```
                <br>
            Unit Price:  $<%# Eval("UnitPrice")%>
                <br>
        </ItemTemplate>
</asp:Repeater>
```

In this ASPX file, we only have a Repeater control, which we will bind with the data in the code-behind file.

Here is the code in the `ProductList.aspx.cs` code-behind file:

```
namespace OMS
{
public partial class _Default : System.Web.UI.Page
    {
        /// <summary>
        /// Page Load method
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        protected void Page_Load(object sender, EventArgs e)
        {
            DataTable dt = DAL.GetAllProducts();
            prodRepeater.DataSource = dt;
            prodRepeater.DataBind();
        }
    }//end class
}//end namespace
```

Note that we don't have any data access code in the code-behind sample above. We are just calling the `GetAllProducts()` method, which has all of data access code wrapped in a different class named DAL. As we saw in the last section of Chapter 1, we can logically separate out the code, by using different namespaces to achieve code re-use and greater architectural flexibility. So we created a new class named DAL under a different namespace from the UI layer code files. Here is the DAL code:

```
namespace OMS.Code
{
    public class DAL
    {
        /// <summary>
        /// Load all comments from the Access DB
        /// </summary>
        public static DataTable GetAllProducts()
        {
            string sCon = ConfigurationManager.ConnectionStrings[0].
```

```
ConnectionString;
            using (SqlConnection cn = new SqlConnection(sCon))
            {
                string sQuery = @"SELECT * FROM OMS_Product";
                SqlCommand cmd = new SqlCommand(sQuery, cn);
                SqlDataAdapter da = new SqlDataAdapter(cmd);
                DataSet ds = new DataSet();
                cn.Open();
                da.Fill(ds);
                return ds.Tables[0];
            }
        }
    }//end class
}//end namespace
```

So we have separated the data access code in a new logical layer, using a separate namespace, `OMS.Code`, and using a new class. Now, if we want to, we can re-use the same code in the other pages as well. Furthermore, methods to add and edit a product can be defined in this class and then used in the UI layer. This allows multiple developers to work on the DAL and UI layers simultaneously.

Even though we have a logical separation of the code in this 2-layer sample architecture, we are still not using real Object Oriented Programming (OOP). All of the Object-Oriented Programming we have used so far has been the default structure the .NET framework has provided, such as the Page class, and so on.

When a project grows big in size as well as complexity, using the 2-layer model discussed above can become cumbersome and cause scalability and flexibility issues. If the project grows in complexity, then we will be putting all of the business logic code in either the DAL or the UI layer. This business logic code includes business rules. For example, if the customer orders a certain number of products in one order, he gets a certain level of discount. If we code such business rules in the UI layer, then if the rules change we need to change the UI as well, which is not ideal, especially in cases where we can have multiple UIs for the same code, for example one normal web browser UI and another mobile-based UI.

We also cannot put business logic code in the DAL layer because the DAL layer should only contain data access code which should not be mixed with any kind of business processing logic. In fact the DAL layer should be quite "dumb"–there should be no "logic" inside it because it is mostly a utility layer which only needs to put data in and pull data out from a data store.

To make our applications more scalable and to reap the benefit of OOP, we need to create objects, and wrap business behavior in their methods. This is where the Domain Model comes into the picture.