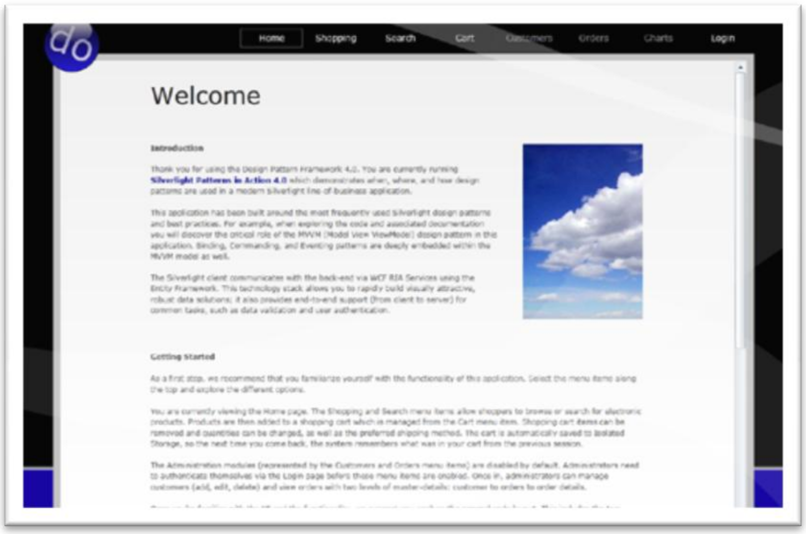
Silverlight Patterns

Reference Application

for .NET 4.0



Welcome

Introduction

Thank you for using the Design Pattern Framework 4.0. You are currently running **Silverlight Patterns in Action 4.0** which demonstrates when, where, and how design patterns are used in a modern Silverlight line of business application.

This application has been built around the most frequently used Silverlight design patterns and best practices. For example, when exploring the code and associated documentation you will discover the critical role of the MVVM (Model View ViewModel) design pattern in this application. Binding, Commanding, and Eventing patterns are deeply embedded within the MVVM model as well.

The Silverlight client communicates with the back-end via WCF RIA Services using the Entity Framework. This technology stack allows you to rapidly build visually attractive, robust data solutions; it also provides end-to-end support (from client to server) for common tasks, such as data validation and user authentication.

Getting Started

As a first step, we recommend that you familiarize yourself with the functionality of this application. Select the menu items along the top and explore the different options.

You are currently viewing the Home page. The Shopping and Search menu items allow shoppers to browse or search for electronic products. Products are then added to a shopping cart which is managed from the Cart menu item. Shopping cart items can be removed and quantities can be changed, as well as the preferred shipping method. The cart is automatically saved to Isolated Storage, so the next time you come back, the system remembers what was in your cart from the previous session.

The Administration modules (represented by the Customers and Orders menu items) are disabled by default. Administrators need to authenticate themselves via the Login page before these menu items are enabled. Once in, administrators can manage customers (add, edit, delete) and view orders with two levels of master-details: customer to orders to order details.

[illegible]

Companion document to
Design Pattern Framework™ 4.0

by

Data & Object Factory, LLC
www.dofactory.com

Index

Introduction

The *Silverlight Patterns 4.0* reference application is designed to demonstrate the use of design patterns and best practices in building RIA (Rich Internet Access) data-centric, business applications using Silverlight. This release is optimized for .NET 4.0 and takes advantage of numerous 4.0 features and enhancements, including Silverlight 4, WCF RIA Services, Entity Framework, and MEF (Managed Extension Framework).

The *Silverlight Patterns 4.0* reference application is separate from the larger *Patterns in Action 4.0* application for two reasons:

1) It's built on a different architecture, and
2) It requires that you have Silverlight 4 Tools for Visual Studio 2010 installed (download from Microsoft) and the Silverlight 4 Toolkit (download from CodePlex).

Even though *Silverlight Patterns 4.0* and associated documentation (this document) are separate, it is assumed you are familiar with concepts discussed in *Patterns in Action 4.0*. These applications have a lot in common*,* including the application's functionality (shopping, cart, and administration), its use of membership, the databases, the notion of layering, numerous design patterns, and more. Some of these concepts will be touched upon here, but not in-depth.

The focus *of Silverlight Patterns 4.0* is on Silverlight-specific patterns and practices. Many other patterns, as featured in *Patterns in Action 4.0* would be useful in this application, but are not included. Example include: logging, Data Transfer Objects (DTO), multiple data providers, and SOA. This is just something to keep in mind.

Running for the first time

When opening the Silverlight solution in Visual Studio 2010 for the first time, you may see some errors and warnings. Some of the assemblies are not created yet and there are bi-directional dependencies between the projects (discussed later). Simply start a full re-build and they will all go away. If they don't be sure you have the required Silverlight SDK and Toolkit installed.

Finding your way

Silverlight Patterns 4.0 comes with several sources of documentation. First, there is this document, the one you are currently reading. It is named *Silverlight Patterns 4.0.pdf*. This is the main document that will guide you through the setup, architecture, and patterns used in the application.

Also, the application code itself is well commented. Each class has XML comments with <summary> information, many with additional <remarks>. Most public and private members (methods, properties, etc) have comments as well.

As mentioned, the Silverlight application has a lot in common with the apps in *Patterns in Action 4.0* solution and therefore the *Patterns in Action 4.0.pdf* document is also helpful to have around.

About this document

The best way to read this document is from beginning to end. Each section builds on the previous one and it is best to follow along in a linear fashion. This document contains six sections:

Application Functionality: This section briefly reviews the functionality (i.e. the end-user experience) of the application.

Architecting for Silverlight: In this section we will review how architecting robust Silverlight applications is different from, say, building a web application. You will learn that building *stateful* apps is hip again.

Silverlight Solution and Projects: This section looks at the *Silverlight Patterns 4.0* Solution and its projects. You will see how the project structure clearly delineates the 2-tier architecture. We will also discuss how to setup and organize your own projects that use RIA services and Entity Framework on the back-end.

MVVM Patterns & Practices: This section describes in detail the core design pattern in this application: MVVM (Model, View, ViewModel). Critical sub-patterns, including eventing, commanding, data binding, and visual state management are also reviewed.

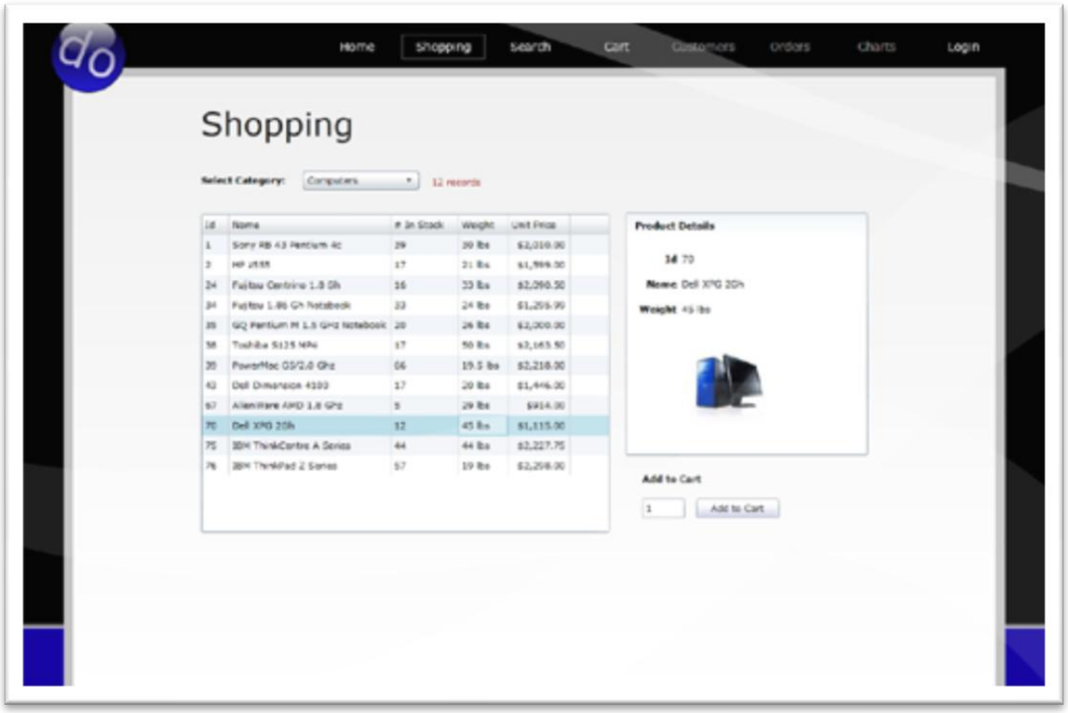
MEF Patterns & Practices: This section describes how MEF (Managed Extensibility Framework) is used to build a composable set of tab pages including charting. It also reviews a pattern that is commonly found when composing applications.

Pattern Fabric: This section briefly touches upon the numerous patterns you find in almost any Silverlight application.

Application Functionality

Here we briefly review the end-user's experience for this application. It is suggested that you explore the running application as well.

Silverlight Patterns 4.0 is a data-centric, e-commerce application in which shoppers search and browse a catalog of electronic products. Products are organized by category. Users select products, view their details, and add these to their shopping carts.



Shopping

Select Category: Computers 12 records

Id	Name	# In Stock	Weight	Unit Price
1	Sony RB 43 Pentium 4c	29	39 lbs	$2,310.00
2	HP z555	17	21 lbs	$1,999.00
24	Fujitsu Centrino 1.8 Gh	16	33 lbs	$2,090.50
34	Fujitsu 1.86 Gh Notebook	33	24 lbs	$1,255.99
35	GQ Pentium M 1.5 GHz Notebook	20	26 lbs	$2,000.00
38	Toshiba S125 MP4	17	50 lbs	$2,163.50
39	PowerMac G5/2.0 Ghz	06	19.5 lbs	$2,218.00
42	Dell Dimension 4100	17	20 lbs	$1,446.00
67	AlienWare AMD 1.8 GHz	5	29 lbs	$914.00
70	Dell XPG 20h	12	45 lbs	$1,115.00
75	IBM ThinkCentre A Series	44	44 lbs	$2,227.75
76	IBM ThinkPad Z Series	57	19 lbs	$2,258.00

Product Details

Id 70

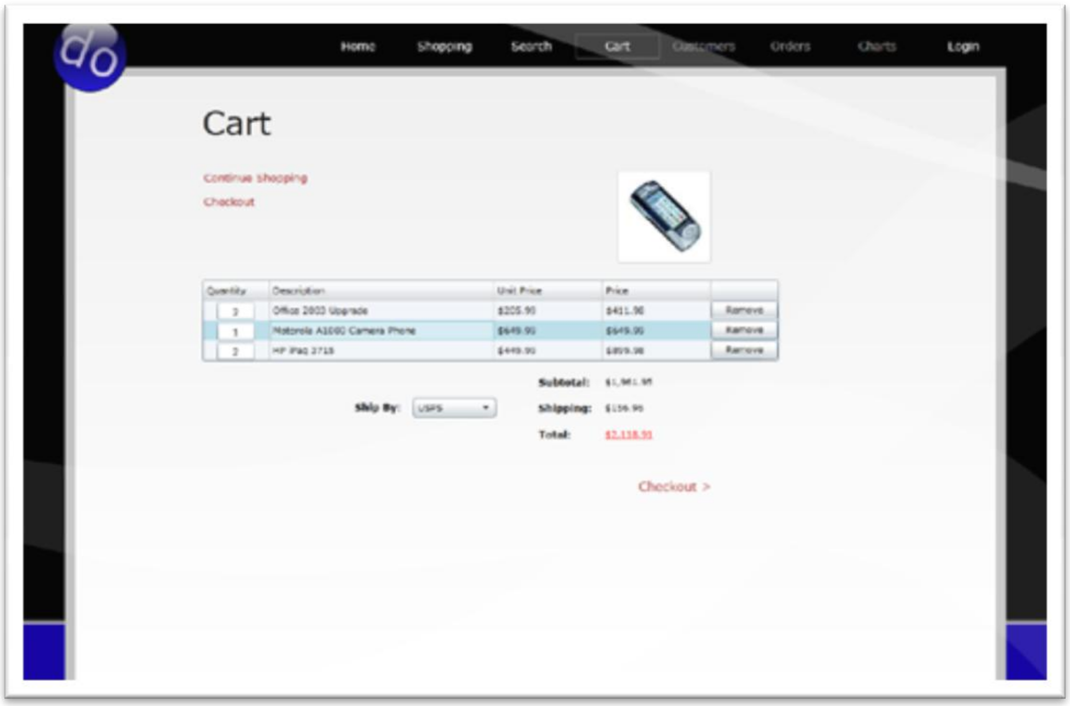
Name Dell XPG 20h

Weight 45 lbs

Add to Cart

1 Add to Cart

Shopping carts are managed by changing quantities and removing items. Shipping costs are computed based on the shipping method selected.



Cart

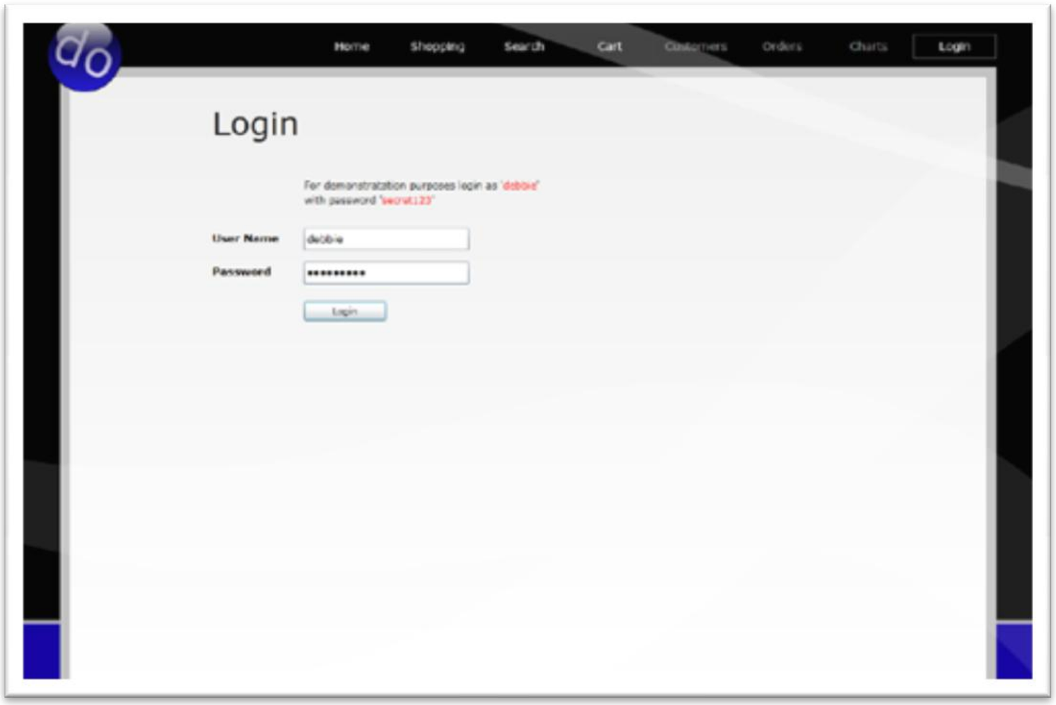
Quantity	Description	Unit Price	Price
2	Office 2003 Upgrade	$205.95	$411.90
1	Motorola A1000 Camera Phone	$649.95	$649.95
2	HP iPaq 2718	$449.95	$899.98

Ship By: USPS

Subtotal: $1,961.95
Shipping: $156.95
Total: $2,118.91

When changing entries in the shopping cart, notice how fast the page updates occur. The reason is that all cart information is stored and re-calculated on the client and, so, there are no server trips involved. The updates are so fast, that it is easy to miss them. For this reason we've added an animation effect to the red total amount, to confirm to the shopper that this figure was indeed adjusted.

The administrative area is represented by the Customers, Orders, and Charts menu items which are disabled by default. A valid login is required to activate the menus. The administrator logs in via the Login menu with the same credentials as used in all other projects: username debbie and password secret123.



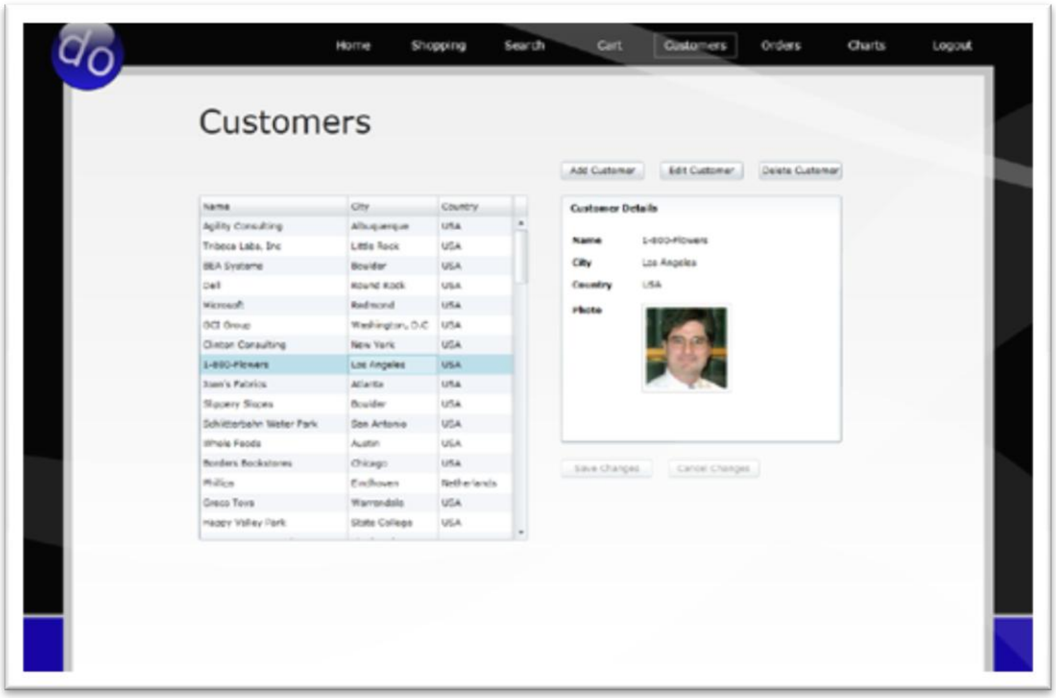
Login

For demonstration purposes login as 'debbie' with password 'secret123'

User Name: debbie

Password: ●●●●●●●●●

Following login, customers can be managed (adding, editing, and deleting) from the Customers menu. The Order details can be searched, viewed and analyzed from the Orders menu. Analytical charts are visible via the Charts menu.

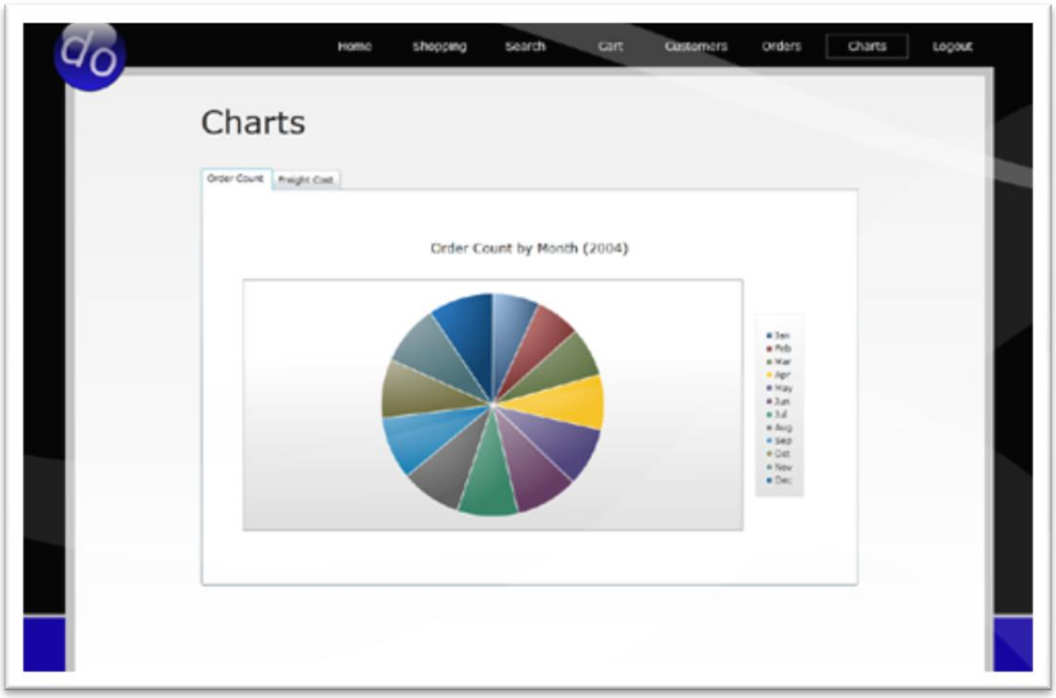


Customers

Name	City	Country
Agility Consulting	Albuquerque	USA
Tribeca Labs, Inc	Little Rock	USA
BEA Systems	Boulder	USA
Dell	Round Rock	USA
Microsoft	Redmond	USA
GCI Group	Washington, D.C	USA
Clinton Consulting	New York	USA
1-800-Flowers	Los Angeles	USA
Joan's Fabrics	Atlanta	USA
Slippery Slopes	Boulder	USA
Schlitterbahn Water Park	San Antonio	USA
Whole Foods	Austin	USA
Borders Bookstores	Chicago	USA
Phillips	Eindhoven	Netherlands
Greco Toys	Warrendale	USA
Happy Valley Park	State College	USA

Customer Details

Name	1-800-Flowers
City	Los Angeles
Country	USA
Photo	



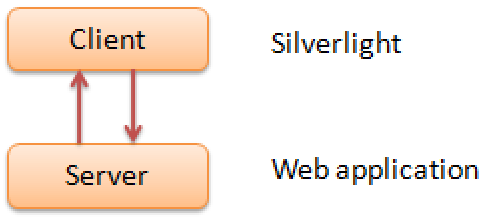
Charts

Architecting for Silverlight

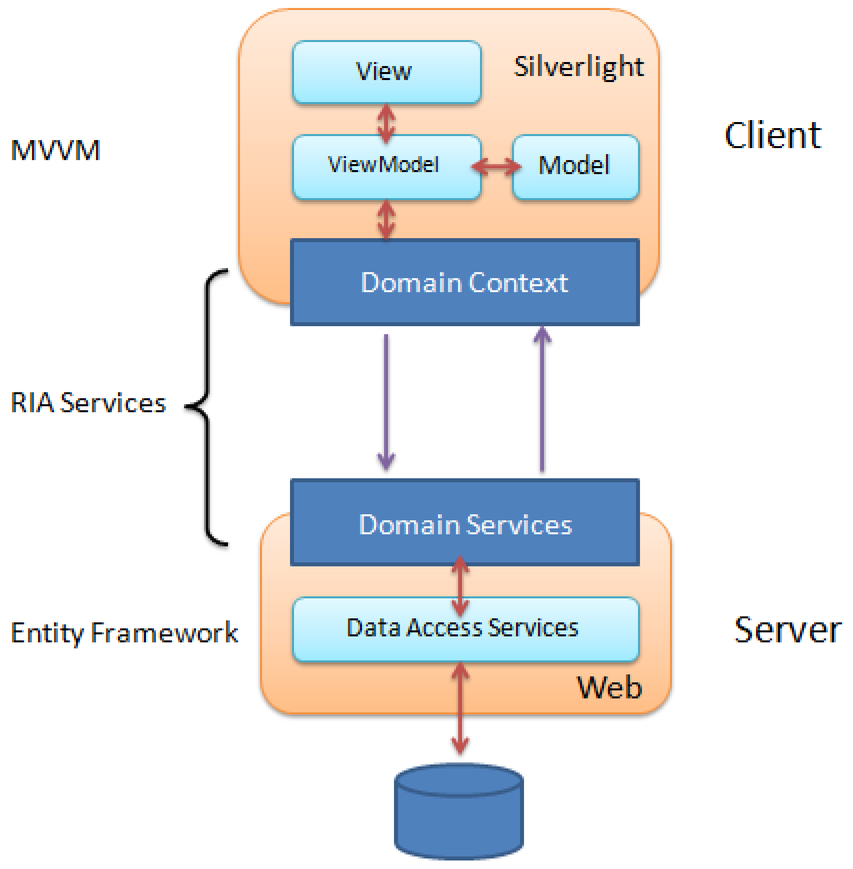
When comparing the *Silverlight Patterns 4.0* application with in *Patterns in Action 4.0* you see strong functional similarities: both support users who are shopping for electronic products and an administrative area to maintain customer and view orders. Also, both use the exact same data and databases (both 'action' db and membership db) as described in *Patterns in Action 4.0*.

But the similarities end there. The underlying *architecture* is very different. Whereas *Patterns in Action 4.0* is built on a 3-tier model, this application follows a more traditional 2-tier client-server model. At a high-level we see a client tier (the Silverlight app) and a server tier (the Web app).

Here is a sketch:



Of course, this is only a high-level view. Each layer has its own set of sub-layers and components and their categorization is open for debate (for example, some will say that the RIA services is a middle tier, etc). Below is a more detailed view of the same architecture.



At the client we have Model, View, and ViewModel components (this is the MVVM pattern discussed later). Of these, only the ViewModel communicates with the server via the Domain Context. The middle section of Domain Context, Domain Services and the way they communicate via WCF comprises the RIA Services. Data Access Services on the server side are implemented with the Entity Framework. At the very bottom we have the database.

Compared to many n-tier architectures, this looks fairly simple. However, building robust, data-centric Silverlight business applications is not necessarily easy. Silverlight itself has a learning curve; the central MVVM design pattern takes time to fully understand, WCF RIA Services are new, MEF is relatively new (if you choose to use it), and you have to understand how to build a solid back-end data access module to support many concurrent users (we use the Entity Framework, which is a large piece software in and of itself).

Stateless versus Stateful

When building web applications, you know that web pages are short lived. Every time the user requests a page they get a new one. Never do they get the same page back. Silverlight is different. The web page that holds the application is loaded once and for the duration of the application, Silverlight with associated assemblies (xap files) stay in memory in your browser.

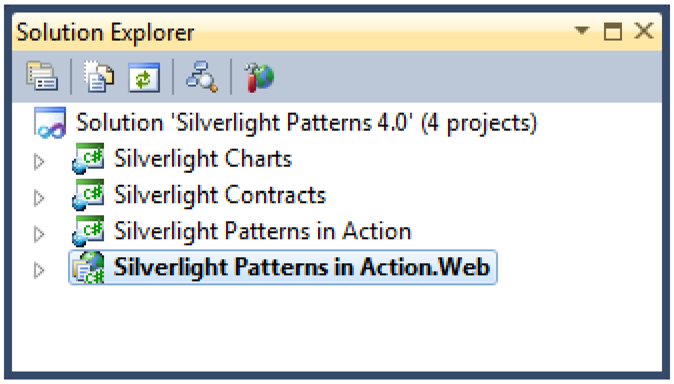
This stay-in-memory model requires that you think differently about *state*. With Silverlight you can reduce the number of server trips by being *stateful* (i.e. maintaining state between page transitions). So, maintaining *state* is a good thing! This is different from Web Applications where many .NET architects instruct their developers to be *stateless* by avoiding the use of Viewstate and Session as much as possible.

Please remember that maintaining state *only* applies to the Silverlight client. On the server side (ASP.NET) nothing has changed and you should continue building *stateless* operations. If necessary, you can still use Session on the server, because Silverlight shares cookies with the browser window.

Silverlight Solution and Projects

This section discusses the *Silverlight Patterns 4.0* Solution and Projects.

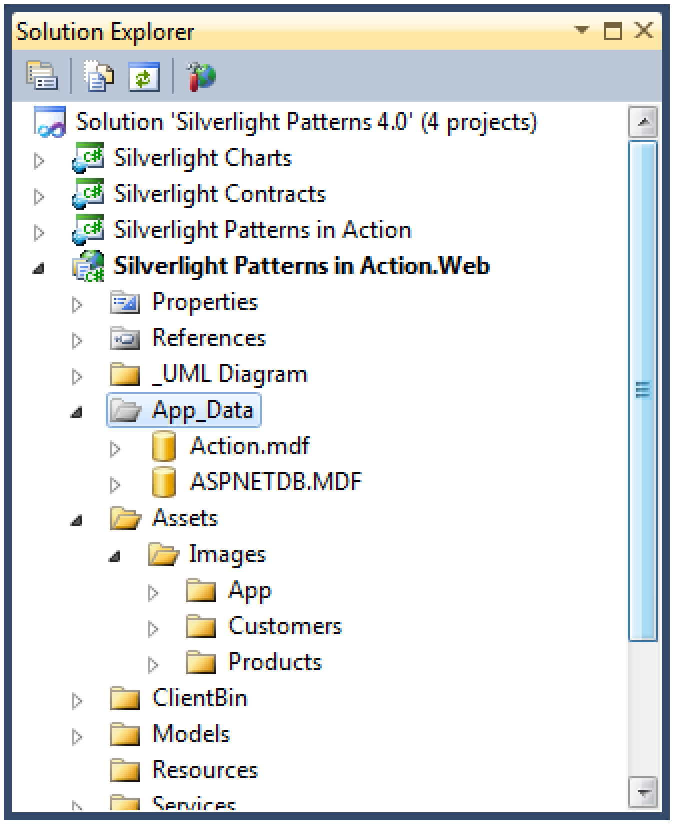
The solution resides in its own folder named \Silverlight Patterns. There are four projects. The two main projects are named 'Silverlight Patterns in Action' and 'Silverlight Patterns in Action.Web'; a *Silverlight* project and a *Web* project respectively. Two supporting projects are *Silverlight Charts* and *Silverlight Contracts* which demonstrate MEF practices. To run the application ensure that **Silverlight Patterns in Action.Web** is your startup project (in bold, see below)



Solution 'Silverlight Patterns 4.0' (4 projects)
- Silverlight Charts
- Silverlight Contracts
- Silverlight Patterns in Action
- **Silverlight Patterns in Action.Web**

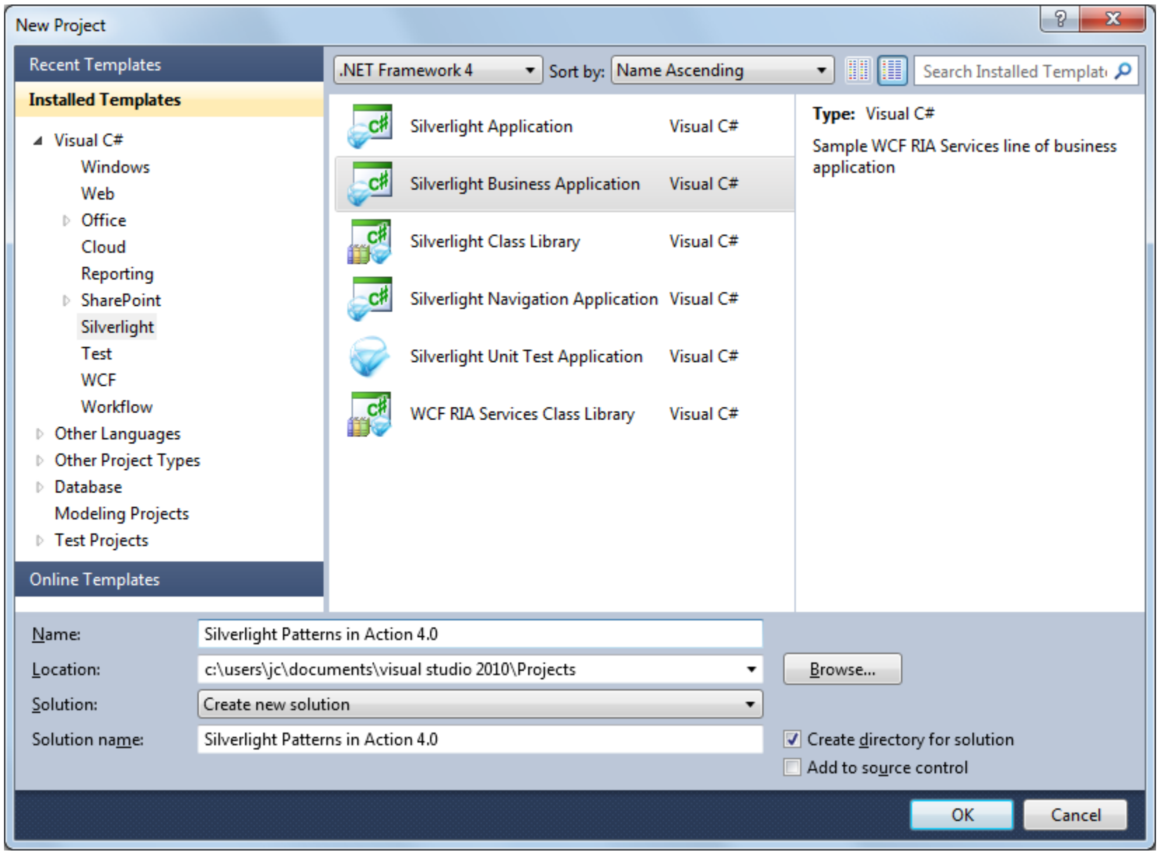
The application is configured to use two SQL Express databases located under the \App_Data folder. They are the same as used in *Patterns in Action 4.0* and are named *Action.mdf* and *Aspnetdb.mdf*. If you choose to run against SQL Server you need to create the appropriate databases and modify two items in web.config under the Web project: the ActionEntities connection string and add a <membership> section. For details consult the documentation for *Patterns in Action 4.0*.

The product and customer Images are stored locally (as opposed to *Patterns in Action 4.0* which uses an image hosting service).



Building your own application:

The easiest way to start a Silverlight data-centric application with WCF RIA Services is by selecting the *Silverlight Business Application* project template.



.NET Framework 4 Sort by: Name Ascending

Silverlight Application	Visual C#
Silverlight Business Application	Visual C#
Silverlight Class Library	Visual C#
Silverlight Navigation Application	Visual C#
Silverlight Unit Test Application	Visual C#
WCF RIA Services Class Library	Visual C#

Type: Visual C#
Sample WCF RIA Services line of business application

Name: Silverlight Patterns in Action 4.0
Location: c:\users\jc\documents\visual studio 2010\Projects
Solution: Create new solution
Solution name: Silverlight Patterns in Action 4.0
[✓] Create directory for solution
[ ] Add to source control

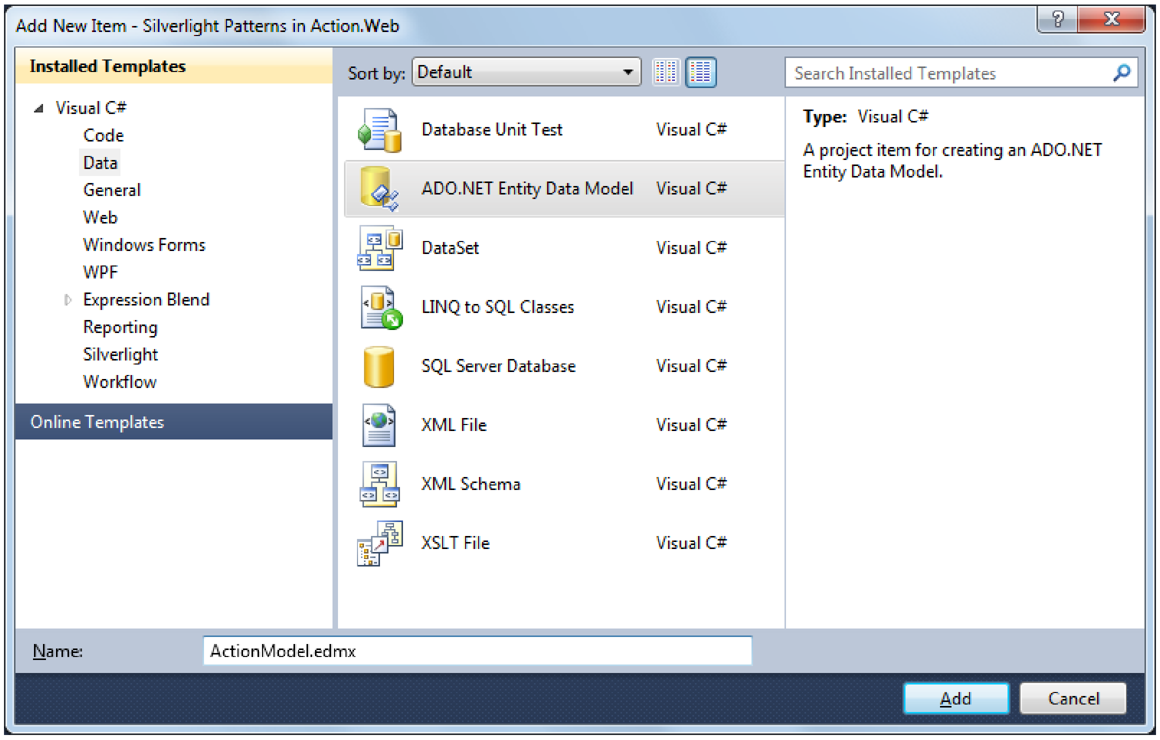
This will create two projects: a Silverlight client project and a Web server project.

The *Silverlight Business Application* project template provides a lot of boiler-plate code, including Membership services (including user authentication and authorization) and a Silverlight paging infrastructure. If your requirements include these services they will make building your application a lot faster.

In *Silverlight Patterns 4.0* we removed the unnecessary code, including several Silverlight pages and User Registration services. We kept the Silverlight Navigation and the User Authentication services. In your own project you should also remove any unnecessary code.

The *Silverlight Business Application* project template creates several links and associations between the two projects, including a RIA Services link, a Silverlight ClientBin link, and linked resources. Details of these links are discussed later in this section.

Now that you have 2 projects, you can add the WCF RIA Services on the Web project. Add a new Item and select the ADO.NET Entity Data Model: In *Silverlight Patterns* 4.0 we use the Entity Framework for data access, but any other data access technology can be used.



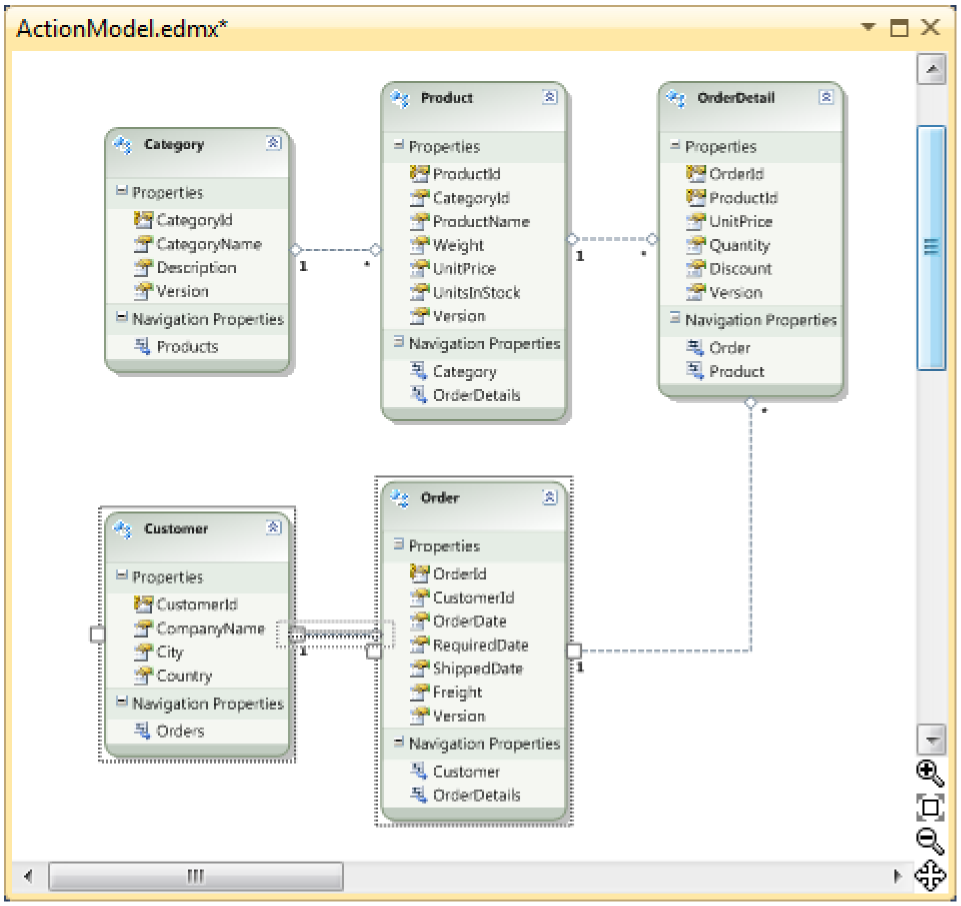
Database Unit Test	Visual C#
ADO.NET Entity Data Model	Visual C#
DataSet	Visual C#
LINQ to SQL Classes	Visual C#
SQL Server Database	Visual C#
XML File	Visual C#
XML Schema	Visual C#
XSLT File	Visual C#

Type: Visual C#

A project item for creating an ADO.NET Entity Data Model.

Name: ActionModel.edmx

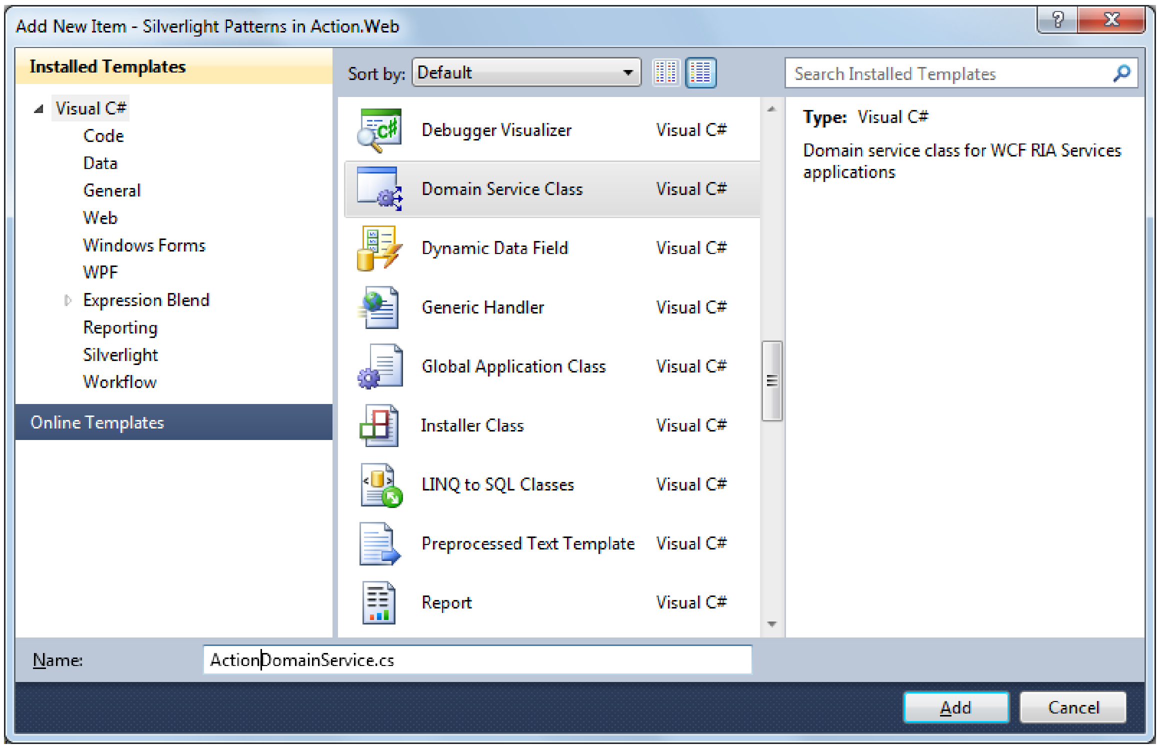
Select all tables in the *action.mdf* database and the following entity model is created.



We named our model: ActionModel.edmx.

It is entirely possible to use POCOs (Plain Old CLR Objects) as your preferred business object model. However, when you do make sure the POCO classes have some notion of identity, which you do by adding the [Key] metadata attribute to one or more properties in your class. A discussion of POCOs with RIA Services is beyond the scope of this reference application.

Next, add a new Domain Service item under the \Services folder.



Debugger Visualizer	Visual C#
Domain Service Class	Visual C#
Dynamic Data Field	Visual C#
Generic Handler	Visual C#
Global Application Class	Visual C#
Installer Class	Visual C#
LINQ to SQL Classes	Visual C#
Preprocessed Text Template	Visual C#
Report	Visual C#

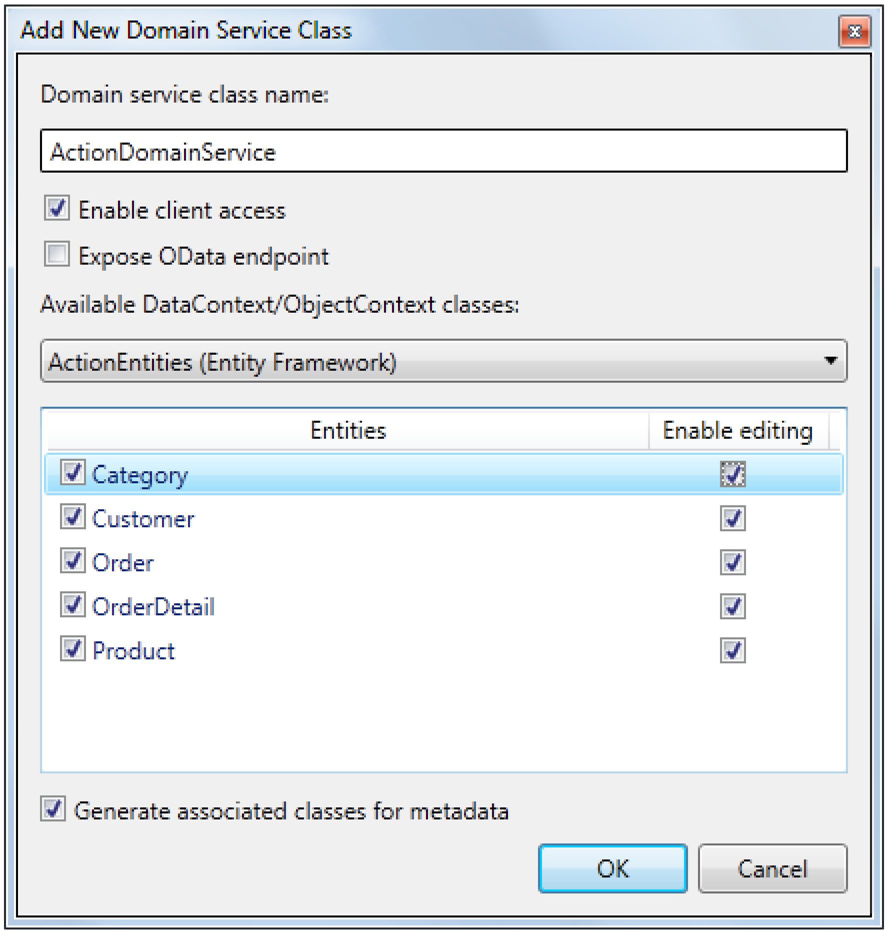
Type: Visual C#

Domain service class for WCF RIA Services applications

Name: ActionDomainService.cs

We named ours: ActionDomainService.cs.

When adding the service, you will be prompted with a Domains Service class dialog box. Select the following details and click OK.



Add New Domain Service Class

Domain service class name: ActionDomainService

[✓] Enable client access

[ ] Expose OData endpoint

Available DataContext/ObjectContext classes: ActionEntities (Entity Framework)

Entities	Enable editing
✓ Category	✓
✓ Customer	✓
✓ Order	✓
✓ OrderDetail	✓
✓ Product	✓

[✓] Generate associated classes for metadata

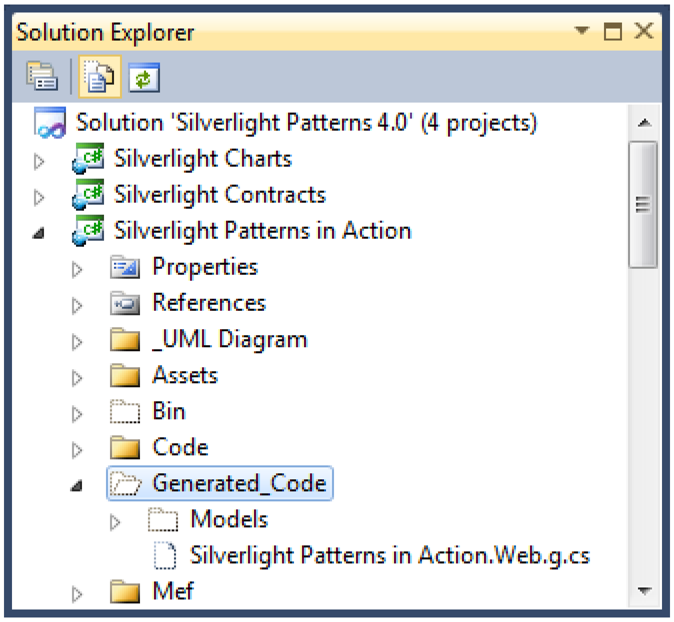
Now that the domain service has been created, let's explore some of the code. First, open ActionDomainService.cs under the \Services folder. The [EnableClientAccess] attribute at the top signals to the compiler that these services are to be made available to the client.

Microsoft uses a naming convention for the different domain service methods that include easy-to-remember words like Get, Insert, Update, and Delete. For example the Customers data access methods are GetCustomer, InsertCustomer, UpdateCustomer, etc. If, for some reason, you need to deviate from this convention, you can still indicate your intentions with special attributes, such as [Insert], [Update], [Delete], [Invoke], [Query] etc.

In *Silverlight Patterns 4.0* we follow the code generated names, although we did add the [Invoke] and [RequiresAuthentication] attributes. [Invoke] signifies that a method is a simple data fetch (as opposed to a [Query] attribute which involves entities being loaded into an Entity Container). The [RequiresAuthentication] attribute is super handy and allows only Authenticated users to execute the class or method.

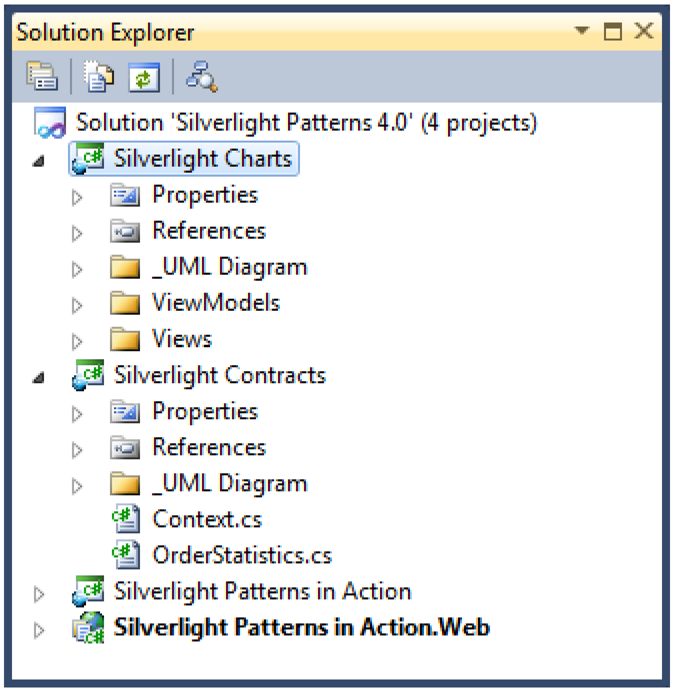
Next, open the generated file named ActionDomainService.metadata.cs. It has the metadata for the RIA service entities. This is the place where you add data annotation property attributes that are used in data validation, such as, [Required], [StringLength(50)], and [Regular Expression ("[A-Z]*")]. Their names are self-explanatory. For the *Silverlight Patterns 4.0* we added a couple [Include] attributes, to indicate to the RIA Services serializer that the referenced member is needed on the client and should be populated.

Now, build (or rebuild) the Solution. This will automatically code-generate some files on the client side in the Silverlight project. They are located under a hidden folder named \Generated_Coder To view this folder select the Silverlight project and click the 'Show all Files' icon (on top of the Solution Explorer). You will now see the \Generated_Code folder with its contents (see image below).



Making the server-side RIA Services available to the client is called *projecting the services* onto the Silverlight project. This is an automatic process and any changes you make in the ActionDomainService.cs will automatically be available to Silverlight (following a compile).

Two other projects in the *Silverlight Patterns 4.0* solution are *Silverlight Charts* and *Silverlight Contracts*. These demonstrate how you use MEF (Managed Extension Framework) to dynamically include a couple of custom charts as plugins. The Silverlight Charts project has two custom UserControls with charts. The Silverlight Contracts project shares the Import/Export MEF contracts supported by the application.



Building from scratch

We described how to build a Silverlight business application using the *Silverlight Business Application* project template. Not only does this template provide a lot of starter code, it also configures the project relationships and links that are required for these kinds of applications. Many development teams may choose to organize their solution in multiple projects, and it is important that you know how to manually create these project dependency links. An overview follows:

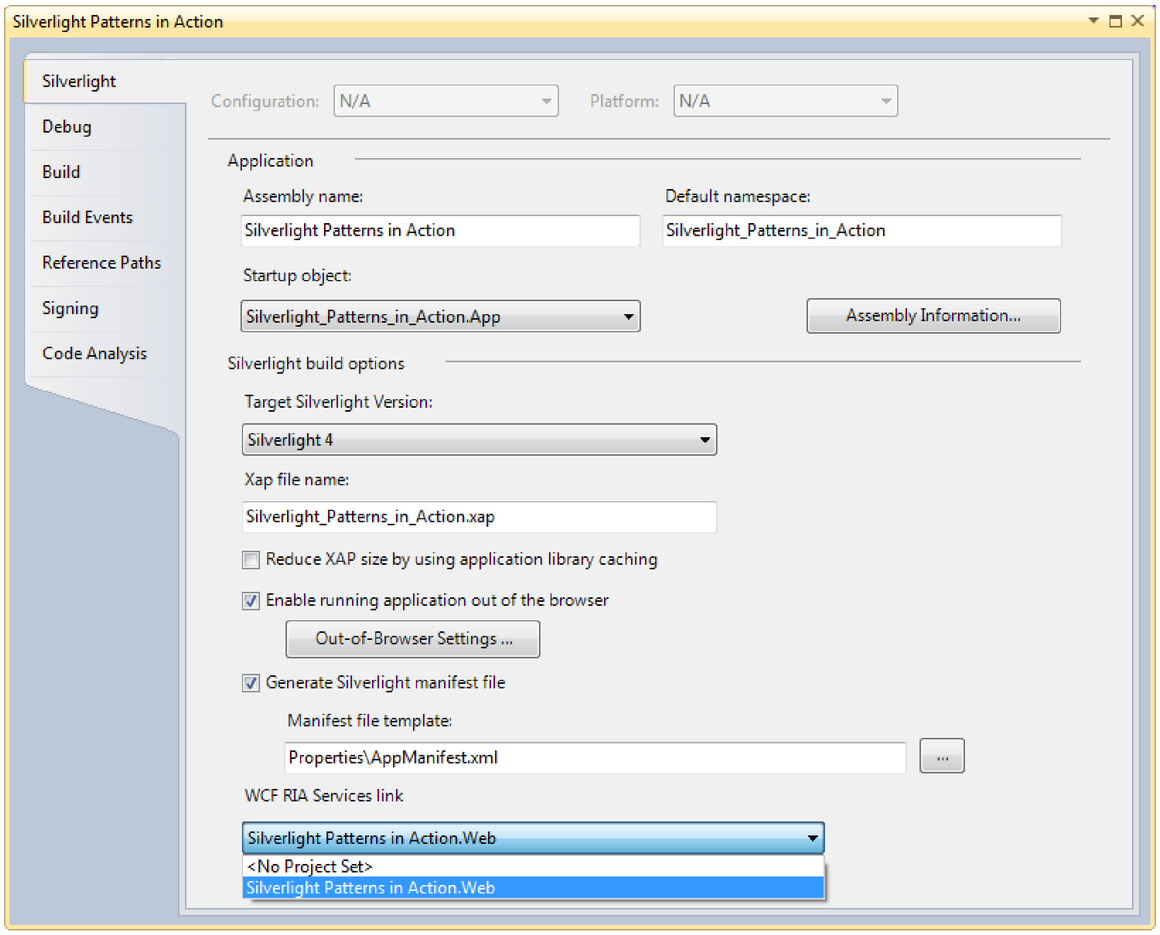
To review, here are the links that are automatically built by the Silverlight Business application project template:

1. RIA Services link
2. Silverlight ClientBin link
3. Linked Resources

We will now discuss how they are configured manually.

RIA Services link

The RIA Services link is managed on the Silverlight project properties page under the ‘Silverlight’ tab. You get to this page by right clicking on the Silverlight project and select the Properties menu item. Once this link is established any changes in the RIA Services will be propagated automatically (following a compile) to the Silverlight application. The Silverlight tab with the WCF RIA Services link is depicted below.



Configuration: N/A
Platform: N/A

Application

Assembly name: Silverlight Patterns in Action
Default namespace: Silverlight_Patterns_in_Action
Startup object: Silverlight_Patterns_in_Action.App

Silverlight build options

Target Silverlight Version: Silverlight 4
Xap file name: Silverlight_Patterns_in_Action.xap

[ ] Reduce XAP size by using application library caching
[✓] Enable running application out of the browser
[✓] Generate Silverlight manifest file

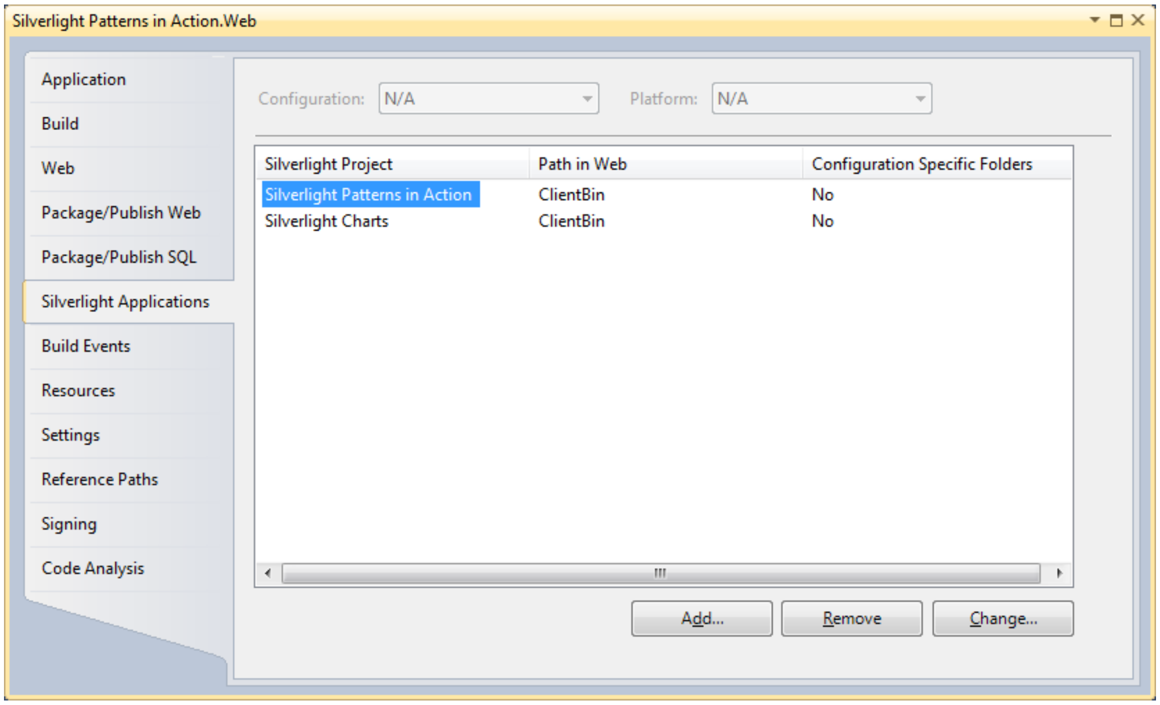
Manifest file template: Properties\AppManifest.xml

WCF RIA Services link: Silverlight Patterns in Action.Web

<No Project Set>
Silverlight Patterns in Action.Web

Silverlight ClientBin link

The web application requires a copy of the Silverlight xap file into its ClientBin folder (as well as a refresh every time the Silverlight app has changed). To configure this, right click the Web application, select Properties and open the Silverlight Applications tab. At the bottom of this tab are the Add, Remove, and Delete buttons. See next screenshot.

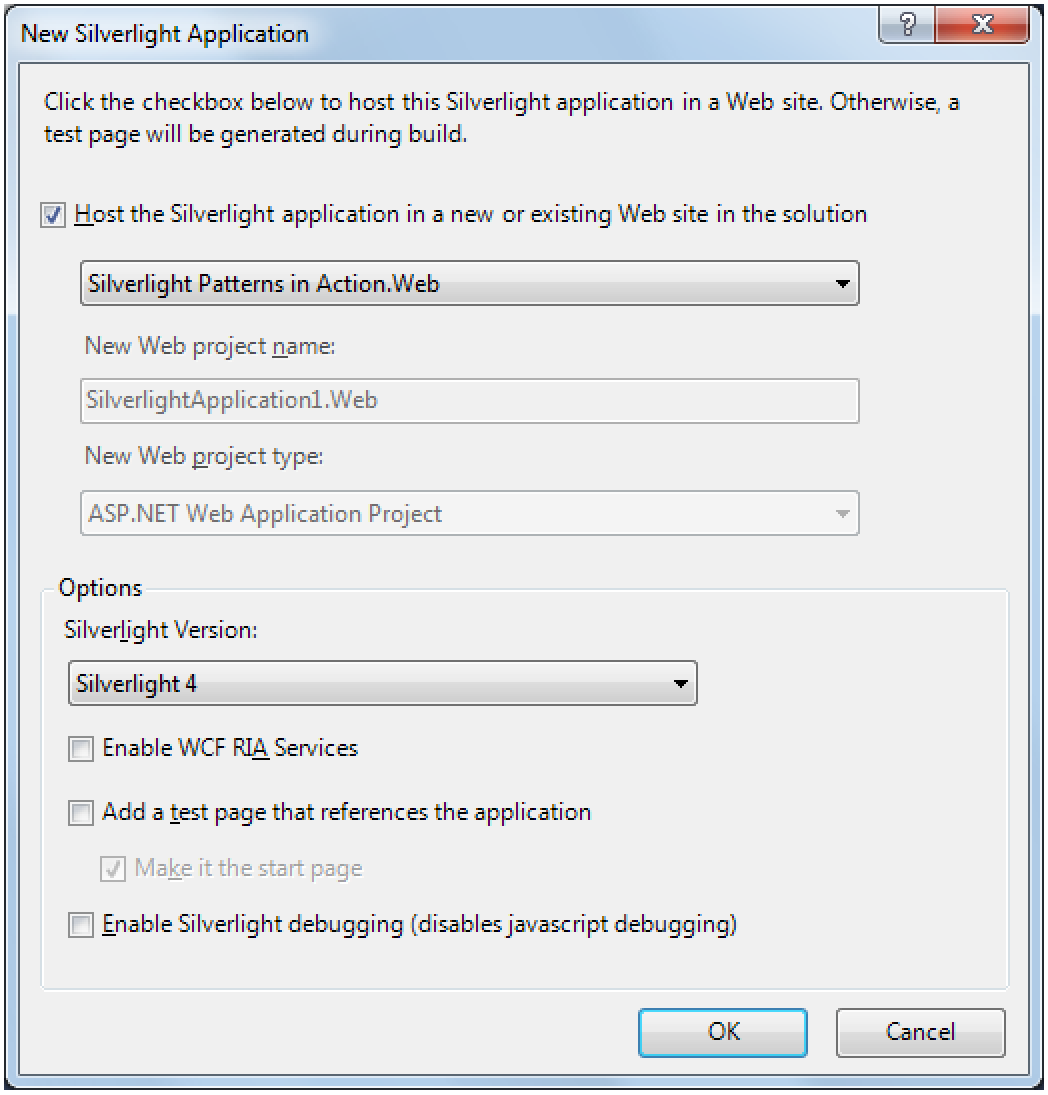


Configuration: N/A Platform: N/A

Silverlight Project	Path in Web	Configuration Specific Folders
Silverlight Patterns in Action	ClientBin	No
Silverlight Charts	ClientBin	No

This screenshot shows two xap files, one for the main Silverlight app and one (Silverlight Charts) in support of MEF. The application has implemented MEF in such a way that it requires the plug-in xap file to reside in the same ClientBin folder.

When building Silverlight plug-in projects (like our Silverlight Charts project), here is how you configure this. Create a new Silverlight Application project and tell it to host the application in the existing web project (see screenshot below). This will create the ClientBin link and copy the xap file ClientBin. By the way, the newly created Silverlight app will probably not need the App.xaml and MainPage.xaml so you can remove these.


New Silverlight Application

Click the checkbox below to host this Silverlight application in a Web site. Otherwise, a test page will be generated during build.

[✓] Host the Silverlight application in a new or existing Web site in the solution

Silverlight Patterns in Action.Web

New Web project name:

SilverlightApplication1.Web

New Web project type:

ASP.NET Web Application Project

Options

Silverlight Version:

Silverlight 4

[ ] Enable WCF RIA Services

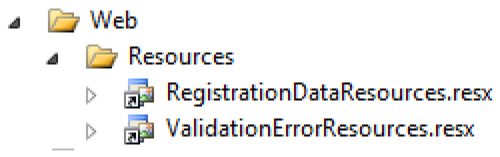
[ ] Add a test page that references the application

[✓] Make it the start page

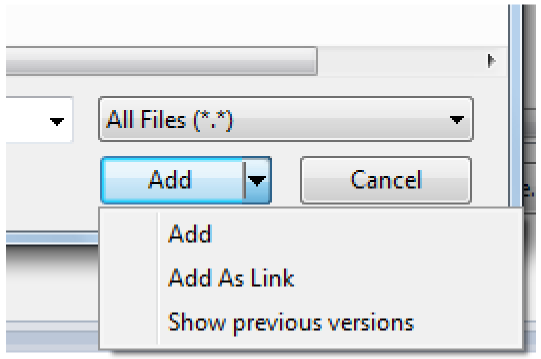
[ ] Enable Silverlight debugging (disables javascript debugging)

Linked Resources

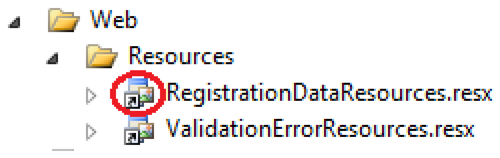
For a consistent set of message strings and labels between the client and server, some of the embedded resource files are shared through a process that links client and server files. Essentially, the client and the server work of the same set of resources. When creating a Silverlight Business Application out of the box, you will find two linked resources under the \Web\Resources folder. Both on the client and the server these files are configured to compile as an 'embedded resource'.



Creating a link to another file is done as follows: right click on the folder in which you need the linked file. Then select: Add an Existing Item. On the file selection dialog select the file in the Web project and then, instead of Add, select the dropdown and select 'Add as Link', like below:



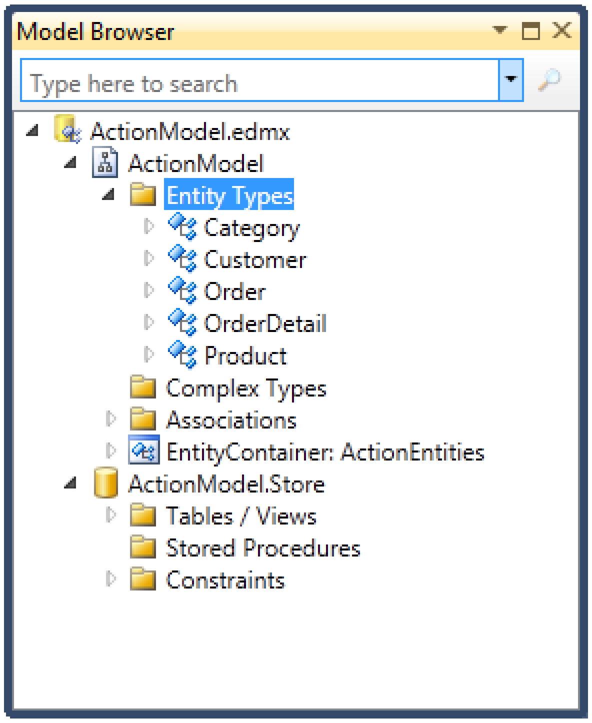
Linked files are recognized by a small link arrow symbol on their icons.



The Web Server

Let's review some details of the Web Server project. Folders you see are \App_Data with the database files, \Assets with application, customer, and product images, \ClientBin with the Silverlight xap files, \Models with auto-generated User classes used with Membership, \Services with Domain Services and Membership's Authentication services, and the Entity Framework's ActionModel.edmx file. Default.aspx is the web page that hosts the Silverlight application.

Open ActionModel.edmx and you'll see that the entities map one-to-one to the tables in the database. Below is a screenshot of the Model browser.



```
ActionModel.edmx
  ActionModel
    Entity Types
      Category
      Customer
      Order
      OrderDetail
      Product
    Complex Types
    Associations
    EntityContainer: ActionEntities
  ActionModel.Store
    Tables / Views
    Stored Procedures
    Constraints
```

Although we have a simple mapping, the Entity Framework can handle far more complex mappings. In fact, it has a special mapping layer that is designed for situations in which entities map to two or more tables. All subsequent CRUD operations (Insert, Update, etc) are then automatically handled by this mapping layer. Complex entity-table mappings is where the Entity Framework shines and it is particularly useful in situations where you must access a legacy database with a data-model that is established and cannot be changed (a common situation in large organizations).

The Silverlight Client

Here are some details of the Silverlight client project. Folders included are: an \Assets folder with string resources and an important *Styles.xaml* file, a \Code folder with numerous utility classes, a \Mef folder with code to support MEF, a \ViewModels folder with several ViewModel classes, and the \Views folder with *xaml* pages and associated code behind files.

As mentioned earlier, entities and domain services are made available to the Silverlight client under a hidden folder named \Generated_Code. As a developer there is no need to reference or open these files.

MainPage.xaml is the main navigation container with header and menu items. Many of the styles in this file use styles that are defined in *Styles.xaml*. This file has been customized to reflect the blue-ish look-and-feel of *Silverlight Patterns 4.0*. Styles.xaml is referenced in App.xaml like so:

```
<ResourceDictionary Source="Assets/Styles.xaml"/>
```

The styling is based on the “Aurora” theme for Silverlight Business Applications which is freely available from the Microsoft Expression website.

When starting the application, you will notice that three menus are disabled: Customers, Orders, and Charts. Open MainPage.xaml and you’ll see why that is. The IsEnabled attribute of these menus are bound to a property called AdminLoggedIn

```
IsEnabled="{Binding AdminLoggedIn}"
```

When AdminLoggedIn is false the menus are disabled, but as soon as the administrator logs in they are enabled. AdminLoggedIn lives on a ViewModel class and this brings us to the next section where we discuss ViewModels as part of the MVVM design pattern and related sub-patterns.

MVVM (Model, View, ViewModel)

MVVM stands for Model – View – ViewModel and was first documented a few years ago by Microsoft. They have since used it extensively to build their own WPF applications, including Visual Studio 2010 and the Expression suite of products. *Silverlight Patterns 4.0* also uses it throughout the application.

MVVM is similar to MVC (Model View Controller) and MVP (Model View Presenter). If you've used either one you will have little difficulty learning MVVM. MVVM is highly specialized pattern that uses modern UI features that are currently available in WPF & Silverlight only, including *Binding*, *Commanding*, and *Visual State Management*.

The Model in MVVM represents the data-side of the application, including business objects. The View refers to the user interface including the elements the user interacts with, such as, pages, menus, buttons, tabs, lists, etc. The ViewModel is an abstraction of the View and provides data and action related information to the View in a loosely coupled way using data binding, commanding, and visual state management.

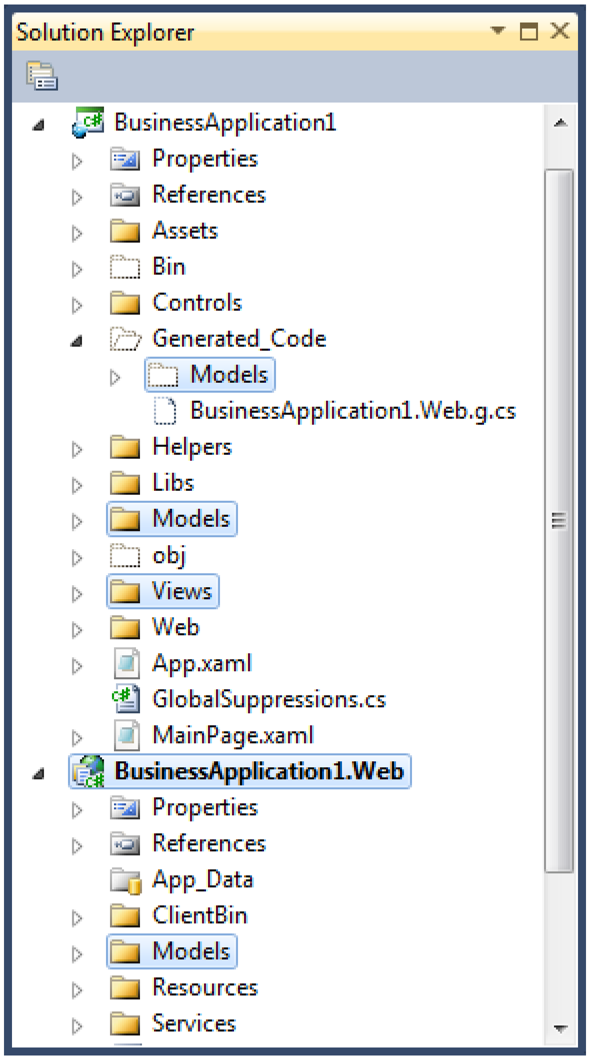
Like all MV patterns MVVM is about separation of concerns. Let the View do what it does best (presenting an interactive user experience) and let the ViewModel handle the communication between the View and the Model (the business objects).

Next, we'll review the relationships among the MVVM components. The ViewModel is the intermediary between the Model and the View. The View and the Model never communicate directly with one another. The ViewModel retrieves and updates data from the data store by communicating with the Model directly. However, the ViewModel itself is not aware of the View and never maintains a reference to the View (this is different from MVP in which the Presenter does have a View reference). The View simply waits for signals (through events, commands, binding notifications, etc) from the ViewModel that something has changed and then the UI will respond accordingly.

When employing MVVM you will find there is little or no code in the code-behind. Most of it ends up in the ViewModel. Please note that the View is very much aware of the ViewModel as it is its only source of data and actionable notifications.

In the Design Pattern Framework (this package) we demonstrate MVVM in WPF and MVVM in Silverlight in. You will find there are some implementation differences, primarily because of a feature gap between the two platforms. Even so, the fundamentals of the MVVM pattern are the same in either platform.

When starting a new Silverlight Business Application project you will immediately see an early hint of the MVVM pattern by the \Models and \Views folders throughout the solution. Just add a ViewModels folder and you are ready to build out MVVM. See image below.



With MVVM, each View typically has its own dedicated ViewModel. The ViewModel prepares the data so that it can be displayed in a format that the View requires. ViewModels are regular classes that implement the INotifyPropertyChanged interface which notifies clients that a property value has changed. In *Silverlight Patterns* 4.0 an abstract ViewModelBase class implements this interface and is the ancestor class to all ViewModels.

In *Silverlight Patterns 4.0* you will find that each View has its own ViewModel. There are two ViewModels share some code; the ShoppingViewModel and SearchViewModel. They have the 'Add to Shopping' functionality in common and so they have a common ancestor class called AddToCartViewModel. It handles the Add to Cart aspect on their respective pages. The Home and Checkout pages have no functionality and so these views have no ViewModel.

The LoginViewModel supports user authentication. This ViewModel is available globally and during the entire lifetime of the application by adding it to the Resources Dictionary in App.xaml. When the application starts an instance of this ViewModel class is created. Similarly, the CartViewModel, which represents the shopping cart, is available globally and for the duration of the application. Here is the application resource dictionary.

```
<ResourceDictionary>
    <app:ResourceWrapper x:Key="ResourceWrapper" />

    <!-- login view model -->

    <viewmodel:LoginViewModel x:Key="MyLoginViewModel" />

    <!-- Shopping cart view model -->
    <viewmodel:CartViewModel x:Key="MyCartViewModel" />

    <!-- converters -->
    <converters:CustomerImageConverter x:Key="MyCustomerImageConverter" />
    <converters:ProductImageConverter x:Key="MyProductImageConverter" />

</ResourceDictionary>
```

Next we will discuss the 4 sub-patterns in MVVM: Commanding, Binding, Visual State Management, and Eventing.

MVVM commanding sub-pattern

Commanding has always been available in WPF, but was only recently added to Silverlight 4 with the addition of the Command property to ButtonBase and Hyperlink. Commanding lets controls know when their actions are available or not. If not, the controls are disabled.

For example, when adding a new customer, you cannot also delete a customer, therefore, while adding, the Delete button needs to be disabled. The Command pattern is designed to help in these kinds of scenarios.

Commanding is based on a simple interface: *ICommand*. It has 3 members,

```
interface ICommand
{
    void Execute(object parameter);
    bool CanExecute(object parameter);
    event EventHandler CanExecuteChanged;
}
```

Commands are found on controls (button, menu item, etc) that can issue a command, i.e. execute something. This is what the Execute method is for. The CanExecute and CanExecuteChanged members work hand in hand. To know when the Execute option changes, the UI control listens to the CanExecuteChanged event. When this fires it checks the boolean CanExecute method. When CanExecute returns false, the UI control disables.

An optional CommandParameter can be added to these controls. Its value is passed as a parameter to the Execute and CanExecute methods.

Let's look at the CustomerViewModel for an example. This class exposes five *ICommand* members named: AddCommand, EditCommand, DeleteCommand, SaveCommand, and DeleteCommand. They will inform the View, whether these actions are available and, when selected, actually execute the action. So, for the Add Customer button, it will determine when it is enabled or disabled, and when clicked it will execute the Execute the action (which, in turn, is passed on to the OnAdd action handler).

All *ICommand* interfaces are wrapped in a custom RelayCommand class which makes it easy to enable and disable commands from the ViewModel itself. Other MVVM implementations sometimes use a similar class named DelegateCommand which is slightly more generic (it uses a delegate method for the CanExecute method), but we don't need that here. RelayCommand takes the action handler as an argument to its constructor. These are the private OnAdd, OnEdit, etc methods found in the ViewModel.

Take the OnSave action handler as an example. It is called when the page is in Add or Edit mode and the user has just modified a customer record. First is raises the Saving event, so that the View can respond (which ends and commits the edit operation), then SubmitChanges is called which tells the RIA Service to send any changes to the database (Add, Edit, Delete). SubmitChanges is an Asynchronous method and a callback is included in the argument list. Then the ViewState's Saved event is triggered which gives the View another chance to respond. In the SubmitChanges callback method the results are evaluated. The results are then presented to the user view via a textual Status property, which we will discuss next when reviewing the binding sub-pattern.

MVVM binding sub-pattern

In Silverlight *binding* is everywhere. The dependency properties make binding extremely convenient and powerful. Here we will focus on how binding facilitates MVVM.

Let's start off with the CustomerViewModel. Open Customers.xaml and see that CustomerViewModel has been added to the page's Resource Dictionary.

```
<navigation:Page.Resources>
    <viewmodel:CustomerViewModel  x:Key="MyCustomerViewModel" />
</navigation:Page.Resources>
```

Silverlight will create an instance of this ViewModel (which is possible because the ViewModel has a default constructor with no arguments) and makes it available as a resource. Next, the ViewModel is bound to the DataContext of LayoutRoot Grid, like so:

```
<Grid x:Name="LayoutRoot"
      DataContext="{Binding Source={StaticResource MyCustomerViewModel}}" >
```

This makes the ViewModel available to all LayoutRoot child elements which can then bind to any of the public properties on the ViewModel without referencing a Source. This is demonstrated by the ViewModel *Status* property which is a simple status message string. At the bottom of the page you will see how it is bound to a TextBlock where it binds to the ViewModel without stating Source (and without 'Path=' because Path is the default). This creates a nice and concise binding expression:

```
<TextBlock Name="TextBlockStatus"  Foreground="Red"
       Text="{Binding Status}" Margin="0,20" />
```

Go up a few lines and see the Command binding in the Save button: again, very concise:

```
    <Button Name="ButtonSave" Content="Save Changes"
Command="{Binding SaveCommand}" />
```

Further up we see how the DataGrid's ItemsSource binds to the Customers collection on the ViewModel. When Silverlight displays the DataGrid it iterates over this collection and each of the DataGrid columns binds to the current instance of the Customer. An example is the Name column which binds to the CompanyName value for each Customer, like so:

```
Text="{Binding CompanyName}"
```

The SelectedItem property of the DataGrid is bound to the CurrentCustomer property: of the ViewModel:

```
ItemsSource="{Binding Customers}"
SelectedItem="{Binding CurrentCustomer, Mode=TwoWay}" >
```

Notice that the Mode is *TwoWay*, meaning the ViewModel can update the View, but the View can also update the ViewModel (i.e. when the user selects a row).

This page also has an example of *element-to-element* binding, meaning View-to-View rather than View-to-ViewModel. Look at the DataForm where the Current Item is bound to the SelectedItem property in the DataGrid:

```
CurrentItem="{Binding ElementName=DataGridCustomers,
                      Path=SelectedItem, Mode=TwoWay}"
```

Sometimes you need a value converter to format the data in the XAML. The Image control on the page demonstrates binding the Image source to the CustomerId property using a Converter and a ConverterParameter.

```
<Image Height="100" Width="100"
       Source="{Binding CustomerId,
                Converter={StaticResource MyCustomerImageConverter},
                ConverterParameter=Large}"  />
```

Lastly, open the Cart.xaml page where we like to point out the Remove button. You will see that the page's DataContext is set to the CartViewModel (which is initialized in the applications static resource, in App.xaml). This makes the shopping cart and shopping cart items available to this page. The DataGrid in this page is bound to CartItems, a collection of shopping cart line items. The Remove command is interesting. It states:

```
<Button Content="Remove" Width="80"
        Command="{Binding Source={StaticResource MyCartViewModel},
                          Path=RemoveCommand}"
                          CommandParameter="{Binding}">
</Button>
```

The DataContext of each cart row is the CartItem class. However, the RemoveCommand is on the ViewModel, which is why the Source="" needs to be specified. The Path is the RemoveCommand on the ViewModel. Notice the CommandParameter which is bound to the entire DataContext at that moment, which is the CartItem itself. Now, when clicking the Remove button, the CartItem is passed as a parameter into the OnRemove method and we know exactly which item to remove.

Understanding what goes on is sometimes tricky, but it demonstrates also how powerful and flexible Silverlight databinding really is.

MVVM visual state management sub-pattern

A common question that comes up with MVVM is how to integrate animations in a way that works well with the MVVM pattern. Animation storyboards are usually defined in XAML and are initiated from the View. This would require that the ViewModel have a reference to the View, but that would be incorrect as this goes against the separation of concern principle in MVVM. Here we solve this problem with *Visual State Management* and *Attached Properties*.

This approach is used in the shopping cart page. Open Cart.xaml and find the VisualStateManager (VSM). The target of the animation is the color of the TextBlock that displays the total cost of the items in the shopping cart. Animation was added because the cart changes so fast that the user may easily miss the changed value. We have two states: Normal and Changed, and a transition of .8 seconds. The xaml is listed below:

```
<VisualStateManager.VisualStateGroups>
    <VisualStateGroup x:Name="CommonStates">

        <VisualStateGroup.Transitions>
            <VisualTransition From="Normal" To="Changed"
                              GeneratedDuration="0:0:0.8"  />
        </VisualStateGroup.Transitions>

        <VisualState x:Name="Changed">
            <Storyboard>
                <ColorAnimation
                     Storyboard.TargetName="textBlockTotal"
                     Storyboard.TargetProperty="(ForeGround).(SolidColorBrush.Color)"
                     From="White" To="Red" Duration="0:0:0" >
                </ColorAnimation>
            </Storyboard>
        </VisualState>
        <VisualState x:Name="Normal"/>

    </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

To change states using the VSM requires that the view be passed as an argument, like so:

```
VisualStateManager.GoToState(view, stateName, true);
```

But with the MVVM pattern the ViewModel does not know about the View. The solution lies with *Attached Properties*. We implemented a helper class with an attached property named VisualState. This property is then attached to the View and in the OnVisualStateChanged callback method the sender is passed (which will be the view) and the new property value (which will be the name of the state). The VSM can then go ahead and call GoToState with the correct arguments.

The helper class is named ShoppingCartStates and is referenced in the Cart.xaml class. Just below this reference you see the DataContext for the page being set to the ViewModel. CurrentCartState is a property on the ViewModel. Here are the relevant lines:

```
viewmodel:ShoppingCartStates.VisualState="{Binding CurrentCartState}"
DataContext="{StaticResource MvCartViewModel}"
```

Run the cart page and you will see the animation effect of the red total amount (the dollar amount at the bottom right of the page).

An advantage of using Visual State Management over other methods is that it integrates well with Blend. The designer who uses Blend does not have to know anything about programming and will be able to completely define the visual effects and transitions between the states using Blend only.

MVVM eventing sub-pattern

An alternative to using the VisualStateManager is to listen to ViewModel events. This is implement by using event handlers in the View's code behind which attach to events on the ViewModel. From an MVVM separation of concern perspective this is no problem because the ViewModel does not know about the View that is listening to the events being fired.

Now open the Login code behind page. In the constructor you see that 3 event handlers are attached to the LoginViewModel. They respond to LoggingIn, LoggedIn, and LoginFailed events. These event handlers will update the view depending on the event being fired.

Open the LoginViewModel and confirm it exposes three public events. Events are raised at the appropriate times by calling RaiseEvent (which is defined in ViewModelBase). We use a common ViewModelEventArgs argument (derived from CancelEventsArgs) which allows the user to cancel an event from the View. The cancel option is used only when deleting a Customer. Of course, in your own applications you may use EventArgument classes that are more event-specific.

Observable items

When programming in Silverlight you are likely to encounter various Observable items, such as the built-in ObservableCollection or custom-built Observable Objects, and sometimes even Observable Commands. Many developers create their own custom Observable classes usually with an ‘Observable’ prefix. What they have in common is that they implement the *INotifyPropertyChanged* and/or the *INotifyCollectionChanged* interfaces, so that they can be ‘observed’, that is, they will notify the observers when any of their properties have changed (this is the observer design pattern).

In Silverlight Patterns 4.0 we have one such custom Observable collection called ObservableCollectionEx. It is used in the shopping Cart page. It derives from ObservableCollection and each item is an object that implements the INotifyPropertyChanged. The Attach and Detach methods are a way to avoid memory leaks and maintain event handlers for each of the items whenever the collection as a whole changes. This collection supports the removal of shopping cart items and changes in the quantities of the shopping cart. When a user makes any of these changes, the ViewModel recalculates the totals for the entire shopping cart.

Isolated Storage

The shopping cart makes use of Silverlight's *Isolated Storage*. Isolated storage has been called 'cookies on steroids' and allows the application to store up to 1Meg of data in a hidden folder on the client machine.

Open the CartViewModel and locate these methods: SaveCartToLocalStorage and LoadCartFromLocalStorage. Both these methods use a helper class named IsolatedStorageHelper that facilitates the reading and writing to a shopping cart file. The entire cart is stored when a change occurs in the cart. When the application starts, the saved cart is re-loaded (in the constructor of CartViewModel). So, when you come back the next day, you will find that the shopping cart is the same when you left. Using local storage explains why the shopping cart page is so incredibly fast; everything stays on the client and there is no need to contact the server.

MEF (Managed Extensibility Framework)

MEF simplifies the creation of extensible applications (by using plug-ins). It offers discovery and composition capabilities that you can use to dynamically load application extensions (i.e. plugins). The MEF *catalog* is responsible for locating extensions. Once located, the MEF *CompositionContainer* will coordinate the creation of the plug-ins and satisfy dependencies.

MEF Composition

A core concept in MEF is the *composable part*. In your code (application and plug-ins) you tag your composable parts with attributes: [Import] for parts that need services, and [Export] on the ones that offer services. MEF will find the appropriate parts, and coordinate and satisfy these dependencies.

Let's explain this with an example in *Silverlight Patterns 4.0*. MEF is used in the Search page where it locates and initializes SearchViewModel. First, notice that the SearchViewModel class has been tagged with an [Export] attribute (of type SearchViewModel).

```
[PartCreationPolicy(CreationPolicy.NonShared)]
[Export(typeof(SearchViewModel))]
public class SearchViewModel : AddToCartViewModel
{
```

In the Search Page, which consumes the SearchViewModel we find a matching [Import] attribute. MEF will match Import and Export tags and coordinate the creation of SearchViewModel and then assign it to the MefSearchViewModel property. See code below.

```
public partial class Search : Page
{
    [Import(typeof(SearchViewModel))]
    public SearchViewModel MefSearchViewModel { get; set; }

    public Search()
    {
        InitializeComponent();

        CompositionInitializer.SatisfyImports(this);

        // Set data context of page to imported viewmodel.
        DataContext = MefSearchViewModel;

        MefSearchViewModel.AddingToCart += viewModel_AddingToCart;
        MefSearchViewModel.AddedToCartFailed += viewModel_AddedToCartFailed;
        MefSearchViewModel.AddedToCart += viewModel_AddedToCart;

        // Set focus on textbox product name
        Dispatcher.BeginInvoke(() => { textBoxProductName.Focus(); });

    }
```

The initial MEF discovery step takes place at application startup (discussed next), but you see in the Search page's constructor that the CompositionInitializer resolves the dependencies for this page with a call to SatisfyImports. After this the MefSearchViewModel is initialized and ready to go. The code then sets the ViewModel to the view's DataContext and several event handlers are attached to it as well.

Run the Search page: search for an item, add it to the cart, and then come back to search for more; the page behaves as you would expect.

Now open the SearchViewModel and change the PartCreationPolicy attribute from *NonShared* to *Shared*. Run again. Now, notice that when you add a product to the cart and come back to the Search page it is the same when you left. To start a new Search, you need to select the 'reset' link. Using the Shared setting, MEF creates just one instance (Singleton) of the SearchViewModel and keeps it in memory until the application ends. So MEF not only manage the creation but also the lifetime of the parts. This page is a nice demonstration of how the Shared and NonShared settings can make a big difference. The example also highlights how in Silverlight *state* can be maintained between page transitions.

MEF Discovery

The MEF discovery of composable parts takes place in application startup. Open file App.xaml.cs and view the InitializeContainer method.

```
private void InitializeContainer()
{
    var catalog = new AggregateCatalog();

    // Add this assembly to list of catalogs
    catalog.Catalogs.Add(new DeploymentCatalog());

    // Add charting assembly to list of catalogs
    var uri = new Uri("SilverlightCharts.xap", UriKind.Relative);
    var chartCatalog = new DeploymentCatalog(uri);
    chartCatalog.DownloadCompleted += catalog_DownloadCompleted;
    catalog.Catalogs.Add(chartCatalog);

    // Perform part composition.
    CompositionHost.Initialize(catalog);

    // Asynchronously download charts imports
    chartCatalog.DownloadAsync();
}
```

First, an aggregate catalog is created. Aggregate catalogs combine multiple catalogs with composable parts. The first deployment catalog added is from the running assembly itself. It locates all [Import] and [Export] attributes in the executing Silverlight application.

Next, it locates the composable parts of a plug-in by the name *SilverlightCharts.xap* and adds these to the aggregate catalog. CompositionHost.Initialize then initializes the compositioning process of the parts. Remember that these calls are made from the client but the plug-in files are located on the server; this is why the xap file catalog requires a final DownLoadAsync call.

The hard-coding of the *SilverlightCharts.xap* file is less-than desirable. It is probably more common to have a folder in which plug-in xap files are dropped and then have these dynamically discovered. However, there is no built-in support for this in MEF. You can write a routine to scan the folder and list all files, but remember that the request comes from the client and the plug-ins are located on the server. There are several ways to deal with this, including passing xap file names as parameters to the Silverlight control at startup time or by calling a WebService or RIA Service. Alternatively you can use a configuration file or database table from which plug-in xap file names are made available.

MEF Contracts

The SilverlightCharts.xap plug-in provides two user controls with charts that display analytical sales data grouped by month. One is a column chart that shows total freight expenses by month, and the other is a pie chart that shows total number of orders placed by month. Each exported chart consists of a View and a ViewModel.

Like other ViewModels these plug-in ViewModels also need to communicate with the RIA services. However, these services are not available to the Charts, so they need to be imported. Essentially, what the main Silverlight application is saying is, we will display any imported chart, but we handle the data access (RIA Services) side of things. With that we have a bi-directional dependency: the application needs to import charts and the charts need to import data access (exported by the application). To overcome circular assembly dependencies a separate contract project is created that defines the *contracts* between the assemblies.

The *Silverlight Contracts* project defines an IContext interface in which the RIA Services web context contract is defined. RIA Service calls are asynchronous so the consumer must pass in a callback to be notified when the call is ready. The IContext contract is shown below.

```
public interface IContext
{
    /// <summary>
    /// Retrieves order statistics asynchronously.
    /// </summary>
    /// <param name="callback">Callback for when retrieval is complete.</param>
    /// <param name="year">Year for which statistics is required.</param>
    void GetOrderStatistics(Action<List<OrderStatistics>> callback, int year);
}
```

Another class in this project is OrderStatistics. It returns the query results from the Context to the charts. The charts display the results by databinding to an observable collection of this type.

To see how MEF wires it all up see the Charts page code behind listed below.

```
public partial class Charts : Page, IPartImportsSatisfiedNotification
{
    /// <summary>
    /// The plugged in charts (these are imported by MEF).
    /// </summary>
    [ImportMany]
    public UserControl[] PluginCharts { get; set; }

    /// <summary>
    /// Constructor.
    /// </summary>
    public Charts()
    {
        InitializeComponent();

        CompositionInitializer.SatisfyImports(this);
    }

    /// <summary>
    /// Once importing is done, we add chart user controls to tab.
    /// </summary>
    public void OnImportsSatisfied()
    {
        tabCharts.Items.Clear();

        foreach (var chart in PluginCharts)
        {
            chart.Margin = new Thickness(20);

            var item = new TabItem();
            item.Header = chart.Tag;
            item.Content = chart;

            tabCharts.Items.Add(item);
        }
    }
}
```

The page implements IPartImportsSatisfiedNotification, which means that when parts composition is complete it will call OnImportsSatisfied. The UserControl array is tagged with [ImportMany] indicating that all available exports will be created and assigned to the array. The CompositionInitializer.SatisfyImports starts the process of resolving the dependencies. Next, the OnImportsSatisfied iterates over all imports and adds these as TabItems to a tab control on the page. In *Silverlight Patterns 4.0* two charts are discovered, resolved, and displayed.

MVVM – MEF pattern

MEF is about creating extensible applications using a mechanism called composition. Through discovery, of exportable and importable parts, you are able to dynamically load application extensions. Following the composition of parts, MEF will have created a tree-like structure that is ready to run the application.

This recursive, tree-like aspect of the composition has led us to observe the following pattern: *Exportable classes with importable members*. Here is a (hypothetical) example of this pattern in the MVVM-MEF realm

```
// View
[Export]
public partial class View : UserControl
{
    [Import]
    private ViewModel ViewModel { get; set; }
}

// View Model
[Export]
public class ViewModel
{
    [Import]
    private Context Context { get; set; }
    |
    [ImportMany(typeof(ICommand))]
    public IEnumerable<ICommand> Commands { get; set; }

    [Import(typeof(string))]
    public string Property {get; set;}
}

// Context
[Export]
public class Context
{
    // Does something..
}
```

We have an exportable View (or page), that is most likely imported by a Frame or App object higher up in the application. The View imports its ViewModel. The ViewModel itself is exported but imports other members, such as Context, Properties, and

Commands. The Context itself is exported and imports perhaps a data base connection, etc. This pattern allows for a lot of flexibility in building a dynamically composable application (Note that it is more common to import and export interfaces, rather than implementations).

Where would you use such a structure? Composition is mostly applicable to (larger) applications where you have a shell application that can import custom plug-ins. Visual Studio and Expression are good examples. Another scenario is where applications have frequent updates and changing requirements. Yet another scenario is where 3rd party developers build components that can plug-in and augment the parent application. Popular applications that offer flexibility and are easily extensible often create little cottage industries of plug-in developers (Excel and Photoshop are good examples). In fact, there reasoning should be the other way around: they have become so popular because they are highly extensible. In the Silverlight arena one potential candidate would be eBay's Simple Lister Application (although we are not aware of them offering MEF plug-ins).

Pattern Fabric

Pattern Fabric is a word that we coined. What we mean is that over the years, design patterns have evolved to a point where it is hard to separate patterns from software development in general. Patterns are advanced OO techniques and therefore are part of the developer's mind. They are also part of programming languages (iterators, events, etc), architectural design (layers, façade), and testing (IoC), essentially 'infiltrating' all aspects of software development. Because they are so pervasive we like to think that patterns are part of the fabric of software engineering, hence *Pattern Fabric*.

In *Silverlight Patterns 4.0* we have focused on Project structure, MVVM, RIA Services, and MEF, without repeating patterns and practices presented in *Patterns in Action 4.0*. This does not mean there are no GoF or Enterprise Patterns in the application. Actually, there are plenty; here are some examples:

- LINQ is at its core an **Iterator**, as it iterates over collections
- Silverlight's Visual Tree is a **Composite** of UIElements, The VisualTreeHelper recursively walks the composite tree
- The RIA Services' WebContext is a **Proxy** standing in for the Server side code
- The Domain Services in RIA Services represents the **Façade** pattern.
- The ICommand interface and **Command** property in Silverlight Buttons are implementations of the Command pattern (hence the name).
- Observable collections implement the **Observer** pattern (hence the name)
- In our entity model, the entities represent the **Active Record** pattern.
- The **Factory Method** in the application's ErrorWindow (open file ActionModel.Designer.cs and you find regions marked as Factory Methods).

There are probably many more but these come to mind. It confirms that patterns are very much part of the 'fabric' of software development.

Summary

Silverlight Patterns 4.0 is a reference application that demonstrates how to build a modern, robust Silverlight business application using design patterns and best practices. Silverlight and RIA Services are relatively new technologies, but we are hopeful that after studying the reference application you are convinced that design patterns and practices are fundamental to establishing a solid Silverlight application architecture. Design patterns and best practices help you architect and design simple, elegant, extensible, and easily maintainable applications that users are demanding

If you have questions on using design patterns, the architecture of *Silverlight Patterns 4.0*, or have suggestion for future enhancements and improvements please do not hesitate to contact us via email at info@dofactory.com, or from our 'contact us' page on our website at www.dofactory.com/contact/contact.aspx. We look forward to hearing from you.

Good luck with your future design pattern and architecture endeavors.