239-254

Let’s consider the *StockLevel* property. At irst, the integer used to deine the stock level can be changed into an unsigned integer because negative values are not expected to be valid values for what the property represents. Even better, an ad hoc value object can be created to represent a quantity as a natural number.

As you can see, though, the *StockLevel*, like all properties in the code snippet, has a private setter. This means that the property is read-only and, subsequently, reading it as an integer is ine. Who’s going to set the *StockLevel* property? There are two possible scenarios:

■ The object is materialized only from the persistence layer.

■ The object has a factory method if it’s allowed to be instantiated from client code.

Actually, the second option is not credible businesswise, though a factory method might be helpful to have for implementation purposes.

The *StockLevel* property indicates in this simple model the number of items of a given product we have in store. This number is decreased as orders are placed and increased when the store manager reills the stock by placing orders to vendors. Subsequently, a method is necessary to programmatically modify the stock level of a product. The store manager is part of the back ofice, which is a different bounded context.

The *Order* aggregate

In I-Buy-Stuff, an order is created when processing the content of a shopping cart, which is represented by an *ShoppingCart* class. The skeleton of the class is shown here:

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; }

...

}

The class needs to have a public factory method because at some point business logic needs to create an order while processing a request. When creating an order, you specify the ID and reference the customer. The date is set automatically, so it is the state of the order.

In the example, we assume that the order is created when the customer clicks to check out. There will likely be a domain service to orchestrate the checkout process; one of the steps is the creation of the order. The factory of the order can be marked internal and can use a nonpublic constructor:

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;

}

To modify the state of the order and add items, you need ad hoc methods. Here are some methods that change the state:

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;

}

As you can see, it’s not rocket science, but using these methods instead of plain setters improves the readability of the code and, more importantly, keeps it aligned to the ubiquitous language.

The conventions used here to name methods, in fact, cannot be arbitrary. Terms like *Archive* and *MarkAsShipped* should be taken from the ubiquitous language and relect the terminology used within the business.

Note: Such methods are often implemented as void methods. This makes total sense because they just represent actions. However, depending on the context it might be interesting to consider a luent interface. You just make the method return *this*, and it enables chaining calls and makes the whole thing more readable. It’s a possibility to consider.

There are a few interesting things to say about the *Items* and *Total* properties. Let’s start with bringing up some issues related to having collections in an aggregate. We describe the issues and provide some suggestions. The deinitive answer, though, will come in the next chapter and from the use of a different DDD supporting architecture—CQRS instead of Domain Model.

The *Items* property is deined as a collection. This means that any code that receives an *Order* instance can enumerate the content of the collection but can’t add or remove items. Unfortunately, there’s no easy way to prevent that client code from accessing and updating content within the collection. Even worse, this aspect remains unaltered even if you further restrict the collection to a basic

*IEnumerable<T>*. The following code is always allowed and, in addition, you can always use LINQ to select a particular order within the collection:

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

In our scenario, an order is read-only. It gets created out of the checkout process; it can’t be modiied, but it can be viewed. So you want the *Items* collection to be consumed as read-only content. How can you prevent changes on the objects within the collection? Unfortunately, even the following won’t work:

public class Order

{

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

public ICollection<OrderItem> Items

{

get { return \_items.AsReadOnly(); }

}

}

The *Items* collection is read-only, but you can still retrieve individual elements and change them programmatically.

The deinitive solution to this problem comes with the separation between queries and commands and with the provision of different models for reading and writing operations. We’ll cover this in the next two chapters.

The *Total* property indicates the total value of the order. How should you deal with that? The property is there to please code that consumes the content of the order. In theory, the total of an order can be calculated on demand, iterating on the order items graph with an ad hoc method:

public Money GetTotal()

{

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

return new Money(Currency.Default, amount);

}

This solution, though, has a signiicant drawback. It forces the *Order* instance to have in memory the entire graph of items. There might be scenarios in which you just want to know the about the total of the order, without all the details. A method like *GetTotal* will force the deserialization of the entire graph and create much more trafic to and from the database.

By having a simple property on the *Order*, you can decide intelligently whether or not to deserialize the entire graph. The *Total* property can be set by the order repository when it returns an instance (which could be a simple SUM operation at the SQL level), or it can be the result of some redundancy in the relational schema of the database. Whenever the order is persisted, the current total is calculated and saved for further queries.

The *FidelityCard* aggregate

The *FidelityCard* aggregate is an extremely simple class that summarizes through the accrual of points the activity of a customer within the site.

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;

}

}

Each order a customer places increases the total points on the card, and those points are then used to determine a reward status the customer can use to get additional discounts. All the logic of discounts and levels is inspired by the business domain and can be as complex and varying as the real world requires.

The content of a *FidelityCard* object is used by domain services to calculate reward levels and discounts.

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*. 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:

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();

}

}

}

The *First* method throws if the order is not found. In this case, the code returns a newly created instance of the class *NotFoundOrder*:

public class NotFoundOrder : Order

{

public static NotFoundOrder Instance = new NotFoundOrder();

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

{

}

}

*NotFoundOrder* is just a derived class that sets all properties to their default values. Any code that expects an *Order* can deal with *NotFoundOrder* as well; and type checking helps you igure out if something went wrong:

if(order is NotFoundOrder)

{

...

}

This is the gist of the *Special Case* pattern. On top of this basic implementation, you can add as many additional features as you want, including a singleton instance.

Persisting the model

A domain model exists to be persisted, and typically an O/RM will do the job. All that an O/RM does as far as persistence is concerned is map properties to columns of a database table and manage reads and writes. This is only a 10,000-foot, bird’s-eye view, though.

What an O/RM does for you

Generally speaking, an O/RM has responsibilities that can be summarized in four points:

■ CRUD

■ Query engine

■Transactional engine

■ Concurrency

The query and transactional engines refer to two speciic design patterns: the Query Object pattern (which you can see at *http://martinfowler.com/eaaCatalog/queryObject.html*), and the Unit of Work pattern (which you can see at *http://martinfowler.com/eaaCatalog/unitOfWork.html*).

Today, on the .NET platform nearly all O/RMs offer an idiomatic implementation of the Query Object pattern based on the LINQ syntax. The *Unit of Work* is offered through the capabilities of the O/RM root object. In Entity Framework, this object is *ObjectContext* in its various lavors, such as

*DbContext*.

An O/RM is an excellent productivity tool. It doesn’t really do magical things, but it saves you a lot of cumbersome and error-prone coding. All that it requires is instructions on how to map properties of objects in the domain model to columns of relational database tables.

When it comes to this, you realize that the database is really important even if you do domaindriven design and build a model that is persistence ignorant and stay as agnostic as possible with regard to databases. In a nutshell, the database and the O/RM are two constraints that typically force you to make concessions and introduce in the model features that only serve the need of persistence.

Making concessions to persistence

The most common concession that you, as a Domain Model architect, have to make to an O/RM is the availability of default constructors on all persistent classes.

protected Customer()

{

// Place here any initialization code you may need

...

}

The constructor is required for the O/RM to materialize an entity from the database. In other words, it still needs low-level tools to create an instance of that class. Factory methods are an abstraction that serves the purpose of the ubiquitous language. The constructor, on the other hand, is the only known way that compilers in C# (and other object-oriented languages) allow you to create fresh instances of a class.

Nicely enough, though, you can use O/RM tools to hide the default parameterless constructor

from public view. When materializing an entity from databases, an O/RM usually returns the instance of a dynamically created class that inherits from the real entity. In this way, the dynamically generated code gains access to the protected constructor.

Beyond this point, other changes you might be forced to do to make the model persistent depend on the capabilities and quirks of the O/RM of choice. For example, up until version 5, Entity Framework was unable to deal with *enum* types. In version 6.1, it is still unable to handle, at least in a default way, arrays of primitive types.

Note: When it comes to arrays, database experts might tell you that arrays do not it nicely in the world of databases and that you should ideally ind a solution that doesn’t require arrays. The point is that relational databases don’t offer a way to read and write arrays directly. As a result, storing arrays requires workarounds, but it is neither impossible nor particularly hard. Yet, when you think about the model in a database-agnostic way, arrays might be excellent modeling tools for business aspects that require sequences of related data.

All this is to say that irst you should strive to get an ideal domain model that matches the features of the ubiquitous language; next, you should strive to make it suitable to the O/RM of choice (notably Entity Framework) to persist it.

Note: Entity Framework is certainly not the only O/RM available for the .NET Framework. A popular competitor is NHibernate. The perception, however, is that the ierce debate about which of the two to use no longer matters. And because it is tightly integrated with Visual Studio, Entity Framework is the irst option that many consider. In addition, there’s a long list of products available from vendors such as Devxpress and Telerik. It’s really mostly a matter of preference.

The Entity Framework Code-First approach

Entity Framework is the O/RM provided by Microsoft .NET. It was not even released when we wrote the irst edition of this book. It can be considered the most natural choice today for any .NET project—nearly a de facto standard.

Essentially, Entity Framework comes in two lavors: Database First and Code First. There’s also a third lavor, called Model First, but it’s a hybrid, middle-way approach that uses the visual diagram model and was soon superseded by Code First. The Database-First approach reads the structure of an existing database and returns a set of anemic classes you can extend with methods using the partial class mechanism. The Code-First approach consists of writing a set of classes—for example, a Domain Model, as discussed so far. Next, you add an extra layer of code (or data annotations) to map properties to tables. This mapping layer tells the O/RM about the database to create (or expect), its

relationships, its constraints and tables and, more importantly, it tells the O/RM where to save or read property values.

The I-Buy-Stuff example uses the Code-First approach.

Code-First delivers a persistence layer centered on a class that inherits from *DbContext*. Here’s an example:

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)

{

...

}

}

The string passed to the constructor is the name of the database or the name of an entry in the coniguration ile indicating where to read details about the connection string and the data provider. When you install Code First via NuGet, you get the SQL Server engine as the default data provider and the LocalDb engine for storage.

The *DbSet* properties abstract the tables being created, and the initializer class can be used to automate the creation, dropping, and illing of the database mostly for the purposes of an initial setup or for debugging.

Mapping properties to columns

In Code First, there are two ways to map properties to columns. You can use data annotation attributes in the source code of domain classes, or you can use the luent Code-First API and write coniguration classes bound together in the *OnModelCreating* overridable method.

Overall, we recommend the luent API rather than data annotations. The reason is that data annotations are intrusive and add noise to an otherwise persistent ignorant domain model isolated from infrastructure technologies.

Here’s some code you want to have in the 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());

The *ComplexType* method lets you tell the O/RM that the speciied type is a complex type in the Entity Framework jargon, which is a concept close to what a value object is in a domain model. The remaining code in the snippet shows mapping classes each taking care of the coniguration of an entity. Note, though, that in case of need you also can have a mapping class for a complex type. It all

depends on the instructions you have for the O/RM. Let’s look at a couple of examples:

public class OrderMap : EntityTypeConfiguration<Order>

{

public OrderMap()

{

ToTable(“Orders”);

HasKey(t => t.OrderId);

HasRequired(o => o.Buyer);

HasMany(o => o.Items);

}

}

The method *HasKey* declares the primary key, whereas *HasRequired* sets a required foreign-key relationship to a single *Customer* object. Finally, *HasMany* deines a one-to-many relationship for *OrderItem* objects.

The following excerpt, on the other hand, shows how to conigure columns from properties:

Property(o => o.OrderId)

.IsRequired()

.HasMaxLength(10)

.HasColumnName(“Id”);

The net effect is that the table *Orders* is going to have a column named *Id* mapped to the property *OrderId*. At the database level, the column doesn’t accept null values and any content longer than 10 characters. Similar methods exist to make a value auto-generated by the database:

Ignore(p => p.Name);

Finally, the *Ignore* method is used to tell the O/RM that the speciied property shouldn’t be persisted. This is what happens, for example, when the property has a computed getter in the C# code.

**Implementing the business logic**

As it often happens, not all the business logic that is required—whether it is rules or tasks—its into the classes of the domain model. At a minimum, you need to have persistence logic stored in repository classes. Most likely, you need domain services. In the I-Buy-Stuff example, there are two main tasks—inding an order and placing an order.

Note: Before we delve deeper into domain services, let’s clarify how the call moves from the web user interface down to the domain layer. As mentioned, our example is an ASP.NET MVC application. This means that any user action (for example, button clicking) ends up in an action method call on a controller class. In Chapter 6, “The presentation layer,” we introduced application services to contain any orchestration logic that gets content from the raw HTTP input and returns whatever is required for the controller to produce an

HTTP response. Application services, though, don’t directly deal with objects in the HTTP

context.

Finding an order

Here’s the code you ind in the controller method that receives the user’s request to retrieve a given order. As you can see, the controller method yields to an application service that gets data from the HTTP request and produces a response ready to be transmitted back as HTML, JSON, or whatever else

is suitable:

public ActionResult SearchResults(int id)

{

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

return View(model);

}

The application service method—*FindOrder*, in the example—calls domain services and repositories as appropriate, does any data adaptation that might be necessary to call into the domain layer, and returns a view model.

Another option for organizing this code might be splitting responsibilities between the application service and controller so that the application service just does orchestration and returns raw data that the controller then packages up in a view model.

**Important** Note: that the view model class is a plain container of data that might come in the form of domain entities or data-transfer objects.

Here’s the actual implementation of the application service method. Admittedly, in this particular case having an application service is probably overkill because all it does is call a repository method. However, we invite you to consider the underlying pattern irst and then apply any simpliication that might work. The shortest path is always preferable as long as you know what you’re doing and where you really want to go! In general, however, authorization code and even app validations might be additional code for the 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: We mentioned repositories in the prior chapter, and we’re mentioning repositories extensively in this one. We provide more details about their role and implementation in Chapter 14, “The infrastructure layer.”

Placing an order

When the user clicks to start shopping on the site, the system serves up the user interface shown in Figure 9-8. The following code governs the process:

public ActionResult New()

{

var customerId = User.Identity.Name;

var shoppingCartModel = \_service.CreateShoppingCartForCustomer(customerId);

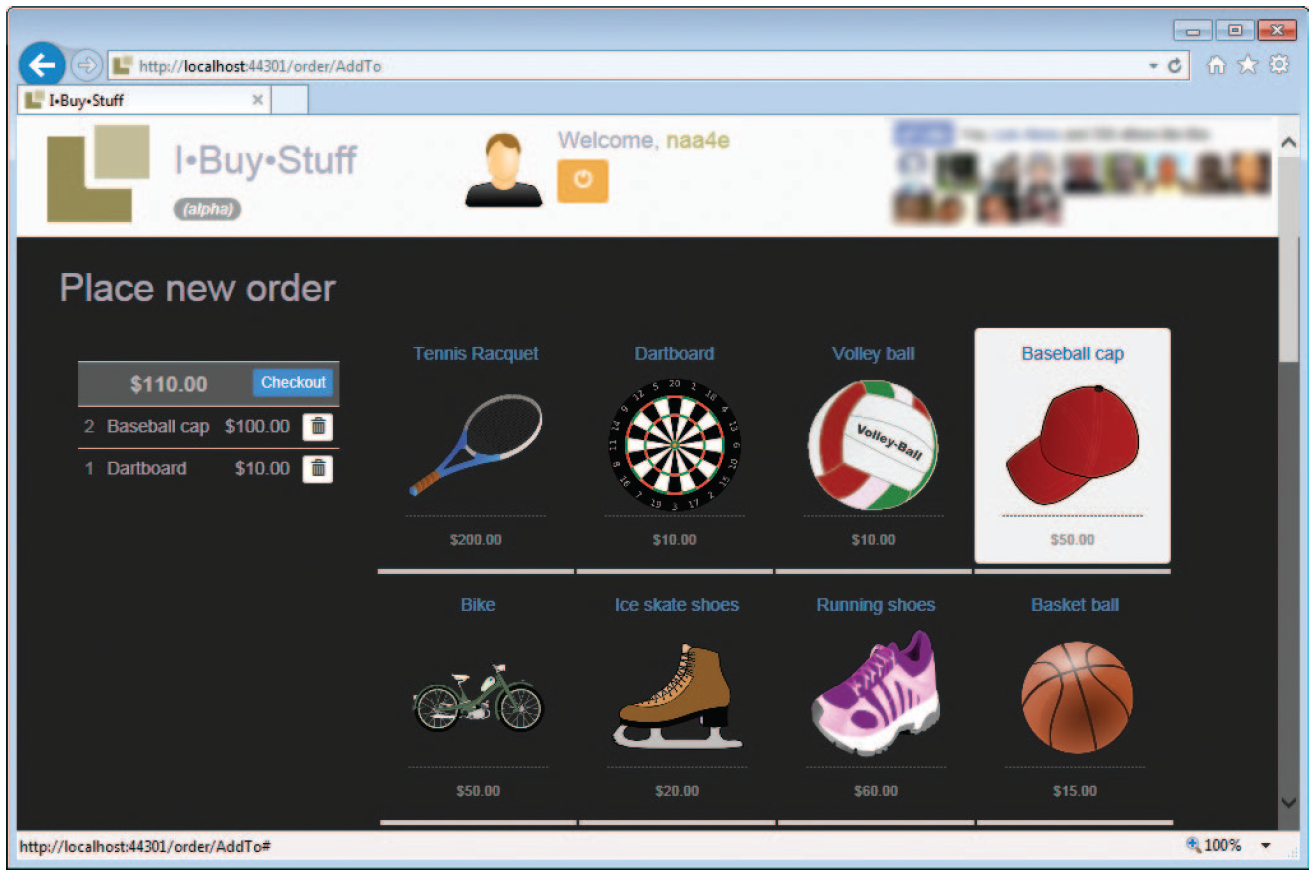
shoppingCartModel.EnableEditOnShoppingCart = true;

SaveCurrentShoppingCart(shoppingCartModel);

return View(“shoppingcart”, shoppingCartModel);

}

The helper application service creates an empty shopping cart for the user and enables interactive UI controls to add or remove items. The shopping cart is saved in the session state for further reference. Note that the actual *ShoppingCart* type is deined in the domain because it carries information that makes up an order request. On the presentation layer, though, the *ShoppingCart* is wrapped in a view-model type enriched with the UI-related properties required to render the HTML page. This also includes the full list of products the user can choose from.



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

The shopping page lets the user add and remove products from the cart. Every add/remove action generates a roundtrip; the state of the shopping cart is retrieved from the session state, updated, and then saved back.

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

{

var cartModel = RetrieveCurrentShoppingCart();

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

SaveCurrentShoppingCart(cartModel);

return RedirectToAction(“AddTo”);

}

The application service adds an element to the shopping cart:

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;

}

The *AddItem* method on the *ShoppingCart* domain object contains a bit of logic to increase the quantity if the product is already in the cart:

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;

}

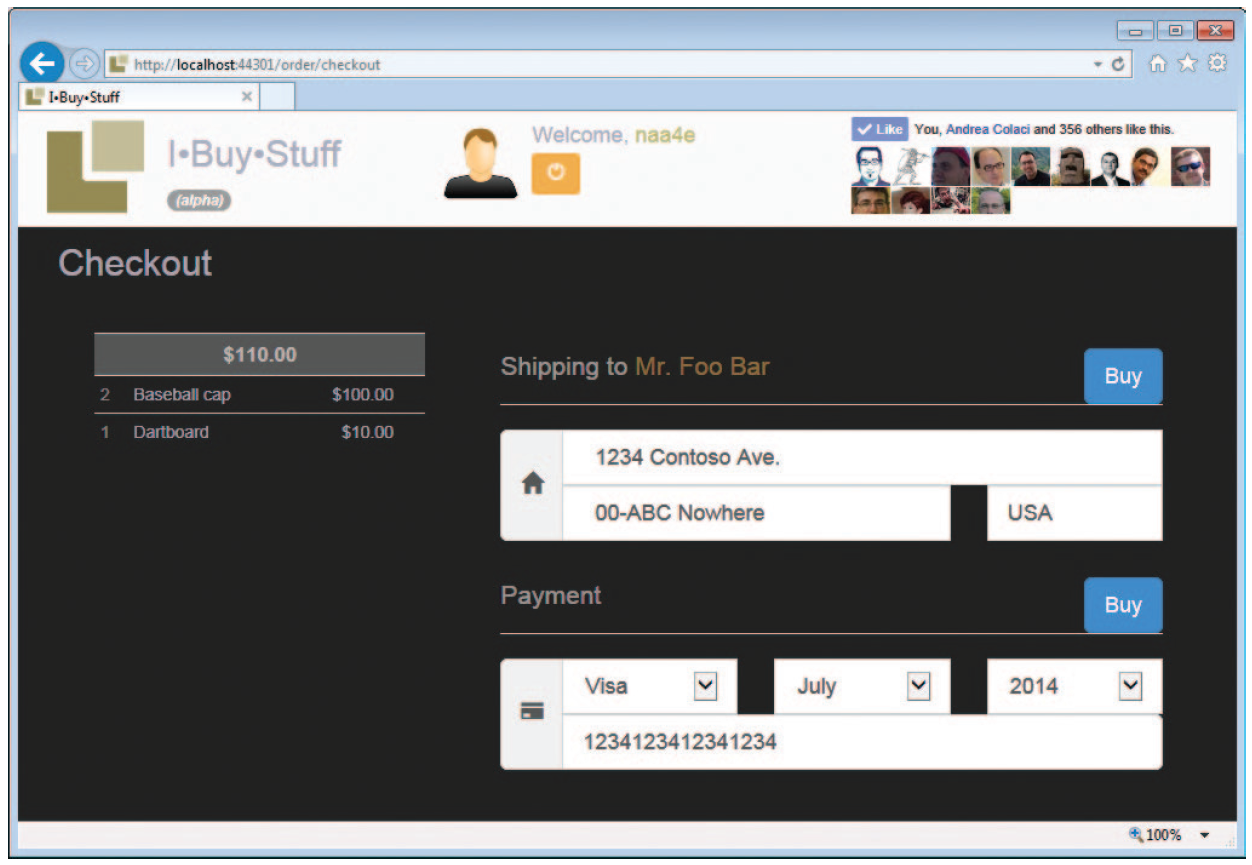
// Create new item

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

return this;

}

The checkout button leads to the page in Figure 9-9. Note that the shopping cart is now rendered in read mode and all actionable buttons are not displayed.



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

Note that the controller method that receives the command to add an item to the cart (or remove an item from it) in the end redirects to the *AddTo* action instead of just rendering the next view. This is done to avoid a repeated form submission if the user refreshes the page. In response to a refresh action (for example, when F5 is pressed), browsers just repeat the last action. Thanks to the redirect, however, the last action is a GET, not a POST.

Processing the order request

When the user inally clicks the Buy button, it’s time for the order to be processed and created. At this time, company policies and business rules must be applied. In general, processing the order is based on a worklow that is orchestrated by the application service invoked from the controller. The typical steps of the worklow are checking the availability of ordered goods, processing payment, forwarding shipping details, and storing order data in the database.

Most of these actions can be carried out in silent mode. Payment, though, might require that an external page be displayed to let the user interact with the banking back end. If payment requires its own user interface, you split the processing of the order into two phases: before and after payment. If not, you can implement the processing of the payment and synchronization with the shipping company as two parallel tasks using the .NET Framework Parallel API. Here’s how to split checkout into two steps:

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);

}

The method *ProcessOrderBeforePayment* on the application service saves checkout information (shipping address and payment details) and checks the stock level of ordered products. Depending on the policies enabled, it might also need to place a reill order to bring the stock level back to a safe value. If something goes wrong (for example, some goods are not available), the response is stored in *TempData* to be displayed by the Denied page across a page redirect.

Note: *TempData* is an ASP.NET MVC facility speciically created to persist request data temporarily across a redirect. This feature exists to enable the Post-Redirect-Get pattern, which protects against the bad effects of page refreshes.

If pre-payment checks go well, the method redirects to the payment page. This is where payment occurs. When done, the page redirects back to the speciied URL. In doing so, the payment page will pass the transaction ID that demonstrates payment. (This is a general description of how most payment APIs actually work.)

The post-payment phase might include booking a delivery through the shipping company back end, registering the order in the system, updating the stock level, and updating the records of the idelity card.

Fidelity card (or customer loyalty program)

Once the order is in the system, a few additional tasks might still be required. In a way, adding an order generates an event within the domain that might require one or more handlers. This introduces the point of *domain events*. In general, a domain event is an event that might be raised within the context of the domain model. Domain events are simply a way for the architect to clean up the design

and enable himself to handle situations in a more lexible way.

In particular, the requirements we have for I-Buy-Stuff state that whenever an order is created the system must update the stock level of the products and add points to the idelity card of the customer.

Both operations can be coded as plain method calls executed at the end of the *ProcessOrderAfterPayment* method in the application service class. It’s plain and simple, and it just works.

However, the logic behind the idelity card is subject to change as marketing campaigns are launched, ended, or modiied. The aforementioned implementation might turn out to be a bit too rigid in terms of maintenance and lead to frequent updates to the binary code. A more extensible design of what might happen once the order is created might smooth the work required in the long run to keep the system aligned to the business needs.

Domain events try to address this scenario. A domain event can be implemented as a plain C# event in some aggregate class, or it can be implemented through a sort of publish/subscribe mechanism, where the information is put on a bus and registered handlers get it and process it as appropriate. More importantly, handlers can be loaded and registered using a dependency-injection interface so that adding and removing a handler is as easy and lightweight as editing the coniguration ile of the application.

**Summary**

This chapter is an attempt to give substance to concepts and ideas that too often remain conined in the space of basic tutorials or in theory. We didn’t just present a domain model where classes have behavior and factories and make limited use of primitive types. We actually presented a multilayer ASP.NET MVC application that proceeds end-to-end from presentation down to the creation of an order in a scenario that is not an unrealistic CRUD.

Along the way, we turned into practice concepts we introduced in past chapters and showed that, while database structure is not a primary concern when modeling, it still is a constraint and concessions are often necessary to make the object model persistent.

We invite you to take a thorough look at the full source code of the I-Buy-Stuff application. The text in the chapter emphasized choices around the domain model; however, the code contains a lot more details and solutions that might be interesting to look at.

**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.