Chapter 6
Improved Data Access

There are several ways you can access non-VFP data (such as SQL Server or Oracle) in VFP applications: remote views, SQL Pass-Through, ADO, and XML. VFP 8 introduces an exciting new technology called CursorAdapter that makes accessing remote data much easier than it was in earlier versions.

More and more VFP developers are storing their data in something other than VFP tables, such as SQL Server or Oracle. There are a lot of reasons for this, including fragility of VFP tables (both perceived and actual), security, database size, and corporate standards. Microsoft has made access to non-VFP data easier with every release, and has even been encouraging it with the inclusion of MSDE (Microsoft Data Engine, a free, stripped-down version of SQL Server) on the VFP CD, starting with version 7.

However, accessing a back-end database has never been quite as easy as using VFP tables. In addition, there are a variety of mechanisms you can use to do this:

- Remote views, which are based on ODBC connections.
- SQL Pass-Through (SPT) functions, such as SQLCONNECT(), SQLEXEC(), and SQLDISCONNECT(), which are also based on ODBC connections.
- ActiveX Data Objects (ADO), which provide an object-oriented front end to OLE DB providers for database engines.
- XML, which is a lightweight, platform-independent, data transport mechanism.

If you've spent any time working with these mechanisms, one of the things you've likely noticed is no two are alike. That means you have a new learning curve with each one, and converting an existing application from one mechanism to another is a non-trivial task.

To help with this problem, Microsoft added a new base class in VFP 8: CursorAdapter. CursorAdapter is one of the biggest new features in VFP 8 because:

- It makes it easier to use ODBC, ADO, or XML, even if you're not very familiar with these technologies.
- It provides a consistent interface to remote data regardless of the access mechanism you choose.
- It makes it easier to switch from one access mechanism to another.

Here's an example of the last point. Suppose you have an application that uses ODBC with CursorAdapters to access SQL Server data, and you want to change to use ADO instead. All you need to do is change the DataSourceType of the CursorAdapters, change the connection to the back-end database, and you're done. The rest of the components in the

application neither know nor care about this; they still see the same cursor regardless of the mechanism used to access the data.

One thing to keep in mind is CursorAdapter is well-named; it acts as an adapter between a source of data and a VFP cursor. So, CursorAdapter creates and manages a VFP cursor, but your forms, reports, and code still operate on the cursor to display and update data.

CursorAdapter PEMs

Let's start examining CursorAdapters by looking at their properties, events, and methods (PEMs).

DataSourceType

This property is very important because it determines the overall behavior of the class and what kinds of values to put into some of the other properties. DataSourceType indicates the mechanism you're using to access the data. The valid choices are "Native" (which indicates you're using native FoxPro tables), "ODBC", "ADO", or "XML".

DataSource

This is the means to access the data. VFP ignores this property when DataSourceType is set to "Native" or "XML". For ODBC, set DataSource to a valid ODBC connection handle (note you have to manage the connection yourself). In the case of ADO, DataSource must be an ADO Recordset with its ActiveConnection property set to an open ADO Connection object (again, you have to manage this yourself).

Alias

As with normal Cursor objects, this property contains the alias of the cursor associated with the CursorAdapter.

UseDEDataSource

This property determines whether the CursorAdapter uses the DataEnvironment's DataSourceType and DataSource properties. If it is set to True (the default is False), you can leave the DataSourceType and DataSource properties alone because the CursorAdapter will use the DataEnvironment's properties instead (VFP 8 adds DataSourceType and DataSource to the DataEnvironment class as well). For example, if you want all the CursorAdapters in a DataEnvironment to use the same ODBC connection, you'd set this to True and use the DataSourceType and DataSource properties in the DataEnvironment to specify the ODBC connection.

SelectCmd

This is the command used to retrieve the data. In the case of all DataSourceTypes except XML, this is typically a SQL SELECT command (such as SELECT * FROM CUSTOMERS) or a stored procedure (for example, EXEC GetCustomersByID 'ALFKI'). In the case of XML, this can either be a valid XML string or an expression (such as a function or method) that returns a valid XML string. In either case, VFP uses an internal XMLTOCURSOR() call to convert the XML to a VFP cursor.

CursorSchema

This property holds the structure of the cursor in the same format you'd use in a CREATE CURSOR command (everything between the parentheses in such a command). Here's an example: CUST_ID C(6), COMPANY C(30), CONTACT C(30), CITY C(25). Although it's possible to leave this blank and tell the CursorAdapter to determine the structure when it creates the cursor, it's better to fill in CursorSchema. For one thing, if CursorSchema is blank or incorrect, you'll either get errors when you open the DataEnvironment of a form or you won't be able to drag and drop fields from the CursorAdapter to the form to create controls. Another reason is this allows you to specify exactly what the structure of the cursor should be. For example, if you want a DateTime field from SQL Server to be converted to a Date field in the VFP cursor, specify "D" for the data type for that field in CursorSchema.

Since VFP has a 255-character limit for values entered into the Property Window, you may have to specify the value for this property in code. Fortunately, the CursorAdapter Builder that comes with VFP (see "Builders" below) can automatically do this for you.

AllowDelete, AllowInsert, AllowUpdate, and SendUpdates

These properties, which default to True, determine whether deletes, inserts, and updates can be done and whether changes are sent to the data source.

KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList

These properties serve the same purpose as the identically-named CURSORSETPROP() properties for views. They are required if you want VFP to automatically update the data source with changes made in the cursor. KeyFieldList is a comma-delimited list of fields (without aliases) that make up the primary key for the cursor. Tables is a comma-delimited list of tables the cursor is based on. UpdatableFieldList is a comma-delimited list of fields (without aliases) that can be updated. UpdateNameList is a comma-delimited list that matches field names in the cursor to field names in the table. The format for UpdateNameList is as follows: CURSORFIELDNAME1 TABLE.FIELDNAME1, CURSORFIELDNAME2 TABLE.FIELDNAME2, ... Note that even if UpdatableFieldList doesn't contain the name of the primary key of the table (because you don't want that field updated), the primary key must still be included in UpdateNameList or updates won't work.

*Cmd, *CmdDataSource, *CmdDataSourceType

If you want to specifically control how VFP deletes, inserts, or updates records in the data source, you can assign the appropriate values to these sets of properties (replace the * above with Delete, Insert, or Update). For example, if you want to use a stored procedure to delete records, set DeleteCmd to something like "EXEC DeleteCustomer CustomerID", and set DeleteCmdDataSource and DeleteCmdDataSourceType to the proper values for the connection.

ConversionFunc

This property specifies conversion functions to be applied to fields when automatic updating is performed. The format for ConversionFunc is as follows: CURSORFIELDNAME1 FUNCTION, CURSORFIELDNAME2 FUNCTION, … (Do not include parentheses at the

end of the function name.) Each function must accept the field name as its only parameter. The function can be a built-in VFP function or a user-defined function (UDF).

For example, if you use the SQL Server VarChar data type, which doesn't have trailing spaces, for customer name and city fields, you'll want to trim the fields in the VFP cursor before sending them to SQL Server. To do that, specify "COMPANY TRIM, CITY TRIM" for ConversionFunc.

Other properties

CursorAdapter has several properties identical to the equivalent CURSORSETPROP() properties: AllowSimultaneousFetch, BatchUpdateCount, CompareMemo, FetchAsNeeded, FetchMemo, FetchSize, MaxRecords, Prepared, UpdateType, UseMemoSize, and WhereType. There are a few other properties as well:

- BufferModeOverride determines how the cursor is buffered (row or table).
- UpdateGram contains the changes made in the cursor in updategram format when the *DataSourceType properties are set to "XML". This property isn't filled in as changes are made, but rather when an update is about to be performed (such as when TABLEUPDATE() has been executed).
- The Flags property contains settings to use when VFP creates the updategram; it has the same values as the Flags parameter in the XMLUPDATEGRAM() function.
- UpdateGramSchemaLocation contains the name and location of a mapping schema if you want to use one; as with Flags, this property works like the equivalent parameter in XMLUPDATEGRAM().
- BreakOnError determines what VFP does when an error occurs in VFP code in any of the CursorAdapter events. If this property is True (the default is False), VFP displays an error message immediately when an error occurs; otherwise, the normal error handling mechanism in place is used.

CursorFill(lUseCursorSchema, lNoData, nOptions, oSource)

This method creates the cursor and fills it with data from the data source (although you can pass True for the lNoData parameter to create an empty cursor). Pass True for the first parameter to use the schema defined in CursorSchema or False to have VFP create an appropriate structure from the data source. We'll discuss the use of the other two parameters later in this chapter.

MULTILOCKS must be set on or this method will fail. If CursorFill fails for any reason, it returns False rather than raising an error; use AERROR() to determine what went wrong (although be prepared for some digging, since the error messages you get often aren't specific enough to tell you exactly what the problem is).

CursorRefresh()

This method is similar to the REQUERY() function: it refreshes the cursor's contents.

CursorAttach(cAlias, lInheritCursorProperties) and CursorDetach()

These methods allow you to attach an existing cursor to a CursorAdapter object or to free the cursor attached to a CursorAdapter. Attaching a cursor means it will be under the control of the CursorAdapter—updates are handled by the CursorAdapter, the cursor is closed when the CursorAdapter is destroyed, and so forth. If you want a cursor to exist after its CursorAdapter is destroyed or no longer want it to be under the control of the CursorAdapter, use CursorDetach.

Before*() and After*()

CursorAdapter has many before and after "hook" events that allow you to customize the behavior of the CursorAdapter. In the case of the Before events, you can return False to prevent the action that triggered it from occurring (this is similar to database events). **Table 1** shows a list of these events.

Table 1. *CursorAdapter has several Before and After events that allow you to hook in additional behaviors when various operations are done to the cursor.*

Event	When Fired
BeforeCursorAttach, AfterCursorAttach	Before and after the CursorAttach method, respectively.
BeforeCursorClose, AfterCursorClose	Before and after the cursor is closed, respectively.
BeforeCursorDetach, AfterCursorDetach	Before and after the CursorDetach method, respectively.
BeforeCursorFill, AfterCursorFill	Before and after the CursorFill method, respectively.
BeforeCursorRefresh, AfterCursorRefresh	Before and after the CursorRefresh method, respectively.
BeforeCursorUpdate, AfterCursorUpdate	Before and after TABLEUPDATE(), respectively; these methods do not fire if the back end is updated implicitly, such as by moving the record pointer when row buffering is used. These events are wrapped around the appropriate Before/AfterDelete, Insert, or Update events, depending on what is happening to the data. For example, for an update operation performed with TABLEUPDATE(), the order of events is BeforeCursorUpdate, BeforeUpdate and AfterUpdate for each record being updated, and finally AfterCursorUpdate.
BeforeDelete, AfterDelete	Before and after a delete operation is sent to the database (even when the delete is done implicitly), respectively. Fires once per record; doesn't fire if BatchUpdateCount is greater than 1.
BeforeInsert, AfterInsert	Before and after an insert operation is sent to the database (even when the insert is done implicitly), respectively. Fires once per record; doesn't fire if BatchUpdateCount is greater than 1.
BeforeUpdate, AfterUpdate	Before and after an update operation is sent to the database (even when the update is done implicitly), respectively. Fires once per record; doesn't fire if BatchUpdateCount is greater than 1.

Some of these events are very interesting and useful. For example, in AfterCursorFill, you can create indexes for the cursor so they're available for SEEK statements or creating relationships between cursors. The before and after events for delete, insert, and update operations (for example, BeforeInsert) receive parameters describing what is happening to the

data, including field and record state (the same value you'd get from GETFLDSTATE(-1)), whether changes are being forced or not, the SQL UPDATE or INSERT command being sent to the database engine (very useful when trying to debug problems), and, if updates are done by deleting the old records and inserting new ones (UpdateType = 2), the SQL DELETE command sent to the database. You can change the commands sent to the database in the Before events if necessary, giving you complete control over how updates are done.

Putting CursorAdapter to work

Here's an example that gets certain fields for Brazilian customers from the Customers table in the Northwind database that comes with SQL Server. The cursor is updateable, so if you make changes in the cursor, close it, and then run the program again, you'll see your changes were saved to the back end.

```
local loCursor as CursorAdapter, ;
  laErrors[1]
set multilocks on
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'Customers'
  .DataSourceType     = 'ODBC'
  .DataSource         = sqlstringconnect('driver=SQL Server;' + ;
    'server=(local);database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .SelectCmd          = "select CUSTOMERID, COMPANYNAME, CONTACTNAME " + ;
    "from CUSTOMERS where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUSTOMERID'
  .Tables             = 'CUSTOMERS'
  .UpdatableFieldList = 'CUSTOMERID, COMPANYNAME, CONTACTNAME'
  .UpdateNameList     = 'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, CONTACTNAME CUSTOMERS.CONTACTNAME'
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
```

The Developer Download files for this chapter, available at ***www.hentzenwerke.com****, include this code in CursorAdapterExample.PRG. You may have to change the setting for the DataSource property to use the appropriate connection information, such as user name and password.*

DataEnvironment, Form, and other changes

To support the new CursorAdapter class, several changes have been made to the DataEnvironment and Form classes and their designers.

First, as mentioned earlier, the DataEnvironment class now has DataSource and DataSourceType properties. It doesn't use these properties itself but they're used by any CursorAdapter member with UseDEDataSource set to True. Second, you can now create

DataEnvironment subclasses visually using the Class Designer. You can even save the DataEnvironment of a form as a class by choosing Save As Class from the File menu when you're in the Form Designer and selecting the DataEnvironment option.

As for forms, you can now specify a DataEnvironment subclass to use by setting the new DEClass and DEClassLibrary properties. If you do this, anything you've done with the existing DataEnvironment (cursors, code, etc.) will be lost, but at least you're warned first.

CURSORGETPROP('SourceType') returns a new range of values: if the cursor was created with CursorFill, the value is 100 + the old value (for example, 102 for remote data). If an existing cursor was attached to the CursorAdapter with CursorAttach, the value is 200 + the old value. If the data source is an ADO Recordset, the value is 104 (CursorFill) or 204 (CursorAttach). You can get a reference to the ADO Recordset object associated with a cursor using CURSORGETPROP('ADORecordset').

Builders

VFP 8 includes new DataEnvironment and CursorAdapter builders that make it easier to work with these classes.

The DataEnvironment Builder is brought up in the usual way; by right-clicking on the DataEnvironment of a form or on a DataEnvironment subclass in the Class Designer and choosing Builder. The "Data Source" page of the DataEnvironment Builder (see **Figure 1**) is where you set data source information. Choose the desired data source type and where the data source comes from. If you choose "Use existing connection handle" (ODBC) or "Use existing ADO Recordset" (ADO), specify an expression containing the data source (such as "goConnectionMgr.nHandle"). You can also choose to use one of the DSNs on your system or a connection string. The Build button, which is only enabled if you choose "Use connection string" for ADO, displays the Data Link Properties dialog, which you can use to build the connection string visually. If you select either "Use DSN" or "Use connection string", the builder generates code in the BeforeOpenTables method of the DataEnvironment to create the desired connection. If you choose "Native", you can select a VFP database container as a data source; in that case, the generated code ensures the database is open (you can also use free tables as the data source).

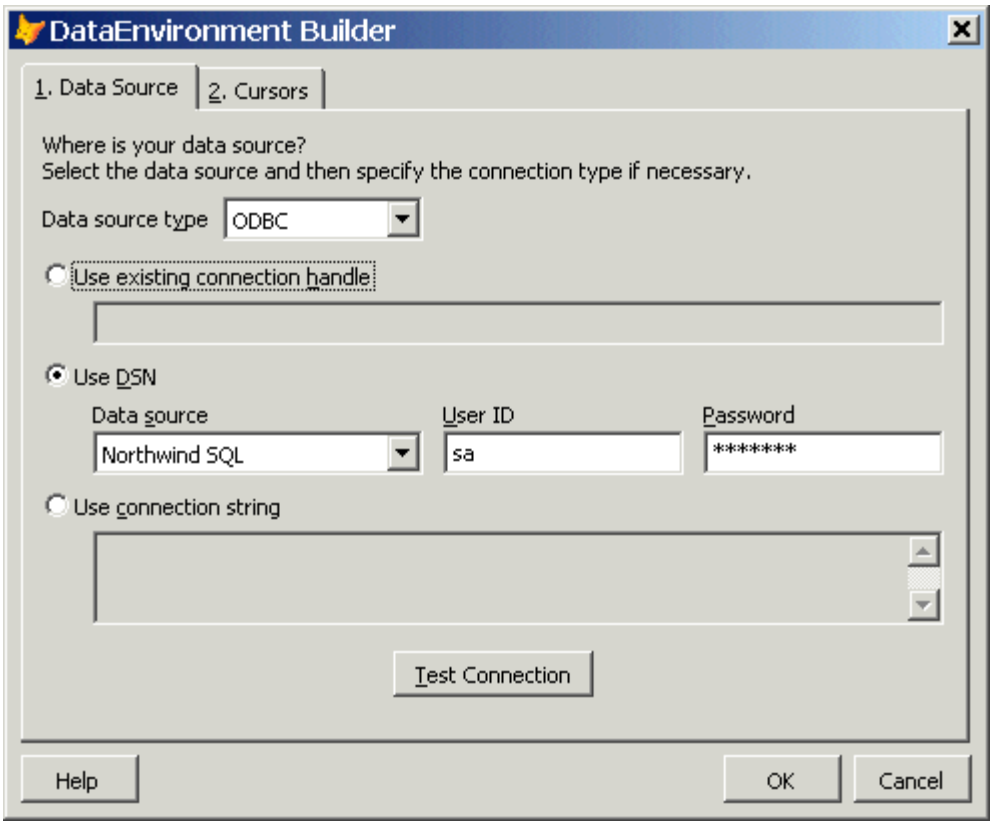

Figure 1. *The Data Source page of the DataEnvironment Builder allows you to specify what type of data source to use and how to connect to it.*

The "Cursors" page, shown in **Figure 2**, allows you to maintain the CursorAdapter members of the DataEnvironment. (Cursor objects don't show up in the builder, nor can they be added.) The Add button allows you to add a CursorAdapter subclass to the DataEnvironment, while New creates a new base class CursorAdapter; in either case, the CursorAdapter Builder is automatically launched so you can work on the new object. Remove deletes the selected CursorAdapter and Builder invokes the CursorAdapter Builder for the selected CursorAdapter. You can change the name of the CursorAdapter object, but you'll need the CursorAdapter Builder to set any other properties.

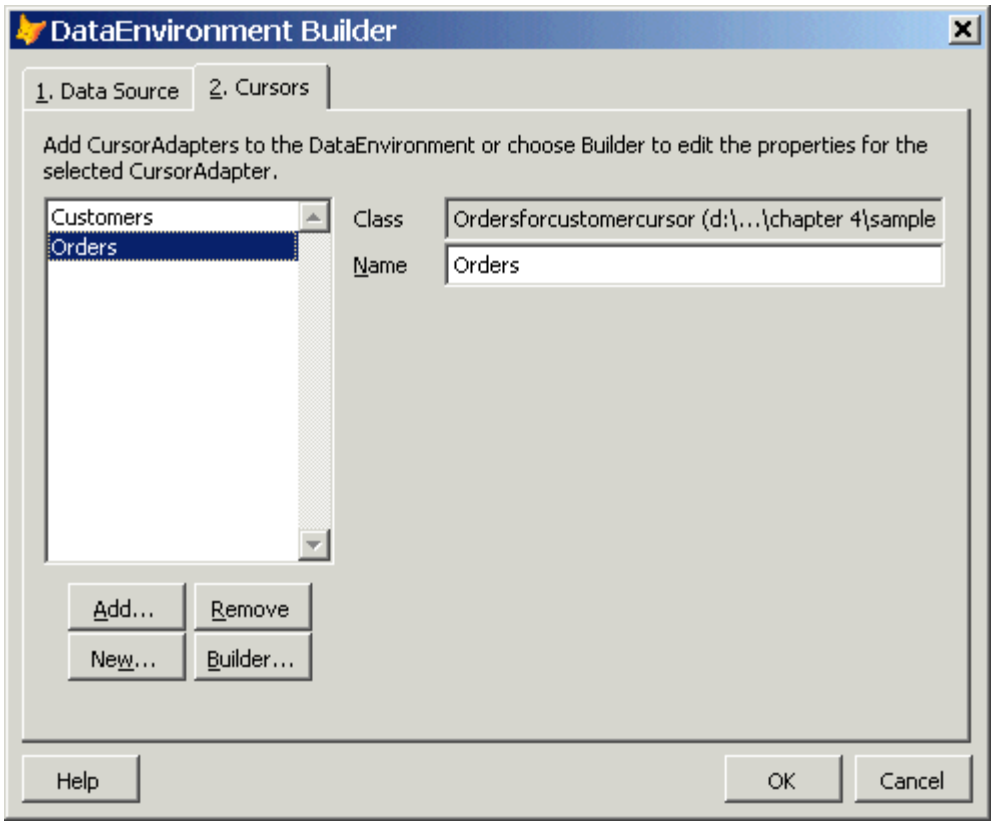

Figure 2. *In the Cursors page, you can add or remove CursorAdapter objects or launch the CursorAdapter Builder for the selected one.*

The CursorAdapter Builder can be invoked by choosing Builder from the shortcut menu for a CursorAdapter or from the DataEnvironment Builder. The "Properties" page (see **Figure 3**) shows the class and name of the object (Name can only be changed if the builder is brought up from a DataEnvironment, since it's read-only for a CursorAdapter subclass), the alias of the cursor it'll create, whether the DataEnvironment's data source should be used or not, and if not, the connection information to use. As with the DataEnvironment Builder, the CursorAdapter Builder generates code to create the desired connection (in the CursorFill method in this case) if you select either "Use DSN" or "Use connection string". You can also specify a connection for the builder to use temporarily; in that case, the builder doesn't generate code for the connection.

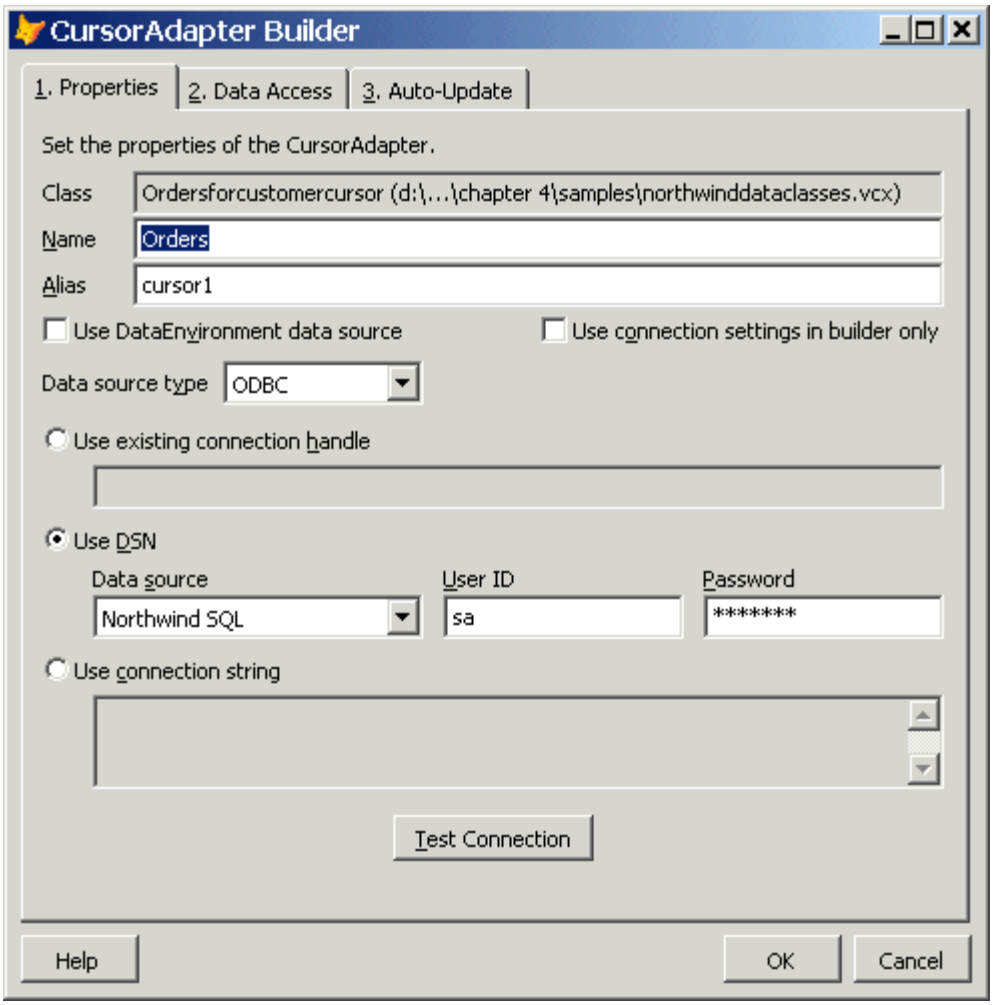

Figure 3. *Use the Properties page of the CursorAdapter Builder to specify several settings for a CursorAdapter, including how it's connected to the database.*

The "Data Access" page, shown in **Figure 4**, allows you to specify the SelectCmd, CursorSchema, and other properties. If you specified connection information, you can click on the Build button for SelectCmd to display the Select Command Builder, which makes it easy to create the SelectCmd. The schema must be less than 255 characters or you'll get a warning message when you click on the OK button.

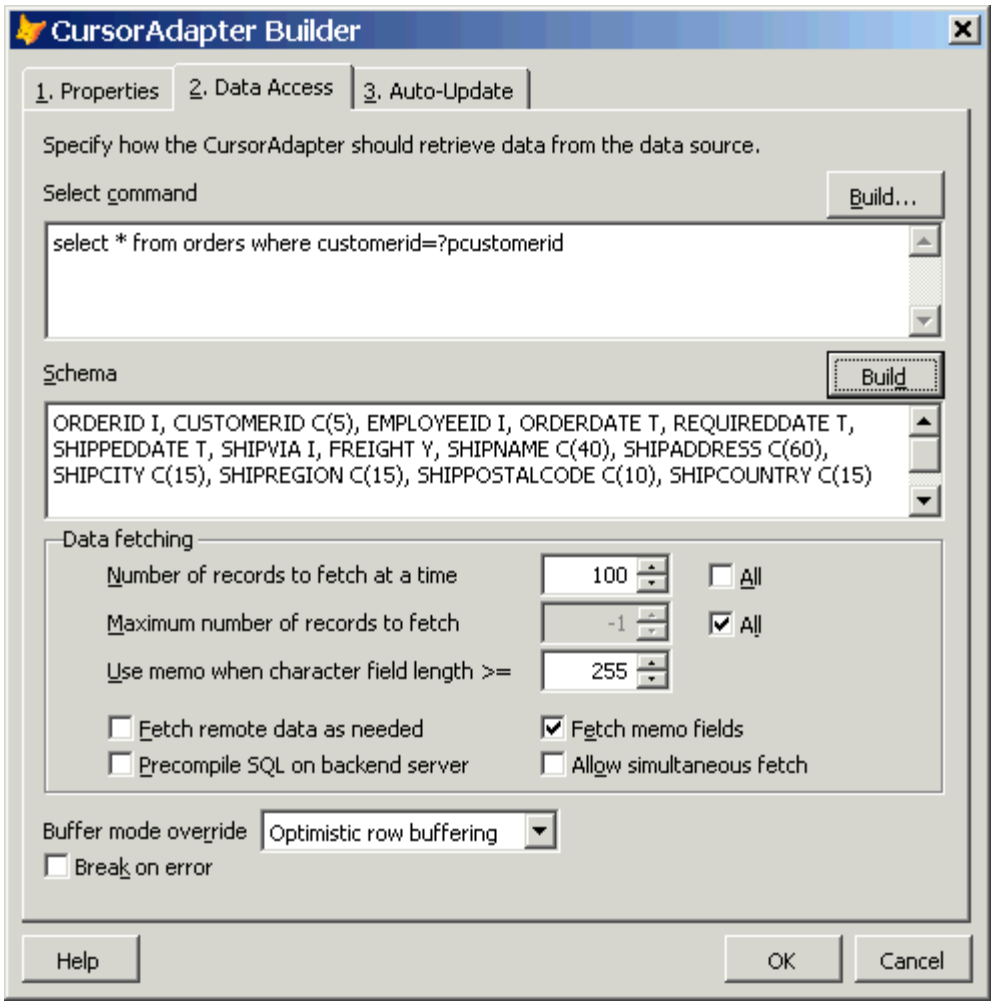

Figure 4. *The Data Access page makes it easy to fill in the SelectCmd and CursorSchema properties, as well as specifying how data access should work.*

The Select Command Builder, illustrated in **Figure 5**, makes short work of building a simple SELECT statement. Choose the desired table from the table drop-down list, and then move the appropriate fields to the selected side. In the case of a native data source, you can add tables to the Table combo box (for example, if you want to use free tables). When you choose OK, the SelectCmd is filled with the appropriate SQL SELECT statement.

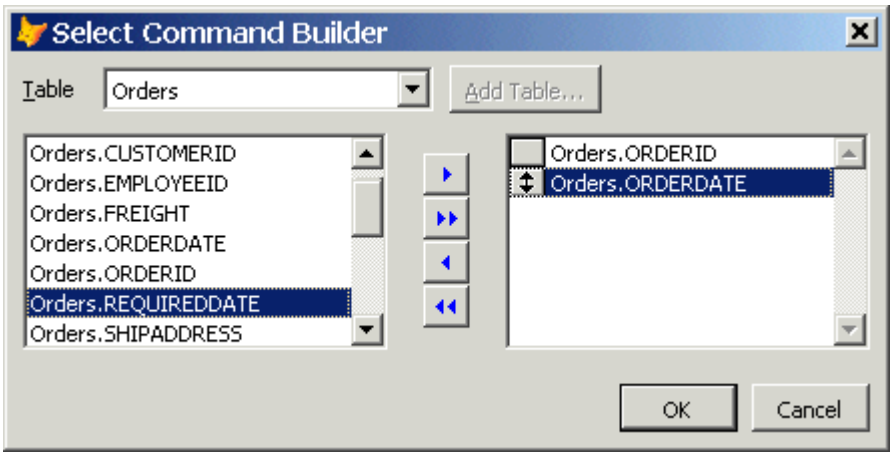

Figure 5. *The Select Command Builder provides a visual tool for creating a SQL SELECT statement.*

Click on the Build button for the CursorSchema to have this property filled in for you automatically. In order for this to work, the builder actually creates a new CursorAdapter object, sets the properties appropriately, and calls CursorFill to create the cursor. If you don't have a live connection to the data source, or CursorFill fails for some reason (such as an invalid SelectCmd), this obviously won't work.

The Auto-Update page is shown in **Figure 6**. Use this page to set the properties necessary for VFP to automatically generate update statements for the data source. The Tables property is automatically filled in from the tables specified in SelectCmd, and the fields grid is filled in from the fields in CursorSchema. As in the View Designer, you select the key fields and which fields are updatable by checking the appropriate column in the grid. You can also set other properties, such as functions to convert the data in certain fields of the cursor before sending it to the data source.

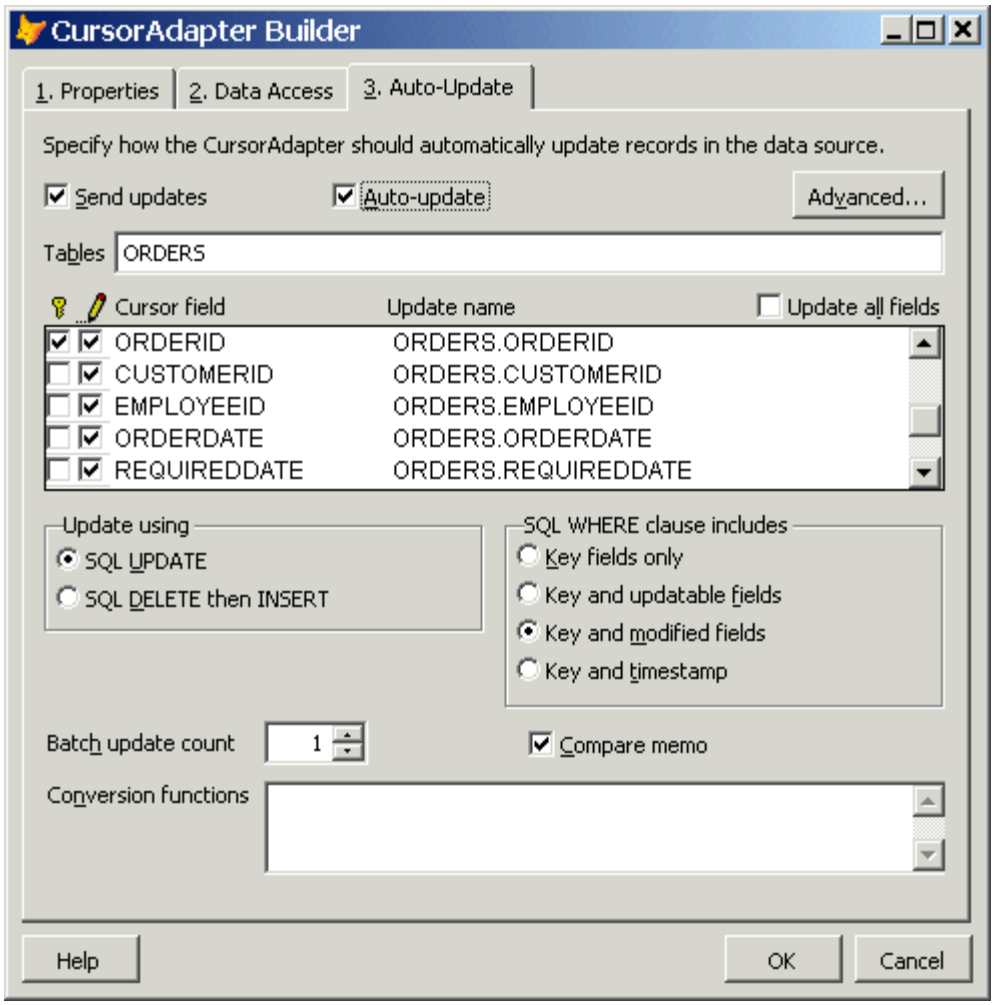

Figure 6. *Use the settings in the Auto-Update page to specify how VFP should generate update statements for the cursor.*

If you want more control over how updates are done, click on the Advanced button to bring up the Advanced Update Properties dialog, shown in **Figure 7**. The Update, Insert, and Delete pages have a nearly identical appearance. They allow you to specify values for the sets of Update, Delete, and Insert properties. This is especially important for XML, because VFP can't automatically generate update statements.

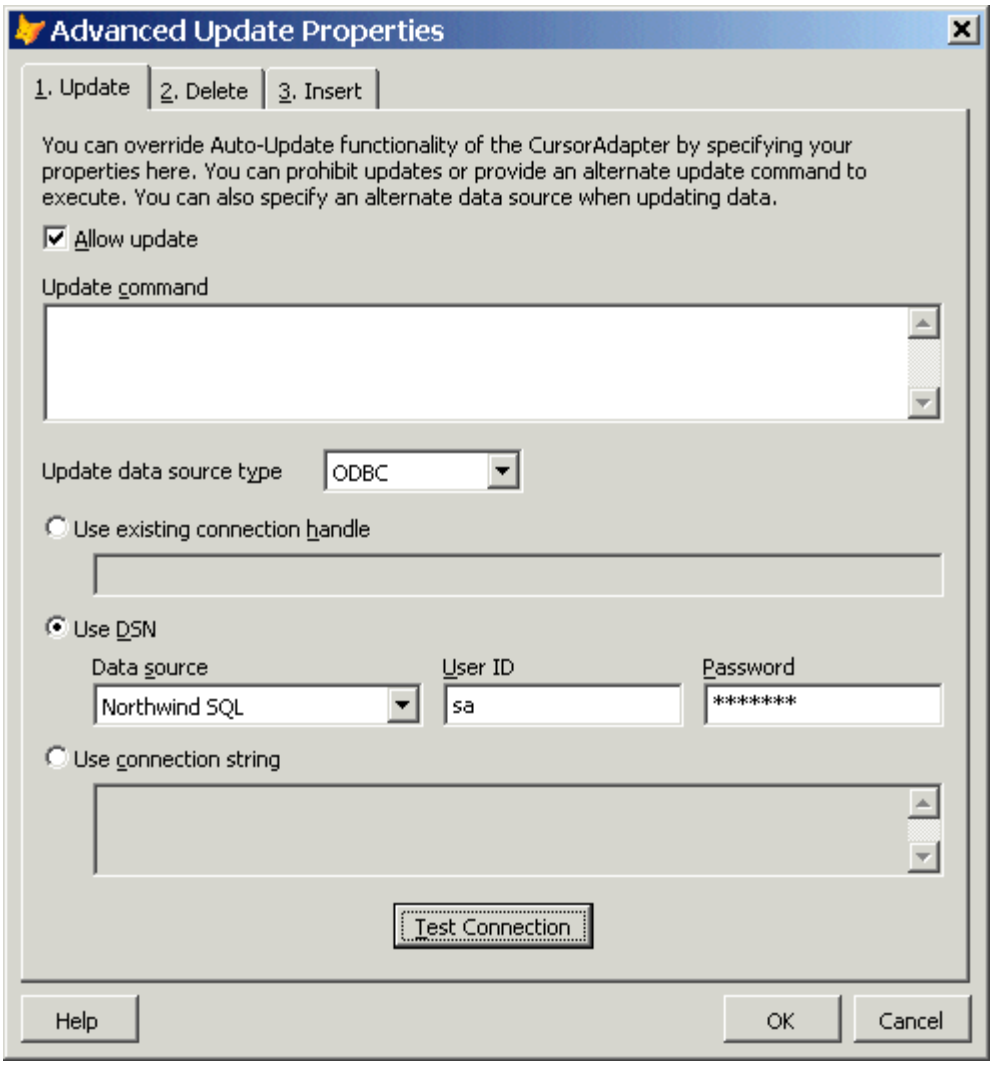

Figure 7. *The Advanced Update Properties dialog allows you to control how updates are sent to the database.*

Data source specifics

The DataSourceType property indicates what mechanism the CursorAdapter uses to talk to the database engine: native (for VFP data), ODBC, ADO, or XML. Each of these types has its own set of rules about how it works, so let's look at the specific details for each one.

Using native data

Even though it's clear CursorAdapter was intended to standardize and simplify access to non-VFP data, you can use it as a substitute for Cursor by setting DataSourceType to "Native". Why would you do this? Mostly as a look toward the future when your application might be upsized; by simply changing the DataSourceType to one of the other choices (and likely changing a few other properties such as setting connection information), you can easily switch to another DBMS such as SQL Server.

When DataSourceType is set to "Native", VFP ignores DataSource. SelectCmd must be a SQL SELECT statement, not a USE command or expression, which means you're always working with the equivalent of a local view rather than with the table directly. You're responsible for ensuring that VFP can find any tables referenced in the SELECT statement, so if the tables aren't in the current directory, you either need to set a path or open the database the tables belong to. As usual, if you want the cursor to be updateable, be sure to set the update properties (KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList).

The following code creates an updateable cursor from the Customer table in the TestData VFP sample database:

```
local loCursor as CursorAdapter, ;
  laErrors[1]
set multilocks on
open database (_samples + 'data\testdata')
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias              = 'customercursor'
  .DataSourceType     = 'Native'
  .SelectCmd          = "select CUST_ID, COMPANY, CONTACT from CUSTOMER " + ;
    "where COUNTRY = 'Brazil'"
  .KeyFieldList       = 'CUST_ID'
  .Tables             = 'CUSTOMER'
  .UpdatableFieldList = 'CUST_ID, COMPANY, CONTACT'
  .UpdateNameList     = 'CUST_ID CUSTOMER.CUST_ID, ' + ;
    'COMPANY CUSTOMER.COMPANY, CONTACT CUSTOMER.CONTACT'
  if .CursorFill()
    browse
    tableupdate(1)
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
close databases all
```

*The Developer Download files at **www.hentzenwerke.com** include this code in NativeExample.PRG.*

Using ODBC

ODBC is actually the most straightforward of the four settings of DataSourceType. You set DataSource to an open ODBC connection handle, set the usual properties, and call CursorFill to retrieve the data. If you fill in KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList, VFP will automatically generate the appropriate UPDATE, INSERT, and

DELETE statements to update the back end with any changes. If you want to use a stored procedure instead, set the *Cmd, *CmdDataSource, and *CmdDataSourceType properties appropriately.

Here's an example that calls the CustOrderHist stored procedure in the Northwind database to get total units sold by product for a specific customer:

```
local loCursor as CursorAdapter, ;
  laErrors[1]
set multilocks on
loCursor = createobject('CursorAdapter')
with loCursor
  .Alias          = 'CustomerHistory'
  .DataSourceType = 'ODBC'
  .DataSource     = sqlstringconnect('driver=SQL Server;server=(local);' + ;
    'database=Northwind;uid=sa;pwd=;trusted_connection=no')
  .SelectCmd      = "exec CustOrderHist 'ALFKI'"
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
```

*The Developer Download files at **www.hentzenwerke.com** include this code in ODBCExample.PRG.*

Using ADO

Using ADO as the data access mechanism with CursorAdapter has a few more issues than using ODBC:

- DataSource must be set to an ADO Recordset with its ActiveConnection property set to an open ADO Connection object.
- If you already have an open ADO Recordset (such as one returned by a middle tier object in an n-tier application), pass it as the fourth parameter to CursorFill. If you do this, the DataSource property is ignored. Also, CursorRefresh works a little differently in this case; you're responsible for refreshing the parameter values before calling CursorRefresh. See the Help topic for CursorRefresh for details.
- For updates to work, the ADO Recordset object must support the Bookmark property. That's only available with client-side cursors, so you should set the CursorLocation property of the Recordset to 3.
- If you want to use a parameterized query (which is usually a better choice than retrieving all records), you have to pass an ADO Command object with its ActiveConnection property set to an open ADO Connection object as the fourth parameter to CursorFill. VFP will take care of filling the Parameters collection of the Command object for you (it parses SelectCmd to find the parameters), but of course the variables containing the values of the parameters must be in scope.

- If you'd rather specify your own delete, insert, or update commands rather than using automatic update, set the appropriate *CmdDataSourceType property to "ADO", *CmdDataSource to an ADO Command object with its ActiveConnection property set to an open ADO Connection object, and *Cmd to the command to execute.
- Using one CursorAdapter with ADO in a DataEnvironment is straightforward; you can set UseDEDataSource to True if you wish, and then set the DataEnvironment's DataSource and DataSourceType properties as you would with the CursorAdapter. However, this doesn't work if there's more than one CursorAdapter in the DataEnvironment. The reason is the ADO Recordset referenced by DataEnvironment.DataSource can only contain a single CursorAdapter's data; when you call CursorFill for the second CursorAdapter, you get a "Recordset is already open" error. So, if your DataEnvironment has more than one CursorAdapter, you must set UseDEDataSource to False and manage the DataSource and DataSourceType properties of each CursorAdapter yourself (or perhaps use a DataEnvironment subclass that manages it for you).

The sample code below shows how to retrieve data using a parameterized query with the help of an ADO Command object. This example also shows the use of the new structured error handling features in VFP 8, discussed in more detail in Chapter 12, "Error Handling". The call to the ADO Connection Open method is wrapped in a TRY structure to trap the COM error the method will throw if it fails. Finally, it shows the use of CursorRefresh to refresh the cursor when the parameter query changes.

```
local loConn as ADODB.Connection, ;
  loCommand as ADODB.Command, ;
  loException as Exception, ;
  loCursor as CursorAdapter, ;
  lcCountry, ;
  laErrors[1]
set multilocks on
loConn = createobject('ADODB.Connection')
with loConn
  .ConnectionString = 'provider=SQLOLEDB.1;data source=(local);' + ;
    'initial catalog=Northwind;uid=sa;pwd=;trusted_connection=no'
  try
    .Open()
  catch to loException
    messagebox(loException.Message)
    cancel
  endtry
endwith
loCommand = createobject('ADODB.Command')
loCursor  = createobject('CursorAdapter')
with loCursor
  .Alias          = 'Customers'
  .DataSourceType = 'ADO'
  .DataSource     = createobject('ADODB.Recordset')
  .SelectCmd      = 'select * from customers where country=?lcCountry'
  lcCountry       = 'Brazil'
  .DataSource.ActiveConnection = loConn
  loCommand.ActiveConnection   = loConn
  if .CursorFill(.F., .F., 0, loCommand)
```

```
    browse
    lcCountry = 'Canada'
    .CursorRefresh()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
```

*The Developer Download files at **www.hentzenwerke.com** include this code in ADOExample.PRG.*

Using XML

There are some issues to be aware of when using XML with CursorAdapters. They are:

- The DataSource property is ignored.
- The SelectCmd property must be set to a source of XML. For example, you can use an expression that returns the XML for the cursor, such as a UDF or object method name. You can also specify the name of an XML document; in that case, pass 512 as the third parameter to CursorFill (which tells it the XML is coming from a file rather than a string).
- Changes made to the cursor are converted to an updategram, which is XML that contains before and after values for changed fields and records, and placed in the UpdateGram property when the update is required.
- In order to write changes back to the data source, UpdateCmdDataSourceType must be set to "XML" and UpdateCmd must be set to an expression (again, likely a UDF or object method) that handles the update. You'll probably want to pass "This.UpdateGram" to the UDF so it can send the changes to the data source. If BufferModeOverride is set to 5-optimistic table buffering, the function specified in UpdateCmd will be called once for each modified record and UpdateGram will only contain changes for record currently being updated.

The XML source for the cursor could come from a variety of places. For example, you could call a UDF that converts a VFP cursor into XML using CURSORTOXML() and returns the results:

```
use CUSTOMERS
cursortoxml('customers', 'lcXML', 1, 8, 0, '1')
return lcXML
```

The UDF could call a Web Service that returns a result set as XML. Here's an example IntelliSense generated from a Web Service. (The details aren't important; it just shows an example of a Web Service. See Chapter 10, "COM and Web Services Enhancements", for a discussion of Web Services.)

```
local loWS as dataserver web service
loWS = NEWOBJECT("Wsclient",HOME()+"ffc\_webservices.vcx")
loWS.cWSName = "dataserver web service"
loWS = loWS.SetupClient("http://localhost/SQDataServer/dataserver.WSDL", ;
  "dataserver", "dataserverSoapPort")
lcXML = loWS.GetCustomers()
return lcXML
```

It could use SQLXML to execute a SQL Server query stored in a template file on a Web Server (for more information on SQLXML, go to **http://msdn.microsoft.com** and search for SQLXML). The following code uses an MSXML2.XMLHTTP object to get all records from the Northwind Customers table via HTTP; this is explained in more detail later.

```
local loXML as MSXML2.XMLHTTP
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/template/' + ;
  'getallcustomers.xml', .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send()
return loXML.responseText
```

Handling updates is more complicated. The data source must either be capable of accepting and consuming an updategram (as is the case with SQL Server 2000) or you have to figure out the changes yourself and issue a series of SQL statements (UPDATE, INSERT, and DELETE) to perform the updates.

Here's an example that uses a CursorAdapter with an XML data source. Notice both SelectCmd and UpdateCmd call UDFs. In the case of SelectCmd, the name of a SQL Server XML template and the customer ID to retrieve is passed to a UDF called GetNWXML, which we'll look at in a moment. For UpdateCmd, VFP passes the UpdateGram property to SendNWXML, which we'll also look at later.

*The Developer Download files at **www.hentzenwerke.com** include all the code necessary for this example: XMLExample.PRG, GetNWXML.PRG, SendNWXML.PRG, and CustomersByID.XML.*

```
local loCustomers as CursorAdapter, ;
  laErrors[1]
set multilocks on
loCustomers = createobject('CursorAdapter')
with loCustomers
  .Alias                  = 'Customers'
  .CursorSchema           = 'CUSTOMERID C(5), COMPANYNAME C(40), ' + ;
    'CONTACTNAME C(30), CONTACTTITLE C(30), ADDRESS C(60), ' + ;
    'CITY C(15), REGION C(15), POSTALCODE C(10), COUNTRY C(15), ' + ;
    'PHONE C(24), FAX C(24)'
  .DataSourceType         = 'XML'
  .KeyFieldList           = 'CUSTOMERID'
  .SelectCmd              = 'GetNWXML([customersbyid.xml?customerid=ALFKI])'
  .Tables                 = 'CUSTOMERS'
  .UpdatableFieldList     = 'CUSTOMERID, COMPANYNAME, CONTACTNAME, ' + ;
```

```
    'CONTACTTITLE, ADDRESS, CITY, REGION, POSTALCODE, COUNTRY, PHONE, FAX'
  .UpdateCmdDataSourceType = 'XML'
  .UpdateCmd               = 'SendNWXML(This.UpdateGram)'
  .UpdateNameList          = 'CUSTOMERID CUSTOMERS.CUSTOMERID, ' + ;
    'COMPANYNAME CUSTOMERS.COMPANYNAME, ' + ;
    'CONTACTNAME CUSTOMERS.CONTACTNAME, ' + ;
    'CONTACTTITLE CUSTOMERS.CONTACTTITLE, ' + ;
    'ADDRESS CUSTOMERS.ADDRESS, ' + ;
    'CITY CUSTOMERS.CITY, ' + ;
    'REGION CUSTOMERS.REGION, ' + ;
    'POSTALCODE CUSTOMERS.POSTALCODE, ' + ;
    'COUNTRY CUSTOMERS.COUNTRY, ' + ;
    'PHONE CUSTOMERS.PHONE, ' + ;
    'FAX CUSTOMERS.FAX'
  if .CursorFill(.T.)
    browse
    tableupdate(.T.)
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
close tables
```

The XML template this code references, CustomersByID.XML, looks like the following:

```
<root xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <sql:header>
    <sql:param name="customerid">
    </sql:param>
  </sql:header>
  <sql:query client-side-xml="0">
    SELECT *
    FROM   Customers
    WHERE CustomerID = @customerid
    FOR XML AUTO
  </sql:query>
</root>
```

Place this file in an IIS virtual directory for the Northwind database (see Appendix 1, "Setting Up SQL Server 2000 XML Access", for details on configuring IIS to work with SQL Server).

Here's the code for GetNWXML. It uses an MSXML2.XMLHTTP object to transfer XML via HTTP. The Open method opens a connection to a URL, which in this case specifies a SQL Server XML template on a Web server. The SetRequestHeader method tells the XMLHTTP object what type of data is being transferred. The Send method sends the request to the server and puts the results into the ResponseText property. The name of the template (and optionally any query parameters) is passed as a parameter to this code.

```
lparameters tcURL
local loXML as MSXML2.XMLHTTP
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/template/' + tcURL, .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send()
return loXML.responseText
```

SendNWXML looks similar, except it expects to be passed an updategram, loads the updategram into an MSXML2.DOMDocument object, and passes that object to the Web server, which will in turn pass it via SQLXML to SQL Server for processing.

```
lparameters tcUpdateGram
local loDOM as MSXML2.DOMDocument, ;
  loXML as MSXML2.XMLHTTP
loDOM = createobject('MSXML2.DOMDocument')
loDOM.async = .F.
loDOM.loadXML(tcUpdateGram)
loXML = createobject('MSXML2.XMLHTTP')
loXML.open('POST', 'http://localhost/northwind/', .F.)
loXML.setRequestHeader('Content-type', 'text/xml')
loXML.send(loDOM)
```

To see how this works, run XMLExample.prg. You should see a single record (the ALFKI customer) in a browse window. Change the value in some field, close the window, and run the PRG again. You should see your change was written to the back end.

Update issue

You have a lot of flexibility when updating the original data source from changes in the cursor created by CursorAdapter. The easiest thing to do is let VFP handle the updates by setting the KeyFieldList, Tables, UpdatableFieldList, and UpdateNameList properties. For more control, set the *Cmd, *DataSource, and *DataSourceType properties (where "*" is "Delete", "Insert", or "Update"). However, there's an issue you should be aware of; update conflicts don't work the same way they do with VFP data.

With UpdateType set to 1 (the default) and automatic updates, VFP sends a SQL UPDATE command to the data source. The UPDATE command is usually something like the following (in this case, CUSTOMERID is the key field and the value of the COMPANYNAME field was changed):

```
UPDATE CUSTOMERS SET COMPANYNAME=?customer.companyname
  WHERE CUSTOMERID=?OLDVAL('customerid','customer') AND
  COMPANYNAME=?OLDVAL('companyname','customer')
```

What happens if another user also changed the company name? In that case, the WHERE clause will fail because no record with the former name of the company can be found. However, this doesn't cause an error. As a result, it appears the update succeeded—TABLEUPDATE() returns True.

This situation may be better or worse if UpdateType is set to 2. In that case, VFP generates SQL DELETE and INSERT commands similar to the following:

```
DELETE FROM CUSTOMERS WHERE CUSTOMERID=?OLDVAL('customerid','customer') AND
  COMPANYNAME=?OLDVAL('companyname','customer')
INSERT INTO CUSTOMERS (CUSTOMERID, COMPANYNAME, CONTACTNAME, CONTACTTITLE,
  ADDRESS, CITY, REGION, POSTALCODE, COUNTRY, PHONE, FAX) VALUES
  (?customer.customerid, ?customer.companyname, ?customer.contactname,
  ?customer.contacttitle, ?customer.address, ?customer.city, ?customer.region,
  ?customer.postalcode, ?customer.country, ?customer.phone, ?customer.fax)
```

In the case of a conflict, the DELETE command will fail, but not cause an error. The INSERT command will fail with a duplicate primary key error if the table has a primary key (which is good, since TABLEUPDATE() will return False) or will succeed and create a duplicate record if the table doesn't have a primary key (which is bad).

One way to handle this is to change the SQL UPDATE and DELETE commands so they cause an error if they fail. In the case of SQL Server, you can use code like the following in the BeforeUpdate method of the CursorAdapter to modify the UPDATE and DELETE commands to raise an error if the process failed:

```
lcErrorCode      = " if @@ROWCOUNT=0 RAISERROR('Update conflict!', 16, 1)"
cUpdateInsertCmd = cUpdateInsertCmd + lcErrorCode
cDeleteCmd       = iif(empty(cDeleteCmd), '', cDeleteCmd + lcErrorCode)
```

Reusable data classes

One thing VFP developers have asked Microsoft to add to VFP for a long time is reusable data environments. For example, you may have a form and a report with exactly the same data setup, but you have to manually fill in the DataEnvironment for each one because DataEnvironments aren't reusable. Some developers (and almost all frameworks vendors) made it easier to create reusable DataEnvironments by creating DataEnvironments in code (they couldn't be subclassed visually) and using a "loader" object on the form to instantiate the DataEnvironment subclass. However, this was kind of a kludge and didn't help with reports.

Now, in VFP 8, we have the ability to create both reusable data classes, which can provide cursors from any data source to anything that needs them, and reusable DataEnvironments, which can host the data classes. Unfortunately, you can't use a DataEnvironment subclass in a report, but you can add CursorAdapters or CursorAdapter subclasses to the report's DataEnvironment to take advantage of reusability there.

Here's an example. If you want a CursorAdapter that works with the Northwind Customers table, it makes more sense to create a subclass to do that rather than create separate instances and fill in the properties every time you need one. **Table 2** shows the properties for a subclass of CursorAdapter called CustomersCursor.

Table 2. *The properties of the CustomersCursor class provide a CursorAdapter that knows how to access and update the Northwind Customers table.*

Property	Value
Alias	Customers
CursorSchema	CUSTOMERID C(5), COMPANYNAME C(40), CONTACTNAME C(30), CONTACTTITLE C(30), ADDRESS C(60), CITY C(15), REGION C(15), POSTALCODE C(10), COUNTRY C(15), PHONE C(24), FAX C(24)
KeyFieldList	CUSTOMERID
SelectCmd	select * from customers
Tables	CUSTOMERS
UpdatableFieldList	CUSTOMERID, COMPANYNAME, CONTACTNAME, CONTACTTITLE, ADDRESS, CITY, REGION, POSTALCODE, COUNTRY, PHONE, FAX
UpdateNameList	CUSTOMERID CUSTOMERS.CUSTOMERID, COMPANYNAME CUSTOMERS.COMPANYNAME, CONTACTNAME CUSTOMERS.CONTACTNAME, CONTACTTITLE CUSTOMERS.CONTACTTITLE, ADDRESS CUSTOMERS.ADDRESS, CITY CUSTOMERS.CITY, REGION CUSTOMERS.REGION, POSTALCODE CUSTOMERS.POSTALCODE, COUNTRY CUSTOMERS.COUNTRY, PHONE CUSTOMERS.PHONE, FAX CUSTOMERS.FAX

You can use the CursorAdapter Builder to do most of the work, especially setting the CursorSchema and update properties. The trick is to turn on the "use connection settings in builder only" option, fill in the connection information so you have a live connection, fill in the SelectCmd, and use the builder to build the rest of the properties for you.

Now, anytime you need records from the Northwind Customers table, you simply use the CustomersCursor class. Of course, we haven't defined any connection information, but that's actually a good thing, because this class shouldn't have to worry about things like how to get the data (ODBC, ADO, or XML) or even what database engine to use (there are Northwind databases for SQL Server, Access, and, new in version 8, VFP). Here's an example that uses the CustomersCursor class; notice it just needs to set the connection information in order to have a fully updateable cursor of Northwind Customers.

```
loCursor = newobject('CustomersCursor', 'NorthwindDataClasses')
with loCursor
  .DataSourceType = 'ODBC'
  .DataSource     = sqlstringconnect('driver=SQL Server;' + ;
    'server=(local);database=Northwind;uid=sa;pwd=;' + ;
    'trusted_connection=no')
  if .CursorFill()
    browse
  else
    aerror(laErrors)
    messagebox(laErrors[2])
  endif
endwith
```

*The Developer Download files at **www.hentzenwerke.com** include this code as TestCustomersCursor.PRG, plus NorthwindDataClasses.VCX, which has the CustomersCursor class definition. Additional sample code not described here is also provided, including subclasses of CursorAdapter and DataEnvironment called SFCursorAdapter and SFDataEnvironment that provide additional functionality and can serve as the parent classes for specialized subclasses.*

One thing to be aware of when you subclass CursorAdapter is the unique order in which events fire at instantiation. For a CursorAdapter instantiated in code, the Init event fires before all others, as you'd expect. However, for a CursorAdapter in the DataEnvironment of a form, here's the event order:

```
DataEnvironment.OpenTables
DataEnvironment.BeforeOpenTables
CursorAdapter.AutoOpen
CursorAdapter.BeforeCursorFill
CursorAdapter.CursorFill
CursorAdapter.AfterCursorFill
CursorAdapter.Init
DataEnvironment.Init
```

Since Init fires after CursorFill, you can't put code that initializes properties such as DataSource and DataSourceType into Init—that code would execute too late. Instead, put this code into BeforeCursorFill or CursorFill.

You can add base class CursorAdapters to the DataEnvironment of a report but you can't add CursorAdapter subclasses, at least not visually. The reason is VFP stores only the class name, not the class library, in the FRX, so when you reopen or run the report, VFP doesn't know where the CursorAdapter subclasses are defined. If you want to use CursorAdapter subclasses, add them programmatically using AddObject or NewObject in the Init of the DataEnvironment.

Summary

CursorAdapter is one of the biggest and most exciting enhancements in VFP 8 because it provides a consistent and easy-to-use interface to remote data, makes it easy to switch from one access mechanism to another, and allows you to create truly reusable data classes.

VFP 8 has a lot of other data-related changes as well. We'll examine these in detail in the next two chapters.

Updates and corrections to this chapter can be found on Hentzenwerke's web site, **www.hentzenwerke.com**. Click on "Catalog" and navigate to the page for this book.