Microsoft

Foreword by Luca Bolognese
LINQ Principal Program Manager, Microsoft Corporation

Programming Microsoft® LINQ

Paolo Pialorsi
Marco Russo

Chapter 6

Tools for LINQ to SQL

The best way to write queries using LINQ to SQL is by having a *DataContext*-derived class in your code that exposes all the tables, stored procedures, and user-defined functions you need as properties of a class instance. You also need entity classes that are mapped to the database objects. As you have seen in previous chapters, this mapping can be made by using attributes to decorate classes or through an external XML mapping file. However, writing this information by hand is tedious and error-prone work. You need some tools to help you accomplish this work.

In this chapter, you will learn about what file types are involved and what tools are available to automatically generate this information. The .NET 3.5 Software Development Kit (SDK) includes a command-line tool named SQLMetal. Microsoft Visual Studio 2008 offers an integrated graphical tool named the Object Relational Designer. We will examine both tools from a practical point of view.

Important In this chapter we use the version of the Northwind database that is included in the C# samples provided with Visual Studio 2008. All the samples are contained in the Microsoft Visual Studio 9.0\Samples\1033\CSharpSamples.zip file in your program files directory if you installed Visual Studio 2008. You can also download an updated version of these samples from *http://code.msdn.microsoft.com/csharpsamples*.

File Types

There are three types of files involved in LINQ to SQL entities and a mapping definition:

- Database markup language (.DBML)
- Source code (C# or Visual Basic)
- External mapping file (XML)

A common mistake is the confusion between DBML and XML mapping files. At first sight, these two files are similar, but they are very different in their use and generation process.

DBML—Database Markup Language

The DBML file contains a description of the LINQ to SQL entities in a database markup language. Visual Studio 2008 installs a DbmlSchema.xsd file, which contains the schema definition of that language and can be used to validate a DBML file. The namespace used for this

file is *http://schemas.microsoft.com/linqtosql/dbml/2007*, which is different from the namespace used by the XSD for the XML external mapping file.

Note You can find the DbmlSchema.xsd schema file in the %ProgramFiles(x86)%\Microsoft Visual Studio 9.0\Xml\Schemas folder.

The DBML file can be automatically generated by extracting metadata from an existing Microsoft SQL Server database. However, the DBML file includes more information than can be inferred from database tables. For example, settings for synchronization and delayed loading are specific to the intended use of the entity. Moreover, DBML files include information that is used only by the code generator that generates C# or Visual Basic source code, such as the base class and namespace for generated entity classes. Listing 6-1 shows an excerpt from a sample DBML file.

Listing 6-1 Excerpt from a sample DBML file

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="Northwind" Class="nwDataContext"
          xmlns="http://schemas.microsoft.com/linqtosql/dbml/2007">
  <Connection Mode="AppSettings"
              ConnectionString="Data Source=..."
              SettingsObjectName="DevLeap.Linq.LinqToSql.Properties.Settings"
              SettingsPropertyName="NorthwindConnectionString"
              Provider="System.Data.SqlClient" />
  <Table Name="dbo.Orders" Member="Orders">
    <Type Name="Order">
      <Column Name="OrderID" Type="System.Int32"
              DbType="Int NOT NULL IDENTITY" IsPrimaryKey="true"
              IsDbGenerated="true" CanBeNull="false" />
      <Column Name="CustomerID" Type="System.String"
              DbType="NChar(5)" CanBeNull="true" />
      <Column Name="OrderDate" Type="System.DateTime"
              DbType="DateTime" CanBeNull="true" />

      ...

      <Association Name="Customer_Order" Member="Customer"
                   ThisKey="CustomerID" Type="Customer"
                   IsForeignKey="true" />
    </Type>
  </Table>
      ...
</Database>
```

The DBML file is the richest container of metadata information for LINQ to SQL. Usually, it can be generated from a SQL Server database and then manually modified, adding information that cannot be inferred from the database. This is the typical approach when using the SQLMetal command-line tool. The Object Relational Designer included in Visual Studio 2008

offers a more dynamic way of editing this file, because programmers can import entities from a database and modify them directly in the DBML file through a graphical editor. The DBML generated by SQLMetal can also be edited with the Object Relational Designer.

The DBML file can be used to generate C# or Visual Basic source code for entities and *DataContext*-derived classes. Optionally, it can also be used to generate an external XML mapping file.

More Info It is beyond the scope of this book to provide a detailed description of the DBML syntax. You can find more information and the whole DbmlSchema.xsd content in the product documentation at *http://msdn2.microsoft.com/library/bb399400.aspx*.

C# and Visual Basic Source Code

The source code written in C#, Visual Basic, or any other .NET language contains the definition of LINQ to SQL entity classes. This code can be decorated with attributes that define the mapping of entities and their properties with database tables and their columns. Otherwise, the mapping can be defined by an external XML mapping file. However, a mix of both is not allowed—you have to choose only one place where the mappings of an entity are defined.

This source code can be automatically generated by tools such as SQLMetal directly from a SQL Server database. The code-generation function of SQLMetal can translate a DBML file to C# or Visual Basic source code. When you ask SQLMetal to directly generate the source code for entities, internally it generates the DBML file that is converted to the entity source code. In Listing 6-2, you can see an excerpt of the C# source code generated for LINQ to SQL entities that were generated from the DBML sample shown in Listing 6-1.

Listing 6-2 Excerpt from the class entity source code in C#

```
[System.Data.Linq.Mapping.DatabaseAttribute(Name="Northwind")]
public partial class nwDataContext : System.Data.Linq.DataContext {

    // ...

    public System.Data.Linq.Table<Order> Orders {
        get { return this.GetTable<Order>(); }
    }
}
 [Table(Name="dbo.Orders")]
public partial class Order : INotifyPropertyChanging, INotifyPropertyChanged {
    private int _OrderID;
    private string _CustomerID;
    private System.Nullable<System.DateTime> _OrderDate;

    [Column(Storage="_OrderID", AutoSync=AutoSync.OnInsert,
            DbType="Int NOT NULL IDENTITY", IsPrimaryKey=true,
            IsDbGenerated=true)]
```

```
public int OrderID {
    get { return this._OrderID; }
    set {
        if ((this._OrderID != value)) {
            this.OnOrderIDChanging(value);
            this.SendPropertyChanging();
            this._OrderID = value;
            this.SendPropertyChanged("OrderID");
            this.OnOrderIDChanged();
        }
    }
}

[Column(Storage="_CustomerID", DbType="NChar(5)")]
public string CustomerID {
    get { return this._CustomerID; }
    set {
        if ((this._CustomerID != value)) {
            if (this._Customer.HasLoadedOrAssignedValue) {
                throw new ForeignKeyReferenceAlreadyHasValueException();
            }
            this.OnCustomerIDChanging(value);
            this.SendPropertyChanging();
            this._CustomerID = value;
            this.SendPropertyChanged("CustomerID");
            this.OnCustomerIDChanged();
        }
    }
}

[Column(Storage="_OrderDate", DbType="DateTime")]
public System.Nullable<System.DateTime> OrderDate {
    get { return this._OrderDate; }
    set {
        if ((this._OrderDate != value)) {
            this.OnOrderDateChanging(value);
            this.SendPropertyChanging();
            this._OrderDate = value;
            this.SendPropertyChanged("OrderDate");
            this.OnOrderDateChanged();
        }
    }
}

[Association(Name="Customer_Order", Storage="_Customer",
             ThisKey="CustomerID", IsForeignKey=true)]
public Customer Customer {
    get { return this._Customer.Entity; }
    set {
        Customer previousValue = this._Customer.Entity;
        if ((previousValue != value)
             || (this._Customer.HasLoadedOrAssignedValue == false)) {
            this.SendPropertyChanging();
```

```
                if ((previousValue != null)) {
                    this._Customer.Entity = null;
                    previousValue.Orders.Remove(this);
                }
                this._Customer.Entity = value;
                if ((value != null)) {
                    value.Orders.Add(this);
                    this._CustomerID = value.CustomerID;
                }
                else {
                    this._CustomerID = default(string);
                }
                this.SendPropertyChanged("Customer");
            }
        }
    }

    // ...
}
```

The attributes that are highlighted in bold in Listing 6-2 are not generated in the source code file when you have SQLMetal generate both the source code file and an external XML mapping file. The XML mapping file will contain this mapping information.

More Info Attributes that define the mapping between entities and database tables are discussed in Chapter 4, "LINQ to SQL: Querying Data," and in Chapter 5, "LINQ to SQL: Managing Data."

XML—External Mapping File

An external mapping file can contain binding metadata for LINQ to SQL entities as an alternative way to store them in code attributes. This file is an XML file with a schema that is a subset of the DBML file. The DBML file also contains information useful for code generators. Attributes defined on class entities are ignored whenever they are included in the definitions of an external mapping file.

The namespace used for this file is *http://schemas.microsoft.com/linqtosql/mapping/2007*, which is different from the one used by the DBML XSD file.

Note The LinqToSqlMapping.xsd schema file should be located in the %ProgramFiles(x86)%\Microsoft Visual Studio 9.0\Xml\Schemas folder. If you do not have that file, you can create it by copying the code from the documentation page at *http://msdn2.microsoft.com/library/bb386907.aspx*.

In Listing 6-3, you can see an example of an external mapping file generated from the DBML file presented in Listing 6-1. We highlighted the *Storage* attribute that defines the mapping between the table column and the data member in the entity class that stores the value exposed through the member property (defined by the *Member* attribute). The value assigned to *Storage* depends on the implementation generated by the code generator; for this reason, it is not included in the DBML file.

Listing 6-3 Excerpt from a sample XML mapping file

```
<?xml version="1.0" encoding="utf-8"?>
<Database Name="northwind"
          xmlns="http://schemas.microsoft.com/linqtosql/mapping/2007">
  <Table Name="dbo.Orders" Member="Orders">
    <Type Name="Orders">
      <Column Name="OrderID" Member="OrderID" Storage="_OrderID"
              DbType="Int NOT NULL IDENTITY" IsPrimaryKey="true"
              IsDbGenerated="true" AutoSync="OnInsert" />
      <Column Name="CustomerID" Member="CustomerID" Storage="_CustomerID"
              DbType="NChar(5)" />
      <Column Name="OrderDate" Member="OrderDate" Storage="_OrderDate"
              DbType="DateTime" />

      ...

      <Association Name="FK_Orders_Customers" Member="Customers"
                   Storage="_Customers" ThisKey="CustomerID"
                   OtherKey="CustomerID" IsForeignKey="true" />
    </Type>
  </Table>
      ...
</Database>
```

More Info If a provider has custom definitions that extend existing ones, the extensions are available only through an external mapping file but not with attribute-based mapping. For example, with an XML mapping file you can specify different *DbType* values for SQL Server 2000, SQL Server 2005, and SQL Server Compact 3.5. External XML mapping files are discussed in Chapter 5.

LINQ to SQL File Generation

Usually, most of the files used in LINQ to SQL are automatically generated by some tool. The diagram in Figure 6-1 illustrates the relationships between the different file types and the relational database. In the remaining part of this section, we will describe the most important patterns of code generation that you can use.

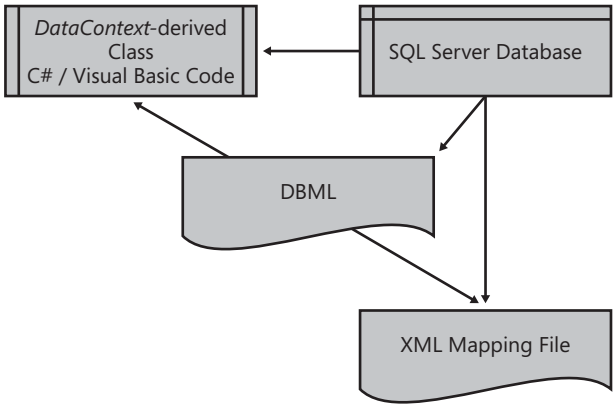


Figure 6-1 Relationships between file types and the relational database

Generating a DBML File from an Existing Database

If you have a relational database, you can generate a DBML file that describes tables, views, stored procedures, and user-defined functions, mapping them to class entities that can be created by a code generator. After it is created, the DBML file can be edited using a text editor or the Object Relational Designer included in Visual Studio 2008.

Generating an Entity's Source Code with Attribute-Based Mapping

You can choose to generate source code for class entities in C# or Visual Basic with attribute-based mapping. This code can be generated from a DBML file or directly from a SQL Server database.

If you start from a DBML file, you can still modify that DBML file and then regenerate the source code. In this case, the generated source code should not be modified because it could be overwritten in the future by code regeneration. You can customize generated classes by using a separate source code file, leveraging the *partial class* declaration of generated class entities. This is the pattern used when working with the Object Relational Designer.

If you generate code directly from a SQL Server database, the resulting source code file can still be customized using partial classes; however, if you need to modify the mapping settings, you have to modify the generated source code. In this case, you probably will not regenerate this file in the future and can therefore make modifications directly on the generated source code in C# or Visual Basic.

Generating an Entity's Source Code with an External XML Mapping File

You can choose to generate source code for class entities in C# or Visual Basic together with an external XML mapping file. The source code and the XML mapping file can be generated from a DBML file or directly from a SQL Server database.

If you start from a DBML file, you can still modify that DBML file and then regenerate the source code and the mapping file. In this case, the generated files should not be modified because they could be overwritten in the future by code regeneration. You can customize generated classes by using a separate source code file, leveraging the *partial class* declaration of the generated class entities. This is the pattern used when you work with the Object Relational Designer.

If you generate code directly from a SQL Server database, the resulting source code file can still be customized using partial classes. Because the mapping information is stored in a separate XML file, you need to modify that file to customize mapping settings. Most likely, you will not regenerate these files in the future and can therefore make modifications directly on the generated files.

Creating a DBML File from Scratch

You can start writing a DBML file from scratch. In this case, you probably would not have an existing database file and would generate the database by calling the *DataContext.CreateDatabase* method on an instance of the generated class inherited from *DataContext*. This approach is theoretically possible when you write the XML file with a text editor, but in practice we expect that it will be done only by using the Object Relational Designer.

Choosing this approach means that entity classes are more important than the database design, and the database design itself is only a consequence of the object model you designed for your application. In other words, you see the relational database as a simple persistence layer (without stored procedures, triggers, and other database-specific features), which should not be accessed directly by consumers that are not using the LINQ to SQL engine. In the real world, we have found this can be the case for applications that use the database as the storage mechanism for complex configurations or to persist very simple information, typically in a stand-alone application with a local database. Whenever a client-server or multitier architecture is involved, chances are that additional consumer applications will access the same database—for example, a tool to generate reports, such as Reporting Services. These scenarios are more database-centric and require better control of the database design, removing the DBML-first approach as a viable option. In these situations, the best way of working is to define the database schema and the domain model separately and then map the entities of the domain model on the database tables.

SQLMetal

SQLMetal is a code-generation command-line tool that can be used to do the following:

- Generate a DBML file from a database
- Generate an entity's source code (and optionally a mapping file) from a database
- Generate an entity's source code (and optionally a mapping file) from a DBML file

The syntax for SQLMetal is the following:

```
sqlmetal [options] [<input file>]
```

In the following sections, we will provide several examples that demonstrate how to use SQLMetal.

More Info A complete description of the SQLMetal command-line options is available at *http://msdn2.microsoft.com/library/bb386987.aspx*.

Generating a DBML File from a Database

To generate a DBML file, you need to specify the */dbml* option, followed by the filename to create. The syntax to specify the database to use depends on the type of the database. For example, a standard SQL Server database can be specified with the */server* and */database* options:

```
sqlmetal /server:localhost /database:Northwind /dbml:northwind.dbml
```

Windows authentication is used by default. If you want to use SQL Server authentication, you can use the */user* and */password* options. Alternatively, you can use the */conn* option, which cannot be used with */server*, */database*, */user*, or */password*. The following command line that uses */conn* is equivalent to the previous one, which used */server* and */database*:

```
sqlmetal /conn:"Server=localhost;Database=Northwind;Integrated Security=yes"
    /dbml:northwind.dbml
```

If you have the Northwind MDF file in the current directory and are using SQL Server Express, the same result can be obtained by using the following line, which makes use of the input file parameter:

```
sqlmetal /dbml:northwind.dbml Northwnd.mdf
```

Similarly, an SDF file handled by SQL Server Compact 3.5 can be specified as in the following line:

```
sqlmetal /dbml:northwind.dbml Northwind.sdf
```

By default, only tables are extracted from a database. You can also extract views, user-defined functions, and stored procedures by using */views*, */functions*, and */sprocs*, respectively, as shown here:

```
sqlmetal /server:localhost /database:Northwind /views /functions /sprocs
    /dbml:northwind.dbml
```

Note Remember that database views are treated like tables by LINQ to SQL.

Generating Source Code and a Mapping File from a Database

To generate an entity's source code, you need to specify the */code* option, followed by the filename to create. The language is inferred by the filename extension, using *CS* for C# and *VB* for Visual Basic. However, you can explicitly specify a language by using */language:csharp* or */language:vb* to get C# or Visual Basic code, respectively. The syntax to specify the database to use depends on the type of the database. A description of this syntax can be found in the preceding section, "Generating a DBML File from a Database."

For example, the following line generates C# source code for entities extracted from the Northwind database:

```
sqlmetal /server:localhost /database:Northwind /code:Northwind.cs
```

If you want all the tables and the views in Visual Basic, you can use the following command line:

```
sqlmetal /server:localhost /database:Northwind /views /code:Northwind.vb
```

Optionally, you can add the generation of an XML mapping file by using the */map* option, as in the following command line:

```
sqlmetal /server:localhost /database:Northwind /code:Northwind.cs /map:Northwind.xml
```

Important When the XML mapping file is requested, the generated source code does not contain any attribute-based mapping.

There are a few options to control how the entity classes are generated. The */namespace* option controls the namespace of the generated code. (By default, there is no namespace.) The */context* option specifies the name of the class inherited from *DataContext* that will be generated. (By default, it is derived from the database name.) The */entitybase* option allows you to define the base class of the generated entity classes. (By default, there is no base class.) For example, the following command line generates all the entities in a *LinqBook* namespace, deriving them from the *DevLeap.LinqBase* base class:

```
sqlmetal /server:localhost /database:Northwind /namespace:LinqBook
    /entitybase:DevLeap.LinqBase /code:Northwind.cs
```

Note If you specify a base class, you have to be sure that the class exists when the generated source code is compiled. It is a good practice to specify the full name of the base class.

If you want to generate serializable classes, you can specify */serialization:unidirectional* in the command line, as in the following example:

```
sqlmetal /server:localhost /database:Northwind /serialization:unidirectional
    /code:Northwind.cs
```

More Info See the section "Entity Serialization" in Chapter 5 for further information about serialization of LINQ to SQL entities, as well as Chapter 18, "LINQ and the Windows Communication Foundation."

Finally, there is a */pluralize* option that controls how the names of entities and properties are generated. When this option is specified, the entity names generated are singular, but table names in the *DataContext*-derived class properties are plural, regardless of the table name's form. In other words, the Customer (or Customers) table generates a *Customer* entity class and a *Customers* property in the *DataContext*-derived class.

Generating Source Code and a Mapping File from a DBML File

The generation of source code and a mapping file from a DBML file is identical to the syntax required to generate the same results from a database. The only change is that instead of specifying a database connection, you have to specify the DBML filename as an input file parameter of the command-line syntax. For example, the following command line generates the C# class code for the Northwind.DBML model description:

```
sqlmetal /code:Northwind.cs Northwind.dbml
```

Important Remember to use the */dbml* option only to generate a DBML file. You do not have to specify */dbml* when you want to use a DBML file as input.

You can use all the options for generating source code and a mapping file that we described in the "Generating Source Code and a Mapping File from a Database" section.

Using the Object Relational Designer

The Object Relational Designer (O/R Designer) is a graphical editor integrated with Visual Studio 2008. It is the standard editor for a DBML file. It allows you to create new entities, edit existing ones, and generate an entity starting from an object in a SQL Server database. (There is support for tables, views, stored procedures, and user-defined functions.) A DBML file can be created by choosing the LINQ To SQL Classes template in the Add New Item dialog box, which you can see in Figure 6-2, or by adding an existing DBML file to a project (using the Add Existing Item command and picking the Data category).

The design surface allows you to drag items from a connection opened in Server Explorer. Dragging an item results in the creation of a new entity deriving its content from the imported object. Alternatively, you can create new entities by dragging items such as Class, Association, and Inheritance from the toolbox. In Figure 6-3, you can see an empty DMBL file opened in Microsoft Visual Studio. On the left are the Toolbox and Server Explorer elements ready to be dragged onto the design surface.

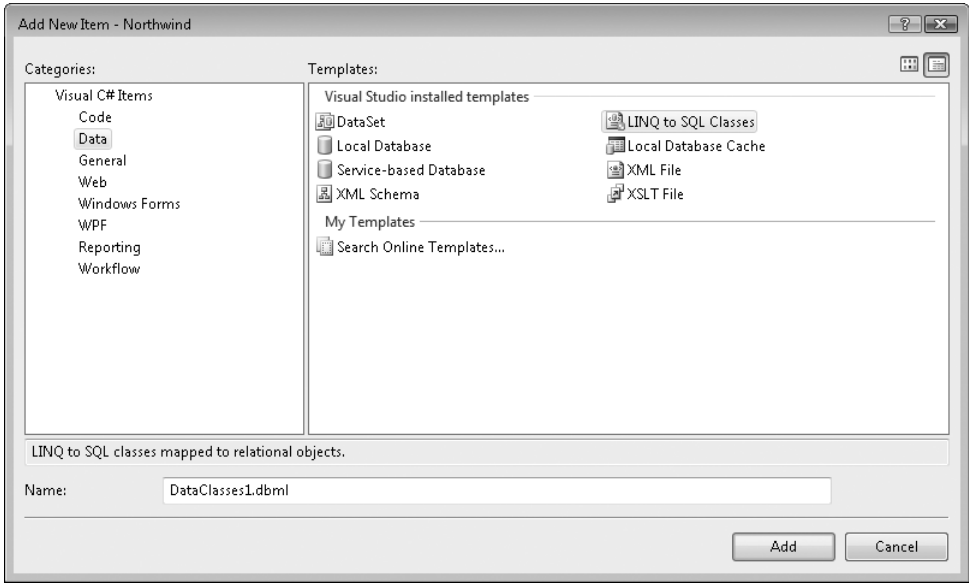


Figure 6-2 Add New Item dialog box

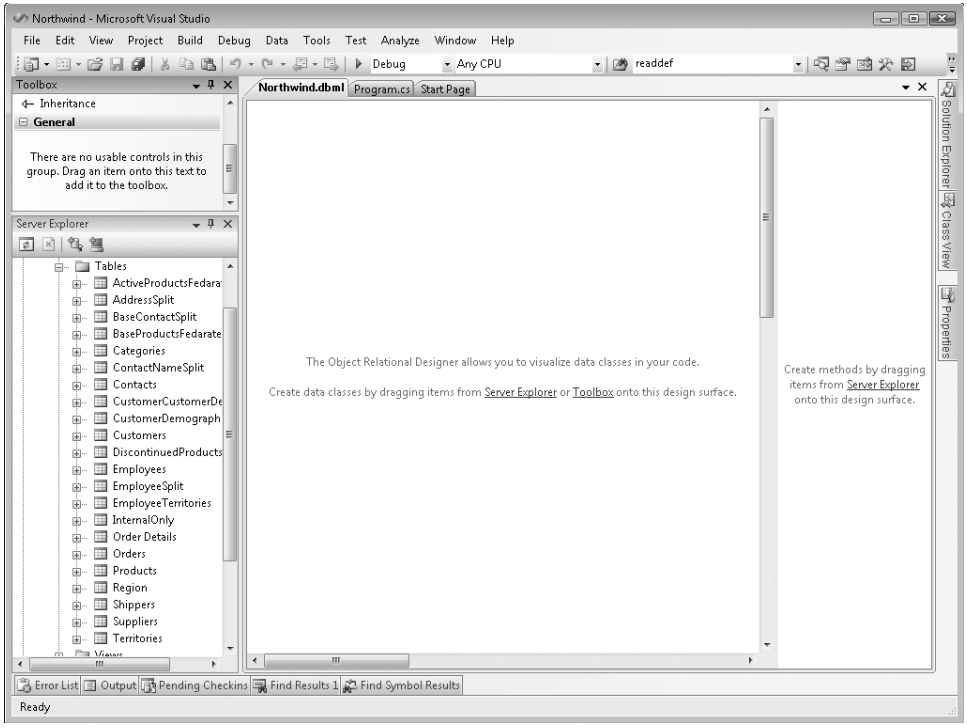


Figure 6-3 Empty DBML file opened with the Object Relational Designer

Dragging two tables, Orders and Order Details, from Server Explorer to the left pane of the DBML design surface results in a DBML file that contains two entity classes, *Order* and

Order_Detail, as you can see in Figure 6-4. Because a foreign key constraint exists in the database between the Order Details and Orders tables, an *Association* between the *Order* and *Order_Detail* entities is generated too.

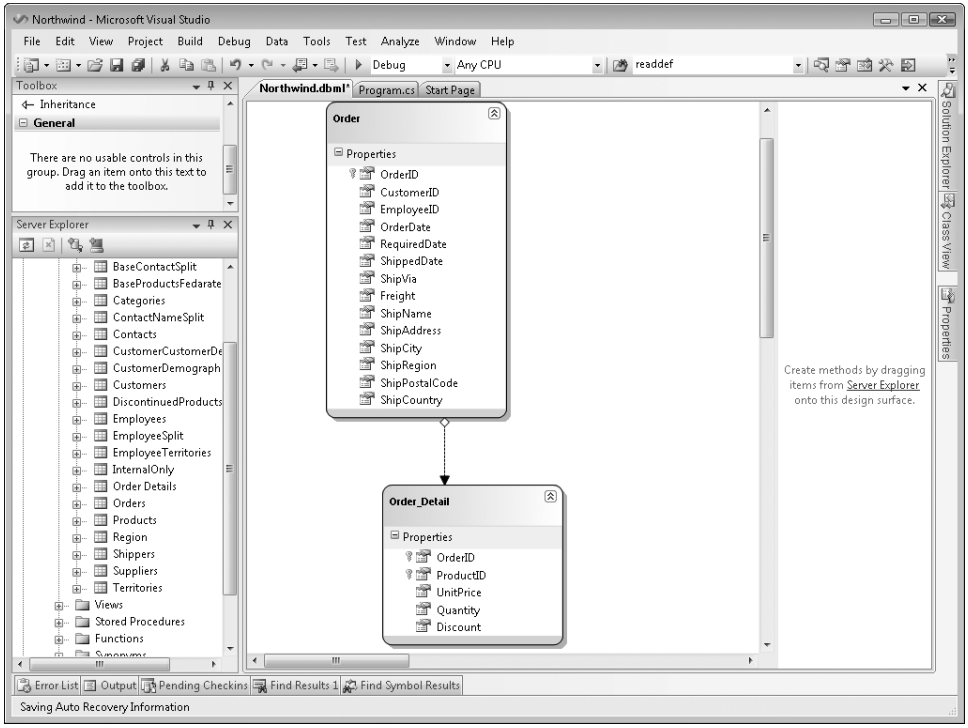


Figure 6-4 Two entities created from a server connection

You can see that plural names have been translated into singular-name entity classes. However, the names of the *Table<T>* properties in the *NorthwindDataContext* class are plural (*Orders* and *Order_Details*), as you can see in the bottom part of the Class View shown in Figure 6-5.

The Class View is updated by Visual Studio 2008 each time you save the DBML file. Every time that this file is saved, two other files are saved too: a *.layout* file, which is an XML file containing information about the design surface, and a *.cs/.vb* file, which is the source code generated for the entity classes. In other words, each time a DBML file is saved from Visual Studio 2008, the code generator is run on the DBML file and the source code for those entities is updated. In Figure 6-6, you can see the files related to our *Northwind.dbml* in Solution Explorer. We have a *Northwind.dbml.layout* file and a *Northwind.designer.cs* file.

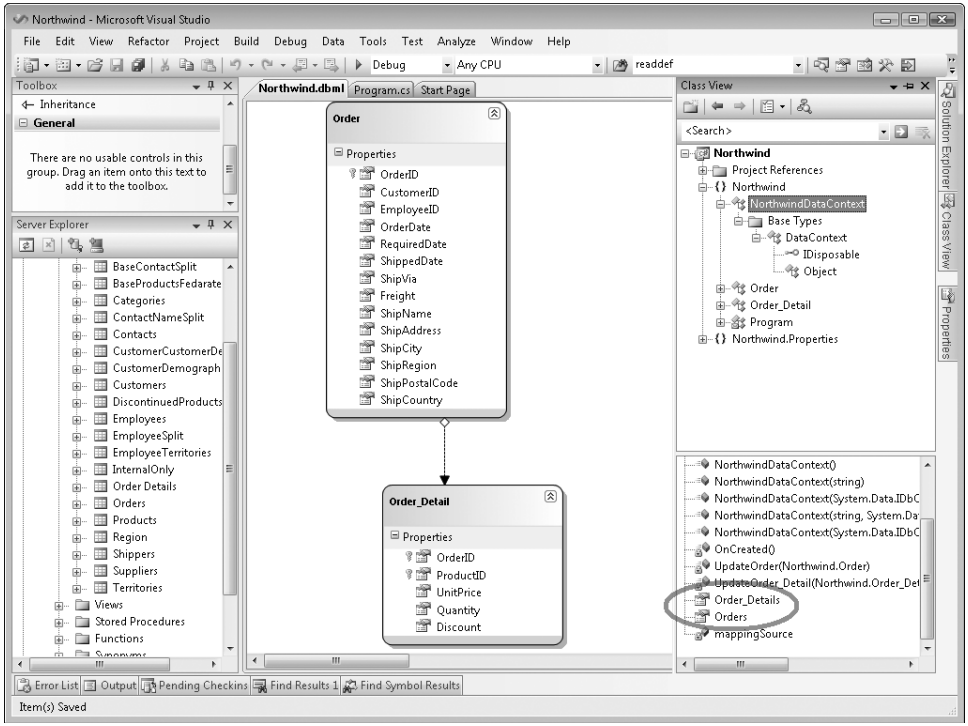


Figure 6-5 Plural names for *Table<T>* properties in a *DataContext*-derived class

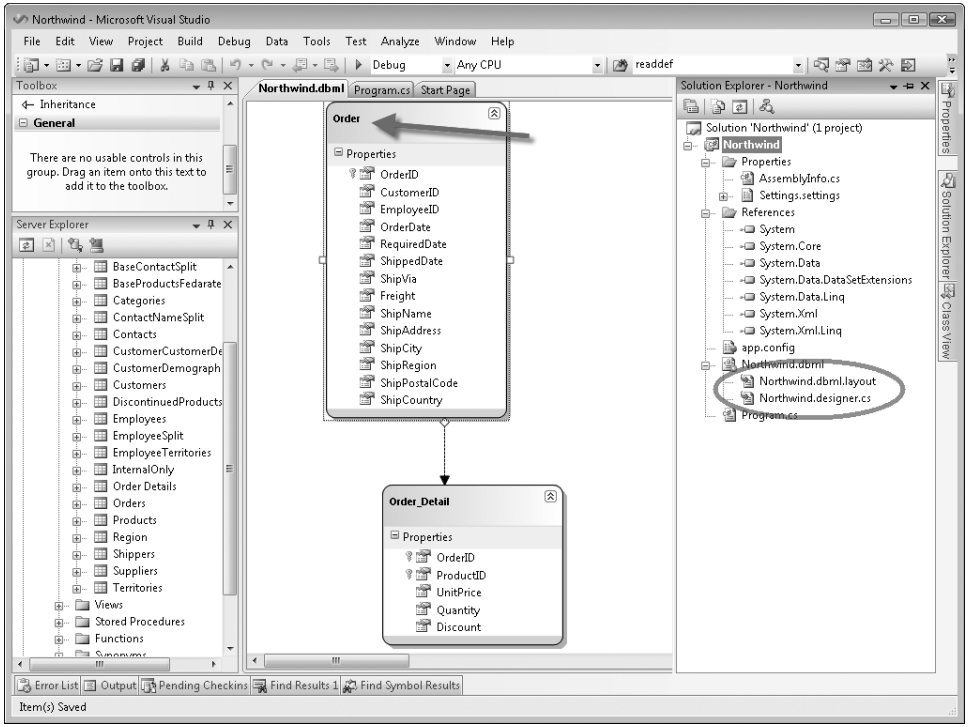


Figure 6-6 Files automatically generated for a DBML file are shown in Solution Explorer

You should not modify the source code produced by the code generator. Instead, you should edit another file containing corresponding partial classes. This file is the *Northwind.cs* file shown in Figure 6-7, which is created the first time you select the View/Code command for the currently selected item in the Object Relational Designer. In our example, we chose View, Code from the context menu on the Order entity, which is indicated by the arrow in Figure 6-6.

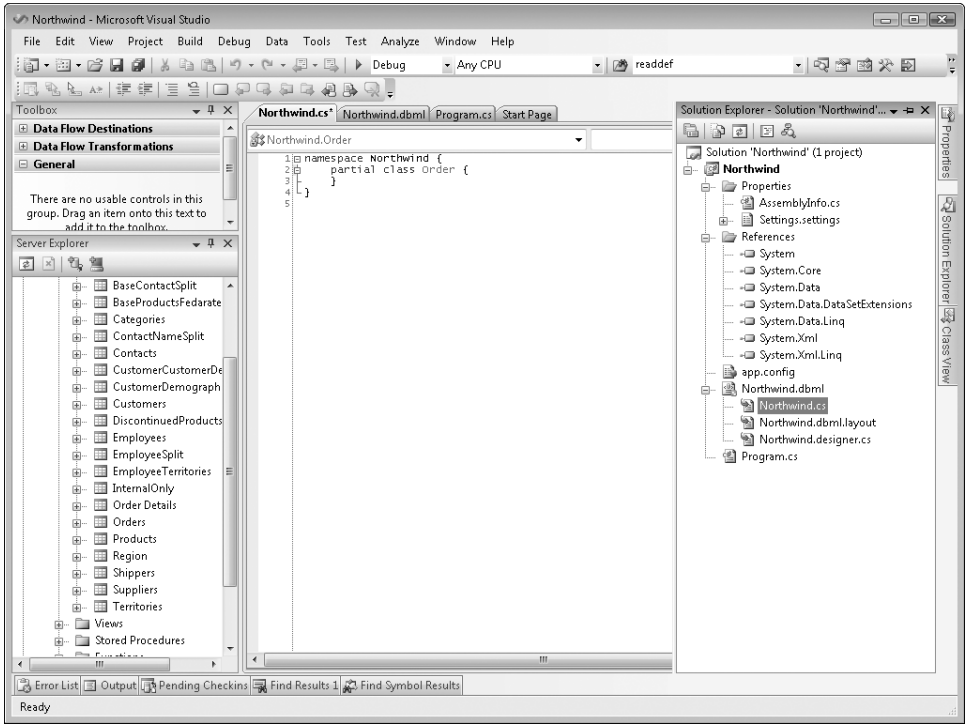


Figure 6-7 Custom code is stored in a separate file under the DBML file in Solution Explorer

At this point, most of the work will be done in the Properties window for each DBML item and in the source code. In the remaining part of this chapter, you will see the most important activities that can be performed with the DBML editor. We do not cover how an entity can be extended at the source-code level because this topic has been covered in previous chapters.

DataContext Properties

Each DBML file defines a class that inherits *DataContext*. This class will have a *Table<T>* member for each entity defined in the DBML file. The class itself will be generated following requirements specified in the Properties window. In Figure 6-8, you can see the Properties window for our *NorthwindDataContext* class.

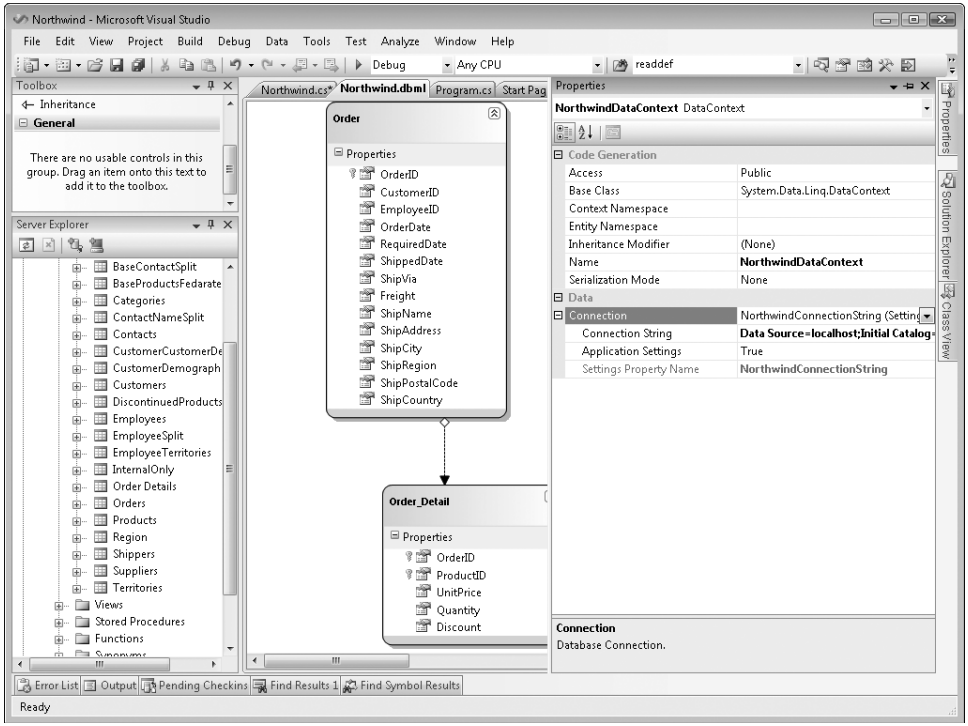


Figure 6-8 *DataContext* properties

The properties for *DataContext* are separated into two groups. The simpler one is Data, which contains the default *Connection* for *DataContext*: if you do not specify a connection when you create a *NorthwindDataContext* instance in your code, this will be the connection used. With Application Settings, you can specify whether the Application Settings file should be used to set connection information. In that case, *Settings Property Name* will be the property to use in the Application Settings file.

The group of properties named Code Generation requires a more detailed explanation, which is provided in Table 6-1.

Table 6-1 Code-Generation Properties for *DataContext*

Property	Description
Access	Access modifier for the *DataContext*-derived class. It can be only *Public* or *Internal*. By default, it is *Public*.
Base Class	Base class for the data context specialized class. By default, it is *System.Data.Linq.DataContext*. You can define your own base class, which would probably be inherited by *DataContext*.
Context Namespace	Namespace of the generated *DataContext*-derived class only. It does not apply to the entity classes. Use the same value in *Context Namespace* and *Entity Namespace* if you want to generate *DataContext* and entity classes in the same namespace.

Table 6-1 Code-Generation Properties for *DataContext*

Property	Description
Entity Namespace	Namespace of the generated entities only. It does not apply to the *DataContext*-derived class. Use the same value in *Context Namespace* and *Entity Namespace* if you want to generate *DataContext* and entity classes in the same namespace.
Inheritance Modifier	Inheritance modifier to be used in the class declaration. It can be *(None)*, *abstract*, or *sealed*. By default, it is *(None)*.
Name	Name of the *DataContext*-derived class. By default, it is the name of the database with the suffix "DataContext". For example, *Northwind-DataContext* is the default name for a *DataContext*-derived class generated for the Northwind database.
Serialization Mode	If this property is set to *Unidirectional*, the entity's source code is decorated with *DataContract* and *DataMember* for serialization purposes. By default, it is set to *None*.

Entity Class

When you select an entity class on the designer, you can change its properties in the Properties window. In Figure 6-9, you can see the Properties window for the selected *Order* entity class.

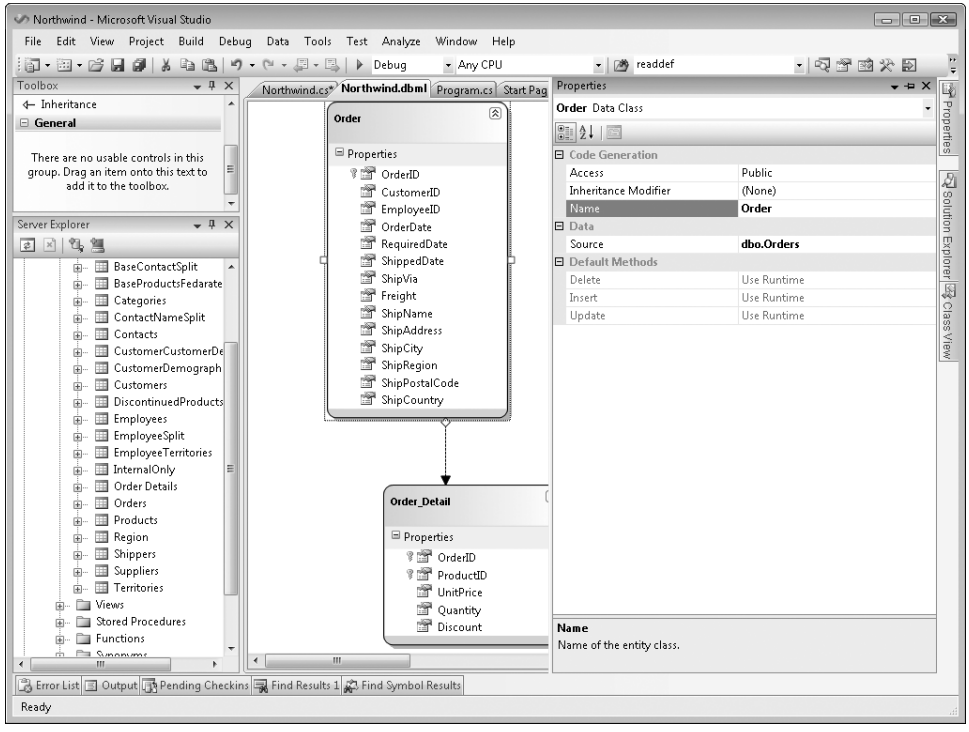


Figure 6-9 Entity class properties

The properties for an entity class are separated into three groups. The Data group contains only *Source*, which is the name of the table in the SQL Server database, including the owner or schema name. This property is automatically filled when the entity is generated by dragging a table onto the designer surface.

The Default Methods group contains three read-only properties—named *Delete*, *Insert*, and *Update*—which indicate the presence of custom Create, Update, Delete (CUD) methods. These properties are disabled if no stored procedures have been defined in the same DBML file. If you have stored procedures to be called for insert, update, and delete operations on an entity, you first have to import them into the DBML file (as described in the "Stored Procedures and User-Defined Functions" section later in this chapter). Then you can edit these properties by associating the corresponding procedure for each of the CUD operations.

Finally, the properties in the group Code Generation are explained in Table 6-2.

Table 6-2 Code-Generation Properties for an Entity Class

Property	Description
Access	Access modifier for the entity class. It can be only *Public* or *Internal*. By default, it is *Public*.
Inheritance Modifier	Inheritance modifier to be used in the class declaration. It can be *(None)*, *abstract*, or *sealed*. By default, it is *(None)*.
Name	Name of the entity class. By default, it is the singular name of the table dragged from a database in the Server Explorer window. For example, *Order* is the default name for the table named Orders in the Northwind database. Remember that the entity class will be defined in the namespace defined by the Entity Namespace of the related *DataContext* class.

Entity Members

When an entity is generated by dragging a table from Server Explorer, it has a set of predefined members that are created by reading table metadata from the relational database. Each of these members has its own settings in the Properties window. You can add new members by clicking on Add/Property on the contextual menu, or simply by pressing the INS key. You can delete a member by pressing the DEL key or by clicking Delete on the contextual menu. Unfortunately, the order of the members in an entity cannot be modified through the Object Relational Designer and can be changed only by manually modifying the DBML file and moving the physical order of the Column tags within a *Type*.

Warning You can open and modify the DBML file with a text editor such as Notepad. If you try to open the DBML file with Visual Studio 2008, remember to use the Open With option from the drop-down list for the Open button in the Open File dialog box, picking the XML Editor choice to use the XML editor integrated in Visual Studio 2008; otherwise, the Object Relational Designer will be used by default. You can also use the Open With command on a DBML file shown in the Solution Explorer in Visual Studio 2008.

When you select an entity member on the designer, you can change its properties in the Properties window. In Figure 6-10, you can see the Properties window for the selected *OrderID* member of the *Order* entity class.

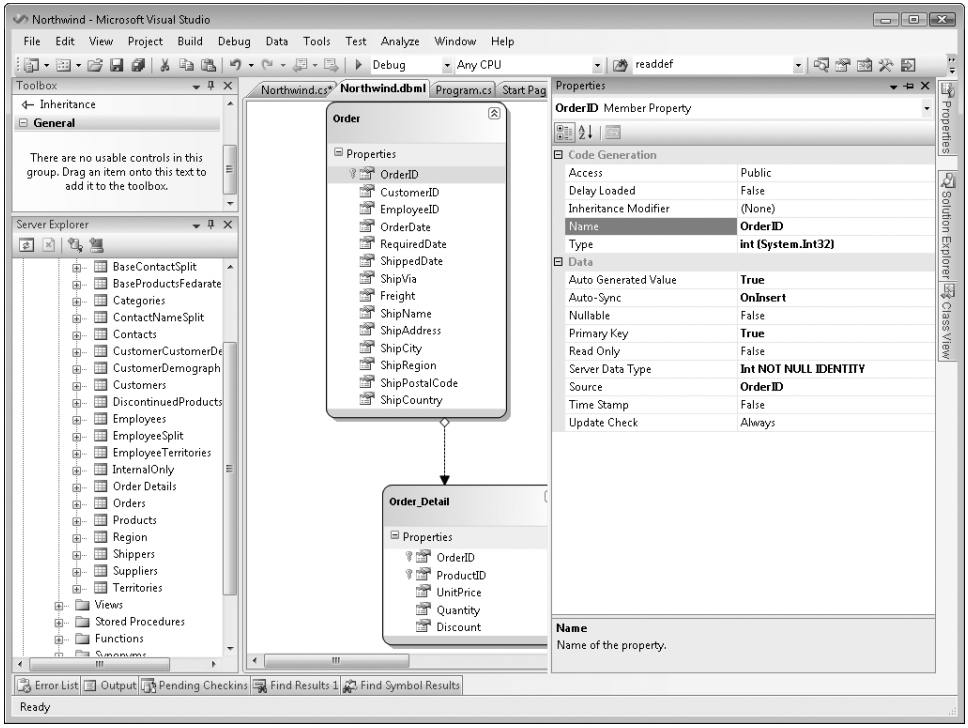


Figure 6-10 Entity member properties

The properties for an entity member are separated into two groups. The Code Generation group controls the way member attributes are generated, and its properties are described in Table 6-3.

Table 6-3 Code-Generation Properties for Data Members of an Entity

Property	Description
Access	Access modifier for the entity class. It can be *Public*, *Protected*, *Protected Internal*, *Internal*, or *Private*. By default, it is *Public*.
Delay Loaded	If this property is set to *true*, the data member will not be loaded until its first access. This is implemented by declaring the member with the *Link<T>* class, which is explained in the "Deferred Loading of Properties" section in Chapter 4. By default, it is set to *false*.
Inheritance Modifier	Inheritance modifier to be used in the member declaration. It can be *(None)*, *new*, *virtual*, *override*, or *virtual*. By default, it is *(None)*.
Name	Name of the member. By default, it is the same column name used in the *Source* property.
Type	Type of the data member. This type can be modified into a *Nullable<T>* according to the *Nullable* setting in the Data group or properties.

The Data group contains important mapping information between the entity data member and the table column in the database. The properties in this group are described in Table 6-4. Many of these properties correspond to settings of the *Column* attribute, which are described in Chapter 4 and Chapter 5.

Table 6-4 Data Properties for Data Members of an Entity

Property	Description
Auto Generated Value	Corresponds to the *IsDbGenerated* setting of the *Column* attribute.
Auto-Sync	Corresponds to the *AutoSync* setting of the *Column* attribute.
Nullable	If this property is set to *true*, the type of the data member is declared as *Nullable<T>*, where *T* is the type defined in the *Type* property. (See Table 6-3.)
Primary Key	Corresponds to the *IsPrimaryKey* setting of the *Column* attribute.
Read Only	If this property is set to *true*, only the *get* accessor is defined for the property that publicly exposes this member of the entity class. By default, it is set to *false*. Considering its behavior, this property could be part of the Code Generation group.
Server Data Type	Corresponds to the *DbType* setting of the *Column* attribute.
Source	It is the name of the column in the database table. Corresponds to the *Name* setting of the *Column* attribute.
Time Stamp	Corresponds to the *IsVersion* setting of the *Column* attribute.
Update Check	Corresponds to the *UpdateCheck* setting of the *Column* attribute.

Association Between Entities

An association represents a relationship between entities, which can be expressed through *EntitySet<T>*, *EntityRef<T>*, and the *Association* attribute we describe in Chapter 4. In Figure 6-4, you can see the association between the *Order* and *Order_Detail* entities expressed as an arrow that links these entities. In the Object Relational Designer, you can define associations between entities in two ways:

- When one or more entities are imported from a database, the existing foreign key constraints between tables, which are also entities of the designed model, are transformed into corresponding associations between entities.
- Selecting the Association item in the Toolbox window, you can link two entities defining an association that might or might not have a corresponding foreign key in the relational database. To build the association, you must have two data members of the same type in the related entities that define the relationship. On the parent side of the relationship, the member must also have the *Primary Key* property set to *True*.

Note An existing database might not have the foreign key relationship that corresponds to an association defined between LINQ to SQL entities. However, if you generate the relational database using the *DataContext.CreateDatabase* method of your model, the foreign keys are automatically generated for existing associations.

When you create an association or double-click an existing one, the dialog box shown in Figure 6-11 is displayed. The two combo boxes, Parent Class and Child Class, are disabled when editing an existing association; they are enabled only when you create a new association by using the context menu and right-clicking on an empty area of the design surface. Under Association Properties, you must select the members composing the primary key under the Parent Class, and then you have to choose the appropriate corresponding members in the Child Class.

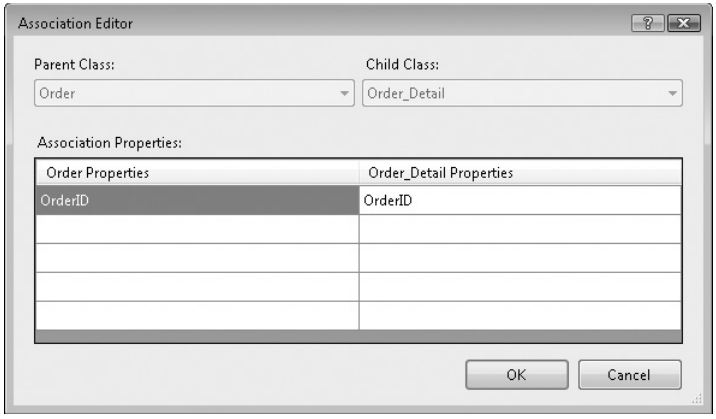


Figure 6-11 Association properties

After you have created an association, you can edit it in more detail by selecting the arrow in the graphical model and then editing it in the Properties window, as shown in Figure 6-12.

By default, the *Association* is defined in a bidirectional way. The child class gets a property with the same name as the parent class (*Order_Detail.Order* in our example), just to get a typed reference to the parent itself. In the parent class, a particular property represents the set of child elements (*Order.Order_Details* in our example). Table 6-5 provides an explanation of all the properties available in an association. As you will see, most of these settings can significantly change the output produced.

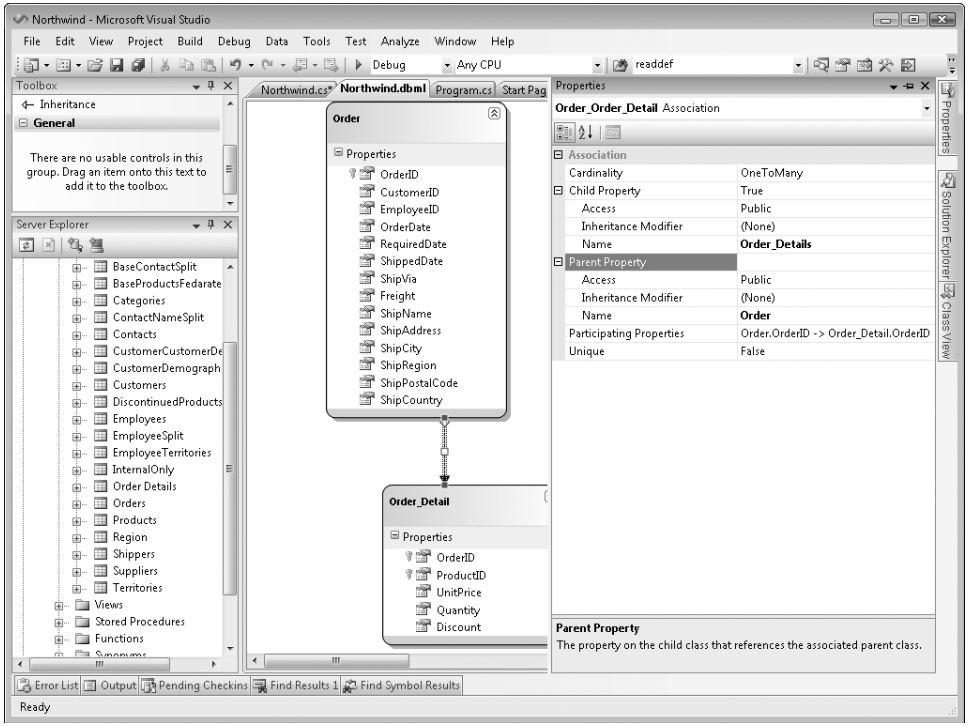


Figure 6-12 Association properties

Table 6-5 Association Properties

Property	Description
Cardinality	Defines the cardinality of the association between parent and child nodes. This property has an impact only on the member defined in the parent class. Usually and by default, it is set to *OneToMany*, which will generate a member in the parent class that will enumerate a sequence of child items. The only other possible value is *OneToOne*, which will generate a single property of the same type as the referenced child entity. See the sidebar "Understanding the Cardinality Property" for more information. By default, this property is set to *OneToMany*. Using the *OneToOne* setting is recommended, for example, when you split a logical entity that has many data members into more than one database table.
Child Property	If this property is set to *False*, the parent class will not contain a property with a collection or a reference of the child nodes. By default, it is set to *True*.
Child Property/Access	Access modifier for the member children in the parent class. It can be *Public* or *Internal*. By default, it is *Public*.

Table 6-5 Association Properties

Property	Description
Child Property/Inheritance Modifier	Inheritance modifier to be used in the member children in the parent class. It can be *(None)*, *new*, *virtual*, *override*, or *virtual*. By default, it is *(None)*.
Child Property/Name	Name of the member children in the parent class. By default, it has the plural name of the child entity class. If you set *Cardinality* to *OneToOne*, you would probably change this name to the singular form.
Parent Property/Access	Access modifier for the parent member in the child class. It can be *Public* or *Internal*. By default, it is *Public*.
Parent Property/Inheritance Modifier	Inheritance modifier to be used in the parent member in the child class. It can be *(None)*, *new*, *virtual*, *override*, or *virtual*. By default, it is *(None)*.
Parent Property/Name	Name of the parent member in the child class. By default, it has the same singular name as the parent entity class.
Participating Properties	Displays the list of related properties that make the association work. Editing this property opens the Association Editor, which is shown in Figure 6-11.
Unique	Corresponds to the *IsUnique* setting of the *Association* attribute. It should be *True* when *Cardinality* is set to *OneToOne*. However, you are in charge of keeping these properties synchronized. *Cardinality* controls only the code generated for the *Child Property*, while *Unique* controls only the *Association* attribute, which is the only one used by the LINQ to SQL engine to compose SQL queries. By default, it is set to *False*.

If you have a parent-child relationship in the same table, the Object Relational Designer automatically detects it from the foreign key constraint in the relational table whenever you drag it into the model. It is recommended that you change the automatically generated name for *Child Property* and *Parent Property*. For example, importing the Employees table from Northwind results in *Employees* for the Child Property Name and *Employee1* for the Parent Property Name. You can rename these more appropriately as *DirectReports* and *Manager*, respectively.

Warning The Child Property and Parent Property of a parent-child *Association* referencing the same table cannot be used in a *DataLoadOptions.LoadWith<T>* call because it does not support cycles.

One-to-One Relationships

Most of the time, you create a one-to-many association between two entities, and the default values of the Association properties should be sufficient. However, it is easy to get lost with a one-to-one relationship. The first point to make is about when to use a one-to-one relationship.

A one-to-one relationship should be intended as a one-to-zero-or-one relationship, where the related child entity might or might not exist. For example, we can define the simple model shown in Figure 6-13. For each *Contact*, we can have a related *Customer*, containing its amount of *Credit*. In the Properties window, you can see highlighted in bold the properties of the association between *Contact* and *Customer* that have been changed from their default values.

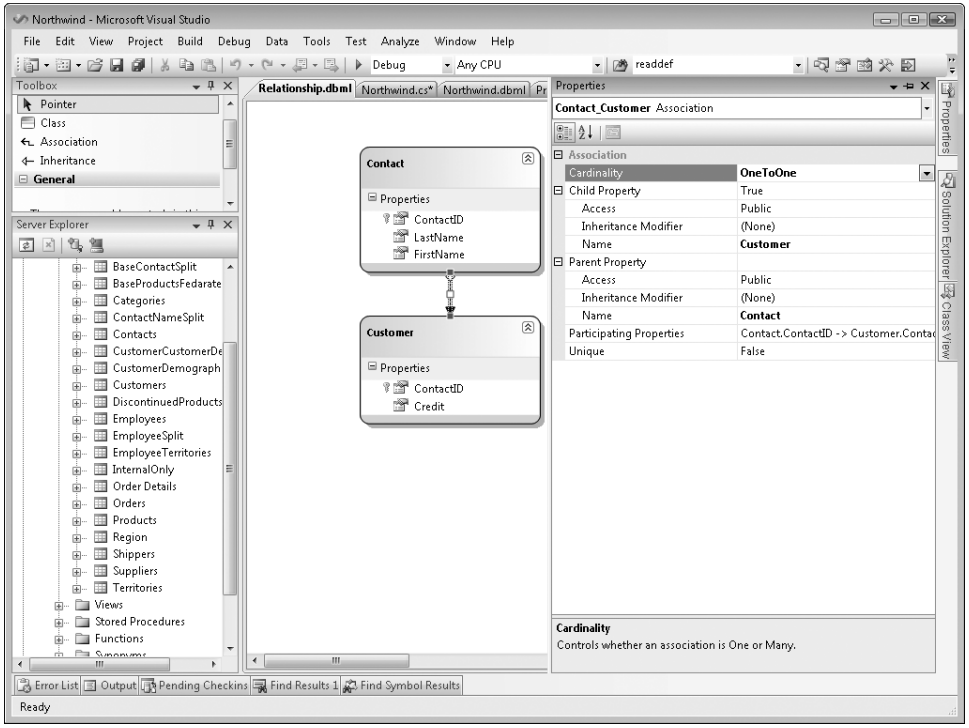


Figure 6-13 Association properties of a one-to-one relationship

Cardinality should already be set to *OneToOne* when you create the *Association*. However, it is always better to check it. You also have to set the *Unique* property to *True* and change the *Child Name* property to the singular *Customer* value.

The *ContactID* member in the *Contact* entity is a primary key defined as *INT IDENTITY* in the database. Thus, it has the *Auto Generated Value* set to *True* and *Auto-Sync* set to *OnInsert*. In the *Customer* entity, you have another member called *ContactID*, which is also a primary key but is not generated from the database. In fact, you will use the key generated for a *Contact* to assign the *Customer.ContactID* value. Thanks to the *Contact.Customer* and *Customer.Contact* properties, you can simply assign the relationship by setting one of these properties, without worrying about the underlying *ContactID* field. In the following code, you can see an example

of two *Contact* instances saved to the *DataContext*; one of them is associated with a *Customer* instance:

```
RelationshipDataContext db = new RelationshipDataContext();

Contact contactPaolo = new Contact();
contactPaolo.LastName = "Pialorsi";
contactPaolo.FirstName = "Paolo";

Contact contactMarco = new Contact();
Customer customer = new Customer();
contactMarco.LastName = "Russo";
contactMarco.FirstName = "Marco";
contactMarco.Customer = customer;
customer.Credit = 1000;

db.Contacts.InsertOnSubmit(contactPaolo);
db.Contacts.InsertOnSubmit(contactMarco);
db.SubmitChanges();
```

We created the relationship by setting the *Contact.Customer* property, but the same result could have been obtained by setting the *Customer.Contact* property. In other words, thanks to the synchronization code automatically produced by the code generator, in our one-to-one relationship the line

```
contactMarco.Customer = customer;
```

produces the same result as writing

```
customer.Contact = contactMarco;
```

However, you have to remember that the *Customer.Contact* member is mandatory if you create a *Contact* instance, while *Contact.Customer* can be left set to the default null value if no *Customer* is related to that *Contact*. At this point, it should be clear why the direction of the association is relevant even in a one-to-one relationship. As we said, it is not really a one-to-one relationship but a one-to-zero-or-one relationship, where the association stems from the parent that always exists to the child that could not exist.

Warning A common error made when defining a one-to-one association is using the wrong direction for the association. In our example, if the association went from *Customer* to *Contact*, it would not generate a compilation error; instead, our previous code would throw an exception when trying to submit changes to the database.

Understanding the *Cardinality* Property

To better understand the behavior of the *Cardinality* property, let's take a look at the generated code. This is an excerpt of the code generated with *Cardinality* set to *OneToMany*. The member is exposed with the plural name of *Customers*.

```
public partial class Contact {
    public Contact() {
        this._Customers = new EntitySet<Customer>(
                                new Action<Customer>(this.attach_Customers),
                                new Action<Customer>(this.detach_Customers));
    }

    private EntitySet<Customer> _Customers;

    [Association(Name="Contact_Customer", Storage="_Customers",
                 ThisKey="ContactID", OtherKey="ContactID")]
    public EntitySet<Customer> Customers {
        get { return this._Customers; }
        set { this._Customers.Assign(value); }
    }
}
```

And this is the code with *Cardinality* set to *OneToOne*. The member is exposed with the singular name of *Customer*. (You need to manually change the *Child Property Name* if you change the *Cardinality* property.)

```
public partial class Contact {
    public Contact() {
        this._Customer = default(EntityRef<Customer>);
    }

    private EntityRef<Customer> _Customer;

    [Association(Name="Contact_Customer", Storage="_Customer",
                 ThisKey="ContactID", IsUnique=true, IsForeignKey=false)]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set {
            Customer previousValue = this._Customer.Entity;
            if ((previousValue != value)
                || (this._Customer.HasLoadedOrAssignedValue == false)) {
                this.SendPropertyChanging();
                if ((previousValue != null)) {
                    this._Customer.Entity = null;
                    previousValue.Contact = null;
                }
                this._Customer.Entity = value;
```

```
                if ((value != null)) {
                    value.Contact = this;
                }
                this.SendPropertyChanged("Customer");
            }
        }
    }
}
```

As you can see, in the parent class we get a *Contact.Customer* member of type *EntityRef<Customer>* if *Cardinality* is set to *OneToOne*. Otherwise, we get a *Contact.Customers* member of type *EntitySet<Customer>* if *Cardinality* is set to *OneToMany*. Finally, the code generated for the *Customer* class does not depend on the *Cardinality* setting.

Entity Inheritance

LINQ to SQL supports the definition of a hierarchy of classes all bound to the same source table. The LINQ to SQL engine generates the right class in the hierarchy, based on the value of a specific row of that table. Each class is identified by a specific value in a column, following the *InheritanceMapping* attribute applied to the base class, as we saw in the section "Entity Inheritance" in Chapter 4.

Creating a hierarchy of classes in the Object Relational Designer starting from an existing database requires you to complete the following actions:

1. Create a Data class for each class of the hierarchy. You can drag the table for the base class from Server Explorer, and then create other empty classes by dragging a Class item from the toolbox. Rename the classes you add according to their intended use.
2. Set the *Source* property for each added class equal to the *Source* property of the base class you dragged from the data source.
3. After you have at least a base class and a derived class, create the Inheritance relationship. Select the Inheritance item in the toolbox, and draw a connection starting from the deriving class and ending with the base class. You can also define a multiple-level hierarchy.
4. If you have members in the base class that will be used only by some derived classes, you can cut and paste them in the designer. (Note that dragging and dropping members is not allowed.)

For example, in Figure 6-14 you can see the result of the following operations:

1. Drag the Contact table from Northwind.

2. Add the other empty Data classes (*Employee*, *CompanyContact*, *Customer*, *Shipper*, and *Supplier*).
3. Put the *dbo.Contacts* value into the *Source* property for all added Data classes. (Note that *dbo.Contacts* is already the *Source* value of the base class *Contact*.)
4. Define the *Inheritance* between *Employee* and *Contact* and between *CustomerContact* and *Contact*.
5. Define the *Inheritance* between *Customer* and *CompanyContact*, *Shipper* and *CompanyContact*, and *Supplier* and *CompanyContact*.
6. Cut the *CompanyName* member from *Contact*, and paste it into *CompanyContact*.
7. Set the Discriminator Property of any *Inheritance* item to *ContactType*. (See Table 6-6 for further information about this property.)
8. Set the Inheritance Default Property of any *Inheritance* item to *Contact*.
9. Set the Base Class Discriminator Value of any *Inheritance* item to *Contact*.
10. Set the Derived Class Discriminator Value to *Employee*, *Customer*, *Shipper*, or *Supplier* for each corresponding *Inheritance* item.

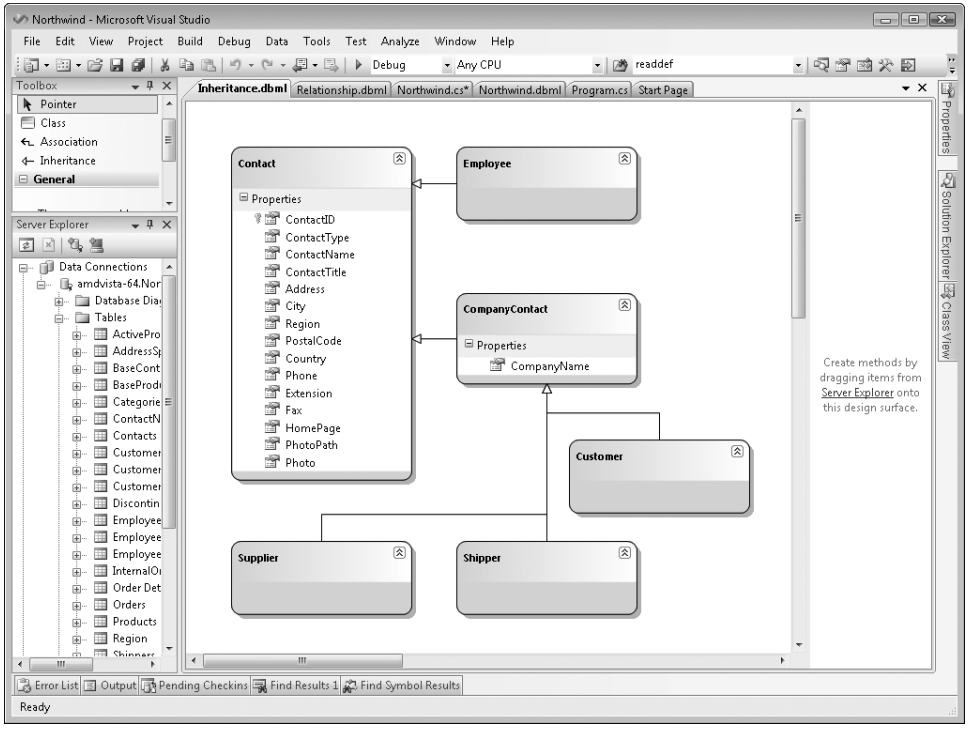


Figure 6-14 Design of a class hierarchy based on the Northwind.Contact table

Our example uses an intermediate class (*CompanyContact*) to simplify the other derived classes (*Supplier*, *Shipper*, and *Customer*). We skipped the *CompanyContact* class that sets the Derived Class Discriminator Value because that intermediate class does not have concrete data in the database table.

In Table 6-6, you can see an explanation of all the properties available for an Inheritance item. We used these properties to produce the design shown in Figure 6-14.

Table 6-6 Inheritance Properties

Property	Description
Inheritance Default	This is the type that will be used to create entities for rows that do not match any defined inheritance codes (which are the values defined for *Base Class Discriminator Value* and *Derived Class Discriminator Value*). This setting defines which of the generated *InheritanceMapping* attributes will have the *IsDefault=true* setting.
Base Class Discriminator Value	This is a value of the *Discriminator Property* that specifies the base class type. When you set this property for an *Inheritance* item, all *Inheritance* items originating from the same data class will assume the same value.
Derived Class Discriminator Value	This is a value of the *Discriminator Property* that specifies the derived class type. It corresponds to the *Code* setting of the *InheritanceMapping* attribute.
Discriminator Property	The column in the database that is used to discriminate between entities. When you set this property for an *Inheritance* item, all *Inheritance* items originating from the same data class will assume the same value. The selected data member in the base class will be decorated with the *IsDiscriminator=true* setting in the *Column* attribute.

Stored Procedures and User-Defined Functions

Dragging a stored procedure or a user-defined function from the Server Explorer window to the Object Relational Designer surface creates a method in the *DataContext* class corresponding to that stored procedure or that user-defined function. In Figure 6-15, you can see an example of the [Customer By City] stored procedure dragged onto the Methods pane of the Object Relational Designer.

Note You can show and hide the Methods pane by using the context menu that opens when you right-click on the design surface.

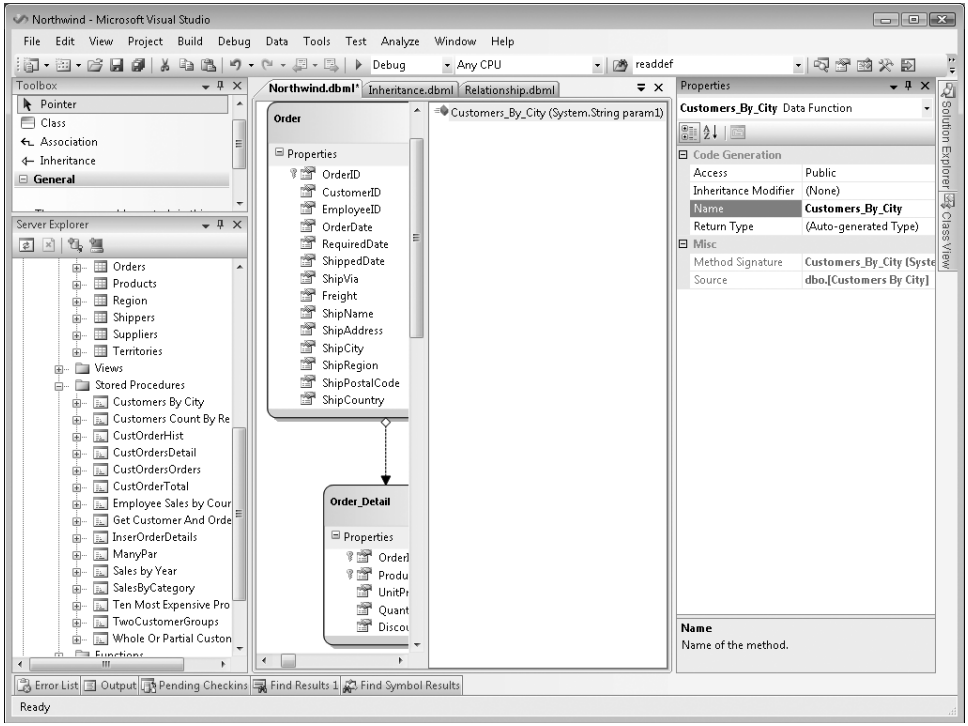


Figure 6-15 Stored procedure imported into a DBML file

When you import either a stored procedure or a user-defined function, a *Data Function* item is created in the *DataContext*-derived class. The properties of a *Data Function* are separated into two groups. The Misc group contains two read-only properties, *Method Signature* and *Source*. The *Source* property contains the name of the stored procedure or user-defined function in the database. The value of the *Method Signature* property is constructed with the *Name* property (shown in Table 6-7) and the parameters of the stored procedure or user-defined function. The group of properties named Code Generation requires a more detailed explanation, which is included in Table 6-7.

Table 6-7 Code-Generation Properties for *Data Function*

Property	Description
Access	Access modifier for the generated method in the *DataContext*-derived class. It can be *Public*, *Protected*, *Protected Internal*, *Internal*, or *Private*. By default, it is *Public*.
Inheritance Modifier	Inheritance modifier to be used in the member declaration. It can be *(None)*, *new*, *virtual*, *override*, or *virtual*. By default, it is *(None)*.

Table 6-7 Code-Generation Properties for *Data Function*

Property	Description
Name	Name of the method representing a stored procedure or a user-defined function in the database. By default, it is derived from the name of the stored procedure or the user-defined function, replacing invalid characters in C# or Visual Basic with an underscore (_). It corresponds to the *Name* setting of the *Function* attribute.
Return Type	Type returned by the method. It can be a common language runtime (CLR) type for scalar-valued user-defined functions, or *Class Data* for stored procedures and table-valued user-defined functions. In the latter case, by default it is *(Auto-generated Type)*. After it has been changed to an existing *Data Class* name, this property cannot be reverted to *(Auto-generated Type)*. See the "Return Type of Data Function" section for more information.

Return Type of Data Function

Usually a stored procedure or a table-valued user-defined function returns a number of rows, which in LINQ to SQL becomes a sequence of instances of an entity class. (We discussed this in the "Stored Procedures and User-Defined Functions" section in Chapter 4.) By default, the *Return Type* property is set to *(Auto-generated Type)*, which means that the code generator creates a class with as many members as the columns returned by SQL Server. For example, the following excerpt of code is part of the *Customers_By_CityResult* type automatically generated to handle the *Customer_By_City* result. (The *get* and *set* accessors have been removed from the properties declaration for the sake of conciseness.)

```
public partial class Customers_By_CityResult {
    private string _CustomerID;
    private string _ContactName;
    private string _CompanyName;
    private string _City;

    public Customers_By_CityResult() { }

    [Column(Storage="_CustomerID", DbType="NChar(5) NOT NULL",
            CanBeNull=false)]
    public string CustomerID { ... }

    [Column(Storage="_ContactName", DbType="NVarChar(30)")]
    public string ContactName { ... }

    [Column(Storage="_CompanyName", DbType="NVarChar(40) NOT NULL",
            CanBeNull=false)]
    public string CompanyName { ... }

    [Column(Storage="_City", DbType="NVarChar(15)")]
    public string City { ... }
}
```

However, you can instruct the code generator to use an existing *Data Class* to store the data resulting from a stored procedure call, setting the *Return Type* property to the desired type. The combo box in the Properties window presents all types defined in the *DataContext*. You should select a type compatible with the data returned by SQL Server.

Important *Return Type* must have at least a public member with the same name of a returned column. If you specify a type with public members that do not correspond to returned columns, these "missing" members will have a default value.

You can create an entity class specifically to handle the result coming from a stored procedure or user-defined function call. In that case, you might want to define a class without specifying a *Source* property. In this way, you can control all the details of the returned type. You can also use a class corresponding to a database table. In this case, remember that you can modify the returned entity. However, to make the *SubmitChanges* work, you need to get the initial value for all required data members of the entity (at least those with the *UpdateCheck* constraint) in order to match the row at the moment of update. In other words, if the stored procedure or user-defined function does not return all the members for an entity, it is better to create an entity dedicated to this purpose, using only the returned columns and specifying the destination table as the *Source* property.

Note To map *Return Type* to an entity during the method construction, you can drag the stored procedure or user-defined function, dropping it on the entity class that you want to use as a return type. In this way, the method is created only if the entity class has a corresponding column in the result for each of the entity members. If this condition is not satisfied, an error message is displayed and the operation is cancelled.

Mapping to Delete, Insert, and Update Operations

All imported stored procedures can be used to customize the Delete, Insert, and Update operations of the entity class. To do that, after you import the stored procedures into *DataContext*, you need to bind them to the corresponding operation in the entity class. Figure 6-16 shows the Configure Behavior dialog box that allows mapping of all the method arguments with the corresponding class properties.

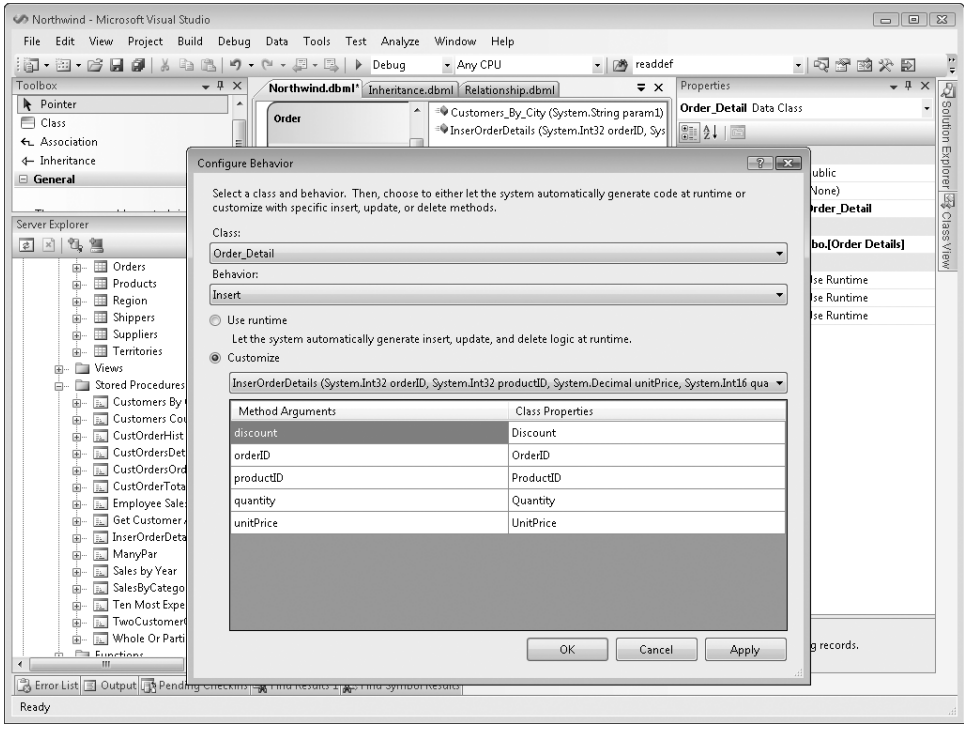


Figure 6-16 Use of a stored procedure to insert an *Order_Detail*

More Info For more information, see the "Customizing Insert, Update, and Delete" section in Chapter 5.

Views and Schema Support

All views in a database can be used to generate an entity class in the DBML file. However, LINQ to SQL does not know whether the view is updatable or not. It is your responsibility to make the right use of an entity derived from a view, trying to update instances of that entity only if they come from an updatable view.

If the database has tables in different schemas, the Object Relational Designer does not consider them when creating the name of data classes or data functions. The schema is maintained as part of the *Source* value, but it does not participate in the name construction of generated objects. You can rename the objects, but they cannot be defined in different namespaces, because all the entity classes are defined in the same namespace, which is controlled by the *Entity Namespace* property of the generated *DataContext*-derived class.

More Info Other third-party code generators might support the use of namespaces, using SQL Server 2005 schemas to create entities in corresponding namespaces.

Summary

In this chapter, we took a look at the tools that are available to generate LINQ to SQL entities and *DataContext* classes. The .NET Framework SDK includes the command-line tool named SQLMetal. Visual Studio 2008 has a graphical editor known as the Object Relational Designer. Both allow the creation of a DBML file, the generation of source code in C# and Visual Basic, and the creation of an external XML mapping file. The Object Relational Designer also allows you to edit an existing DBML file, dynamically importing existing tables, views, stored procedures, and user-defined functions from an existing SQL Server database.

Chapter 16

LINQ and ASP.NET

ASP.NET 3.5 is one of the areas of the .NET Framework in which LINQ has been deeply embedded. In this chapter, we will focus on the new features available in ASP.NET 3.5 targeting LINQ—in particular, for LINQ to SQL and LINQ to Objects.

ASP.NET 3.5

In this section, we will cover some of the new features and capabilities available in ASP.NET 3.5 and Microsoft Visual Studio 2008. These features are not all related to LINQ itself, but we will use them during the remaining parts of this chapter. If you are already familiar with ASP.NET 3.5, you can skip or read quickly through this section.

ASP.NET 3.5 extends the ASP.NET 2.0 control and class framework by providing new controls such as *ListView* and *DataPager* and a set of new AJAX-enabling controls. Other improvements involve the IDE of Visual Studio 2008, with new HTML and cascading style sheets (CSS) designers, improved Microsoft IntelliSense, and the capability to design nested master pages.

ListView

ListView is a new control that takes the place of the *DataGrid*, *GridView*, *DataList*, and *Repeater* controls. It allows full control over the data template used to render data binding contents, including the container. This new control allows you to define data binding editing, insertion, deletion, selection, paging, and sorting with full control over the HTML generated. It should be used in place of many of the other controls available prior to ASP.NET 3.5. In fact, using *ListView* you can bind items as you do with *DataGrid*, *GridView*, *DataList*, or *Repeater*, or you can define a freer layout by declaring a set of user-defined HTML/ASPX templates.

The *ListView* control allows you to define a rich set of templates. *LayoutTemplate* and *ItemTemplate* are mandatory, while all the others are optional. Let's start with an ASPX page used to render a set of Northwind customers, shown in Listing 16-1.

Listing 16-1 A sample ASPX page using a *ListView* control instance.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-1.aspx.cs"
    Inherits="Listing16_1" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-1</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        DataSourceID="customersDataSource">
        <LayoutTemplate>
            <ul>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </ul>
        </LayoutTemplate>
        <ItemTemplate>
            <li>
                <b><asp:Label runat="server" Text='<%# Eval("CustomerId") %>' /></b> -
                <asp:Label runat="server" Text='<%# Eval("ContactName") %>' /><br />
                <asp:Label runat="server" Text='<%# Eval("CompanyName") %>' />
                [<asp:Label runat="server" Text='<%# Eval("Country") %>' />]
            </li>
        </ItemTemplate>
    </asp:ListView>
    <asp:SqlDataSource ID="customersDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
    SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName], [Country]
                   FROM [Customers]" />
        </div>
    </form>
</body>
</html>
```

As you can see, we have defined an *<asp:ListView />* element, bound to a *SqlDataSource* control, with a couple of templates in it. The first one, named *LayoutTemplate*, declares how to render the full layout of the container of data items. It could be a TABLE, DIV, SPAN, UL, OL, or any other element you want to use to wrap your set of data. Any data-bound content you are going to render through the *ListView* is rendered as defined in the *ItemTemplate* element. As you can see, the *LayoutTemplate* contains an *<asp:PlaceHolder />* control with an ID *"itemPlaceholder"*. This control will be used as the placeholder for each data-bound item to render. The value of *"itemPlaceholder"* for the ID of the placeholder can be configured, but the key point here is that *ItemTemplate* defines how to render each data-bound item, placing the result of its rendering inside the *LayoutTemplate* result, where a placeholder control has been defined.

The following is the full list of the templates available while defining a *ListView* control:

- *ItemSeparatorTemplate*: As its name implies, this template defines the content to render between data-bound items.
- *AlternatingItemTemplate*: Defines alternate content to distinguish between consecutive items.
- *SelectedItemTemplate*: Defines how to render a selected item to differentiate it from the others.

- *EditItemTemplate*: Defines the content to render in place of the *ItemTemplate*, for an item that has editing status.
- *InsertItemTemplate*: Defines the content to render to insert a new item. By configuring the *InsertItemPosition* property of the *ListView* control, you can decide to render the *InsertItemTemplate* at the top or at the bottom of the rendered list.
- *GroupTemplate*: Defines the content to render to wrap a set of *ItemTemplate* or *EmptyItemTemplate* items. This template is useful for defining output such as multicolumn lists, where output is grouped in rows made of columns. The *GroupTemplate* defines how to wrap a set of data-bound items, where the number of items for each group is defined by the property *GroupItemCount*.
- *GroupSeparatorTemplate*: Defines the content to render between each group of data-bound items.
- *EmptyItemTemplate*: Rendered whenever there is an empty item to render in a group. Imagine a situation in which you have to render a set of items grouped in rows of three columns each, but your data source is not arranged in multiples of three. You will end your data-bound list with one or two empty columns. This template declares how to render those empty columns.
- *EmptyDataTemplate*: Rendered when the data source is empty.

All the template elements in the preceding list use the common and well-known ASP.NET data-binding expressions to define which fields or properties to display.

In Listing 16-2, you can see an example of a *ListView* control rendering a list of items in a grid with multiple rows and columns.

Listing 16-2 A sample ASPX page using a *ListView* control instance to render a list with multiple rows and columns.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-2.aspx.cs"
    Inherits="Listing16_2" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        ItemPlaceholderID="dataItemPlaceholder"
        GroupPlaceholderID="groupPlaceholder"
        DataSourceID="customersDataSource" GroupItemCount="5">
        <LayoutTemplate>
            <table cellpadding="2" cellspacing="3">
                <asp:PlaceHolder ID="groupPlaceholder" runat="server" />
            </table>
```

```
        </LayoutTemplate>
        <GroupTemplate>
            <tr>
                <asp:PlaceHolder ID="dataItemPlaceholder" runat="server" />
            </tr>
        </GroupTemplate>
        <ItemTemplate>
            <td align="center" style="background-color: LightGreen;">
                <b><asp:Label ID="Label1" runat="server"
                              Text='<%# Eval("CustomerId") %>' /></b> -
                <asp:Label ID="Label2" runat="server"
                              Text='<%# Eval("ContactName") %>' /><br />
                <asp:Label ID="Label3" runat="server"
                              Text='<%# Eval("CompanyName") %>' />
                [<asp:Label ID="Label4" runat="server"
                             Text='<%# Eval("Country") %>' />]
            </td>
        </ItemTemplate>
        <EmptyItemTemplate>
            <td align="center" style="background-color: LightGreen;"> </td>
        </EmptyItemTemplate>
    </asp:ListView>
    <asp:SqlDataSource ID="customersDataSource" runat="server"
    ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
    SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName],  [Country]
                   FROM [Customers]" />
    </div>
    </form>
</body>
</html>
```

In Listing 16-2, you can see how to declare a custom value for the ID of the element to use as the data-bound item placeholder, using the *ItemPlaceholderID* attribute. One more thing to notice in Listing 16-2 is the attribute *GroupingPlaceholderID*, which is used to declare the ID of the placeholder describing each group of data items.

ListView Data Binding

The *ListView* control can be bound to any data source control available in ASP.NET, including the *LinqDataSource* control, which we will cover in detail later in this chapter, or any data source that implements the *IEnumerable* interface.

To bind *ListView* to a data source control, you simply need to assign the ID of the data source control to the *DataSourceID* property of the *ListView* instance. You can see this in Listing 16-1 and Listing 16-2, where we bind a *ListView* control to a *SqlDataSource* control instance.

To bind *ListView* to a data source that implements the *IEnumerable* interface, you need to programmatically set the *DataSource* property, referencing the enumeration of items and invoking the *DataBind* method of the control instance.

In Listing 16-3, you can see a sample ASPX page using this technique, while in Listing 16-4 you can see the corresponding C# code, which sets the *DataSource* property and invokes the *DataBind* method.

Listing 16-3 A sample ASPX page excerpt using a *ListView* control bound to a data source by user code.

```
<asp:ListView ID="customersList" runat="server">
    <LayoutTemplate>
        <ul>
            <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
        </ul>
    </LayoutTemplate>
    <ItemTemplate>
        <li>
            <b><asp:Label ID="Label1" runat="server"
                          Text='<%# Eval("CustomerId") %>' /></b> -
            <asp:Label ID="Label2" runat="server"
                          Text='<%# Eval("ContactName") %>' /><br />
            <asp:Label ID="Label3" runat="server"
                          Text='<%# Eval("CompanyName") %>' />
            [<asp:Label ID="Label4" runat="server"
                         Text='<%# Eval("Country") %>' />]
        </li>
    </ItemTemplate>
</asp:ListView>
```

Listing 16-4 The code-behind class of the ASPX page shown in Listing 16-3.

```
public partial class Listing16_3 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        if (!this.IsPostBack) {
            NorthwindDataContext dc = new NorthwindDataContext();
            customersList.DataSource = dc.Customers;
            customersList.DataBind();
        }
    }
}
```

One interesting feature of the *ListView* control is its capability, while it is bound to a data source, to manage a set of data keys to identify each data-bound item. This is useful for data sources where each item has multiple values for its primary key or identifier. To leverage this feature, you can set the *DataKeyNames* property of the *ListView* instance with an array of *String* variables, representing the fields that are part of the item identifier (the record primary key fields in the case of a data base record set). Later in your code, you can get the identifier of each data-bound item from the *ListView* control instance, retrieving the value from the read-only indexer property *DataKeys* of the *ListView* or retrieving the value of the *SelectedDataKey* property. In Listing 16-5, you can see the code-behind class of a sample page that uses this feature to extract the identifier of the item currently selected by the user by leveraging

the *ListView* control's selection events. The data source of this sample is the Northwind's Order Details table, where each order detail is identified by the *OrderID* and *ProductID* fields.

Listing 16-5 The code-behind class of the ASPX page using the *DataKeyNames* and *DataKeys* properties.

```
public partial class Listing16_6 : System.Web.UI.Page {
    protected void orderDetailsList_SelectedIndexChanged(Object sender, EventArgs e) {
        Int32? selectedItemOrderID =
                  ((Int32?)orderDetailsList.SelectedDataKey.Values[0]) ?? null;
        Int32? selectedItemProductID =
                  ((Int32?)orderDetailsList.SelectedDataKey.Values[1]) ?? null;

        selectedItemId.Text = String.Format("Selected item ID: {0} - {1}",
            selectedItemOrderID,
            selectedItemProductID);
    }
}
```

In Listing 16-6, you can see the ASPX page corresponding to the code shown in Listing 16-5.

Listing 16-6 The ASPX code of the page using the *DataKeyNames* property.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-6.aspx.cs"
    Inherits="Listing16_6" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="orderDetailsList" runat="server"
        DataKeyNames="OrderID, ProductID"
        DataSourceID="orderDetailsDataSource"
        OnSelectedIndexChanged="orderDetailsList_SelectedIndexChanged">
        <LayoutTemplate>
            <table border="1" cellpadding="2" cellspacing="3">
                <tr>
                    <th style="text-align: center;">Action Command</th>
                    <th style="text-align: center;">OrderID + ProductID</th>
                    <th style="text-align: center;">UnitPrice</th>
                    <th style="text-align: center;">Quantity</th>
                    <th style="text-align: center;">Total</th>
                </tr>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </table>
        </LayoutTemplate>
        <ItemTemplate>
            <tr>
```

```
                <td style="text-align: center;">
                    <asp:Button CommandName="Select"
                    Text="Select" runat="server" />
                </td>
                <td style="text-align: center;">
                    <b><asp:Label runat="server"
                    Text='<%# Eval("OrderID") %>' /> -
                    <asp:Label runat="server"
                    Text='<%# Eval("ProductID") %>' /></b><br />
                </td>
                <td style="text-align: center;">
                    <asp:Label runat="server"
                    Text='<%# Eval("UnitPrice", "{0:c}") %>' />
                </td>
                <td style="text-align: center;">
                    <asp:Label runat="server"
                    Text='<%# Eval("Quantity") %>' />
                </td>
                <td style="text-align: right;">
                    <asp:Label runat="server"
                    Text='<%# Eval("Total", "{0:c}") %>' />
                </td>
            </tr>
        </ItemTemplate>
        <SelectedItemTemplate>
            <tr style="background-color: LightGreen;">
                <td style="text-align: center;">
                    <asp:Button ID="Button1" CommandName="Select"
                        Text="Select" runat="server" />
                </td>
                <td style="text-align: center;">
                    <b><asp:Label ID="Label1" runat="server"
                    Text='<%# Eval("OrderID") %>' /> -
                    <asp:Label ID="Label2" runat="server"
                    Text='<%# Eval("ProductID") %>' /></b><br />
                </td>
                <td style="text-align: center;">
                    <asp:Label ID="Label3" runat="server"
                    Text='<%# Eval("UnitPrice", "{0:c}") %>' />
                </td>
                <td style="text-align: center;">
                    <asp:Label ID="Label4" runat="server"
                    Text='<%# Eval("Quantity") %>' />
                </td>
                <td style="text-align: right;">
                    <asp:Label ID="Label5" runat="server"
                    Text='<%# Eval("Total", "{0:c}") %>' />
                </td>
            </tr>
        </SelectedItemTemplate>
    </asp:ListView>

    <asp:SqlDataSource ID="orderDetailsDataSource" runat="server"
        ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
        SelectCommand="SELECT TOP 20 [OrderID], [ProductID], [UnitPrice],
```

```
        [Quantity], [UnitPrice] * [Quantity] AS [Total]
        FROM [Order Details]" />
    </div>
    <div>
       <asp:Label ID="selectedItemId" runat="server" />
    </div>
    </form>
</body>
</html>
```

DataPager

One more new control available in ASP.NET 3.5 is *DataPager*, which is useful for automating paging tasks in data-bound controls. This new control can be applied only to controls implementing the *IPageableItemContainer* interface, like the new *ListView* control we just described. This interface provides a couple of properties—one to set the current page size (*MaximumRows*) and one to set the starting record index (*StartRowIndex*)—one method (*SetPageProperties*) to set those properties, and an event (*TotalRowCountAvailable*) to notify you of the availability of that information as well as the total number of rows. In Listing 16-7, you can see the definition of the *IPageableItemContainer* interface.

Listing 16-7 The *IPageableItemContainer* interface definition.

```
public interface IPageableItemContainer {
    // Events
    event EventHandler<PageEventArgs> TotalRowCountAvailable;

    // Methods
    void SetPageProperties(int startRowIndex, int maximumRows, bool databind);

    // Properties
    int MaximumRows { get; }
    int StartRowIndex { get; }
}
```

The *DataPager* control uses this interface to talk with its paged control.

Note The *DataPager* control by itself does not paginate the data source; instead, it informs the bound control to paginate it. In the case of smart data sources, such as the *LinqDataSource*, which we will cover later in this chapter, the result is efficient and the query extracts exactly the required fields. However, with controls not designed or configured for paging, the paging task occurs in memory—thus, eventually creating performance issues on large sets of data.

To map a *DataPager* instance to a data-binding control, you can set the *PagedControlID* property of the pager control or, in the case of a *ListView* control, you can define the *DataPager*

element inside the *LayoutTemplate* of the paged control. In Listing 16-8, you can see an ASPX page using a *DataPager* instance to paginate a list of records bound to a *ListView* control.

Listing 16-8 A sample ASPX page using a *ListView* control paged by a *DataPager* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-8.aspx.cs"
    Inherits="Listing16_8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
    <asp:ListView ID="customersList" runat="server"
        DataSourceID="customersDataSource">
        <LayoutTemplate>
            <table cellpadding="2" cellspacing="3">
                <tr>
                    <th style="text-align: center">CustomerId</th>
                    <th style="text-align: center">CompanyName</th>
                    <th style="text-align: center">ContactName</th>
                    <th style="text-align: center">Country</th>
                </tr>
                <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
            </table>
        </LayoutTemplate>
        <ItemTemplate>
            <tr>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("CustomerId") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("CompanyName") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("ContactName") %>' />
                </td>
                <td style="text-align: center">
                    <asp:Label runat="server" Text='<%# Eval("Country") %>' />
                </td>
            </tr>
        </ItemTemplate>
    </asp:ListView>

     <asp:DataPager ID="customersListPager" runat="server"
        PagedControlID="customersList" PageSize="5">
        <Fields>
            <asp:NumericPagerField ButtonCount="20" />
        </Fields>
    </asp:DataPager>
```

```
    <asp:SqlDataSource ID="customersDataSource" runat="server"
        ConnectionString="<%$ ConnectionStrings:NorthwindConnectionString %>"
        SelectCommand="SELECT [CustomerID], [CompanyName], [ContactName],
        [Country] FROM [Customers]" />

    </div>
    </form>
</body>
</html>
```

As you can see from Listing 16-8, the *DataPager* control provides a *PageSize* property to define the number of records to show for each page. Under the covers, this property sets the *MaximumRows* property of the *DataPager* itself and also for the paged control, using the corresponding property of the *IPageableItemContainer* interface.

The pager control renders the paging UI using a set of fields that can be defined within the *DataPager* element. These fields describe what kind of paging buttons to provide to the user. For instance, there is a *NextPreviousPagerField* that describes the appearance of the buttons to be used to move to the next page, previous page, first page, and last page. This field also enables you to define the CSS style, the images used, and the visibility of the paging buttons. There is also a *NumericPagerField* that describes the appearance, CSS style, and visibility of the numeric paging buttons. In Listing 16-9, you can see an example of using *DataPager* with a custom set of fields defined.

Listing 16-9 A sample ASPX page showing an excerpt of a *DataPager* control with configured pager fields.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5">
    <Fields>
        <asp:NextPreviousPagerField ButtonType="Link"
            FirstPageText="&lt;&lt;" PreviousPageText="&lt;"
            ShowFirstPageButton="true" ShowLastPageButton="false"
            ShowPreviousPageButton="true" ShowNextPageButton="false"  />
        <asp:NumericPagerField ButtonCount="20" ButtonType="Link" />
        <asp:NextPreviousPagerField ButtonType="Link"
            LastPageText="&gt;&gt;" NextPageText="&gt;"
            ShowFirstPageButton="false" ShowLastPageButton="true"
            ShowPreviousPageButton="false" ShowNextPageButton="true" />
    </Fields>
</asp:DataPager>
```

If you need to customize the paging fields to a great extent, you can define a *TemplatePagerField*, which enables you to freely define the layout of the paging UI. In such a situation, the template will be able to manage the properties of the *DataPager* to be aware of its properties—such as *StartRowIndex*, *PageSize*, and *TotalRowCount*—so that it can generate the appropriate HTML output. In Listing 16-10, you can see an example of using a custom field template to render pages in a drop-down list.

Listing 16-10 A sample ASPX page excerpt using a *DataPager* control with a custom template for pager fields.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5">
    <Fields>
      <asp:TemplatePagerField>
        <PagerTemplate>
            Go to page:
            <uc:PagerDropDownList runat="server" AutoPostBack="true"
                TotalPageCount="<%# Math.Ceiling
                    ((double)Container.TotalRowCount / Container.PageSize) %>" />
        </PagerTemplate>
      </asp:TemplatePagerField>
    </Fields>
</asp:DataPager>
```

In the code in Listing 16-10, the *<uc:PagerDropDownList />* element refers to a custom user control that we defined to handle page numbers that are rendered inside a common drop-down list. In Listing 16-11, you can see the source code of this custom user control.

Listing 16-11 The code of a custom user control to render page numbers in a drop-down list.

```
public class PagerDropDownList : DropDownList {

    public Int32 TotalPageCount { get; set; }

    protected override void OnPreRender(EventArgs e) {
        base.OnPreRender(e);

        for (Int32 c = 1; c <= TotalPageCount; c++) {
            this.Items.Add(new ListItem(c.ToString("00")));
        }

        DataPager pager = ((DataPagerFieldItem)this.Parent).Pager;
        Int32 currentPageIndex = (Int32)Math.Ceiling(
            (double)pager.StartRowIndex / pager.PageSize);

        this.Items[currentPageIndex].Selected = true;
    }

    protected override void OnSelectedIndexChanged(EventArgs e) {
        base.OnSelectedIndexChanged(e);

        Int32 currentPageIndex = Int32.Parse(this.SelectedValue);
        DataPager pager = ((DataPagerFieldItem)this.Parent).Pager;
        pager.SetPageProperties((currentPageIndex - 1) * pager.MaximumRows,
            pager.MaximumRows, true);
    }
}
```

Note that all the pager fields inherit from the abstract base class *DataPagerField*; thus, you can define your own fields if you need to.

One last thing to note about the *DataPager* control is that it provides you with the capability to use a querystring field to receive the current page number to show. By default, the *DataPager* uses HTTP POST commands to navigate through the pages. By the way, sometimes it can be useful to make paging available through querystring parameters. Think about a Web site publishing products grouped by pages. With HTTP POST–based paging, which is the default for *DataPager*, search engines crawl just the first page of each set of products. Using an HTTP GET–based rendering pattern allows for the indexing of contents, based on different URLs for each single page, thereby improving the efficiency of content crawling. To leverage this feature, you need to set the value of the *QueryStringField* property of the *DataPager* control. In the case of a null or empty string value for this property, the *DataPager* control uses HTTP POST; otherwise, it looks for a querystring field declaring the page to show. The name of this querystring field will be the value assigned to the *QueryStringField* property of the *DataPager* instance. In the case of an ASPX page with multiple data sources, data bound controls, and *DataPager* controls, you should set different values of this property for each *DataPager* instance; otherwise, unexpected paging behavior could occur. Consider also that using a querystring-based paging technique forces the *DataPager* fields to render paging buttons as *HyperLink* controls, ignoring any explicit setting of the *ButtonType* property of the fields. In Listing 16-12, you can see an example of using the *QueryStringField* property.

Listing 16-12 A sample ASPX page excerpt with a *DataPager* control using querystring paging.

```
<asp:DataPager ID="customersListPager" runat="server"
    PagedControlID="customersList" PageSize="5"
    QueryStringField="page">
    <Fields>
        <asp:NextPreviousPagerField ButtonType="Link"
            FirstPageText="&lt;&lt;" PreviousPageText="&lt;"
            ShowFirstPageButton="true" ShowLastPageButton="false"
            ShowPreviousPageButton="true" ShowNextPageButton="false"  />
        <asp:NumericPagerField ButtonCount="20" ButtonType="Link" />
        <asp:NextPreviousPagerField ButtonType="Link"
            LastPageText="&gt;&gt;" NextPageText="&gt;"
            ShowFirstPageButton="false" ShowLastPageButton="true"
            ShowPreviousPageButton="false" ShowNextPageButton="true" />
    </Fields>
</asp:DataPager>
```

Note If you want to delve deeper into ASP.NET development techniques, we suggest you read the great book *Essential ASP.NET 2.0* by Fritz Onion with Keith Brown (Addison-Wesley Professional, 2006). To improve your knowledge about ASP.NET 3.5 in particular, we suggest you read the book *Microsoft ASP.NET 3.5 Developer Reference* by Dino Esposito (Microsoft Press, 2008).

LinqDataSource

The biggest news concerning ASP.NET 3.5 from a LINQ point of view is the *LinqDataSource* control. This new control implements the *DataSource* control pattern and allows you to use LINQ to SQL or LINQ to Objects to data bind any bindable rendering control using an ASPX declarative approach. For instance, you can use the *LinqDataSource* control to bind a *DataGrid*, *GridView*, *Repeater*, *DataList*, or *ListView* control. This control resembles the *SqlDataSource* and *ObjectDataSource* from previous versions of ASP.NET, but it targets a LINQ to SQL data model or a custom set of entities using LINQ to Objects.

If you have ever used the *SqlDataSource* and *ObjectDataSource* controls, you probably know that you have to define custom queries or methods, respectively, to provide data querying, inserting, editing, and deleting features. With the *LinqDataSource* control, you can leverage the out-of-the-box LINQ environment to get all of these functionalities, without having to write specific code. Of course, you can always decide to customize this behavior, but it is important to know that it is your choice and it is not mandatory.

For instance, imagine that you have the list of customers from the Northwind database. In Figure 16-1, you can see the complete LINQ to SQL data model schema.

The DBML file can be directly added to the *App_Code* folder of the ASP.NET Web project, or it can be defined in a separate and dedicated class library project. In the case of a sample prototypal solution or a simple Web application, you can declare the data model within the Web project itself. If you are developing an enterprise solution, it is better to divide the layers of your architecture into dedicated projects, thus defining a specific class library for the data model definition.

Note For an enterprise-level solution, read Chapter 15, "LINQ in a Multitier Solution," which discusses how to deal with architectural and design matters related to LINQ adoption.

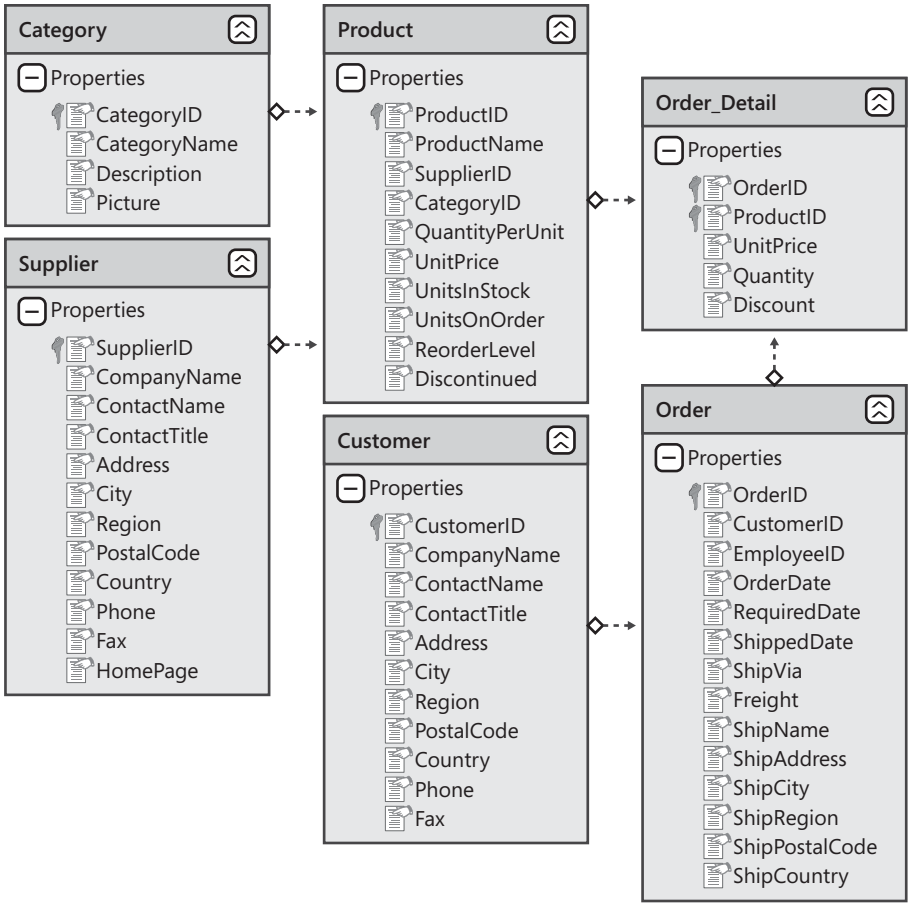


Figure 16-1 The LINQ to SQL Database Model (DBML) schema used in the samples of this chapter

Consider the ASPX page shown in Listing 16-13, where we retrieve a set of customers and bind it to a *GridView* control. As you can see, the *LinqDataSource* control declares a *ContextTypeName* property that maps to the previously defined *NorthwindDataContext* type, and a *TableName* property that corresponds to the *Customers* property of the *DataContext* instance, representing a class of type *Table<Customer>* of LINQ to SQL.

Listing 16-13 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind customers into a *GridView* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-13.aspx.cs"
    Inherits="Listing16_13" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-13</title>
</head>
```

```
<body>
    <form id="form1" runat="server">
    <div>
        <asp:GridView runat="server" DataSourceID="customersDataSource" />
        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext"
            TableName="Customers" />
    </div>
    </form>
</body>
</html>
```

Using Microsoft Visual Studio 2008, you can define these and many other properties using the graphical page designer. In Figure 16-2 and Figure 16-3, you can see the steps for graphically configuring a *LinqDataSource* control by using a designer wizard. To start the wizard you simply need to insert a *LinqDataSource* control into an ASPX page, click on the control task menu, and choose the Configure Data Source task activity. In the first step, presented in the Choose A Context Object screen, you can define the main source for the control.

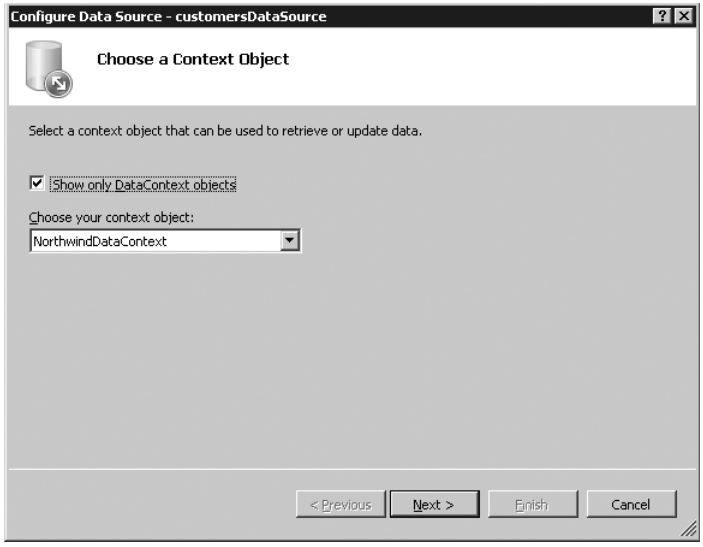


Figure 16-2 The Choose A Context Object screen of the *LinqDataSource* control configuration designer

If you keep the Show Only DataContext Objects option selected, the drop-down list will show only objects inheriting from the LINQ to SQL *DataContext* type, like the *NorthwindDataContext* shown in Figure 16-1. If you clear that option, the designer shows you any .NET type available in the project or in the user-defined projects referenced, including custom entities that can eventually be queried by using LINQ to Objects or any instance of *IEnumerable<T>*.

Note When referencing user-defined class libraries, remember that the *LinqDataSource* designer works with compiled assemblies. Therefore, you need to compile your solution to access newly created or modified types.

In this last case, the drop-down list will also show all the entity types defined in the LINQ to SQL data model, not only the *DataContext* type. This is useful to query a property of type *EntitySet<T>* offered by a specific LINQ to SQL entity—for example, the set of orders of a specific customer instance in a master-detail rendering page.

After clicking the Next button in the designer wizard, you are presented with the Configure Data Selection configuration panel, shown in Figure 16-3.

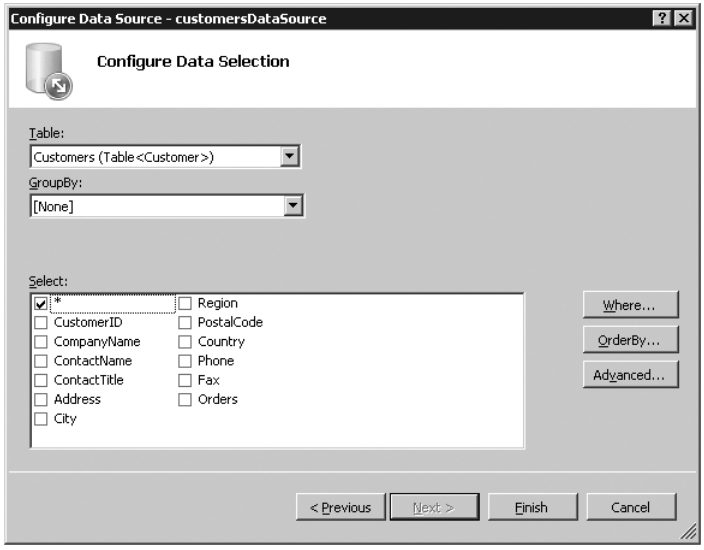


Figure 16-3 The Configure Data Selection screen of the *LinqDataSource* control configuration designer

Here you can define the whole set of configuration parameters of the *LinqDataSource* control. The Table drop-down list allows you to select the *Table<T>* to query, in the case of a *DataContext*, or the *IEnumerable* property of a custom type to query with LINQ to Objects. The GroupBy drop-down list is used to define a grouping rule, eventually with an inner ordering for each group. The Select section provides a list that allows you to choose the fields or properties to select while querying the data source. By default, the *LinqDataSource* control projects all the fields (*), but you can define a custom projection predicate. There are also a set of buttons to define selection rules (the Where button) and ordering conditions (the OrderBy button). Through those buttons, you can define static filtering and ordering rules, but you can also define dynamic parameters, mapping their values to other controls, cookie values, form input elements, ASP.NET *Profile* variables, querystring parameters, or ASP.NET *Session*

variables. This behavior is exactly the same as all the other *DataSource* controls, such as *SqlDataSource* and *ObjectDataSource*.

There is one last command, named Advanced, that determines whether you allow automatic inserts, deletes, or updates. By default, the *LinqDataSource* control simply selects data in a read-only manner. Keep in mind that automatic data modification is allowed only when the following conditions are satisfied: the projection returns all the data source items (having *Select* of "*" enabled), there are no grouping rules, the Context Object is set to a class inheriting from *DataContext,* and consequently the queried collection is of type *Table<T>*. If any of these conditions is not true, a custom LINQ query will occur, and the projected results will be an *IEnumerable<T>*, where *T* is an anonymous type, which is a read-only type by design.

Note Think carefully about the previous conditions. Whenever you need to query an updatable set of items using the automatic engine of the *LinqDataSource* control, you need to query its full set of fields/properties (SELECT * FROM ...) even if you need to change a few of them. It is a behavior that is acceptable in very simple solutions with only one full table mapped directly to the UI, but in more complex and common solutions it is better to use custom selection and data updating rules.

After configuring the *LinqDataSource* control, you are ready to bind it to any bindable control, as we did in Listing 16-13.

All the parameters available through the UI designer can be defined in markup inside the ASPX source code of the page. In fact, the result of using the designer affects the markup definition of the *LinqDataSource* control instance. In Listing 16-14, you can see an example of a *LinqDataSource* control bound to the list of *CompanyName*, *ContactName*, and *Country* for Northwind's customers. The list is filtered by *Country*, mapped to a drop-down list with automatic post-back enabled, and ordered by *ContactName* value.

Listing 16-14 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind's customers into a *GridView* control with some filtering and ordering rules.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-14.aspx.cs"
    Inherits="Listing16_14" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-14</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <br />Country: 
        <asp:DropDownList ID="ddlCountries" runat="server"
            AutoPostBack="True" DataSourceID="countriesDataSource"
            DataTextField="Country" DataValueField="Country" />
        <br />
```

```
        <asp:GridView ID="customersGrid" runat="server"
            DataSourceID="customersDataSource" AutoGenerateColumns="False">
            <Columns>
                <asp:BoundField DataField="CompanyName"
                    HeaderText="CompanyName" ReadOnly="True"
                    SortExpression="CompanyName" />
                <asp:BoundField DataField="ContactName"
                    HeaderText="ContactName" ReadOnly="True"
                    SortExpression="ContactName" />
                <asp:BoundField DataField="Country"
                    HeaderText="Country" ReadOnly="True"
                    SortExpression="Country" />
            </Columns>
        </asp:GridView>
        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext"
            Select="new (CompanyName, ContactName, Country)"
            TableName="Customers" Where="Country == @Country">
            <WhereParameters>
                <asp:ControlParameter ControlID="ddlCountries"
                    Name="Country" PropertyName="SelectedValue"
                    Type="String" />
            </WhereParameters>
        </asp:LinqDataSource>
        <br />
        <asp:LinqDataSource ID="countriesDataSource" runat="server"
            ContextTypeName="NorthwindDataContext" GroupBy="Country"
            OrderGroupsBy="key" Select="new (key as Country)"
            TableName="Customers">
        </asp:LinqDataSource>
    </div>
    </form>
</body>
</html>
```

Under the covers of this page, the *LinqDataSource* control converts its configuration to a dynamic query expression executed against the source context.

Paging Data with *LinqDataSource* and *DataPager*

If you define a *DataPager* control that is applied to a rendering control such as a *ListView* and mapped to a *LinqDataSource* control instance, you will be able to paginate the data source at the level of the LINQ query. In fact, the *LinqDataSource* control will be configured to select a maximum number of records (*MaximumRows*) starting from a start record index (*StartRowIndex*), depending on the *DataPager* configuration. Internally, the *LinqDataSource* control will translate these parameters into a query ending with a *.Skip(StartRowIndex).Take(MaximumRows)* expression.

In Listing 16-15, you can see a sample ASPX page querying the list of Northwind's customers, filterable by country using a drop-down list, and paged with a page size of five customers per page.

Listing 16-15 A sample ASPX page using a *LinqDataSource* control to render the list of Northwind's customers into a *ListView* control with filtering and paging through a *DataPager* control.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-15.aspx.cs"
    Inherits="Listing16_15" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-15</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <br />Country: 
        <asp:DropDownList ID="ddlCountries" runat="server"
            AutoPostBack="True" DataSourceID="countriesDataSource"
            DataTextField="Country" DataValueField="Country" />
        <br />
        <asp:ListView ID="customersList" runat="server"
            DataSourceID="customersDataSource">
            <LayoutTemplate>
                <table cellpadding="5" cellspacing="0" border="1">
                    <tr>
                        <th style="text-align: center">CustomerId</th>
                        <th style="text-align: center">CompanyName</th>
                        <th style="text-align: center">ContactName</th>
                        <th style="text-align: center">Country</th>
                    </tr>
                    <asp:PlaceHolder ID="itemPlaceholder" runat="server" />
                </table>
            </LayoutTemplate>
            <ItemTemplate>
                <tr>
                    <td style="text-align: center">
                        <asp:Label ID="Label1" runat="server"
                            Text='<%# Eval("CustomerId") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label2" runat="server"
                            Text='<%# Eval("CompanyName") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label3" runat="server"
                            Text='<%# Eval("ContactName") %>' />
                    </td>
                    <td style="text-align: center">
                        <asp:Label ID="Label4" runat="server"
                            Text='<%# Eval("Country") %>' />
                    </td>
                </tr>
            </ItemTemplate>
        </asp:ListView>
        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext"
```

```
            Select="new (CustomerID, CompanyName, ContactName, Country)"
            TableName="Customers" Where="Country == @Country">
            <WhereParameters>
                <asp:ControlParameter ControlID="ddlCountries"
                    Name="Country" PropertyName="SelectedValue"
                    Type="String" />
            </WhereParameters>
        </asp:LinqDataSource>
        <asp:DataPager ID="customersPager" PagedControlID="customersList"
            runat="server" PageSize="5">
            <Fields>
                <asp:NumericPagerField ButtonCount="5" ButtonType="Link" />
            </Fields>
        </asp:DataPager>
        <br />
        <asp:LinqDataSource ID="countriesDataSource" runat="server"
            ContextTypeName="NorthwindDataContext" GroupBy="Country"
            OrderGroupsBy="key" Select="new (key as Country)"
            TableName="Customers">
        </asp:LinqDataSource>
    </div>
    </form>
</body>
</html>
```

In Figure 16-4, you can see the HTML output of the page shown in Listing 16-15.

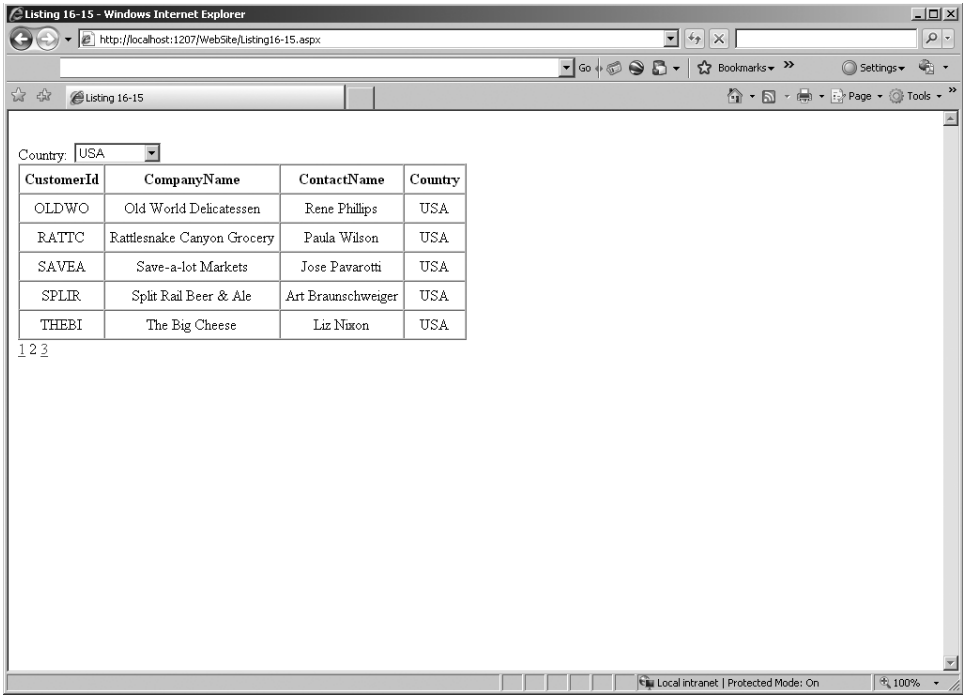


Figure 16-4 The HTML output of the page shown in Listing 16-15.

You can take a closer look at this behavior by subscribing to the *Selecting* event offered by each *LinqDataSource* control instance. This event allows you to inspect the actual selecting, filtering, ordering, grouping, and paging parameters of the executing query. Within the event code, you can also personalize the values of these parameters by customizing the query result before its execution. In Listing 16-16, you can see an example of using the *Selecting* event to define a custom filtering parameter. The *Arguments* property of the *LinqDataSourceSelectEventArgs* instance, received by the event handler, describes the paging configuration.

Note The technique used for pagination by combining a *DataPager* and a *LinqDataSource* is a good approach from an ASP.NET point of view because it requests only the data displayed in one page from the data source. However, if the query is made on a large table without filters in a relational database such as a SQL Server database and the user selects the last page, the resulting query requires a complete scan of the table. Even with an existing index corresponding to the desired order, this operation consumes resources of the database server and time to complete.

Additionally, if the query is the result of complex filters on a large table, each movement in the pages of the *DataPager* control could still require long execution times, because the query is executed every time the user requests a new page. In scenarios where the table size is large and/or the filter operation requires a particular amount of time, you might need to use a more complex architecture with temporary data caching of the results of such a queries.

Listing 16-16 An example of using the *Selecting* event of a *LinqDataSource* control.

```
public partial class Listing16_16 : System.Web.UI.Page {
    protected void customersDataSource_Selecting(Object sender,
        LinqDataSourceSelectEventArgs e) {
        // Forces sorting by ContactName
        e.Arguments.SortExpression = "ContactName";
    }
}
```

Handling Data Modifications with *LinqDataSource*

Whenever you need to modify data through a *LinqDataSource* instance, you need to configure it with a type inherited from *DataContext* as its *ContextTypeName*, a *TableName* corresponding to the name of a property of type *Table<T>* belonging to the *DataContext* type, a null or empty value for the *Select* property, and a null value for the *GroupBy* property. You also need to set to true the *EnableDelete*, *EnableInsert*, or *EnableUpdate* flags to support deletion, insertion, or modification of data items, respectively. As you have already seen, you can enable these flags using the designer interface or the ASPX markup. Regardless of the way in which you configure the *LinqDataSource* instance, after enabling data modifications you will be able to change the data source by using a data-bound control that supports editing, such as *GridView*, *ListView*, or *DetailsView*. In Listing 16-17, you can see an example of an ASPX page that uses an editable *GridView* to show the list of Northwind's customers.

Listing 16-17 A sample ASPX page excerpt using an editable *LinqDataSource* control.

```
<form id="form1" runat="server">
    <div>
        <asp:GridView ID="customersGrid" runat="server"
            DataKeyNames="CustomerID" DataSourceID="customersDataSource"
            AutoGenerateEditButton="true" AutoGenerateColumns="true" />

        <asp:LinqDataSource ID="customersDataSource" runat="server"
            ContextTypeName="NorthwindDataContext" TableName="Customers"
            OrderBy="ContactName" EnableUpdate="true" />

    </div>
</form>
```

If you need to modify or examine the values of the fields before effectively executing data modification tasks, you can subscribe to the events *Deleting*, *Inserting*, *Selecting*, and *Updating*. When you want to examine field values after data modification, you can handle the corresponding post-events, such as *Deleted*, *Inserted*, *Selected*, and *Updated*. Keep in mind that if you want to specify a default value for empty fields, during modification of the data source you can add parameters to the *InsertParameters*, *UpdateParameters*, and *DeleteParameters* of the control.

The events occurring before the data modification receive event arguments specific to each modification operation. For instance, a *Selecting* event handler receives a *LinqDataSourceSelectEventArgs* object, allowing you to customize the query parameters as well as the result, as you will see in the next section.

A *Deleting* event handler receives an argument of type *LinqDataSourceDeleteEventArgs*. This type provides an *OriginalObject* property that contains the data item that will be deleted. It also provides a property named *Exception*, of type *LinqDataSourceValidationException*, that describes any exception related to data validation that occurred within the entity before its deletion. If you want to handle such an exception by yourself and do not want to throw it again, you can set to true the *ExceptionHandled* Boolean property of the event argument.

An *Inserting* event handler receives an argument of type *LinqDataSourceInsertEventArgs*. This type provides the data item that is going to be inserted as the value of its *NewObject* property, while the *Exception* and *ExceptionHandled* properties work just like those for the *Deleting* event argument, obviously referencing validation exceptions related to data item insertion.

An *Updating* event handler receives an event argument of type *LinqDataSourceUpdateEventArgs*, describing the original and actual state of the data item through the *OriginalObject* and *NewObject* properties, respectively. Again, the handling of any kind of validation exception is based on the *Exception* property and *ExceptionHandled* flag.

Maybe you are wondering how the *LinqDataSource* control keeps track of the original values of entities between page postbacks. Under the covers, the *LinqDataSource* control saves each entity field's original value into the page *ViewState*, except for fields that are marked as

UpdateCheck.Never in the data model. For more details about *UpdateCheck* configuration, refer to Chapter 6, "Tools for LINQ to SQL." Using this technique, the *LinqDataSource* control is able to transparently handle data concurrency checks.

In Listing 16-18, you can see an example of code validating the updating of customer information, before any real data modification occurs.

Listing 16-18 The code-behind class of a page handling a custom validation rule while updating customer information through an editable *LinqDataSource* control.

```
protected void customersDataSource_Updating(object sender,
    LinqDataSourceUpdateEventArgs e) {
    if (((Customer)e.OriginalObject).Country != (((Customer)e.NewObject).Country)) {
            e.Cancel = true;
    }
}
```

The interesting part of this approach is that all the plumbing is handled transparently by the *LinqDataSource* control. Later in this chapter, you will see how to manually handle data selections based on custom LINQ queries.

While each pre-event handler receives a specific event argument type instance, all the post-event handlers receive an event argument of type *LinqDataSourceStatusEventArgs*, which allows you to examine the result of the data modification task. In fact, this event argument type provides the resulting data item in its *Result* property in the case of a successful modification. In the case of data modification failure, the post-event argument will have a null value for the *Result* property and will provide the exception that occurred in its *Exception* property. As with pre-events, you can set an *ExceptionHandled* Boolean property of the event argument to not throw the exception again. In Listing 16-19, you can see an excerpt of code handling a concurrency exception while updating a data item.

Listing 16-19 Sample code handling a concurrency exception while editing a data item through a *LinqDataSource* control.

```
protected void customersDataSource_Updated(object sender,
    LinqDataSourceStatusEventArgs e) {
    if ((e.Result == null) && (e.Exception != null)) {
        if (e.Exception is ChangeConflictException) {
            // Handle data concurrency issue
            // TBD ...

            // Stop exception bubbling
            e.ExceptionHandled = true;
        }
    }
}
```

There are three final events that are useful while handling the *LinqDataSource* control: *ContextCreating*, *ContextCreated*, and *ContextDisposing*. The first two occur when the *DataContext* is

going to be created or has been created, respectively. When *ContextCreating* occurs, you can specify a custom *DataContext* instance, setting the *ObjectInstance* property of the event argument of type *LinqDataSourceContextEventArgs* you receive. If you ignore this event, the *LinqDataSource* control will create a *DataContext* by itself, based on the type name provided through the *ContextTypeName* property. You can use this event if you want to customize the *DataContext* creation—for instance, to provide a custom connection string, a user-defined *SqlConnection* instance, or a custom *MappingSource* definition. The *ContextCreated* event, on the other hand, allows you to customize the *DataContext* instance already created. It receives a *LinqDataSourceStatusEventArgs*, like post-event data modification events, but in this case the *Result* property of the event argument contains the *DataContext* type instance. In the case of errors while creating the *DataContext*, you will find an exception in the *Exception* property of the event argument and a value of null for the *Result* property. Listing 16-20 shows an example of handling the *ContextCreated* event to define a custom data shape for querying data.

Listing 16-20 Sample code customizing the *DataContext* of a *LinqDataSource* control, just after its creation.

```
protected void customersDataSource_ContextCreated(object sender,
    LinqDataSourceStatusEventArgs e) {
    NorthwindDataContext dc = e.Result as NorthwindDataContext;
    if (dc != null) {
        // Instructs the DataContext to load orders of current year with
        // each customer instance
        DataLoadOptions dlo = new DataLoadOptions();
        dlo.AssociateWith<Customer>(c => c.Orders
            .Where(o => o.OrderDate.Value.Year == DateTime.Now.Year));

        dc.LoadOptions = dlo;
    }
}
```

The *ContextDisposing* event is useful for handling custom or manual disposing of the *DataContext* type instance and receives an event argument of type *LinqDataSourceDisposeEventArgs*. This event occurs during the *Unload* event of the *LinqDataSource* control and provides the *DataContext* that is going to be disposed of in the *ObjectInstance* property of the event argument.

Note Because the *LinqDataSource* control is hosted by an ASP.NET environment, internally the *DataContext* used to query the data source is created and released for each single request with a pattern similar to the one shown later in this chapter and in Chapter 18, "LINQ and the Windows Communication Foundation." In the case of an uneditable *LinqDataSource* control, the *ObjectTrackingEnabled* property of the *DataContext* instance is set to false; otherwise, it is left to its default value of true.

Using Custom Selections with *LinqDataSource*

Sometimes you need to select data by using custom rules or custom user-defined stored procedures. Whenever you are using a *LinqDataSource* in these situations, you can subscribe to the *Selecting* event of the data source control. The *Selecting* event handler receives a *LinqDataSourceSelectEventArgs* instance that can be used to change the selection parameters, as you saw in the previous section. However, it can also be used to return a completely customized selection by setting the *Result* property of the event argument to a custom value. In Listing 16-21, you can see an example of code handling the *Selecting* event to return the result of a custom stored procedure instead of a standard LINQ to SQL query.

Listing 16-21 The code-behind class of a page handling a custom selection pattern for a *LinqDataSource* control using a stored procedure.

```
protected void customersOrdersDataSource_Selecting(object sender,
    LinqDataSourceSelectEventArgs e) {
    NorthwindDataContext dc = new NorthwindDataContext();
    e.Result = dc.CustOrdersOrders("ALFKI");
}
```

In the case of an editable *LinqDataSource* control, the custom selection code should return a set of items with the same type *T* referenced by the *Table<T>* type of the *TableName* property of the data source control. The code to select the custom result could be any code returning a set of items consistent with the configured *TableName*; thus, you can use a stored procedure as well as customized explicit LINQ to SQL queries. Listing 16-22 shows a *Selecting* event implementation that uses a custom LINQ query to select a customized data source.

Listing 16-22 The code-behind class of a page handling a custom selection pattern for a *LinqDataSource* control using an explicit LINQ query.

```
protected void customersDataSource_Selecting(object sender,
    LinqDataSourceSelectEventArgs e) {
    NorthwindDataContext dc = new NorthwindDataContext();

    e.Result =
        from   c in dc.Customers
        where  c.Country == "USA" && c.Orders.Count > 0
        select c;
}
```

One more thing to notice is that in the case of data paging, the custom result will be paged too. This paging occurs because the *Result* property of the *Selecting* event argument will be managed by the *LinqDataSource* engine anyway, applying any paging rule to it. In the case of an *IQueryable<T>* result, as in Listing 16-22, the paging will occur on the LINQ query expression tree, preserving performance and efficiency.

When you programmatically set a custom result during the *Selecting* event, the *ContextCreated* event of the *LinqDataSource* control is not raised.

Using *LinqDataSource* with Custom Types

A common usage of the *LinqDataSource* control is querying data available through LINQ to SQL, but it can also be used to query custom types and entities. If you define a *ContextTypeName* that does not correspond to a type inherited from the LINQ to SQL *DataContext*, the *LinqDataSource* will switch from LINQ to SQL to LINQ to Objects queries. Consider the code in Listing 16-23, which describes user-defined *Customer* and *Order* types and is not related to a LINQ to SQL data model.

Listing 16-23 User-defined *Customer* and *Order* entities not related to a LINQ to SQL data model.

```
public class Customer {
    public Int32 CustomerID { get; set; }
    public String FullName { get; set; }
    public List<Order> Orders { get; set; }
}

public class Order {
    public Int32 OrderID { get; set; }
    public Int32 CustomerID { get; set; }
    public Decimal EuroAmount { get; set; }
}
```

Now consider a *CustomerManager* class offering a property named *Customers* and of type *List<T>*, where *T* is a *Customer* type, as shown in Listing 16-24.

Listing 16-24 A *CustomerManager* type providing a property of type *List<Customer>*.

```
public class CustomerManager {

    public CustomerManager() {
        this.Customers = new List<Customer> {
            new Customer { CustomerID = 1, FullName = "Paolo Pialorsi",
              Orders = new List<Order> {
                new Order { OrderID = 1, CustomerID = 1, EuroAmount = 100},
                new Order { OrderID = 2, CustomerID = 1, EuroAmount = 200}
              }
            },
            new Customer { CustomerID = 2, FullName = "Marco Russo",
              Orders = new List<Order> {
                new Order { OrderID = 3, CustomerID = 2, EuroAmount = 150},
                new Order { OrderID = 4, CustomerID = 2, EuroAmount = 250},
                new Order { OrderID = 5, CustomerID = 2, EuroAmount = 130},
                new Order { OrderID = 6, CustomerID = 2, EuroAmount = 220}
              }
            },
            new Customer { CustomerID = 3, FullName = "Andrea Pialorsi",
              Orders = new List<Order> {
                new Order { OrderID = 7, CustomerID = 3, EuroAmount = 900},
                new Order { OrderID = 8, CustomerID = 3, EuroAmount = 2500}
              }
            }
```

```
        };
    }

    public List<Customer> Customers { get; set; }
}
```

You can use the *CustomerManager* type as the value for the *ContextTypeName* of a *LinqDataSource* control, while the "Customers" string could be the value for the *TableName* property of the *LinqDataSource* control. Internally, the data source control queries the collection of items returned from the *CustomerManager* class, allowing your code to query a custom set of entities instead of a LINQ to SQL data model. Nevertheless, keep in mind that using LINQ to Objects instead of LINQ to SQL requires you to load into memory the whole set of data—before any filtering, sorting, or paging task. Be careful using this technique because it could be very expensive for your CPU and memory if your code is processing a large amount of data or many page requests. In Listing 16-25, you can see an ASPX page using this kind of configuration.

Listing 16-25 A sample ASPX page using a *LinqDataSource* control linked to a set of user-defined entities.

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Listing16-25.aspx.cs"
    Inherits="Listing16_25" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Listing 16-25</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:GridView ID="customersGrid" runat="server"
            DataSourceID="customersDataSource">
        </asp:GridView>

        <asp:LinqDataSource
            ID="customersDataSource" runat="server"
            ContextTypeName="DevLeap.Linq.Web.DataModel.CustomerManager"
            TableName="Customers" />

    </div>
    </form>
</body>
</html>
```

Binding to LINQ queries

At this point, we have worked with LINQ using the new *LinqDataSource* control. However, in ASP.NET you can bind a bindable control to any kind of data source that implements the *IEnumerable* interface. Every LINQ query, when it is enumerated, provides a result of type *IEnumerable<T>*, which internally is also an *IEnumerable*. Therefore, any LINQ query can be used as an explicit data source in user code. In Listing 16-26, you can see an example of a page using a LINQ query in its *Page_Load* event to bind a custom list of Northwind products to a *GridView* control.

Listing 16-26 A sample page code based on a user-explicit LINQ query in the *Page_Load* event.

```
public partial class Listing16_26 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        NorthwindDataContext dc = new NorthwindDataContext();

        var query =
          from   c in dc.Customers
          where  c.Country == "USA" && c.Orders.Count > 0
          select new {
            c.CustomerID, c.ContactName,
            c.CompanyName, c.Country,
            OrdersCount = c.Orders.Count };

        customersGrid.DataSource = query;
        customersGrid.DataBind();
    }
}
```

These considerations enable you to use the complete LINQ query syntax, particularly LINQ to SQL and LINQ to XML, to query custom data shapes and many different content types, binding the results to an ASP.NET control. For instance, Listing 16-27 shows the code-behind class of an ASPX page that renders the list of posts from one blog via LINQ to XML applied to the blog RSS feed.

Listing 16-27 The code-behind class of a page reading an RSS feed using a LINQ to XML query.

```
public partial class Listing16_27 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        XElement feed = XElement.Load(
            "http://introducingling.com/blogs/MainFeed.aspx");

        var query =
            from   f in feed.Descendants("item")
            select new {
              Title = f.Element("title").Value,
              PubDate = f.Element("pubDate").Value,
              Description = f.Element("description").Value
            };
```

```
        blogPostsGrid.DataSource = query;
        blogPostsGrid.DataBind();
    }
}
```

Querying data sources of any kind and shape using the many different flavors of LINQ is a challenging activity. However, the most common use of user-defined explicit LINQ queries is in the field of LINQ to SQL. Sooner or later, almost every Web application will need to query a set of records from a database. What is of most interest occurs when the data needs to be updated. Using the *LinqDataSource* control, we saw that everything is automated and that the data source control handles updates, insertions, deletions, and selections by itself, eventually raising a concurrency exception when data concurrency issues occur.

When querying data manually, you need to take care of many details by yourself. First of all, consider that ASP.NET is an HTTP-based development platform. Therefore, every request you handle can be considered independent from any other, even if it comes from the same user/browser. Given this, you need to keep track of changes applied to your data between multiple subsequent requests, possibly avoiding use of *Session* variables or shared (static) objects, because this can reduce the scalability of your solution. On the other hand, you should not keep an in-memory instance of the *DataContext* used to query a LINQ to SQL data model. You should instead create, use, and dispose of the *DataContext* for each request, as the *LinqDataSource* control does. In Listing 16-28, you can see an example of this technique for explicitly querying the list of Northwind's customers.

Listing 16-28 The code-behind class of a page explicitly querying Northwind's customers via LINQ to SQL.

```
public partial class Listing16_28 : System.Web.UI.Page {
    protected void Page_Load(object sender, EventArgs e) {
        NorthwindDataContext dc = new NorthwindDataContext();

        var query =
          from   c in dc.Customers
          select new {
            c.CustomerID, c.ContactName, c.Country, c.CompanyName};

        customersGrid.DataSource = query;
        customersGrid.DataBind();
    }
}
```

In this sample, we use the query result to bind a *GridView* control, configured to be editable. Listing 16-29 shows the corresponding ASPX page source code.

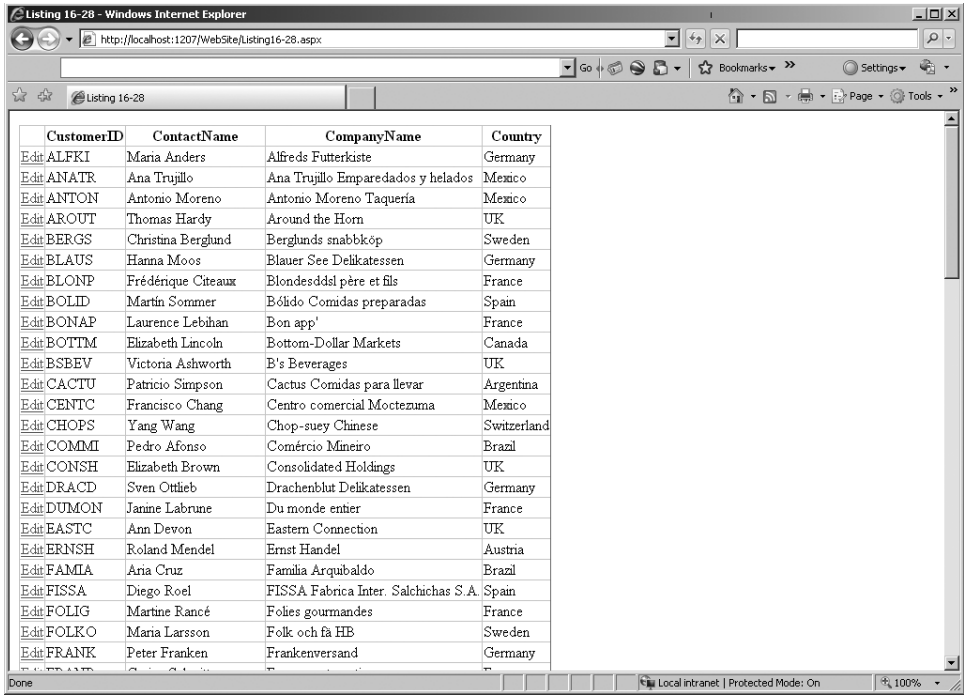
Listing 16-29 The ASPX page code excerpt based on the code-behind class shown in Listing 16-28.

```
<body>
    <form id="form1" runat="server">
    <div>

        <asp:GridView ID="customersGrid" runat="server"
            AutoGenerateEditButton="true"
            AutoGenerateColumns="true" />

    </div>
    </form>
</body>
```

Now here's the interesting part. Imagine that your user decides to change some fields for the currently selected customer, using an editable *GridView* like the one shown in Figure 16-5.



	CustomerID	ContactName	CompanyName	Country
Edit	ALFKI	Maria Anders	Alfreds Futterkiste	Germany
Edit	ANATR	Ana Trujillo	Ana Trujillo Emparedados y helados	Mexico
Edit	ANTON	Antonio Moreno	Antonio Moreno Taquería	Mexico
Edit	AROUT	Thomas Hardy	Around the Horn	UK
Edit	BERGS	Christina Berglund	Berglunds snabbköp	Sweden
Edit	BLAUS	Hanna Moos	Blauer See Delikatessen	Germany
Edit	BLONP	Frédérique Citeaux	Blondesddsl père et fils	France
Edit	BOLID	Martín Sommer	Bólido Comidas preparadas	Spain
Edit	BONAP	Laurence Lebihan	Bon app'	France
Edit	BOTTM	Elizabeth Lincoln	Bottom-Dollar Markets	Canada
Edit	BSBEV	Victoria Ashworth	B's Beverages	UK
Edit	CACTU	Patricio Simpson	Cactus Comidas para llevar	Argentina
Edit	CENTC	Francisco Chang	Centro comercial Moctezuma	Mexico
Edit	CHOPS	Yang Wang	Chop-suey Chinese	Switzerland
Edit	COMMI	Pedro Afonso	Comércio Mineiro	Brazil
Edit	CONSH	Elizabeth Brown	Consolidated Holdings	UK
Edit	DRACD	Sven Ottlieb	Drachenblut Delikatessen	Germany
Edit	DUMON	Janine Labrune	Du monde entier	France
Edit	EASTC	Ann Devon	Eastern Connection	UK
Edit	ERNSH	Roland Mendel	Ernst Handel	Austria
Edit	FAMIA	Aria Cruz	Familia Arquibaldo	Brazil
Edit	FISSA	Diego Roel	FISSA Fabrica Inter. Salchichas S.A.	Spain
Edit	FOLIG	Martine Rancé	Folies gourmandes	France
Edit	FOLKO	Maria Larsson	Folk och fä HB	Sweden
Edit	FRANK	Peter Franken	Frankenversand	Germany

Figure 16-5 The HTML output of the page defined by Listings 16-28 and 16-29.

What you get back when the user presses the Update button on the page is the index of the selected/edited item in the *GridView*, as well as the values of the controls rendering the editable row. To update the data source, you need to create a new *DataContext* instance, which will be used during the entire unit of work that handles this single page request, and query the *DataContext* to get the entity corresponding to the data item that is going to be updated.

After you have retrieved the data item entity from the original store, you can explicitly change its properties, handling the user modifications, and then you can submit changes to the persistence layer, using the *SubmitChanges* method of the *DataContext* object. In Listing 16-30, you can see an example of the required code.

Listing 16-30 The code-behind class of a page explicitly updating a Northwind's customer instance with LINQ to SQL.

```
protected void UpdateCustomerInstance(String customerID,
        String contactName, String country, String companyName) {
    NorthwindDataContext dc = new NorthwindDataContext();

    Customer c = dc.Customers.First(c => c.CustomerID == customerID);
    if (c != null) {
        c.ContactName = contactName;
        c.Country = country;
        c.CompanyName = companyName;
    }

    dc.SubmitChanges();
}
```

The preceding example is missing an important part of the process: the concurrency check. In fact, we simply use the input coming from the user to modify the persistence layer. However, we have no guarantees about the exclusiveness of the operation we are performing. When the user decides to update a data item, you can leverage a specific *DataContext* method, called *Attach*. This method allows you to attach an entity to a *DataContext* instance, eventually providing its original state to determine whether any modification happened or simply to notify the *DataContext* that the entity has been changed. If the entity type has an *UpdateCheck* policy defined (see Chapter 6 for further details), the *DataContext* will be able to reconcile the entity with the one actually present in the database. In Listing 16-31, you can see an example of a code-behind class using this technique.

Listing 16-31 The code-behind class of a page updating a Northwind customer instance with LINQ to SQL, tracking the original state of the entity.

```
protected void AttachAndUpdateCustomerInstance(String customerID,
    String contactName, String country, String companyName) {

    NorthwindDataContext dc = new NorthwindDataContext();

    Customer c = new Customer();
    c.CustomerID = customerID;
    c.ContactName = contactName;
    c.Country = country;
    c.CompanyName = companyName;

    // The Boolean flag indicates that the item has been changed
    dc.Customers.Attach(c, true);
    dc.SubmitChanges();
}
```

Summary

This chapter showed you how to leverage the new features and controls available in ASP.NET 3.5 to develop data-enabled Web applications, using LINQ to SQL and LINQ in general. Consider that what you have seen is really useful for rapidly defining Web site prototypes and simple Web solutions. On the other hand, in enterprise-level solutions you will probably need at least one intermediate layer between the ASP.NET presentation layer and the data persistence one, represented by LINQ to SQL. In real enterprise solutions, you usually also need a business layer that abstracts all business logic, security policies, and validation rules from any kind of specific persistence layer. And you will probably have a Model-View-Controller or Model-View-Presenter pattern governing the UI. In this more complex scenario, chances are that the *LinqDataSource* control will be tied to entities collections more often than to LINQ to SQL results.

Programming Microsoft® LINQ

Dig into LINQ—and transform the way you work with data.

With LINQ, you can query data—no matter what the source—directly from Microsoft Visual Basic® or C#. Guided by two data-access experts who've worked in depth with LINQ and the Microsoft development teams, you'll learn how Microsoft .NET Framework 3.5 implements LINQ, and how to exploit it. Study and adapt the book's examples—and deliver your *own* solutions faster and with leaner code.

Discover how to:

- Transcend data boundaries using LINQ's unified syntax
- Query relational databases with *LINQ to SQL*—and dynamically manage tables, views, and stored procedures
- Read, write, and manage XML content more efficiently with *LINQ to XML*
- Explore how LINQ works with Windows® Communication Foundation, Windows Presentation Foundation, Microsoft Silverlight™, and ASP.NET
- Review best practices for data-enabled Web applications and service development
- Extend LINQ—creating custom operators and providers
- Get a preview of *Parallel LINQ* (*PLINQ*) and *LINQ to Entities*

Get code samples on the Web

*For **system requirements**, see the Introduction.*


About the Authors

Paolo Pialorsi is a consultant, trainer, and author who specializes in developing solutions with Microsoft .NET, XML, and Web services. He has written four books and speaks at industry conferences.

Marco Russo trains and consults with professional developers working with the .NET Framework and Microsoft SQL Server®. He's active in developer communities and blogs, and has written three books.

The authors are founders of DevLeap, a firm dedicated to educating and mentoring the professional developer community.

RESOURCE ROADMAP

Developer Step by Step
- Hands-on tutorial covering fundamental techniques and features
- Practice files on CD
- Prepares and informs new-to-topic programmers

Developer Reference
- Expert coverage of core topics
- Extensive, pragmatic coding examples
- Builds professional-level proficiency with a Microsoft technology

Focused Topics
- Deep coverage of advanced techniques and capabilities
- Extensive, adaptable coding examples
- Promotes full mastery of a Microsoft technology

See inside cover for more information

ISBN-13: 978-0-7356-2400-9
ISBN-10: 0-7356-2400-3

Part No. X14-71508

U.S.A. $49.99
[*Recommended*]

Programming/
Microsoft Visual Studio/LINQ

Microsoft® Visual Studio® 2008

Microsoft®