Patterns in Action

Reference Application

for .NET 2.0

Companion document to
Design Pattern Framework 2.0™

by

Data & Object Factory
www.dofactory.com

Index

Introduction

The *.NET Patterns in Action 2.0* reference application is designed to demonstrate how to use design patterns and best practices in building 3-tier, enterprise-quality applications that support a variety of database platforms. This release is optimized for .NET 2.0 and takes advantage of many new 2.0 features. These include generics, the built-in provider pattern, master pages, object data sources, and many more.

Goals and Objectives

The following list of keywords summarize the goals and objectives for the *Patterns in Action 2.0* reference application:

Educational –the purpose of *Patterns In Action 2.0* is to educate you on when, where, and how to use design patterns in a modern, 3-tier, enterprise web application. New in this release are a number of SOA (Service Oriented Architecture) design patterns that demonstrate how to build a Web Service Provider and a Web Service Consumer (a fully functional Windows application) using proven SOA patterns.

Productivity – the design pattern knowledge and skills that you will gain from *Patterns in Action 2.0*, combined with the new .NET 2.0 features offers a great opportunity for enhanced productivity. Microsoft's goal for ASP.NET 2.0 was to reduce the amount of code written by 70%. This number may be a stretch, but the enhancements in .NET 2.0 are very impressive. Design patterns help you establish the architecture in which to reach maximum productivity.

Extensibility – extensibility is more or less implicit in applications that effectively use design patterns. Most design patterns promote the idea of coding against interfaces and base classes, which makes changing and enhancing your application at a later stage much easier. Extensibility may sound like a buzzword, but if you have experience

building and deploying applications, you know that as soon as your application is released, requests for changes and enhancements are coming in.

Simplicity – with simplicity we do not mean simplistic or unsophisticated. What we mean is that the architecture and design are as simple as possible, well thought out, clean, crisp, and easy to understand to all developers on the team.

Elegance - we believe in 'elegant' code. Code should be easy to navigate, self-documenting, and should read ‘like a story’. In fact, elegance goes beyond programming – it applies to all aspects of the application, ranging from the user interface (i.e. easy-to-use and attractive), all the way down to the database (i.e. robust data model in 3rd normal form). Elegance is hard to measure, but you know it when you see it. Design patterns, in effect, are elegant solutions.

Maintainability - building maintainable code goes hand in hand with the two previous points: simplicity and elegance. Code that is simple and elegant is 1) easy to navigate, 2) easy to understand, 3) easy to explain to colleagues, and therefore, much easier to maintain.

Vertical Tiers – actually, we at Data & Object Factory coined this term. Based on our experience, applications that are designed around autonomous functional modules (vertical tiers) are the easiest to understand and maintain. With ‘vertical tiers’, we don’t just mean modularized code as in objects or components. Vertical tiers are larger in scope and represent vertical ‘slices’ of the application each with a particular functional focus. Some examples are: employee maintenance, account management, reporting, and role management. Not only do developers benefit from clearly defined vertical tiers, other stakeholders will benefit also including analysts, designers, programmers, testers, data base modelers, decision makers, and ultimately the end-users.

> Applications frequently do not have clearly marked functional areas. Let’s look at an example. Say, you are planning to build a system that, among other things, manages employees. Without knowing the exact functional requirements, you already know that there will be an employee vertical tier. This employee module

is where employees can be listed, searched, added, edited, deleted and printed. These are all basic operations that apply to any principal entity in an application.

In addition, as a developer you know there will be an employee database table (possibly named 'employee', 'person', or 'party'), an employee business object, and an employee data access component. After reading this document, you will also realize that the application will have an employee façade (or service).

We believe that the best applications (granted, 'best' is subjective) are built by architects who think in vertical tiers and then apply the design patterns to make these 'slices of functionality' or modules a reality.

Enterprise Architecture – building enterprise level applications requires deep understanding of enterprise architecture and design, which includes proven design patterns and best practices. Designing multi-user applications (supporting, say hundreds of concurrent users) requires that you consider complex issues such as scalability, redundancy, fail-over, security, transaction management, performance, error handling, logging, and more. If you are involved in building these systems, you are expected to bring to the table practical experience as well as familiarity with design patterns and best practice techniques.

What is "Patterns in Action 2.0"?

A quick overview of the functionality of the application follows:

Patterns in Action 2.0 is a web-based e-commerce web 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 cart. Their shopping carts can be managed by removing items and changing quantities. In the shopping cart, shipping costs are computed based on the shipping method selected. A separate administrative module allows administrators to view and maintain (add, edit, delete) customer records as well as analyze customer orders and order details.

Patterns in Action 2.0 comes with a Web Service that exposes the administrative tasks mentioned above as a service that is consumed by a client application. A Windows Application is included that consumes these public services. The technology in which a Web Service client consumes the services of a Web Service is part of a new architectural approach called Service Oriented Architecture (SOA).

Many articles have been written about the exact meaning of SOA, but at its core SOA is a way to integrate and aggregate applications from one or more autonomous service systems. SOA has gained a lot of traction lately. *Patterns in Action 2.0* demonstrates some of the 'early discovered' SOA design patterns and best practices. We quality the patterns as ‘early discovered’ because SOA is a young field that is changing day by day and is still in the process of reaching maturity.

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 it in a linear fashion. This document contains the following sections:

Setup and Configuration: This section describes how to setup and configure the application. It discusses the .NET solution, the database, and the web.config configuration file.

Finding Your Way: This section lists the documentation that is available for *Patterns In Action 2.0*. In addition to this document (the one you’re reading right now) there are:

1) class and type reference guide,
2) in-line code comments, and
3) class diagrams for each of the 12 projects that make up the application.

Application Functionality: This section presents the functionality of the application by stepping you through the e-commerce application in which users shop and where administrators manage customer’s records and their orders. It also discusses the web

service and web service client (a windows application). The latter is designed for managers to manage customers, orders and order details.

Application Architecture: This section provides an overview of the 3-tiers used to construct the application: the Presentation tier, the Business tier, and the Data tier. It discusses how the different tiers communicate (i.e. who calls who) as well as the place to add your 'non-functional' items, such as, validation, user authorization, and transaction management.

. NET Solutions and Projects: This section looks at the .NET Solution with its 12 projects. The projects are organized according to the solution's 3-tier architecture and it is important that you have a good grasp of this structure. ASP.NET 2.0 web site projects are quite different from those in VS 2003. A great feature of ASP.NET 2.0 is the built-in development Web Server which makes it easy to develop, run, and test web sites without IIS.

Design Patterns and Best Practices: This section lists and catalogues the numerous design patterns that are used in *Patterns in Action 2.0*. They are organized in 3 separate groups: GoF Patterns, Enterprise Patterns, and SOA Patterns.

Setup and Configuration

Solution setup:

The solution includes class libraries, a Web Application, a Web Service, and a Windows Application. We recommend you copy the entire *Patterns in Action 2.0* solution to the following folders:

For the C# edition:

C:\Program Files\DoFactory\Design Pattern Framework 2.0 CS\Patterns in Action

For the VB edition:

C:\Program Files\DoFactory\Design Pattern Framework 2.0 VB\Patterns in Action

Your folder structure should look like this:

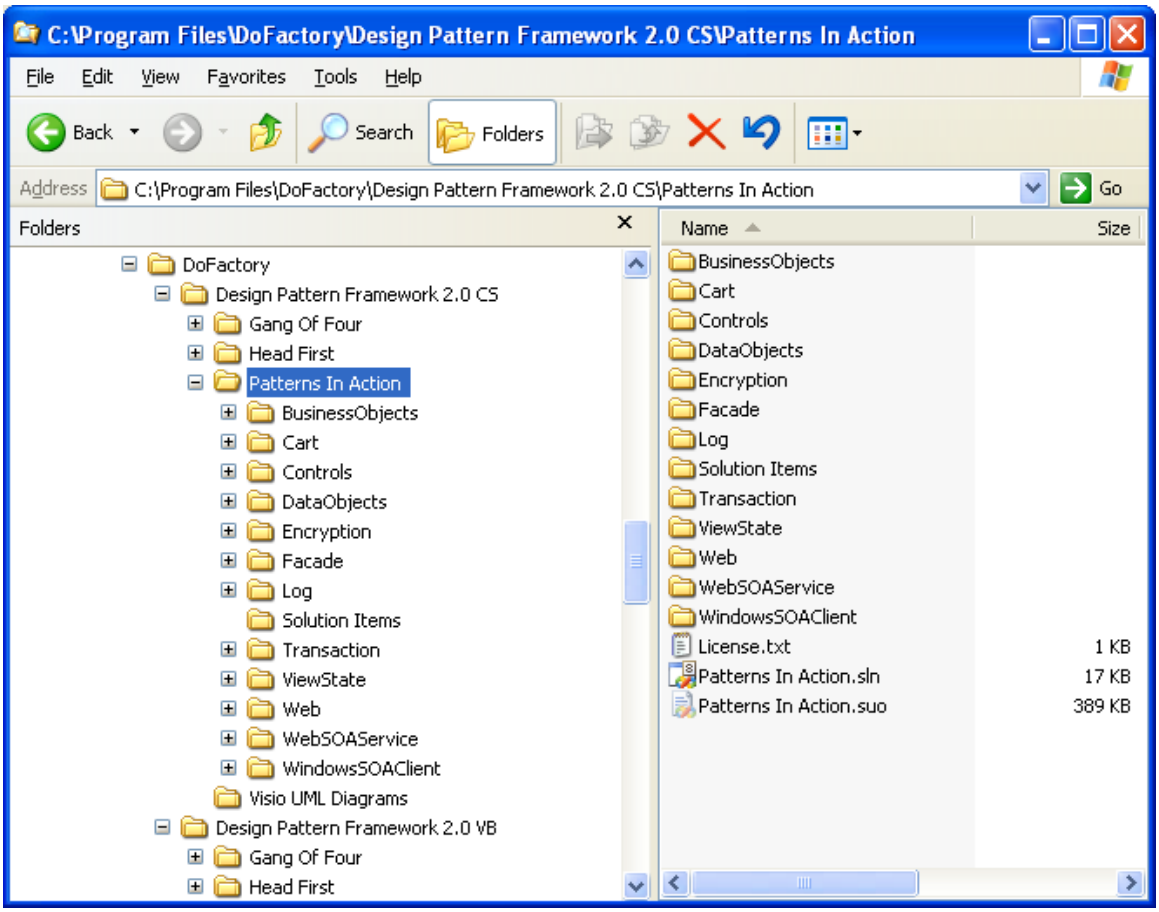
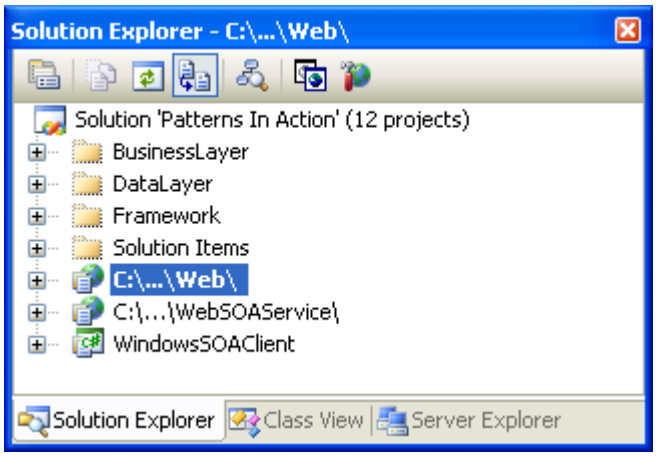


Figure 1: Solution folders

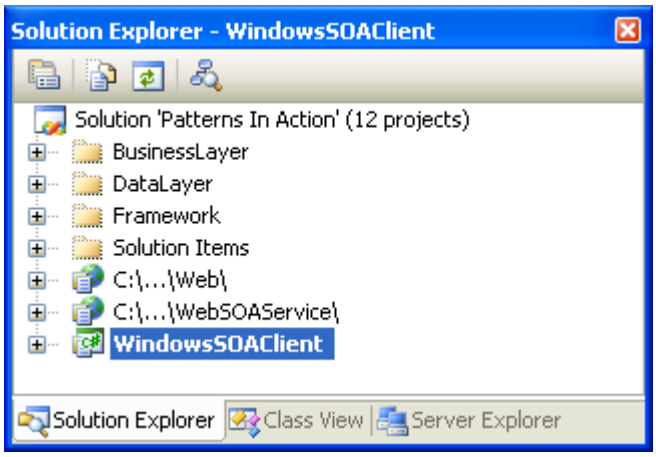
Double click on the solution file named "**Patterns In Action.sln**" and Visual Studio 2005 launches with the solution open.

Once in Visual Studio 2005, you can select one of two Startup Projects.

1) To run the Web Application select the **\Web** application as the Startup Project (it will show in bold). See below.

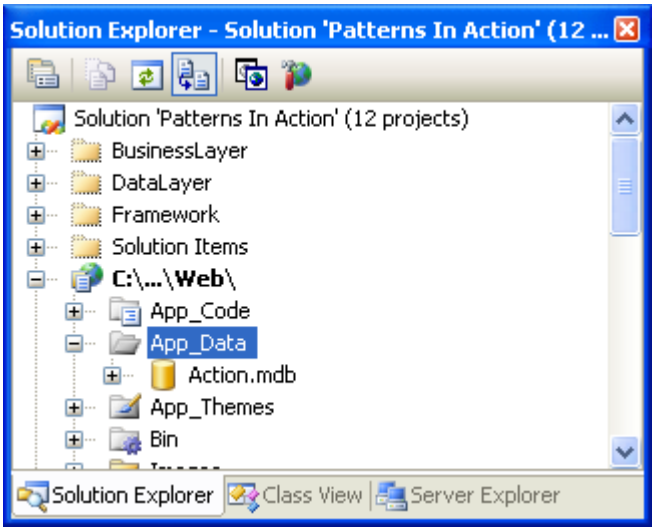


2) Alternatively, you can run the Windows Application (a web service consumer) by selecting the **WindowsSOAClient** project as your Startup Project. See below.

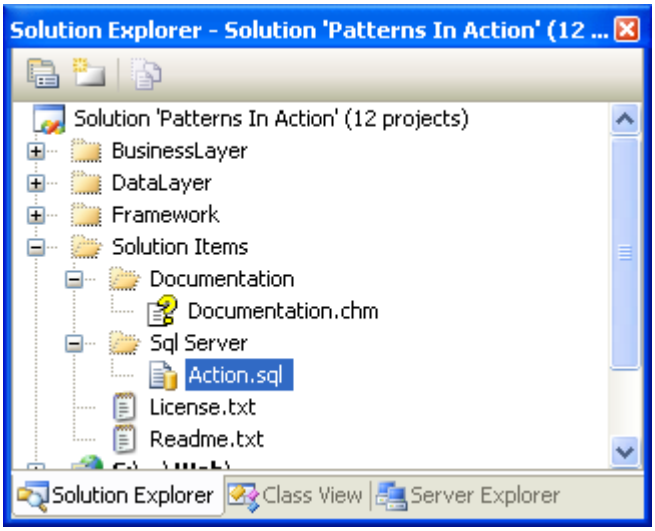


Database Setup:

Patterns in Action 2.0 supports two databases: MS Access and SQL Server. By default the application runs against a local MS Access database called **Action.mdb**. This database resides in the standard ASP.NET 2.0 data folder named \App_Data\. See image below. A second copy of the database resides in \App_Data\ under the Web Service project.



To configure *Patterns in Action 2.0* to run against a Sql Server database is easy. These are the steps: First, create an empty database named **Action** (or a similar name). Second, run the scripts named *Action.sql* against this new database. This will create the data model and enter the required data in the database. You can find the *Action.sql* file in the .NET solution under a folder named \Solution Items\Sql Server\. See image below. Finally, you will need to adjust your web.config file (described in the next section on web.config) and you're ready to go. Please note that the entire solution has two web.config files: one for the Web Site and another for the Web Service. If you change one, you most likely want to change the other as well.



For .NET developers using Oracle, some placeholder data access code for Oracle is included. The port to Oracle will be easy because the application uses no database specific SQL or stored procedures. You will need to program the Oracle specific data access components, but again, this will be easy because the SQL is identical to the SQL used in Sql Server. Finally, you will need to build and populate the Oracle database.

Web.config Setup:

Web.config is the configuration file for ASP.NET web sites and web services. In this file, you configure your database and several other custom application options. The most important items are listed below.

```
<appSettings>
  <!-- Provider. Options are: System.Data.OleDb, System.Data.SqlClient,
       or System.Data.OracleClient -->
  <add key="DataProvider" value="System.Data.OleDb"/>

  <!-- Log Severity. Options are: Debug, Info, Warning, Error,
       Warning, or Fatal -->
  <add key="LogSeverity" value="Error"/>

  <!-- Default Shipping Method. Options are: Fedex, UPS, or USPS  -->
  <add key="ShippingMethod" value="Fedex"/>
</appSettings>
```

```
<!-- Connection string settings -->
<connectionStrings>
  <add name="System.Data.OleDb"
   connectionString="Provider=Microsoft.Jet.OLEDB.4.0;Data
                     Source=|DataDirectory|action.mdb"/>
  <add name="System.Data.SqlClient"
   connectionString="Server=(local);Initial Catalog=Action;User
                     Id=sa;Password=secret;"/>
  <add name="System.Data.OracleClient"
   connectionString="Data Source=MyOracleActionDB;User
                     Id=scott;Password=tiger;Integrated Security=no;"/>
</connectionStrings>
```

The default database is MS Access for which System.Data.OleDb is the data provider. Valid DataProvider values and their database mapping are:

- System.Data.OleDb is used for MS Access,
- System.Data.SqlClient is used for Sql Server, and
- System.Data.OracleClient is used for Oracle.

To use Sql Server, for example, you set DataProvider to System.Data.SqlClient. Be sure that the associated connectionstring (under the <connectionStrings> section) for the System.Data.SqlClient is correct. At a minimum, you must adjust the User Id and Password. Similarly, if you wish to run against Oracle set the proper DataProvider and associated connectionString.

Two application settings are configured in web.config: they are LogSeverity and ShippingMethod.

With LogSeverity you specify at what level error messages should be logged by the Logging system. For example if the log severity level is ‘Warning’, and a log message of level ‘Error’ is issued by the application, then it is logged (because the severity for a ‘Error’ is higher than or equal to Warning). A log message with severity level ‘Info’ will not be logged. Logging is explained later in this document.

ShippingMethod is also configured in web.config. It denotes the default shipping method used in every user's shopping cart. Possible values are: Fedex, UPS, and USPS (US Postal Service). Users have the ability to override the default setting and choose their own shipping method. Shipping methods are explained later in this document.

Finding your way

Patterns in Action 2.0 comes with several sources of documentation. First, there is this document, “Patterns in Action 2.0.pdf”, which will guide you through the setup, functionality, architecture, and design patterns used in the application.

Secondly, a reference document is included that has details on all types, classes, interfaces, methods, arguments, events, etc. used in this application. Strangely enough, Microsoft has dropped support for XML documentation in ASP.NET 2.0 applications -- therefore the types in two of the12 projects are not part of this document. This reference document is located at: \Solutions Items\Documentation\PatternsInAction.chm.

Thirdly, the application code is well commented. Each class has <summary> information and many have additional <remarks>. Design patterns that are used in these classes are listed. All public and private members (methods, properties, etc) have comments as well.

Finally, each project has folder named _UML Diagram\ that contains a class diagram of the major classes in the project. If you are visually inclined and understand UML then these diagrams may be helpful in understanding the classes and their relationships.

Application Functionality

This section presents the functionality of the *Patterns in Action 2.0*, i.e. what does the application do and how are users navigating though the application The application itself is simple and straightforward. This was done on purpose to maximize learning and minimize destractions with unnecessary details and complexities.

You can run *Patterns in Action 2.0* in one of two ways:

1) As a Web Application, and
2) As a Windows Application.

Web Application:

The Web Application is a basic e-commerce application. To run it, select C:\…\Web\ as your Startup Project and select Run (hit F5).

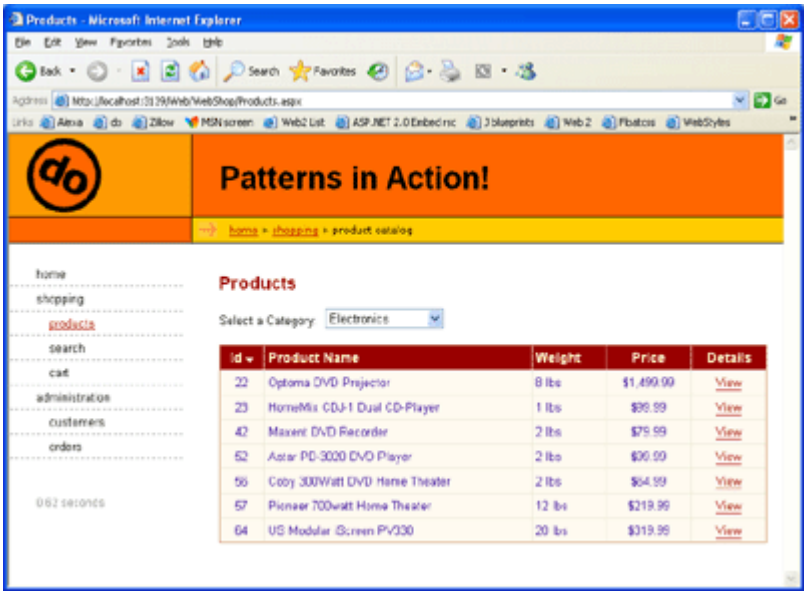


Figure 2: web application screenshot

Begin by playing the role of an online shopper shopping for electronic products. Run the following steps by selecting the menu items under shopping. Browse for products by

product category, sort the list, and view details. Then, search for products by entering a partial name and/or price range, sort the resulting list of products, and view product details. In the product detail page add a product to your shopping cart. Add several more items to your cart. In the shopping cart page, remove items, change quantities, recalculate the total and subtotal, etc. Select a different shipping method and notice that the cost of shipping and the total cost are adjusted accordingly.

Next, play the role of administrator whose task it is to manage the customers in the database as well analyze their orders and order details. This functionality is available from the menu items under administration. Open the list of customers. This page allows three basic customer maintenance operations: Add, Edit, and Delete. Experiment with these options. A business rule in the application states that customers with orders cannot be deleted. Therefore, to be able to delete we suggest you first Add a new customer to the list. After that, sort by Customer Id (descending) and see that your new customer appears on top of the list. Select Edit and change some fields of the new customer and save your changes. Finally, from the customer list, Delete the customer from the database.

Orders can be viewed on the Orders page. It contains a customer list with order totals and last order date. Sort by these order-related fields by clicking on their headers. See who has the most orders placed (the Xio Zing Shoppe). Finally, view all orders for a customer and order details (line items) for one of the orders. All these pages follow the master-detail paradigm, that is, the list of customers is the master and their orders are the details. Each order, in turn, is a master as well, because orders have order details (line items). Master-detail is a very common User Interface pattern in applications.

There are a couple more items we'd like to point out. The application uses the new SiteMapPath control (also called 'bread crumb' control) just below the header. It displays the current position in the site map (as defined in the Web.sitemap file). Notice that the selected menus are highlighted (red text with underscore) depicting the current page selection. Finally, a little below the menu you notice a message in gray that displays the time (in milliseconds) to render the page.

Windows Application:

To start the Windows Application set WindowsSOAClient as the Startup Project and select Run (hit F5). This application requires the availability of a Web Service, which, when running from within Visual Studio 2005 is automatically launched using the built-in ASP.NET Web Server. The web services are on ports 2668 (C#) and 2660 (VB). These port numbers do not change because we turned off dynamic port assignments. To assign your own port numbers see: http://msdn2.microsoft.com/en-us/library/ms178109(VS.80).aspx. If you use IIS, or your web service port numbers are different from 2668 or 2660, then you can configure the Web Service Url from within the Login dialog window (through a link label on the top right).

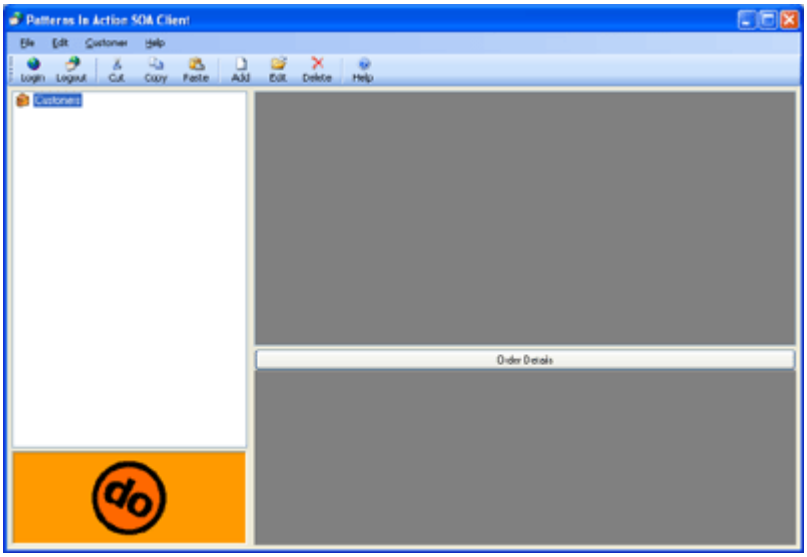


Figure 3: Windows application (not logged in)

When starting the web service client you see the main form with three empty panes. Select the File->Login menu item or click the Login toolbar button. This opens the login dialog in which login credentials are entered.

Figure 4: Login form

To keep it simple *Patterns in Action 2.0* supports just one set of valid credentials (in a real application you will have many user credentials stored in a database). They are:

User Name: **Jill**
Password: **LochNess24**
Security Token: **ABC123**

Text boxes in the login dialog are pre-populated with these values and selecting OK will log you in. Before logging in you may want to explore two link labels on this dialog. The "What are my credentials" link label opens a simple dialog which shows you what the valid credentials are (in case you changed and forget them). Selecting the "Set Web Service Url" link label opens a dialog where the URL for the Web Service is entered. Most likely, you won't need this, but it is available for when the web service is deployed on a remote server, different port number, etc. Choosing OK will log you in.

Note: If you still have difficulty logging in (or receive 404 errors) double check that no file named *app_offline.htm* has been created in the Web Service folder. We found that this file may be created following VB.NET compile errors. If you see this file, simple delete it.

Once logged in, a list of customers will display in the tree view on the left. Remember that this application is a client to a Web Service and the response will not be immediate

(particularly when using MS Access). Click on one of the customers and (following a slight delay) the customer's orders and order detail data are retrieved and displayed. Highlight an order and notice that order details display instantly at the bottom pane (both orders and associated order details have been retrieved for the selected customer).

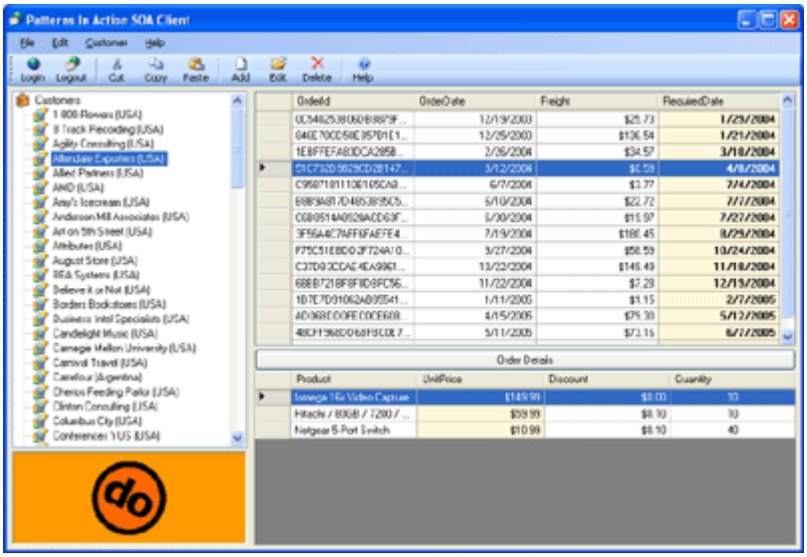


Figure 5: Populated with customers, order, and order details

Once order and order details have been retrieved from the Web Service, they are cached on the client. Over time the performance of the application improves as more and more customer orders have been retrieved and viewed.

The customer maintenance functionality is same here as it is in the Web Application -- you can Add, Edit, and Delete customer records. Not only are they functionally the same, they also share a very large part of the application. Both use the same Façade, Business layer, and Data layer. The details of this are explained later in this document.

Experiment with the customer maintenance options from the Customer menu or by clicking the Add, Edit, Delete toolbar icons. The operations themselves are straightforward and self-explanatory. The one thing to remember is that the middle tier (Business layer) contains a business rule that states that customers with orders cannot be deleted. To explore the delete functionality you will first have to create a new customer (note: new customers are added to the bottom of the customer tree), edit it if you want, and then delete it from the list of customers.

As a last step select Logout, which disconnects you from the Web Service. This also destroys the Session object on the Web Service.

Application Architecture

Before making decisions on the design and architecture of an application, you must have a good understanding of the functional and non-functional requirements. Functional requirements include the business processes that the application will perform. An example may be: the user can login or setup a new account; he/she should be able to search for a product from a catalog of products, etc. Functional requirements are often captured in use-case artifacts. Non- functional requirements (i.e. operational requirements) are harder to pin down – they determine the desired levels of scalability, availability, maintainability, and security of the application. Non-functional requirements provide the environmental details that are necessary to run the system effectively and efficiently.

Designing business applications involves making decisions about the logical and physical architecture. A layered approach is generally accepted as the best because it logically separates the major concerns of the application. In addition, the application will be loosely coupled (i.e. more flexible), more easily maintained, easy to enhance, and it provides the option to physically distribute the layers across multiple dedicated servers.

Layered Architecture:

Patterns in Action 2.0 has been built with a 3-tier architecture (Please note that these are horizontal tiers, which are entirely different from the previously discussed vertical tiers or modules). The next page shows a cross-sectional view of the web application with different layers including some of the design patterns that are involved in this layering.

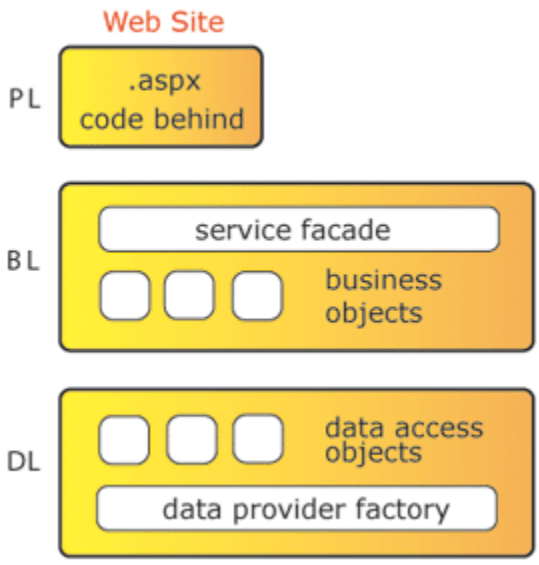


Figure 6: Web Application Cross Section

All ASP.NET developers are familiar with the presentation layer (**PL**); it is made up of .aspx pages and associated code behind pages. Many applications have all their code in the code behind pages, but in a 3-tier architecture the code behind holds relatively little code. In fact, sometimes no code at all because business logic and data access has been factored out into other layers. The primary concern of the code behind is to respond to user events and access to the business layer throught a specialized set of objects called a Façade (named after the design pattern by the same name) which provides data to and from the aspx page.

The business layer (**BL**) contains two major categories of classes: the Façade and the Business objects. The Façade is the entry point into the Business layer and exposes a simple, course grained interface. The Façade is a busy place because it manages business objects and coordinates several common operational tasks, such as data validation, user authorization, and transaction management. Business objects represent the domain logic that is relevant to the application, that is, the data and the business rules are important to the business. Communication between the PL and BL takes place via the service façade. The PL never directly instantiates and maintains business objects; it does this indirectly via the façade layer only.

The main responsibility of the data layer (**DL**) is to access the database and map database tables to business objects and vice versa. It is the only place in the application where data access occurs, including database connections and executing dynamic SQL and/or stored procedures. Data access object (DAO) interfaces represent the data access functionality required to run the application. These interfaces are implemented by database specific data access classes (one set for MS Access, one for Sql Server, one for Oracle, etc). This model makes it easy to switch to a different database just by changing a web.config configuration setting. *Patterns in Action 2.0* supports MS Access and Sql Server.

The Data Provider Factory classes are new in ADO.NET 2.0. They have made the data access provider patterns in prior releases of *Patterns in Action* obsolete. The Data Provider Factory is a built-in implementation of Microsoft's provider design pattern which is a new feature in .NET 2.0. The provider pattern is used throughout .NET 2.0. A custom ViewState provider that demonstrates the use of the provider design pattern is included in *Patterns in Action 2.0* and described later in this document.

Web Service architecture:

The Web Service is interesting in that it integrates like lego blocks into the 3-tier architecture without code changes. The Web Service is essentially another PL sitting on top of the BL -- in parallel to the Web Site. Furthermore, it accesses the BL through the Façade just as the Web Site. The Façade exposes the same interface and executes the same functional and operational procedures. The Web Service does not have to re-implement the tasks that the façade and underlying layers already handle. In essence we have reached total code reusabilty. Below is a cross-sectional view of the layers.

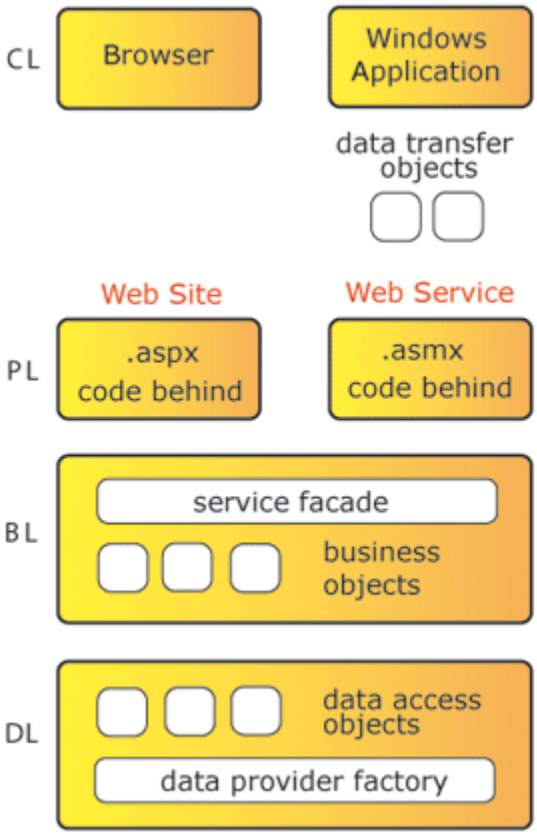


Figure 7: Web Application and SOA Cross Section

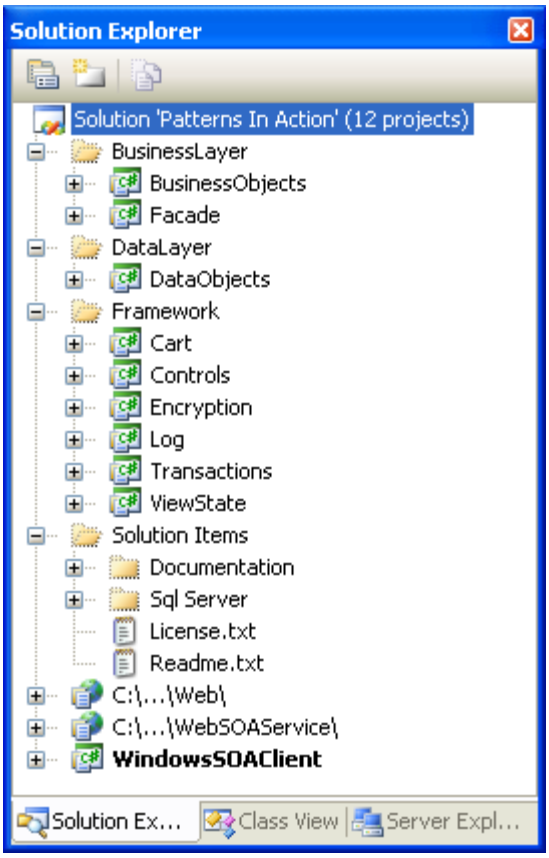
First of all, you'll notice that this image depicts 4 layers. There is a new top tier called Client Layer (**CL**) that was not shown before. Some architects refer to their architecture as 4-tier, 5-tier, and sometimes even more. In reality though they are usually talking about the same basic 3-tier model, except that they choose to include additional components or sub-systems that are required to run the applications and declare these another tier. Some consider the client tier (browser) to be the 1st tier and the database the 5th tier. Actually, they have a point, particularly when the browser contains application logic (using Javascript) and the database contains business logic (using triggers and stored procedures). However, we will stay with how this is referred to in the industry and continue referring to our architecture as 3-tier but realize that the client layer and database layer are important parts of the application.

The CL for the Web Site is simply a browser. The CL for the Web Service is a full functioning Windows application (WinForms), but it could have been any device that understands XML / SOAP protocols over HTTP, such as, hand-held devices. In *Patterns in Action 2.0* the CL is a Windows application. The interface (or contract) between the Web Service Provider and the Web Service Consumer is built around a messaging model in which messages contain Data Transfer Objects. Data Transfer Objects is an Enterprise design pattern of objects that contain business data only, but no behavior (methods or properties).

When comparing [Web Site + browser] versus [Web Service + Windows application] you'll find they are very different (e.g. deployment, user experience, etc). However, with a 3-tier architecture we have succeeded in building a Business layer and Data Layer that can be used by any kind of presentation layer. No code changes are required.

The .NET Solution and Projects

This section explores the .NET Solution and Projects used in *Patterns in Action 2.0.* Open the application in Visual Studio 2005. Collapse all projects and folders and then open the first level folders until you see all projects. Don't open the Projects just yet. Your Solution Explorer should look like the image below:



The Solution Explorer shows 12 Projects. They are organized mostly along the layers of the application.

When running the **Web Application** the following tiers & projects are invoked:

Tier	Project
Client tier	The browser
Presentation tier	C:\…\Web\ Web Site Project
Business tier	BusinessLayer Folder with BusinessObjects and Façade Projects
Data tier	DataLayer Folder with DataObjects Project

When running the **Windows Application** the following tiers & projects are invoked:

Tier	Project
Client tier	WindowsSOAClient Windows Application Project
Presentation tier	C:\…\WebSOAService\ Web Service Project
Business tier	BusinessLayer Folder with BusinessObjects and Façade Projects
Data tier	DataLayer Folder with DataObjects Project

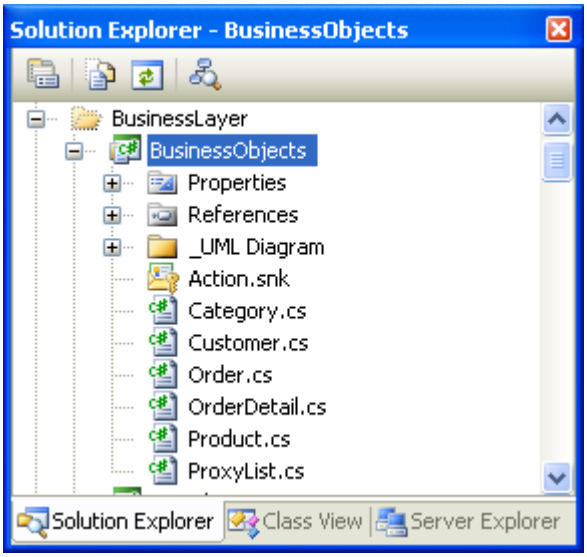
Framework is a folder that contains six projects offering supporting functionalities. With the exception of the Cart and Controls projects, they all represent the non-functional (or operational) aspect of the application. They are helper classes that make the application more scalable, more secure, perform better, easier to manage, etc. The six projects are:

Framework Project	Description
Cart	A non-persistent e-commerce shopping cart
Controls	A web control library project
Encryption	A security library with encryption/decryption functionality
Log	An error logging system
Transactions	Manages database transactions for different databases
ViewState	A ViewState service provider used to accelerate web applications

We'll now look at each project in more detail:

BusinessObjects:

Expand the BusinessObjects Project until you see the view below:



Project BusinessObjects is a class library that contains business objects (also called domain objects). Business objects are objects that encapsulate data with associated behavior that is relevant to the business. Business objects in *Patterns in Action 2.0* are: Category, Product, Customer, Order, and Order Detail. Business rules may or may not be encoded in these classes. In Patterns in Action 2.0 the objects are simple and have little or no business rules. In a more complex real-world application, you may find business rules encoded in these objects.

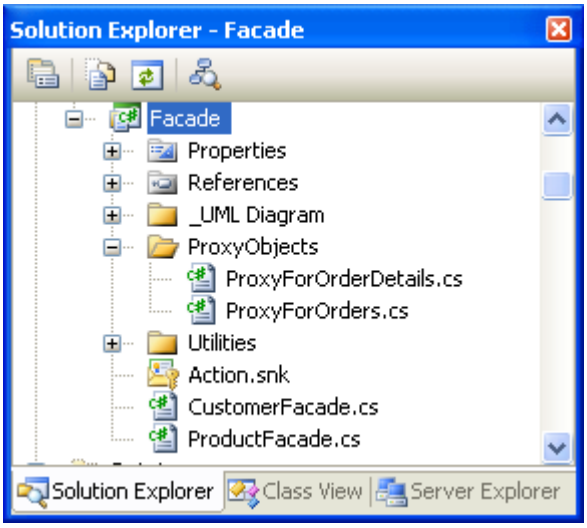
Let's look at an example of how a business rule may be encoded in a customer business object. There may be a rule that states that customers are ranked according to their recent order history. Customers with 10 or more orders over the last 3 months receive gold status, customers with 5 to 10 orders receive silver status, and customers with fewer than 5 orders are given bronze status. These statuses determine the relative priority treatment when calling a 1-800 number. When an order is placed, the rule in the business object re-evaluates the customer's ranking.

Business objects are not directly involved with data access, so there are no Load, Update, or Save methods. Data access is handled entirely by the Data Layer, which accepts and returns business objects. The Data Layer accepts business objects, then gets and sets their data via object properties. The Data Layer understands the object model, as well as the data model, and maps one model to the other.

This project contains an abstract class named ProxyList, which represents a reference to a list of items. Proxy objects are frequently used to manage a limited resource, and in this case, the resource is database access and system memory. In addition to the Proxy design pattern, ProxyList also implements the Lazy Load pattern, meaning it loads the list of data only when necessary. ProxyList is used in the Façade layer where Customer objects proxy their Orders, and where Order business objects proxy their Order Details. The result is that Orders for a Customer are loaded only when absolutely needed. Likewise, Orders details for an Order are loaded only when absolutely necessary.

Façade:

Expand the Façade project until you see the view below:



The Façade represents the exposed ‘service interface’ of the Business Layer. Every call into the Business Layer must go through the Façade. The PL should never bypass the

Façade and try to call directly into the Business Layer or Data Layer. Architects frequently talk about accessing the 'Service Layer' or the 'Services' which is just another name for Façade.

Façades are grouped by functionality, so you may have Membership Façade, Employee Façade, Reporting Façade, and others. It is strongly advised to group functionality in the Façades so that it matches the 'Vertical Tiers' or modules described earlier in this document. *Pattern in Action 2.0* has two Façades matching the main modules in the application: a Product Façade which supports the shoppers and their product search activities, and a Customer Façade which supports the administrative tasks of maintaining customers and their orders.

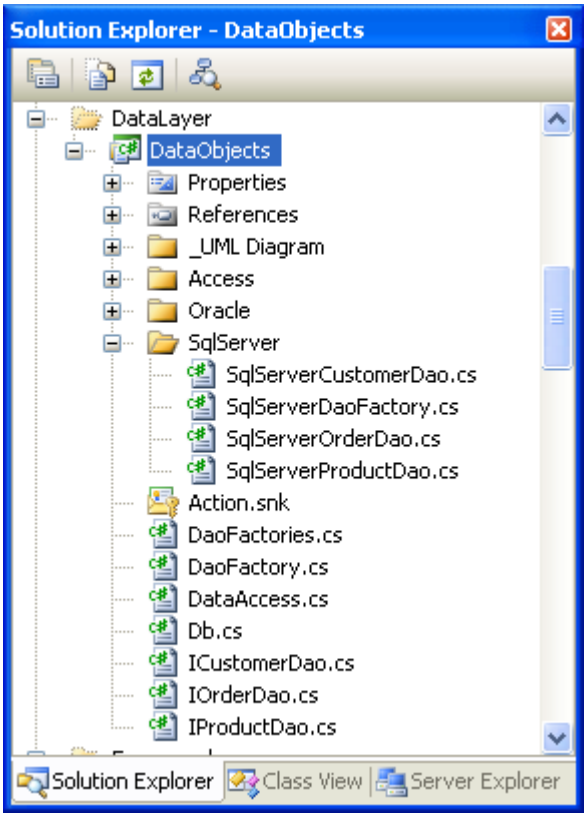
The Façade or Service concept is fundamental to most modern 3-tier architectures. Not only does the Web Application access the Business Layer via the Façade, but also any other client such as Web Services. Like any other PL there should be no direct calls from the Web Service into the Business Layer or the Data Layer; everything should be going through the Façade. It is hard to overstate the importance of the Façade design pattern in modern .NET architectures.

In addition to managing business objects and processing the standard business logic, every interface method in the Façade validates the incoming arguments, authorizes the action for the currently logged-on user, and handles and coordinates database transactions possibly covering numerous business object changes. Everything that goes in or leaves the façade must be checked and validated – even when, for example, data validation or user authentication has already taken place at the PL. The reason is reusability; that is, different PLs (possibly written by different developers or teams) may be accessing the façade. Nothing can be assumed from the client and therefore facades have to take a very conservative position in order to maintain the integrity of the system.

The Façade project implements two ProxyObjects that derive from the ProxyList abstract class. These are proxy objects that 'stand in' for Orders in Customer objects and for Order Details in Order objects.

DataObjects:

Expand the DataObjects project until you see the view below:



The Data Tier resides in single class library project named DataObjects. This project is located in the DataLayer folder. If you have used an older release of the Design Pattern Framework, then please note that this release is quite different from any previous release. The reason is that .NET 2.0 comes natively with several DbProviderFactories, which fully replace our original Data Provider Factories design patterns. DbProviderFactories are built around Microsoft's new Provider design pattern that is used throughout .NET 2.0. This pattern is an aggregate of three of the GoF design patterns and is discussed later in this document.

Three DAO (Data Access Object) interfaces, namely ICustomerDao, IOrderDao, and IProductDao, define the interface between the Business Layer and the Data Layer. The Business Layer does not know anything about data access and the three database

independent interfaces make this possible. The granularity of the methods and properties in Data Access Object interfaces is usually the same or somewhat larger than the business objects and/or the tables in the database. For example, IOrderDao deals with orders and order details, and IProductDao supports product categories and products. Most arguments and return values in these interfaces are business objects or lists of business objects.

As mentioned before, the business objects themselves are not involved with databases or data access. This is entirely handled by the Data Layer, which maps the object model to the relational data model and vice versa.

For each database vendor, such as MS Access, Sql Server, and Oracle, a separate implementation for these interfaces needs to be written. Three folders: Access, SqlServer, and Oracle contain these implementations. The Access folder for example contains AccessCustomerDao, AccessOrderDao, and AccessProductDao. In *Patterns in Action 2.0* the Sql commands for the supported databases are pretty much the same, but if there were Sql or other differences they would manifest themselves in the database specific implementation files.

Microsoft's' Data Provider design pattern participates in handling the database differences. At the highest level there is DoaFactories, which, given a data provider name, returns a database specific Factory. Database specific factories are derived from a base class named DoaFactory. The implementations are found in the aforementioned Access, SqlServer, and Oracle folders. For example, the Access version is called AccessDoaFactory.

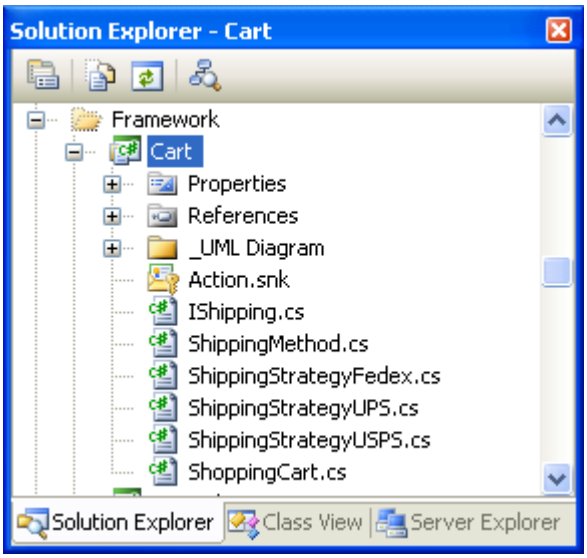
DoaFactories return database specific implementations of the three IDoa interfaces. For example, the AccessDoaFactory, returns AccessCustomerDao, AccessProductDoa, and AccessOrderDao. These latter classes are called indirectly in the Business Layer for business object persistence. The Business Layer uses a static class named DataAccess, which shields it from Factory and other data access details. You find the DataAccess class used in the Façade part of the business layer.

Finally, a helper class named Db is the low level workhorse of the Data Layer. It handles all ADO.NET 2.0 data access calls and also shields the rest of the system from low level database differences, such as retrieval of newly generated identifiers (identities or auto-numbers) from the database.

The Data Layer uses many commonly used design patterns, including, Factory, Singleton, and the new Provider pattern. Finally, notice how important the use of interfaces and abstract classes is to these patterns.

Cart:

Expand the Cart project until you see the view below:



The Cart project is a class library with standard e-commerce shopping cart functionality. This cart is non-persistent, that is, it is only lives for the duration of the session. When a user's session expires, the cart is destroyed as well. It would not be hard to make the shopping cart persistent, either by using cookies or by adding a Cart database table.

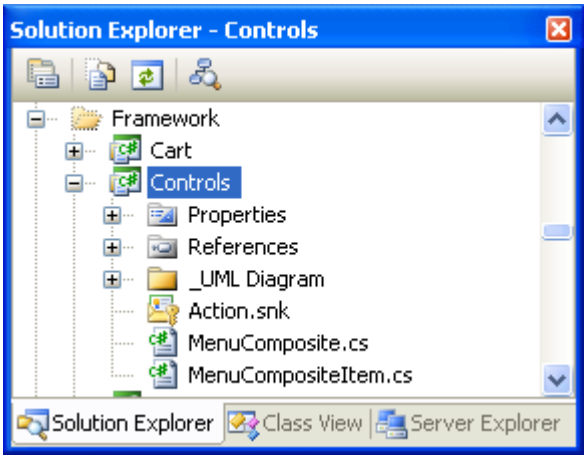
What is unique about this cart is that its data is kept in a DataTable object. We could have created a Cart object class, but the DataTable provides an easy alternative. The DataTable has built-in facilities to databind to a web page. ShoppingCart is the container

class for the DataTable as well as all cart maintenance methods: AddItem, RemoveItem, ReCalculate, ShippingStrategy. etc.

The remaining types in this project are involved with shipping and shipping calculations. The GoF Strategy pattern is used to facilitate a simple 'plug and play' model in which different strategies can be swapped out for another. Possible shipping methods include Fedex, UPS, and USPS (United States Postal Services).

Controls:

Expand the Controls project until you see the view below:



The Controls library is a Web Control library with a custom menu control. This control is located on the ASP.NET Master page and is rendered on every web page. It demonstrates the use of the Composite design pattern, which is used to build tree-like data structures. The menu is a two-level tree hierarchy.

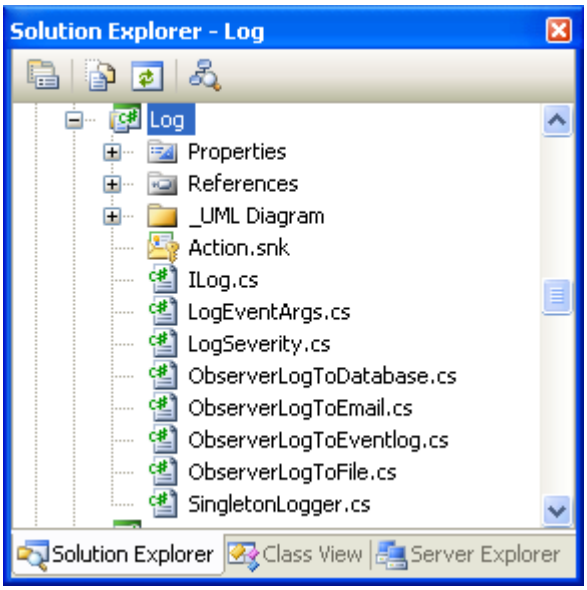
Encryption:

Expand the Encryption project until you see the view below:

The Encryption class library has a static helper class that encrypts and decrypts strings using a hard-coded *key* and *iv* vector. It uses the TripleDes encryption algorithm provided by the .NET Framework. In your own projects, you should generate and use your own *key* and *iv* values and store these in a safe place (preferably outside the code).

Log:

Expand the Log project until you see the view below:



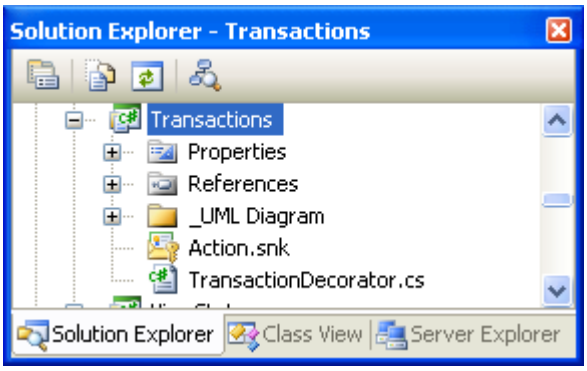
The Log project offers a logging facility that allows you to log error conditions to any kind of storage or output device. The project has four sample classes that log to a database, email, event log, and a file. It would be easy to extend these to other output devices.

The Observer design pattern plays a key role in this library. It allows Observer classes to attach (subscribe) and detach (unsubscribe) to and from a central Logger (subject) and be notified about log events.

Log events have a severity. During development / QA phases you may want to log messages that are tagged with Debug or Info severity, whereas in production you're more likely to only log messages with a level of Warning or Error and higher.

Transactions:

Expand the Transactions project until you see the view below:



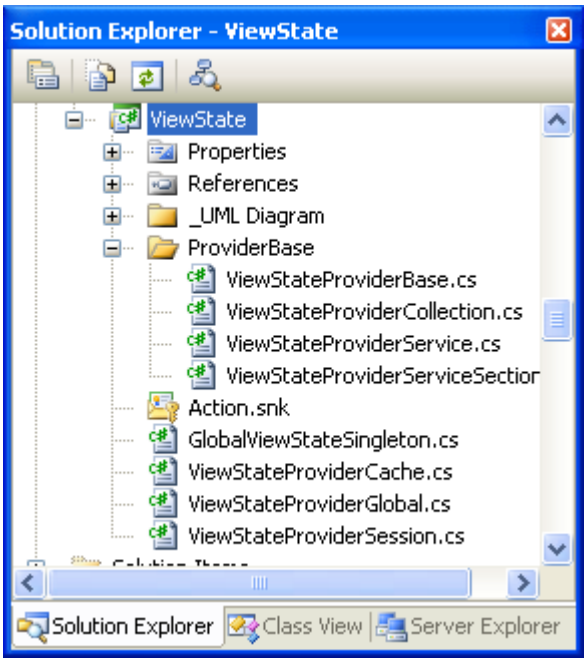
The Transactions class library contains a class named TransactionDecorator which 'decorates' the built-in new .NET 2.0 TransactionScope class using the Decorator design pattern. .NET 2.0 supports a new transaction framework that is available through the System.Transaction namespace. TransactionScope makes a code block transactional by implicitly enlisting data base connections in a distributed transaction.

Microsoft recommends you use the *using* language block to ensure that Dispose is called on the TransactionScope object. In addition, the Complete method must be called within the *using* block to commit the transaction. The exact same rules apply to the TransactionDecorator object.

The main purpose of the Decorator pattern is to 'embrace and extend' the functionality of the object being decorated. The TransactionDecorator embraces the TransactionScope but weeds out the MS Access TransactionScope object calls (MS Access is not supported by TransactionScope).

ViewState:

Expand the ViewState project until you see the view below:



The ViewState in ASP.NET is a great feature that makes keeping track of page state easy. The downside is that it adds a considerable amount of data to every page (stored in a hidden field named "__VIEWSTATE"). In particular when working with list-type controls, such as, DataList and GridView, the increase in page size is very significant. Select View->Source on your browser and you can actually see the data that travels with these pages.

The ViewState class library in *Patterns in Action 2.0* offers an alternative by keeping the Viewstate data on the server rather than sending it over to the client with every page. So, where then is the viewstate data kept? Alternatives include: in the Cache, in Session, or in a globally accessible HashTable data structure. All three providers are implemented in *Patterns in Action 2.0* using Microsoft's Provider design pattern.

The Provider design pattern is not a small pattern. It offers an infrastructure in which several elements need to be implemented. We'll step through the different elements and the classes that implement them.

A 'provider' is a pluggable and configurable component that extends or replaces current system functionality. As an example, ASP.NET offers a built-in Session management system that uses the provider design pattern. It is configurable in web.config where you could select a different provider (for example, Sql Server). Alternatively, you could write your own and customize the way Session data is stored. The provider model offers considerable flexibility.

Providers require their own configuration section in web.config (or app.config). This custom section then contains a list of registered providers one of which is marked as the default. Below is a subset of our web.config, which demonstrates how the ViewState provider is configured.

```
<!--
 Declare viewstateService as a valid section in this file
 Note: this section must be first element under <configuration>
-->
<configSections>
 <sectionGroup name="system.web">
  <section name="viewstateService"
  type="DoFactory.Framework.ViewState.ViewStateProviderServiceSection,
        ViewState"
        allowDefinition="MachineToApplication"
        restartOnExternalChanges="true"/>
 </sectionGroup>
</configSections>

...

<system.web>
 <!--  Custom viewstate provider service  -->
  <viewstateService defaultProvider="ViewStateProviderGlobal">
   <providers>
     <add name="ViewStateProviderCache"
       type="DoFactory.Framework.ViewState.ViewStateProviderCache"/>
     <add name="ViewStateProviderGlobal"
       type="DoFactory.Framework.ViewState.ViewStateProviderGlobal"/>
     <add name="ViewStateProviderSession"
       type="DoFactory.Framework.ViewState.ViewStateProviderSession"/>
   </providers>
  </viewstateService>

 ...
```

The top half demonstrates how a new <viewstateService> section is defined to be referenced later in the configuration file. The bottom half shows how the three viewstate providers are registered. The ViewStateProviderGlobal is set to be the default.

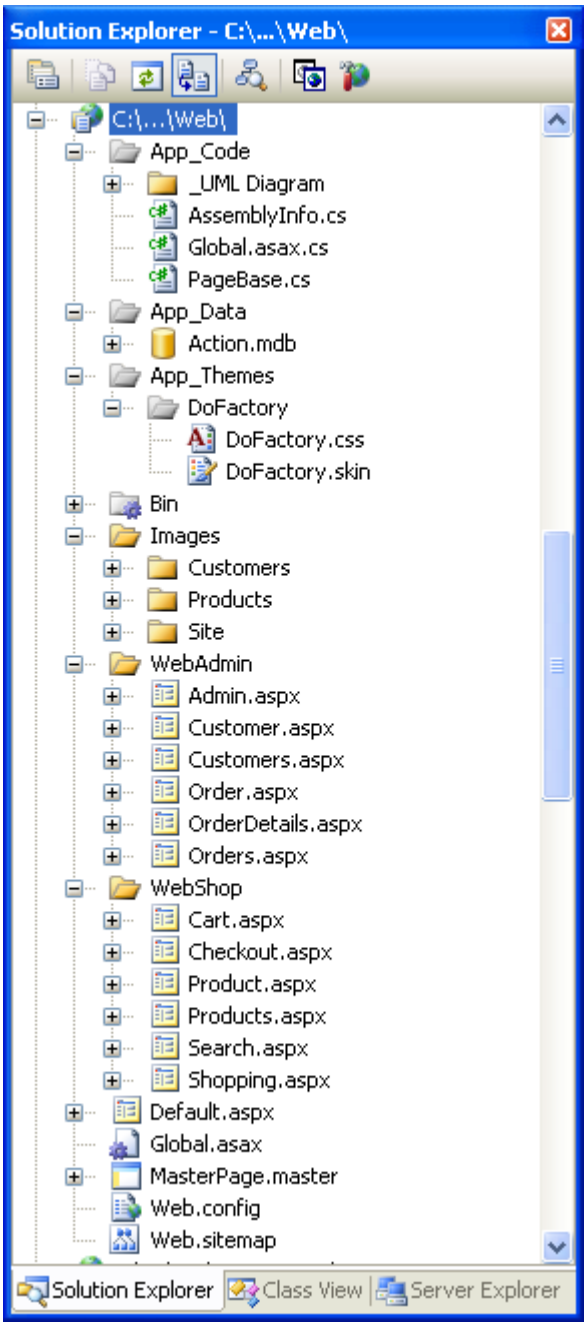
Folder ProviderBase contains the classes that perform the 'plumbing' for the viewstate provider. Abstract class ViewStateProviderBase declares abstract methods that need to be implemented by the ViewStateProvider instances; they are SavePageState and LoadPageState. ViewStateProviderCollection is a collection of ViewStateProviders that are read into memory from the web.config. A static class ViewStateProviderService ensures that the viewstate providers are loaded and that the default provider is set correctly. Finally, ViewStateProviderServiceSection represents the custom section in the web.config file.

The three viewstate providers are ViewStateProviderCache, ViewStateProviderSession, and ViewStateProviderGlobal. They derive from ViewStateProviderBase and implement (override) the two abstract methods SavePageState and LoadPageState. The Global provider has a helper class named GlobalViewStateSingleton.

The performance improvement by keeping ViewState data on the server can be very significant. You could consider using in your own applications. However, we would like point out that ViewState replacement is a complex topic and different scenarios and page sequences need to be thoroughly tested. Before you take this route, please know that the code in *Patterns In Action 2.0* is written for educational purposes only and may or may not work under different scenarios and configurations.

C:\...\Web\

Expand the \Web\ project until you see the view below:



The Web Site uses several features introduced in NET 2.0 including Master Page, Web.sitemap, and the new \App_ \ application folders.

The Default.aspx page and Master page are located in the project root. The shopping module and related pages are located in the \WebShop\ folder. The administration module and related pages reside in the \WebAdmin\ folder. Images are located in the \Images\ folders.

Three *App_* \ folders are used. They are:

App_code\ : contains three files: AssemblyInfo, which adds assembly information, Global.asax.cs (or .vb) which has been separated out from Global.asax and is a closer representation of the VS 2003 model which we prefer, and PageBase, which is the base class to all pages in *Patterns In Action 2.0*. PageBase supports functionality that is shared across the site and includes 1) page render timings, 2) gridview sorting, 3) shopping cart access, 4) viewstate management, and 5) javascript registration.

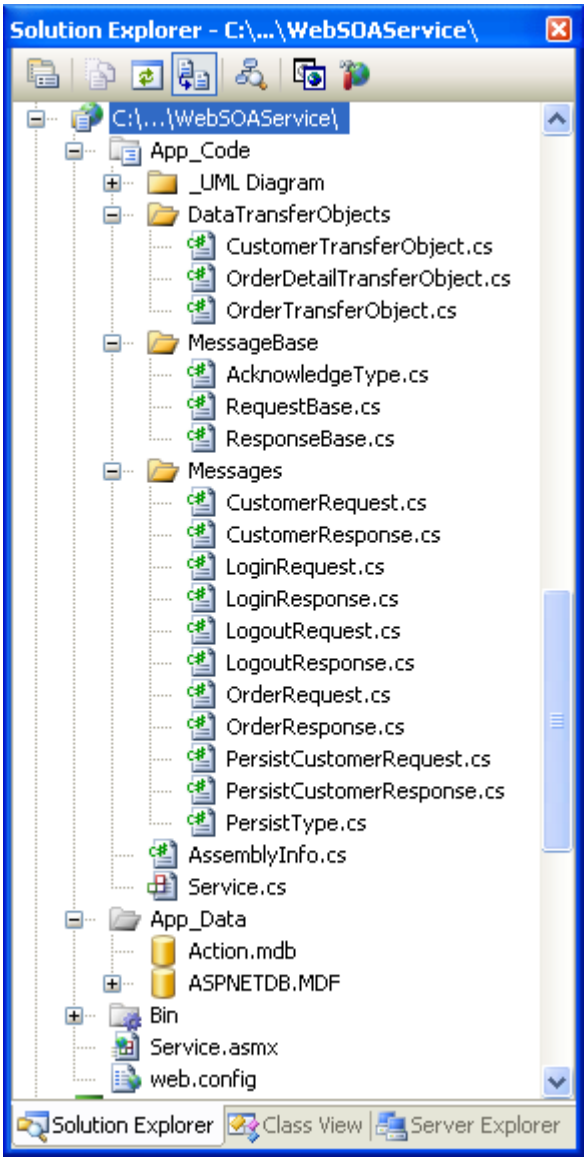
App_Data\ : contains the Action.mdb MS Access database. It is easily referenced with a connectionstring with this syntax: `Data Source=|DataDirectory|action.mdb`

App_Themes\ : contains themes that control the overall appearance and look and feel of an application. *Patterns in Action 2.0* has one theme, named DoFactory. DoFactory.css is a style sheet the controls the formatting of the HTML elements. DoFactory.skin controls the appearance of several of the ASP.NET controls.

The following items may be of interest to .NET developers new to ASP.NET 2.0:
The file *Web.sitemap* defines the hierarchical structure of the web site and is the data source for the SiteMapPath control displayed along the top of each page. When using Master pages there frequently is a need to a) access a control on the Master page from the content pages, and b) access an item on the content page from the Master page. *Patterns in Action 2.0* does both; the code behind page of the Master page demonstrates how this bi-directional access is handled.

C:\…\WebSOAService\

Expand the \WebSOAService\ project until you see the view below:



Most classes in the Web Service project are located in the \App_Code\ folder. Only Service.asmx and Web.config are the only files located in the project root.

Service.cs (or Services.vb) is the file that contains the public web service methods. It includes some private helper methods to support identifier encryption and request caching (both supporting a couple of SOA design patterns). All other classes support the

web method arguments (messages). They reside in three folders: DataTransferObjects, MessageBase, and Messages.

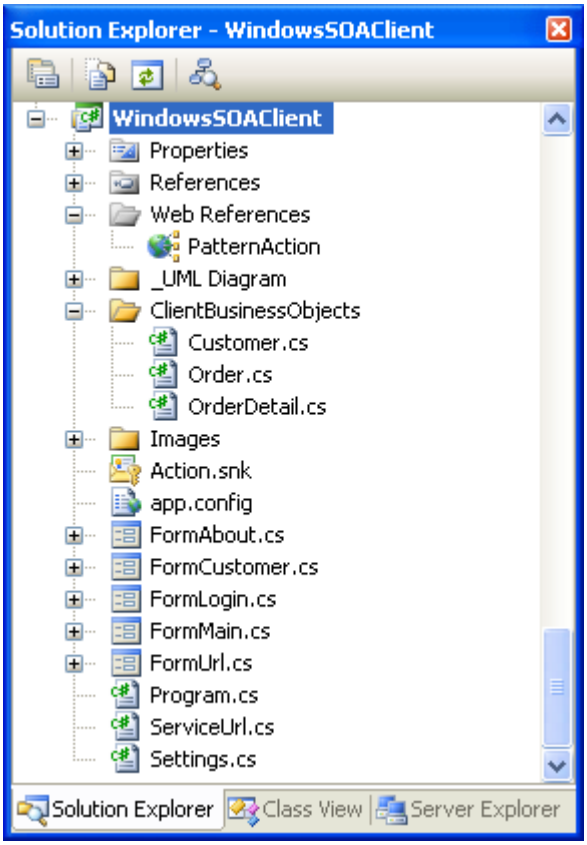
The Data Transfer Object design pattern is an Enterprise pattern that creates objects for the sole purpose of transferring business data – there is no behavior (methods or properties) associated with these objects. Three such objects are used in passing data to and from the Web Service.

Folder MessageBase contains two base classes, RequestBase and ResponseBase. They contain data members that are used in derived RequestMessages and derived ResponseMessages respectively.

The folder named Messages contains pairs of Response and Request objects -- one pair for every web method. For example, the method signature for GetCustomers has a single argument named CustomerRequest and a return value named CustomerResponse. CustomerResponse includes an array of CustomerTransferObjects in which each array item represents a customer.

WindowsSOAClient

Expand the WindowSOAClient project until you see the view below:



Project WindowsSOAClient represents standard Windows application, which is a rich client of the Web Service. It communicates exclusively with the web service – there is no data stored locally such as in a local database or xml file.

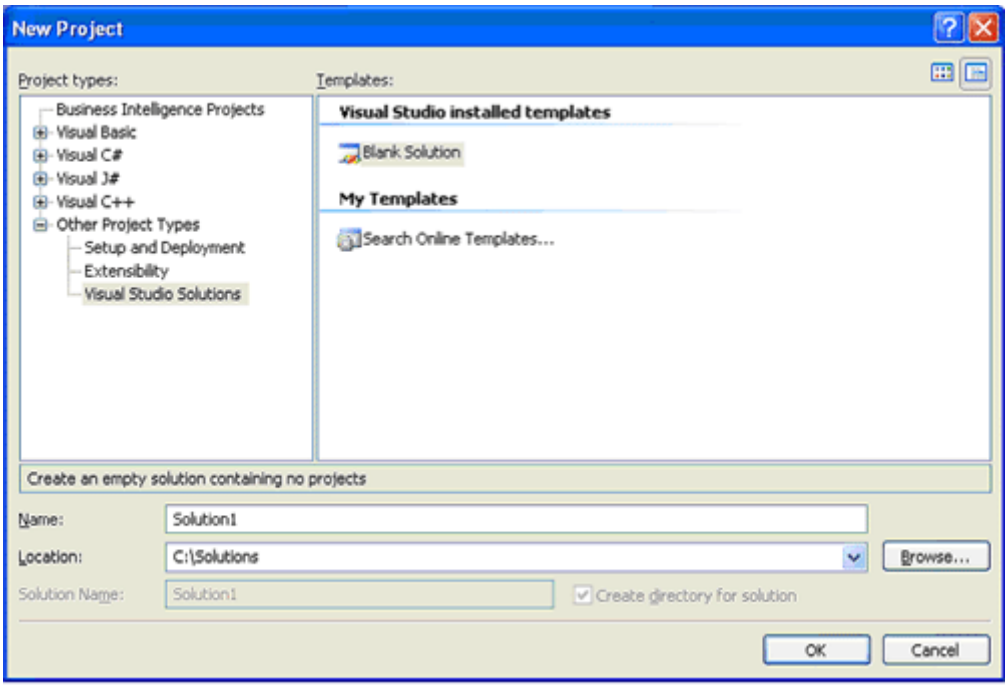
The WebReference folder contains the auto-generated Web Reference proxy object named PatternAction. FormMain is the main form from which all web service access is managed. The other forms are supporting dialogs whose role it is to pass data back and forth to FormMain. Dialog result values determine whether FormMain processes the dialog data or not. This model of placing all logic in FormMain works for this particular application because it is highly focused on the main form, but may not be appropriate for applications with more complex child forms.

This application has its own set of Business Objects in parallel to the ones in the Business Layers on the server side. The client view of a Business Objects is not necessarily the same as the serve view. One difference is that on the client side the business object identifiers are strings (i.e. encrypted database identities, which by the way is also one of the SOA patterns). On the server side, these identifiers are integer values. The three client side business objects are located in the ClientBusinessObjects folder.

Building your own Pattern-based .NET Solution

This section describes how to setup your own .NET 2.0 Solution following the structure used in *Pattern in Action 2.0*. There are some gotchas and this step by step guide will help you get started.

Step 1: Create a Blank Solution



Step 2: Within this Solution immediately create a Solutions Folder. This will prevent a newly added project from 'taking over' the solution role. If you were to add a Web Site without this folder, it would take on the role of the Solution.

Step 3: Create folders according to how you wish to arrange your projects and layers. In *Patterns In Action 2.0* the folders are Business Layer, Data Layer, and Framework, each with its own set of projects.

Step 4: Create class library projects according to your needs. In *Patterns in Action 2.0* we have 2 projects under Business Layer, 1 project under Data Layer, and 6 projects under Framework, one of which contains custom web controls.

Step 5: Then add a "File System" type Web Site and place it in a folder within the solutions folder. *Patterns in Action 2.0* uses a folder named: \Web\.

Step 6: Next, add a "File System" type Web Service and place it in a folder within the solutions folder. *Patterns in Action 2.0* uses a folder named \WebSOAService\.

Step 7: Create a new Windows application, the SOA Client, and place it the Solution folder. In *Patterns in Action 2.0,* this project resides at the solution root level, but you may also create it in a new folder depending on your preference.

Design Patterns and Best Practices

Patterns in Action 2.0 shows how you use of design patterns in a real-world e-commerce scenario. The design patterns and associated best practices in this application can be categorized according to their origin and their purpose. This section describes the three categories found in *Patterns in Action 2.0*. They are: 1) Gang of Four Design Patterns, 2) Fowler's Enterprise Design Patterns, and a new category, 3) Service Oriented Architecture (SOA) Design Patterns.

Gang of Four Design Patterns:

Design Patterns were first 'hatched' in the mid 90's when object-oriented languages began to take hold. In 1997, the Gang of Four (GoF) published their seminal work called "*Design Patterns, Elements of Reusable Object-Oriented Software*" in which they describe 23 different design patterns. These patterns are still highly relevant today, but over time as thousands and thousands of developers have worked with them it is clear that some pattern are more relevant and valuable than others.

There is a group of GoF patterns that has become critical to the success of many enterprise level business applications. These include Factory, Proxy, Singleton, Facade, and Strategy. Many well-designed, mission-critical applications make use of these patterns. Experienced .NET developers and architects use these pattern names as part of their vocabulary. They may say things like: 'this class is a stateless Singleton Proxy', or 'that is a Singleton Factory of Data Provider Factories". This may seem intimidating at first, but once you're familiar with the basics of these patterns, they become second nature. As a .NET architect, you are expected to be familiar with these terms.

Another group of patterns is more applicable to highly specialized, niche type applications. The Flyweight pattern is an example as it is used primarily in word processors and graphical editors. Likewise, the Interpreter pattern is valuable for building scripting parsers, but its usefulness is minimal outside those types of applications. Both, Flyweight and Interpreter are highly specialized design patterns.

Several patterns have proven so immensely useful that they ended up in programming languages themselves. Examples are Iterator and Observer. The *foreach* (*For Each* in VB) language construct is an implementation of the Iterator pattern. The .NET eventing model with events and delegates is an implementation of the Observer design pattern. These examples show just how pervasive design patterns have become in everyday programming. Please note that this document does not further address or elaborate on these 'language' patterns specifically.

The majority of the GoF patterns falls into a category that is important but at a more granular and localized level (unlike the application level architecture patterns such as Façade and Factory). They are used in more specialized and focused circumstances. Examples include: State, Decorator, Builder, Prototype, and Template. The State pattern, for example, is used when you have clearly defined state transitions such as a credit card application process that goes through a sequence of steps. The Decorator is used for extending the functionality of an existing class. The Template is used to provide a way to defer implementation details to a derived class while leaving the general algorithmic steps in tact. Several others exist. Again, they are frequently used, but at a more detailed and focused level within the application.

Finally, there is a small group of patterns that is rarely used. These include the Visitor and Memento design patterns.

A note about the Factory pattern: the GoF patterns contain two Factory patterns, namely Abstract Factory and Factory Method. The Abstract Factory pattern is essentially a generalization of the Factory Method as it creates families of related classes that interact with each other in predictable ways. Over time the differences between the two have become blurry as developers mostly talk about the Factory pattern, meaning a class that manufactures objects that share a common interface or base class. These manufactured objects may or may not interact with each other. In *Patterns in Action 2.0* we have also adopted this usage and refer to it simply as the Factory pattern.

Microsoft introduced the Provider design pattern in .NET 2.0. Although it is not a GoF pattern (it was not part of the original 23 patterns), it is included here because you'll find

it used throughout .NET 2.0. Provider is essentially a blending of three GoF design patterns. Functionally, it is closest to the Strategy design pattern, but it makes extensive use of the Factory and Singleton patterns.

Patterns in Action 2.0 is a reasonable representation of how patterns are used in the real world. It includes only the most valuable and most frequently used design patterns in real-world application development. We could have crammed all 23 GoF patterns in the application, but that would have skewed reality by not reflecting the real-world usage of design patterns.

The table below summarizes the GoF patterns used in the application, their location, and their usage. These patterns are referenced also in the code comments. Possibly the best way to study these is to have the source code side-by-side with the 69 Design Pattern Projects that are part of the Design Pattern Framework.

GoF Design Pattern	Project	Class	Usage
Proxy	BusinessObjects	ProxyList	Used as base class for proxy objects representing a list of items.
Façade	Façade	CustomerFacade ProductFacade	Used as the façade (or service interface) into the business layer. All communication to the business layer goes through the façade.
Proxy	Façade	ProxyForOrderDetails ProxyForOders	Used as proxies for list of orders and list of order details. A way to limit database access to what is absolutely necessary.
Factory	DataObjects	DaoFactories DaoFactory	Used in the manufacture of other classes. Each database (MS Access, Sql Server, Oracle) has its own Factory which creates database specific data access classes.
Proxy	DataObjects	DataAccess	Used as easy data access interface for the business layer.
Singleton	DataObjects	Db	Used for low level data access. Only one stateless Db object is required for the entire application.
Strategy	Cart	IShipping ShippingStrategyFedex ShippingStrategyUPS ShippingStrategyUSPS	Used to selectively choose a different shipping method that computes shipping costs.

Composite	Controls	MenuComposite MenuCompositeItem	Used to represent the hierarchical tree structure of the menu.
Observer	Log	ObserverLogToDatabase ObserverLogToEmail ObserverLogToEventLog ObserverLogToFile	Used to 'listen to' error events and log messages to a logging output device, such as email, event log, etc.
Decorator	Transactions	TransactionDecorator	Used to 'embrace and extend' the functionality of the built-in TransactionScope class.
Provider (Microsoft Pattern)	ViewState	ViewStateProviderCache ViewStateProviderGlobal ViewStateProviderSession	Used to build several providers that can hold ViewState on the server and therefore limit the size of the pages being sent to the browser.
Singleton	ViewState	GlobalViewStateSingleton	Used to hold all available view state providers.
Iterator	All Projects	foreach language construct (For Each in VB)	Iterator is built in the .NET programming languages.
Observer	All Projects	.NET event model	Observer is built in the .NET programming languages.

Enterprise Design Patterns:

In 2003, Martin Fowler's book was published titled: "*Patterns of Enterprise Application Architecture*". Supposedly, it was written for both Java and .NET developers, but there is a slant towards the Java side of things. This book provides an extensive catalog of patterns and best practices used in developing data driven enterprise-level applications. The word 'pattern' is used more loosely in this book. It is best to think about these patterns as best practice solutions to commonly occurring enterprise application problems. It proposes a layered architecture in which presentation, domain model, and data source make up the three principal layers. This layering exactly matches the 3-tier model employed in *Patterns in Action 2.0*.

Initially, many of the Enterprise patterns in this book may seem trivial and irrelevant to .NET developers. The reason is that the .NET Framework has many Enterprise patterns built-in. This shields .NET developers from having to write any code that implements the design pattern. In fact, developers do not even have to know that there is a common pattern or best practice underlying the feature under consideration. There are many examples of this, such as, Client Session State (this is the Session object), Model View Controller (which is not really applicable to ASP.NET), Record Set (which is DataTable object), and the Page Controller (pages that derive from the Page class -- i.e. the code behind)

Even so, Fowler's book is useful in that it provides a clear catalog of patterns for those architecting large and complex enterprise level applications. It also offers a glimpse into the issues that Java developers have to be concerned about which is useful for teams that are supporting multiple platforms. In a sense, Microsoft has made many decisions by providing best practice solutions to common problems. For example, most developers do not question the DataTable object -- they simply use it. As a .NET developer, you could write your own in-memory database table, but most would never consider doing such a 'silly' thing.

The Java world was earlier in accepting and embracing the concept of Design Patterns and they got a head start. However, with more and more developers moving to the pure OO world of .NET the .NET community is catching up quickly.

Enterprise Design Patterns	Project	Class	Usage
Domain Model	BusinessObjects	Catalog, Product, Customer, Order, Order Detail	Business Objects is essentially a different name for Domain model.
Identity Field	BusinessObjects	Category, Product, Customer, Order	The identity value is the only link between the database record and the business object.
LazyLoad	BusinessObjects	ProxyList	The basic elements for applying the LazyLoad pattern are laid out in this base class.
LazyLoad	Façade	ProxyForOrderDetails ProxyForOrders	Implementation of the LazyLoad pattern outlined in the ProxyList base class;
Remote Façade	Façade	CustomerFacade ProductFacade	The API is course grained because it deals with business objects rather than attribute values.
Service Layer	Façade	CustomerFacade ProductFacade	A service layer that sits on top of the Domain model (business objects)
LazyLoad	Façade	CustomerFacade	LazyLoad proxy objects are assigned to Customer and Order business objects
Transaction Script	Façade	CustomerFacade ProductFacade	The Façade API is course grained and each method handles indivual presentation layer requests.
Table Module	Cart	ShoppingCart	Shopping Cart logic is handled by one class holding a DataTable. Cart is non-persistent, but by making it persistent it would be a single table..
Data Table Gateway	Cart	ShoppingCart	Again, if this were a persistent shopping cart one instance would handle all rows in the database table.
Transform View	Controls	MenuComposite	Menu items are processed and then transformed into HTML
LazyLoad	ViewState	ViewStateProviderService	ViewState Providers are loaded only when really necessary
Page Controller	Web Site	PageBase	Each page has its own controller code. This is how in ASP.NET pages are built.

			Shared page code is kept in PageBase class.
Data Transfer Object	WebSOAService	CustomerTransferObject OrderTransferObject OrderDetailTransferObject	Provides a way to transfer data between processes. These are objects that only hold data; they don't have behavior (properties or methods).

Service Oriented Architecture (SOA) Design Patterns:

The *Patterns in Action 2.0* reference application introduces and new group of design patterns, namely design patterns for SOA (Service Oriented Architecture). SOA is all about sharing functional components across an organization, or in many cases across the world. SOA offers a way to build applications as a set of services that are accessible through a message based interface. Web Services using XML and SOAP protocols transported over HTTP usually play a central role in this architecture.

Pattern In Action 2.0 demonstrates how the 3-tier model with Façade and other design patterns provide an excellent foundation to building Service Oriented Architectures with Web Services. Using a Web Service, you have the ability to expose and publish the functionality of an application to the Internet in a platform independent manner. The consumers (or clients) of your Web Service can be any device and / or platform that understands the XML / SOAP / HTTP protocols.

The cross-section diagram (Figure 7) earlier in this document shows that Web Services sit on top of the Façade, just like the Web Site. That is, the Web Service communicates with the Façade only and no additional Business Layer or Data Layer coding is required (or allowed). The Façade exposes a rich interface that internally handles the business logic as well as issues such as data validation, user authorization, and transaction management. The 3-tier model in combination with common design patterns makes it easy to expose the application's functionality to other external clients. In *Patterns in Action 2.0* the external client is a Windows application (WindowsSOAClient).

Note: SOA is young field and many of the patterns and best practices are still in the process of being discovered and need further solidification. *Patterns in Action 2.0* gives you a snapshot of the current state-of-the-art as the development community continues to discover solutions to the challenges that are unique to the world of SOA.

When working with SOA you will also run into GoF design patterns commonly used in the .NET Web Services architecture (Proxy, Command, and Observer come to mind) –

however, these are not highlighted in this discussion. The focus here is on some of the new patterns and best practices that are uniquely designed to address SOA-specific challenges.

So, what are these SOA-specific challenges? Actually, there are many. An important issue is that Web Services are subject to network latency and network failure. Furthermore, calling over remote connections is computationally expensive which results in slower response times. Another important consideration is that services are built around contracts (i.e. the service interface). Changing the interface will cause problems for existing clients. Web service interface designers need to be aware of this. Next, a more in-depth look follows at these service contracts and related topics.

SOA best practice design principles

Web Services provide a contract that defines its public interface. WSDL is used to inform the service consumer what is in the contract. The only way that service consumers interact with a service is through its public interface. Therefore, the design of the interface requires special attention. Below are some of the best practices (we prefer not to call them patterns at this point) that have evolved in the SOA space:

Keep the interface simple and small. Expose a minimal number of web methods that can handle different request scenarios. Favor 'chunky' interfaces over 'chatty' interfaces, that is, make fewer calls and pass more information with each call.

Keep your calls atomic and stateless. Web methods should not depend on each other. Each method call should be autonomous and execute a complete unit-of-work. Calls should be stateless, meaning that a call does not leave a state behind that the next call depends on. For example, don't do things like: call OpenResultSet and then make subsequent calls like GetNextRecord, GetLastRecord, etc. This would create a stateful contract in which a ResultSet is maintained between calls on the server.

Hide internal (private) data and processing details. Letting implementation details leak outside the interface leads to tight coupling. Tight coupling is undesirable because

giving the client access to these details gives it the opportunity to control how things are done. This prevents the service from evolving over time and doing things differently.

Always consider versioning. Versioning, the process of changing the service as requirements change is a complex topic. However, you must deal with it because change is an inevitable part of any successful web service. Backward compatibility needs to be maintained (at least for some time) or else your service clients will stop functioning. Once a contract (or API) is published, it cannot be modified. Some versioning approaches are listed next:

> *Amazon* uses a version number (release date) as part of it service URL. This results in the following URLs:
> www.companyname.com/service/01-12-06/ and
> www.companyname.com/service/03-08-06/ and
> www.companyname.com/service/ will always support the latest version.
>
> *Ebay* and *PayPal* include a version number and build number in their messages. Both service client and web service then are in complete agreement as to what version is being used. However, this does not solve the problem for when the API changes (method signature and/or argument type). Having messages in the form of a formatted XML string will prevent the API from changing though.
>
> It is possible to create web service with a single method that accepts a single argument as an XML or comma separated string. This argument contains an encoded request with items such as action, version number, and data. An internal dispatcher (or router) then redirects the request to the appropriate programs once it reaches the web server. This model is sometimes referred to as a 'super-router'. It will always be backward compatible, despite changes in the XML schema. This may seem attractive, but it is generally not a recommended approach because it renders the interface (or contract) meaningless. A contract should have clearly defined service semantics.

Assume the worst. From a service consumer perspective, you need to take the approach that the service will be unreliable, slow, and may not even be available at

times. The goal is to build reliable systems that continue to function even in the unpredictable and unreliable SOA space.

Think in terms of message end-points. There are two ways to look at Web Services. One as objects (or components) deployed on the web, and the other as messaging end-points. This difference may seem unimportant, but it does influence how interfaces are designed. From an OO perspective the objects on the web view is weak because Web Services do not support inheritance or (true) polymorphism. Furthermore, an object view would suggest RPC type interfaces, which is not desirable because it turns out that these are not very flexible. Interfaces designed around messages offer developers a more flexible model in which messages evolve over time with minimum impact on the clients. When a web service is viewed as a group of message end-points, the emphasis shifts from public methods to public messages.

Now that we have discusses some of principles and best practices, it is time to examine some 'early discovered' SOA patterns that are used in *Patterns in Action 2.0*.

Document-Centric Design Pattern

One of the challenges service designers face is how do define contracts that are flexible and are compliant with the SOA design principles discussed above. One of the answers is that they must think in terms of document-centric APIs. A document in this context is simply a unit of information, a complete package of data that represents something of value to the business.

Frequently these documents or messages are composed of smaller pieces of information. For example, *Patterns in Action 2.0* has a service method that responds with a document that contains a sorted array of Customers, but it also comes with other information relevant to the client who is requesting the document. They include a version number, an acknowledgement, a correlationId, and an error message. Some of these data items are discussed later.

Document-centric services evolve easier because the actions and associated data are contained in a single, more flexible, message rather than an RPC method with hard-coded arguments.

Say, you have a web method with the following signature:

C#: bool SaveEmployee(int, string, double),
VB: SaveEmployee(Integer, String, Double) As Boolean

You have deployed your service and clients are using the service. After a while, you realize that an additional argument is needed, for example a String. With the above method signature you're out of luck because the additional argument will violate the contract that you have with your service clients. Document-centric contracts are easier to evolve since all information exchange occurs within the document payload instead of a fixed and hard-to-change RPC method.

A web service cannot demand or expect that clients use the web service methods in a certain manner or that they call methods in a pre-defined order. Each service method should be atomic and run as single transaction. This suggests that larger and more complete sets of data are exchanged. Of course, there will always be a need for long running transactions (hours or days) and for this there is another pattern: the reservation design pattern which is discussed later in this document.

Request-Response Design Pattern

Now that we agree on document-centric services, what does the method signature (or API) look like? We want to avoid 'RPC' type interfaces in which methods have a fixed number of arguments. RPC type methods look like this:

boolean = *Method* (integer, string, double)

Instead, it is better think in terms of a message based calling pattern where messages represent a complete unit of work and follow a request / response model. The model is

simple: send a request message (with all the required information) and then respond with a response message (containing the entire result set and all its data). The method signature looks like this:

response = *Method* (request)

It is simple, consistent, easy to understand, and yet offers far more flexibility than standard method signatures. All web service methods in *Patterns In Action 2.0* follow this exact same request / response pattern.

Reservation Design Pattern

Web services methods should be atomic, stateless, and are independent from one another. However, there are scenarios where this is not possible. For example, what do you do if a method requires a long running complex business process that takes hours or days to complete?

Let's say you're building a loan application system. The ApplyForLoanApproval method may require several steps, which take a couple of days. The application may have to go through a series of credit and background checks and perhaps include human intervention by a loan officer. What is needed here is a loan application number -- more generically called a reservation number. This will allow the client to request status or provide additional information if necessary.

In *Patterns In Action 2.0*, the response message payloads include a reservation number and a reservation expiration date. However, there are no long running transactions and therefore these data members are not used.

Idempotent Design Pattern:

One of the challenges with SOA is that the service provider has virtually no control over the service client. A client could be repeating the same request multiple times and this

can have adverse side effects. For example, running a financial transaction multiple times instead of just once can be bad (or good, it depends who you're talking to). Another example: updating all employee salaries by 3% multiple times is probably not what you want.

The Idempotent Design Pattern is designed to address this problem. Idempotent is a term that comes from mathematics meaning: "Acting as if used only once, even if used multiple times". In SOA, it means that a request can be sent multiple times without adverse effect. The pattern requires that every request message be tagged with a unique request identifier (also called a unit-of-work). A good candidate for such an identifier is a GUID. The Web Service needs to keep track of the request identifiers by maintaining a buffer of previously received request identifiers. Then, when the same request is sent multiple times it can respond accordingly. This approach ensures that a request be processed only once. Of course, the service has different options on how to respond to repeated requests; it could throw an exception, return an error, return a cached response, or it could simple run the same process again.

This is a first line of defense, not a complete answer to a difficult to handle problem. Repeated requests can take different forms, such as, a 'service of denial' attack (whereby a client maliciously floods the server with an overwhelming number of requests), or a client going haywire in an incorrectly written loop, or a user on the client side who pushes the submit button too often and too fast.

To confirm receipt the service returns the *requestId* to client in a variable called *correlationId*. This is particularly important for asynchronous web request calls (which are not included in the *Patterns in Action 2.0* reference application).

Message Router Design Pattern:

As an alternative to using the name of the service method, the message payload may also contain instructions about which operations to perform (i.e. what class to invoke or what method to call). This approach of sending messages that include instructions on where to route the request is called a Message Router Design Pattern. Another name for

this pattern is Command Message – this, of course, because it is closely related to GoF's Command pattern.

In *Patterns in Action 2.0* the Message Router design pattern in demonstrated by the PersistCustomer web method and PersistCustomerRequest message. This message contains a complete customer record together with an Insert, Update, or Delete instruction. This example is relatively simple, but it could easily be enhanced with arrays of data records and additional operations.

Private Identifier Design Pattern:

Instances of business objects usually have their own identity. This allows us to distinguish one object instance from another and makes it possible to locate a particular object among others. The need for a unique object identity is analogous to database records, which have unique primary key values. In fact, in most applications the business objects use the database primary key values as their identifiers.

The business layer in *Patterns in Action 2.0* has Customer, Order, Product, and Category business objects have identifiers with the same values as the surrogate keys in the associated database records (note: surrogate keys are the auto-number values in MS Access and identity values in Sql Server). These identifiers work well, because they are immutable and unique within their class. Once a primary key value is generated, it will never change.

Object identifiers are part of the data transfer objects that are transported in SOA messages. Since web services have no control over the clients, you want to protect the actual database primary key values because they may pose a security risk. For example, if the identifiers are simple integer identity values, then it does not take much for the client to start guessing valid identifiers.

The Private Identifier Design Patterns addresses this challenge by encrypting the identifiers in such a way that it guarantees that these encrypted values do not change over time. This non-changing characteristic is important because a client may choose to

store these objects in a local database, make changes locally, and then perform a batch update to the server later. So, it is important that both client and service understand the encrypted identifiers and that they be immutable.

Patterns in Action 2.0 uses the .NET built-in tripleDes algorithm to perform the encryption. The required *key* and *iv* vector are hard-coded in the code. In your own work, you will want to generate a different key and iv and store these at a secure place, ideally outside the code such as in the Windows registry.

Patterns in Action 2.0 demonstrates primary key encryption only. If foreign keys are present in your business objects, you must remember to encrypt these as well. Use the same encryption algorithm and keys as for the associated primary keys or else primary and foreign key mismatches will occur. It should be noted that foreign key values are changeable by the client, but primary key values are not. Again, this is similar to what you find in database operations.

Below is a summary of SOA patterns in *Patterns in Action 2.0.*

SOA Design Patterns	Project	Class	Usage
Document Centric	WebSOAService	All messages	Each service method represent a business process as a complete unit of work. This requires a document-centric viewpoint. No stateful type interfaces.
Request-Response	WebSOAService	Service.cs (.vb)	All methods in the the service interface follows the same simple, yet flexible signature model: response = method (request)
Reservation	WebSOAService	MessageBase	A way to support long running transactions without explicit statefulness on the server.
Idempotent	WebSOAService	MessageBase	Tag each request with a request number and return it as a correlation identifiers in the response messages.
Message Router	WebSOAService	PersistType	Embed the operation to be executed by the receiver in the actual message. Also called Command design pattern.
Private Identifier	WebSOAService	MessageBase	Hide internal identifiers (primary key and foreign key values) from external clients.

Summary

Patterns In Action 2.0 is a reference application that demonstrates where, when, and how design patterns are used in a modern, 3-tier, enterprise level, e-commerce environment. We are hopeful that now that you have studied the application you are convinced that design patterns form an integral part in modern-day application architecture. Design patterns 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 *Patterns in Action 2.0*, or have suggestion for future enhancements and improvements please do not hesitate to contact us. Contact us via email at info@dofactory.com, or via our 'contact us' page on our website at www.dofactory.com/contact/contact.aspx. We look forward to hearing from you.

Good luck in your future architecting and design pattern endeavors.