Reusable Data Classes

Doug Hennig

VFP developers have wanted reusable data classes for a long time. Although there were various ways of handling this in earlier versions, they were often kludges. Now in VFP 8, we have true reusable data classes. This month, Doug looks at subclasses of CursorAdapter and DataEnvironment, creates some reusable data classes from these subclasses, and shows how these data classes can be used in both forms and reports.

Over the past two issues, we've looked at the new CursorAdapter base class added in VFP 8. This is, in my opinion, one of the most important changes in VFP 8 because it provides us with an easy-to-use, consistent interface to non-VFP data such as SQL Server. As we'll see this month, they also form the basis of reusable data classes.

Before we look at reusable data classes, let's look at subclasses of CursorAdapter and DataEnvironment that I created that provide additional behavior and will form the starting point for our reusable data classes.

SFCursorAdapter

SFCursorAdapter (in SFDataClasses.vcx) is a subclass of CursorAdapter that has some additional functionality added:

- It can automatically handle parameterized queries; you can define the parameter values as static (a constant value) or dynamic (an expression, such as "=Thisform.txtName.Value", that's evaluated when the cursor is opened or refreshed).
- It can automatically create indexes on the cursor after it's opened.
- It does some special things for ADO, such as setting the DataSource property to an ADO RecordSet, setting the ActiveConnection property of the RecordSet to an ADO Connection object, and creating and passing an ADO Command object to the CursorFill method when a parameterized query is used.
- It provides some simple error handling (the cErrorMessage property is filled with the error message).
- It has Update and Release methods that are missing in CursorAdapter.

The Init method creates two collections (using the new Collection base class, which maintains collections of things), one for parameters that may be needed for the SelectCmd property, and one for tags that should be created automatically after the cursor is opened. It also sets MULTILOCKS on, since that's required for CursorAdapter cursors.

```
This.oParameters = createobject('Collection')
This.oTags       = createobject('Collection')
set multilocks on
```

The AddParameter method adds a parameter to the parameters collection. Pass this method the name of the parameter (this should match the name as it appears in the SelectCmd property) and optionally the value of the parameter (if you don't pass it now, you can set it later by using the GetParameter method). This code shows a couple of new features in VFP 8: the new Empty base class, which has no properties, events, or methods (PEMs), making it ideal for lightweight objects, and the ADDPROPERTY() function, which acts like the AddProperty method for those objects that don't have that method.

```
lparameters tcName, ;
  tuValue
local loParameter
loParameter = createobject('Empty')
addproperty(loParameter, 'Name',  tcName)
```

```
addproperty(loParameter, 'Value', tuValue)
This.oParameters.Add(loParameter, tcName)
```

Use the GetParameter method to return a specific parameter object; this is normally used when you want to set the value to use for the parameter.

```
lparameters tcName
local loParameter
loParameter = This.oParameters.Item(tcName)
return loParameter
```

The SetConnection method is used to set the DataSource property to the desired connection. If DataSourceType is "ODBC", pass the connection handle. If it's "ADO", DataSource needs to be an ADO RecordSet with its ActiveConnection property set to an open ADO Connection object, so pass the Connection object and SetConnection will create the RecordSet and set its ActiveConnection to the passed object.

```
lparameters tuConnection
with This
  do case
    case .DataSourceType = 'ODBC'
      .DataSource = tuConnection
    case .DataSourceType = 'ADO'
      .DataSource = createobject('ADODB.RecordSet')
      .DataSource.ActiveConnection = tuConnection
  endcase
endwith
```

To create the cursor, call the GetData method rather than CursorFill, since it handles parameters and errors automatically. Pass .T. to GetData if you want the cursor created but not filled with data. The first thing this method does is create privately-scoped variables with the same names and values as the parameters defined in the parameters collection (the GetParameterValue method called from here returns either the Value of the parameter object or the evaluation of the Value if it starts with an "="). Next, if we're using ADO and there are any parameters, the code creates an ADO Command object and sets its ActiveConnection to the Connection object, then passes the Command object to the CursorFill method; the CursorAdapter requires this for parameterized ADO queries. If we're not using ADO or we don't have any parameters, the code simply calls CursorFill to fill the cursor. Note that .T. is passed to CursorFill, telling it to use CursorSchema, if CursorSchema is filled in (this is the behavior I wish the base class had). If the cursor was created, the code calls the CreateTags method to create the desired indexes for the cursor; if not, it calls the HandleError method to handle any errors that occurred.

```
lparameters tlNoData
local loParameter, ;
  lcName, ;
  luValue, ;
  llUseSchema, ;
  loCommand, ;
  llReturn
with This

* If we're supposed to fill the cursor (as opposed to
* creating an empty one), create variables to hold any
* parameters. We have to do it here rather than in a
* method since we want them to be scoped as private.

  if not tlNoData
    for each loParameter in .oParameters
      lcName  = loParameter.Name
      luValue = .GetParameterValue(loParameter)
      store luValue to (lcName)
    next loParameter
  endif not tlNoData
```

```
* If we're using ADO and there are any parameters, we
* need a Command object to handle this.

  llUseSchema = not empty(.CursorSchema)
  if '?' $ .SelectCmd and (.DataSourceType = 'ADO' or ;
    (.UseDEDataSource and .Parent.DataSourceType = 'ADO'))
    loCommand = createobject('ADODB.Command')
    loCommand.ActiveConnection = iif(.UseDEDataSource, ;
      .Parent.DataSource.ActiveConnection, ;
      .DataSource.ActiveConnection)
    llReturn = .CursorFill(llUseSchema, tlNoData, ;
      .nOptions, loCommand)
  else

* Try to fill the cursor.

    llReturn = .CursorFill(llUseSchema, tlNoData, ;
      .nOptions)
  endif '?' $ .SelectCmd ...

* If we created the cursor, create any tags defined for
* it. If not, handle the error.

  if llReturn
    .CreateTags()
  else
    .HandleError()
  endif llReturn
endwith
return llReturn
```

There are a few methods we won't look at here; feel free to examine them yourself. HandleError uses AERROR() to determine what went wrong and puts the second element of the error array into the cErrorMessage property. Requery is similar to GetData, except it refreshes the data in the cursor. Call this method rather than CursorRefresh because, as with GetData, it handles parameters and errors. Update is simple: it just calls TABLEUPDATE() to update the original data source and calls HandleError if it failed. AddTag adds information about an index you want created after the cursor is created to the tags collection, while CreateTags, which is called from GetData, uses the information in that collection in INDEX ON statements.

Here's an example of using this class, taken from TestCursorAdapter.prg, included with this month's Subscriber Downloads. It retrieves records from the Customers table in the Northwind database that comes with SQL Server. It uses a parameterized statement for SelectCmd, shows how the AddParameter method allows you to handle parameters, and demonstrates how you can automatically create tags for the cursor using the AddTag method.

```
lnHandle = sqlstringconnect('driver=SQL Server;' + ;
  'server=(local);database=Northwind;uid=sa;pwd=;' + ;
  'trusted_connection=no')
loCursor = newobject('SFCursorAdapter', 'SFDataClasses')
with loCursor
  .DataSourceType = 'ODBC'
  .Alias          = 'Customers'
  .SelectCmd      = 'select * from customers ' + ;
    'where country = ?pcountry'
  .SetConnection(lnHandle)
  .AddParameter('pcountry', 'Brazil')
  .AddTag('CustomerID', 'CustomerID')
  .AddTag('Company',    'upper(CompanyName)')
  .AddTag('Contact',    'upper(ContactName)')
  if .GetData()
    messagebox('Brazilian customers in CustomerID order')
    set order to CustomerID
    go top
    browse
    messagebox('Brazilian customers in Contact order')
    set order to Contact
```

```
    go top
    browse
    messagebox('Canadian customers')
    loParameter = .GetParameter('pcountry')
    loParameter.Value = 'Canada'
    .Requery()
    browse
  else
    messagebox('Could not get the data. The error ' + ;
      'message was:' + chr(13) + chr(13) + .cErrorMessage)
  endif .GetData()
endwith
sqldisconnect(lnHandle)
```

SFDataEnvironment

SFDataEnvironment (also in SFDataClasses.vcx) is much simpler than SFCursorAdapter, but has some useful functionality added:

- The GetData method calls the GetData method of all SFCursorAdapter members so you don't have to call them individually. Similarly, the Requery and Update methods call the Requery and Update methods of every SFCursorAdapter member.
- Like SFCursorAdapter, the SetConnection method sets DataSource to an ADO RecordSet and sets the ActiveConnection property of the RecordSet to an ADO Connection object. However, it also calls the SetConnection method of any SFCursorAdapter member that has UseDEDataSource set to .F.
- It provides some simple error handling (a cErrorMessage property is filled with the error message).
- It has a Release method.

We'll just look at a couple of the methods in this class. GetData is very simple: it just calls the GetData method of any member object that has that method.

```
lparameters tlNoData
local loCursor, ;
  llReturn
for each loCursor in This.Objects
  if pemstatus(loCursor, 'GetData', 5)
    llReturn = loCursor.GetData(tlNoData)
    if not llReturn
      This.cErrorMessage = loCursor.cErrorMessage
      exit
    endif not llReturn
  endif pemstatus(loCursor, 'GetData', 5)
next loCursor
return llReturn
```

SetConnection is a little more complicated: it calls the SetConnection method of any member object that has that method and that has UseDEDataSource set to .F., then uses code similar to that in SFCursorAdapter to set its own DataSource if any of the CursorAdapters have UseDEDataSource set to .T.

```
lparameters tuConnection
local llSetOurs, ;
  loCursor, ;
  llReturn
with This

* Call the SetConnection method of any CursorAdapter that
* isn't using our DataSource.

  llSetOurs = .F.
  for each loCursor in .Objects
    do case
      case upper(loCursor.BaseClass) <> 'CURSORADAPTER'
```

```
      case loCursor.UseDEDataSource
        llSetOurs = .T.
      case pemstatus(loCursor, 'SetConnection', 5)
        loCursor.SetConnection(tuConnection)
    endcase
  next loCursor

* If we found any CursorAdapters that are using our
* DataSource, we'll need to set our own.

  if llSetOurs
    do case
      case .DataSourceType = 'ODBC'
        .DataSource = tuConnection
      case .DataSourceType = 'ADO'
        .DataSource = createobject('ADODB.RecordSet')
        .DataSource.ActiveConnection = tuConnection
    endcase
  endif llSetOurs
endwith
```

TestDE.prg shows the use of SFDataEnvironment as a container for a couple of SFCursorAdapter classes. Since this example uses ADO, each SFCursorAdapter needs its own DataSource, so UseDEDataSource is set to .F. (the default). Notice that a single call to the DataEnvironment SetConnection method takes care of setting the DataSource property for each CursorAdapter.

```
loConn = createobject('ADODB.Connection')
loConn.ConnectionString = 'provider=SQLOLEDB.1;' + ;
  'data source=(local);database=Northwind;uid=sa;' + ;
  'pwd='
loConn.Open()
set classlib to SFDataClasses
loDE = createobject('SFDataEnvironment')
with loDE
  .AddObject('CustomersCursor', 'SFCursorAdapter')
  with .CustomersCursor
    .Alias          = 'Customers'
    .SelectCmd      = 'select * from customers'
    .DataSourceType = 'ADO'
  endwith
  .AddObject('OrdersCursor', 'SFCursorAdapter')
  with .OrdersCursor
    .Alias          = 'Orders'
    .SelectCmd      = 'select * from orders'
    .DataSourceType = 'ADO'
  endwith
  .SetConnection(loConn)
  if .GetData()
    select Customers
    browse nowait
    select Orders
    browse
  else
    messagebox('Could not get the data. The error ' + ;
      'message was:' + chr(13) + chr(13) + .cErrorMessage)
  endif .GetData()
endwith
loConn.Close()
```

Reusable Data Classes

Now that we have CursorAdapter and DataEnvironment subclasses to work with, let's talk about 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 that have 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. As of this writing, you can't use CursorAdapter or DataEnvironment subclasses in a report, but you can programmatically add CursorAdapter subclasses (such as in the Init method of the DataEnvironment) to take advantage of reusability there.

Let's create data classes for the Northwind Customers and Orders tables. CustomersCursor (in NorthwindDataClasses.vcx) is a subclass of SFCursorAdapter with the properties shown in Table 1.

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

Table 1: Properties for CustomersCursor.

You don't really think I typed the values of all those properties into the Property Window, do you? Of course not! I used the CursorAdapter builder to do all the work. 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, then 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, since 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).

However, notice that this cursor deals with all records in the Customers table. Sometimes, you only want a specific customer. So, CustomerByIDCursor is a subclass of CustomersCursor with SelectCmd changed to "select * from customers where customerid = ?pcustomerid" and the following code in Init:

```
lparameters tcCustomerID
dodefault()
This.AddParameter('pCustomerID', tcCustomerID)
```

This creates a parameter called pCustomerID (which is the same name as the one specified in SelectCmd) and sets it to any value passed. If no value is passed, use GetParameter to return an object for this parameter and set its Value property before calling GetData.

The OrdersCursor class is similar to CustomersCursor except it retrieves all records from the Orders table, and OrdersForCustomerCursor is a subclass that retrieves only those orders for a specific customer.

To test how this works, run TestCustomersCursor.prg. It retrieves a single customer from the Customers table in the SQL Server version of the Northwind database, and then does the same thing using the Access version. This shows how not specifying the connection information in the class itself makes it more flexible and therefore reusable.

Example: Form

Now that we have some reusable data classes, let's put them to use. First, let's create a subclass of SFDataEnvironment called CustomersAndOrdersDataEnvironment that contains CustomerByIDCursor and OrdersForCustomerCursor classes. It has AutoOpenTables set to .F. because we need to set connection information before the tables can be opened, and UseDEDataSource for both CursorAdapters set to .T. This DataEnvironment can now be used in a form to show information about a specific customer, including its orders.

Let's create such a form. CustomerOrders.scx has DEClass and DEClassLibrary set to CustomersAndOrdersDataEnvironment and NorthwindDataClasses.vcx, respectively, so we use our reusable DataEnvironment. It has a fair bit of code in the Load method, but that's because it supports ADO, ODBC, and XML data sources, and it creates its own connections, so most of the code deals with handling those issues. If it only supported a single type, such as ODBC, and if another object was responsible for managing connections (as would be the case in a real application), the code would be as simple as:

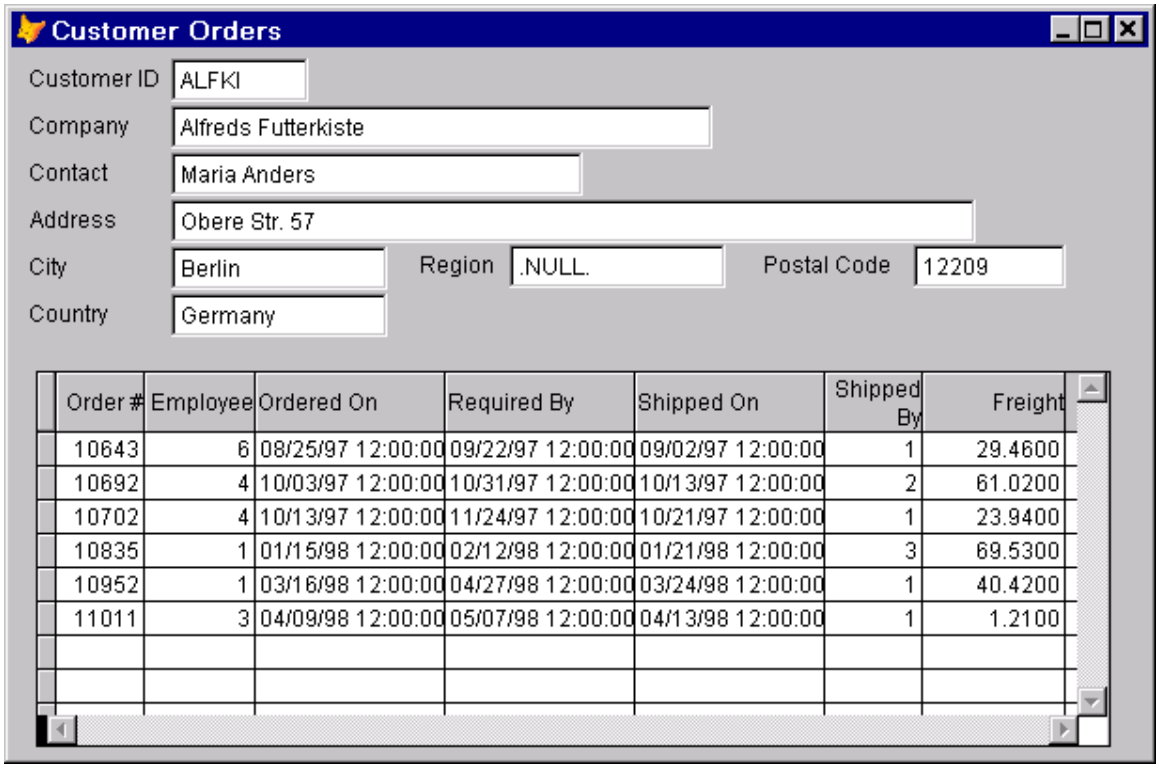
```
with This.CustomersAndOrdersDataEnvironment

* Get the connection.

  lnHandle = oApp.oConnectionMgr.GetConnectionHandle()
  .SetConnection(lnHandle)

* Specify that the value for the cursor parameters will
* be filled from the CustomerID textbox.

  loParameter       = ;
    .CustomerByIDCursor.GetParameter('pCustomerID')
  loParameter.Value = '=Thisform.txtCustomerID.Value'
  loParameter       = ;
    .OrdersForCustomerCursor.GetParameter('pCustomerID')
  loParameter.Value = '=Thisform.txtCustomerID.Value'

* Create empty cursors and display an error message if we
* failed.

  if not .GetData(.T.)
    messagebox(.cErrorMessage)
    return .F.
  endif not .GetData(.T.)
endwith
```

This code uses the GetParameter method of both CursorAdapter objects to set the Value of the pCustomerID parameter to the contents of a textbox in the form. Note the use of the "=" in the value; this means the Value property will be evaluated every time it's needed, so we essentially have a dynamic parameter (saving the need to constantly change the parameter to the current value as the user types in the textbox). The GetData method is called to create empty cursors so the data binding of the controls will work.

The txtCustomerID textbox isn't bound to anything. It calls the Requery method of the DataEnvironment followed by the form's Refresh method. This causes the cursors to be requeried and the controls to be refreshed when a customer ID is entered. The other textboxes in the form are bound to fields in the Customers cursor created by the CustomersByIDCursor object. The grid is bound to the Orders cursor created by the OrdersForCustomerCursor object.

Run the form and enter "ALFKI" for the customer ID (see Figure 1). When you tab out of the textbox, you should see the customer address information and orders appear. Try changing something about the customer or order, then closing the form, running it again, and entering "ALFKI" again. You should see that the changes you made were written to the backend database without any effort on your part.

Figure 1. CustomerOrders.scx shows how a reusable DataEnvironment can be used in a form.



Cool, huh? That wasn't a lot more work than creating a form based on local tables or views. Even better, try changing the ccDATASOURCETYPE constant defined in the Load method to "ADO" or "XML" and notice that the form looks and works exactly the same. (If you want to use XML, note that you'll have to set up a SQLXML virtual directory for the Northwind database as discussed in last month's article, and copy the XML template files provided with this month's Subscriber Downloads to that directory.)

Example: Report

Let's try a report. The biggest issue here is that, unlike a form, we can't tell a report to use a DataEnvironment subclass, nor can we drop CursorAdapter subclasses in the DataEnvironment. So, we'll have to put some code into the report to add CursorAdapter subclasses to the DataEnvironment. Although it might seem logical to put this code into the BeforeOpenTables event of the report's DataEnvironment, that won't actually work because for reasons I don't understand, BeforeOpenTables fires on every page when you preview the report. So, we'll put the code into the Init method. As with CustomerOrders.scx, CustomerOrders.frx has more complicated code than a real report would have due to the needs of the demo. Without these requirements, it would be as simple as:

```
with This
  set safety off

* Get the connection.

  .DataSource = oApp.oConnectionMgr.GetConnectionHandle()

* Create CursorAdapter objects for Customers and Orders.

  .NewObject('CustomersCursor', 'CustomersCursor', ;
    'NorthwindDataClasses')
  .CustomersCursor.AddTag('CustomerID', 'CustomerID')
  .CustomersCursor.UseDEDataSource = .T.
  .NewObject('OrdersCursor', 'OrdersCursor', ;
    'NorthwindDataClasses')
  .OrdersCursor.AddTag('CustomerID', 'CustomerID')
  .OrdersCursor.UseDEDataSource = .T.
```

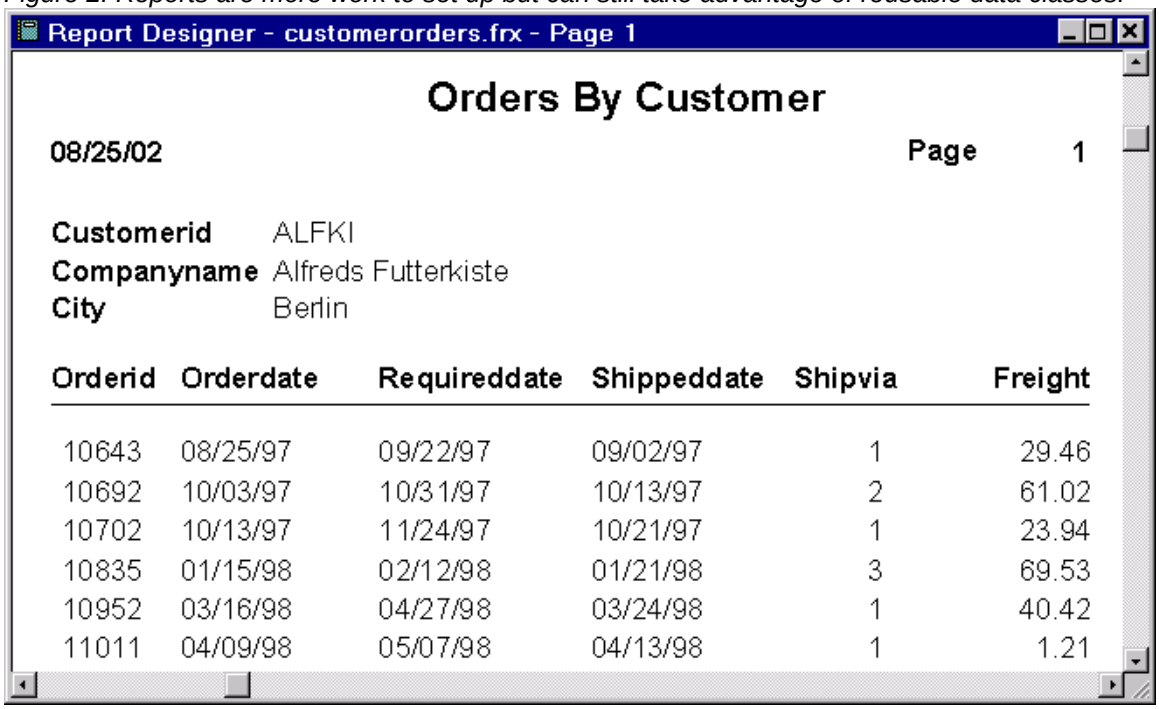
```
* Get the data and display an error message if we failed.

  if not .CustomersCursor.GetData()
    messagebox(.CustomersCursor.cErrorMessage)
    return .F.
  endif not .CustomersCursor.GetData()
  if not .OrdersCursor.GetData()
    messagebox(.OrdersCursor.cErrorMessage)
    return .F.
  endif not .OrdersCursor.GetData()

* Set a relation from Customers to Orders.

  set relation to CustomerID into Customers
endwith
```

There's more code here than in the form because we can't use a DataEnvironment subclass and have to code the behavior ourselves.

Now, how do we conveniently put the fields on the report? Since the CursorAdapter don't exist in the DataEnvironment at design time, we can't just drag fields from them to the report. Here's a tip: create a PRG that creates the cursors and leaves them in scope (either by suspending or making the CursorAdapter objects public), then use the Quick Report function to put fields with the proper sizes on the report.

Figure 2 shows what this report looks like when you preview it. As with the form, try changing the #DEFINE statement in DataEnvironment.Init to try it with other data source types.

Figure 2. Reports are more work to set up but can still take advantage of reusable data classes.



Summary

This ends our examination of the new CursorAdapter base class. I'm very excited about CursorAdapter and plan on revamping the data handling components of my framework to take full advantage of it.

Next month, we'll look at how error handling has been greatly improved in VFP 8 with the new structured error handling command TRY ... CATCH ... FINALLY ... ENDTRY and the new Exception base class.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, co-author of "What's New in Visual FoxPro 7.0" and "The Hacker's Guide to Visual FoxPro 7.0", both from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals", both from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com