Introduction to N-Tier Development

(or, How I Learned to Stop Worrying and Love the COM)

By Doug Hennig

Overview

As applications get ever more complex, designing application in components becomes very important to the successful completion of projects. This session discusses what n-tier development is, why it's important to modern application development, and shows strategies of how you can break out the tiers using Visual FoxPro.

A Common Scenario?

Let's say you work for an organization that uses a VFP-based accounting system that was developed in-house. Your boss asks you to create a contact management application that uses the accounting customers table as the source of contacts (you can't use a commercial application like Act or Maximizer because they have their own contacts files). So, you create a VFP form with controls bound to the fields in the customers table, and incorporate business rules such as the postal code can't be left blank and credit limits over $15,000 have to be approved by a manager. Everyone in the company likes and uses your application.

Then, one day, your boss tells you that the current accounting system isn't cutting it anymore and they're buying a commercial accounting system like ACCPAC. She wants you to modify the contact management application to work with ACCPAC. Since ACCPAC uses Pervasive SQL (formerly Btrieve) as its database engine, you have a problem, since you can't bind controls directly to the Pervasive data. You purchase one of the third-party libraries for accessing Pervasive data in VFP (such as DBFTrieve) and completely change the form so now the data from Pervasive goes to and from form properties, and the controls in the form are bound to these properties. Of course, although the business rules are still the same, the code that implements them has to be rewritten because the data isn't in a cursor any more. You roll the new application out after several months work and it works great.

Then, one day, your boss tells you that due to company growth, the accounting system is being upgraded to the SQL Server version of ACCPAC. Since VFP can hit SQL Server data with remote views, you completely redo your form again so it's more like the first version, although the data is in views rather than tables. Not everyone is happy with the first implementation, though, because it's really slow over the network. So, you implement bandwidth-saving features like eliminating grids and only getting the data you actually need. A few more months at the drawing board, but now it works again.

Then, one day, your boss tells you that to save on WAN costs, the company wants remote offices to access the contact management application over the Internet…

Getting the picture? Business needs change. Unless you want to redevelop that application every time something changes, you need to use a more flexible design in your application. Although, like OOP, n-tier architecture isn't the magic bullet to solve all problems, it makes for a better design because it relies heavily on components that can be swapped in and out as business needs change.

What is N-Tier?

To help understand application design, it's often useful to think in layers. The pieces that the user sees and interacts with can be called the user interface (UI) or presentation layer. The code that determines which records are valid and how data should be organized can be called the business layer ("business" because this code usually implements what are called "business rules", such as not allowing a new invoice if the customer is over their credit limit). The code dealing with reading from and writing to the data store can be called the data layer.

The applications we've all written in the past had all these layers, but the layers were usually hidden by the fact that everything was in one place. You'd create a form with controls bound to the fields in a table. The Valid method of a textbox ensures only valid data is entered for that field. The Previous and Next buttons do a SKIP –1 and SKIP, respectively, in the table, then call the Refresh method of the form to update the controls. When the user clicks on the Save button, the code in the Click method of the button makes sure the information is correct (for example, the postal code hasn't been left blank), then uses TABLEUPDATE to update the table. Of course, like most smart developers, you didn't put all this code in every form, but instead created a maintenance form class that handled all the common tasks, so each form created from it just had the necessary code (often just the business rules) and controls that made that form unique.

The problem with this design is that everything is bundled together. What if you want to import some records from another source rather than typing them in? You still need to do data validation, so you end up doing one of two things: running the maintenance form with the NOSHOW option and calling its validation method (ugh!), or copying and pasting that code from the form into the import PRG (double ugh!). A better approach (obviously) is to pull the validation code out of the form and put it into its own module (PRG, class, etc.). Now, you can call that module from anywhere that needs to validate the data. The validation rules have been separated from the UI.

The same thing holds true for data access. In the contact management scenario, every time the data store changed, the form had to almost completely be redone, because every aspect of it (controls and code) was intimately bound to the data access mechanism. Our import program would also have to be rewritten, since it can't use APPEND FROM for Pervasive data, for example. Instead, what if we create a data access component that worries about the details like how to get a record, how to save a record, etc., and presents a common interface (programmatic, not visual) to anything that needs access to the data? Should this component be the same one that implements the data validation? No, because notice that even though the data access mechanism changed a lot in our scenario, the business rules stayed fairly constant.

What we've done is separate the UI, data validation (or business rules), and data access layers into separate components. That's the basics of n-tier design.

Why the "n"? This used to be called 3-tier because of the three layers involved, but as applications get more complex, sometimes more than three layers are involved. For example, although credit card processing would obviously fall into the business layer, it might be handled by a different component (even on a different machine), so what we think of as the business tier really consists of several layers. So, the term "n-tier" is used to generically describe an architecture that consists of multiple layers.

Misnomers

N-tier is all about design and architecture, not implementation. This means that:

- N-tier doesn't necessarily involve COM, DCOM, or COM+. Although Microsoft's marketing literature for Windows DNA would lead to believe otherwise, n-tier just means that individual duties are split into components that talk to each other. COM is a great mechanism for inter-process communication, but n-tier is still a viable solution even if you're just developing a VFP application that won't ever expose services to VB, Excel, or IIS (of course, you should strongly resist the temptation to use the famous last words "we'll never need to …" <g>).
- N-tier doesn't necessarily mean multi-language or multi-platform. Not every application will have a VB or DHTML front-end. Even Internet-based applications could use a fat VFP app on the client site.
- N-tier doesn't necessarily mean distributed (that is, different components running on different computers). You could create an application where everything, including the data store, is on the same machine, and it would still make sense to use an n-tier architecture.

Advantages of N-Tier

There are a lot of good reasons to use n-tier architecture in your designs:

- Maintainability: it’s much easier to maintain, debug, and deploy components than it is to change monolithic applications. “Let’s see, was that validation code in the Valid method of the textbox, in the Click method of the Save button, in the Save method of the form, in the Save method of the form class, in the application object …”.

- Reusability: the Holy Grail of programming can actually be reached (or at least we’re closer to it <g>) when components are used. As I mentioned earlier, if the business rules are placed into their own component, that component can be used by anything that needs to validate data: a VFP form, an import PRG, an Excel spreadsheet, a VB app, etc.

- Flexibility: when business needs change, the ability to simply swap an old component for a new one is very compelling. In the contact management scenario, if an n-tier architecture was used, changing data stores would simply mean swapping the data access component for a different one. The business and UI layers wouldn’t change (OK, that may be a little simplistic, but certainly the impact on those layers is much less than what happens in a monolithic application).

- Scalability: IBM had a great TV commercial for their e-business solutions. Three or four kids run up to an ice cream truck, and the vendor happily serves them. Then lots more kids come. Soon, he’s surrounded by thousands of kids. He doesn’t have the resources to service them. We certainly want to avoid that problem with our applications. While you can put a monolithic application on a bigger box as demand increases, what if it increases a thousand-fold (as may be the case for a Web app)? The fastest machine in the world may not help because everyone is trying to hit the same machine simultaneously. Also, if you’re using a database licensed by the connection, as more users come online, the application costs go way up. Using components allows an implementation in which different pieces run on different machines (or even multiple copies of the same component on different, load-balanced servers), and database connections are pooled so users are only attached to the database when they’re actually getting or storing data.

- Distributability (there actually isn’t such a word, but you know what I mean <g>): it’s pretty much impossible to create a Web application without using components because by definition, different parts of the application reside on different machines. Also, it’s much easier to update a component on a server than an entire app on every client’s desktop.

Where Does VFP Fit?

Since most everything can communicate via COM these days, why not bow to marketing pressure and use VB to create our middle-tier objects? I believe that VFP’s strengths – incredibly powerful and fast native database engine, extremely fast string processing, rich language – make it an even better candidate for middle-tier components that for the UI layer, where it definitely lags behind VB in things like API access, ability to create ActiveX controls, and ability to respond to events.

Here are a few reasons why VFP makes a great middle-tier object:

- The key to developing generic code is the use of meta data. You definitely don’t want to hit SQL Server every time you need to know the structure of a table, so keeping that meta data in local storage makes a lot of sense. The best possible place to store meta data is in VFP tables because of VFP’s speed and language.

- Middle-tier components often need to massage data before sending it to the UI layer or to the data layer. What’s better than a language that was designed from the ground up around data processing commands and functions?

- VFP is orders of magnitude faster than VB in string processing. I’m constantly amazed at how fast VFP can process large amounts of text. I once heard a story about a development team that rewrote a component, originally developed in VB that did a lot of string parsing and processing, in VFP and saw a hundred-fold improvement in performance.

Designing the Layers

The biggest challenge for VFP developers in designing n-tier applications is learning to not put any data access or business rules into the UI, something that's been ingrained in us over the years. The UI should have UI stuff only. No processing, no business rules, no touching data directly. One technique that I use to help me keep this straight is to ask the question: "what if I want to use this programmatically?" I should be able to do anything in a VFP PRG that I could do in a form (with the exception of seeing the data presented nicely, of course). If I can't, there's either business or data stuff in the UI that doesn't belong. In fact, I usually write a PRG that tests the business and data classes *before* creating the UI so I know it'll work.

Let's take a look at how the layers are designed.

Data Layer

The data layer is the only one that talks to the data store. It's the only one that knows details like where and how the data is stored, how to access it, whether or not to use transactions, whether tables are accessed directly or only through stored procedures, etc. This provides security (if a client of the data layer doesn't know where the data is located, it can't go and mess things up), flexibility (you can totally rewrite the data layer to change from direct table access to using only stored procedures, and the client components won't know the difference), and scalability (only the data layer has a connection to the database, and that connection might only be maintained for as long it is takes to do a single task, which helps minimize concurrent connections).

How data is read from and written to the data store, and how the data layer transfers data to and from client components, is a design decision you'll have to make based on the data store used, company policies (such as data can only be accessed through stored procedures), and the needs and abilities of the client components. ADO is typically used on the client component side (using disconnected, client-side recordsets), since it provides a language-independent, object-oriented interface to data. However, XML is becoming more popular and may be the mechanism of choice in the future. I know people who've created a data object that exposes fields in a table as properties of the object, but you should resist this temptation because it's not scalable at all; if the data object is instantiated on a server using DCOM, every access to a "field" will require a round-trip to the server to get the value of the property. On the data store side, direct access might be used if the data store is VFP tables, ODBC might be used (either remote views or SQL passthrough), or it could be ADO or XML here as well.

Other design decisions: you can create a single object that deals with all the data, one object per "entity" (such as customer or order), or something in between. You can have a generic GetData method that must be told what data to get or specific methods like GetCustomerData and GetOrderData that know exactly what to do.

Business Layer

The business layer acts a middle-tier between the UI layer and the data layer. It typically implements business rules, such as "the contact name can't be left blank" and "you can't submit an order if the customer is over their credit limit". It's the only thing that talks to the data layer, so the UI can't talk to the data layer directly and get around the business rules. The business layer should know nothing about how or where the data is stored or accessed, only attributes of the data itself (such as field names, data types, sizes, permissible values, etc.)

ADO is often the data mechanism of choice for communicating with the data layer, but as I mentioned earlier, that may ultimately be supplanted by XML. How data is sent to the UI layer depends on the needs and capabilities of that layer; ADO, RDS, and XML are all possibilities, but a specialized business object that creates a VFP cursor could also be used under special circumstances.

As with data objects, you can create a single business object that contains all the business rules, one object per entity, or something in between.

UI Layer

The UI, or presentation, layer can be anything that presents a UI to the user and talks to the business layer. VFP or VB forms, Excel documents, and HTML or DHTML documents (usually served up by ASP or some other Web server process) are all typical clients. In our contact management scenario, one client might be a VFP form that's part of an app used by local users, and another might be an HTML page displayed in a browser at remote offices. Both of these clients get a record from the customers table via the business object, display it, and allow the user to edit it and save the changes. The business object will validate the data and either inform the UI client that there's a problem with the data or submit it to the data object to update the data store.

Contacts Design

Let's redesign our contact management application using an n-tier architecture. Some people in our organization will use a VFP application for data entry, while others, because they're spreadsheet jockeys, will use Excel for data querying. Although I didn't create one, an ASP page would certainly be another useful example using this design.

Figure 1 shows the model for this design. UI Object is a placeholder for any UI layer object; it has references to a business object called Customer and an ADO RecordSet. Customer is a subclass of BusinessClass, an abstract business object, and provides the business layer for customer records (another business object, Orders, isn't shown in this diagram). BusinessClass has a reference to a data class called DataClass, which provides data layer services. DataClass maintains a reference to an ADO Connection object.

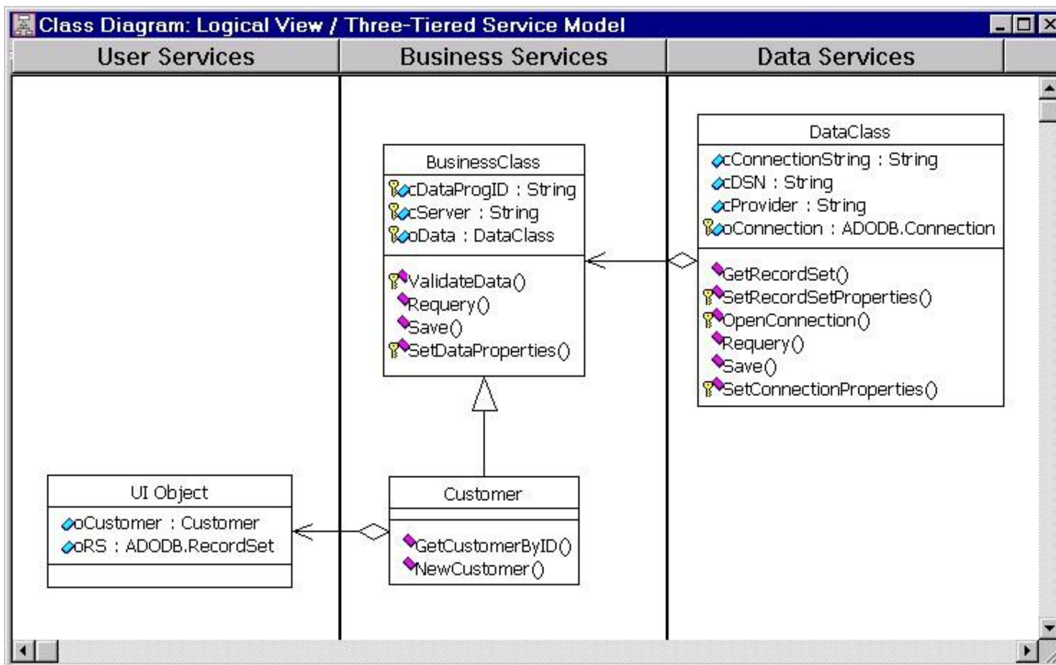


Figure 1. Object model for Contacts.

This model consists of the following tiers:

- UI: this could be a VFP form, Excel spreadsheet, ASP page, etc.
- Business: implements business rules for customers and orders and acts as a middle tier between the UI and data layers.
- Data: provides data handling services. This actually consists of several sub-tiers: DataClass (the data object), ADO (for data access), and the DBMS which actually stores the data (the Jet engine for this example).

Let's look at these tiers in more detail. The samples that accompany this session use VFP 7 because of COM improvements in that version (such as strong typing and PEM attributes); however, the VFP6 subdirectory contains a VFP 6 version of these samples if you don't have VFP 7 installed.

COMBase

Before we look at the data and business layers, let's briefly examine the COMBase class, which the other classes we'll see are based on. This simple class is a subclass of Session because it's lightweight, has a private datasession, and, in VFP 7, exposes only custom properties and methods in the type library so it has a small COM footprint. COMBase has a Release method and implements an error handling scheme that displays no UI. Instead, ErrorMessage contains the error message and ErrorOccurred is .T. if an error occurred. The ResetError method clears the error information. Despite its name, this class isn't marked OLEPublic since, as we'll see later, not all subclasses may be COM objects, and we certainly don't want this abstract class to be a COM object.

Data Layer

DataClass (in LAYERS.PRG) is based on COMBase and uses ADOVFP.H as its include file. This class uses ADO as the mechanism to marshal records between itself, clients, and data sources. It keeps a reference to an ADO Connection object so it can talk to the data source. However, it doesn't maintain any reference to a RecordSet object; instead, the client will pass a RecordSet to work with to various methods of DataClass.

DataClass is marked OLEPublic (so it's a COM object) because although it can be subclassed for additional behavior, it's a useable class by itself.

The Init method simply creates an ADO Connection object into the oConnection property:

```
This.oConnection = createobject('ADODB.Connection')
```

The Destroy method closes and destroys the connection:

```
with This
    if .oConnection.State = adStateOpen
        .oConnection.Close()
    endif .oConnection.State = adStateOpen
    .oConnection = .NULL.
endwith
```

The protected OpenConnection method is responsible for opening the connection to the data source if it isn't already open. In this class, there are two ways you can specify what data source to open: by setting the cDSN property to an ODBC data source name, or by setting the cProvider and cConnectionString properties to an OLE DB provider name and connection string, respectively. OpenConnection calls the protected SetConnectionProperties method to set some properties of oConnection (this method could be overridden in a subclass to set additional properties), then opens the connection and returns .T if it succeeded.

```
local llReturn
with This

* If the connection isn't already open, open it.

    if .oConnection.State <> adStateOpen

* Set properties of the connection object first.

        .SetConnectionProperties()

* If we aren't using a DSN, the Provider and
* ConnectionString properties of the connection
* object have been set, so open the connection.
* Otherwise, use the DSN to open the connection.

        if empty(.cDSN)
            .oConnection.Open()
        else
            .oConnection.Open(.cDSN)
        endif empty(.cDSN)
```

```
* Return .T. if the connection was opened.

        llReturn = .oConnection.State = adStateOpen
    else
        llReturn = .T.
    endif .oConnection.State <> adStateOpen
endwith
return llReturn
```

GetRecordSet fills an ADO RecordSet object with records using the specified command. Rather than creating and returning a RecordSet itself, it accepts the RecordSet to work with as a parameter. GetRecordSet first ensures the connection is open, then cancels any pending changes and closes the RecordSet if it was open. It then connects the RecordSet to the data source, calls the protected SetRecordSetProperties method to set some properties of the RecordSet (this method could be overridden in a subclass to set additional properties), and opens the RecordSet. It then disconnects the RecordSet and returns .T. if the RecordSet was successfully opened.

```
lparameters toRS, ;
    tcCommand
local llReturn
with This

* Open the connection if necessary. If we succeeded,
* carry on.

    if .OpenConnection()
        .ResetError()

* If the recordset is already open, close it.

        if toRS.State = adStateOpen
            toRS.CancelBatch()
            toRS.Close()
        endif toRS.State = adStateOpen

* Set properties of the recordset, then open it.

        toRS.ActiveConnection = .oConnection
        .SetRecordSetProperties(toRS)
        toRS.Open(tcCommand)

* Disconnect the recordset.

        toRS.ActiveConnection = .NULL.
        llReturn = toRS.State = adStateOpen

* We couldn't open the connection, so return .F.

    else
        llReturn = .F.
    endif .OpenConnection()
endwith
return llReturn
```

The Save method updates the data source with changes in the passed RecordSet. Like GetRecordSet, it first ensures the connection is open, then determines how to save the RecordSet; if the RecordSet uses batch optimistic locking, the update is wrapped in a transaction. The RecordSet is connected to the data source and Update or UpdateBatch is used to update the data source. If the update fails, Cancel or CancelBatch is used to undo the changes. Then the RecordSet is disconnected again and a value indicating success or failure is returned.

```
lparameters toRS
local llReturn
with This

* Open the connection if necessary. If we succeeded,
* reset the error flag and determine how to save the
* recordset.

    if .OpenConnection()
        .ResetError()
        do case

* If we have a batch optimistic recordset, start a
* transaction, do a batch update, and either commit or
```

```
* rollback as necessary.

            case toRS.LockType = adLockBatchOptimistic
                .oConnection.BeginTrans()
                toRS.ActiveConnection = .oConnection
                toRS.UpdateBatch()
                if .ErrorOccurred
                    .oConnection.RollbackTrans()
                    toRS.CancelBatch()
                else
                    .oConnection.CommitTrans()
                endif .ErrorOccurred
                toRS.ActiveConnection = .NULL.

* If we have an optimistic recordset, update it.

            case toRS.LockType = adLockOptimistic
                toRS.ActiveConnection = .oConnection
                toRS.Update()
                if .ErrorOccurred
                    toRS.Cancel()
                endif .ErrorOccurred
                toRS.ActiveConnection = .NULL.

* We don't have an updatable recordset, so raise an
* error.

            otherwise
                error 'The recordset is not updatable.'
        endcase
        llReturn = not .ErrorOccurred
    else
        llReturn = .F.
    endif .OpenConnection()
endwith
return llReturn
```

The Delete method deletes the current record in the passed RecordSet. The reason this method is required rather than just calling the Delete method of the RecordSet is that unless the RecordSet's LockType property is set to optimistic batch locking, RecordSet.Delete requires an open connection to delete the record, and fails to do so, even if RecordSet.Update is called, if the connection isn't open.

```
lparameters toRS
with toRS
    .ActiveConnection = This.oConnection
    .Delete()
    .ActiveConnection = .NULL.
endwith
llReturn = not This.ErrorOccurred
return llReturn
```

The Requery method (we won't look at it here) connects to the data source and requeries the passed RecordSet.

Business Layer

BusinessClass (in LAYERS.PRG) is based on COMBase. This class doesn't know anything about data; it has no connection to any data source and doesn't maintain a reference to a RecordSet. It's simply a go-between for clients and the data layer to ensure that business rules are satisfied. It keeps a reference to a DataClass object.

Because it must be subclassed to implement the exact business rules needed, BusinessClass is not marked OLEPublic (so it's not a COM object).

The Init method instantiates a DataClass object into its protected oData property. The ProgID for the object is stored in the cDataProgID property; if the object exists on a different machine (that is, the object should be instantiated using DCOM), the cServer property should contain the name of the server. After instantiating the object, the SetDataProperties method is called to set properties of the data object. This method could be overridden in a subclass to, for example, tell DataClass what data source it should connect to if you don't want to create a subclass specifically for a given data source.

Although the Init method of COM objects can't receive parameters, if this class is instantiated as a VFP class (usually for debugging purposes), it'll accept a ProgID for the data class. That way, although cDataProgID may be set to something like "Contacts.DataClass", you can pass "DataClass" to have that class instantiated as a VFP object. This makes it possible to debug both the business and data objects in VFP, which is next-to-impossible to do if they're instantiated as COM objects.

```
lparameters tcProgID
local lcDataProgID
with This

* Although the Init method of COM objects can't receive
* parameters, if this class is instantiated as a VFP
* class (usually for debugging purposes), we'll accept a
* ProgID for our data class.

    lcDataProgID = iif(vartype(tcProgID) = 'C' and ;
        not empty(tcProgID), tcProgID, '')

* Instantiate the data object we're working with and set
* any properties we need.

    do case
        case not empty(lcDataProgID)
            .oData = createobject(lcDataProgID)
        case empty(.cServer)
            .oData = createobject(.cDataProgID)
        otherwise
            .oData = createobjectex(.cDataProgID, ;
                .cServer)
    endcase
    .SetDataProperties()
endwith
```

The Save method accepts a RecordSet object and calls the protected ValidateData method (abstract in this class) to ensure the data is valid. If so, the Save method of the data object is called to do the actual work of updating the data source. If that method failed, the data object's ErrorMessage property is put into this object's property (so a client can see what went wrong). The method returns a value indicating success or failure.

```
lparameters toRS
local llReturn
with This
    .ResetError()
    do case

* Validate the recordset. If it failed, return .F.

        case not .ValidateData(toRS)
            llReturn = .F.

* Have the data object save the recordset. If it failed,
* get the error message and return .F.

        case not .oData.Save(toRS)
            .ErrorMessage  = .oData.ErrorMessage
            .ErrorOccurred = .T.
            llReturn       = .F.

* Everything worked, so return .T.

        otherwise
            llReturn = .T.
    endcase
endwith
return llReturn
```

The Delete method accepts a RecordSet object and simply passes the object on to the Delete method of the data object. Additional work could be done in a subclass.

```
lparameters toRS
local llReturn
with This
    llReturn = .oData.Delete(toRS)
    if not llReturn
```

```
        .ErrorMessage  = .oData.ErrorMessage
        .ErrorOccurred = .T.
    endif not llReturn
endwith
return llReturn
```

The Requery method (we won't look at it here) simply passes the RecordSet to the Requery method of the data object.

That's it for code in this class. You'll need to subclass it to implement specific methods and business rules for your requirements. Let's look at a couple of subclasses: Customer and Orders.

Customer (in CONTACTS.PRG) is designed to work with records from the Customers table in the Northwind sample database that comes with Visual Studio. It's marked as OLEPublic because it's a COM object. cDataProgID is set to Contacts.DataClass so it uses the DataClass data object built into the same DLL. The SetDataProperties method sets the provider and connection string for the data object.

```
with This
    .oData.cProvider         = 'Microsoft.Jet.OLEDB.4.0'
    .oData.cConnectionString = 'Data Source=D:\' + ;
        'Program Files\Microsoft Visual Studio\VB98' + ;
        '\Nwind.mdb'
endwith
```

For demo purposes, I put code into ValidateData to not allow a fax number of 999-999-9999; obviously, a real class would have real rules in this method.

```
lparameters toRS
local llReturn
do case
    case toRS.Fields('Fax').Value = '999-999-9999'
        error 'The fax number cannot be 999-999-9999.'
        llReturn = .F.
    otherwise
        llReturn = .T.
endcase
return llReturn
```

I added a couple of new methods to this class. GetCustomerByID accepts a RecordSet and an ID value, and uses the GetRecordSet method of the data object to fill the RecordSet with all fields from the Customers table for the specified customer number. It returns .T. if the record was found, and .F. if not or if something went wrong (if it returns .F., ErrorMessage will be empty if it simply didn't find a matching record).

```
lparameters toRS, ;
    tcID
local lcCommand, ;
    llOK
with This
    .ResetError()
    lcCommand = "select * from Customers where " + ;
        "CustomerID = '" + tcID + "'"
    llOK = .oData.GetRecordSet(toRS, lcCommand)
    if llOK
        llOK = not toRS.EOF
    else
        .ErrorMessage  = .oData.ErrorMessage
        .ErrorOccurred = .T.
    endif llOK
endwith
return llOK
```

NewCustomer accepts a RecordSet and puts an empty customer record into it. It uses GetCustomerByID, passing it a non-existent key value, so the RecordSet has the correct structure, then uses the AddNew method of the RecordSet to create the new record.

```
lparameters toRS
local llReturn
This.GetCustomerByID(toRS, '-1')
llReturn = not This.ErrorOccurred
if llReturn
    toRS.AddNew()
    llReturn = not This.ErrorOccurred
endif llReturn
return llReturn
```

Orders (in CONTACTS.PRG) is designed to work with records from the Orders and Order Details tables in the Northwind sample database. Like Customer, it's marked OLEPublic, cDataProgID is set to Contacts.DataClass, and SetDataProperties sets the provider and connection string for the data object (although it uses the MSDataShape provider since this class creates hierarchical RecordSets).

This class has one custom method, GetOrdersForCustomer, which accepts a RecordSet and a customer ID and fills the RecordSet with a hierarchical cursor of orders and order details information. Don't worry about the syntax of the command to get the data; it uses Shape syntax which is beyond the scope of this session.

```
lparameters toRS, ;
    tcCustomerID
local lcCommand, ;
    llOK
with This
    .ResetError()
    lcCommand = 'shape {select OrderID, OrderDate, Freight from ' + ;
        "Orders where CustomerID = '" + tcCustomerID + "'} " + ;
        'as Orders append ({select OrderID, ProductName, ' + ;
        '"Order Details".UnitPrice, Quantity, ' + ;
        '"Order Details".UnitPrice * Quantity as LineTotal from ' + ;
        '"Order Details" inner join Products on ' + ;
        '"Order Details".ProductID = Products.ProductID} ' + ;
        'as OrderDetails relate OrderID to OrderID) as OrderDetails'
    llOK = .oData.GetRecordSet(toRS, lcCommand)
    if llOK
        llOK = not toRS.EOF
    else
        .ErrorMessage  = .oData.ErrorMessage
        .ErrorOccurred = .T.
    endif llOK
endwith
return llOK
```

UI Layer

To show how the business class layer can be used with any kind of UI layer, I created four examples of different clients. The first one, TESTBUSINESS.PRG, does everything in code and only displays results through MESSAGEBOX dialogs. It first gets and displays the record for the ALFKI customer:

```
oBusiness = createobject('Contacts.Customer')
oRS = createobject('ADODB.RecordSet')
oBusiness.GetCustomerByID(oRS, 'ALFKI')
messagebox('Found customer ALFKI: ' + ;
    trim(oRS.Fields('CompanyName').Value))
```

Next, it tries the change the fax number to 999-999-9999, which isn't allowed by the business object so an error is displayed:

```
oRS.Fields('Fax').Value = '999-999-9999'
llOK = oBusiness.Save(oRS)
if not llOK
    messagebox('Cannot save change to customer ' + ;
        'ALFKI:' + chr(13) + chr(13) + ;
        oBusiness.ErrorMessage)
endif not llOK
```

It then creates a new customer and saves it:

```
oBusiness.NewCustomer(oRS)
oRS.Fields('CustomerID').Value  = 'STONE'
oRS.Fields('CompanyName').Value = ;
    'Stonefield Systems Group Inc.'
oBusiness.Save(oRS)
```

Finally, it deletes the new record:

```
oRS.Delete()
oBusiness.Save(oRS)
```

The second example is a VFP form, CONTACTS.SCX. This form has a pageframe with two pages. The first page has controls bound to fields in the RecordSet referenced in the oRS property of the form (for example, txtCompanyName has "Thisform.oRS. Fields('CompanyName').Value" as its ControlSource). The second page has a Microsoft Hierarchical

FlexGrid which displays the hierarchical orders data in the oOrderRS RecordSet. The Load method of the form instantiates the oRS and oOrderRS RecordSets as well as Customer (oCustomer) and Orders (oOrders) business objects. Note that in this case, these objects are instantiated as VFP classes rather than COM objects just to show how debugging can be done. The Valid method of txtCustomerID uses oCustomer.GetCustomerByID to fill oRS with information for the specified customer and oOrders.GetOrderForCustomer to fill oOrderRS with information about orders for that customer. If the customer isn't found and you choose "Yes" when asked if you want to create a new customer, the NewCustomer method of the form creates an empty RecordSet by calling oCustomer.NewCustomer. Save and Cancel buttons allow you to save or cancel changes.

Figure 2. CONTACTS.SCX shows using the business objects in a VFP form.

The next example is an Excel spreadsheet, ORDERS.XLS, that displays the orders for the specified customer and calculates totals. The GetOrders macro, fired when you click on the Orders button, instantiates RecordSet and Contacts.Orders objects and uses the GetOrdersForCustomer method to fill the RecordSet with orders for the customer number entered into cell B1. A loop then displays the contents of the RecordSet in the spreadsheet, totaling each order as it goes.

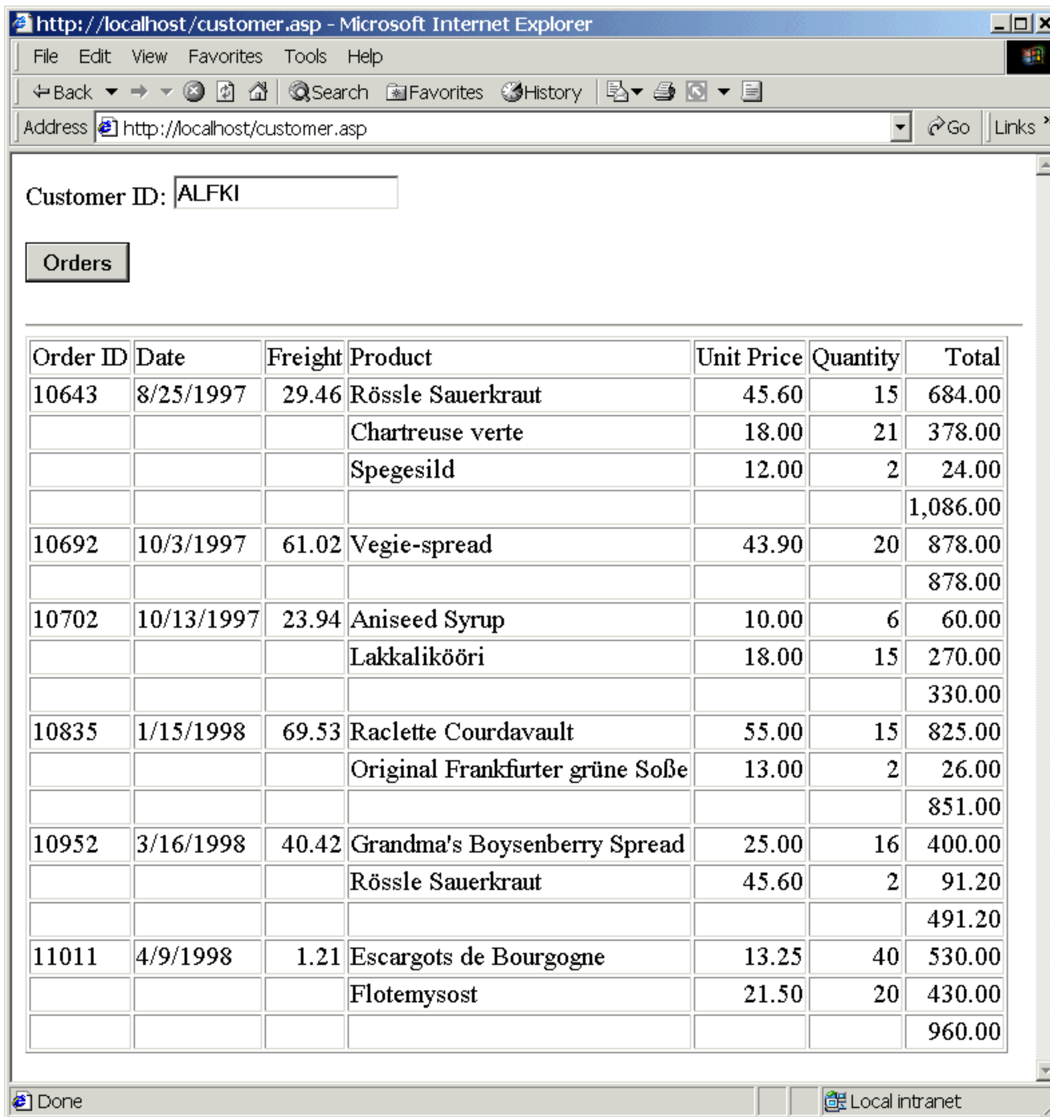
orders.xls

	A	B	C	D	E	F	G
1	**Customer #:**	ALFKI	Orders	Clear			
2							
3	**Order ID**	**Date**	**Freight**	**Product**	**Unit Price**	**Quantity**	**Total**
4	10643	08/25/1997	$29.46	Rössle Sauerkraut	$45.60	15	$684.00
5				Chartreuse verte	$18.00	21	$378.00
6				Spegesild	$12.00	2	$24.00
7							$1,086.00
8							
9	10692	10/03/1997	$61.02	Vegie-spread	$43.90	20	$878.00
10							$878.00
11							
12	10702	10/13/1997	$23.94	Aniseed Syrup	$10.00	6	$60.00
13				Lakkalikööri	$18.00	15	$270.00
14							$330.00
15							
16	10835	01/15/1998	$69.53	Raclette Courdavault	$55.00	15	$825.00
17				Original Frankfurter grün	$13.00	2	$26.00
18							$851.00
19							
20	10952	03/16/1998	$40.42	Grandma's Boysenberry	$25.00	16	$400.00
21				Rössle Sauerkraut	$45.60	2	$91.20
22							$491.20
23							
24	11011	04/09/1998	$1.21	Escargots de Bourgogne	$13.25	40	$530.00
25				Flotemysost	$21.50	20	$430.00
26							$960.00

Sheet1 / Sheet2 / Sheet3

Figure 3. ORDERS.XLS shows using the business objects in Excel.

The final example is an ASP page, CUSTOMER.ASP, that essentially does the same thing as ORDERS.XLS, but in a Web page instead.



Order ID	Date	Freight	Product	Unit Price	Quantity	Total
10643	8/25/1997	29.46	Rössle Sauerkraut	45.60	15	684.00
			Chartreuse verte	18.00	21	378.00
			Spegesild	12.00	2	24.00
						1,086.00
10692	10/3/1997	61.02	Vegie-spread	43.90	20	878.00
						878.00
10702	10/13/1997	23.94	Aniseed Syrup	10.00	6	60.00
			Lakkalikööri	18.00	15	270.00
						330.00
10835	1/15/1998	69.53	Raclette Courdavault	55.00	15	825.00
			Original Frankfurter grüne Soße	13.00	2	26.00
						851.00
10952	3/16/1998	40.42	Grandma's Boysenberry Spread	25.00	16	400.00
			Rössle Sauerkraut	45.60	2	91.20
						491.20
11011	4/9/1998	1.21	Escargots de Bourgogne	13.25	40	530.00
			Flotemysost	21.50	20	430.00
						960.00

Figure 4. CUSTOMER.ASP shows using the business objects in an ASP page.

Summary

As developers, we need to move away from putting all our eggs (UI, business rules, and data access) into one basket (form, PRG, etc.). Even applications that serve a small number of users on a LAN can benefit from an n-tier architecture, and will certainly be much easier to maintain, scale up, or deploy as an Internet application in the future. The biggest challenge for VFP developers in designing n-tier applications is learning to not put any data access or business rules into the UI, something that's been ingrained in us over the years.

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of the award-winning Stonefield Database Toolkit (SDT). He writes the monthly "Reusable Tools" column for FoxTalk, and is the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at Microsoft FoxPro Developers Conferences (DevCon) since 1997, as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP).

Copyright © 2001 Doug Hennig. All Rights Reserved

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
Web: www.stonefield.com