**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:

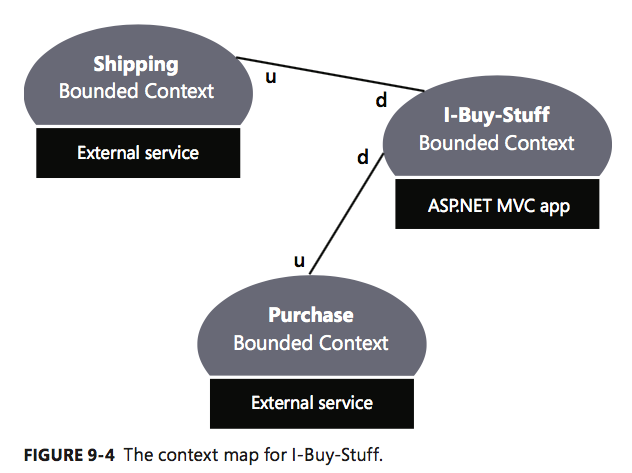
* Sử dụng một giới hạn ngữ cảnh lớn.
* Tách thành từng miền nhỏ, với dữ liệu trùng lặp, và có thể tương tác thông qua một thế thống publish/subscribe.
* Cô lập một vài giới hạn ngữ cảnh lớn hơn và phải xác thực chúng thông qua các dịch vụ web.

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

...

}

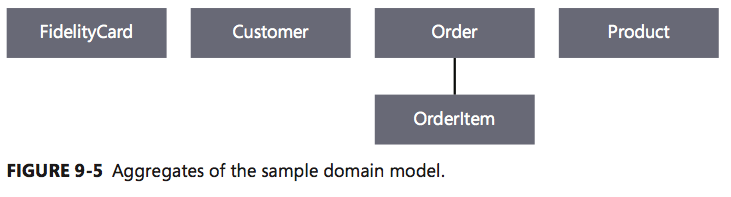
Đơn vị tiền tệ trong ví dụ này là một đối tượng giá trị được xây dựng xung quanh biểu tượng và tên.

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**

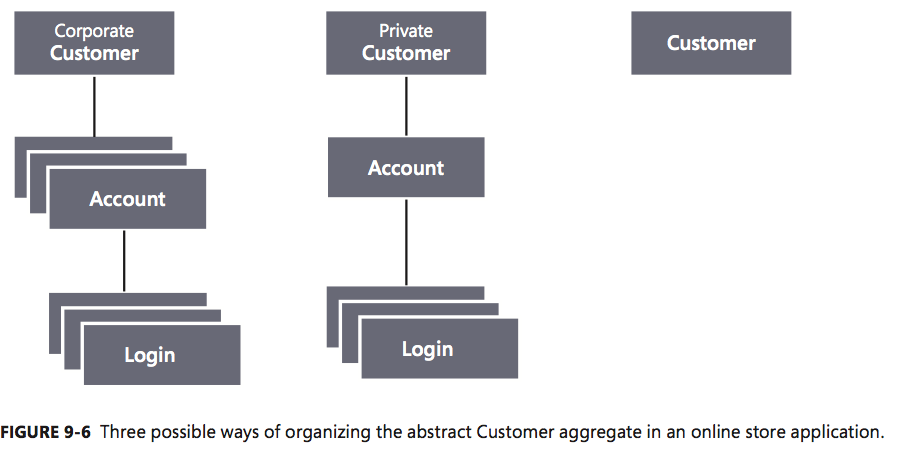
Bây giờ hãy tập trung vào các thực thể được liệt kê trong Bảng 9-2 và được sử dụng trong ứng dụng mẫu. Ứng dụng I-Buy-Stuff có nghĩa là một cửa hàng trực tuyến đơn giản, và điều này được phản ánh bởi số lượng hạn chế của các cơ quan và mối quan hệ hạn chế của chúng. (Xem Hình 9-5). Trong một trang web 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.

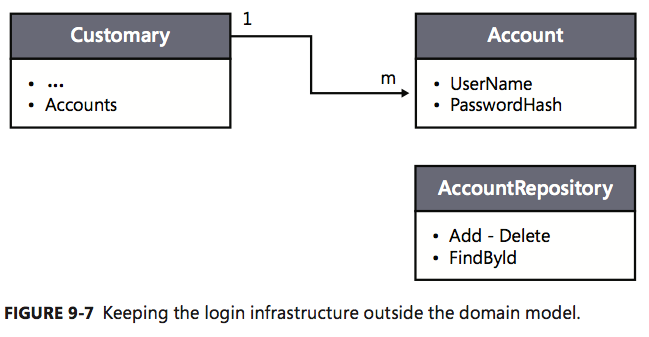
...

}

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

... }