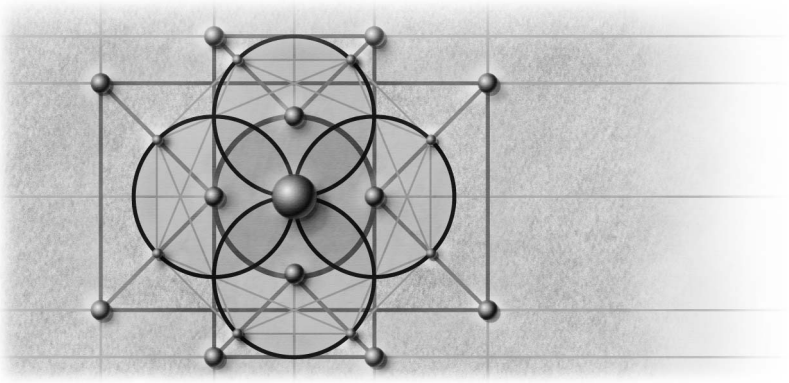

14

Resolving Data Access Issues

Most business applications have some sort of data access. This fact is not surprising when you consider that conceptually, most businesses are themselves based around data: customers and orders, inventory and purchase information—all of this business information needs to be stored in such a way that it can be easily retrieved, edited, and stored again. Countless Microsoft Visual Basic applications have been developed to provide a friendly face for information in databases. Each new version of Visual Basic has delivered enhancements to data access over previous versions. Visual Basic .NET is no exception. With this version, we see continued support for ActiveX Data Objects (ADO) code and data binding. Visual Basic .NET also supports Data Access Objects (DAO) and Remote Data Objects (RDO) code, but it does not support data binding for DAO and RDO. Finally, it introduces a new disconnected type of data access called ADO.NET. This chapter looks at upgrading the three major data access technologies, concentrating on ADO.

The examples in this chapter are all included on the companion CD. They use the Northwind Microsoft Access database. This database is distributed with Microsoft Access, Microsoft Office XP Professional, Visual Basic 6, and various other Microsoft products. The examples in this chapter assume that this database is in the location C:\NWind.mdb. If it is somewhere else, you will need to edit the projects to point to the correct location or move the database to this location.

Data Access in Visual Basic

Data access is made up of three components: code that manipulates data objects, data binding, and design-time tools such as the ADO data environment. The users of your applications may regard the run-time behavior as another component of data access; but we will consider the run-time behavior as an element of each of the three components. Let's look quickly at the three components and see where they differ between Visual Basic 6 and Visual Basic .NET.

Code

DAO, RDO, and ADO are implemented as COM libraries, so most code works exactly the same in Visual Basic .NET as it did in Visual Basic 6. For example, the following ADO code opens the Northwind database and then executes a *Select* statement that returns the list of employees. The first name and last name of the employee is then shown in a message box:

```
Dim cn As New Connection
Dim rs As Recordset
cn.Open "Provider=Microsoft.Jet.OLEDB.3.51;Data
Source=c:\temp\nwind.mdb"
Set rs = cn.Execute("Select * From Employees")
MsgBox rs!FirstName & " " & rs!LastName
rs.Close
cn.Close
```

This code upgrades to the following:

```
Dim cn As ADODB.Connection = New ADODB.Connection()
Dim rs As ADODB.Recordset
cn.Open("Provider=Microsoft.Jet.OLEDB.3.51;Data
Source=c:\temp\nwind.mdb")
rs = cn.Execute("Select * From Employees")
MsgBox(rs.Fields("FirstName").Value & " " & rs.Fields("Last-
Name").Value)
rs.Close()
cn.Close()
```

As we've seen in other examples elsewhere in this book, after it is upgraded the code looks essentially the same, just more explicit. It works perfectly in Visual Basic .NET. Likewise, most RDO and DAO code works perfectly after upgrading, with a few differences that we will discuss later in this chapter.

One major difference across all the data access technologies is how you access fields of a *Recordset* (or *Resultset* for RDO). In Visual Basic 6, it is common to access fields using the shorthand coding convention *RecordsetName!FieldName*. In Visual Basic .NET, this code needs to be expanded in order to resolve the default properties. The Upgrade Wizard does this for you, but the code looks a little different after upgrading. Let's look at an example:

```
Dim rs As Recordset
Dim myString As String
myString = rs!Lastname
```

This code assigns the string *myString* to the field *Lastname* of the *Recordset rs*. It upgrades to the following:

```
Dim rs As ADODB.Recordset
Dim myString As String
myString = rs.Fields("Lastname").Value
```

Notice that *rs!Lastname* was expanded to *rs.Fields("Lastname").Value*. The Upgrade Wizard will handle all of these cases for you.

Data Binding

Data binding allows you to bind the value of a control on a form to a field in a database. Visual Basic 6 supports data binding using ADO, DAO, and RDO. You can bind a control to an ADO data environment, an ADO Data control, a DAO Data control, or an RDO Data control. Visual Basic .NET, however, supports ADO data binding only, not DAO or RDO data binding. This distinction comes from the way in which the different types of data binding are implemented.

DAO and RDO are both older data access technologies. DAO data binding was introduced in Visual Basic 3, and RDO data binding debuted in Visual Basic 4. When the Visual Basic development team first implemented these older forms of data binding, they built them into the forms package. This implementation allowed a seamless integration, but it also tied the data binding technology to Visual Basic Forms. In Visual Basic .NET, Visual Basic Forms has been replaced with Windows Forms—a redesigned forms package. The designers of Windows Forms decided not to build DAO and RDO data binding into Windows Forms, and therefore DAO and RDO data binding are not supported. ADO data binding is supported because it is not built into the forms package; instead it is implemented in the COM library MSBind.dll. This library manages ADO data binding in Visual Basic 6, and the updated library, called Microsoft.VisualBasic.Compatibility.Data, manages data binding in Visual Basic .NET. The Upgrade Wizard upgrades data binding to both the ADO Data control and the ADO data environment.

ADO Data Environment

The Visual Basic 6 ADO data environment lets you visually design database connections and database commands using the ADO Data Environment Designer. Although Visual Basic .NET does not support the ADO data environment, the Upgrade Wizard upgrades the connections, commands, and *Recordset*s to a class that has the same run-time behavior as the Visual Basic 6 data environment.

Components That Don't Upgrade

The components that can't automatically be upgraded to Visual Basic .NET cause errors in the upgraded project. For example, if a Visual Basic 6 form has a DAO Data control, the control is removed during upgrade and EWIs are inserted into the upgrade report. Any code that references the control will cause a compile error.

What should you do if your project uses RDO or DAO data binding? The only solution is to reimplement the data binding in Visual Basic .NET, using ADO or ADO.NET data binding. The best choice is generally ADO.NET. As we discuss in the next section, the design-time experience for ADO.NET is better than for ADO.

Figure 14-1 gives an overview of the different data access technologies in Visual Basic 6 and shows how they upgrade to Visual Basic .NET. Technologies on the left side without an arrow do not automatically upgrade to Visual Basic .NET.

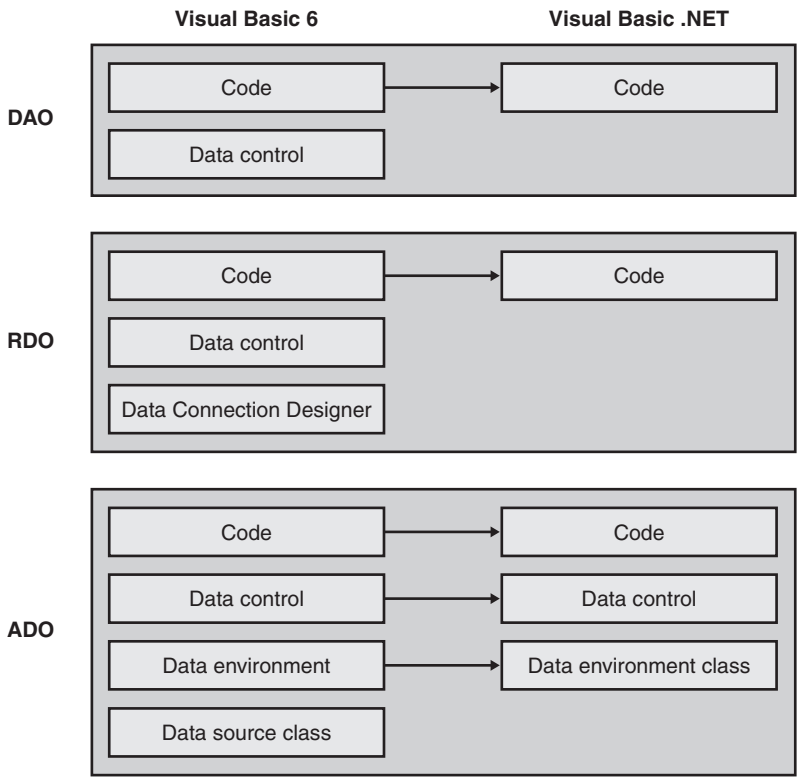

Figure 14-1 How data access technologies upgrade.

ADO.NET Is the Future

The future of data binding and data access is with ADO.NET, not with DAO, RDO, or ADO. In Visual Basic .NET Windows Forms and Web Forms, ADO.NET offers a rich editing experience, with designers for connections, schemas, and data binding. Unfortunately, the ADO editing experience is not as rich as that of either ADO.NET in Visual Basic .NET or ADO in Visual Basic 6. Although ADO data binding is useful for maintaining existing applications, we advise you to use ADO.NET for new applications that use data access, especially those that use data binding.

If ADO.NET is the future, you may ask, why doesn't the Upgrade Wizard upgrade DAO, RDO, and ADO to ADO.NET? Unfortunately, there is no one-to-one mapping to ADO.NET. ADO.NET is a disconnected architecture, and it is distinct enough that automatic upgrading to it is not possible. For example, the *Recordset* object in ADO has the concept of a current row; this is a cursor that points to the current row as you step through a *Recordset* using the *MoveFirst*, *MoveNext*, *MoveLast*, and *MovePrevious* methods. The closest equivalent to a *Recordset* in ADO.NET is a *DataTable* (or *DataSet*, a collection of *DataTable*s). A *DataTable* does not have a current row, and there is no equivalent to the *Move* method of a *Recordset*. For this reason, an automatic upgrade from DAO, RDO, and ADO is not possible. For an introduction to working with ADO.NET, see Chapter 20.

General Issues with Data Access Code

This section looks at some of the changes that are made to your data access when it is upgraded from Visual Basic 6 to Visual Basic .NET. It also discusses solutions to problems you may encounter when upgrading data access code.

DAO and RDO Module Methods

Both DAO and RDO expose module methods. As we discussed in Chapter 13, module methods are properties and functions of the COM library itself that are made globally available. They can be used without creating an instance of any class. For example, the following DAO code opens a database using the *OpenDatabase* module method:

```
Dim db As Database
Set db = OpenDatabase("c:\temp\NWind.mdb")
```

When Visual Basic 6 executes this code, it recognizes that *OpenDatabase* is a method of the DAO COM library rather than of a particular class within the

library. Visual Basic .NET does not support module methods; every method must be qualified with a class.

Because module methods are actually methods of the COM library's underlying *AppID* class, the Upgrade Wizard locates the *AppID* class within the COM component's type library. It then creates a global variable of this class and places it in a module called UpgradeSupport.vb. If this module doesn't already exist, the wizard creates it for you. The name of the variable it creates is *DAODBEngine_definst* for DAO, *RDOrdoEngine_definst* for RDO. When the previous example is upgraded, the wizard creates an UpgradeSupport.vb module with the following code:

```
Module UpgradeSupport
   Friend DAODBEngine_definst As New DAO.DBEngine
End Module
```

The code that uses the *OpenDatabase* method is upgraded to be a method of this default instance:

```
Dim db As DAO.Database
db = DAODBEngine_definst.OpenDatabase("c:\temp\NWind.mdb")
```

Default instances are a good solution for module methods. If you're continuing to write DAO and RDO code, you should be aware of this difference and use the global variable to access the module methods. For example, you could add the following code to the upgraded project that uses the DAO *BeginTrans* and *CommitTrans* methods:

```
DAODBEngine_definst.BeginTrans()
DAODBEngine_definst.CommitTrans()
```

Similarly, you can add the following DAO methods and properties:

BeginTrans	*CommitTrans*	*CompactDatabase*
CreateDatabase	*CreateWorkspace*	*Idle*
OpenConnection	*Properties*	*RegisterDatabase*
RepairDatabase	*Rollback*	*SetOption*
SystemDB	*Version*	*Workspaces*

You can also add the following RDO methods and properties:

rdoCreateEnvironment	*rdoRegisterDataSource*

If you are writing a new Visual Basic .NET project that uses module methods, you will need to add a global default instance variable to your application to access these methods.

ADO Version 2.7

ADO comes in a number of versions: 2.1, 2.2, 2.5, 2.6, and now version 2.7. You can use any of these versions in your application. The different versions are merely different type libraries that point to the same DLL: Msado15.dll. Each time you install a new version of Microsoft Data Access Components (MDAC), it updates the underlying Msado15.dll and installs a new type library for the latest version. ADO 2.7 is installed with Visual Basic .NET. When you upgrade your project, the Upgrade Wizard automatically changes any references to earlier versions of ADO to ADO 2.7. These changes don't cause any problems with upgraded code, since ADO 2.7 is backward compatible with previous versions.

Errors in Events

Most data access applications use events in one way or another. For example, if your project uses the ADO Data control, you may have an event handler for the *ADODC_MoveComplete* event. Another common usage is to declare an *ADO.Connection* variable using the *WithEvents* keyword and to define one or more event handlers for the variable. In Visual Basic .NET, if a COM object raises an event, and an untrapped error occurs in the event, the error will be silently ignored. Let's look at an example—the Visual Basic 6 project prjADOEvent (included on the companion CD). This project has a TextBox bound to an ADO Data control that accesses the Employees table in the Northwind database. When you run this project from the Visual Basic 6 IDE and click the Raise Error button, an error occurs in the *ADODC1_MoveComplete* event, as shown in Figure 14-2.

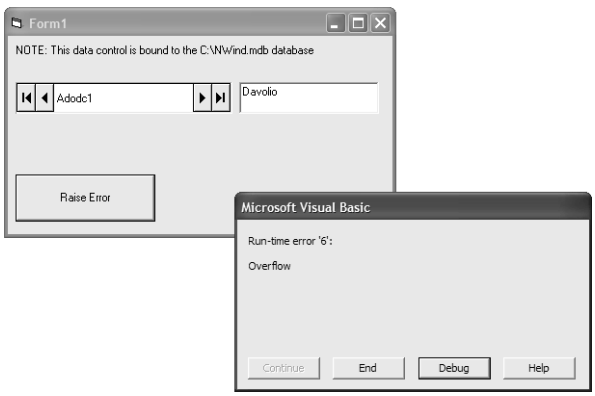

Figure 14-2 Error occurring in the Visual Basic 6 project prjADOEvent.

Try upgrading the application to Visual Basic .NET. Now when you click the Raise Error button, no error occurs. Did Visual Basic .NET somehow fix the

error? Unfortunately not. It is simply ignoring the error, as it does all errors occurring in events raised by COM objects. The solution is to put error trapping code in the event. In this example, the following Visual Basic 6 code for the *MoveComplete* event:

```
Private Sub Adodc1_MoveComplete(ByVal adReason As ADODB.EventReasonE-
num, _
   ByVal pError As ADODB.Error, adStatus As ADODB.EventStatusEnum, _
   ByVal pRecordset As ADODB.Recordset)
   Dim x As Integer, y As Integer
If m_RaiseError Then
   x = x / y
End If
End Sub
```

is upgraded to

```
Private Sub Adodc1_MoveComplete(ByVal adReason As _
   ADODB.EventReasonEnum, _
   ByVal pError As ADODB.Error, ByRef adStatus As _
   ADODB.EventStatusEnum, _
   ByVal pRecordset As ADODB.Recordset) Handles Adodc1.MoveComplete
   Dim x, y As Short
   If m_RaiseError Then
      x = x / y
   End If
End Sub
```

Let's change this event to include *Try...Catch* error trapping:

```
Private Sub Adodc1_MoveComplete(ByVal adReason As _
   ADODB.EventReasonEnum, _
ByVal pError As ADODB.Error, ByRef adStatus As _
   ADODB.EventStatusEnum, _
ByVal pRecordset As ADODB.Recordset) Handles Adodc1.MoveComplete
   Try
       Dim x, y As Short
       If m_RaiseError Then
           x = x / y
       End If
   Catch ex As Exception
       MsgBox(ex.Message)
   End Try
End Sub
```

Now Visual Basic .NET catches and displays the error, as shown in Figure 14-3. You should add error trapping code to all DAO, RDO, and ADO events that may raise errors.

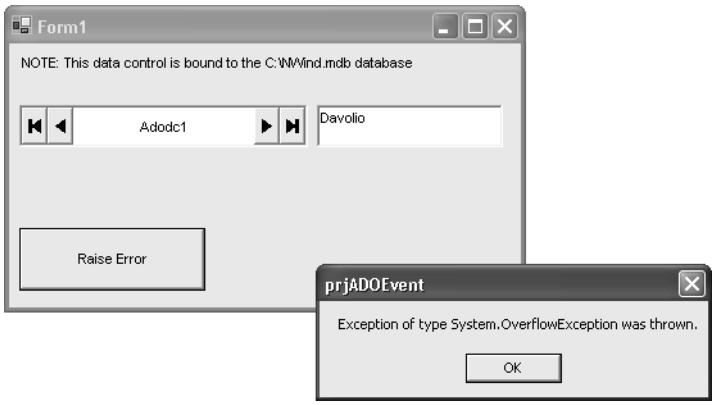

Figure 14-3 Error caught and displayed by Visual Basic .NET.

RDO Connection

A common technique when programming with RDO in Visual Basic 6 is to let the user configure the database connection at run time, rather than requiring it to be set up at design time. To do this, you write code that opens an *rdoConnection* with a blank *Connect* argument, as in the following code:

```
Dim cn As rdoConnection
Dim en As rdoEnvironment
Set en = rdoEnvironments(0)
Set cn = en.OpenConnection("")
```

When this code runs, it brings up the Select Data Source dialog box, as shown in Figure 14-4.

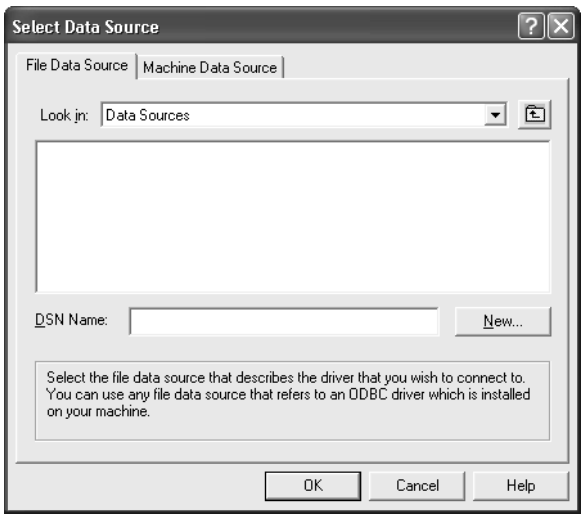

Figure 14-4 Select Data Source dialog box.

The code upgrades to the following:

```
Dim cn As RDO.rdoConnection
Dim en As RDO.rdoEnvironment
en = RDOrdoEngine_definst.rdoEnvironments(0)
cn = en.OpenConnection("")
```

Whether this upgraded code works or not in Visual Basic .NET depends on where it is located. If it is located in a *Button_Click* event, it will work. However, if it is in *Form_Load*, it will fail with the error shown in Figure 14-5.

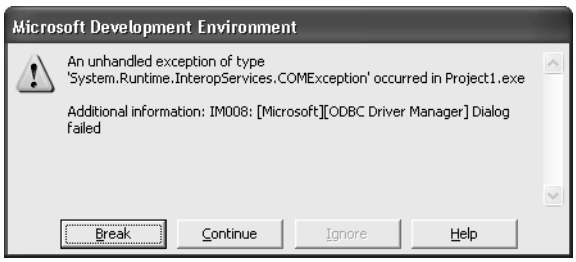

Figure 14-5 RDO connection error in Visual Basic .NET.

The reason that it fails in *Form_Load* is that the dialog box tries to retrieve a handle for the parent window. In Visual Basic 6 that handle exists, but in Visual Basic .NET it hasn't been created yet. The solution is to move the code from *Form_Load* to another event, such as a *Button_Click* event.

Null, *vbNullString*, and *vbNullChar*

As we discussed in Chapter 11, the *Null* value works differently in Visual Basic .NET than it does in Visual Basic 6. The use of *Null* can sometimes lead to subtle, hard-to-find errors. You may find one such error with *vbNullString*. It is a common practice in Visual Basic 6 to use *vbNullString* to reset a *Recordset* filter, as in the following code:

```
Dim rs As New ADODB.Recordset
rs.Filter = vbNullString
```

This code sets the *Recordset* filter to an empty string. After upgrading, it becomes the following:

```
Dim rs As New ADODB.Recordset()
rs.Filter = vbNullString
```

When run, this code causes a COM exception in Visual Basic .NET because *vbNullString* has the value *Nothing* in Visual Basic .NET, whereas in Visual Basic 6 it has the empty string value "". The *Recordset* filter is expecting a string

value rather than a *Null*, and hence the error. The solution is to change the code to use an empty string:

```
Dim rs As New ADODB.Recordset()
rs.Filter = ""
```

This code works perfectly.

ADO Data Environment

Visual Basic 6 introduced the ADO data environment, which provides a visual way to add and edit data objects. With the ADO Data Environment Designer, you can add connections, commands, and child commands. At run time, the ADO data environment acts like a class, with collections of connections, commands, and *Recordset*s.

Because Visual Basic .NET does not support the ADO data environment, the Upgrade Wizard upgrades data environments to a class that has the same run-time behavior. Since it's a class, not an ADO data environment, the editing experience is less rich than in Visual Basic 6. The programming model for the upgraded class is almost identical to Visual Basic 6. For example, suppose your project has a data environment named *myDataEnvironment*, with a command called *myCommand*. The following code would assign a *Recordset* to the result of the command:

```
Dim rs As Recordset
Set rs = myDataEnvironment.rsmyCommand
```

After upgrading, this code looks like the following:

```
Dim rs As ADODB.Recordset
rs = myDataEnvironment.rsmyCommand
```

With a few exceptions that we will discuss now, the upgraded data environment behaves like the original Visual Basic 6 version.

Calling *Dispose*

When your application shuts down, it's important to dispose of the upgraded ADO data environment. This closes all *Recordset*s, commands, and connections. To dispose of the ADO data environment, call the *Dispose* method. For example, suppose your application has an ADO data environment named *DataEnvironment1*. You would dispose of it with the following line of code:

```
DataEnvironment1.Dispose
```

You should put this line somewhere in your application shutdown code.

Initialize Event

In Visual Basic 6, the ADO data environment supports an *Initialize* event and a *Finalize* event. When your application is upgraded, the code in these events is upgraded, but as procedures. The events won't be fired when the application is run. You should move the code in the *Initialize* procedure into the *New* event and the code in the *Finalize* procedure into the *Dispose* event.

Cursor Location with Microsoft Access Databases

In Visual Basic 6, you could set the cursor location of an ADO connection object to *adUseServer* or *adUseClient*, signifying a server-side or client-side cursor location, respectively. If the database is a Microsoft Access database, the property is ignored and a client-side cursor is always used. In Visual Basic .NET, trying to use a server-side cursor with a Microsoft Access database causes a runtime exception, since the value is invalid. This situation may cause problems in upgraded applications if you have inadvertently set the cursor to be server-side. The solution is to remove the line of code in the Visual Basic .NET project that sets the cursor location. For example,

```
Me.Adodc1.CursorLocation = ADODB.CursorLocationEnum.adUseServer
```

This rule applies to the ADO data environment, the ADO Data control, and connections created in code.

ADO Data Binding

With **ADO data binding**, you can bind controls on forms to fields in a database. In Visual Basic 6, you can bind a control to one of three ADO objects:

- ADO data environment
- ADO Data control
- ADO data source classes

The Upgrade Wizard can upgrade the first two data binding technologies, but it cannot upgrade data binding to ADO data source classes. Before we look at how data binding is upgraded, let's take a minute to review how it works in Visual Basic 6.

Each Visual Basic 6 control supports either complex or simple binding. **Complex binding** occurs when a control binds to an entire *Recordset*. The control manages its own data binding. The Microsoft DataGrid ActiveX control is a good example of complex binding. Complex binding is the simplest to upgrade, since it's only a matter of setting the control's *DataSource* to a *Record-*

set and then letting the control manage its own binding. The Upgrade Wizard manages this without problems.

Simple binding is more involved. Simple binding occurs when a control binds its value to the current record in a *Recordset*. Binding a TextBox *Text* property to an ADO Data control is a good example of simple binding. Simple binding in Visual Basic 6 is controlled by four properties of the control. Table 14-1 lists these properties. In Visual Basic 6, these four properties are "magic" properties. When you set them to valid entries, Visual Basic 6 adds an entry to an MSBind binding collection. If you change the properties at run time, Visual Basic 6 removes the old entry from the binding collection and adds a new entry.

Table 14-1 Properties Controlling Simple Binding in Visual Basic 6

Property	Description
DataSource	The name of the ADO Data control or ADO data environment to bind to
DataMember	For ADO data environments only, the name of the command to bind to
DataField	The field of the *Recordset* to bind to
DataFormat	The format in which the data is to be displayed

Windows Forms controls do not have these properties, so the Upgrade Wizard adds code to the form to manage the binding collection. For example, suppose that a form has a TextBox that is bound to a field of an ADO Data control. When it upgrades the form, the Upgrade Wizard does the following:

- Inserts a form variable that manages the control binding. The variable is declared in the form's declarations section. If the ADO Data control is called Adodc1, the binding variable is named *ADOBind_Adodc1*.
- Inserts a method that sets up the binding collection. This method is named *VB6_AddADODataBinding* and is called from the *Form.New* event.
- Inserts a method that removes the bindings and cleans up the objects when the form closes. This method is named *VB6_RemoveADODataBinding* and is called from the *Form.Dispose* event.

There are some data binding elements that the upgrade can't handle automatically. Let's look at them now.

Control Arrays of ADO Data Controls

Control arrays of ADO Data controls cannot automatically be upgraded from Visual Basic 6 to Visual Basic .NET. If you have a form with a control array of ADO Data controls, the control arrays will lose their settings during the upgrade, and the controls will not be bound. The best solution is to remove the controls from the control array in Visual Basic 6 before upgrading.

Setting Data Binding Properties at Run Time

Many projects that use data binding have code that adjusts data binding properties at run time. There are several reasons for this: you may need to change the database location, depending on whether the application is being run against a development database or a production database; or you may need to dynamically change the field a control is bound to. Code that changes data binding properties at run time may need some modifications after upgrading. The reason for requiring these modifications is that the underlying data binding mechanism has changed—you no longer use the *DataSource* and *DataField* properties to set up data binding. Instead, you add and remove binding entries of a data binding variable.

Let's walk through an example to see how to upgrade code that adjusts binding properties at run time. On the companion CD, you will find a project called prjADORuntime. This Visual Basic 6 project has a TextBox bound to an ADO Data control. The binding is changed at run time to point to the Northwind database. In the *Form_Load* event, the code changes the connection string for the ADO Data control and then changes the data binding for the TextBox. When it is run, the form looks like Figure 14-6.

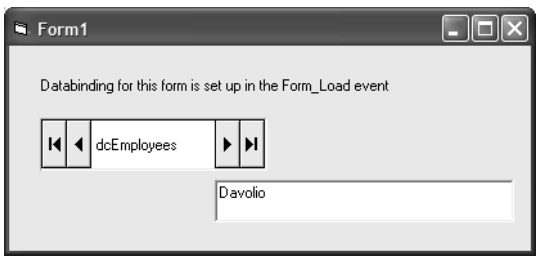

Figure 14-6 Setting up data binding dynamically at run time.

Here is the relevant code in *Form_Load* that changes the data binding:

```
'Set up connection string and recordsource for the ADODC
Me.dcEmployees.ConnectionString = _
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
   m_NorthwindDatabase
Me.dcEmployees.CommandType = adCmdTable
Me.dcEmployees.RecordSource = "Employees"
'Set up binding for the textbox
Set Me.txtLastname.DataSource = Me.dcEmployees
Me.txtLastname.DataField = "Lastname"
```

After upgrading, this section of code becomes the following:

```
'Set up connection string and recordsource for the ADODC
Me.dcEmployees.ConnectionString = _
   "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
   m_NorthwindDatabase
Me.dcEmployees.CommandType = ADODB.CommandTypeEnum.adCmdTable
Me.dcEmployees.RecordSource = "Employees"
'Set up binding for the textbox
'UPGRADE_ISSUE: TextBox property txtLastname.DataSource was not
'upgraded.
Me.txtLastname.DataSource = Me.dcEmployees
'UPGRADE_ISSUE: TextBox property txtLastname.DataField was not
'upgraded.
Me.txtLastname.DataField = "Lastname"
```

Changing the properties of the Data control is fine, but the code that sets up the binding for a particular control also needs to be modified. This code should be changed from

```
'UPGRADE_ISSUE: TextBox property txtLastname.DataSource was not
'upgraded.
Me.txtLastname.DataSource = Me.dcEmployees
'UPGRADE_ISSUE: TextBox property txtLastname.DataField was not
'upgraded.
Me.txtLastname.DataField = "Lastname"
```

to the following:

```
ADOBind_dcEmployees.Remove("txtLastname")
ADOBind_dcEmployees.Add(txtLastname, "Text", "Lastname", _
   Nothing, "txtLastname")
```

The *ADOBind_dcEmployees* variable is the global binding variable. This code removes the entry for the TextBox *txtLastname* and then adds a new entry to bind *txtLastname* to the *Lastname* field. Once this modification has been made, the code works in Visual Basic .NET.

If you want, you can go one step further and adjust the architecture of the form so that binding is set up only after the ADO Data control properties have been set at run time. If you look inside the *Form_New* event, you will find a line that reads

```
VB6_AddADODataBinding()
```

Remove this line of code and add it to the *frmMain_Load* event, so that the *Load* event looks like the following:

```
Private Sub frmMain_Load(ByVal eventSender As System.Object, _
ByVal eventArgs As System.EventArgs) Handles MyBase.Load
   If DoesDatabaseExist(m_NorthwindDatabase) Then
      'Set up connection string and recordsource for the ADODC
      Me.dcEmployees.ConnectionString = _
      "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" & _
      m_NorthwindDatabase
      Me.dcEmployees.CommandType = ADODB.CommandTypeEnum.adCmdTable
      Me.dcEmployees.RecordSource = "Employees"
      VB6_AddADODataBinding()
   Else
      MsgBox("Northwind database can't be found at " & _
      m_NorthwindDatabase, MsgBoxStyle.Critical)
      End
   End If
End Sub
```

What we have done here is set up the correct connection string for the ADO Data control before adding the binding. The Visual Basic .NET data binding code may be more verbose than that in Visual Basic 6, but it also offers more control over how and when the data binding is set up.

If you are adjusting the data binding at run time, it is important to set some binding at design time first. The Upgrade Wizard adds the binding variables and procedures only if it detects that a form has data binding. If the binding is set only at run time, the Upgrade Wizard will not create the binding variable and procedures.

Conclusion

This chapter has looked at how to upgrade data access to Visual Basic .NET. DAO, RDO, and ADO code upgrades easily. ADO data binding also upgrades well, but Visual Basic .NET does not support DAO and RDO data binding. If your application relies heavily on data binding, it may be useful to reimplement the data binding in ADO.NET, since Visual Basic .NET has a rich design-time experience for ADO.NET data binding. For more information on moving ADO to ADO.NET, see Chapter 20.