ADO.NET: From Novice to Pro, Visual Basic .NET Edition

PETER WRIGHT

Apress™

ADO.NET: From Novice to Pro, Visual Basic .NET Edition
Copyright © 2002 by Peter Wright

ISBN (pbk): 1-59059-060-0

Printed and bound in the United States of America 12345678910

Technical Reviewer: Ildiko Blackburn

Editorial Directors: Dan Appleman, Gary Cornell, Jason Gilmore, Simon Hayes, Karen Watterson, John Zukowski

Managing Editor: Grace Wong

Project Manager: Tracy Brown Collins

Copy Editor: Nicole LeClerc

Production Manager: Kari Brooks

Compositor: Susan Glinert

Artist and Cover Designer: Kurt Krames

Proofreaders: Beth Christmas and Brendan Sanchez

Indexer: Carol Burbo

Manufacturing Manager: Tom Debolski

Marketing Manager: Stephanie Rodriguez

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States, phone 1-800-SPRINGER, email `orders@springer-ny.com`, or visit `http://www.springer-ny.com`.

Outside the United States, fax +49 6221 345229, email `orders@springer.de`, or visit `http://www.springer.de`.

For information on translations, please contact Apress directly at 2560 9th Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com` in the Downloads section.

CHAPTER 6

Introducing Windows Forms Applications

"One picture is worth ten thousand words."

—Frederick R. Barnard, *Printer's Ink*, 1921

VISUAL STUDIO .NET'S support for the console (text-based output into a DOS-like window) is wonderful. By using the console application templates it's dead easy to knock out programs to try out ideas, test classes and, of course, focus on learning important concepts. However, in the world of production applications, the GUI is king, whether it's deployed as part of a desktop application or over the Web using ASP.NET.

In this chapter and the next you'll look at those two things: using ADO in Windows Forms–based desktop applications and using ADO in Web-based applications. Before going any further, though, I need to evangelize a little, particularly for those of you out there who've deployed production applications prior to .NET.

You see, developers are a strange bunch. On the one hand, they all love powerful frameworks, such as .NET, and truly appreciate the amount of work a powerful, well-written framework can save them. On the other hand, there comes a point where a framework can do so much and take so much out of the hands of the programmer that many developers resist its adoption. Sometimes this resistance is well placed. Take Visual Basic, for example (not VB .NET). VB has historically included a number of controls for data binding—binding data from data sources and automatically displaying it on-screen through bound controls. Historically, this always involved something known as a *data control.* On the surface it looked neat: Stick a data control on a form, bind a number of other GUI controls to the data control, and then at runtime just load some data into the data control and you instantly have a form of data the user can view, update, and navigate, with little or no effort on the part of the developer. This power, though, also took away control from the developer. In earlier versions of VB in particular, developers taking this approach were forced to fight the data control to claw back some degree of control over how the user worked with data, and they inevitably ended up writing more code than if they had just loaded the data and manually populated the controls on the form themselves. The data control was also somewhat inefficient in the way it did things.

For these reasons, there's a lot of resistance to "bound" user interfaces. Visual Studio .NET and the user interface controls in the .NET Framework support data binding, but it is quite unlike anything you've seen before. The decision to use bound controls in .NET is not a decision to forgo control and power in the pursuit of an easy life. The bound controls in .NET do not limit the control programmers have over their data access architecture, and they do not impose unwanted restrictions on the design of the GUI. Bound controls in .NET are incredibly powerful, stunningly easy to use, and if you have experience with prior versions of ADO, DAO, and RDO, you should be pleasantly surprised by just what you can accomplish with them.

In this chapter I'll lead you through some hands-on examples of common ADO-related GUI tasks. You'll see how to use a DataGrid, how to use the Visual Studio wizards and designers to create your DataAdapters and Connections with little or no code, and how to data bind any visual control you choose. In the next chapter you'll drill down in more detail and learn how to take control of data binding through code, and also how to validate and explore data in visual bound controls, again through code.

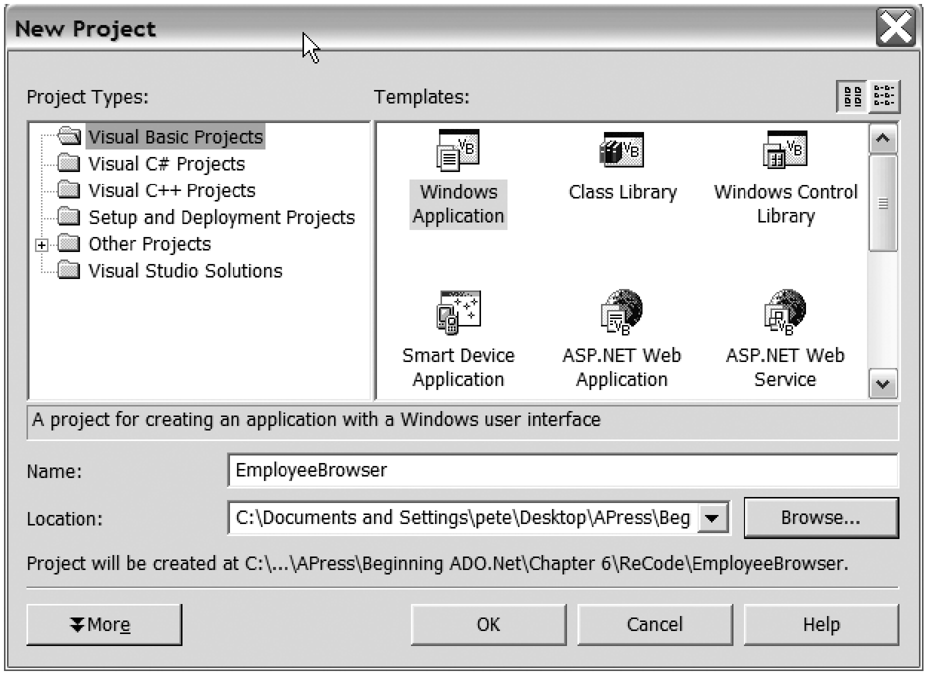
A Simple Complex-Bound Example

Strange heading, isn't it? In a moment you're going to walk through a very simple data-binding solution, connect to a database, and display the results of a select inside a grid. The complex part of the equation is simply the name given to the type of binding that you're going to use. Windows Forms applications can make use of either **simple binding** or **complex binding**. *Simple binding,* as its name suggests, is pretty trivial; you just bind a property of a control to a column in a data source (DataSet, DataTable, or DataView). The key here, and I'm going to put this in bold just so you don't miss it, is **with simple binding you can bind ANY property of a control to a column**. Think about that for a second. Just imagine that you have a picture in a database but also store the picture's dimensions. Using simple binding, you can pull the picture itself into a picture box and then automatically set up the picture box's dimensions based on the size of the image in the database.

Complex binding is the process of linking one or more data sources into a control and allowing the user to navigate around the data. For example, the DataGrid that you're about to use has the capability to display and let the user navigate around multiple related tables. Obviously, that's quite complex.

Let's get started on the application. You're going to create a Windows Forms application that lets the user see information on employees in the Northwind table and also enables them to update the employee information.

Create a new project in Visual Studio .NET, but unlike the previous examples, select the VB .NET Windows Application as the template, as shown in Figure 6-1.



A project for creating an application with a Windows user interface

Name: EmployeeBrowser

Location: C:\Documents and Settings\pete\Desktop\APress\Beg

Project will be created at C:\...\APress\Beginning ADO.Net\Chapter 6\ReCode\EmployeeBrowser.

Figure 6-1. Select the VB .NET Windows Application template for the projects in this chapter.

Name your project whatever you like—I chose EmployeeBrowser just to make it obvious. After a short pause the project should be created and you'll be looking at a blank form ready to start work.

This is really just a two-tier application—all your logic is going to go behind the form, and you're going to connect straight to the database from it. In the next chapter you'll see how to add tiers to grab and validate the data, something you'll be doing a lot of when you move into the world of ASP.NET and Web service development.

Think back to the previous discussions on ADO.NET for a moment. You need to connect to the database and grab a list of employees. Once that list is grabbed, you're going to allow the user to update the data, add new items, and save it all en masse when they're done. Obviously, this is going to require a Connection, a Data-Adapter, and a DataSet. Now, you could go ahead and create these things in code, but that would be denying yourself access to some of the really powerful features of the Visual Studio .NET IDE.

Open the Toolbox at the left of the Form Designer and then select the Data tab, just like in Figure 6-2.

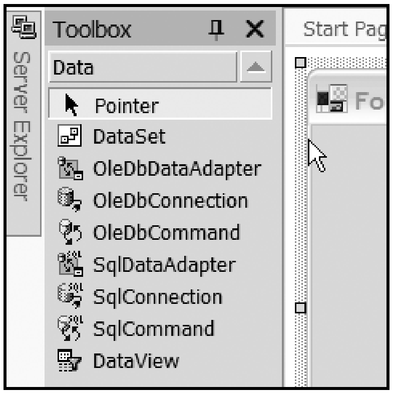


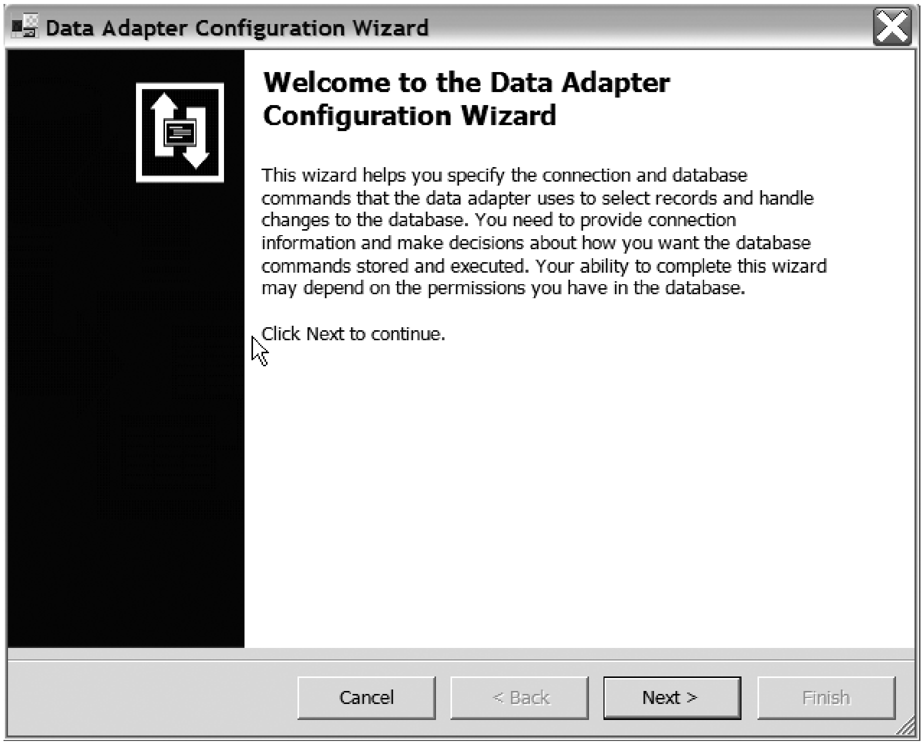
Figure 6-2. The Data tab of the Toolbox, within the VS.NET IDE

The list may surprise you. You've already done a lot of work with ADO DataSets, Connections, and Adapters, but there they all are, shown on the Data tab within the Visual Studio Designer's Toolbox. Of course, these components don't have any visual aspects to them, but double-clicking them will add them to the form in code for you. More to the point, some very handy wizards and other features come into play if you work with them this way, instead of through code.

Go ahead and double-click the SqlDataAdapter to start one such wizard—the Data Adapter Configuration Wizard—shown in Figure 6-3 in the next section.

The Data Adapter Configuration Wizard

Setting up everything you need to work with a DataAdapter and Connection in code can be a time-consuming and data entry–intensive process. The wizard can do it all for you. Now, before those with a pathological fear of anything that saves time and effort run for the hills, I should point out that the wizard is very flexible. You can override much of its operation and completely drill down into every aspect of setting the Adapter up. So, use it. The wizard in this case is your friend. Figure 6-3 shows the first page of the wizard.



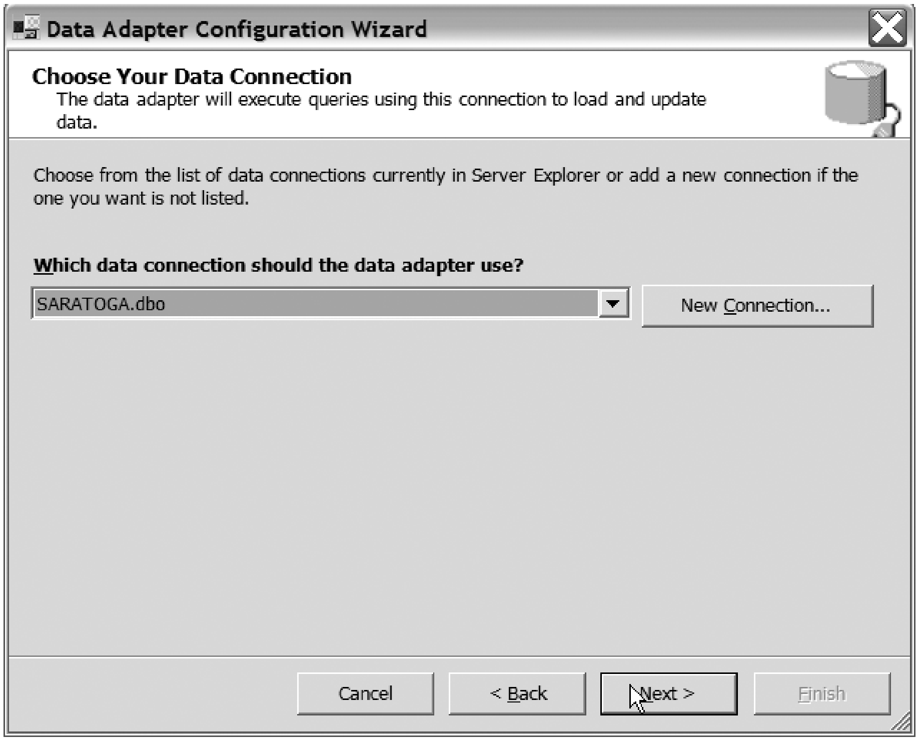
Welcome to the Data Adapter Configuration Wizard

This wizard helps you specify the connection and database commands that the data adapter uses to select records and handle changes to the database. You need to provide connection information and make decisions about how you want the database commands stored and executed. Your ability to complete this wizard may depend on the permissions you have in the database.

Click Next to continue.

Figure 6-3. The start of the Data Adapter Configuration Wizard

Click the Next button to advance to the second page of the wizard, and you'll see a window like that shown in Figure 6-4.


Data Adapter Configuration Wizard

Choose Your Data Connection

The data adapter will execute queries using this connection to load and update data.

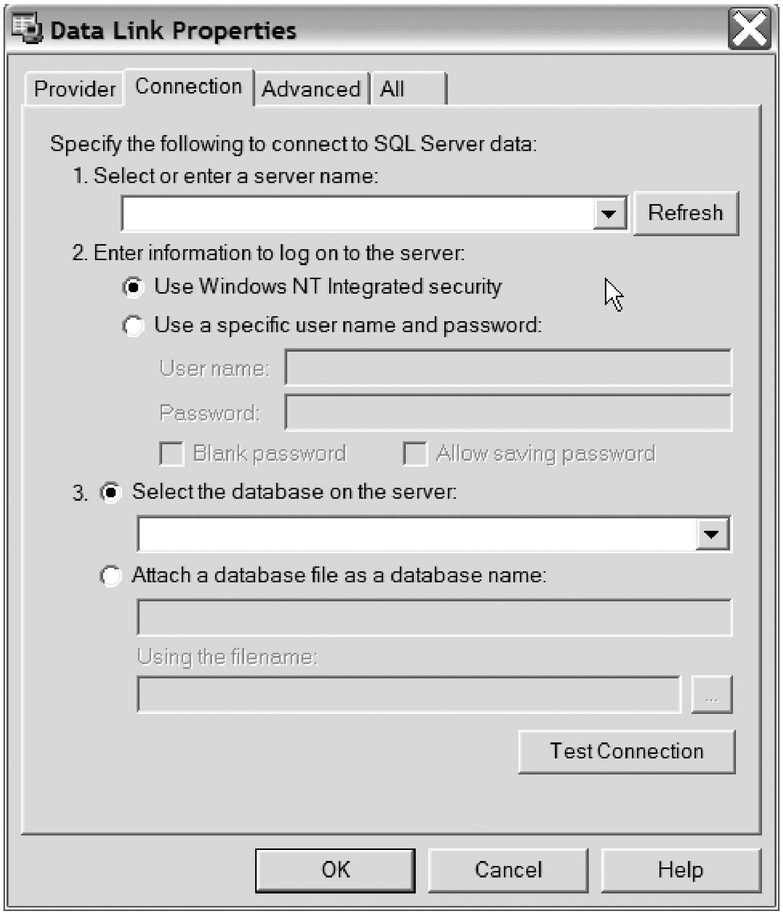
Choose from the list of data connections currently in Server Explorer or add a new connection if the one you want is not listed.

Which data connection should the data adapter use?

SARATOGA.dbo

Figure 6-4. Page 2 of the Data Adapter Configuration Wizard deals with setting up the database connection that your new Adapter is going to use.

The connection part of the wizard will appear. Unless you've used the wizard before, you'll probably find that the database connection shown doesn't match the one you need. In Figure 6-4, for example, the connection that's about to be used will connect to the master database on my Saratoga (my machines are named after ships on *Star Trek*) server. This isn't a problem—just click the New Connection button to advance to the next dialog box, which is shown in Figure 6-5.



Specify the following to connect to SQL Server data:

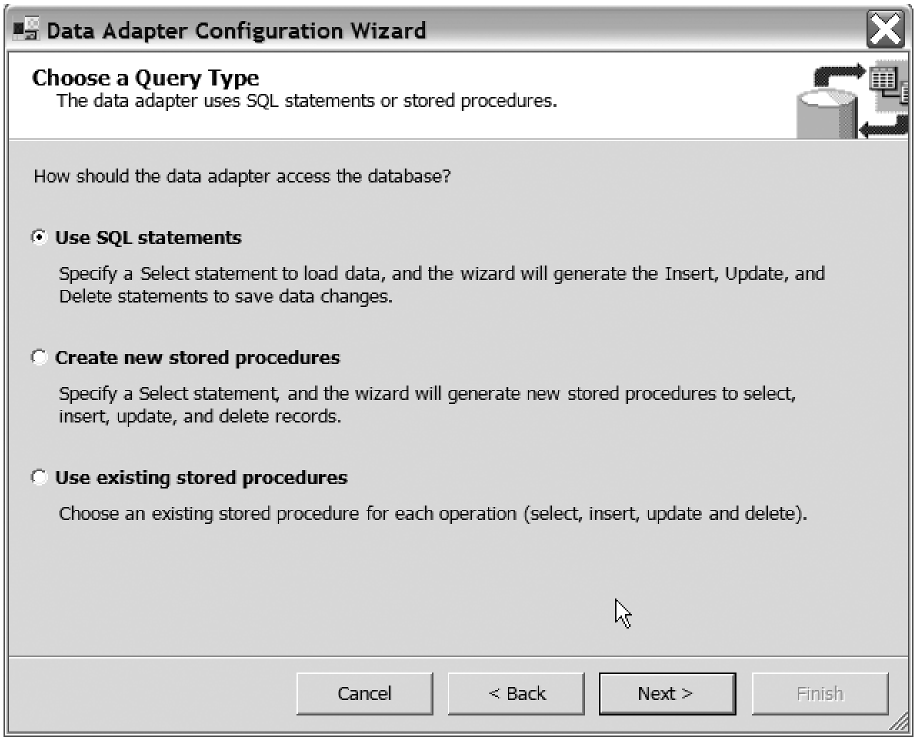
1. Select or enter a server name:

2. Enter information to log on to the server:
   - [✓ Use Windows NT Integrated security]
   - Use a specific user name and password:
     - User name:
     - Password:
     - Blank password
     - Allow saving password

3. [✓ Select the database on the server:]
   - Attach a database file as a database name:
   - Using the filename:

Figure 6-5. The Data Link Properties dialog box

The Data Link Properties dialog box will appear. This dialog box allows you to not only set up every aspect of the connection that you want your DataAdapter to use, but you can also use it to test the connection before it gets embedded in your code. Go ahead and choose, or enter, your own server name and the Northwind database on that server. Clicking OK will save the details you enter and return you to the second page of the Data Adapter Configuration Wizard, where you now just need to click Next to move on to the third page. (Incidentally, the connection information is saved so that the next time the wizard runs, you can just select the connection from the drop-down list on the second page of the wizard.) Your window should now look like Figure 6-6.



Choose a Query Type
The data adapter uses SQL statements or stored procedures.

How should the data adapter access the database?

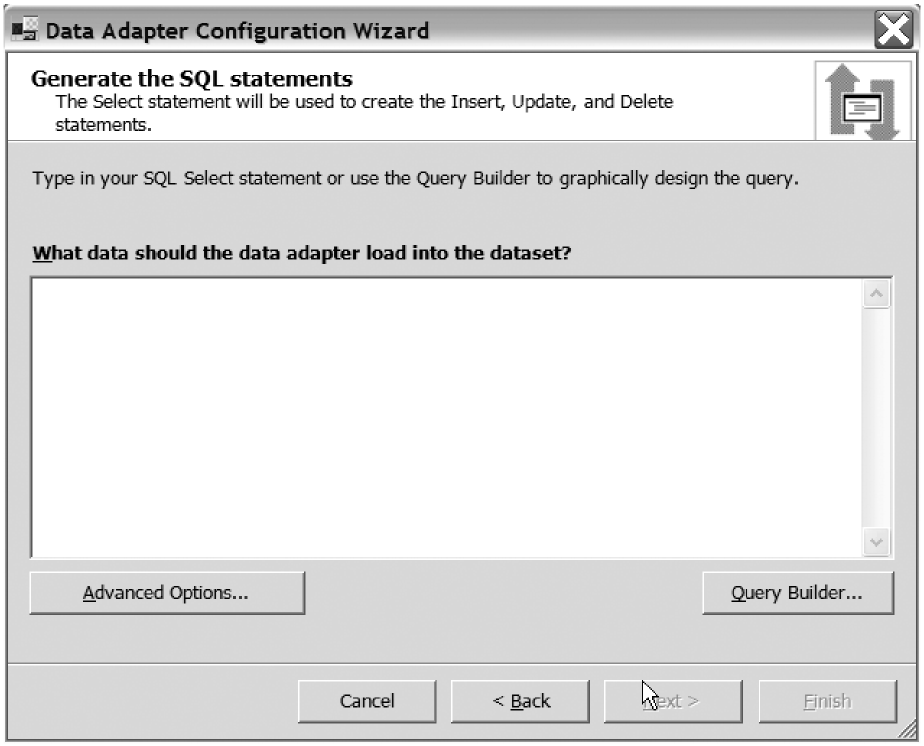
[✓ Use SQL statements]
Specify a Select statement to load data, and the wizard will generate the Insert, Update, and Delete statements to save data changes.

Create new stored procedures
Specify a Select statement, and the wizard will generate new stored procedures to select, insert, update, and delete records.

Use existing stored procedures
Choose an existing stored procedure for each operation (select, insert, update and delete).

Figure 6-6. The Query Type page

The Query Type page will appear. For now, you can just click the Next button. By default the wizard expects that you're going to want to embed SQL in your code, but as Figure 6-6 shows, you can also choose to create brand-new stored procedures or use existing ones. You'll look at these options later. Clicking Next with the "Use Sql statements" option selected takes you to the Query Builder page shown in Figure 6-7.



Generate the SQL statements
The Select statement will be used to create the Insert, Update, and Delete statements.

Type in your SQL Select statement or use the Query Builder to graphically design the query.

What data should the data adapter load into the dataset?

Figure 6-7. The Query Builder page of the Data Adapter Configuration Wizard

The Query Builder page has a wealth of options, although you wouldn't think so to look at it. From here you can just key in a **Select** statement and click Next to let the wizard automatically generate the **Update**, **Delete**, and **Insert** statements. The wizard can also manage concurrency issues for you by writing these new queries in such a way that they always check the database copies of records before posting new data in. If the database record changed since it was pulled into the DataSet and edited, the Insert, Update, or Delete operation will fail.

You can turn all this off, of course, and if your SQL isn't up to writing the Select statement by hand, you can use the Query Builder I touched on back in Chapter 1 to build the statement for you. That's what you're going to do now. Click the Query Builder button to see the query builder dialog box in Figure 6-8.

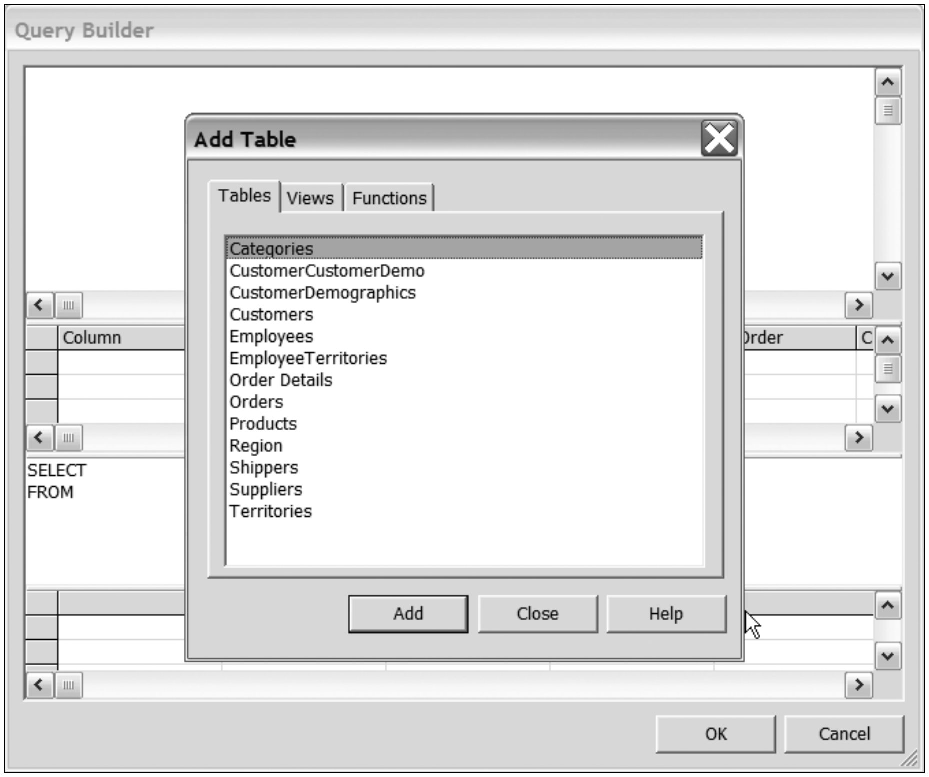
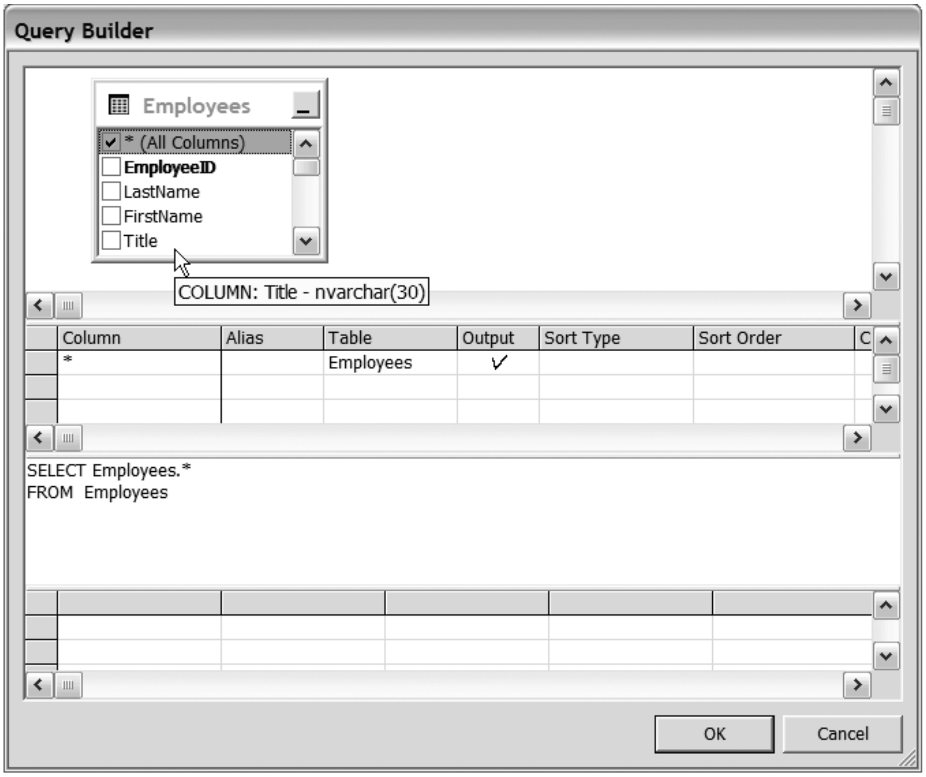


Figure 6-8. Visual Studio .NET's Query Builder is embedded in the Data Adapter Configuration Wizard.

I'm not going to go through how to use the Query Builder in detail again—just refer back to Chapter 1 if you get really lost. For now, just choose to add the Employee table into the query and then, from the list of columns that you want to pull in, select the * column to show that you want to grab all the columns. The window should look like the one shown in Figure 6-9.

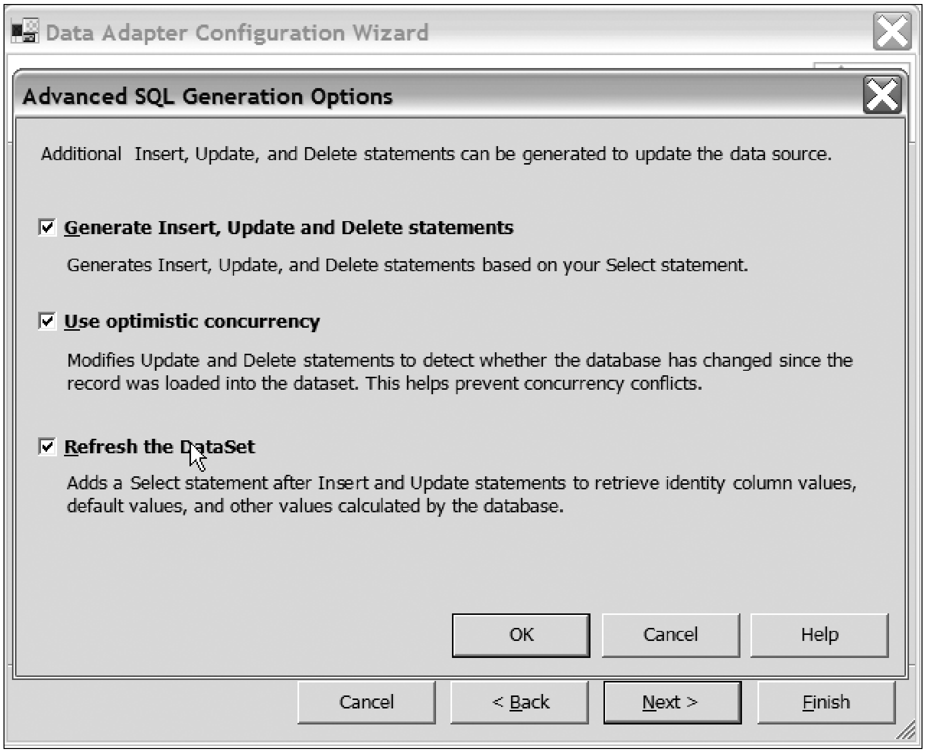


Column	Alias	Table	Output	Sort Type	Sort Order
*		Employees	✓		

```
SELECT Employees.*
FROM  Employees
```

*Figure 6-9. Select the * column to bring in every column of the Employee table in your Select statement.*

When you are done, click OK to return to the Query Builder page of the wizard, and then click Advanced Options to take a look at the other options available to you, as shown in Figure 6-10.


Data Adapter Configuration Wizard
Advanced SQL Generation Options

Additional Insert, Update, and Delete statements can be generated to update the data source.

☑ Generate Insert, Update and Delete statements

Generates Insert, Update, and Delete statements based on your Select statement.

☑ Use optimistic concurrency

Modifies Update and Delete statements to detect whether the database has changed since the record was loaded into the dataset. This helps prevent concurrency conflicts.

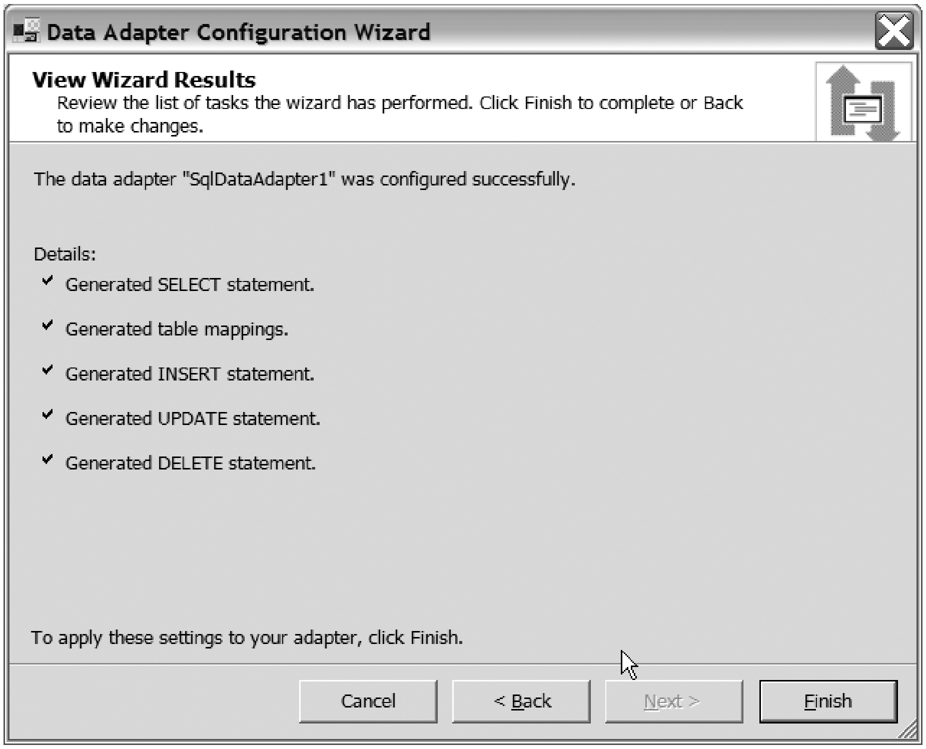
☑ Refresh the DataSet

Adds a Select statement after Insert and Update statements to retrieve identity column values, default values, and other values calculated by the database.

Figure 6-10. The Advanced button on the Query Builder page allows even more control over the creation of the DataAdapter.

The Advanced tab provides you with a set of options that are, by and large, quite self-explanatory. Unchecking the top option prevents the wizard from automatically generating SQL statements for the Update, Insert, and Delete commands inside the Adapter. The second option relates to the first. If you check this option, and if you have asked the wizard to build the other SQL statements for you, then that second option will cause the new SQL statements to be written such that they check the rows in the database before updating them. This is important, since you are going to be working with disconnected data in DataSets. It makes sure that if the rows in the database have in any way changed since you last grabbed them, any updates you asked to make are rejected. Finally, the “Refresh the DataSet” option extends the Update, Insert, and Delete commands even further to make them reselect data after it has been posted back to the database. You may be working with tables that have self-calculating columns, for example, and so leaving this option checked makes sure that a select is done to update the DataSet after posting new information back.

For now, just leave the settings all checked and click OK. When you return to the Query Builder page, click Next to view the wizard results shown in Figure 6-11.


Data Adapter Configuration Wizard

View Wizard Results
Review the list of tasks the wizard has performed. Click Finish to complete or Back to make changes.

The data adapter "SqlDataAdapter1" was configured successfully.

Details:
- ✓ Generated SELECT statement.
- ✓ Generated table mappings.
- ✓ Generated INSERT statement.
- ✓ Generated UPDATE statement.
- ✓ Generated DELETE statement.

To apply these settings to your adapter, click Finish.

Figure 6-11. The Wizard Results page

At this point you can click the Back button to walk back through the process (most commonly to change the queries) or click Finish to create the DataAdapter. Click Finish.

You should now see two new objects underneath the form in the Visual Studio Designer: a SqlDataAdapter and a SqlConnection (see Figure 6-12). These were both created by the wizard and are ready to go. It's a good idea to change the names, though—just because you built them with a wizard doesn't mean that you have to throw clarity and naming conventions out the window.

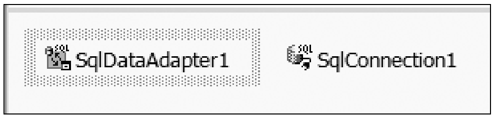


Figure 6-12. The new objects created by the Data Adapter Configuration Wizard

Click the SqlDataAdapter (just once—don't double-click it) and change its Name property in the property inspector to **employeeAdapter**. Do the same for the SqlConnection; click it and change its name in the property inspector to **northwindConnection**. Your Connection and DataAdapter are now ready for use, without you having to write a single line of code.

That whole process may have seem somewhat long-winded, but that's really only because you were reading along as you worked through the pages of the wizard. After a couple of tries with the wizard on your own, you'll find it's incredibly quick and easy to use it to get an Adapter and Connection onto your form—much easier than doing it all through code. Creating Adapters this way in particular brings another advantage with it, over and above the manual-coding approach. That advantage is the typed DataSet.

Typed DataSets

If you can take a step back for a moment and look over the DataSet code in Chapters 4 and 5 objectively, from the point of view of someone who has never ever touched .NET, it soon becomes obvious that regardless of how much power a DataSet brings to the mix, it's cumbersome to use. All that Dataset.Tables("TableName").Columns("ColumnName") hassle isn't conducive to headache-free coding.

The *typed DataSet* solves a lot of problems related to the DataSet's verbosity. With a typed DataSet you start with an XML Schema representing the data the DataSet is going to hold and use it to create a new "type." This type is then used to instantiate a class that contains all the functionality of a standard DataSet but that also adds to it by using real table and column names as properties. So, with a typed DataSet, code such as the following:

```
northwindDataSet.Tables("Employees").Rows(0).Item("LastName") = "Bloggs"
```

becomes

```
northwindDataset.Employees(0).LastName = "Bloggs"
```

I think you'll agree that the latter is far more preferable.

Building a typed DataSet manually takes a considerable amount of effort. The most time-consuming aspect is the process of producing a complete XSD schema for the tables you'll hold in the DataSet. For a complex DataSet, especially one that relies on relationships and constraints, that's not a trivial undertaking. I do cover XSD later in the book, because it is genuinely useful to know at times, but for creating a typed DataSet there's a handy tool built into the Visual Studio .NET IDE that's a lot less painful to use.

Click the DataAdapter to select it, and then right-click. A context-sensitive menu will appear that looks like the one in Figure 6-13.

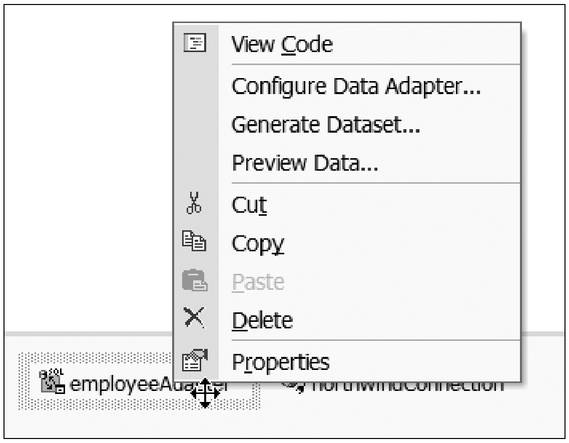
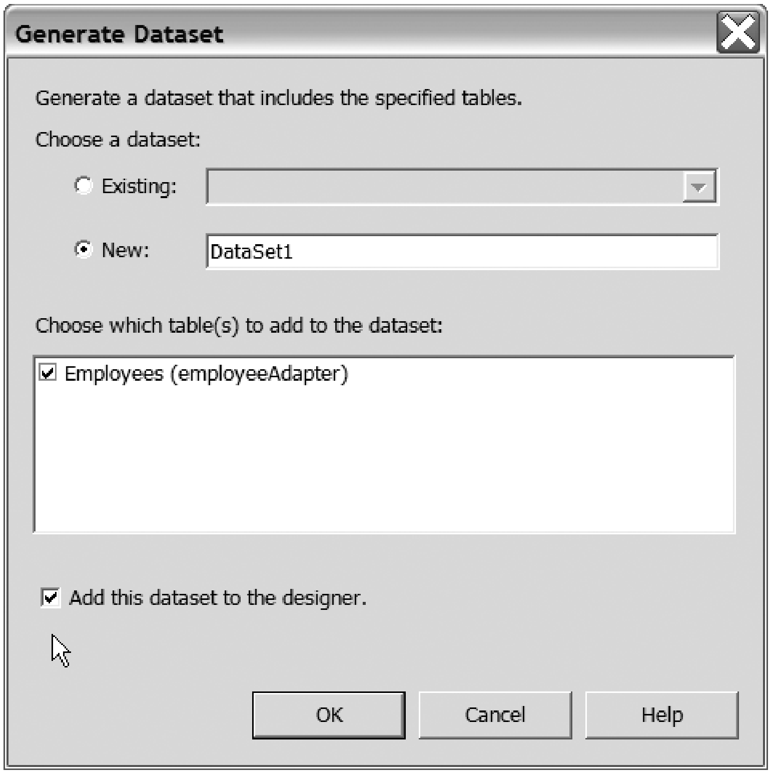


Figure 6-13. The context-sensitive menu attached to a DataAdapter

The same functionality is available through the Data menu at the top of the Visual Studio IDE. The pop-up menu lets you back into the Data Adapter Configuration Wizard through the Configure Data Adapter item and also lets you see a preview of the data that the Adapter will provide. Of most interest to you is the Generate Dataset item. Click it to bring up the Generate Dataset dialog box shown in Figure 6-14.


Generate Dataset

Generate a dataset that includes the specified tables.

Choose a dataset:

( ) Existing:

[✓ New:] DataSet1

Choose which table(s) to add to the dataset:

☑ Employees (employeeAdapter)

☑ Add this dataset to the designer.

Figure 6-14. The Generate Dataset dialog box

The Generate Dataset dialog box appears quite straightforward, but it can catch you out. First, make sure that the tables you want to use in the DataSet are checked—the wizard will use these tables to build the schema that ultimately defines the DataSet type. You only have one table in the list, the Employees table, so make sure it's checked. Also, make sure the "Add this dataset to the designer" option is checked; this will cause the new DataSet to appear in the designer alongside the Connection and Adapter.

Pay attention to the name of the DataSet next in the New text box. This can be a source of confusion. The name here is not, as you might believe, the name of the DataSet that you want to create. It is the name of the type used to create the DataSet. Think of this as naming the class that you are going to create an object from. Change the name here from DataSet1 to **TEmployeeDS**, and then click the OK button.

The new type will be created, and a DataSet created from that type will appear in the designer right alongside the Adapter and Connection objects (see Figure 6-15).

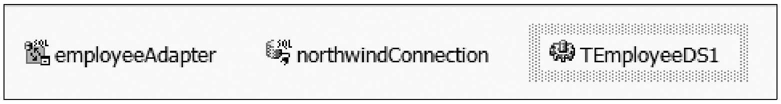


Figure 6-15. The new, typed DataSet

Notice how the name is different from the type. The type you created is called **TEmployeeDS**, but the *actual* DataSet is called **TEmployeeDS1**. Select it and change its name in the property inspector to **employeeDataset** (normally you'd name it after the database or some other useful name that indicates all the data the DataSet contains—you're just working with one table, though, hence the name "employeeDataset").

At this point you have a typed DataSet in your project ready to be used. You can get at the tables in the DataSet by name, such as (don't type this)

```
Dim myTable as DataTable = employeeDataset.employees
```

You can also get at columns in the table by name—just suffix "column" onto the end of the column name:

```
Dim myColumn = myTable.myFieldColumn
```

If you do want to fiddle with the DataSet contents using code, there are a couple of things to watch out for. With a typed DataSet, you can set the value of a column on a specific row like this:

```
MyDataset.myTableName(rownumber).columnName = "columnvalue"
```

However, if you want to get or set the value of a column on the current row, you still need to go through the Items collection:

```
MyDataSet.myTableName.Item("columnName") = "columnvalue"
```

I cover this whole area in great detail in the next chapter, when you'll start exploring every feature of data binding and developing GUI-based ADO applications.

The DataGrid

The DataGrid control is huge. It has a mass of events and scads of properties, collections, and collections within collections attached. To be quite frank, it's got enough functionality to easily fill a chapter on its own. You're going to focus in this chapter on the more common things you'll want to do with it, though, and dive in deeper in the next chapter.

You should still have your blank form with its three data objects patiently waiting underneath. The next step is to add in the DataGrid and bind it to those components. Go to the Toolbox, click Windows Forms, and double-click the DataGrid component. It will appear at a default size and location within your form. Resize it so that it takes up the bulk of the form, as shown in Figure 6-16.

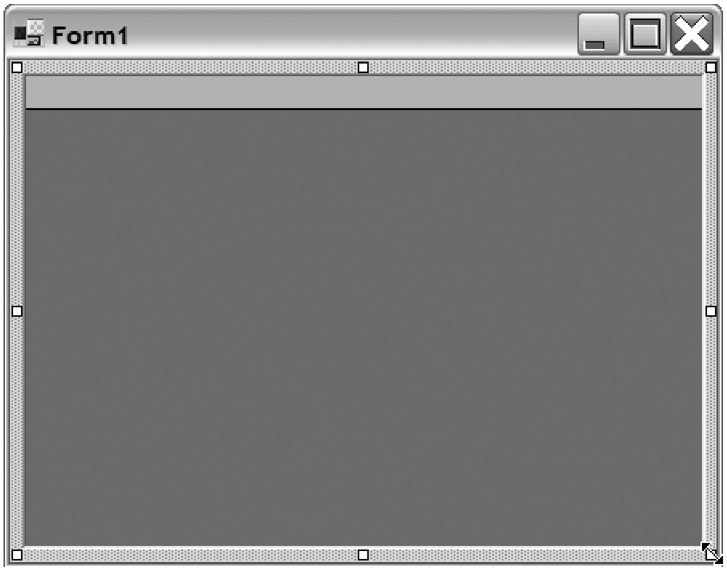


Figure 6-16. The DataGrid, unbound and naked

Binding the DataGrid is very simple: You just need to set up the **DataSource** and **DataMember** properties. DataSource is the DataSet that you want to connect to, and DataMember is the specific element that you want to bind to. Go ahead and set the DataSource property to **employeeDataset** and the DataMember property to **Employees** (see Figure 6-17). When you work with these properties in the property inspector, you'll find that you can use drop-down lists to select the elements in question. You'll also find that once you have both properties filled, the DataSet instantly changes, automatically showing a list of the columns in the Employees table. You'll see how to configure this list in a moment.

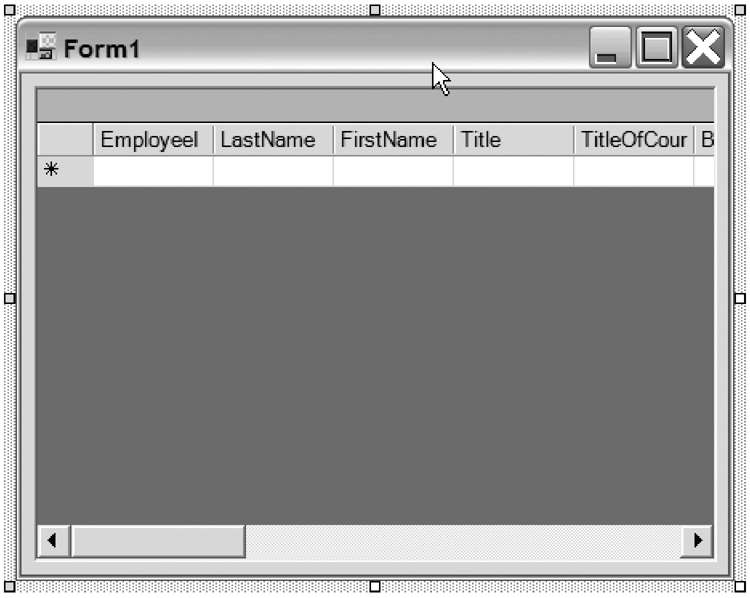


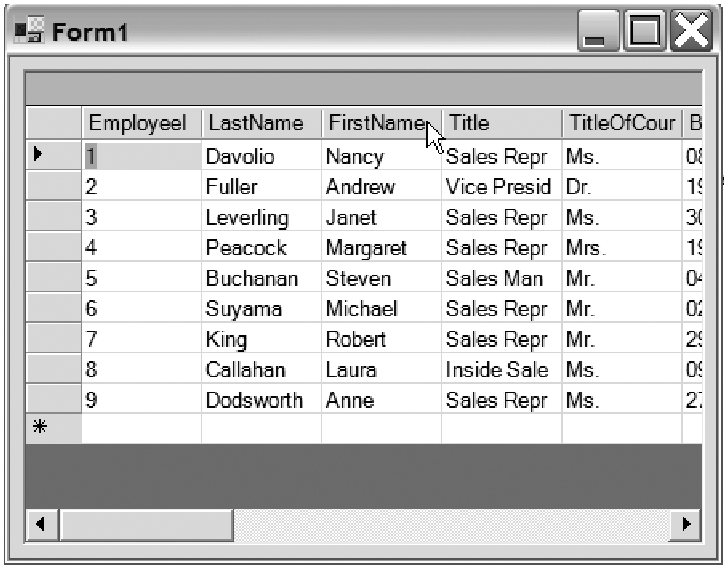
Figure 6-17. The DataGrid, bound to the Employees table in the employeeDataset DataSet

At this point you've written no code, and if you were to run the application (don't) you'd find it working perfectly, except for one small snag. The DataSet that the grid is bound to won't have been populated with any data, so when the program runs, you'll find yourself staring at an empty grid. Entering information at this point would populate the underlying DataSet, but that's hardly useful in an application that you were originally planning to use to navigate data in a database. Some code is required.

Double-click the form, not the grid, to open the code editor. You should find yourself automatically dropped into the form's Load event, which is just where you want to be. Add some code so that your form's Load event looks like this:

```
    Private Sub Form1_Load(ByVal sender As System.Object,  _
              ByVal e As System.EventArgs) Handles MyBase.Load
        employeeAdapter.Fill(employeeDataset)
End Sub
```

Don't forget to add the Imports System.Data.SqlClient line and Option Strict On right above the form's class definition if you want type safety and a nice, fast application. Run the program now and when the form loads, you'll see a fully populated DataGrid like the one shown in Figure 6-18.



	Employeel	LastName	FirstName	Title	TitleOfCour	B
▶	1	Davolio	Nancy	Sales Repr	Ms.	0
	2	Fuller	Andrew	Vice Presid	Dr.	1
	3	Leverling	Janet	Sales Repr	Ms.	3
	4	Peacock	Margaret	Sales Repr	Mrs.	1
	5	Buchanan	Steven	Sales Man	Mr.	0
	6	Suyama	Michael	Sales Repr	Mr.	0
	7	King	Robert	Sales Repr	Mr.	2
	8	Callahan	Laura	Inside Sale	Ms.	0
	9	Dodsworth	Anne	Sales Repr	Ms.	2
*						

Figure 6-18. The bound grid in full flight

You can add rows to this grid, change the data already displayed in the grid, and even delete rows by clicking to the left of the row in question and then pressing the Delete key. The changes will all be replicated in the DataSet. The changes don't make their way down to the database yet, though, since that's the job of the DataAdapter; the grid just works with disconnected information in a DataSet.

Stop the application from running and change the layout of the form by adding two buttons (name them okButton and cancelButton), so that your form looks like the one shown in Figure 6-19.

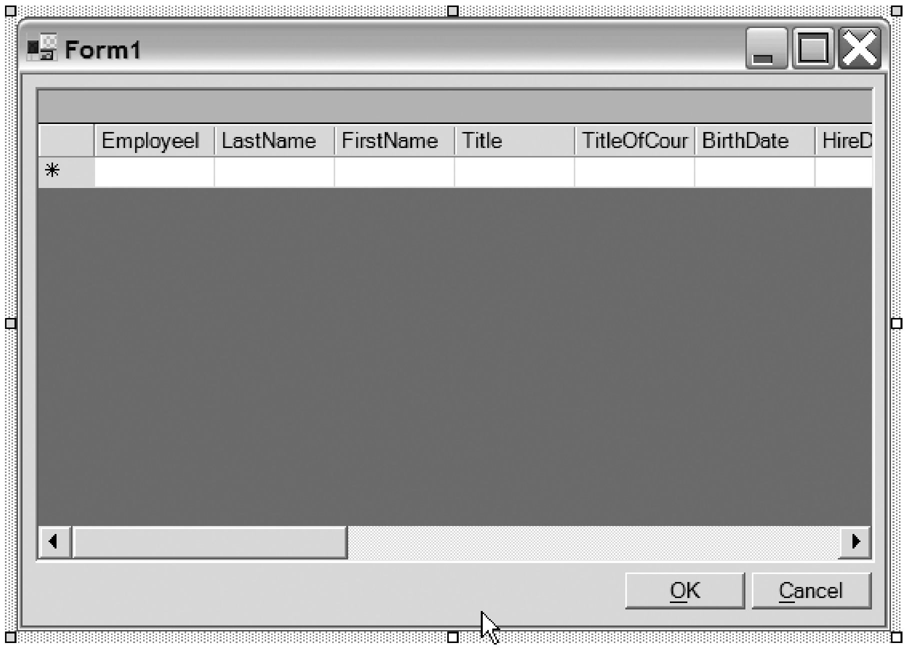


Figure 6-19. The OK and Cancel buttons will be hooked up to replicate changes to the database.

Code the Click handlers for the two buttons as follows:

```
Private Sub okButton_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles okButton.Click
    employeeAdapter.Update(employeeDataset)
End Sub

Private Sub cancelButton_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles cancelButton.Click
    employeeDataset.RejectChanges()
End Sub
```

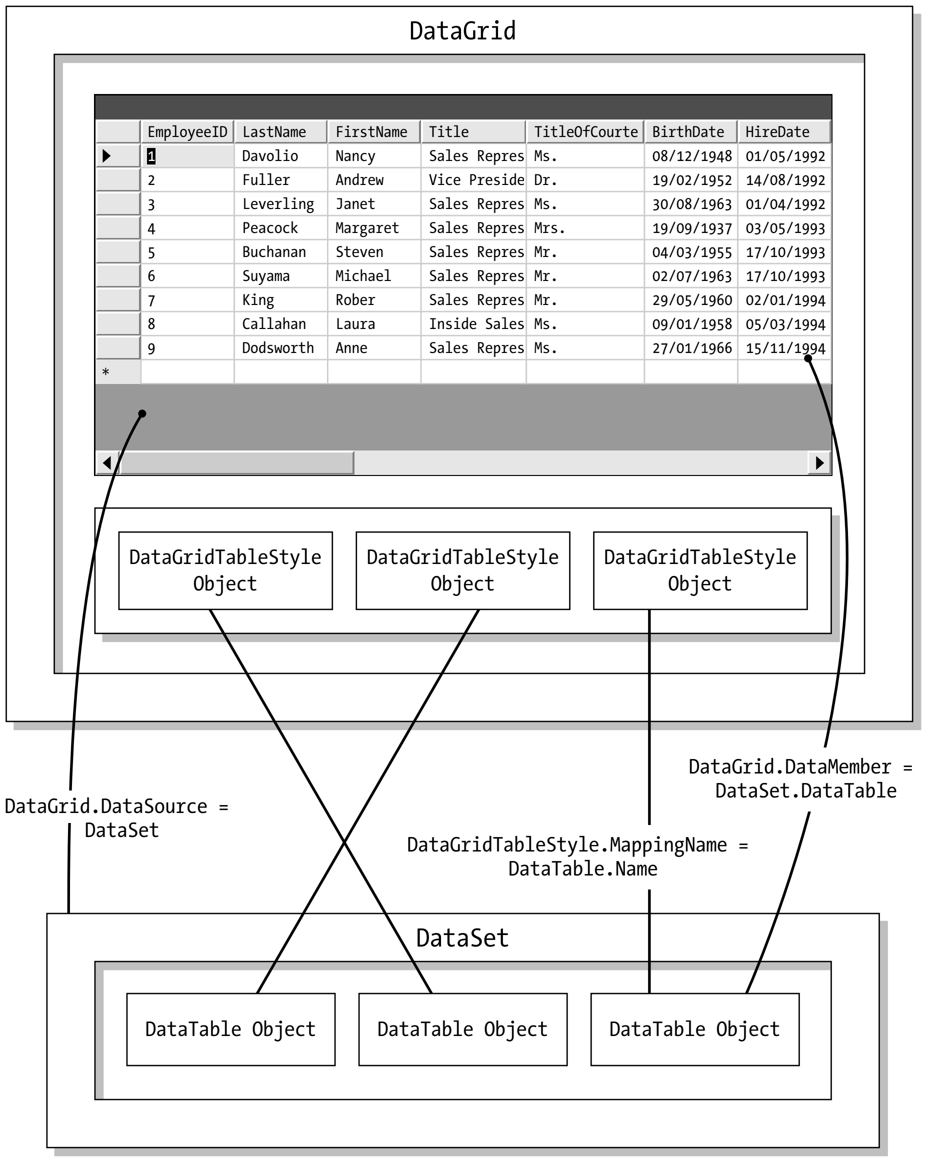
When the OK button is clicked, you pass the DataSet to employeeAdapter.Update() to accept all the changes and save them to the database. Alternately, when the Cancel button is clicked, you just call **RejectChanges()** on the DataSet to put the DataSet back in the condition it was in before any changes took place. Run the application and have a play. In particular, try deleting a bunch of rows and click Cancel. Then, try adding in a new employee and click OK—pay attention to just how rapid that update is.

Customizing the DataGrid

The visual appearance, including the layout, of data in a DataGrid is controlled by a **DataGridTableStyle** object. As its name implies, the DataGridTableStyle object defines the style to apply for a specific table displayed in a DataGrid. The DataGrid actually contains a collection of these, which you can get at through the **TableStyles** collection property, and relates each one to a table of data through the **DataGridTableStyle.MappingName** property. The reason the grid stores a collection is that a DataGrid is quite capable of displaying data from multiple tables at the same time. When a new table is shown, the grid searches its Table-Styles collection for a DataGridTableStyle object and then uses its properties to define such things as the color to use for the grid lines, the font to use for the headings, and so on.

As if that weren't enough, each DataGridTableStyle object holds a collection of **DataGridColumnStyle** objects. These objects control which columns from the DataSource to show in the grid and how those grid columns should be formatted. You can see the hierarchy of the objects in Figure 6-20.

When you set the DataMember property of the DataGrid earlier, the DataGrid searched through its DataGridTableStyle objects looking for one with the same name as the table set in DataMember. When it failed to find one, it automatically mapped that table name to a default internal style that renders every field in a table a default size. You cannot customize this default style, so it's always a good idea to create your own TableStyle objects.


DataGrid
DataGridTableStyle Object
DataGridTableStyle Object
DataGridTableStyle Object
DataGrid.DataSource = DataSet
DataGrid.DataMember = DataSet.DataTable
DataGridTableStyle.MappingName = DataTable.Name
DataSet
DataTable Object
DataTable Object
DataTable Object

	EmployeeID	LastName	FirstName	Title	TitleOfCourte	BirthDate	HireDate
▶	1	Davolio	Nancy	Sales Repres	Ms.	08/12/1948	01/05/1992
	2	Fuller	Andrew	Vice Preside	Dr.	19/02/1952	14/08/1992
	3	Leverling	Janet	Sales Repres	Ms.	30/08/1963	01/04/1992
	4	Peacock	Margaret	Sales Repres	Mrs.	19/09/1937	03/05/1993
	5	Buchanan	Steven	Sales Repres	Mr.	04/03/1955	17/10/1993
	6	Suyama	Michael	Sales Repres	Mr.	02/07/1963	17/10/1993
	7	King	Rober	Sales Repres	Mr.	29/05/1960	02/01/1994
	8	Callahan	Laura	Inside Sales	Ms.	09/01/1958	05/03/1994
	9	Dodsworth	Anne	Sales Repres	Ms.	27/01/1966	15/11/1994
*							

Figure 6-20. The links between the DataGrid and the information that it displays

The first step is to clear out the DataMember property of the DataGrid. Just select the text currently in the property and press the Backspace key. The grid will instantly revert back to looking the way it did when you first dropped it onto the form—with no columns defined and no data on display. Also, set the AllowNavigation property to False—I'll cover what this does later.

Still in the property window, find and click in the TableStyles property. You'll notice that within the property inspector the property shows a button containing ellipses (…) to the right of the entry area. Click this button and you'll be taken to the grid styles editor, as shown in Figure 6-21.

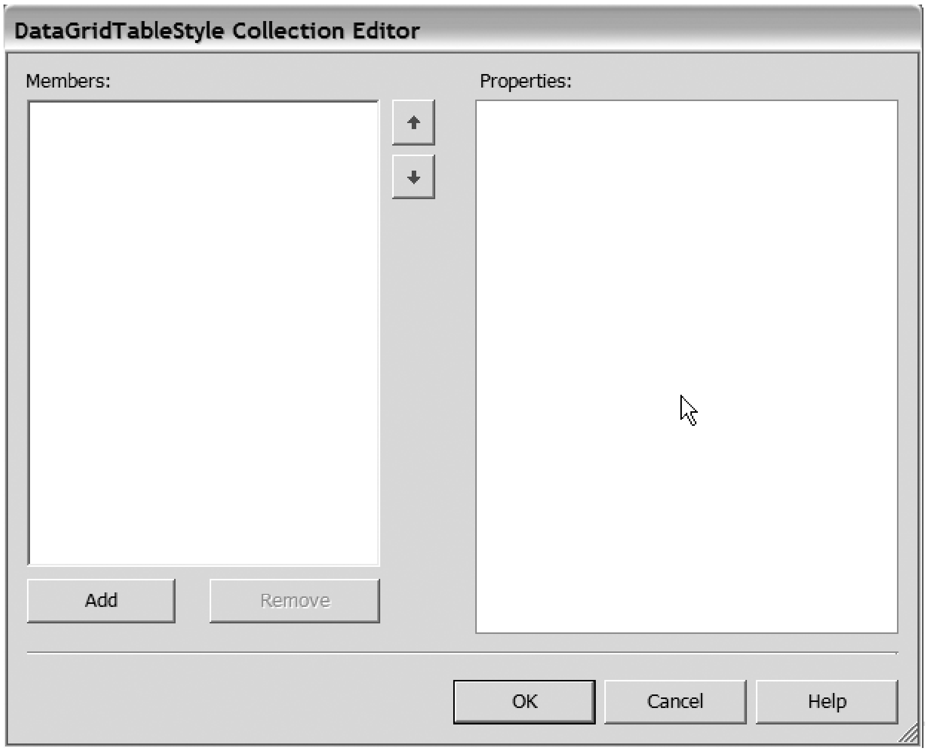
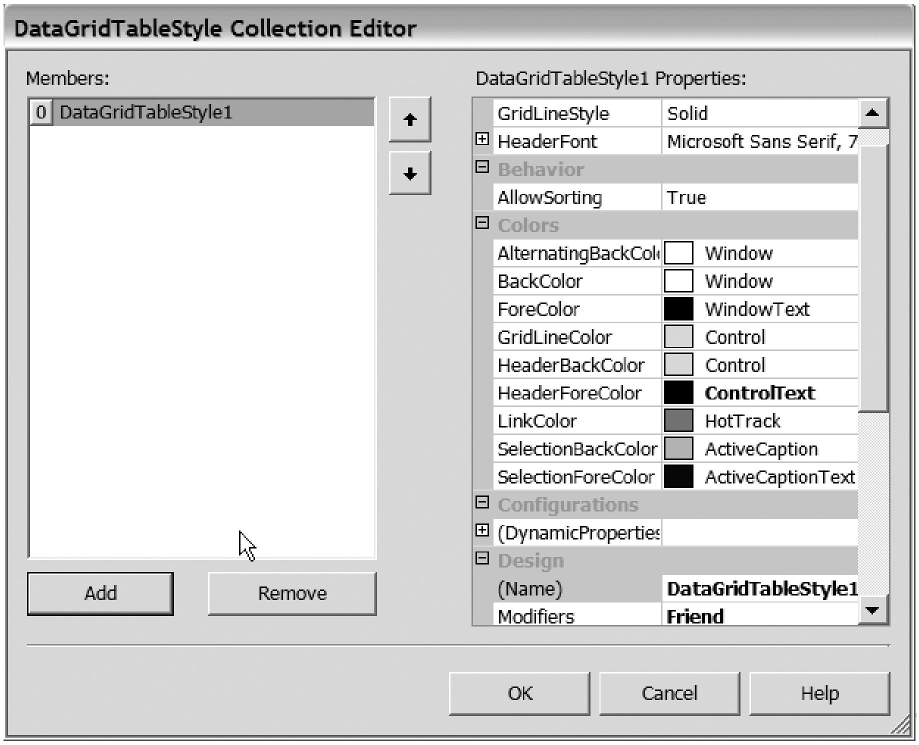


Figure 6-21. The grid styles editor allows you to add in new DataGridTableStyle objects and link them to specific table names.

When the editor appears, click the Add button at its bottom left. A new DataGridTableStyle object will be created and shown in the list on the left, while the right-hand side of the editor dialog box will change to show you a complete list of properties for that object, as shown in Figure 6-22.



Members:

0 DataGridTableStyle1

DataGridTableStyle1 Properties:

GridLineStyle	Solid
⊞ HeaderFont	Microsoft Sans Serif, 7
⊟ Behavior	
AllowSorting	True
⊟ Colors	
AlternatingBackCol	☐ Window
BackColor	☐ Window
ForeColor	■ WindowText
GridLineColor	☐ Control
HeaderBackColor	☐ Control
HeaderForeColor	■ **ControlText**
LinkColor	■ HotTrack
SelectionBackColor	■ ActiveCaption
SelectionForeColor	■ ActiveCaptionText
⊟ Configurations	
⊞ (DynamicProperties	
⊟ Design	
(Name)	**DataGridTableStyle1**
Modifiers	**Friend**

Figure 6-22. Adding a new DataGridTableStyle object causes the editor to show you all of that object's properties and allows you to edit them.

You now need to relate your new DataTableStyle object to a table. To achieve this, all you need to do is set the MappingName property to the name of the table. Click this property, drop down the list of tables, and select the Employees Table.

When you've done that, click OK to close the DataGridTableStyle Collection Editor. You have a table style defined now for the Employees table, so go ahead, click the grid and use the property explorer to set the DataMember property back to Employees.

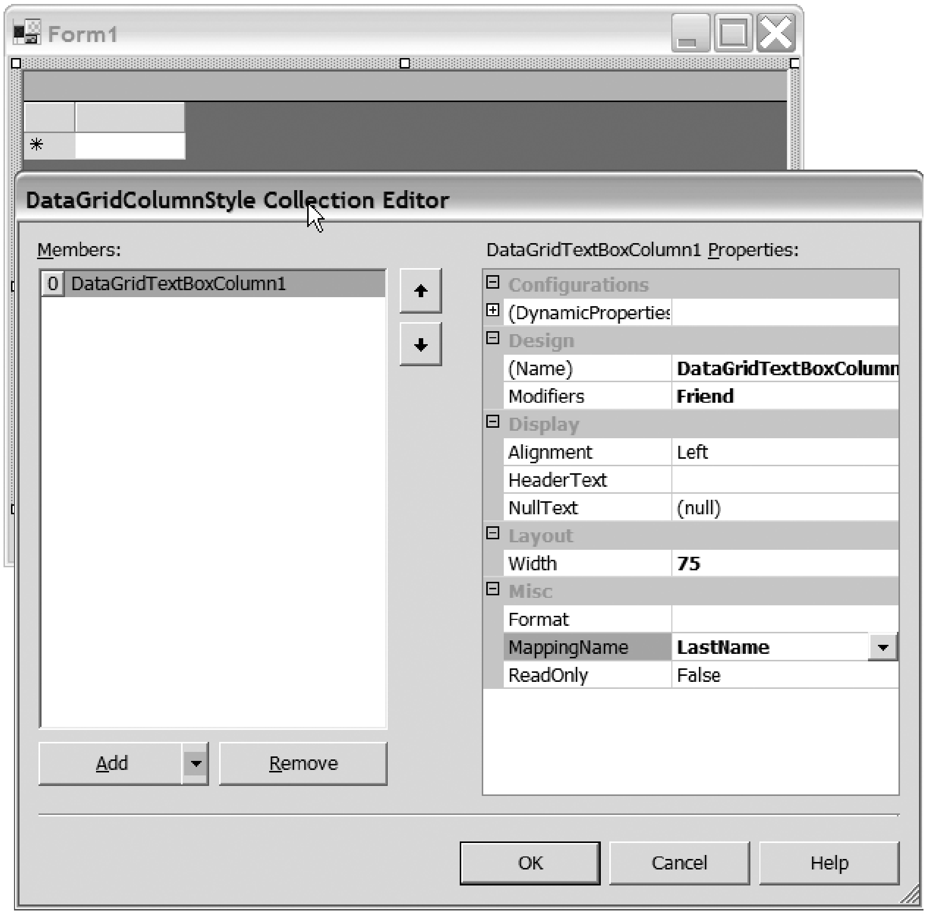
You won't see any change on the grid when you do this, but it's a very important step. Since you've linked a table to the grid that has a matching TableStyle, the grid will start to use that new TableStyle. Any changes you make to it, including adding and deleting columns in the TableStyle, will be instantly shown in the grid. That's exactly what you're going to do now. Let me just recap the steps you've taken so far. To make the grid use your own TableStyle instead of its default, you need to do the following:

- Clear out the DataMember property of the grid.
- Open up the TableStyles Collection Editor by clicking the ellipses in the TableStyles property in the property explorer.
- Add a new TableStyle and use its MappingName property to map it to the table you are going to view.
- Close the editor down and set the DataMember property to the same table you set for the new TableStyle.

Bring up the TableStyles Collection Editor once again. When it appears, click the **GridColumnStyles** property within the editor and click the ellipses (…) button that appears in that property. This will pop open the DataGridColumnStyles editor. It should look almost exactly the same as the TableStyles Collection Editor the first time you opened it. It works in a very similar way as well. It's this editor that you use to add columns to the TableStyle object in order to let the grid know exactly which columns you want it to display and the formatting to use on those columns.

Go ahead and click the Add button to add a new column. Then, click in the MappingName property for that column and select LastName as the column you want to display, as shown in Figure 6-23.

As soon as you set the MappingName property of the column, you should see the new column appear in the grid. Set up the Header Text property to hold the column's caption. You may also want to change the Width property to make the column wider (a value of 130 works well).



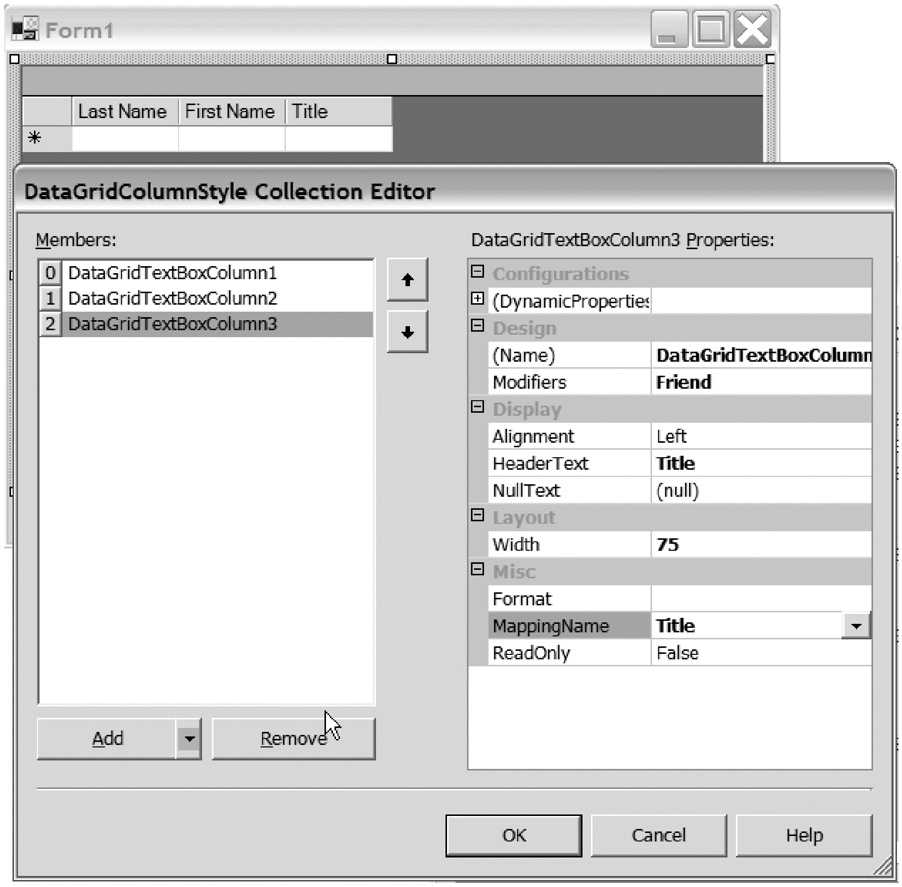
0 DataGridTextBoxColumn1

DataGridTextBoxColumn1 Properties:

Configurations	
(DynamicProperties	
Design	
(Name)	DataGridTextBoxColumn
Modifiers	Friend
Display	
Alignment	Left
HeaderText	
NullText	(null)
Layout	
Width	75
Misc	
Format	
MappingName	LastName
ReadOnly	False

Figure 6-23. The column editor lets you add columns to a TableStyle object and map it to a specific column in the data source.

Go ahead and add in the FirstName and Title columns, as displayed in Figure 6-24.



Members:

0 DataGridTextBoxColumn1
1 DataGridTextBoxColumn2
2 DataGridTextBoxColumn3

DataGridTextBoxColumn3 Properties:

Configurations	
(DynamicProperties	
Design	
(Name)	DataGridTextBoxColumn
Modifiers	Friend
Display	
Alignment	Left
HeaderText	Title
NullText	(null)
Layout	
Width	75
Misc	
Format	
MappingName	Title
ReadOnly	False

Figure 6-24. *The DataGrid is set up to display just the information you need.*

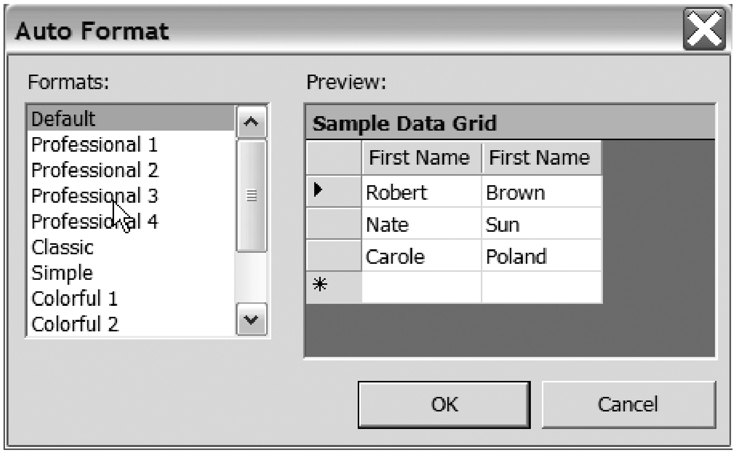
You're done—your grid is customized and complete. Before you finish here, though, there's another neat feature of the column editor that you should know about.

You are not limited to just adding textual columns to the grid. Notice the small drop-down arrow to the right of the Add button. If you click this arrow, you'll find that you can add either a text column or a DataGridBoolColumn. This is really just a check box, but it's useful for linking to "Bit" fields.

Also, when you added the columns to the grid, they were automatically named by the editor. The Name property is exposed within the editor, allowing you to easily assign a name that you can use in code later if you want to take control of the formatting of the grid at runtime. The same applies to the TableStyle object as well.

Reformatting the Grid

The DataGrid is pretty powerful straight out of the box, but it's also pretty plain and dull to look at. That's easily changed, and it doesn't require a great deal of clicking or typing. If you still have the application you just worked on loaded (if not, load it), right-click the grid. When the pop-up menu appears, click AutoFormat and you'll see the Auto Format dialog box appear (see Figure 6-25).

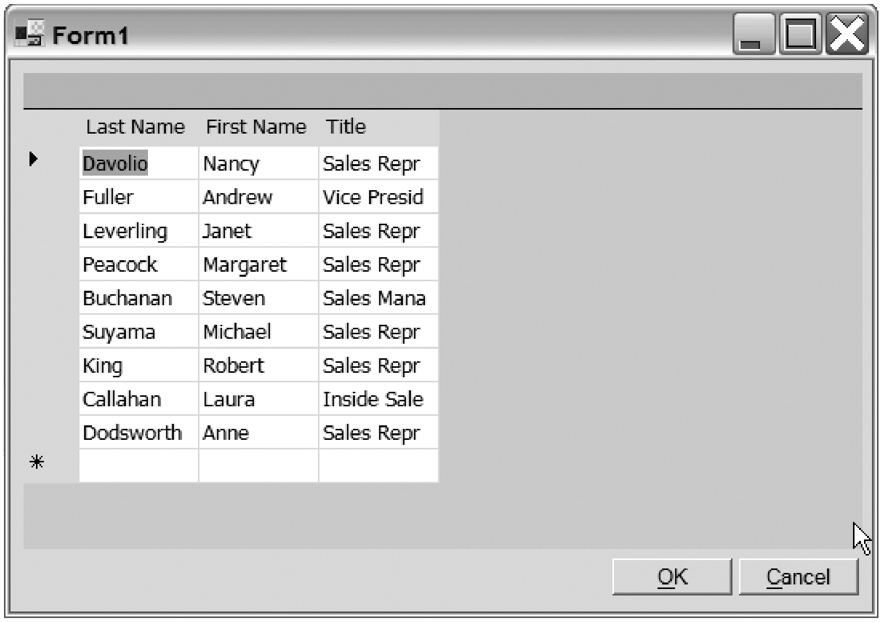


	First Name	First Name
▶	Robert	Brown
	Nate	Sun
	Carole	Poland
*		

Figure 6-25. The Auto Format dialog box

If you've ever used tables in Word or Excel, you'll recognize this dialog box instantly. The list box on the left shows a list of prebuilt visual styles that can be applied to the grid, while the right-hand side shows a preview of what the selected style looks like. Take a look at each style, and when you find one you're happy with, select it and click the OK button.

The grid on your form will instantly adopt a (hopefully) more pleasing visual style. See Figure 6-26 for an example.



Last Name	First Name	Title
Davolio	Nancy	Sales Repr
Fuller	Andrew	Vice Presid
Leverling	Janet	Sales Repr
Peacock	Margaret	Sales Repr
Buchanan	Steven	Sales Mana
Suyama	Michael	Sales Repr
King	Robert	Sales Repr
Callahan	Laura	Inside Sale
Dodsworth	Anne	Sales Repr

Figure 6-26. The grid with a new format applied

A Complex Complex-Binding Example

The last example was pretty simple: Just connect to a data source and display a single result set from that data source. What about something more complex? What about the master-detail problem, where selecting a row from one result set shows related rows from another result set?

The key to solving the problem lies within the XML Schema that's produced for a typed DataSet. You can easily link two tables in a typed DataSet through the schema and have Visual Studio automatically build the data relation between those two tables. Bound controls can monitor row-change events and refresh their contents when a master row changes.

Setting a Master-Child Relation

Start up a new project. Just as in the last example, select the VB .NET Windows Application template as the foundation for your new application, and call the project Master-Detail.

The first step, as in the last example, is to set up the DataAdapters you'll use to grab information. Go ahead and set up two DataAdapters called employeeAdapter and regionAdapter. For employeeAdapter, set the SQL for the select command (within the Data Adapter Configuration Wizard) to the following:

```
SELECT DISTINCT
    Employees.LastName,
    Employees.FirstName,
    Employees.Title,
    Employees.EmployeeID,
    Region.RegionID,
    Region.RegionDescription
FROM
    Employees INNER JOIN
    EmployeeTerritories ON
        Employees.EmployeeID = EmployeeTerritories.EmployeeID
           INNER JOIN Territories ON
             EmployeeTerritories.TerritoryID = Territories.TerritoryID
                  INNER JOIN Region ON
                      Territories.RegionID = Region.RegionID
```

And for the region Adapter, set the SQL for the select command to this:

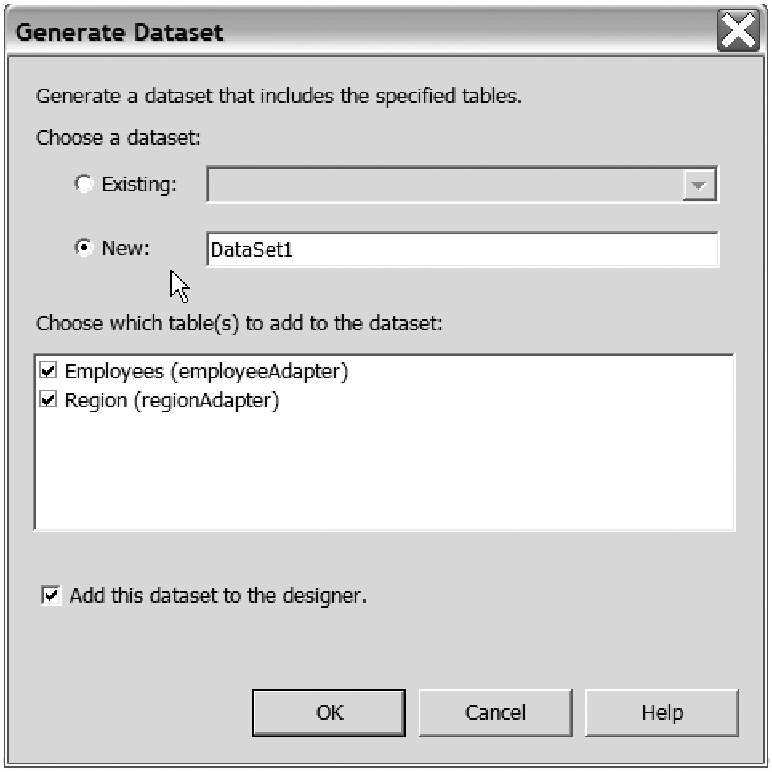
```
SELECT Region.* from Region
```

The joins on the employeeAdapter's Select statement will prevent the wizard from figuring out how to write the Update, Insert, and Delete statements for the Adapter. As a result, the Results page of the wizard will contain a set of warning messages. This is quite normal, and since this application doesn't need to update the database, you can ignore the messages. In a real application, you'd respond to this by writing your own commands and dropping them into the Adapter, as you saw how to do in earlier chapters.

Before you go any further, you might want to rename the Connection object, just for completeness, to **northwindConnection**.

The next step is to build your typed DataSet. Unlike the typed DataSet in the last example, this one will include two tables, so select the Data item from the menu bar at the top of Visual Studio .NET and then choose Generate Dataset.

There are two tables listed in the Generate Dataset dialog box this time. Make sure your dialog box looks like the one in Figure 6-27, set the name to **TemployeeRegionDS**, and click OK. As you saw earlier, this generates the type and then creates a DataSet from that type within the Visual Studio Designer. Change the name of the one in the designer to **northwindDS**.


Generate Dataset

Generate a dataset that includes the specified tables.

Choose a dataset:

( ) Existing:

(•) New: DataSet1

Choose which table(s) to add to the dataset:

☑ Employees (employeeAdapter)
☑ Region (regionAdapter)

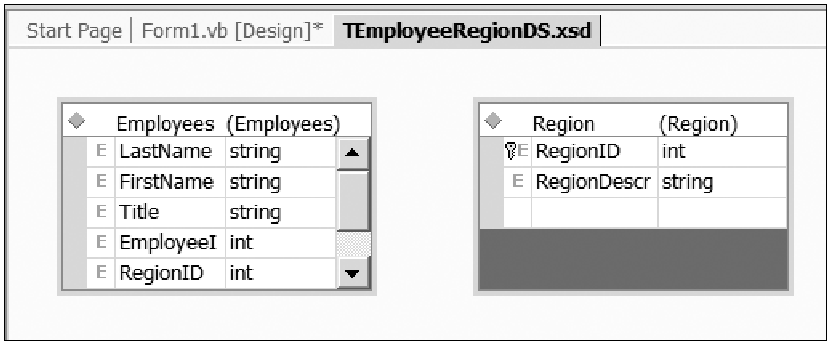
☑ Add this dataset to the designer.

Figure 6-27. Because you have two Adapters pulling data from two data sources, the Generate Dataset dialog box shows two tables.

When you create a typed DataSet like this, an XML Schema Definition (XSD) file is added to the project, which defines the columns, tables, and so forth exposed from the DataSet. If you have a DataSet with more than one table in it, as you do now, you can also use this schema to define relationships. Look in the Solution Explorer and you'll see a file called TemployeeRegionDS.xsd. Double-click it to open the schema editor (see Figure 6-28).

It looks just like the database diagram tool, doesn't it? It works in a similar way. What you want to do here is draw a link showing that many employees can service a region.

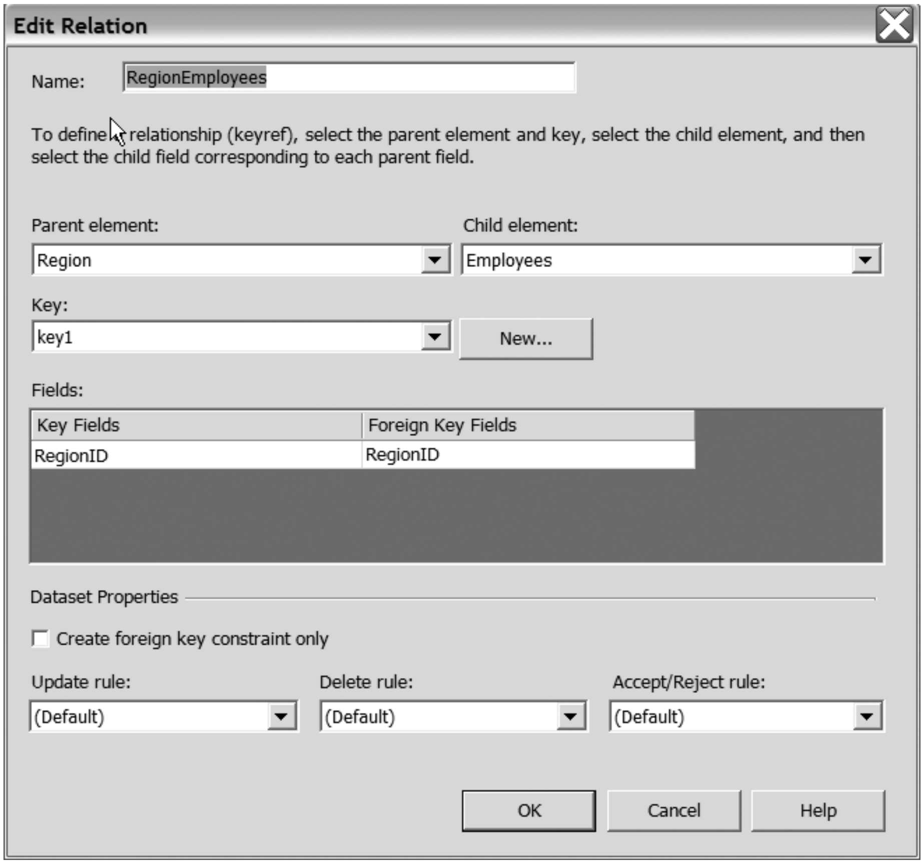
Click the RegionID field in the Region table to select that field, and then click and drag the arrow to the left of the column across to the RegionID column on the Employees table. This will pop open a relationship editor dialog box (see Figure 6-29).



Employees	(Employees)
E LastName	string
E FirstName	string
E Title	string
E EmployeeI	int
E RegionID	int

Region	(Region)
E RegionID	int
E RegionDescr	string

Figure 6-28. Double-clicking a schema file in Visual Studio .NET opens the schema editor.



Name: RegionEmployees

To define relationship (keyref), select the parent element and key, select the child element, and then select the child field corresponding to each parent field.

Parent element: Region

Child element: Employees

Key: key1

Fields:

Key Fields	Foreign Key Fields
RegionID	RegionID

Dataset Properties

☐ Create foreign key constraint only

Update rule: (Default)

Delete rule: (Default)

Accept/Reject rule: (Default)

Figure 6-29. The relationship editor appears when you drag a relationship between two columns.

The relationship editor actually serves two purposes. If you leave the “Create foreign key constraint only” check box unchecked, you are actually creating a relationship between two tables. If that box is checked, then the editor will create just the constraints, not the relationship. Leave that box unchecked in this case and click OK. When the schema designer returns to view, you’ll see the relationship drawn on the diagram. Time now to draw the user interface.

Drop back into the form designer (click the Form1.vb tab at the top of the designer). There is a chance that the relationship won’t always return data in part because of the big join on the employee select. Since you aren’t updating the database in this application, click the DataSet and set its Enforce Constraints property to False. Now drop a list box and a DataGrid onto the form, just like in Figure 6-30.

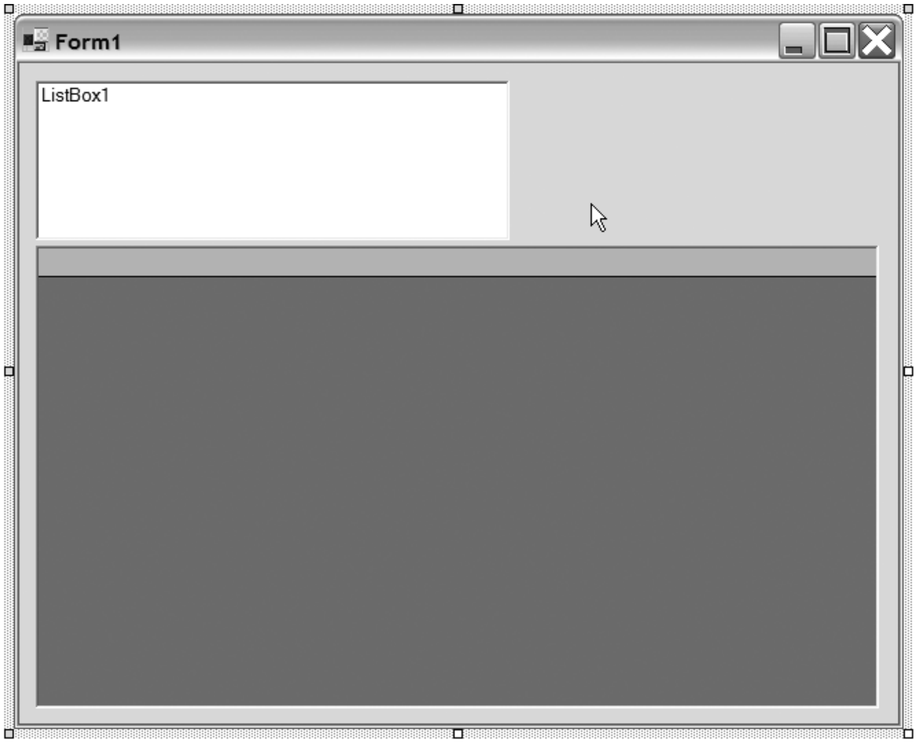


Figure 6-30. Add a list box and DataGrid to your master-detail form.

You're just going to hook these up to the relevant tables now, but you won't spend any time defining which columns to appear in the grid—you can do that yourself later as an exercise.

First up, let's set up the list box. It has a DataSource property, just like the DataGrid, and a DisplayMember property for binding to a specific column to list.

Set the DataSource to the northwindDS DataSet, and set the DisplayMember to **Region.RegionDescription**.

Next, the DataGrid. Because you're working in a master-detail form here, setting up the grid is subtly different from the previous example. Specifically, the DataMember property needs to point at the relationship you defined and not a table. In this way, the grid can track row-change events through the relationship and refresh its contents accordingly.

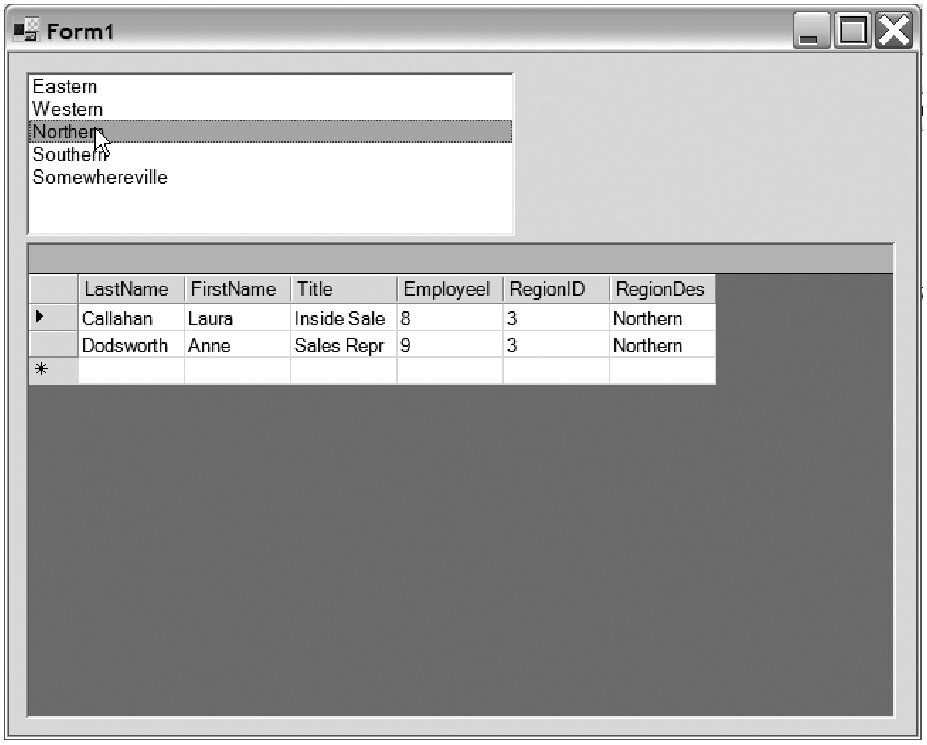
Set up the DataGrid's DataSource property to the northwindDS DataSet, and then for the DataMember, select Region.RegionEmployees—the relationship you built. Just as before, the grid will change to show all the columns linked through that relationship. (Note: If you didn't save the changes you made to the XSD file earlier, the relationship won't appear as a choice for the DataMember. If that's the case for you, just click the Save All button on Visual Studio's toolbar and try again.)

All that remains is to populate the DataSet. Double-click in the form to drop into the code editor and change the Form_Load event so that it reads like this:

```
    Private Sub Form1_Load(ByVal sender As System.Object, _
             ByVal e As System.EventArgs) Handles MyBase.Load
        employeeAdapter.Fill(northwindDS)
        regionAdapter.Fill(northwindDS)
 End Sub
```

Once again, don't forget to add Option Strict On to the top of the code.

You're all set, so go ahead and run the application. The top list shows the regions, and the grid shows the employees working within that region. If you click through the different regions in the list box, the grid will automatically update to show the relevant employees, as in Figure 6-31.



	LastName	FirstName	Title	Employeel	RegionID	RegionDes
▸	Callahan	Laura	Inside Sale	8	3	Northern
	Dodsworth	Anne	Sales Repr	9	3	Northern
*						

Figure 6-31. The finished master-detail application

As you've seen in this example, the DataGrid is not the only control that supports complex binding; the list box does too. Combo boxes and the checked combo box work in exactly the same fashion.

Go ahead and drop a combo box to the right of the list box. Set its DataSource to the DataSet and its DisplayMember to Region.RegionID. If you run the application now, not only does selecting a region from the list box update the grid, it also updates the text in the combo box. Similarly, selecting a RegionID from the combo box updates the selected item in the list box and also refreshes the grid.

DataGrid Navigation

Master-detail forms like the one in the previous example are very common in older-style applications. The DataGrid does provide a way to let you work with related information across tables in a DataSet without having to go to all that effort, though.

Fire up a new project in Visual Studio .NET, again making sure you choose the VB .NET Windows Application template. Call this project Navigation.

Add in three SqlDataAdapters to the form. Call them customersAdapter, OrdersAdapter, and OrderDetailsAdapter. These three adapters should all talk to the Northwind database. The Select statements for each, in order, are as follows:

```
Select * from Customers
Select * from Orders
Select * from [Order Details]
```

Name the Connection northwindConnection, just for completeness.

Now use the Data menu in Visual Studio to generate a typed DataSet. Make sure that you select all three tables to be added to the DataSet when asked. I called the type TNorthwindDataset and the DataSet itself I named northwindDataset.

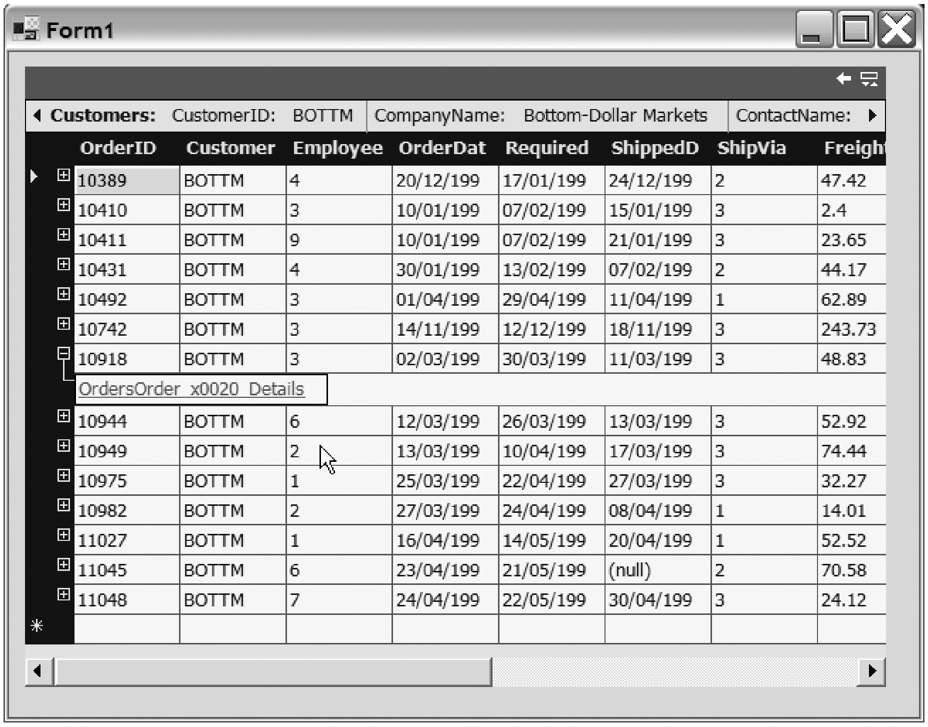
Just as before, you need to define the relationships in place here, so double-click the XSD file in the Solution Explorer to fire up the schema editor. When the schema editor pops up, set up relationships between the Customers.CustomerID field and the Orders.CustomerID field (the Customers table is the parent in this relationship), and between the Orders.OrderID field and the OrderDetails.OrderID field (the Orders table is the parent in this relationship).

When you're done, click Save, return to the form designer, and add a DataGrid to the form. Set the DataSource to northwindDataset, and set the DataMember to the Customers table. Feel free to also use the AutoFormatter to change the look of the grid to something more aesthetically pleasing.

Finally, build the DataSet by coding the Form_Load event as follows:

```
    Private Sub Form1_Load(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles MyBase.Load
        customersAdapter.Fill(northwindDataset)
        ordersAdapter.Fill(northwindDataset)
        orderDetailsAdapter.Fill(northwindDataset)
    End Sub
```

When you're done, run the application. You should see a window like the one shown in Figure 6-32. Each customer will have a small plus sign (+) next to them. You can click this plus sign to view the orders of a customer. Similarly, you can expand each order to show the order details attached to that customer.



Customers: CustomerID: BOTTM CompanyName: Bottom-Dollar Markets ContactName:

OrderID	Customer	Employee	OrderDat	Required	ShippedD	ShipVia	Freigh
10389	BOTTM	4	20/12/199	17/01/199	24/12/199	2	47.42
10410	BOTTM	3	10/01/199	07/02/199	15/01/199	3	2.4
10411	BOTTM	9	10/01/199	07/02/199	21/01/199	3	23.65
10431	BOTTM	4	30/01/199	13/02/199	07/02/199	2	44.17
10492	BOTTM	3	01/04/199	29/04/199	11/04/199	1	62.89
10742	BOTTM	3	14/11/199	12/12/199	18/11/199	3	243.73
10918	BOTTM	3	02/03/199	30/03/199	11/03/199	3	48.83
OrdersOrder_x0020_Details							
10944	BOTTM	6	12/03/199	26/03/199	13/03/199	3	52.92
10949	BOTTM	2	13/03/199	10/04/199	17/03/199	3	74.44
10975	BOTTM	1	25/03/199	22/04/199	27/03/199	3	32.27
10982	BOTTM	2	27/03/199	24/04/199	08/04/199	1	14.01
11027	BOTTM	1	16/04/199	14/05/199	20/04/199	1	52.52
11045	BOTTM	6	23/04/199	21/05/199	(null)	2	70.58
11048	BOTTM	7	24/04/199	22/05/199	30/04/199	3	24.12

Figure 6-32. The navigable grid

The grid is able to automatically do this by examining the relations in the DataSet and allowing you to navigate through them at runtime. In fact, you don't even need to set up the DataMember property of the grid at all. If you leave it blank, when you run the application you'll need to click the plus sign and choose which table you wish to start exploring.

Simple Binding Other Controls

I mentioned earlier how any control can be bound to a column in a data source. Let's take a look at how.

Start up a new project (called SimpleBinding) and design a form like the one shown in Figure 6-33.

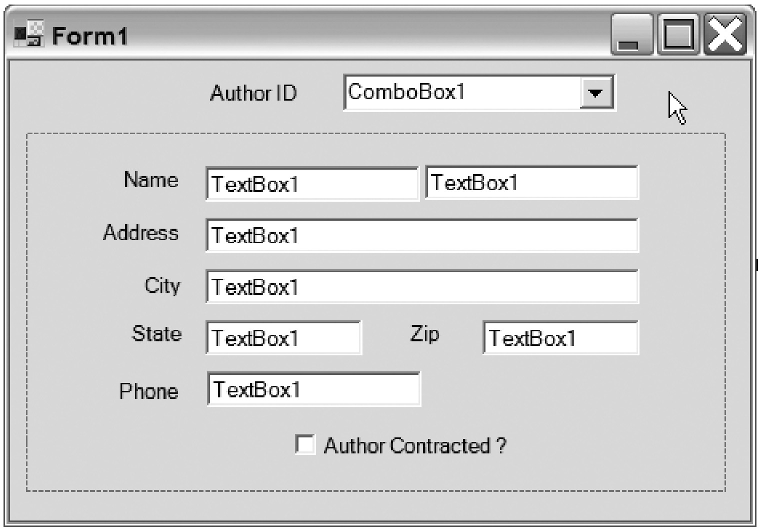


Figure 6-33. A simple data-entry/browsing form

This is quite a straightforward form. The user selects an Author ID from the combo box at the top of the form and the text boxes and check box instantly change to show the complete author information.

Once you've built your form, you'll need to add in a **SqlDataAdapter** and **SqlConnection**, and generate a typed DataSet. I'll let you work through that on your own—just make sure that the Adapter talks to the Pubs database and the Select statement for the Adapter is

```
SELECT au_id, au_lname, au_fname, phone, address, city, state, zip, contract
     FROM authors
```

For naming, I called the DataSet type TpubsDataset, and the DataSet itself pubsDataset.

Once you have the Data objects added to your form, simple binding them is quite straightforward. First, the combo box (which isn't strictly a simple bound control).

Just as with the other controls you've looked at, the combo box has **DataSource** and **DisplayMember** properties. Set the DataSource to **pubsDataset** (the DataSet you added to the form), and set the display member to **authors.au_id**.

Now, click the first text box on your form and take a look at the properties window. In the Data section you'll find a DataBindings option. Expand it to see what's underneath and your property explorer will look like the one in Figure 6-34.

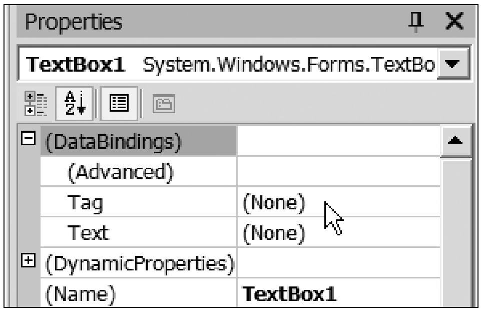


Figure 6-34. The DataBindings properties for the text box

Every visual control in the Toolbox has DataBindings collection. Expand the collection in the property window and you'll see a list of the most common properties that people want to bind to that particular control. In the case of a text box, most developers are interested in binding the Tag and Text properties. Click in the Text property, click the drop-down button that appears, and you'll see a list of data tables available to you, as shown in Figure 6-35. Go ahead and choose the fname field from the authors table.

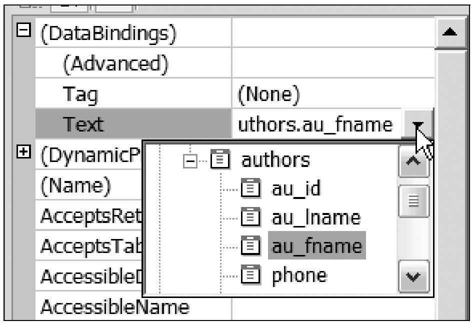
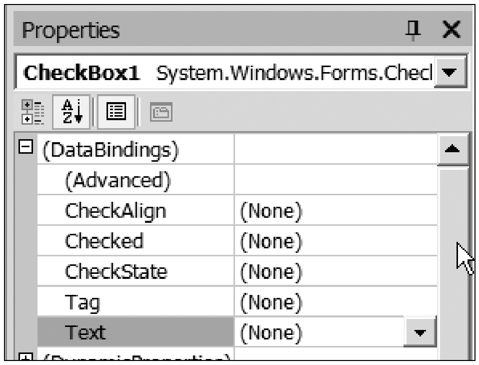


Figure 6-35. Dropping down the Text property in the DataBindings section of a text box allows you to browse and choose which field you want to bind to the text box's Text property.

You can set up the bindings on the rest of the text boxes on the form in exactly the same way, but obviously make sure you choose different fields for each text box or your form will be pretty useless at runtime.

When you're done, take a look at the DataBindings available for the check box at the bottom of the form (you can see them in Figure 6-36).



(DataBindings)	
(Advanced)	
CheckAlign	(None)
Checked	(None)
CheckState	(None)
Tag	(None)
Text	(None)

Figure 6-36. The DataBindings of a check box

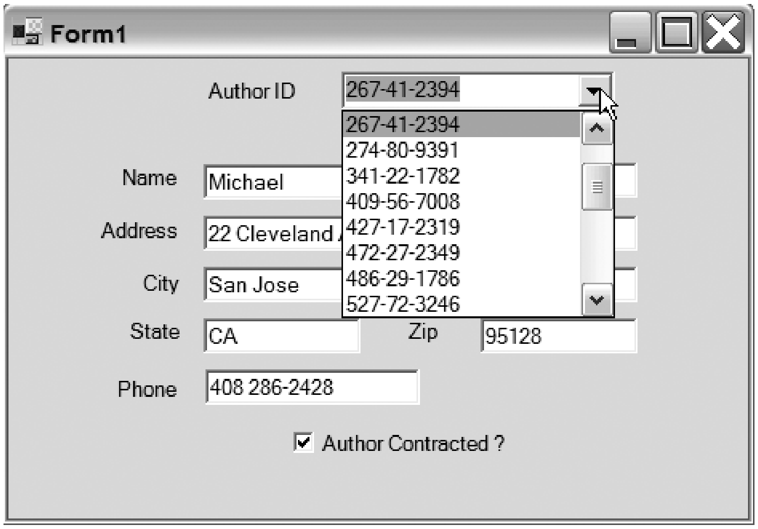
There's a bunch of common properties there. You're most interested in the **Checked** property, so go ahead and choose it, drop down the list of available fields, and select the Contract field of the Authors table.

All that remains is to populate the DataSet. Add code to the Form_Load event so that it looks like this:

```
Private Sub Form1_Load(ByVal sender As System.Object, _
         ByVal e As System.EventArgs) Handles MyBase.Load
    authorAdapter.Fill(pubsDataset)
End Sub
```

Obviously, you'll need to substitute the name you gave your Adapter for authorAdapter in the code here.

Don't forget to add Option Strict On to the very top of the form's source file. When you're done, run the application and explore—your application should look like the one shown in Figure 6-37.



Name	Michael
Address	22 Cleveland /
City	San Jose
State	CA
Zip	95128
Phone	408 286-2428
Author Contracted ?	✓

Figure 6-37. The simple bound form in action

It's worth pointing out that you can use this form to change the data on display. You don't have code in the application to update the database (authorAdapter.Update(pubsDataset)), so you won't do any damage to the database. Changes you make are only propagated as far as the in-memory tables.

Summary

This chapter focused on the concepts behind simple and complex binding, and how to put those concepts into action using the tools built into Visual Studio .NET. There was a great deal of resistance in earlier versions of Visual Basic to using the bound controls, since you ended up writing code that effectively fought against the data control at runtime to achieve the required results. There are no such problems with bound controls in .NET. The data-binding mechanisms in .NET work with you, not against you.