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);

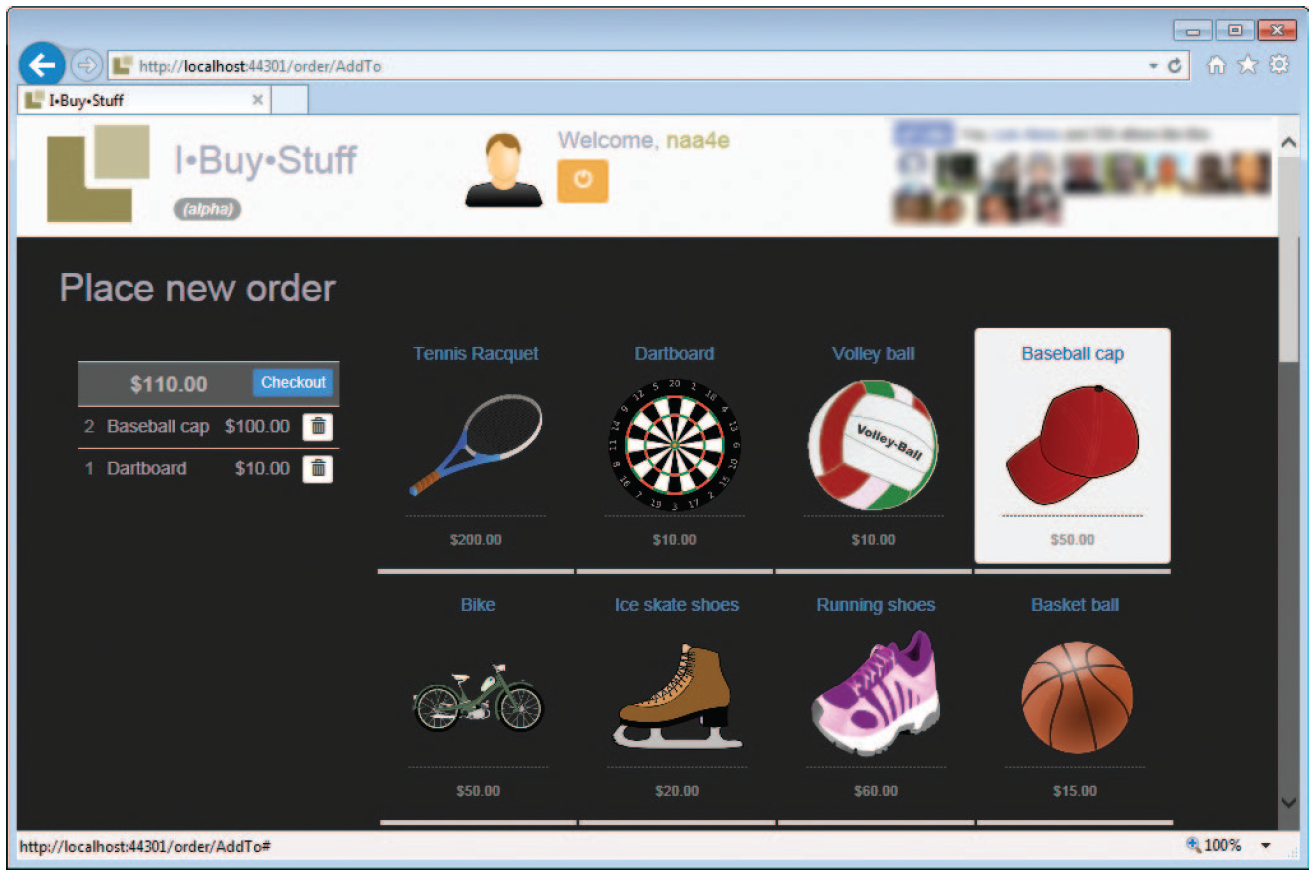
shoppingCartModel.EnableEditOnShoppingCart = true;

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;

}

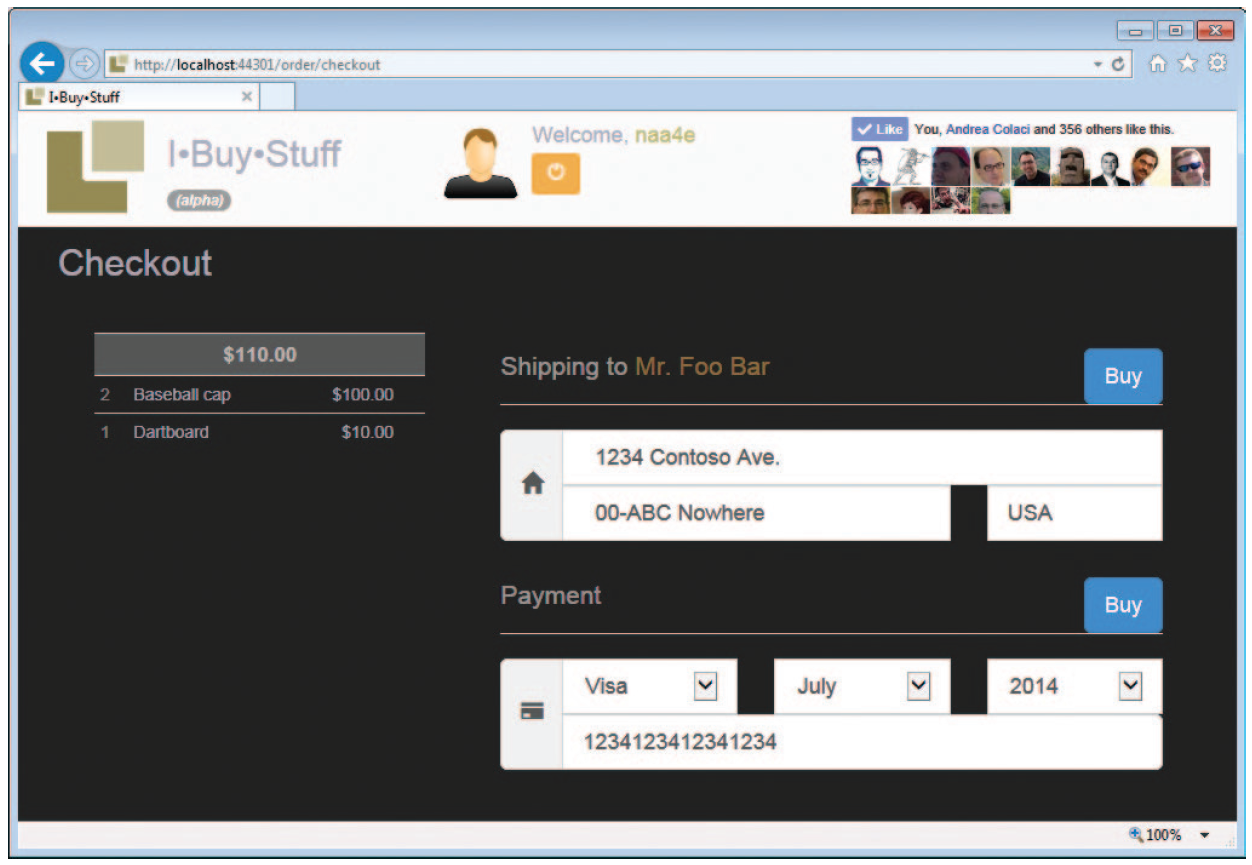
// Create new item

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.