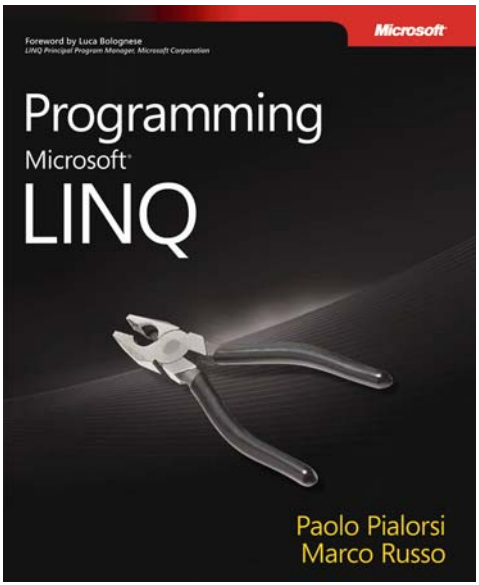


# Programming Microsoft® LINQ

*Paolo Pialorsi*
*Marco Russo*

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/10827.aspx

9780735624009

Microsoft Press

# Table of Contents

# Part V Applied LINQ

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.

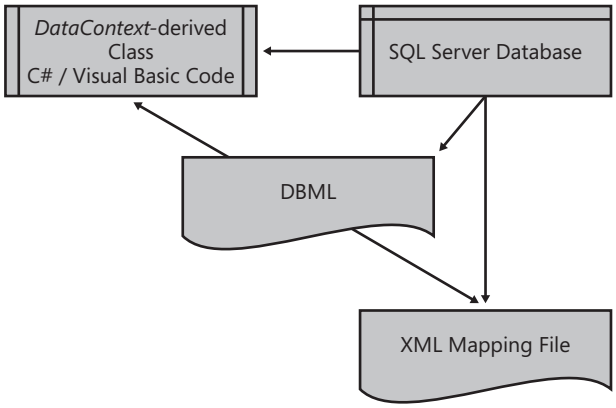


**Figure 6-1** Relationships between file types and the relational database

## 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.

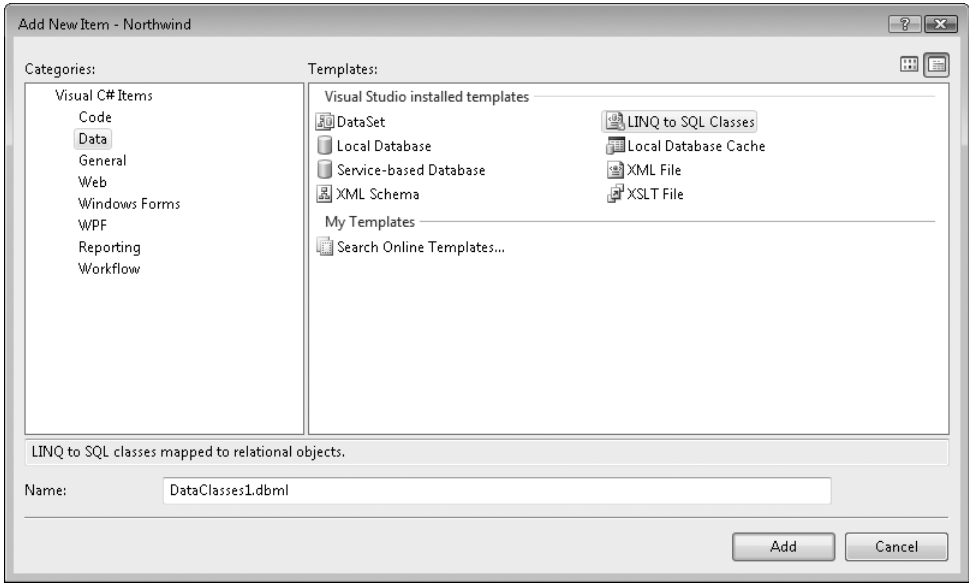


**Figure 6-2** Add New Item dialog box

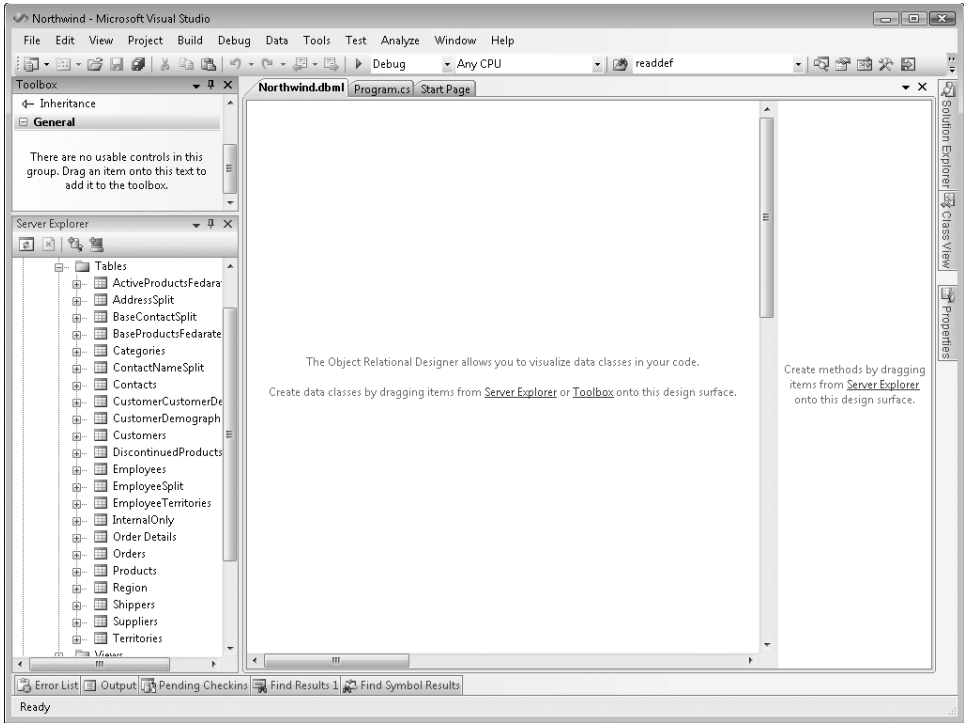


**Figure 6-3** Empty DBML file opened with the Object Relational Designer

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.

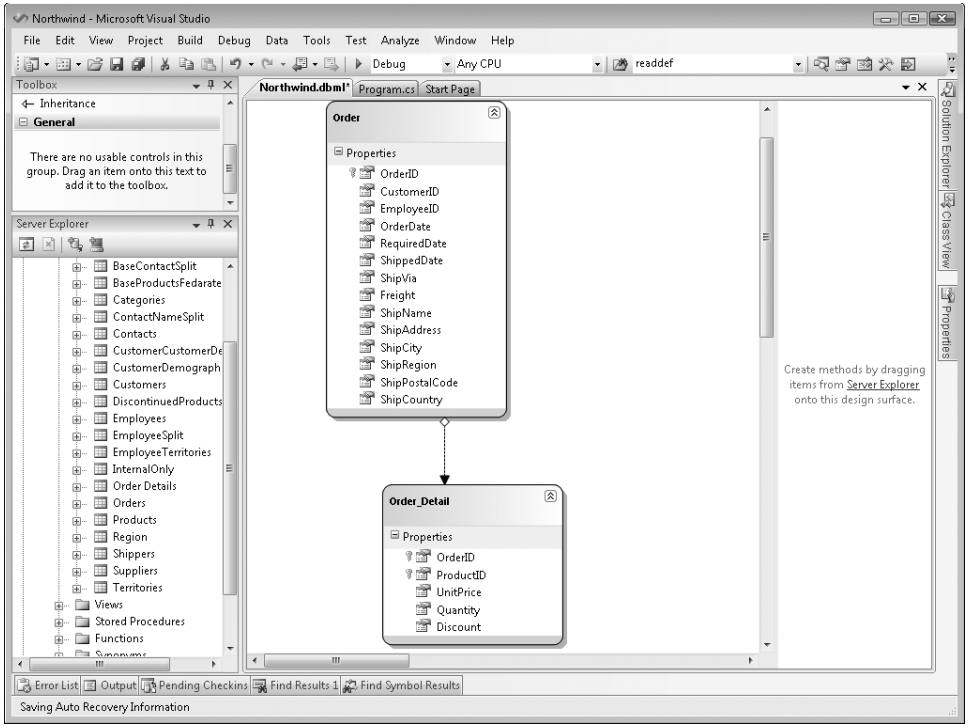


**Figure 6-4** Two entities created from a server connection

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.

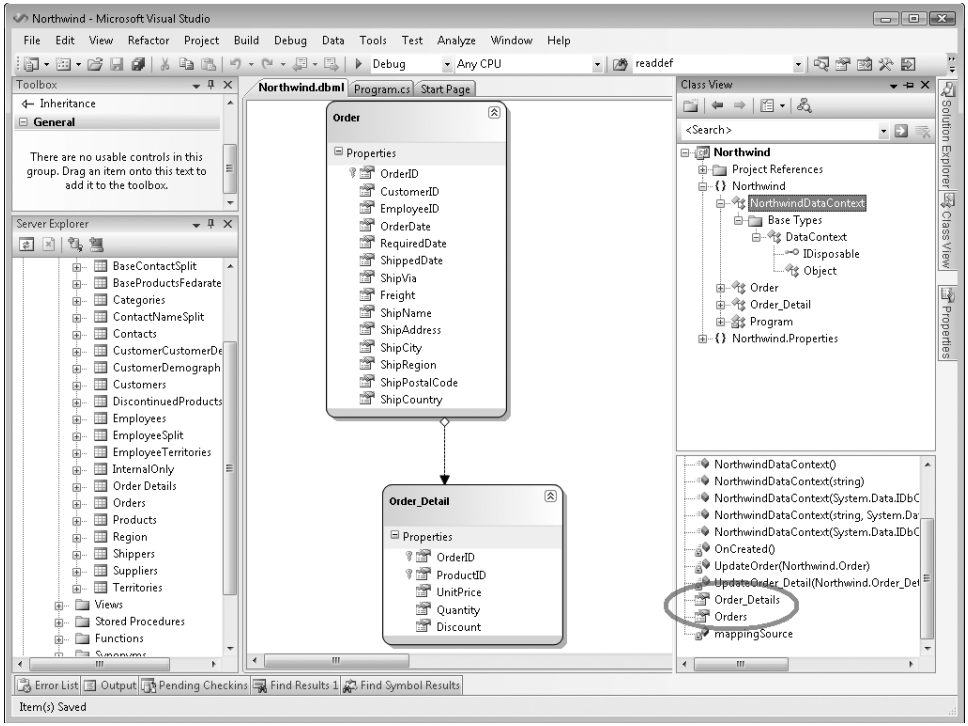


**Figure 6-5** Plural names for *Table<T>* properties in a *DataContext*-derived class

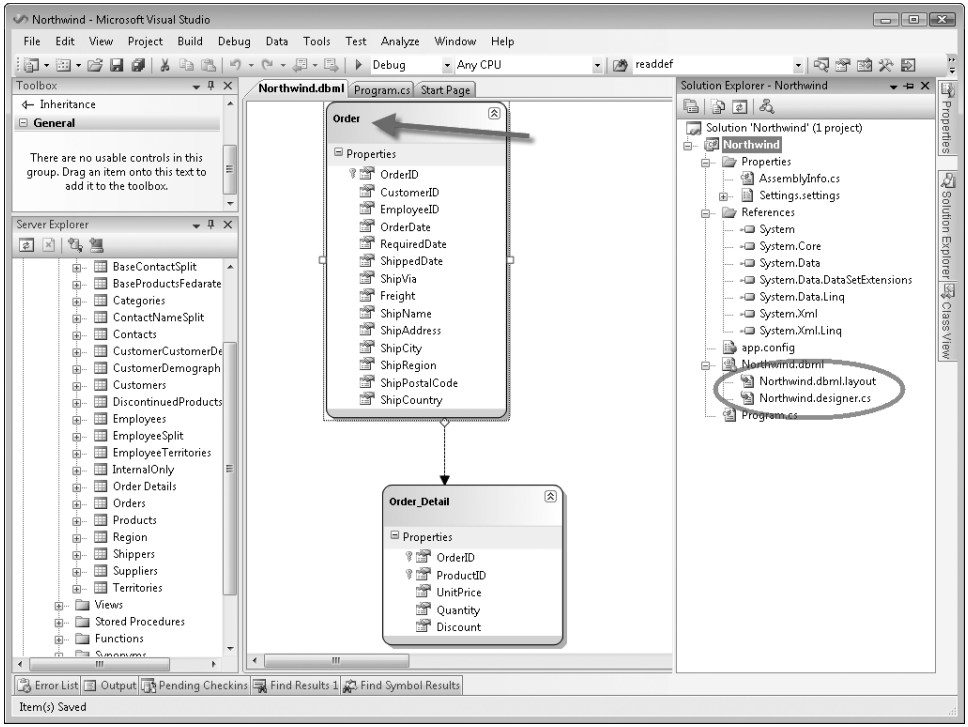


**Figure 6-6** Files automatically generated for a DBML file are shown in Solution Explorer

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.

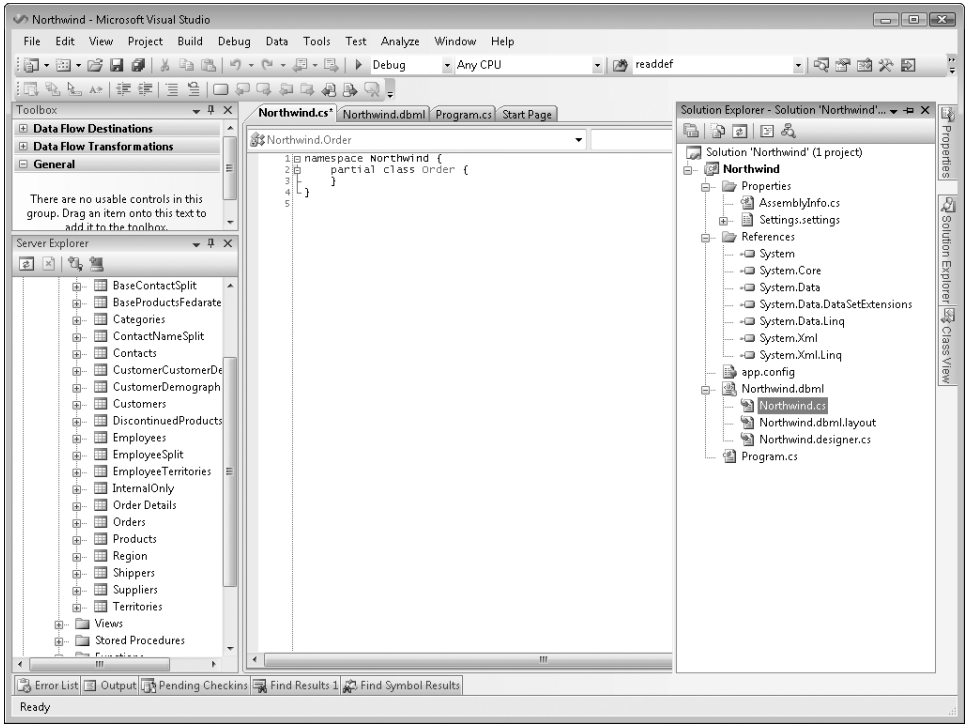


**Figure 6-7** Custom code is stored in a separate file under the DBML file in Solution Explorer

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.

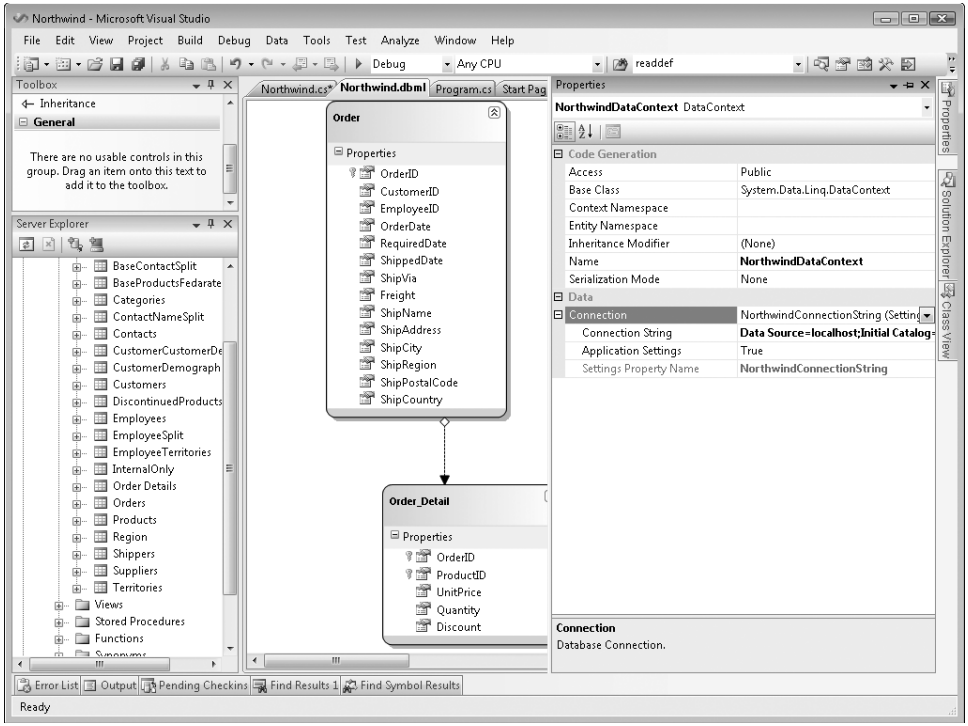


**Figure 6-8** *DataContext* properties

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.

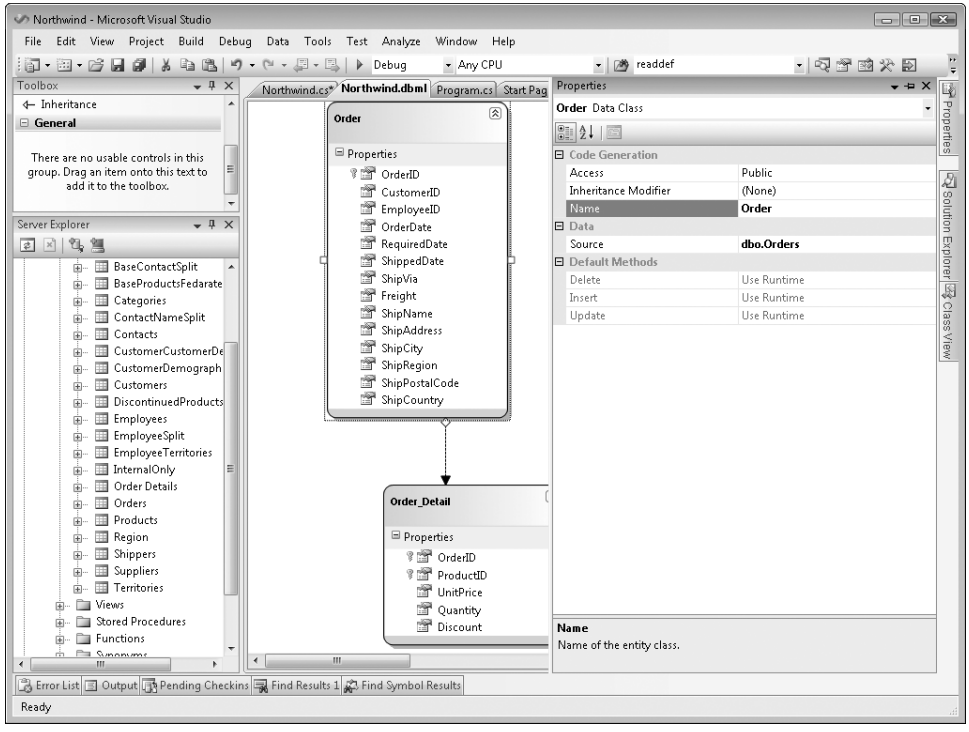


**Figure 6-9** Entity class properties

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.<br>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.

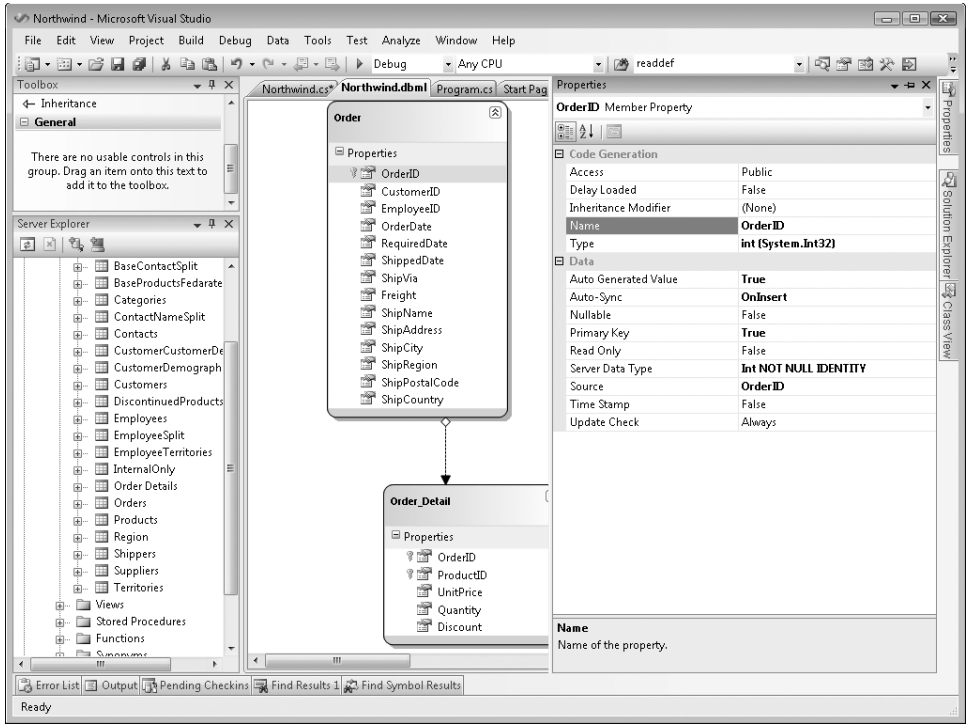


**Figure 6-10** Entity member properties

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.

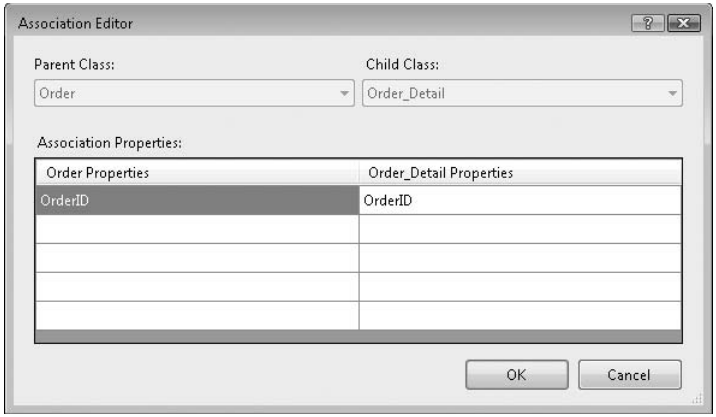


**Figure 6-11** Association properties

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.

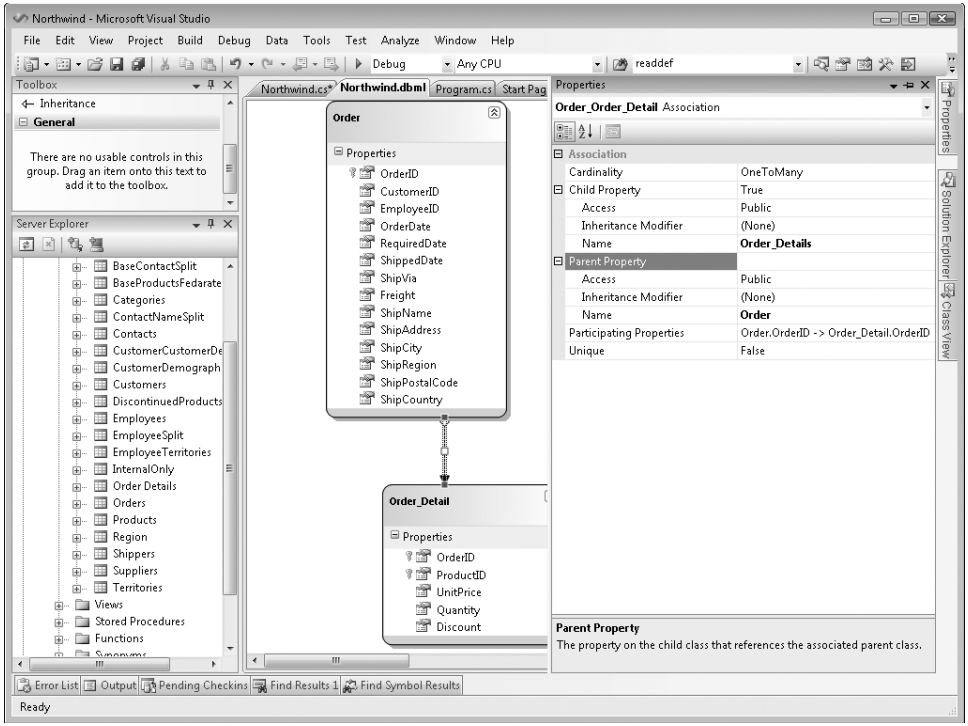


**Figure 6-12** Association properties

**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.<br>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.

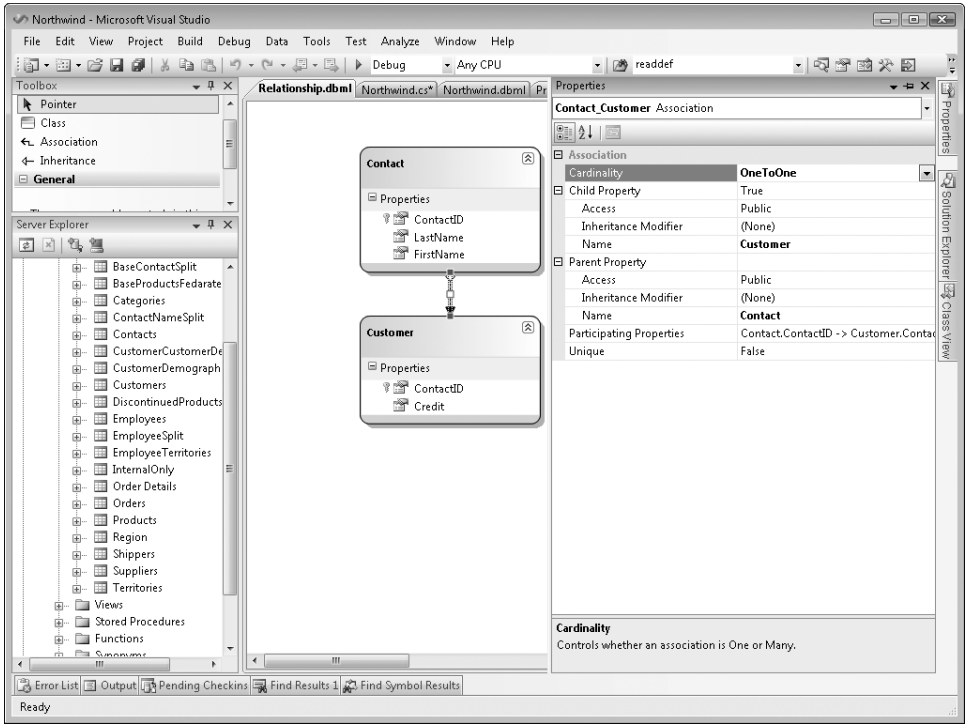


**Figure 6-13** Association properties of a one-to-one relationship

*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.

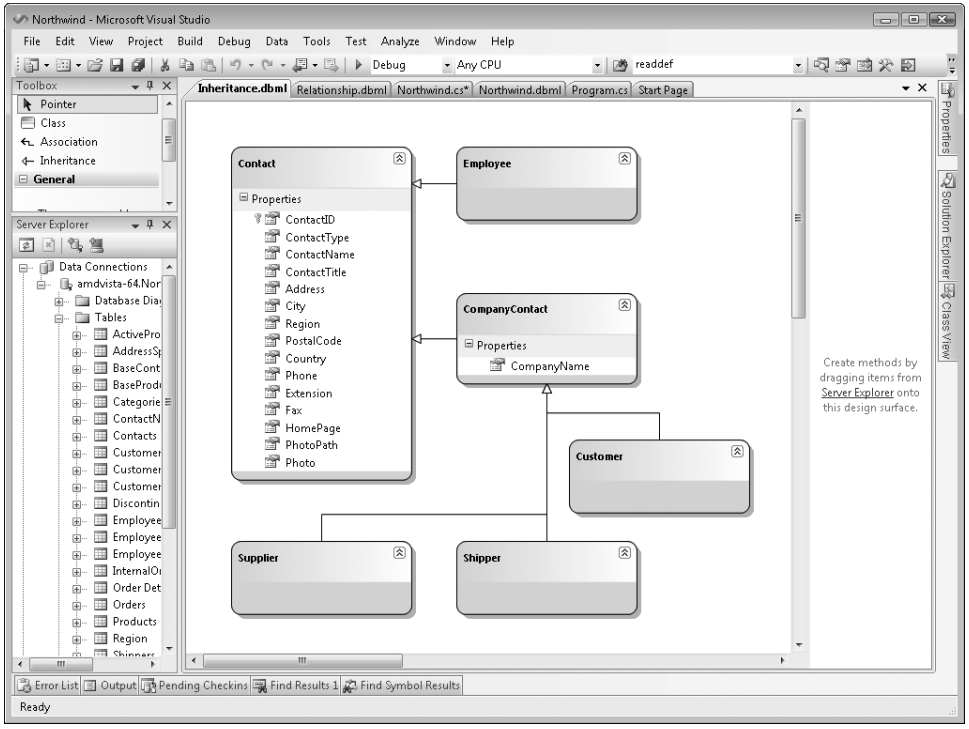


**Figure 6-14** Design of a class hierarchy based on the Northwind.Contact table

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.

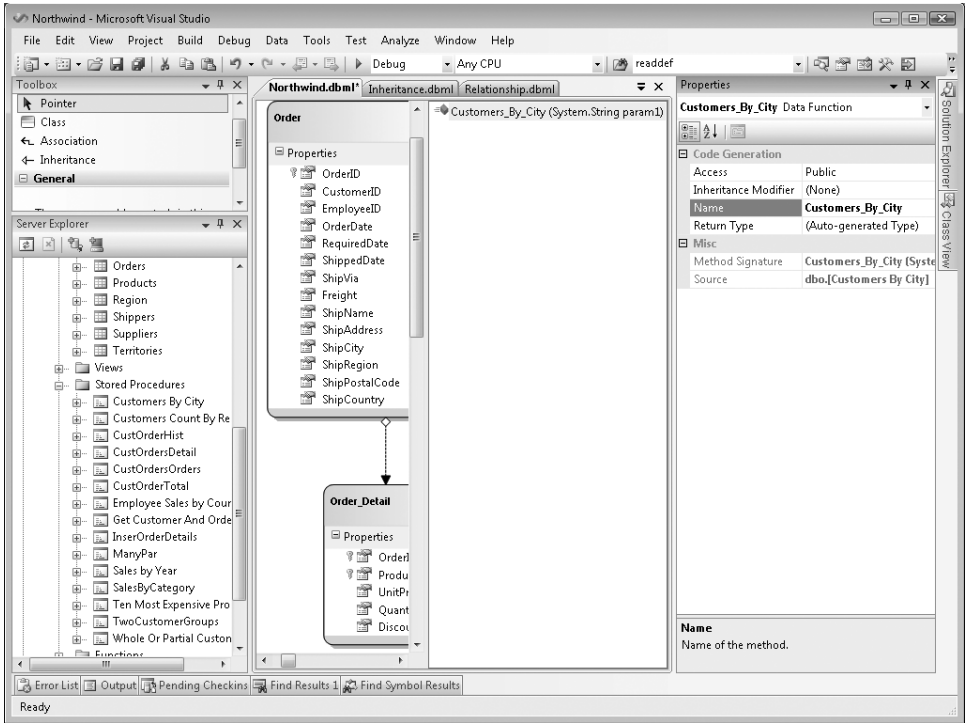


**Figure 6-15** Stored procedure imported into a DBML file

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.

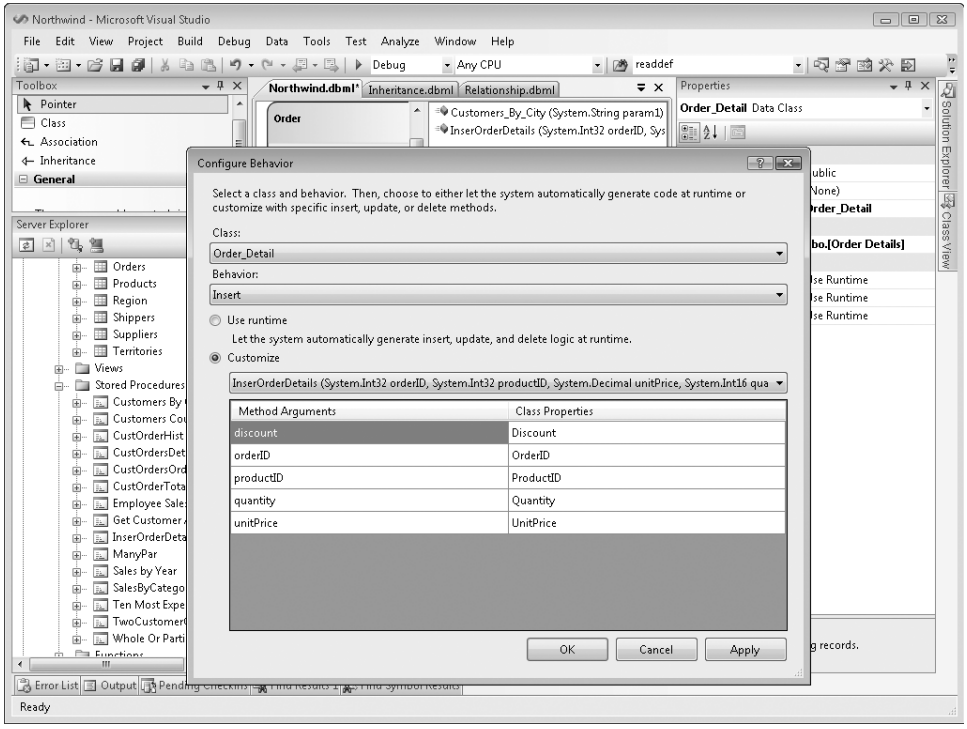


**Figure 6-16** Use of a stored procedure to insert an *Order_Detail*

**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.