu. Ứng dụng I-Buy-Stuff có nghĩa là một cửa hàng trực tuyến đơn giản, và điều này được phản ánh bởi số lượng hạn chế của các cơ quan và mối quan hệ hạn chế của chúng. (Xem Hình 9-5). Trong một trang web mà chúng ta chỉ xem xét một số trường hợp sử dụng như tìm kiếm và đặt hàng, Khách hàng, Đơn đặt hàng và Sản phẩm là tập hợp và Khách hàng và Sản phẩm là tập hợp đơn. Cũng có thể nói cho FidelityCard.



Hãy để ý nhiều hơn về tập hợp. Tuy nhiên, từ quan điểm hành vi, Tập trung vào các trường hợp sử dụng mà chúng tôi đang cân nhắc, tập hợp thú vị nhất là Đặt hàng. Tuy nhiên, một số cân nhắc quan tâm cũng áp dụng cho Khách hàng.

**Sự tổng hợp khách hàng**

Khách hàng được đặc trưng bởi một số dữ liệu cá nhân như tên, địa chỉ, tên người dùng và chi tiết thanh toán. Rõ ràng chúng ta giả định rằng cửa hàng trực tuyến của chúng ta chỉ có một loại khách hàng và không phân biệt, chẳng hạn như giữa khách hàng doanh nghiệp và khách hàng cá nhân. Ví dụ: khách hàng doanh nghiệp có thể có nhiều người dùng mua sắm theo cùng một tài khoản khách hàng. Hãy xem hình 9-6.



Biểu đồ đầu tiên mô tả một khách hàng doanh nghiệp, trong một tài khoản tài chính, có thể có nhiều người dùng (ví dụ như một số nhân viên) với các địa chỉ giao hàng khác nhau và người đăng nhập vào hệ thống bằng các chứng chỉ khác nhau. Biểu đồ trung tâm đại diện cho một khách hàng cá nhân với một tài khoản người dùng. Cuối cùng, biểu đồ thứ ba đại diện cho việc triển khai chúng tôi sử dụng trong chương này, nơi khách hàng đại diện cho một tài khoản người dùng duy nhất và thông tin đăng nhập được lưu trữ bên trong thực thể Khách hàng.

**Chú ý:** Như chúng ta đã nói ở trên trong bảng 9-2, hầu hết các chuyên gia về tên miền của các cửa hàng trực tuyến, người dùng và khách hàng sẽ là các khái niệm và thực thể riêng biệt. Điều này không ảnh hưởng đến tính tổng quát của ví dụ của chúng ta, nhưng điều quan trọng cần nhớ là phải nhận ra sự tinh tế và chiều sâu của thiết kế theo miền

Một khía cạnh khác để xem xét là đăng nhập vào hệ thống. Bạn nên cân nhắc việc sử dụng các thực thể khác biệt để giữ các chứng chỉ và phần còn lại của dữ liệu và hành vi? Đó có thể là những gì bạn sẽ muốn làm hầu hết thời gian. Có hai sự cân nhắc nữa để chỉ cho bạn theo hướng này:

* Mỗi khách hàng hoặc tài khoản có thể có nhiều cách để đăng nhập, bao gồm các thông tin đăng nhập được lưu trữ trong hệ thống cục bộ và đăng nhập xã hội nhiều lần.
* Nếu bạn sử dụng ASP.NET Identity để thực hiện thẩm định, thì ở giai đoạn hiện tại của công nghệ, bạn buộc các lớp người dùng phải thực hiện giao diện bên ngoài cố định(giao diện IUser). Điều này sẽ làm giảm POCO (nền tảng cũ của đối tượng C#) trạng thái của tên miền.

Trong ứng dụng mẫu, khách hàng trùng với người dùng đăng nhập. Lớp khách hàng có cấu trúc sau:

public class Customer : IAggregateRoot

{

public static Customer CreateNew(Gender gender, string id, string firstname, string lastname, string email)

{

var customer = new Customer {

CustomerId = id,

Address = Address.Create(),

Payment = NullCreditCard.Instance,

Email = email,

FirstName = firstname,

LastName = lastname,

Gender = gender

};

return customer;

}

public string CustomerId { get; private set; }

public string PasswordHash { get; private set; }

public string FirstName { get; private set; }

public string LastName { get; private set; }

public string Email { get; private set; }

public Gender Gender { get; private set; }

public string Avatar { get; private set; }

public Address Address { get; private set; }

public CreditCard Payment { get; private set; }

public ICollection<Order> Orders { get; private set; }

// More methods here such as SetDefaultPaymentMode.

// A customer may have a default payment mode (e.g., credit card), but

// different orders can be paid in different ways.

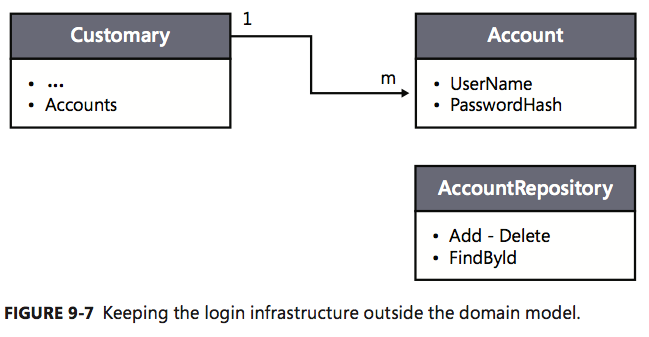
...

}

Khi khách hàng được tạo ra, thuộc tính Name cho biết tên đăng nhập trong khi FirstName và LastName xác định tên hiển thị đầy đủ. Thẻ tín dụng và thông tin địa chỉ có thể được thêm vào thông qua các phương pháp đặc biệt. Khách hàng được tạo ra thông qua trang Đăng ký kinh điển của hầu hết các trang web, và các thông tin được lưu trữ trong cơ sở dữ liệu cục bộ. Nếu người dùng đăng nhập qua Facebook hoặc Twitter, một cá thể khách hàng mới cũng được tạo ra nếu nó không tồn tại.

**Chú ý:** Bạn có thể thấy rằng với phương pháp cực kỳ đơn giản này cùng một cá nhân kết thúc với hai hồ sơ khách hàng riêng biệt nếu cô ấy đăng nhập thông qua Twitter và Facebook. Điều đó làm cho nó cần thiết để cải thiện mã để hỗ trợ đăng nhập nhiều cho mỗi khách hàng. Lý tưởng nhất là bạn có thể muốn duy trì một bảng các tài khoản và thiết lập quan hệ một-nhiều với khách hàng.

Việc thực hiện cụ thể bạn tìm thấy trong mã mẫu sử dụng các phương thức ASP.NET Identity để làm Đăng nhập và đăng xuất. Tuy nhiên, nó bỏ qua ASP.NET Identity hoàn toàn khi nói đến việc thêm, truy xuất và xác nhận người dùng. Lý do là để tránh làm cho mô hình miền rõ ràng phụ thuộc vào khuôn khổ ASP.NET Identity. Hình 9-7 phác thảo một sự điều chỉnh có thể đối với mô hình để giữ đăng nhập bên ngoài mô hình miền và liên kết nhiều tên người dùng với cùng một thể hiện của khách hàng.



**Chú ý:** Sử dụng lược đồ thể hiện trong hình 9-7, nơi Tài khoản là một phần của miền, làm cho toàn bộ mô hình có tính co dãn và dễ dàng thích ứng với các kịch bản thực. Tuy nhiên, nó không giải quyết vấn đề pha trộn mô hình miền với sự phụ thuộc vào giao diện ASP.NET Identity IUser. Nếu bạn muốn tận dụng cơ sở vật chất sẵn có của ASP.NET Identity cho đến việc quản lý các tài khoản này, bạn nên giữ mối quan tâm về xác thực trên mô hình miền. Nếu bạn thấy Tài khoản như là một phần của một trong các thực thể, chúng tôi khuyên bạn nên sử dụng kho lưu trữ tài khoản để thực hiện CRUD thay vì đối tượng lưu trữ người dùng kinh điển của ASP.NET Identity.

**Sự tổng hợp sản phẩm**

Một trong những trách nhiệm của một thực thể đang mang dữ liệu và dữ liệu bắt nguồn từ hành vi mong đợi của thực thể. Thực thể sản phẩm phải nói với thế giới về mô tả và giá cả của nó. Ngoài ra, với mục đích sử dụng trường hợp, phải biết được sản phẩm hiện đang "nổi bật" trên trang web hay không và mức tồn kho hiện tại là bao nhiêu. Ví dụ như các mô tả dưới đây:

public class Product : IAggregateRoot

{

public int Id {get; private set;}

public string Description {get; private set;}

public Money UnitPrice {get; private set;}

public int StockLevel {get; private set;}

public bool Featured {get; private set;}

// Methods

public void SetAsFeatured() { ... }

...

// Identity management as for Customer

... }

239-254 Nguyen Van Quang

Hãy xem xét thuộc tính StockLevel. Đầu tiên, số nguyên được sử dụng để xác định stock level có thể được thay đổi thành một unsigned integer hay không, vì giá trị âm không phải là các giá trị hợp lệ đối với những gì thuộc tính đại diện. Thậm chí tốt hơn, một đối tượng giá trị đặc biệt có thể được tạo ra để đại diện cho số lượng như một số tự nhiên.

Tuy nhiên, như bạn thấy, StockLevel, giống như tất cả các thuộc tính trong đoạn mã, đều có private setter. Điều này có nghĩa thuộc tính là read-only và sau đó, đọc nó như một số nguyên là tốt. Ai sẽ thiết lập thuộc tính StockLevel? Hai trương hợp có thể xảy ra:

* Đối tượng chỉ được thực hiện chỉ từ lớp persistence (persistence layer).
* Đối tượng có một factory method nếu nó được cho phép để thể hiện từ client code.

Trên thực tế, lựa chọn thứ hai là lựa chọn không đáng tin cậy, mặc dù factory method có thể hữu ích để có cho việc thực hiện mục đích.

Thuộc tính StockLevel trong mô hình đơn giản này chỉ ra số lượng mặt hàng của một sản phẩm đang có trong kho. Con số này giảm khi đơn đặt hàng được đặt và tăng lên khi người quản lý kho đặt hàng đến nhà cung cấp. Sau đó, một phương pháp là cần phải lập trình để sửa đổi stock level của một sản phẩm. Người quản lý cửa hàng là một phần của công ty, vốn là một bounded context khác.

**The *Order* aggregate (Nhận, tổng hợp đơn đặt hàng)**

Trong I-Buy-Stuff, một đơn đặt hàng được tạo ra khi xử lý nội dung của một giỏ hàng, được đại diện bởi lớp ShoppingCart. Khung (or cấu trúc) của lớp được hiển thị ở đây:

public class Order : IAggregateRoot

{

public int OrderId { get; private set; }

public Customer Buyer { get; private set; }

public OrderState State { get; private set; }

public DateTime Date { get; private set; }

public Money Total { get; private set; }

public ICollection<OrderItem> Items { get; private set; }

...

}

Lớp cần phải có public factory method vì ở một số logic nghiệp vụ cần phải tạo ra một đơn đặt hàng trong khi xử lý một yêu cầu. Khi tạo đơn đặt hàng, bạn chỉ định ID và reference (liên kết, ràng buộc, tham khảo) khách hàng. Ngày được đặt tự động, vì vậy nó là trạng thái của đơn đặt hàng.

Trong ví dụ, chúng tôi giả sử đơn đặt hàng được tạo ra khi khách hàng click vào check out. Sẽ có một “domain service” để sắp xếp quá trình thanh toán; Một trong những bước là tạo ra thứ tự. Bộ máy đặt hàng có thể được đánh dấu bên trong và sử dụng một cấu trúc không công khai:

internal static Order CreateFromShoppingCart(ShoppingCart cart)

{

var order = new Order(cart.CustomerId, cart.Customer);

return order;

}

protected Order(int orderId, Customer customer)

{

OrderId = orderId;

State = OrderState.Pending;

Total = Money.Zero;

Date = DateTime.Today;

Items = new Collection<OrderItem>();

Buyer = customer;

}

Để sửa đổi trạng thái của đơn đặt hàng và thêm items (sản phẩm), bạn cần có phương thức “ad hoc” (đặc biệt). Dưới đây là một số phương pháp thay đổi trạng thái:

public void Cancel()

{

if (State != OrderState.Pending)

throw new InvalidOperationException(

“Can’t cancel an order that is not pending.”);

State = OrderState.Canceled;

}

public void MarkAsShipped()

{

if (State != OrderState.Pending)

throw new InvalidOperationException(

“Can’t mark as shipped an order that is not pending.”);

State = OrderState.Shipped;

}

Như bạn thấy, đây không phải là “rocket science”, nhưng bằng cách sử dụng phương thức này thay vì các bộ setters sẽ cải thiện khả năng đọc của code và quan trọng hơn, giữ nó aligned (liên kết, theo) với ngôn ngữ phổ biến.

Conventions (quy tắc) được sử dụng để đặt tên cho các phương thức, trên thực tế, không thể đặt tùy tiện. Các điều khoản như *Archive* và *MarkAsShipped* phải được lấy từ ngôn ngữ phổ biến và thuật ngữ phản chiếu được sử dụng trong business (kinh doanh).

**Note**: Giống như các phương thức thường được thực hiện như là phương thức void. Nó làm trên “total sense” (toàn bộ) bởi vì chúng chỉ đại diện cho hành động. Tuy nhiên, tùy thuộc vào ngữ cảnh có thể xem xét chúng như là một fluent interface. Bạn chỉ cần thực hiện các phương thức để trả về *this*, nó cho phép gọi chuỗi và làm cho mọi thứ trở nên dễ đọc hơn. Đó là một khả năng để xem xét.

Có một số điều thú vị để nói về các thuộc tính *Items* và *Total*. Hãy bắt đầu với việc đưa ra một số vấn đề liên quan đến việc “having collections in an aggregate”. Chúng tôi mô tả các vấn đề và cung cấp một số gợi ý. Tuy nhiên, câu trả lời chính xác sẽ xuất hiện trong chương tiếp theo và từ việc sử dụng một kiến trúc hỗ trợ DDD khác - CQRS thay vì Domain Model.

Thuộc tính *Items* được định nghĩa như là một bộ sưu tập. Điều này có nghĩa là bất kỳ mã nào nhận được một *Order* instance có thể liệt kê nội dung của bộ sưu tập nhưng không thể thêm hoặc xoá các mục. Thật không may, không có cách nào dễ dàng để ngăn chặn mã khách hàng đó truy cập và cập nhật nội dung trong bộ sưu tập. Thậm chí tệ hơn, khía cạnh này vẫn không thay đổi ngay cả khi bạn tiếp tục giới hạn bộ sưu tập đến một IEnumerable cơ bản <T>. Đoạn mã sau luôn được cho phép và, ngoài ra, bạn luôn có thể sử dụng LINQ để chọn một thứ tự cụ thể trong bộ sưu tập:

someOrder.Items.First().Quantity++;

Trong kịch bản này, đơn hàng chỉ có thể đọc. Nó được tạo ra trong quá trình checkout (thanh toán); nó không thể được điều chỉnh, nhưng nó có thể được xem. Vì vậy, bạn muốn *Items* collection được dùng dưới dạng nội dung chỉ đọc. Làm thế nào bạn có thể ngăn ngừa thay đổi đối tượng trong collection? Thật không may, ngay cả những điều sau đây sẽ không hoạt động:

public class Order

{

private Collection<OrderItem> \_items = new Collection<OrderItem>();

public ICollection<OrderItem> Items

{

get { return \_items.AsReadOnly(); }

}

}

*Items* collection là chỉ được đọc, nhưng bạn vẫn có thể truy lục các phần tử riêng lẻ và thay đổi chúng theo chương trình.

Giải pháp dứt khoát cho vấn đề này đi kèm với sự tách biệt giữa truy vấn và lệnh và với việc cung cấp các mô hình khác nhau cho việc đọc và viết các thao tác. Chúng tôi sẽ trình bày điều này trong hai chương kế tiếp.

*Total* property chỉ ra tổng giá trị của đơn đặt hàng. Làm thế nào bạn phải đối phó với điều đó? Thuộc tính đó được đặt ở trong nội dung của đơn đặt hàng. Về lý thuyết, tổng của một đơn đặt hàng có thể được tính theo yêu cầu, lặp lại trên đồ thị đơn đặt hàng bằng một ad hoc method (phương pháp đặc biệt):

public Money GetTotal()

{

var amount = Items.Sum(item => item.GetTotal().Value);

return new Money(Currency.Default, amount);

}

Tuy nhiên, giải pháp này có một hạn chế đáng kể. Nó buộc *Order* instance phải có trong bộ nhớ “entire graph” của các item. Có thể có các tình huống mà bạn chỉ muốn biết về tổng số đơn đặt hàng, mà không có tất cả các chi tiết. Một phương pháp như *GetTotal* sẽ buộc “deserialization” của entire graph và tạo ra nhiều lượng truy xuất hơn nữa đến cơ sở dữ liệu.

Bằng cách có một thuộc tính đơn giản trong *Order*, bạn có thể quyết định một cách thông minh liệu có nên loại bỏ entire graph hay không. Thuộc tính *Total* có thể được thiết lập bởi kho lưu trữ lệnh khi nó trả về một instance (có thể là một thao tác SUM đơn giản ở trong SQL), hoặc nó có thể là kết quả trong giản đồ quan hệ của cơ sở dữ liệu. Bất cứ khi nào đơn đặt hàng còn duy trì (kiểu như chưa bị chấm dứt), tổng số được tính và lưu cho các truy vấn khác.

**The *FidelityCard* aggregate**

*FidelityCard* aggregatelà một lớp rất đơn giản tóm tắt thông qua việc tích luỹ điểm hoạt động của một khách hàng trong trang web.

public class FidelityCard

{

public static FidelityCard CreateNewCard(string number, Customer customer)

{

var card = new FidelityCard {Number = number, Owner = customer};

return card;

}

protected FidelityCard()

{

Number = “”;

Owner = UnknownCustomer.Instance;

Points = 0;

}

public string Number { get; private set; }

public Customer Owner { get; private set; }

public int Points { get; private set; }

public int AddPoints(int points)

{

Points += points;

return Points;

}

}

Mỗi đơn hàng của khách hàng sẽ tăng tổng số điểm trên thẻ, và những điểm này sau đó được sử dụng để xác định tình trạng thưởng xuyên mà khách hàng có thể sử dụng để được giảm giá. Tất cả logic của mức chiết khấu và mức đều lấy cảm hứng từ lĩnh vực kinh doanh và có thể phức tạp và khác nhau như nhu cầu thực tế.

Nội dung của đối tượng *FidelityCard* được sử dụng bởi các domain services để tính mức thưởng và mức chiết khấu.

Special cases

In the context of a domain model, one of the most commonly used accessory design patterns is the Special Case pattern, which is deined here: [h*ttp://martinfowler.com/eaaCatalog/specialCase.html*](http://martinfowler.com/eaaCatalog/specialCase.html). The pattern addresses a simple question: when some code needs to return, say, a *Customer* object but no suitable object is found, what is the best practice? Should you return NULL? Should you return odd values? Should you make an otherwise clean API overly complex and make it distinguish whether or not a result exists? Have a look at this code. It belongs to the *OrderRepository* class of I-Buy-Stuff and retrieves an order by ID while restricting the search to a particular customer ID:

Trong ngữ cảnh của domain model, một trong những design patterns phổ biến nhất là Special Case pattern, được thực hiện ở đây: [h*ttp://martinfowler.com/eaaCatalog/specialCase.html*](http://martinfowler.com/eaaCatalog/specialCase.html). Mẫu xác định một câu hỏi đơn giản: khi một số mã cần phải trả về, một đối tượng Customer không tìm thấy đối tượng phù hợp, thực tiễn tốt nhất là gì? Bạn nên trả lại NULL? Nên trả lại các odd values? Nếu tạo một API hoàn toàn sạch sẽ phức tạp và làm cho nó phân biệt kết quả có tồn tại hay không? Hãy xem mã này. Nó thuộc về lớp *OrderRepository* của I-Buy-Stuff và lấy ra một đơn hàng bằng ID trong khi hạn chế tìm kiếm cho một ID khách hàng cụ thể:

public Order FindByCustomerAndId(int id, string customerId)

{

using (var db = new DomainModelFacade())

{

try

{

var order = (from o in db.Orders

where o.OrderId == id &&

o.Buyer.CustomerId == customerId

select o).First();

return order;

}

catch (InvalidOperationException)

{

return new NotFoundOrder();

}

}

}

Phương pháp thứ nhất quăng ra nếu order không được tìm thấy. Trong trường hợp này, mã trả về một instance vừa được tạo ra của lớp *NotFoundOrder*:

public class NotFoundOrder : Order

{

public static NotFoundOrder Instance = new NotFoundOrder();

public NotFoundOrder() : base(0, UnknownCustomer.Instance)

{

}

}

*NotFoundOrder* chỉ là một lớp bắt nguồn mà đặt tất cả các thuộc tính cho các giá trị mặc định của chúng. Bất kỳ mã nào dự kiến một *Order* có thể đối phó với *NotFoundOrder*; Và kiểm tra kiểu sẽ giúp bạn điều chỉnh nếu có sự cố:

if(order is NotFoundOrder)

{

...

}

Đây là vấn đề quan trọng của *Special Case* pattern. Ngoài việc triển khai cơ bản này, bạn có thể thêm nhiều tính năng bổ sung như bạn muốn, bao gồm cả singleton instance.

**Persisting the model**

Một domain model tồn tại để được duy trì, và thường là một O/RM sẽ làm điều đó. Tất cả những gì một O/RM làm là map thuộc tính đến cột của một bảng cơ sở dữ liệu và quản lý việc đọc và ghi. Dù đây chỉ là 10,000-foot, bird’s-eye view.

**What an O/RM does for you**

Nói chung, một O/RM có thể tóm tắt nhiệm vụ bằng bốn điểm sau:

* CRUD (Create-Read-Update-Delete)
* Query engine (công cụ truy vấn)
* Transactional engine (công cụ transaction)
* Concurrency (truy cập đồng thời)

Query và transactional engines đề cập đến 2 design patterns, cụ thể là the Query Object pattern (xem tại [*http://martinfowler.com/eaaCatalog/queryObject.html*](http://martinfowler.com/eaaCatalog/queryObject.html)), và the Unit of Work pattern (xem tại [*http://martinfowler.com/eaaCatalog/unitOfWork.html*](http://martinfowler.com/eaaCatalog/unitOfWork.html)).

Ngày nay, trên nền tảng .NET gần như tất cả các O/RMs cung cấp một sự thực hiện theo mẫu của Query Object pattern dựa trên cú pháp LINQ. *Unit of Work* được cung cấp thông qua các tính năng gốc của đối tượng O/RM. Trong Entity Framework, đối tượng này là *ObjectContext* trong các “lavor” khác nhau của nó, chẳng hạn như *DbContext*.

Một O/RM là một công cụ năng suất tuyệt vời. Nó không thực sự làm những điều kỳ diệu, nhưng nó tiết kiệm cho bạn rất nhiều đoạn dễ bị lỗi và cồng kềnh. Tất cả những gì nó cần là các hướng dẫn về cách ánh xạ các thuộc tính của các đối tượng trong mô hình miền với các cột của các bảng cơ sở dữ liệu quan hệ.

Khi nói đến điều này, bạn nhận ra rằng cơ sở dữ liệu thực sự rất quan trọng ngay cả khi bạn thiết kế domain driven và xây dựng một mô hình “persistence ignorant” và ở trạng thái “as agnostic as possible” đối với cơ sở dữ liệu. Tóm lại, cơ sở dữ liệu và O/RM là hai trở ngại điển hình buộc bạn phải “concessions” và giới thiệu các model features chỉ phục vụ cho “persistence”.

**Making concessions to persistence**

“Concession” phổ biến nhất mà bạn, như một người kiến trúc Domain Model, phải làm cho một O/RM là sự sẵn có của các default constructors trên tất cả các lớp persistent.

protected Customer()

{

// Place here any initialization code you may need

...

}

Constructor là cần thiết cho O/RM để cụ thể hoá một thực thể từ cơ sở dữ liệu. Nói cách khác, nó vẫn cần các công cụ cấp thấp để tạo ra một instance của lớp đó. Các phương thức Factory này là trừu tượng nhằm phục vụ cho mục đích của ubiquitous language (ngôn ngữ phổ biến). Constructor, mặt khác, là cách duy nhất được biết đến là trình biên dịch trong C # (và các ngôn ngữ hướng đối tượng khác) cho phép bạn tạo các instance mới của một lớp.

Thật tuyệt, mặc dù, bạn có thể sử dụng các công cụ O/RM để ẩn constructor mặc định không tham số từ public view. Khi cụ thể hoá một thực thể từ cơ sở dữ liệu, một O/RM thường trả về instance của một lớp học được tạo tự động kế thừa từ thực thể thực sự. Bằng cách này, mã được sinh tự động tạo ra truy cập vào các protected constructor.

Ngoài điểm này, những thay đổi khác mà bạn có thể buộc phải làm để làm cho mô hình “persistent” phụ thuộc vào khả năng và khiếm khuyết của O/RM. Ví dụ: cho đến phiên bản 5, Entity Framework không thể xử lý các loại *enum*. Trong phiên bản 6.1, nó vẫn không thể xử lý, ít nhất là trong một cách mặc định, các mảng của các kiểu nguyên thủy.

**Note**: Khi nói đến mảng, các chuyên gia cơ sở dữ liệu có thể cho bạn biết rằng các mảng không đặc biệt trong thế giới của cơ sở dữ liệu và bạn nên tìm một giải pháp lý tưởng mà không cần đến mảng. Vấn đề là các cơ sở dữ liệu quan hệ không cung cấp một cách để đọc và ghi các mảng trực tiếp. Kết quả là việc lưu trữ các mảng đòi hỏi cần được giải quyết, nhưng nó không phải là không thể cũng không đặc biệt khó khăn. Tuy nhiên, khi bạn nghĩ về mô hình theo cách không tin cậy lắm trong cơ sở dữ liệu, các mảng có thể là công cụ mô hình tuyệt vời cho các khía cạnh nghiệp vụ đòi hỏi trình tự các dữ liệu liên quan.

Tất cả điều này là để nói rằng trước tiên bạn nên cố gắng để có được một domain model lý tưởng mà phù hợp với các tính năng của ngôn ngữ phổ biến; Tiếp theo, bạn nên cố gắng làm sao cho nó phù hợp với O/RM (nên kể đến Entity Framework) để tồn tại nó.

**Note**: Entity Framework chắc chắn không phải là O/RM duy nhất có sẵn cho .NET Framework. Một đối thủ phổ biến là NHibernate. Tuy nhiên, nhận thức là cuộc tranh luận giữa hai bên về việc sử dụng không còn là vấn đề nữa. Và bởi vì nó được tích hợp chặt chẽ với Visual Studio, Entity Framework là lựa chọn hàng đầu mà nhiều người xem xét. Ngoài ra, có một danh sách dài các sản phẩm có sẵn từ các nhà cung cấp như Devxpress và Telerik. Nó thực sự được ưu tiên hơn.

**The Entity Framework Code-First approach**

Entity Framework là O/RM được cung cấp bởi Microsoft .NET. Nó thậm chí không được phát hành khi chúng tôi viết phiên bản đầu tiên của cuốn sách này. Nó có thể được coi là sự lựa chọn tự nhiên nhất hiện nay đối với bất kỳ dự án .NET nào - gần như là một tiêu chuẩn trên thực tế.

Về cơ bản, Entity Framework có hai loại: Database First vaf Code First. Cũng có loại thứ ba, được gọi là Model First, nhưng đó là một phương pháp lai, cách trung gian giữa visual diagram model và đã sớm bị thay thế bởi Code First. Database-First đọc cấu trúc của một cơ sở dữ liệu đã có và trả về một tập các lớp “anemic” mà bạn có thể mở rộng bằng các phương pháp sử dụng “the partial class mechanism”. Cách tiếp cận Code-First bao gồm việc viết một loạt các lớp -ví dụ như một Domain Model, như đã thảo luận cho đến thời điểm này. Tiếp theo, bạn thêm một lớp thêm (hoặc chú thích dữ liệu) để map các thuộc tính cho các bảng. Lớp mapping này báo cho O/RM biết về cơ sở dữ liệu để tạo ra (hoặc mong đợi), các mối quan hệ của nó, các ràng buộc và các bảng và quan trọng hơn, nó báo cho O/RM nơi để lưu hoặc đọc các giá trị thuộc tính.

Ví dụ về I-Buy-Stuff sử dụng cách tiếp cận Code-First.

Code-First cung cấp một lớp persistence centered kế thừa từ lớp DbContext. Đây là một ví dụ:

public class DomainModelFacade : DbContext

{

static DomainModelFacade()

{

Database.SetInitializer(new SampleAppInitializer());

}

public DomainModelFacade() : base(“naa4e-09”)

{

Products = base.Set<Product>();

Customers = base.Set<Customer>();

Orders = base.Set<Order>();

FidelityCards = base.Set<FidelityCard>();

}

public DbSet<Order> Orders { get; private set; }

public DbSet<Customer> Customers { get; private set; }

public DbSet<Product> Products { get; private set; }

public DbSet<FidelityCard> FidelityCards { get; private set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

...

}

}

Chuỗi được truyền đến constructor là tên của cơ sở dữ liệu hoặc tên của một mục nhập trong quá trình sắp xếp con trỏ chỉ ra nơi để đọc chi tiết về chuỗi kết nối và data provider. Khi bạn cài đặt Code First qua NuGet, bạn sẽ có được công cụ SQL Server làm nhà cung cấp dữ liệu mặc định và công cụ LocalDb để lưu trữ.

Các thuộc tính trừu tượng *DbSet* từ các bảng đang được tạo ra, và lớp khởi tạo có thể được sử dụng để tự động hóa việc tạo, dropping và filling cơ sở dữ liệu chủ yếu cho các mục đích của một thiết lập ban đầu hoặc để debugging.

**Mapping properties to columns**

Trong Code First, có hai cách để ánh xạ thuộc tính vào các cột. Bạn có thể sử dụng các thuộc tính dữ liệu “annotation” trong mã nguồn của các domain classes, hoặc bạn có thể sử dụng thông thạo Code-First API và viết các lớp configuration kết hợp với nhau trong phương thức Overridable OnModelCreating.

Nói chung, chúng tôi khuyên bạn nên sử dụng thông thạo API hơn là data annotations. Lý do là data annotations bị xâm nhập và làm thay đổi vào domain model được phân lập từ các công nghệ cơ sở hạ tầng.

Dưới đây là một số mã bạn muốn có trong mapping layer:

modelBuilder.ComplexType<Money>();

modelBuilder.ComplexType<Address>();

modelBuilder.ComplexType<CreditCard>();

modelBuilder.Configurations.Add(new FidelityCardMap());

modelBuilder.Configurations.Add(new OrderMap());

modelBuilder.Configurations.Add(new CustomerMap());

modelBuilder.Configurations.Add(new OrderItemMap());

modelBuilder.Configurations.Add(new CurrencyMap());

Phương pháp *ComplexType* cho phép bạn báo với O/RM rằng loại được chỉ định là một loại phức tạp trong thuật ngữ Entity Framework, đó là một khái niệm gần với giá trị của đối tượng trong domain model. Mã còn lại trong đoạn trích cho thấy trong mỗi mapping classes đang quan tâm tới sự sắp xếp của một thực thể. Lưu ý, tuy nhiên, trong trường hợp cần bạn cũng có thể có một mapping class cho một loại phức tạp. Tất cả phụ thuộc vào các hướng dẫn bạn có cho O/RM. Xem một vài ví dụ:

public class OrderMap : EntityTypeConfiguration<Order>

{

public OrderMap()

{

ToTable(“Orders”);

HasKey(t => t.OrderId);

HasRequired(o => o.Buyer);

HasMany(o => o.Items);

}

}

Phương thức *HasKey* khai báo khóa chính, trong khi *HasRequired* set một mối quan hệ khóa nước ngoài bắt buộc với một đối tượng *Customer* duy nhất. Cuối cùng, *HasMany* đặt một mối quan hệ một-nhiều đối với các đối tượng *OrderItem*.

Mặt khác, phần trích dẫn sau đây cho thấy làm thế nào để cấu hình cột từ thuộc tính:

Property(o => o.OrderId)

.IsRequired()

.HasMaxLength(10)

.HasColumnName(“Id”);

Hiệu ứng tiếp theo là bảng *Orders* sẽ có một cột có tên *Id* được ánh xạ tới thuộc tính *OrderId*. Ở mức cơ sở dữ liệu, cột không chấp nhận các giá trị null và bất kỳ nội dung nào dài hơn 10 ký tự. Các phương pháp tương tự tồn tại để tạo ra một giá trị tự động tạo bởi cơ sở dữ liệu:

Ignore(p => p.Name);

Cuối cùng, phương thức *Ignore* được sử dụng để nói cho O/RM rằng tài sản quy định không nên tiếp tục tồn tại. Đây là những gì xảy ra, ví dụ, khi tài sản có một getter tính trong mã C#.

**Implementing the business logic**

Vì nó thường xảy ra, không phải tất cả logic nghiệp vụ được yêu cầu - cho dù đó là quy tắc hoặc nhiệm vụ - nó vào các lớp học của domain model. Tối thiểu, bạn cần phải có logic tồn tại được lưu trữ trong các lớp kho. Rất có thể, bạn cần các domain services. Trong ví dụ I-Buy-Stuff, có hai nhiệm vụ chính-tìm một đơn đặt hàng và đặt hàng.

**Note**: Trước khi chúng tôi nghiên cứu sâu vào domain services, hãy làm rõ luồng gọi từ giao diện người dùng web xuống domain layer. Như đã đề cập, ví dụ của chúng tôi là một ứng dụng ASP.NET MVC. Điều này có nghĩa là bất kỳ hành động nào của người dùng (ví dụ như cách nhấp vào nút) kết thúc bằng cách gọi phương thức trên lớp điều khiển. Trong chương 6, "Lớp hiển thị", chúng tôi giới thiệu các application services để chứa bất kỳ logic điều khiển nào nhận được nội dung từ đầu vào HTTP và trả về bất cứ điều gì cần thiết cho bộ điều khiển để tạo ra phản hồi HTTP. Tuy nhiên, dịch vụ ứng dụng không trực tiếp xử lý các đối tượng trong ngữ cảnh HTTP.

**Finding an order**

Đây là mã bạn tìm thấy trong phương thức điều khiển nhận được yêu cầu của người dùng để lấy ra một đơn hang nhất định. Như bạn thấy, phương thức controller mang lại cho một application service lấy dữ liệu từ yêu cầu HTTP và tạo ra một phản hồi đã sẵn sàng để truyền trở lại dưới dạng HTML, JSON hoặc bất cứ cái gì phù hợp:

public ActionResult SearchResults(int id)

{

var model = \_service.FindOrder(id, User.Identity.Name);

return View(model);

}

Phương thức application service - *FindOrder*, trong các domain services và các kho lưu trữ thích hợp, có bất kỳ sự thích ứng dữ liệu nào có thể cần thiết để gọi vào domain layer và trả về một view model.

Một cách khác để tổ chức mã này có thể là chia tách nhiệm vụ giữa application service và controller để application service chỉ thực hiện dàn xếp và trả về dữ liệu thô mà bộ điều khiển gói lên trong view model.

**Important** Note: lớp view model là một vùng chứa dữ liệu có thể đến dưới dạng các domain entities hoặc các đối tượng truyền dữ liệu.

Dưới đây là thực hiện thực tế của phương thức application service. Phải thừa nhận, trong trường hợp cụ thể này có một application service có lẽ là quá mức cần thiết bởi vì tất cả những gì nó làm là gọi một phương thức lưu trữ. Tuy nhiên, chúng tôi mời bạn xem xét mẫu đầu tiên cơ bản và sau đó áp dụng bất kỳ sự đơn giản nào có thể có hiệu quả. Con đường ngắn nhất luôn luôn phù hợp miễn là bạn biết bạn đang làm gì và bạn muốn đi đâu! Nói chung, tuy nhiên, authorization code và thậm chí xác nhận ứng dụng có thể là mã bổ sung cho các application services.

public SearchOrderViewModel FindOrder(int orderId, string customerId)

{

var order = \_orderRepository.FindByCustomerAndId(orderId, customerId);

if (order is NullOrder)

return new SearchOrderViewModel();

return SearchOrderViewModel.CreateFromOrder(order);

}

**Note**: Chúng tôi đã đề cập đến kho lưu trữ trong chương trước và chúng tôi đề cập đến các kho chứa rộng trong kho này. Chúng tôi cung cấp thêm chi tiết về vai trò và sự thực hiện của họ trong Chương 14, “The infrastructure layer.”

**Placing an order**

Khi người dùng nhấp chuột để bắt đầu mua sắm trên trang web, hệ thống phục vụ cho giao diện người dùng thể hiện trong hình 9-8. Đoạn mã sau điều khiển quá trình:

public ActionResult New()

{

var customerId = User.Identity.Name;

var shoppingCartModel = \_service.CreateShoppingCartForCustomer(customerId);

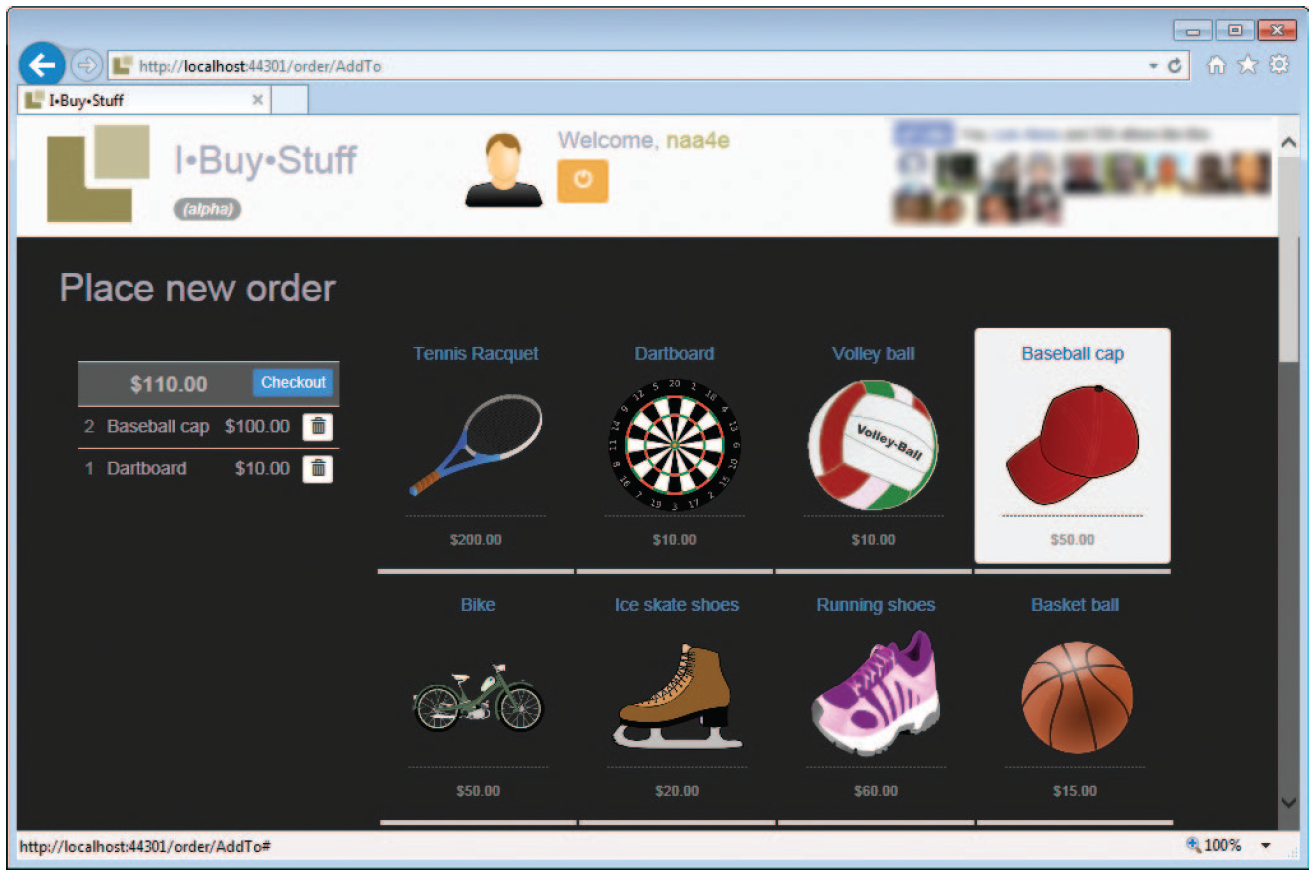
shoppingCartModel.EnableEditOnShoppingCart = true;

SaveCurrentShoppingCart(shoppingCartModel);

return View(“shoppingcart”, shoppingCartModel);

}

Application service giúp tạo ra giỏ hàng trống cho người dùng và cho phép điều khiển giao diện tương tác để thêm hoặc xóa các sản phẩm. Giỏ hàng được lưu trong trạng thái phiên để tham khảo thêm. Lưu ý rằng loại *ShoppingCart* thực tế được tạo ra trong miền bởi vì nó mang thông tin tạo ra yêu cầu đặt hàng. Tuy nhiên, trên lớp giao diện, *ShoppingCart* được gói trong một view-model được bổ sung với các thuộc tính liên quan đến UI yêu cầu để hiển thị trang HTML. Điều này cũng bao gồm danh sách đầy đủ các sản phẩm mà người dùng có thể lựa chọn.



**FIGURE 9-8** The shopping cart for the current user.

Trang mua sắm cho phép người dùng thêm và xóa các sản phẩm khỏi giỏ hàng. Mỗi hành động bổ sung / loại bỏ tạo ra một roundtrip; trạng thái của giỏ hàng được lấy ra từ session state (trạng thái phiên), cập nhật và sau đó được lưu lại.

public ActionResult AddToShoppingCartCommand(int productId, int quantity=1)

{

var cartModel = RetrieveCurrentShoppingCart();

cartModel = \_service.AddProductToShoppingCart(cart, productId, quantity);

SaveCurrentShoppingCart(cartModel);

return RedirectToAction(“AddTo”);

}

Application service thêm một phần tử vào giỏ hàng:

public ShoppingCartViewModel AddProductToShoppingCart(

ShoppingCartViewModel cart, int productId, int quantity)

{

var product = (from p in cart.Products where p.Id == productId select p).Single();

cart.OrderRequest.AddItem(quantity, product);

return cart;

}

Phương thức *AddItem* đối với domain object *ShoppingCart* chứa logic để tăng số lượng nếu sản phẩm đã có trong giỏ hàng:

public ShoppingCart AddItem(int quantity, Product product)

{

var existingItem = (from i in Items

where i.Product.Id == product.Id

select i).SingleOrDefault();

if (existingItem != null)

{

existingItem.Quantity++;

return this;

}

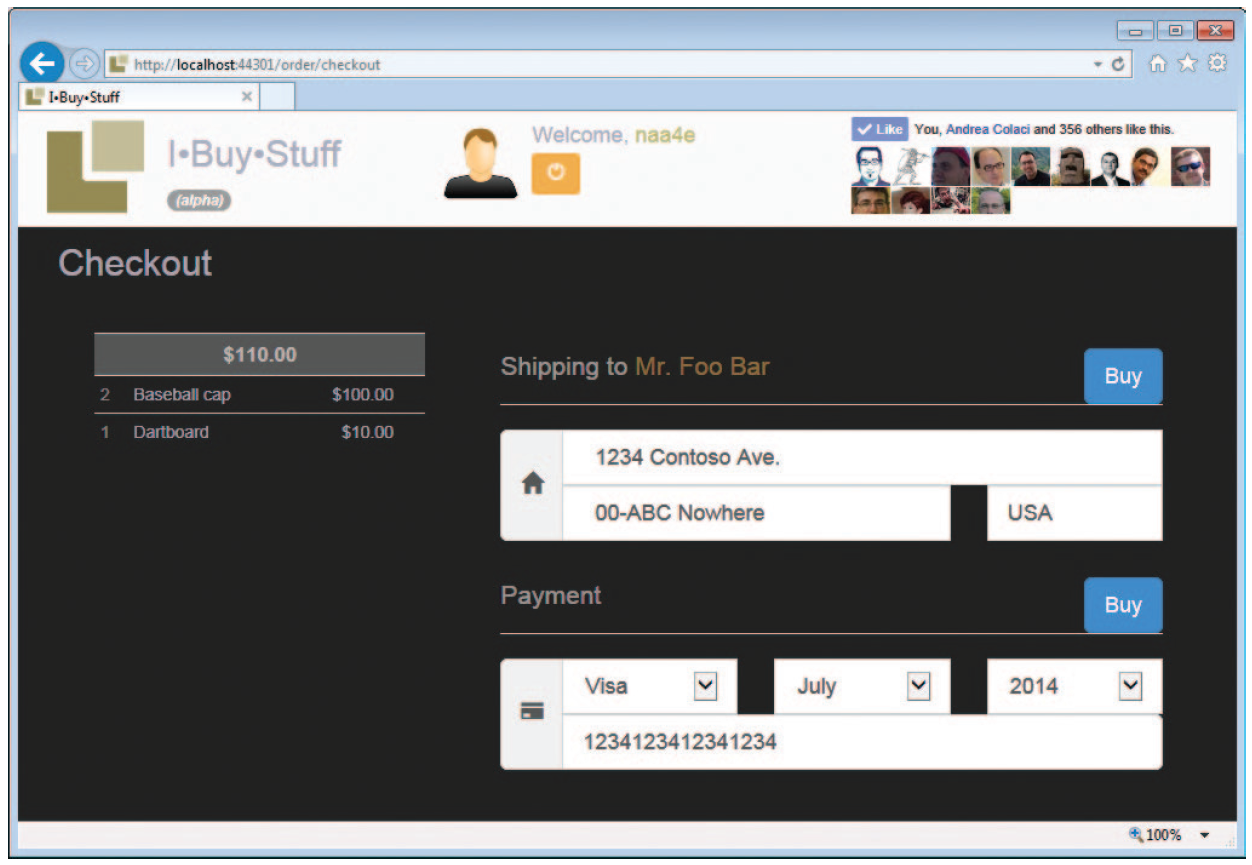
// Create new item

Items.Add(ShoppingCartItem.Create(quantity, product));

return this;

}

Nút thanh toán dẫn đến trang trong Hình 9-9. Lưu ý rằng giỏ hàng hiện được hiển thị ở chế độ đọc và tất cả các nút có thể thao tác không được hiển thị.



**FIGURE 9-9** The checkout page of the I-Buy-Stuff site.

Lưu ý rằng phương thức controller nhận lệnh để thêm một mục vào giỏ hàng (hoặc xóa một mục khỏi nó) cuối cùng chuyển hướng sang hành động *AddTo* thay vì chỉ hiển thị chế độ xem tiếp theo. Thao tác này được thực hiện để tránh gửi biểu mẫu lặp lại nếu người dùng làm mới trang. Để đáp lại hành động làm mới (ví dụ: khi F5 được nhấn), trình duyệt chỉ cần lặp lại hành động cuối cùng. Nhờ chuyển hướng, tuy nhiên, hành động cuối cùng là GET chứ không phải POST.

**Processing the order request**

Khi người dùng nhấp vào nút Buy, đã đến lúc để xử lý và tạo ra đơn đặt hàng. Vào thời điểm này, các chính sách của công ty và các quy tắc kinh doanh phải được áp dụng. Nhìn chung, việc xử lý thứ tự dựa trên một workflow được dàn xếp bởi application service được gọi từ controller. Các bước điển hình của workflow đang kiểm tra tính sẵn có của hàng đặt hàng, xử lý thanh toán, chuyển tiếp chi tiết vận chuyển, và lưu trữ dữ liệu lệnh trong cơ sở dữ liệu.

Hầu hết các hành động này có thể được thực hiện ở silent mode. Tuy nhiên, thanh toán có thể yêu cầu trang bên ngoài được hiển thị để cho phép người dùng tương tác với banking back end. Nếu thanh toán yêu cầu giao diện người dùng riêng, bạn sẽ chia tách việc xử lý đơn hàng thành hai giai đoạn: trước và sau khi thanh toán. Nếu không, bạn có thể thực hiện việc xử lý thanh toán và đồng bộ hóa với công ty vận chuyển như là hai tác vụ song song sử dụng .NET Framework Parallel API. Dưới đây là cách chia kiểm ra thành hai bước:

public ActionResult Checkout(CheckoutInputModel checkout)

{

// Pre-payment steps

var cart = RetrieveCurrentShoppingCart();

var response = \_service.ProcessOrderBeforePayment(cart, checkout);

if (!response.Denied)

return Redirect(Url.Content(“~/fake\_payment.aspx?returnUrl=/order/endcheckout”));

TempData[“ibuy-stuff:denied”] = response;

return RedirectToAction(“Denied”);

}

public ActionResult EndCheckout(string transactionId)

{

// Post-payment steps

var cart = RetrieveCurrentShoppingCart();

var response = \_service.ProcessOrderAfterPayment(cart, transactionId);

var action = response.Denied ? “denied” : “processed”;

return View(action, response);

}

Phương thức *ProcessOrderBeforePayment* trên application service lưu thông tin thanh toán (địa chỉ giao hàng và chi tiết thanh toán) và kiểm tra stock level của các sản phẩm được đặt hàng. Tùy thuộc vào các chính sách được kích hoạt, nó cũng cần phải đặt một yêu cầu điền lại để đưa stock level trở lại một giá trị an toàn. Nếu xảy ra sự cố (ví dụ: một số hàng hoá không có sẵn), phản hồi được lưu trữ trong *TempData* được hiển thị bởi trang Denied trên trang chuyển hướng.

**Note**: *TempData* là một cơ sở ASP.NET MVC được tạo ra để duy trì dữ liệu yêu cầu tạm thời qua chuyển hướng. Tính năng này tồn tại để kích hoạt mô hình Post-Redirect-Get để bảo vệ chống lại những ảnh hưởng xấu của việc làm mới trang.

Nếu kiểm tra trước thanh toán tốt, phương thức sẽ chuyển hướng đến trang thanh toán. Đây là nơi thanh toán xảy ra. Khi hoàn tất, trang sẽ chuyển hướng lại đến URL được chỉ rõ. Khi làm như vậy, trang thanh toán sẽ vượt qua ID giao dịch thể hiện thanh toán. (Đây là mô tả chung về cách hầu hết các API thanh toán thực sự hoạt động.)

Giai đoạn sau thanh toán có thể bao gồm đặt chỗ giao hàng thông qua shipping company back end, đăng ký đơn đặt hàng trong hệ thống, cập nhật stock level và cập nhật hồ sơ của idelity card.

**Fidelity card (or customer loyalty program)**

Một khi lệnh đã có trong hệ thống, có thể một số tasks bổ sung vẫn được yêu cầu. Theo cách nào đó, việc thêm đơn hàng tạo sự kiện trong miền có thể yêu cầu một hoặc nhiều xử lý. Điều này giới thiệu điểm của *domain events*. Nói chung, *domain events* là sự kiện có thể được đưa ra trong domain model. Domain events đơn giản là một cách để kiến trúc sư dọn dẹp thiết kế và cho phép mình xử lý các tình huống một cách dễ hiểu hơn.

Cụ thể, các yêu cầu chúng tôi có đối với I-Buy-Stuff cho biết rằng bất cứ khi nào một lệnh được tạo ra, hệ thống phải cập nhật stock level của sản phẩm và thêm điểm vào fidelity card của khách hàng.

Cả hai hoạt động có thể được mã hoá như các phương thức được thực hiện vào cuối phương thức *ProcessOrderAfterPayment* trong lớp application service. Nó rõ ràng và đơn giản, và nó chỉ hoạt động.

Tuy nhiên, logic đằng sau fidelity card có thể thay đổi khi các chiến dịch tiếp thị được đưa ra, kết thúc, hoặc điều chỉnh. Việc thực hiện nói trên có thể trở nên quá cứng nhắc về mặt bảo trì và dẫn đến việc cập nhật thường xuyên cho mã nhị phân. Thiết kế mở rộng hơn những gì có thể xảy ra khi đặt hàng có thể làm công việc đòi hỏi trong thời gian dài để giữ cho hệ thống phù hợp với nhu cầu kinh doanh.

Domain events cố gắng giải quyết tình huống này. Một domain event có thể được thực hiện như là một sự kiện C# đơn giản trong một số lớp tổng hợp hoặc nó có thể được thực hiện thông qua một loại cơ chế xuất bản / đăng ký, nơi thông tin được đặt trên BUS và các bộ xử lý đã đăng ký nhận nó và xử lý nó một cách thích hợp. Quan trọng hơn, các trình điều khiển có thể được nạp và đăng ký sử dụng một giao diện dependency-injection để thêm và gỡ bỏ một trình điều khiển dễ dàng và nhẹ như chỉnh sửa các configuration file của ứng dụng.

**Summary**

Chương này cung cấp các khái niệm và ý tưởng ít được kết nối trong không gian của các hướng dẫn cơ bản hoặc lý thuyết. Chúng tôi không chỉ trình bày domain model mà các lớp học có hành vi và factories và sử dụng hạn chế các loại nguyên thủy. Chúng tôi thực sự trình bày một ứng dụng ASP.NET MVC nhiều lớp tiến hành từ đầu đến cuối từ khi tạo ra một trật tự trong một kịch bản không phải là một CRUD không thực tế.

Trong quá trình thực hiện, chúng ta đã chuyển các khái niệm và thực hành đã được giới thiệu ở các chương trước, mặc dù cấu trúc cơ sở dữ liệu không phải là mối quan tâm hàng đầu khi xây dựng mô hình nhưng nó vẫn là một ràng buộc thường cần thiết để làm cho mô hình đối tượng bền vững.

Chúng tôi mời bạn xem toàn bộ mã nguồn của ứng dụng I-Buy-Stuff. Văn bản trong chương nhấn mạnh các lựa chọn xung quanh domain model; Tuy nhiên, mã chứa nhiều chi tiết và giải pháp có thể rất thú vị để xem.

**Finishing with a smile**

StackOverlow.com, TopEdge.com, and other websites contain an endless collection of jokes about developers. In particular, we looked for jokes about developers and light bulbs. Here’s our selection:

■ How many developers are needed to change a light bulb? None. The light bulb works just ine in *their* ofices.

■ How many developers are needed to change a light bulb? None. It’s a hardware problem.

■ How many testers are needed to change a light bulb? They can only assert the room is dark, but they can’t actually ix the problem.

■ How many Java developers are needed to change a light bulb? Well, the question is ill-posed and shows you’re still thinking procedurally. A properly designed light bulb object would inherit a *change* method from a base light bulb class. Therefore, all you have to do is send a light bulb change message.

■ How many Prolog developers does it take to change a light bulb? Yes.