Then update the /Views/Nav/Menu.ascx template to render a special CSS class to indicate the highlighted link:

```
<% foreach(var link in Model) { %>
    <a href="<%= Url.RouteUrl(link.RouteValues) %>"
       class="<%= link.IsSelected ? "selected" : "" %>"
    >
        <%= link.Text %>
    </a>
<% } %>
```

Finally, we have a working navigation widget that highlights the current page, as shown in Figure 5-5.

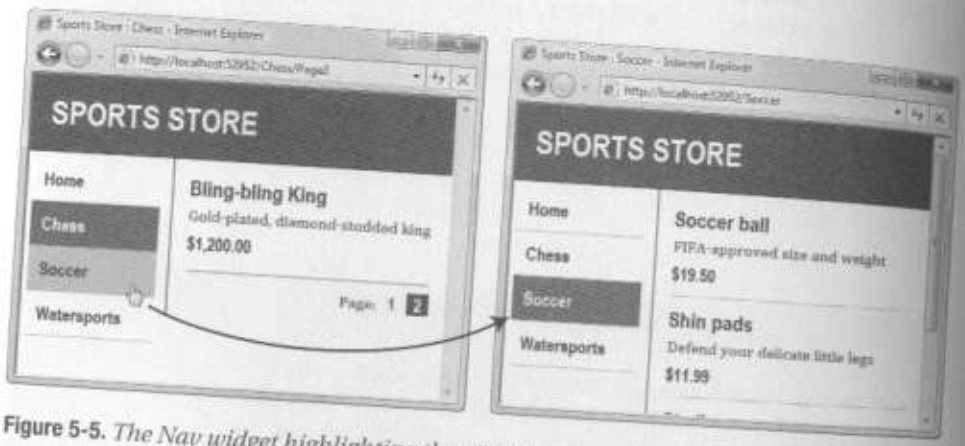

Figure 5-5. *The Nav widget highlighting the visitor's current location as they move*

Building the Shopping Cart

The application is coming along nicely, but it still won't sell any products, because there are no Buy buttons and there's no shopping cart. It's time to rectify that. In this section, you'll do the following:

- Expand your domain model to introduce the notion of a Cart, with its behavior defined in the form of unit tests, and work with a second controller class, CartController
- Create a custom *model binder* that gives you a very elegant (and testable) way for action methods to receive a Cart instance relating to the current visitor's browser session
- Learn why using multiple <form> tags can be a good thing in ASP.NET MVC (despite being nearly impossible in traditional ASP.NET WebForms)
- See how Html.RenderAction() can be used to make a reusable cart summary control quickly and easily (in comparison to creating NavController, which was a lengthy task)

In outline, you'll be aiming for the shopping cart experience shown in Figure 5-6.

Figure 5-6. *Sketch of shopping cart flow*

On product list screens, each product will appear with an "Add to cart" button. Clicking this adds the product to the visitor's shopping cart, and takes the visitor to the "Your cart" screen. That displays the contents of their cart, including its total value, and gives them a choice of two directions to go next: "Continue shopping" will take them back to the page they just came from (remembering both category and page number), and "Check out now" will go ahead to whatever screen completes the order.

Defining the Cart Entity

Since a shopping cart is part of your application's business domain, it makes sense to define Cart as a new model class. Put a class called Cart into your DomainModel project's Entities folder:

```
namespace DomainModel.Entities
{
    public class Cart
    {
        private List<CartLine> lines = new List<CartLine>();
        public IList<CartLine> Lines { get { return lines; } }

        public void AddItem(Product product, int quantity) { }
        public decimal ComputeTotalValue() { throw new NotImplementedException(); }
        public void Clear() { throw new NotImplementedException(); }
    }

    public class CartLine
    {
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

Domain logic, or business logic, is best situated on your domain model itself. That helps you to separate your business concerns from the sort of web application concerns (requests,

responses, links, paging, etc.) that live in controllers. So, the next step is to design and implement the following business rules that apply to `Cart`:

- The cart is initially empty.
- A cart can't have more than one line corresponding to a given product. (So, when you add a product for which there's already a corresponding line, it simply increases the quantity.)
- A cart's *total value* is the sum of its lines' prices multiplied by quantities. (For simplicity, we're omitting any concept of delivery charges.)

TESTING: SHOPPING CART BEHAVIOR

The existing trivial implementation of `Cart` and `CartLines` gives you an easy foothold to start defining their behaviors in terms of tests. Create a new class in your `Tests` project called `CartTests`:

```
[TestFixture]
public class CartTests
{
    [Test]
    public void Cart_Starts_Empty()
    {
        Cart cart = new Cart();
        Assert.AreEqual(0, cart.Lines.Count);
        Assert.AreEqual(0, cart.ComputeTotalValue());
    }

    [Test]
    public void Can_Add_Items_To_Cart()
    {
        Product p1 = new Product { ProductID = 1 };
        Product p2 = new Product { ProductID = 2 };

        // Add three products (two of which are same)
        Cart cart = new Cart();
        cart.AddItem(p1, 1);
        cart.AddItem(p1, 2);
        cart.AddItem(p2, 10);

        // Check the result is two lines
        Assert.AreEqual(2, cart.Lines.Count, "Wrong number of lines in cart");

        // Check quantities were added properly
        var p1Line = cart.Lines.Where(l => l.Product.ProductID == 1).First();
        var p2Line = cart.Lines.Where(l => l.Product.ProductID == 2).First();
        Assert.AreEqual(3, p1Line.Quantity);
        Assert.AreEqual(10, p2Line.Quantity);
    }

    [Test]
    public void Can_Be_Cleared()
    {
        Cart cart = new Cart();
        cart.AddItem(new Product(), 1);
        Assert.AreEqual(1, cart.Lines.Count);

        cart.Clear();
        Assert.AreEqual(0, cart.Lines.Count);
    }

    [Test]
    public void Calculates_Total_Value_Correctly()
    {
        Cart cart = new Cart();
        cart.AddItem(new Product { ProductID = 1, Price = 5 }, 10);
        cart.AddItem(new Product { ProductID = 2, Price = 2.1M }, 3);
        cart.AddItem(new Product { ProductID = 3, Price = 1000 }, 1);

        Assert.AreEqual(1056.3, cart.ComputeTotalValue());
    }
}
```

(In case you're unfamiliar with the syntax, the `M` in `2.1M` tells the C# compiler that it's a `decimal` literal value.)

This is simple stuff—you'll have no trouble implementing these behaviors with some tight C# 3 syntax:

```
public class Cart
{
    private List<CartLine> lines = new List<CartLine>();
    public IList<CartLine> Lines { get { return lines.AsReadOnly(); } }

    public void AddItem(Product product, int quantity)
    {
        // FirstOrDefault() is a LINQ extension method on IEnumerable
        var line = lines
                   .FirstOrDefault(l => l.Product.ProductID == product.ProductID);
        if (line == null)
            lines.Add(new CartLine { Product = product, Quantity = quantity });
        else
            line.Quantity += quantity;
    }
```

```
        public decimal ComputeTotalValue()
        {
            // Sum() is a LINQ extension method on IEnumerable
            return lines.Sum(l => l.Product.Price * l.Quantity);
        }

        public void Clear()
        {
            lines.Clear();
        }
    }
```

This will make your `CartTests` pass. Actually, there's one more thing: visitors who change their minds will need to remove items from their cart. To make the `Cart` class support item removal, add the following extra method to it:

```
public void RemoveLine(Product product)
{
    lines.RemoveAll(l => l.Product.ProductID == product.ProductID);
}
```

(Adding a test for this is an exercise for the enthusiastic reader.)

■ **Note** Notice that the `Lines` property now returns its data in *read-only* form. That makes sense: code in the UI layer shouldn't be allowed to modify the `Lines` collection directly, as it might ignore and violate business rules. As a matter of encapsulation, we want all changes to the `Lines` collection to go through the `Cart` class API.

Adding "Add to Cart" Buttons

Go back to your partial view, `/Views/Shared/ProductSummary.ascx`, and add an "Add to cart" button:

```
<div class="item">
    <h3><%= Model.Name %></h3>
    <%= Model.Description %>

    <% using(Html.BeginForm("AddToCart", "Cart")) { %>
        <%= Html.Hidden("ProductID") %>
        <%= Html.Hidden("returnUrl",
                        ViewContext.HttpContext.Request.Url.PathAndQuery) %>
        <input type="submit" value="+ Add to cart" />
    <% } %>

    <h4><%= Model.Price.ToString("c")%></h4>
</div>
```


Check it out—you're one step closer to selling some products (see Figure 5-7).

Figure 5-7. *"Add to cart" buttons*

Each of the "Add to cart" buttons will POST the relevant `ProductID` to an action called `AddToCart` on a controller class called `CartController`. Note that `Html.BeginForm()` renders forms with a `method` attribute of POST by default, though it also has an overload that lets you specify GET instead.

However, since `CartController` doesn't yet exist, if you click an "Add to cart" button, you'll get an error from the IoC container ("Value cannot be null. Parameter name: service.").

To get the black "Add to cart" buttons, you'll need to add more rules to your CSS file:

```
FORM { margin: 0; padding: 0; }
DIV.item FORM { float:right; }
DIV.item INPUT {
    color:White; background-color: #333; border: 1px solid black; cursor:pointer;
}
```

Multiple <form> Tags

In case you hadn't noticed, using the `Html.BeginForm()` helper in this way means that each "Add to cart" button gets rendered in its own separate little HTML `<form>`. If you're from an ASP.NET WebForms background, where each page is only allowed one single `<form>`, this probably seems strange and alarming, but don't worry—you'll get over it soon. In HTML terms, there's no reason why a page shouldn't have several (or even hundreds of) `<form>` tags, as long as they don't overlap or nest.

Technically, you don't *have* to put each of these buttons in a separate `<form>`. So why do I recommend doing so in this case? It's because you want each of these buttons to invoke an HTTP POST request with a different set of parameters, which is most easily done by creating a

separate <form> tag in each case. And why is it important to use POST here, not GET? Because the HTTP specification says that GET requests must be *idempotent* (i.e., not cause changes to anything), and adding a product to a cart definitely changes the cart. You'll hear more about why this matters, and what can happen if you ignore this advice, in Chapter 8.

Giving Each Visitor a Separate Shopping Cart

To make those "Add to cart" buttons work, you'll need to create a new controller class, `CartController`, featuring action methods for adding items to the cart and later removing them. But hang on a moment—what cart? You've defined the `Cart` class, but so far that's all. There aren't yet any instances of it available to your application, and in fact you haven't even decided how that will work.

- Where are the `Cart` objects stored—in the database, or in web server memory?
- Is there one universal `Cart` shared by everyone, does each visitor have a separate `Cart` instance, or is a brand new instance created for every HTTP request?

Obviously, you'll need a `Cart` to survive for longer than a single HTTP request, because visitors will add `CartLines` to it one by one in a series of requests. And of course each visitor needs a separate cart, not shared with other visitors who happen to be shopping at the same time; otherwise, there will be chaos.

The natural way to achieve these characteristics is to store `Cart` objects in the `Session` collection. If you have any prior ASP.NET experience (or even classic ASP experience), you'll know that the `Session` collection holds objects for the duration of a visitor's browsing session (i.e., across multiple requests), and each visitor has their own separate `Session` collection. By default, its data is stored in the web server's memory, but you can configure different storage strategies (in process, out of process, in a SQL database, etc.) using `web.config`.

ASP.NET MVC Offers a Tidier Way of Working with Session Storage

So far, this discussion of shopping carts and `Session` is obvious. But wait! You need to understand that even though ASP.NET MVC shares many infrastructural components (such as the `Session` collection) with older technologies such as classic ASP and ASP.NET WebForms, there's a different philosophy regarding how that infrastructure is supposed to be used.

If you let your controllers manipulate the `Session` collection directly, pushing objects in and pulling them out on an ad hoc basis, as if `Session` were a big, fun, free-for-all global variable, then you'll hit some maintainability issues. What if controllers get out of sync, one of them looking for `Session["Cart"]` and another looking for `Session["_cart"]`? What if a controller assumes that `Session["_cart"]` will already have been populated by another controller, but it hasn't? What about the awkwardness of writing unit tests for anything that accesses `Session`, considering that you'd need a mock or fake `Session` collection?

In ASP.NET MVC, the best kind of action method is a *pure function* of its parameters. By this, I mean that the action method reads data only from its parameters, and writes data only to its parameters, and does not refer to `HttpContext` or `Session` or any other state external to the controller. If you can achieve that (which you can do normally, but not necessarily always), then you have placed a limit on how complex your controllers and actions can get. It leads to a

semantic clarity that makes the code easy to comprehend at a glance. By definition, such stand-alone methods are also easy to unit test, because there is no external state that needs to be simulated.

Ideally, then, our action methods should be given a `Cart` instance as a parameter, so they don't have to know or care about where those instances come from. That will make unit testing easy: tests will be able to supply a `Cart` to the action, let the action run, and then check what changes were made to the `Cart`. This sounds like a good plan!

Creating a Custom Model Binder

As you've heard, ASP.NET MVC has a mechanism called model binding that, among other things, is used to prepare the parameters passed to action methods. This is how it was possible in Chapter 2 to receive a `GuestResponse` instance parsed automatically from the incoming HTTP request.

The mechanism is both powerful and extensible. You'll now learn how to make a simple custom model binder that supplies instances retrieved from some backing store (in this case, `Session`). Once this is set up, action methods will easily be able to receive a `Cart` as a parameter without having to care about how such instances are created or stored. Add the following class to the root of your `WebUI` project (technically it can go anywhere):

```
public class CartModelBinder : IModelBinder
{
    private const string cartSessionKey = "_cart";

    public object BindModel(ControllerContext controllerContext,
                            ModelBindingContext bindingContext)
    {
        // Some modelbinders can update properties on existing model instances. This
        // one doesn't need to - it's only used to supply action method parameters.
        if(bindingContext.Model != null)
            throw new InvalidOperationException("Cannot update instances");

        // Return the cart from Session[] (creating it first if necessary)
        Cart cart = (Cart)controllerContext.HttpContext.Session[cartSessionKey];
        if(cart == null) {
            cart = new Cart();
            controllerContext.HttpContext.Session[cartSessionKey] = cart;
        }
        return cart;
    }
}
```

You'll learn more model binding in detail in Chapter 12, including how the built-in default binder is capable of instantiating and updating any custom .NET type, and even collections of custom types. For now, you can understand `CartModelBinder` simply as a kind of `Cart` factory that encapsulates the logic of giving each visitor a separate instance stored in their `Session` collection.

The MVC Framework won't use `CartModelBinder` unless you tell it to. Add the following line to your `Global.asax.cs` file's `Application_Start()` method, nominating `CartModelBinder` as the binder to use whenever a `Cart` instance is required:

```
protected void Application_Start()
{
    // ... leave rest as before ...
    ModelBinders.Binders.Add(typeof(Cart), new CartModelBinder());
}
```

Creating CartController

Let's now create `CartController`, relying on our custom model binder to supply `Cart` instances. We can start with the `AddToCart()` action method.

TESTING: CARTCONTROLLER

There isn't yet any controller class called `CartController`, but that doesn't stop you from designing and defining its behavior in terms of tests. Add a new class to your `Tests` project called `CartControllerTests`:

```
[TestFixture]
public class CartControllerTests
{
    [Test]
    public void Can_Add_Product_To_Cart()
    {
        // Arrange: Set up a mock repository with two products
        var mockProductsRepos = new Moq.Mock<IProductsRepository>();
        var products = new System.Collections.Generic.List<Product> {
            new Product { ProductID = 14, Name = "Much Ado About Nothing" },
            new Product { ProductID = 27, Name = "The Comedy of Errors" },
        };
        mockProductsRepos.Setup(x => x.Products)
                         .Returns(products.AsQueryable());
        var cart = new Cart();
        var controller = new CartController(mockProductsRepos.Object);

        // Act: Try adding a product to the cart
        RedirectToRouteResult result =
            controller.AddToCart(cart, 27, "someReturnUrl");

        // Assert
        Assert.AreEqual(1, cart.Lines.Count);
        Assert.AreEqual("The Comedy of Errors", cart.Lines[0].Product.Name);
        Assert.AreEqual(1, cart.Lines[0].Quantity);

        // Check that the visitor was redirected to the cart display screen
        Assert.AreEqual("Index", result.RouteValues["action"]);
        Assert.AreEqual("someReturnUrl", result.RouteValues["returnUrl"]);
    }
}
```

Notice that `CartController` is assumed to take an `IProductsRepository` as a constructor parameter. In IoC terms, this means that `CartController` has a dependency on `IProductsRepository`. The test indicates that a `Cart` will be the first parameter passed to the `AddToCart()` method. This test also defines that, after adding the requested product to the visitor's cart, the controller should redirect the visitor to an action called `Index`.

You can, at this point, also write a test called `Can_Remove_Product_From_Cart()`. I'll leave that as an exercise.

Implementing AddToCart and RemoveFromCart

To get the solution to compile and the tests to pass, you'll need to implement `CartController` with a couple of fairly simple action methods. You just need to set an IoC dependency on `IProductsRepository` (by having a constructor parameter of that type), take a `Cart` as one of the action method parameters, and then combine the values supplied to add and remove products:

```
public class CartController : Controller
{
    private IProductsRepository productsRepository;
    public CartController(IProductsRepository productsRepository)
    {
        this.productsRepository = productsRepository;
    }

    public RedirectToRouteResult AddToCart(Cart cart, int productID,
                                           string returnUrl)
    {
        Product product = productsRepository.Products
                                            .FirstOrDefault(p => p.ProductID == productID);
        cart.AddItem(product, 1);
        return RedirectToAction("Index", new { returnUrl });
    }

    public RedirectToRouteResult RemoveFromCart(Cart cart, int productID,
                                                string returnUrl)
    {
        Product product = productsRepository.Products
                                            .FirstOrDefault(p => p.ProductID == productID);
        cart.RemoveLine(product);
        return RedirectToAction("Index", new { returnUrl });
    }
}
```

The important thing to notice is that `AddToCart` and `RemoveFromCart`'s parameter names match the `<form>` field names defined in `/Views/Shared/ProductSummary.ascx` (i.e., `productId` and `returnUrl`). That enables ASP.NET MVC to associate incoming form POST variables with those parameters.

Remember, `RedirectToAction()` results in an HTTP 302 redirection.[4] That causes the visitor's browser to rerequest the new URL, which in this case will be `/Cart/Index`.

Displaying the Cart

Let's recap what you've achieved with the cart so far:

- You've defined `Cart` and `CartLine` model objects and implemented their behavior. Whenever an action method asks for a `Cart` as a parameter, `CartModelBinder` will automatically kick in and supply the current visitor's cart as taken from the `Session` collection.
- You've added "Add to cart" buttons on to the product list screens, which lead to `CartController`'s `AddToCart()` action.
- You've implemented the `AddToCart()` action method, which adds the specified product to the visitor's cart, and then redirects to `CartController`'s `Index` action. (`Index` is supposed to display the current cart contents, but you haven't implemented that yet.)

So what happens if you run the application and click "Add to cart" on some product? (See Figure 5-8.)

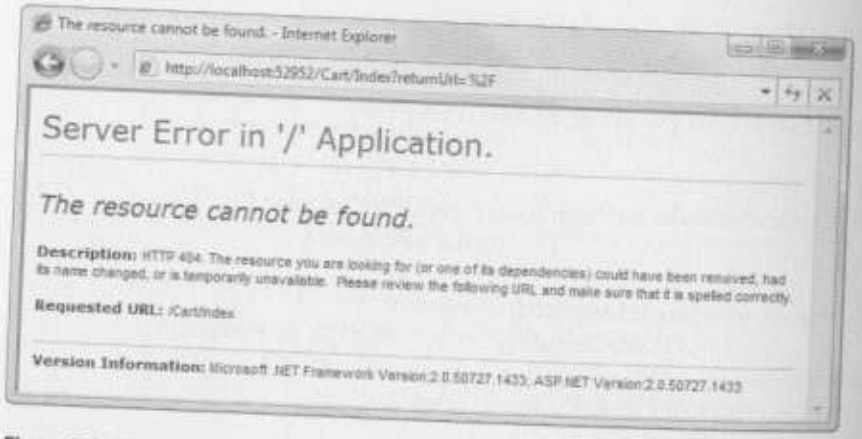

Figure 5-8. *The result of clicking "Add to cart"*

4. Just like `Response.Redirect()` in ASP.NET WebForms, which you could actually call from here, but that wouldn't return a nice `ActionResult`, making the controller hard to test.

Not surprisingly, it gives a 404 Not Found error, because you haven't yet implemented `CartController`'s `Index` action. It's pretty trivial, though, because all that action has to do is render a view, supplying the visitor's `Cart` and the current `returnUrl` value. It also makes sense to populate `ViewData["CurrentCategory"]` with the string `Cart`, so that the navigation menu won't highlight any other menu item.

TESTING: CARTCONTROLLER'S INDEX ACTION

With the design established, it's easy to represent it as a test. Considering what data this view is going to render (the visitor's cart and a button to go back to the product list), let's say that `CartController`'s forthcoming `Index()` action method should set `Model` to reference the visitor's cart, and should also populate `ViewData["returnUrl"]`:

```
[Test]
public void Index_Action_Renders_Default_View_With_Cart_And_ReturnUrl()
{
    // Set up the controller
    Cart cart = new Cart();
    CartController controller = new CartController(null);

    // Invoke action method
    ViewResult result = controller.Index(cart, "myReturnUrl");

    // Verify results
    Assert.IsEmpty(result.ViewName); // Renders default view
    Assert.AreSame(cart, result.ViewData.Model);
    Assert.AreEqual("myReturnUrl", result.ViewData["returnUrl"]);
    Assert.AreEqual("Cart", result.ViewData["CurrentCategory"]);
}
```

As always, this won't compile because at first there isn't yet any such action method as `Index()`.

Implement the simple `Index()` action method by adding a new method to `CartController`:

```
public ViewResult Index(Cart cart, string returnUrl)
{
    ViewData["returnUrl"] = returnUrl;
    ViewData["CurrentCategory"] = "Cart";
    return View(cart);
}
```

This will make the unit test pass, but you can't run it yet, because you haven't yet defined its view template. So, right-click inside that method, choose Add View, check "Create a strongly typed view," and choose the "View data class" `DomainModel.Entities.Cart`.

When the template appears, fill in the <asp:Content> placeholders, adding markup to render the Cart instance as follows:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
    SportsStore : Your Cart
</asp:Content>

<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h2>Your cart</h2>
    <table width="90%" align="center">
        <thead><tr>
            <th align="center">Quantity</th>
            <th align="left">Item</th>
            <th align="right">Price</th>
            <th align="right">Subtotal</th>
        </tr></thead>
        <tbody>
            <% foreach(var line in Model.Lines) { %>
                <tr>
                    <td align="center"><%= line.Quantity %></td>
                    <td align="left"><%= line.Product.Name %></td>
                    <td align="right"><%= line.Product.Price.ToString("c") %></td>
                    <td align="right">
                        <%= (line.Quantity*line.Product.Price).ToString("c") %>
                    </td>
                </tr>
            <% } %>
        </tbody>
        <tfoot><tr>
            <td colspan="3" align="right">Total:</td>
            <td align="right">
                <%= Model.ComputeTotalValue().ToString("c") %>
            </td>
        </tr></tfoot>
    </table>
    <p align="center" class="actionButtons">
        <a href="<%= Html.Encode(ViewData["returnUrl"]) %>">Continue shopping</a>
    </p>
</asp:Content>
```

Don't be intimidated by the apparent complexity of this view template. All it does is iterate over its `Model.Lines` collection, printing out an HTML table row for each line. Finally, it includes a handy button, "Continue shopping," which sends the visitor back to whatever product list page they were previously on.

The result? You now have a working cart, as shown in Figure 5-9. You can add an item, click "Continue shopping," add another item, and so on.

Figure 5-9. *The shopping cart is now working.*

To get this appearance, you'll need to add a few more CSS rules to `/Content/styles.css`:

```
H2 { margin-top: 0.3em }
TFOOT TD { border-top: 1px dotted gray; font-weight: bold; }
.actionButtons A {
    font: .8em Arial; color: White; margin: 0 .5em 0 .5em;
    text-decoration: none; padding: .15em 1.5em .2em 1.5em;
    background-color: #353535; border: 1px solid black;
}
```

Eagle-eyed readers will notice that there isn't yet any way to complete and pay for an order (a convention known as *checkout*). You'll add that feature shortly; but first, there are a couple more cart features to add.

Removing Items from the Cart

Whoops, I just realized I don't need any more soccer balls, I have plenty already! But how do I remove them from my cart? Update `/Views/Cart/Index.aspx` by adding a Remove button in a new column on each `CartLine` row. Once again, since this action causes a permanent side

effect (it removes an item from the cart), you should use a `<form>` that submits via a POST request rather than an `Html.ActionLink()` that invokes a GET:

```
<% foreach(var line in Model.Lines) { %>
    <tr>
        <td align="center"><%= line.Quantity %></td>
        <td align="left"><%= line.Product.Name %></td>
        <td align="right"><%= line.Product.Price.ToString("c") %></td>
        <td align="right">
            <%= (line.Quantity*line.Product.Price).ToString("c") %>
        </td>
        <td>
            <% using(Html.BeginForm("RemoveFromCart", "Cart")) { %>
                <%= Html.Hidden("ProductID", line.Product.ProductID) %>
                <%= Html.Hidden("returnUrl", ViewData["returnUrl"]) %>
                <input type="submit" value="Remove" />
            <% } %>
        </td>
    </tr>
<% } %>
```

Ideally, you should also add blank cells to the header and footer rows, so that all rows have the same number of columns. In any case, it already works because you've already implemented the `RemoveFromCart(cart, productId, returnUrl)` action method, and its parameter names match the `<form>` field names you just added (i.e., `ProductId` and `returnUrl`) (see Figure 5-10).

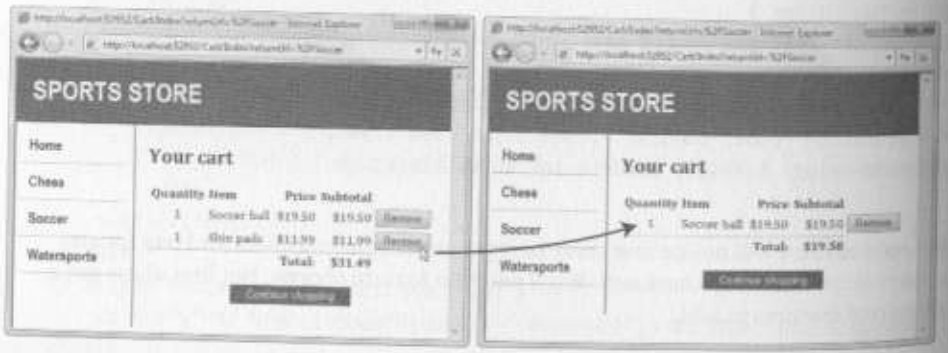

Figure 5-10. *The cart's Remove button is working.*

Displaying a Cart Summary in the Title Bar

SportsStore has two major usability problems right now:

- Visitors don't have any idea of what's in their cart without actually going to the cart display screen.
- Visitors can't get to the cart display screen (e.g., to check out) without actually adding something to their cart.

To solve both of these, let's add something else to the application's master page: a new widget that displays a brief summary of the current cart contents and offers a link to the cart display page. You'll do this in much the same way as you implemented the navigation widget (i.e., as an action method whose output you can inject into `/Views/Site.Master`). However, this time it will be much easier, demonstrating that `Html.RenderAction()` widgets can be quick and simple to implement.

Add a new action method called `Summary()` to `CartController`:

```
public class CartController : Controller
{
    // Leave rest of class as-is

    public ViewResult Summary(Cart cart)
    {
        return View(cart);
    }
}
```

As you see, it can be quite trivial. It needs only render a view, supplying the current cart data so that its view can produce a summary. You could write a unit test for this quite easily, but I'll omit the details because it's so simple.

Next, create a partial view template for the widget. Right-click inside the `Summary()` method, choose Add View, check "Create a partial view," and make it strongly typed for the `DomainModel.Entities.Cart` class. Add the following markup:

```
<% if(Model.Lines.Count > 0) { %>
    <div id="cart">
        <span class="caption">
            <b>Your cart:</b>
            <%= Model.Lines.Sum(x => x.Quantity) %> item(s),
            <%= Model.ComputeTotalValue().ToString("c") %>
        </span>
        <%= Html.ActionLink("Check out", "Index", "Cart",
            new { returnUrl = Request.Url.PathAndQuery }, null)%>
    </div>
<% } %>
```

To plug the widget into the master page, add to `/Views/Shared/Site.Master`:

```
<div id="header">
    <% if(!(ViewContext.Controller is WebUI.Controllers.CartController))
        Html.RenderAction("Summary", "Cart"); %>
    <div class="title">SPORTS STORE</div>
</div>
```

Notice that this code uses the `ViewContext` object to consider what controller is currently being rendered. The cart summary widget is hidden if the visitor is on `CartController`, because it would be confusing to display a link to checkout if the visitor is already checking out. Similarly, `/Views/Cart/Summary.ascx` knows to generate no output if the cart is empty.

Putting such logic in a view template is at the outer limit of what I would allow in a view template; any more complicated and it would be better implemented by means of a flag set by the controller (so you could test it). But of course, this is subjective. You must make your own decision about where to set the threshold.

Now add one or two items to your cart, and you'll get something similar to Figure 5-11.

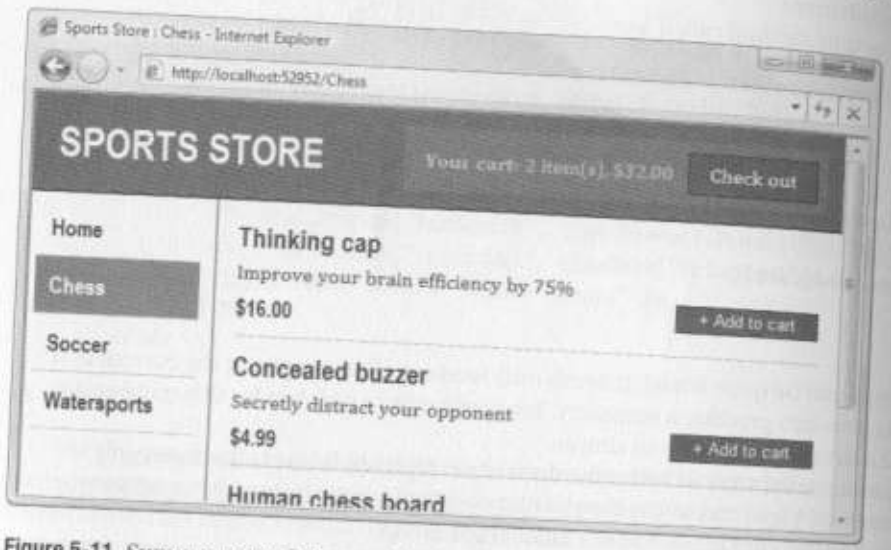

Figure 5-11. *Summary.ascx being rendered in the title bar*

Looks good! Or at least it does when you've added a few more rules to `/Content/styles.css`:

```
DIV#cart { float:right; margin: .8em; color: Silver;
  background-color: #555; padding: .5em .5em .5em 1em; }
DIV#cart A { text-decoration: none; padding: .4em 1em .4em 1em; line-height:2.1em;
  margin-left: .5em; background-color: #333; color:White; border: 1px solid black;}
DIV#cart SPAN.summary { color: White; }
```

Visitors now have an idea of what's in their cart, and it's obvious how to get from any product list screen to the cart screen.

Submitting Orders

This brings us to the final customer-oriented feature in SportsStore: the ability to complete, or check out, an order. Once again, this is an aspect of the business domain, so you'll need to add a bit more code to the domain model. You'll need to let the customer enter shipping details, which must be validated in some sensible way.

In this product development cycle, SportsStore will just send details of completed orders to the site administrator by e-mail. It need not store the order data in your database. However, that plan might change in the future, so to make this behavior easily changeable, you'll implement an abstract order submission service, `IOrderSubmitter`.

Enhancing the Domain Model

Get started by implementing a model class for shipping details. Add a new class to your `DomainModel` project's `Entities` folder, called `ShippingDetails`:

```
namespace DomainModel.Entities
{
    public class ShippingDetails : IDataErrorInfo
    {
        public string Name { get; set; }
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }
        public string City { get; set; }
        public string State { get; set; }
        public string Zip { get; set; }
        public string Country { get; set; }
        public bool GiftWrap { get; set; }

        public string this[string columnName] // Validation rules
        {
            get {
                if ((columnName == "Name") && string.IsNullOrEmpty(Name))
                    return "Please enter a name";
                if ((columnName == "Line1") && string.IsNullOrEmpty(Line1))
                    return "Please enter the first address line";
                if ((columnName == "City") && string.IsNullOrEmpty(City))
                    return "Please enter a city name";
                if ((columnName == "State") && string.IsNullOrEmpty(State))
                    return "Please enter a state name";
                if ((columnName == "Country") && string.IsNullOrEmpty(Country))
                    return "Please enter a country name";
                return null;
            }
        }

        public string Error { get { return null; } } // Not required
    }
}
```

Just like in Chapter 2, we're defining validation rules using the `IDataErrorInfo` interface, which is automatically recognized and respected by ASP.NET MVC's model binder. In this example, the rules are very simple: a few of the properties must not be empty—that's all. You could add arbitrary logic to decide whether or not a given property was valid.

This is the simplest of several possible ways of implementing server-side validation in ASP.NET MVC, although it has a number of drawbacks that you'll learn about in Chapter 11 (where you'll also learn about some more sophisticated and powerful alternatives).

TESTING: SHIPPING DETAILS

Before you go any further with `ShippingDetails`, it's time to design the application's behavior using tests. Each `Cart` should hold a set of `ShippingDetails` (so `ShippingDetails` should be a property of `Cart`), and `ShippingDetails` should initially be empty. Express that design by adding more tests to `CartTests`:

```
[Test]
public void Cart_Shipping_Details_Start_Empty()
{
    Cart cart = new Cart();
    ShippingDetails d = cart.ShippingDetails;
    Assert.IsNull(d.Name);
    Assert.IsNull(d.Line1); Assert.IsNull(d.Line2); Assert.IsNull(d.Line3);
    Assert.IsNull(d.City); Assert.IsNull(d.State); Assert.IsNull(d.Country);
    Assert.IsNull(d.Zip);
}

[Test]
public void Cart_Not_GiftWrapped_By_Default()
{
    Cart cart = new Cart();
    Assert.IsFalse(cart.ShippingDetails.GiftWrap);
}
```

Apart from the compiler error ("'DomainModel.Entities.Cart' does not contain a definition for 'ShippingDetails' . . . "), these tests would happen to pass because they match C#'s default object initialization behavior. Still, it's worth having the tests to ensure that nobody accidentally alters the behavior in the future.

To satisfy the design expressed by the preceding tests (i.e., each `Cart` should hold a set of `ShippingDetails`), update `Cart`:

```
public class Cart
{
    private List<CartLine> lines = new List<CartLine>();
    public IList<CartLine> Lines { get { return lines.AsReadOnly(); } }

    private ShippingDetails shippingDetails = new ShippingDetails();
    public ShippingDetails ShippingDetails { get { return shippingDetails; } }

    // (etc... rest of class unchanged)
```

That's the domain model sorted out. The tests will now compile and pass. The next job is to use the updated domain model in a new checkout screen.

Adding the "Check Out Now" Button

Returning to the cart's `Index` view, add a button that navigates to an action called `CheckOut` (see Figure 5-12):

```
...
<p align="center" class="actionButtons">
    <a href="<%= Html.Encode(ViewData["returnUrl"]) %>">Continue shopping</a>
    <%= Html.ActionLink("Check out now", "CheckOut") %>
</p>
</asp:Content>
```

Figure 5-12. *The "Check out now" button*

Prompting the Customer for Shipping Details

To make the "Check out now" link work, you'll need to add a new action, `CheckOut`, to `CartController`. All it needs to do is render a view, which will be the "shipping details" form:

```
[AcceptVerbs(HttpVerbs.Get)]
public ViewResult CheckOut(Cart cart)
{
    return View(cart.ShippingDetails);
}
```

(It's restricted only to respond to GET requests. That's because there will soon be another method matching the `CheckOut` action, which responds to POST requests.)

Add a view template for the action method you just created (it doesn't matter whether it's strongly typed or not), containing the following markup:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
    SportsStore : Check Out
</asp:Content>
```

```
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h2>Check out now</h2>
    Please enter your details, and we'll ship your goods right away!
    <% using(Html.BeginForm()) { %>
        <h3>Ship to</h3>
        <div>Name: <%= Html.TextBox("Name") %></div>
        <h3>Address</h3>
        <div>Line 1: <%= Html.TextBox("Line1") %></div>
        <div>Line 2: <%= Html.TextBox("Line2") %></div>
        <div>Line 3: <%= Html.TextBox("Line3") %></div>
        <div>City: <%= Html.TextBox("City") %></div>
        <div>State: <%= Html.TextBox("State") %></div>
        <div>Zip: <%= Html.TextBox("Zip") %></div>
        <div>Country: <%= Html.TextBox("Country") %></div>

        <h3>Options</h3>
        <%= Html.CheckBox("GiftWrap") %> Gift wrap these items

        <p align="center"><input type="submit" value="Complete order" /></p>
    <% } %>
</asp:Content>
```

This results in the page shown in Figure 5-13.

Figure 5-13. *The shipping details screen*

Defining an Order Submitter IoC Component

When the user posts this form back to the server, you could just have some action method code that sends the order details by e-mail through some SMTP server. That would be convenient, but would lead to three problems:

Changeability: In the future, you're likely to change this behavior so that order details are stored in the database instead. This could be awkward if `CartController`'s logic is mixed up with e-mail-sending logic.

Testability: Unless your SMTP server's API is specifically designed for testability, it could be difficult to supply a mock SMTP server during unit tests. So, either you'd have to write no unit tests for `CheckOut()`, or your tests would have to actually send real e-mails to a real SMTP server.

Configurability: You'll need some way of configuring an SMTP server address. There are many ways to achieve this, but how will you accomplish it cleanly (i.e., without having to change your means of configuration accordingly if you later switch to a different SMTP server product)?

Like so many problems in computer science, all three of these can be sidestepped by introducing an extra layer of abstraction. Specifically, define `IOrderSubmitter`, which will be an IoC component responsible for submitting completed, valid orders. Create a new folder in your `DomainModel` project, `Services`,[5] and add this `interface`:

```
namespace DomainModel.Services
{
    public interface IOrderSubmitter
    {
        void SubmitOrder(Cart cart);
    }
}
```

Now you can use this definition to write the rest of the `CheckOut` action without complicating `CartController` with the nitty-gritty details of actually sending e-mails.

Completing CartController

To complete `CartController`, you'll need to set up its dependency on `IOrderSubmitter`. Update `CartController`'s constructor:

```
private IProductsRepository productsRepository;
private IOrderSubmitter orderSubmitter;
public CartController(IProductsRepository productsRepository,
                      IOrderSubmitter orderSubmitter)
```

5. Even though I call it a "service," it's not going to be a "web service." There's an unfortunate clash of terminology here: ASP.NET developers are accustomed to saying "service" for ASMX web services, while in the IoC/domain-driven design space, services are components that do a job but aren't entity or value objects. Hopefully it won't cause much confusion in this case (`IOrderSubmitter` looks nothing like a web service).

```
    {
        this.productsRepository = productsRepository;
        this.orderSubmitter = orderSubmitter;
    }
```

TESTING: UPDATING YOUR TESTS

At this point, you won't be able to compile the solution until you update any unit tests that reference `CartController`. That's because it now takes two constructor parameters, whereas your test code tries to supply just one. Update each test that instantiates a `CartController` to pass `null` for the `orderSubmitter` parameter. For example, update `Can_Add_ProductTo_Cart()`:

```
var controller = new CartController(mockProductsRepos.Object, null);
```

The tests should all still pass.

TESTING: ORDER SUBMISSION

Now you're ready to define the behavior of the POST overload of `CheckOut()` via tests. Specifically, if the user submits either an empty cart or an empty set of shipping details, then the `CheckOut()` action should simply redisplay its default view. Only if the cart is non-empty *and* the shipping details are valid should it submit the order through the `IOrderSubmitter` and render a different view called `Completed`. Also, after an order is submitted, the visitor's cart must be emptied (otherwise they might accidentally resubmit it).

This design is expressed by the following tests, which you should add to `CartControllerTests`:

```
[Test] public void
Submitting_Order_With_No_Lines_Displays_Default_View_With_Error()
{
    // Arrange
    CartController controller = new CartController(null, null);
    Cart cart = new Cart();
    // Act
    var result = controller.CheckOut(cart, new FormCollection());
    // Assert
    Assert.IsEmpty(result.ViewName);
    Assert.IsFalse(result.ViewData.ModelState.IsValid);
}

[Test] public void
   Submitting_Empty_Shipping_Details_Displays_Default_View_With_Error()
{
    // Arrange
    CartController controller = new CartController(null, null);
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Act
    var result = controller.CheckOut(cart, new FormCollection {
        { "Name", "" }
    });
    // Assert
    Assert.IsEmpty(result.ViewName);
    Assert.IsFalse(result.ViewData.ModelState.IsValid);
}

[Test] public void
Valid_Order_Goes_To_Submitter_And_Displays_Completed_View()
{
    // Arrange
    var mockSubmitter = new Moq.Mock<IOrderSubmitter>();
    CartController controller = new CartController(null, mockSubmitter.Object);
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    var formData = new FormCollection {
        { "Name", "Steve" }, { "Line1", "123 My Street" },
        { "Line2", "MyArea" }, { "Line3", "" },
        { "City", "MyCity" }, { "State", "Some State" },
        { "Zip", "123ABCDEF" }, { "Country", "Far far away" },
        { "GiftWrap", bool.TrueString }
    };

    // Act
    var result = controller.CheckOut(cart, formData);

    // Assert
    Assert.AreEqual("Completed", result.ViewName);
    mockSubmitter.Verify(x => x.SubmitOrder(cart));
    Assert.AreEqual(0, cart.Lines.Count);
}
```

To implement the POST overload of the `CheckOut` action, and to satisfy the preceding unit tests, add a new method to `CartController`:

```
[AcceptVerbs(HttpVerbs.Post)]
public ViewResult CheckOut(Cart cart, FormCollection form)
{
    // Empty carts can't be checked out
    if(cart.Lines.Count == 0) {
        ModelState.AddModelError("Cart", "Sorry, your cart is empty!");
        return View();
    }
```

```
    // Invoke model binding manually
    if (TryUpdateModel(cart.ShippingDetails, form.ToValueProvider())) {
        orderSubmitter.SubmitOrder(cart);
        cart.Clear();
        return View("Completed");
    }
    else // Something was invalid
        return View();
}
```

When this action method calls `TryUpdateModel()`, the model binding system inspects all the key/value pairs in `form` (which are taken from the incoming `Request.Form` collection—i.e., the text box names and values entered by the visitor), and uses them to populate the correspondingly named properties of `cart.ShippingDetails`. This is the same model binding mechanism that supplies action method parameters, except here we're invoking it manually because `cart.ShippingDetails` isn't an action method parameter. You'll learn more about this technique, including how to use prefixes to deal with clashing names, in Chapter 11.

Also notice the `AddModelError()` method, which lets you register any error messages that you want to display back to the user. You'll cause these messages to be displayed shortly.

Adding a Fake Order Submitter

Unfortunately, the application is now unable to run because your IoC container doesn't know what value to supply for `CartController`'s `orderSubmitter` constructor parameter (see Figure 5-14).

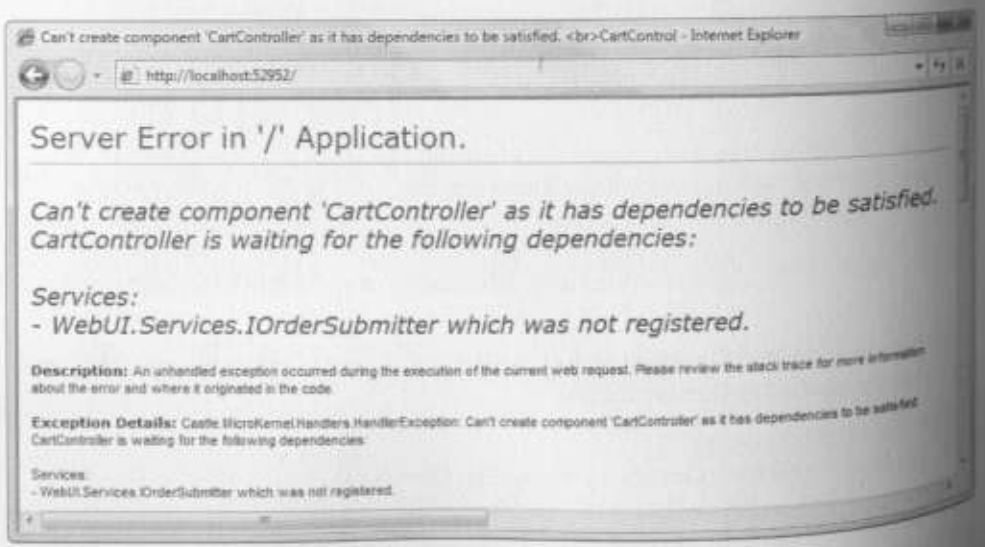

Figure 5-14. *Windsor's error message when it can't satisfy a dependency*

To get around this, define a `FakeOrderSubmitter` in your `DomainModel` project's `/Services` folder:

```
namespace DomainModel.Services
{
    public class FakeOrderSubmitter : IOrderSubmitter
    {
        public void SubmitOrder(Cart cart)
        {
            // Do nothing
        }
    }
}
```

Then register it in the `<castle>` section of your `web.config` file:

```
<castle>
  <components>
    <!-- Leave rest as is - just add this new node -->
    <component id="OrderSubmitter"
               service="DomainModel.Services.IOrderSubmitter, DomainModel"
               type="DomainModel.Services.FakeOrderSubmitter, DomainModel" />
  </components>
</castle>
```

You'll now be able to run the application.

Displaying Validation Errors

If you go to the checkout screen and enter an incomplete set of shipping details, the application will simply redisplay the "Check out now" screen without explaining what's wrong. Tell it where to display the error messages by adding an `Html.ValidationSummary()` into the `CheckOut.aspx` view:

```
<h2>Check out now</h2>
Please enter your details, and we'll ship your goods right away!
<%= Html.ValidationSummary() %>
... leave rest as before ...
```

Now, if the user's submission isn't valid, they'll get back a summary of the validation messages, as shown in Figure 5-15. The validation message summary will also include the phrase "Sorry, your cart is empty!" if someone tries to check out with an empty cart.

Also notice that the text boxes corresponding to invalid input are highlighted to help the user quickly locate the problem. ASP.NET MVC's built-in input helpers highlight themselves automatically (by giving themselves a particular CSS class) when they detect a registered validation error message that corresponds to their own name.

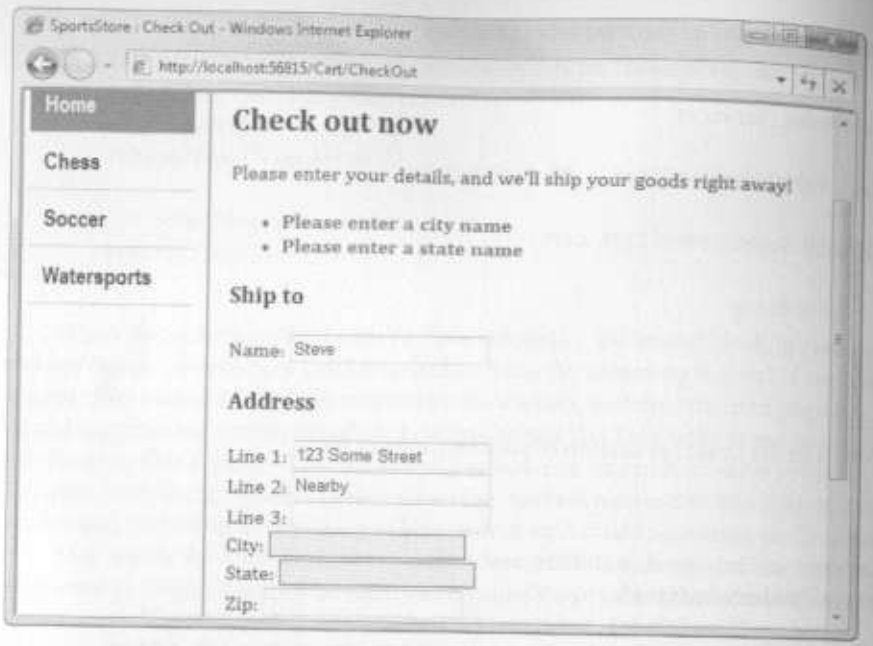

Figure 5-15. *Validation error messages are now displayed.*

To get the text box highlighting shown in the preceding figure, you'll need to add the following rules to your CSS file:

```
.field-validation-error { color: red; }
.input-validation-error { border: 1px solid red; background-color: #ffeeee; }
.validation-summary-errors { font-weight: bold; color: red; }
```

Displaying a "Thanks for Your Order" Screen

To complete the checkout process, add a view template called `Completed`. By convention, it must go into the `WebUI` project's `/Views/Cart` folder, because it will be rendered by an action on `CartController`. So, right-click `/Views/Cart`, choose Add ➤ View, enter the view name `Completed`, make sure "Create a strongly typed view" is *unchecked* (because we're not going to render any model data), and then click Add.

All you need to add to the view template is a bit of static HTML:

```
<asp:Content ContentPlaceHolderID="TitleContent" runat="server">
    SportsStore : Order Submitted
</asp:Content>
<asp:Content ContentPlaceHolderID="MainContent" runat="server">
    <h2>Thanks!</h2>
    Thanks for placing your order. We'll ship your goods as soon as possible.
</asp:Content>
```

Now you can go through the whole process of selecting products and checking out. When

Figure 5-16. *Completing an order*

Implementing the EmailOrderSubmitter

All that remains now is to replace `FakeOrderSubmitter` with a real implementation of `IOrderSubmitter`. You could write one that logs the order in your database, alerts the site administrator by SMS, and wakes up a little robot that collects and dispatches the products from your warehouse, but that's a task for another day. For now, how about one that simply sends the order details by e-mail? Add `EmailOrderSubmitter` to the `Services` folder inside your `DomainModel` project:

```
public class EmailOrderSubmitter : IOrderSubmitter
{
    const string MailSubject = "New order submitted!";
    string smtpServer, mailFrom, mailTo;
    public EmailOrderSubmitter(string smtpServer, string mailFrom, string mailTo)
    {
        // Receive parameters from IoC container
        this.smtpServer = smtpServer;
        this.mailFrom = mailFrom;
        this.mailTo = mailTo;
    }

    public void SubmitOrder(Cart cart)
    {
        // Prepare the message body
        StringBuilder body = new StringBuilder();
        body.AppendLine("A new order has been submitted");
        body.AppendLine("---");
        body.AppendLine("Items:");
        foreach (var line in cart.Lines)
        {
            var subtotal = line.Product.Price * line.Quantity;
            body.AppendFormat("{0} x {1} (subtotal: {2:c}", line.Quantity,
                                                            line.Product.Name,
                                                            subtotal);
```

```
            body.AppendFormat("Total order value: {0:c}", cart.ComputeTotalValue());
            body.AppendLine("---");
            body.AppendLine("Ship to:");
            body.AppendLine(cart.ShippingDetails.Name);
            body.AppendLine(cart.ShippingDetails.Line1);
            body.AppendLine(cart.ShippingDetails.Line2 ?? "");
            body.AppendLine(cart.ShippingDetails.Line3 ?? "");
            body.AppendLine(cart.ShippingDetails.City);
            body.AppendLine(cart.ShippingDetails.State ?? "");
            body.AppendLine(cart.ShippingDetails.Country);
            body.AppendLine(cart.ShippingDetails.Zip);
            body.AppendLine("---");
            body.AppendFormat("Gift wrap: {0}",
                cart.ShippingDetails.GiftWrap ? "Yes" : "No");

            // Dispatch the email
            SmtpClient smtpClient = new SmtpClient(smtpServer);
            smtpClient.Send(new MailMessage(mailFrom, mailTo, MailSubject,
                                                    body.ToString()));
        }
    }
```

To register this with your IoC container, update the node in your `web.config` file that specifies the implementation of `IOrderSubmitter`:

```
<component id="OrderSubmitter"
           service="DomainModel.Services.IOrderSubmitter, DomainModel"
           type="DomainModel.Services.EmailOrderSubmitter, DomainModel">
  <parameters>
    <smtpServer>127.0.0.1</smtpServer> <!-- Your server here -->
    <mailFrom>sportsstore@example.com</mailFrom>
    <mailTo>admin@example.com</mailTo>
  </parameters>
</component>
```

Exercise: Credit Card Processing

If you're feeling ready for a challenge, try this. Most e-commerce sites involve credit card processing, but almost every implementation is different. The API varies according to which payment processing gateway you sign up with. So, given this abstract service:

```
public interface ICreditCardProcessor
{
    TransactionResult TakePayment(CreditCard card, decimal amount);
}
```



```
public class CreditCard
{
    public string CardNumber { get; set; }
    public string CardholderName { get; set; }
    public string ExpiryDate { get; set; }
    public string SecurityCode { get; set; }
}

public enum TransactionResult
{
    Success, CardNumberInvalid, CardExpired, TransactionDeclined
}
```

can you enhance `CartController` to work with it? This will involve several steps:

- Updating `CartController`'s constructor to receive an `ICreditCardProcessor` instance.
- Updating `/Views/Cart/CheckOut.aspx` to prompt the customer for card details.
- Updating `CartController`'s POST-handling `CheckOut` action to send those card details to the `ICreditCardProcessor`. If the transaction fails, you'll need to display a suitable message and *not* submit the order to `IOrderSubmitter`.

This underlines the strengths of component-oriented architecture and IoC. You can design, implement, and validate `CartController`'s credit card–processing behavior with unit tests, without having to open a web browser and without needing any concrete implementation of `ICreditCardProcessor` (just set up a mock instance). When you want to run it in a browser, implement some kind of `FakeCreditCardProcessor` and attach it to your IoC container using `web.config`. If you're inclined, you can create one or more implementations that wrap real-world credit card processor APIs, and switch between them just by editing your `web.config` file.

Summary

You've virtually completed the public-facing portion of SportsStore. It's probably not enough to seriously worry Amazon shareholders, but you've got a product catalog browsable by category and page, a neat little shopping cart, and a simple checkout process.

The well-separated architecture means you can easily change the behavior of any application piece (e.g., what happens when an order is submitted, or the definition of a valid shipping address) in one obvious place without worrying about inconsistencies or subtle indirect consequences. You could easily change your database schema without having to change the rest of the application (just change the LINQ to SQL mappings). There's pretty good unit test coverage, too, so you'll be able to see if you break anything.

In the next chapter, you'll complete the whole application by adding catalog management (i.e., CRUD) features for administrators, including the ability to upload, store, and display product images.