PINNACLE

FoxTalk

Solutions for Microsoft® FoxPro® and Visual FoxPro® Developers

Turn Your VFP App Client/Server: A 12-Step Program, Part 2

Jim Falino

WIN 3.0 5.0 6.0 DOWNLOAD

Last month, Jim began the first of this two-part series on the steps you need to take to develop client/server applications with Visual FoxPro (see Part 1 in the April 1999 issue of *FoxTalk*). Jim raises six more important issues this month and concludes with a sample form that can access both VFP and SQL Server tables.

I have a lot of information I'd like to pass along, so please see Part 1 for an extended introduction. Let's get right to it. One reminder: All examples are from the SQL Server 7.0 Northwind Traders database.

Step 7: Use surrogate primary keys for all tables

Even before VFP 3.0 gave us the ability to add primary keys to our tables, using a field—or group of fields—to uniquely identify a table row had been suggested by many to be a sound practice. In file-server applications, though, in many situations there really was no requirement to do this—it was only a good idea. However, with views, the rules change.

In order to use updateable views, there must exist a primary key on the table on which the view is based (I typically don't update data from a multi-table view). The reason is that since the view creates a local result set when opened (USE <Viewname>), you need some mechanism to be able to find the underlying record on the server that you're attempting to update. This wasn't always necessary in a file-server model, whereby a shared record could be directly edited.

In order for updates to work, views require that you either set the KeyField property at design time in the View Designer or with

Continues on page 4

May 1999

Volume 11, Number 5

6.0 Applies to VFP v6.0 (Tahoe) | 5.0 Applies to VFP v5.0 | 3.0 Applies to VFP v3.0 | 2.x Applies to FoxPro v2.x

DOWNLOAD

Accompanying files available online at http://www.pinpub.com/foxtalk

UNIX MAC DOS WIN

Applies specifically to one of these platforms.

Certification: Should You Care?

Whil Hentzen

HERE'S a trick question: If you were interviewing candidates for a development job, and you had to choose between someone with 15 years of experience and absolutely no credentials and a 25-year-old with a year of experience and six certifications, who would you hire?

Witness these messages I've received over the past couple of months.

"I wonder what the VFP 6 exam will be like. I laughed out loud when the Visual FoxPro 3.0 exam came out—only two questions out of 100 or more had 'visuals' along with the question, and they both showed screen shots of the Menu Builder—the one part of *Visual* FoxPro that: a) hadn't changed a bit from 2.x to 3.0; and b) was probably the least 'visual' tool in all of VFP. What were the test designers thinking of?"

"I took the VB 5.0 exam on a lark last year when my free coupons ran out. Didn't study at all. Passed five of the eight sections with 100 percent, never heard of a dynaset, and just missed passing."

"I had a voucher for a free exam that expired shortly, and since the VFP test wasn't out yet, I decided to take the VB test. Of course, I waited until the weekend before (the test was Monday), and you know the rest—well, except the result of the test. I scored 850+ out of 1,000, where 714 is required to pass. So I went from a regular guy who's never developed a VB application to a MS Certified VB6 Developer who's never developed a VB application."

Obviously, there's a danger with certifications. First of all, of course, is that certification only measures what's on the test—and, thus, your ability to take a test. You all know some people who weren't all that smart, but got through school because they were really good at taking tests. And there were, of course, those for whom the opposite applied <s>.

Second, a test doesn't necessarily measure what the tool is actually used for in the real world—just what the developer of the test thinks should be on the test. Microsoft has often tilted their tests toward their view of what they want their tool to be used for—never mind the current work-a-day world where you have to deliver apps for customers.

Third, it doesn't consider what you know about developing applications. I've found it ironic to have the titles for the various certifications include the word "Developer"—because a developer's skill set must include a wide range of abilities that don't have anything to do with syntax and hands-on keyboarding. You could ace every test offered by Microsoft and Oracle combined and still not be able to normalize your way out of a paper bag or write a specification longer than half a sheet of paper.

It's much like learning to drive a nail with a hammer, and slicing up some wood with a saw. Just because you're certified with a hammer and saw doesn't mean you're qualified to build a house, does it? But it *does* mean that you can distinguish between the two, and you have a basic idea of what each is used for.

So, if you're a developer, understand what you're going to get out of becoming certified. You'll get a foundation of knowledge and some understanding of the breadth of the product—as Microsoft's test creators view it—but it doesn't mean you're an expert. View certification, as one of my friends put it, "as a driver's learning permit. You've learned enough to read the manuals, and you have some idea of what Microsoft thinks its tool is used for, but now you need to go learn how to really use it."

If you're a development firm, certification is a marketing tool you can use with potential customers. You can explain that your developers have a consistent footing in terms of foundation knowledge, instead of happenstance knowledge picked up randomly here and there. It means that your firm cares enough about the education of its developers that you were willing to foot the bill for the certifications, whatever that means in your particular situation—whether you paid for the exams, gave employees time off, or whatever. And it means that you've got a never-ending supply of CDs, newsletters, magazines, and other information from Microsoft—so you're well-informed and are staying on the leading edge of the learning curve.

If you're an employer or hiring a development firm, the next developer or shop that comes in with a bunch of certifications is worth looking at, because it means they had enough initiative to get certified, but it doesn't mean you can turn off your judgment meter. You still need to evaluate the developer as you would any other potential candidate. Use certification credentials as a tie-breaker between two otherwise equal candidates.

And the answer to my trick question? "You don't have enough information to make a decision." You can't let a resume do your thinking for you, and you need to understand what certification means and doesn't mean. ▲

12-Step Program . . .

Continued from page 1

DBSETPROP("Viewname.KeyField1", "Field", "KeyField", .T.), or set the KeyFieldList property on the open cursor at runtime using CURSORSETPROP ("KeyFieldList", <comma-delimited list of primary key fields>).

In either case, the view will use the key field(s) to find the record on the server you've requested to SQL Update or SQL Delete. The keys will also be used to detect update conflicts—reporting back an error if anyone has either deleted the record you're trying to update or has changed the key. Aside from being a necessity for updateable views, adding primary keys to your tables also makes querying for one particular record simpler. You'll find that in the client/server world, you'll often need that ability.

The argument for surrogate keys

If you do have multiple fields that make up your primary keys, I suggest using a system-generated surrogate key in addition. Having this field, typically an integer type, will give you the luxury of a unique Row ID that you'll be glad you have. Along with this, for child tables you'll want to add one more integer field to serve as the foreign key. You'd populate it with its parent's surrogate primary key before saving a new child.

The benefits of surrogate keys include faster joins, faster updates, faster deletes, and the simplified retrieval of the children of a parent. The only drawback is generating them. I use a system table of next numbers—it contains one row per table in the application—but this can cause multi-user contention problems if you're not careful. Since these keys really are meaningless (that's why they're also called abstract keys), you shouldn't attempt to get the next sequential number from the server while in the midst of a transaction. This might create too much contention on the system table when you need primary keys for a highly active table. It's better to get the key outside of the transaction and risk losing it if an update fails.

One other thought might be to use one of several techniques to generate a unique ID locally—eliminating the trip to the server and any possible multi-user contention. Much like the way I never let a contractor leave my house without recommending a good roofer, I never let a conversation with a developer end without asking about their unique ID generation technique. Two interesting ones I've heard that use client-side generation are:

1. Use a GUID. They're 26 characters long, (supposedly) unique across the world, and lightning-fast to generate. They're not beautiful to look at, but it might be well worth the pain.

2. Get a unique ID from a server-side system table on application startup, and then use a local application property as a counter. Concatenate the two to generate a unique alphanumeric string. Although I'm simplifying it a bit, I believe this technique is used in Code Book.

Step 8: Add a timestamp column to every table

Client/server applications almost exclusively use optimistic locking. Given that fact, multi-user contention checking becomes much more of a challenge to implement. You must continually remind yourself while coding, "I might not have the latest version of this record set."

Although the primary key is used to detect update conflicts, its ability is limited to detecting whether another user has either deleted the record you're trying to update or has changed the key. The primary key won't help you determine whether another user made changes to non-key fields before you committed your changes.

Typically, client/server applications use some form of timestamp column—one that's re-stamped as part of every update—to indicate the latest version of a record. (I say "some form of timestamp" because the datatype can be almost anything.) This column is then queried on the server during every update to determine whether it conflicts with the version on the local workstation. For example, a VFP view might generate the following SQL statement after updating the customer view:

```
Replace state With 'NY' For city = 'New York' In
vcustomers
TableUpdate(.T., .F., "vcustomers"):


* This is what is auto-generated by the view
* and passed to the server.
* There would be one of these per updated record.
Update customers Set state = 'NY' ;
  Where cust_pkey = ?vcustomers.cust_pkey and ;
  timestamp = ?vcustomers.timestamp
```

If this SQL update statement fails with an Update Conflict error, you can report to your user that someone else changed this record while you were editing it. Where you go from here depends on how nice you are. Ultimately, either a requery must occur or you can try to issue TableUpdate again with the Force parameter set to true (which isn't recommended unless you provide the user with a field-by-field comparison of the changes).

The view property—WhereType—is responsible for the multi-user contention check. You have four options to determine what fields are compared when the contention check portion of the SQL statement is

generated. Key and Modified Fields and Key and Updatable Fields sound like powerful features, but I feel that in most cases these options are too dangerous to implement. (I wouldn't want user 1 and user 2 to update two different columns of the same record without first knowing of each other's changes.)

That leaves Key Fields Only and Key and Timestamp. Key and Timestamp is only supported if your database supports a timestamp-type column. This column is automatically re-stamped by the server on every update. The client application doesn't maintain this field—the view will just include the column in the contention check.

SQL Server has support for a timestamp column. In fact, the SQL Server Upsizing Wizard has an option to create a timestamp column for each table. The column isn't actually a datetime datatype, but rather a system-generated unique string. The beauty of it is that the server automatically manages it and that it's much more accurate than the datetime datatype (you'll never have to worry about two users updating at the same exact time).

Trap!

Some behaviors to be aware of when using the SQL Server timestamp column:

If you don't requery your view after every save, the timestamp column in your local view cursor won't be refreshed with the current value on the server. Thus, subsequent attempts to save the same record will fail with an Update Conflict error when the two timestamps are compared.

Tip!

Workarounds to the shortcomings of the Upsizing Wizard:

The SQL Server Upsizing Wizard has many problems that make it, in my opinion, almost unusable. Most people I know have had to write their own tools to get around the issues. The one saving grace is that the source code for all of the wizards and builders now ships with version 6.0 of VFP. It's really not that tough to hack your way through your stumbling blocks. The source code is located in \<location of VFP 6.0>\Tools\Xsource\Xsource.zip.

So Key and Timestamp sounds great for SQL Server, but what about other back ends that don't support a timestamp column? And how can my prototype with local views against a VFP database work with the same set of code? My workaround is to add your own timestamp column and maintain it from either the client-side or a server-side update trigger. You then make the field a KeyField (see Step 7) and use the Key Fields Only WhereType. So your KeyFieldList might be cust_pkey and timestamp. (Note that the fields in the KeyField or KeyFieldList property of a view don't actually have to be primary or candidate keys.)

You can use a datetime-type column for this purpose. However, the VFP datetime column is only precise to a second, so conceivably two users can update the same record within the same second. If you feel that this is a potential problem for your application, use some other technique to generate a unique number or string. I used this workaround until I upsized to SQL Server and included its timestamp-type column. (Since you have no control over this field name, you might want to call your VFP timestamp column something different so there's no conflict when you upsize.) Then you can just change the WhereType to Key and Timestamp, and you're done.

Step 9: Life without dates

It was a bit unsettling—okay, maybe more like a sudden jolt—to find out that SQL Server, like most big-time databases, doesn't support a date datatype. You must use a datetime-type field instead. If you're converting an existing VFP application that uses date fields, the problems this causes could be quite extensive. It really all depends on how your application uses dates and how much you care about seeing 12:00:00 AM tacked onto the end of your date fields. I've summarized the issues I've been confronted with as well as their possible workarounds.

Controls bound to date fields

In cases where showing the time portion of a datetime field is just plain inappropriate, you obviously have to perform some sort of trickery to lose the default time of 12:00:00 AM. You have several options. You could use DBSetProp to change the datatype of the view definition from datetime to date. (As mentioned in Part 1, having a local DBC of views eliminates any multi-user contention issues.)

```
Create SQL View vOrder Remote Connection ;
   Remote1 As Select * from orders
DBSetProp("vOrders.orderdate", ;
   "Field", "DataType", "D(8)")
Use vOrders
```

Doing so will not only truncate the time portion, but also provide you with a VFP date type field in your view—enabling your date-specific code to work unchanged. And fortunately, when you issue TableUpdate

on a view with a date-type field in it, a default time of 12:00:00:000 AM is automatically appended to it on SQL Server . . . without error.

If you have controls that aren't bound to a view but rather to a read-only SQL Passthrough (SPT) cursor, you can use the back end's data type conversion function to do the truncation. Here's a SQL statement that could be passed to SQL Server that would do just that:

```
SELECT orderid, ;
  customerid, ;
  CONVERT(char(10), orderdate, 101) as orderdate ;
  from orders
```

Note that 101 refers to the optional style argument of the CONVERT function. 1 indicates American format—mm/dd/yy. You then add 100 if you want the century included.

To make the CONVERT function work against local data as well, you might want to create a stored procedure called "Convert" in the database of views. Have it accept the three arguments of SQL Server's CONVERT function and, if passed a datetime-type, return SubStr(TToC(pDateTimeField),1,10). You'd also need to write a Char stored procedure as well to handle the SQL Server function datatype, char(10).

Datetime fields can't be "empty"

As if losing date fields wasn't bad enough, you also must contend with the fact that datetime fields can't be empty on most databases. (There's a whole world of limited functionality beyond FoxPro, isn't there?!) Datetime columns must be populated by a valid date and time or be Null.

If you allow your datetime fields to accept Null, it quickly solves the user-interface problem but probably generates others. You could experience unexpected results by blindly introducing Nulls. (For example, comparing two dates where one is Null returns Null, Empty(NullDate) returns .F., and so on.) But if you're prepared to tackle these issues, Nulls really come in handy when presenting data to users because of VFP's Set NullDisplay To environment command, and the NullDisplay property of controls like the text box.

If you don't want to use Nulls, VFP and SQL Server are actually pretty kind to you. Sending an empty date to SQL Server causes no error. However, SQL Server will create a default datetime of 01/01/1900 12:00:000 AM. Now when you query that data, you'll see 01/01/1900 on the local cursor—assuming you've used one of the techniques discussed earlier for truncating the time. I don't feel that that's such a horrible thing for a user to see, but if you've chosen to Set Century Off, 01/01/00 will look like January 1, 2000. (As if Y2K bugs weren't bad enough, upsizing will create a Y1.9K bug!)

I've come up with three possible workarounds. The first is to add a column default to all datetime fields—one that will make it very clear that this is a bogus date—like 01/01/9999. (Note that the datetime datatype has a date range from January 1, 1753, to December 31, 9999.) Again, you'd need Set Century On to differentiate it from 1999. A similar option is to add a column default to all datetime fields that follows a business rule. Perhaps all Ship Dates can default to eight weeks after the Order Date, a required field.

The third option involves some more sleight-of-hand. Wouldn't it be great if there were a way to clear out each of the dates equal to {01/01/1900} in the open view cursor, send no updates to the server, and leave no pending changes? Well, it can be done, and it goes a little something like this:

```
Procedure OpenView
LParameters pcView
* Retrieve data form server
Use (pcView) in 0
* Prevent update statements from going to server
CursorSetProp('SendUpdates', .F., pcView)
* Strip 01/01/1900 out of this cursor
RemoveDefaultDates(pcView)
* Remove pending changes/clear the change buffer
TableUpdate(.T., .T., pcView)
* Restore update capability to view
CursorSetProp('SendUpdates', .T., pcView)
```

For brevity, I won't provide code for function RemoveDefaultDates. But all it does is loop through the columns for date-type fields and then replace them with {} if they equal {01/01/1900}. Now all data entry screens will look and work as they did with a VFP back end.

Well, that works for views, but what about SPT cursors? You'd just need to run the function RemoveDefaultDates against any cursor that will be eventually presented to the user. (Hopefully, you've built a report printing class.)

Tip!

Differences between remote SPT cursors and local SPT cursors:

SQL Passthrough cursors are read-only when created from file-server data sources, but read/write when created from a client/server data source. So, if you're prototyping locally, you'll of course need to first make the cursor read/write before changing it.

Date math

You need to be aware of any date math you might have in your application. Being that you have datetime fields on

the server (and might have them on a local cursor too) where you originally had date fields, errors in calculations could occur. For example:

```
?Date() + 20      && 03/08/99 + 20 = 03/28/99
?Datetime() + 20  && 03/08/99 10:35:15 PM + 20 = ;
                     && 03/08/99 10:35:35 PM!
 * against SQL Server:
 * ExpectedDate is 14 DAYS after order date
Select orderdate, ;
  Orderdate + 14 as ExpectedDate ;
  From Orders

 * against VFP:
 * ExpectedDate is 14 SECONDS after order date
Select orderdate, ;
  Orderdate + 14 as ExpectedDate ;
  From Orders
```

You can see that adding a number to a datetime field can have different results based on the back end. It's tough to tell whether seconds or days will be added. A workaround might be to make the incrementing number a function. That way, you have the necessary hook to determine the back end and provide the appropriate calculation.

Step 10: Avoiding that Empty() feeling about Nulls

If you're like me, perhaps you've taken advantage of the flexibility that the VFP Empty() function provides. Empty(eExpression) works for any data type, except object. So you could trap for 0, or an empty string, or an empty date, or logical false with this one function without even checking for the datatype. Bad move.

Problems can occur because after upsizing, you might find some unexpected return values from functions and queries. They might even be of a different data type than you expect. When these values are evaluated with Empty()—or any function, for that matter—it could cause major problems. For example:

```
?Empty({})            && true
?Empty({01/01/1900})  && false
?Empty(Null)          && false
```

You've seen the {01/01/1900} issue in Step 9—life without dates, so you know it can occur. Clearing them out locally seems to be the easiest workaround. But what about the Null problem? Most know that Null is Not Empty, but did you know that even if you have no columns in your database that accept Null, you still might get Nulls back from the server? Try this on SQL Server:

```
Function GetMaxOrderQty
lcSQL = "SELECT MAX(Quantity) AS nMaxQty" +;
" FROM [Order Details] " +;
" WHERE ProductID = 99 "
SQLExec(lcSQL, "cBigOrder")
If Reccount("cBigOrder") > 0
  lnRetval = 0
Else
  lnRetval = cBigOrder.nMaxQty
Endif
Return lnRetval
```

What's the datatype of the return value? It depends. If there are any records found for ProductdID = 99, the answer is numeric. If there are no records found for ProductdID = 99, the answer is Null—despite the fact that the Quantity column doesn't accept Nulls.

The reason is because in SPT queries to SQL Server, you won't get a result cursor without records for aggregate queries such as these. You'll get one record regardless of whether any matches are found, unless you have a Group By clause on a non-aggregate column. And the aggregate column(s) will have a value of Null. That could really sting you in many places if you're not careful to avoid it.

In the preceding example, lnRetval would be Null because there's no ProductID = 99. To avoid getting Null return values, you have several choices. Of course, finding all of the aggregate functions in your code and doing an IsNull() check is one way. You could also write a wrapper for Empty, perhaps IsEmpty, that traps for Null, and 01/01/1900 as well. This can be used to check the values of different data types without the fear of encountering unexpected results.

Lastly, adding the Count() function, as in Count(*) as Cnt, to all queries such as these gives you a common way to check for a return value. The Cnt column will always be numeric, so now you can check for Cnt > 0 instead of Reccount() > 0 without fear of Nulls.

Step 11: The case for unbound controls

You've seen a lot of shortcomings to working with fields that are of different data types—depending on the back end. I see them as well. These problems make it very difficult to write one set of code to access multiple data sources. Obviously, not all applications require such flexibility, but for those that either need or would like this functionality, unbound controls might be the answer.

Unbound controls on a form don't have a datatype. You take the result set of a query and essentially "paint" the controls with the values of the field contents. That would give you the necessary hooks to manipulate the data any way you need to before presenting it to the user.

This provides great flexibility, but a ton of coding. Can you just imagine how much code it would take to read the data, loop through the controls, and paint the form, then loop through the controls again to create the SQL code that will write the updates to the server? Not to mention having to add the field-level, data type validation that you get for free with a bound control. The average FoxPro developer is so used to bound controls that he or she would rather chew glass than write all of that tedious code. There must be a better way.

If I had to live my life over again, I'd change only one thing: I'd have created yet another layer of data abstraction. (Okay, two things: I also would have bought

Microsoft stock 10 years ago. Who knew?) The extra layer would be implemented in the form of a read/write cursor with the same structure and content of the view's result set.

Controls can then still be bound to a data source, while at the same time you have the ability to manipulate the cursor in any way before presenting it. So the process would go something like this:

```
Procedure Load
* For this test, open the form with data.
Use vOrders In 0
* Get a copy of the result set.
Select * from vOrders Into Cursor cOrders1 NoFilter
* Make a read-write copy of cOrders1.
Use (DBF()) In 0 Again Alias cOrders
* Close the temporary read-only cursor.
Use In cOrders1
*
* Now the Init of the controls fire,
* which are each bound to cOrders.
* cOrders will be the form's master alias.
EndProc
```

Now just edit the form as usual. On the Save, dump the edited cursor back into the view and issue TableUpdate against it. The entire view-based framework discussed earlier would all still apply—you're just slipping in one additional layer. And with small result sets—a client/server requirement—coupled with the lightning-fast VFP data engine, performance shouldn't be an issue.

By providing yourself with this opportunity to manipulate the data before presenting it, you get the best of bound and unbound controls. This will make possible the addition of new features where the underlying data doesn't match what the user sees—for instance, multi-currency, multi-language, and character representations of data like 4 DOZ (four dozen).

Step 12: Creating a form that accesses multiple back ends

I've provided a form that's indicative of how you could design a form that needs to access multiple data sources. It's rather down and dirty—with all instance-level code—for ease of learning, but you could always enhance it once you understand the design. Also for simplicity, I've created only one filter field and haven't implemented Query by Form.

Before you can run the form, do the following: Extract the files and subfolders in 05FALINO.ZIP—available in the Subscriber Downloads at www.pinpub.com/foxtalk—to any directory. You'll find the form Orders, the bmps to support the form, and main.prg. The subdirectory AppData contains a copy of the Northwind Traders database in VFP format. (I used the SQL Server Data Transformation Wizard to do this.) The SSViews directory contains a remote view, vOrders, in a DBC called AppViews. Similarly, the VFPViews directory contains a local view, vOrders, in a DBC called AppViews. (Refer to Step 5 in last month's article for more on the file and directory layout.)

Run VFP version 6.0 and set the default to the directory where you extracted the files. Have SQL Server 7.0 installed and running. In the Control Panel, create an ODBC User DSN that connects to the Northwind Traders database and call it NORTHWIND_SS7.

To run the form against SQL Server, issue: MAIN("NORTHWIND_SS7"). To run the form against a the VFP back end, issue: MAIN("NORTHWIND_VFP"). As you can see from **Figure 1**, the title bar indicates the data source. Choose Find, enter a customer like "VINET," and then choose Retrieve. The five orders for this customer will be retrieved from the server.

Don't get too excited to see some magical code, because there really isn't any. In fact, it's not only simple, but almost the identical code you'd see in a form bound to buffered tables. The only differences are the addition of the NoData clause of the Use command in the Load, the setting of the cCustomer private variable that serves as the view parameter, and the subsequent Requery function call that will retrieve the data for the appropriate customer. These are the only two procedures of even remote interest (pun highly calculated):

```
PROCEDURE Load
* Important for local views
Set Exclusive Off
* Need this for table buffering
Set MultiLocks On
Set Data To AppViews
* No data on load, please
Use vOrders NoData
* Set optimistic table buffering
CursorSetProp("Buffering", 5)
ENDPROC

PROCEDURE cmdfind.Click
PRIVATE cCustomerID
cCustomerID = ""
IF Thisform.PendingChanges()
  WITH Thisform
    .Lockscreen = .T.
    IF Not Thisform.lFindMode
      thisform.lFindMode = .T.
      * Change to find mode
      this.Caption = "\<Retrieve"
```

Continues on page 19

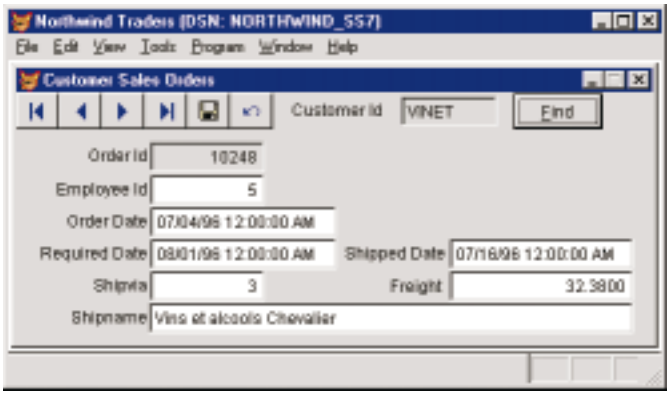

Figure 1. The Customer Sales Orders form can access data from multiple data sources.

Seeing Patterns: The Singleton

Jefferey A. Donnici

This month, the Best Practices column is back with another chapter in the continuing "Seeing Patterns" saga. The purpose of this series is to discuss common object-oriented design patterns in a context that's familiar to the Visual FoxPro developer—using VFP terminology, analogies, and examples. This month, the pattern being dissected is the Singleton pattern, which is also the first "creational pattern" to be addressed in this series.

WELCOME back to the "Seeing Patterns" series in Best Practices. For those who are just joining us, the purpose of this series is to explore the use of a variety of object-oriented design patterns, using VFP examples for illustration. If you get the chance, take a look through some of the recent Best Practices columns for a brief introduction to the theories and ideas involved with design patterns. Each of the patterns I discuss was first introduced in *Design Patterns: Elements of Reusable Object-Oriented Software*, by E. Gamma, R. Helm, R. Johnson, and J. Vlissides (Addison-Wesley, ISBN 0-201-63361-2). The focus with each of these columns is to explore the patterns in a manner that's familiar to VFP developers, using samples that can be readily understood. Because the discussions here are intended to be a brief "refresher" for each pattern and not a complete, "soup-to-nuts" discussion, you'll probably find it useful to have a copy of the book nearby when reading this series.

As I've mentioned in the past, the term "design patterns" refers to patterns that are intended to solve an object-oriented design problem. Although this series uses VFP as the language of illustration, I want to stress again that these patterns are equally useful in any other object-oriented language. As such, the examples I'm providing here aren't the only possible implementations of a design based on these patterns.

Creational patterns

As promised last time, I'm going to focus this month on a "creational pattern"—the Singleton. Before I get started with the Singleton pattern itself, however, I want to first discuss what a "creational" pattern is and how these patterns differ from other types of patterns.

Creational patterns are patterns that deal specifically with the process of object instantiation. As applications become more complex, they must evolve to use aggregation and composition more than simple inheritance. While inheritance is a useful tool and a fundamental part of object-oriented development, it's often over-used and results in unnecessarily complex class hierarchies. This, in turn, leads to increased maintenance and reduced reusability across the foundation of an application. With composition, a number of smaller classes that each define a very specific, very basic functionality are combined at runtime to provide larger, more complex behaviors. Different combinations of these smaller classes yield different behaviors, but knowing what to instantiate, when to instantiate it, and who's responsible for instantiating different classes becomes key in determining the specific desired behavior. That's where creational patterns come in.

As explained in *Design Patterns*, "There are two recurring themes in these patterns. First, they all encapsulate knowledge about which concrete classes the system uses. Second, *they hide how instances of these classes are created* and put together. All the system at large knows about the objects is their interfaces as defined by abstract classes . . . They let you configure a system with 'product' objects that vary widely in structure and functionality. Configuration [of the classes] can be static (that is, specified at compile-time) or dynamic (at runtime)."

Here's a rundown of the creational patterns described in *Design Patterns*, along with brief descriptions of how each might be used to dynamically change a system's behavior through selectively creating members of an overall composite. I'll use a system's reporting capabilities to describe how each type of pattern might be put into practice.

- **Abstract Factory**—This pattern uses an object's interface as a "blueprint" to describe how other objects are created. For example, a "Report Factory" class might contain methods and / or properties that tell another object, which might receive the "factory" as a parameter, how to build a report (orientation, fonts, paper size, and so forth). Providing a different descriptor object as a parameter changes the way the report-building object creates the report.

- **Builder**—Similar to the Abstract Factory, except that this patterns calls for the "blueprint" component and the "constructor" component to be the same thing. When the Builder component uses its own, or, more

accurately, its members' methods and properties to build the report itself, using inheritance allows subclasses to create more specific types of reports. This allows the same report "constructor" component to create different types of reports by varying the member "blueprint" information.

- **Factory Method**—This pattern, also known as the "Virtual Constructor," describes that a component is responsible for creating another object, but leaves the decision over *which* object to create up to subclasses. For example, you might have an abstract "reporting system" component with a CreateReport() method. You might also have another class that models the report itself. In practice, however, you might need to support reports built with VFP's own reporting engine, as well as some third-party reporting tool. In this example, a subclassed "reporting system" component for each report type would have an overridden CreateReport() method that deals with the specific "report model" it's responsible for. The abstract framework for creating a report doesn't change; it just lets the actual construction be handled by concrete subclasses.

- **Prototype**—With the Prototype pattern, a "client" component uses a "prototype" component whenever multiple objects of that prototype's behavior/appearance are required. For example, suppose your report construction mechanism calls for several columns to be built in a spreadsheet. With this pattern, the report constructor would instantiate only one "column prototype" and use that prototype instance repeatedly to build the columns. The "column prototype" might have a "CreateColumn(tcColumn, tcField)" method, where "tcColumn" is a parameter describing the column to be built within the spreadsheet and "tcField" is the field to be displayed in that column. By using one "prototype column" to model how a column is created on the report, multiple columns can be created by essentially "copying" the one prototype with different values for each successive copy.

Finally . . . the Singleton

With that, you should have a handle on what a creational pattern is and how they're different from structural patterns, like the Bridge and Decorator, and behavioral patterns, like the Mediator and Chain of Responsibility. The Singleton pattern, like the other creational patterns, describes the instantiation or creation of a component, but it also describes how other components interact with that instance. With the Singleton pattern, there should be only one instance of the "Singleton class," and the rest of the system should have a single, global point of access to it. It's worth noting that just about any of the other creational patterns can also use the Singleton pattern as a part of their design.

In *Design Patterns*, the authors use the example of a printing system and explain that, "although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager." By providing a single point of access to a system resource, device contention and prioritization can be managed in one place, providing for less overhead and increased efficiency. Not only should this type of component have only one instance, but it should also be easily accessible to the "clients" that must use it. Further, the "clients" that use that instance should be able to use subclasses or more specialized instances of the Singleton class without having to be aware of, or compensate for, the change.

For several familiar examples of a Singleton type of approach, I'd suggest you open your Windows Control Panel (in any 32-bit version of Windows) and look through some of the Control Panel applets available to you. Here, for example, are a few of the obvious "Singleton" designs you'll find:

- **Modems**—Remember back in the old days (say, four years ago) when every application had to be told what type of modem you had, how to dial, what connect strings to use, and so forth? With Windows, the operating system manages a single collection of modems in the system, with all of the necessary info about each accessible to applications that might use the modem. Additionally, this "modem manager" assures that no two applications are trying to use the modem at the same time.

- **ODBC**—Imagine having to tell every application you install about all of the different types of data sources you have on your machine. Or imagine having to load a set of proprietary data-source drivers for every application that needs access to your data. No, thanks.

- **Printers**—One of the best things about the Windows and Mac operating systems is the common print manager shared by the whole operating system. Sure, there are problems with errant printer drivers now and then, but it wasn't that long ago that us DOS types had to tell every application on our systems about the old LaserJet II we had connected to LPT1—and then hope that the application knew how to talk to it. With these newer OSs, we install one driver, and every application talks to it via the single printer management mechanism.

- **Regional Settings**—I like my date formats to be just so, and I really prefer my system clock to show me 24-hour time instead of "a.m." or "p.m." Plus, different countries use different symbols for the decimal place and digit grouping symbols. Given all of the applications you have that display numbers or have the time and date in their status bars, would *you* want to specify this information for every app?

As I was working on this article, another great example of a Singleton pattern occurred to me. I got a phone call from someone telling me that they'd sent some information to me via my personal e-mail address. When I tried to connect to my local Internet Service Provider, I got a message telling me that there was already a Dial-Up Networking connection established. Sure enough, I'd forgotten that I was already connected to my office's Windows NT network. Suddenly, it struck me right between the eyes that I was looking at the Singleton pattern in action.

With Dial-Up Networking, each connection (or "connectoid," as many refer to them) that's been defined describes how the computer will connect with other computers. This can be a TCP/IP protocol connection for Internet connectivity or "WinSock" access, or it could be some other protocol that's required for the type of system you're connecting with. The point, however, is that there can be *only one* Dial-Up Networking connection established at a time. When a second connection is attempted, which is what I was trying to do, a gentle warning is given that multiple "instances" aren't allowed. Also, just as the Singleton pattern describes, there's only one "global point of access" to the Dial-Up Networking component. That point of access is a "system folder," surprisingly named "Dial-Up Networking." You can view this folder in the Windows Explorer where it's listed with some of the other "Singleton" functions of the operating system—Control Panel, Printers, and the Recycle Bin.

Back To VFP

Okay, so now you have a pretty good idea of what the Singleton pattern is about and how it's used throughout the Windows operating system. Let's take a look at how you might see the Singleton pattern used in a VFP application.

Back in my December 1998 column ("Seeing Patterns: The Mediator"), I discussed the idea of using a Forms Manager in your applications. The idea is that a Forms Manager provides a central place for accessing all of the open forms in an application. In that column, I presented the basic skeleton for a Forms Manager class, which contained the following methods:

- **GetFormCount**—Returns the number of open forms in the application.
- **NoForms**—Returns a logical indicating whether or not the Forms Manager has an empty forms collection.
- **FindForm**—Returns the index number of the passed form within the Forms Manager's collection.
- **FormExists**—Returns a logical to indicate whether or not the passed form appears in the Forms Manager's collection.
- **AddForm**—Receives a form object as a parameter and adds a reference to that form to the Forms Manager's collection, increasing the collection size as needed.
- **RemoveForm**—Removes the reference to the passed form from the Forms Manager's collection and adjusts the size of the collection accordingly.
- **RemoveAllForms**—Removes all forms from the Forms Manager's collection by iterating through the collection and calling RemoveForm for each.
- **GetFormRef**—Searches for a form that matches the passed parameters and returns an object reference to the passed form. If no match is found, a NULL value is returned.

Obviously, a "system resource" like a Forms Manager component is a good candidate for the Singleton pattern. Not only should there only ever be one Forms Manager for an application, but there must also be a single point of access to it. Without that single point of access, there's no way for other modules or objects within the application to know how to get information about the system's forms.

I'll save the discussion of a "system resource" Singleton for the next example, though, because I want to focus instead on that last method in the preceding list—*GetFormRef*. The purpose of that method is to return an object reference to the form that matches the passed parameters. If there's no open form that matches, then a NULL value is returned. This method, as well as the global accessibility of the Forms Manager, lets us create a "Singleton Form class" that we can use for all forms that should only ever be opened once. You might, for example, have a navigation window that you only want opened once. In a multi-windowed system, however, your user might not see the already-open instance and choose the toolbar button or menu option that opens your navigation form. Instead of opening yet another instance of that form, you might want to just bring the instance that's already open to the top so that

the effect for the user is the same.

To do this, the Singleton Form class must check the Forms Manager for a previous instance before it completes its instantiation. If there's already an open instance, then the form class shouldn't instantiate and should instead activate the existing form instance. The following code in a base "Singleton Form" class would handle this functionality for me:

```
*-- Check the app's oForms collection
*-- to see if there's already an instance of
*-- this class running. If so, bring it to
*-- the front and don't finish instantiating this
*-- second instance.
loPrevForm = ;
 goApp.oForms.GetFormRef(THIS.Class, "CLASS")
IF TYPE("loPrevForm") == "O"
  ACTIVATE WINDOW (loPrevForm.Name)
  RETURN .F.
ELSE
  oApp.oFormsManager.AddForm(THISFORM)
ENDIF
```

The loPrevForm variable is used to store the return value from the Forms Manager's GetFormRef method. If that method returns an object reference, then I know that there's already an instance of this form class in the Forms Manager. In that case, I use ACTIVATE WINDOW to bring that previous instance up to the top. The user doesn't have to know whether I created a new instance or am reusing the earlier instance because the objective for them is completed. If a NULL value is returned from the GetFormRef method, then I know there isn't a previous existence in the collection, so I call the AddForm method of the Forms Manager, passing the newly created form as a parameter.

So, while the Forms Manager itself is a good example of a Singleton pattern at work, it can also serve its Mediator role and let us create multiple "Singleton Forms" within an application. Reusing open forms like this saves on instantiation time, as well as the memory overhead of multiple, redundant form instances.

Designing a system resource

The "security mechanism" is one of the more common system-wide resources we must deal with in our database applications. The fundamental property of all database applications is that they manage data, and sometimes that data can be of a sensitive nature. For example, you don't want every data entry clerk to have access to your employee salary tables, right? Similarly, you probably don't want the volunteer at the admitting desk of your local hospital to have access to every patient's medical records. Other times, the data isn't especially sensitive, but it's mission-critical data that absolutely must be available and accurate at all times. For that reason, you don't want every employee to be able to change the data or lock tables and perform database maintenance.

Enter the security system. A good security system requires the user to log in to the application so that the system can decide which functions to enable, which to disable, which data to hide completely, which to provide on a read-only basis, and which data can be modified or deleted by that user. Most security systems allow an administrator to define certain "roles" and then assign those roles to the users that will be using the system. This makes administration and management of the system easier because all of the individual settings and tasks don't have to be enabled or disabled on a per-user basis. Those of you who are familiar with Windows NT administration will recognize this approach in the "User Manager for Domains" tool that lets you assign "group" permissions and then add or remove users to and from the group.

In this example, I've mocked up a sample security component and sample "application." For our purposes here, the "application" is actually a form (see **Figure 1**)—while the form is open, the application is "running," and the application will "shut down" when the form is gone. The form contains a combo box to indicate the current security level. As the security level is changed, different "capabilities" in the application are made visible / invisible or enabled / disabled (that is, controls on the form will have their properties changed to reflect the security level). The form also contains a button that will add another instance of the security component, the cstSecurity class, to the application's oSecurity property. More accurately, it *attempts* to add another instance and, as you'll see, fails because the security component has been designed following the Singleton pattern.

To reflect the nature of the Singleton pattern, the security component must only allow a single instance, and it should have a *single, global point of access* to its

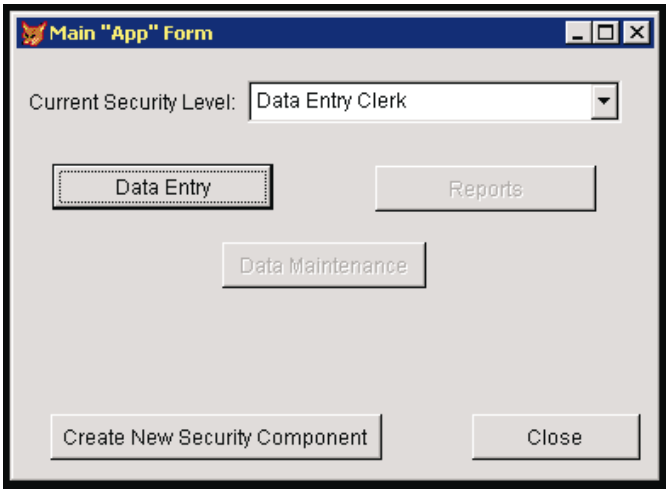

Figure 1. The security "application" is actually a form that displays and/or enables specific controls based on the current security level.

interface. Those who know my development style might well be thinking that I've finally flipped my lid and used the PUBLIC keyword in my code. Not so, my friends: By instantiating the application object (the form) as a PRIVATE variable, its interface is accessible to everything else in the application because the rest of the application "stems from" the private application object. Whew . . . no PUBLIC, and I get to maintain my dignity.

First things first—you can run this example (available in the Subscriber Downloads at www.pinpub.com/foxtalk) by issuing the following command in the Command Window:

```
DO SECURITY
```

In the Load() event of the "application," I populate an array with the five different security levels that this application allows. For simplicity's sake, I'm just using an array here, but a "real-world" application should use a "roles" table and a "users" table, mapping each user to a role and then providing access to various data and functionality with a roles-based mechanism. Also in the Load() event, I'm calling the AddSecurity method of the application, which contains the following code:

```
LOCAL loSecurity

*-- Add the security component to the "app". Note
*-- that I'm passing an instance of the "app"
*-- form to the security component. This will
*-- let the "Singleton" component verify that only
*-- one instance is being created -- it can check
*-- the application's ".oSecurity" member before
*-- returning .T. from the Init().
loSecurity = CREATEOBJECT("cstSecurity",THISFORM)

IF TYPE("loSecurity") == "O" AND ;
   NOT ISNULL(loSecurity)

   THIS.oSecurity = loSecurity
ENDIF
```

As the comments make clear, I'm passing a reference to the "app/form" to the cstSecurity class's Init event. This lets the class verify that there's only one instance of the security component in the application. You can verify that this portion of the Singleton pattern's structure is working by pressing the "Create New Security Component" button on the form. This attempts to run the AddSecurity method again, but the security component's Init event catches the second instant and returns .F. so that the second instance isn't actually created.

```
LPARAMETERS toApp

IF TYPE("toApp")<>"O"
  RETURN .F.
ENDIF

*-- If there's already a security object in
*-- place, then the Singleton security component
*-- won't allow multiple instances.
IF TYPE("toApp.oSecurity") == "O" AND ;
   NOT ISNULL(toApp.oSecurity)

   WAIT WINDOW "Only one instance of the " + ;
               "security component is allowed."
   RETURN .F.
ENDIF

RETURN
```

Once the security component has been instantiated, it's stored in the application's oSecurity property, where it will always be accessible to the application. As the security level is changed in the combo box, the new value is stored to the security component. Whenever an object or module in the application needs the current setting, it can get the value with the following method call:

```
lnSecurityLevel = poApp.oSecurity.GetSecurityLevel()
```

From there, each control or function is free to respond or behave according to the current security level of the application. For example, when the security level is changed, the form's ReflectSecurity method is called. That method, in turn, calls the ReflectSecurity method of any of its member controls that has that method (verified via PEMSTATUS()). In this example, each of the form's command buttons is an instance of cmdFormLaunch, which has the following code in its ReflectSecurity() method:

```
LOCAL lnSecLevel

*-- If we have a valid security object in place,
*-- use it to determine the current security
*-- level. Then change this instance's Enabled
*-- and Visible properties accordingly.
IF TYPE("poApp.oSecurity") == "O" AND ;
   NOT ISNULL(poApp.oSecurity)

 lnSecLevel   = poApp.oSecurity.GetSecurityLevel()
 THIS.Enabled = (lnSecLevel >= THIS.nMinToEnable)
 THIS.Visible = (lnSecLevel >= THIS.nMinToDisplay)

ENDIF
```

The nMinToEnable and nMinToDisplay properties of the command buttons indicate the minimum security level required for the button to be enabled and/or displayed, respectively. This allows each button to take care of itself with regard to allowing the current user access to the functionality that the button represents (see **Figure 2**). In the typical application, this sort of "self-policing" security mechanism might be put into place for menu bars, toolbar buttons, or even individual rows in list box or combo box controls.

Additionally, I've specified a base form class, called frmBaseSecure, that has an nSecurityLevel property defined. That property indicates the minimum application security level required for that form to open. In that form's Init, the following code verifies that the current user has permissions to open the form subclassed from frmBaseSecure:

```
LOCAL lnSecLevel, llRetVal

*-- Get the current security level from the
*-- security component on the application.
```

```
lnSecLevel = poApp.oSecurity.GetSecurityLevel()
llRetVal   = .T.

*-- If the security level is less than the
*-- security level required by the form instance,
*-- let the user know that they don't have the
*-- necessary permissions.
IF lnSecLevel < THIS.nSecurityLevel
  WAIT WINDOW "You don't have the appropriate" + ;
              " permissions to run this form."
  llRetVal = .F.
ENDIF

RETURN llRetVal
```

If the form is being instantiated when the current security level falls below the security level required for that form, then a WAIT WINDOW is presented to the user. Additionally, the Init event returns .F. so that the form is never really instantiated.

Using this sort of mechanism, the single security component stored in the application's oSecurity property is providing the same security information to three different types of "restrictions" in the system: hide functionality that the current user shouldn't see, disable functionality that the user can see but not execute, and trap functionality during execution based on the current security level. Obviously, this is a very basic example of how security might be implemented in an application, but it demonstrates the utility of a "single point of access" to the security system resource.

Portable patterns

For those of you who are working with other object-oriented languages, I hope you're finding these patterns discussions useful. I've been working with Java a bit lately, and it's rewarding to be able to put all of these concepts into practice with another language. While there are times when I really long for the good old Command Window in my Java tools, it's great to be able to solve design problems there with patterns-based solutions that I've successfully used in VFP. If you can find the time, I encourage you to look into other object-oriented environments. You might find that the varying environments and capabilities bring a deeper understanding or fresh perspective to the design issues you face in VFP. Now if someone would just invent the 48-hour day.

Next month, I'll take a look at the "Iterator" pattern. Like other "behavioral" patterns we've discussed, such as the Chain of Responsibility, Mediator, Observer, and Strategy patterns, the Iterator pattern describes a component's responsibilities and collaborations with other components. In the meantime, don't hesitate to send me e-mail if you have any ideas, suggestions, or questions about the Best Practices column. ▲

DOWNLOAD **05DONNIC.ZIP at www.pinpub.com/foxtalk**

Jefferey A. Donnici is the senior Internet developer at Resource Data International, Inc., in Boulder, CO. While it was much more humorous, he had to delete the initial draft of this column when he learned that "simpleton" and "singleton" weren't synonymous. Jeff is a Microsoft Certified Professional and a four-time Microsoft Developer Most Valuable Professional. 303-444-7788, fax 303-928-6605, jdonnici@compuserve.com, or jdonnici@resdata.com.

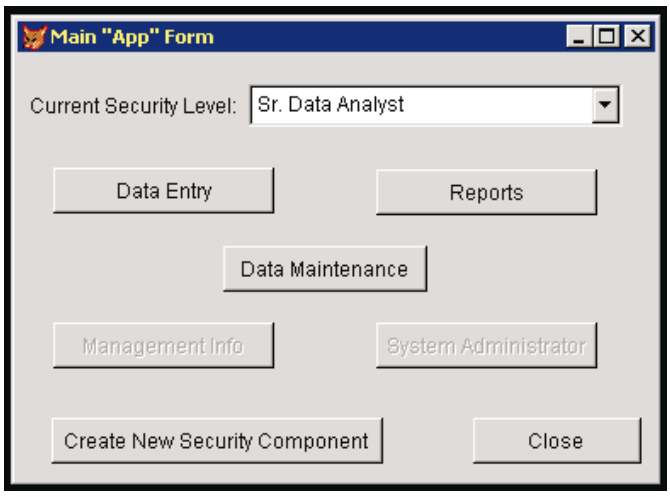

Figure 2. As the current security level of the application is changed (via the combo box), various controls that represent different functionality are made visible or enabled.

Matchmaker, Mitchmoker . . . Is That a Match? Or De-mystifying De-duplication

Andrew Coates

UNIX MAC DOS WIN 2.x 3.0 5.0 6.0 DOWNLOAD

Finding duplicate entries in your data or matching records from different tables is a problem that has plagued database designers since databases began (and even earlier). In this article, Andrew demonstrates some of the techniques you can use to match records.

YOU know the story—two of the departments in the company have each maintained records about their customers, and now the IT manager has decided that the databases need to be merged. The only problem is, how do you decide whether a record from the first database matches one from the other? In this article, I'll explore some options for matching these records (or for detecting duplicates in a data set, which is essentially the same thing). I'll discuss some concepts about what you're trying to achieve, talk about what you can match, and delve briefly into name and address standardization. Next, I'll talk about automating the matching process and a couple of algorithms you can use for matching the sounds, not the spelling, of words. By the end of this article, you should at least be able to plan a matching strategy that fits your needs best.

Matching/de-duping

Matching and de-duplication are essentially the same process—that of finding records that refer to the same entity, either in two (or more) separate tables, which I call matching, or in one table, which I call de-duplication. If you're working with a single table, then you've probably at least got the advantage of having the information you're examining in the same format for each side of the comparison. When you're matching, you often need to do some pre-processing to get the information from the first table into the same format as the information in the second table. On occasion, it's also useful to do some standardization before attempting to de-duplicate.

Name and address standardization

How do you do this pre-processing? Well, to make sure you're comparing apples to apples, you need to carry out some form of standardization. To standardize names, you might need to break apart names that are stored as one string into their component parts. For example, the single field ContactName containing "Mr Andrew C Coates BE" can be broken down into the following:

ContactTitle	"Mr"
ContactFirstname	"Andrew"
ContactInitials	"C"
ContactSurname	"Coates"
ContactPostNom	"BE"

Doing this will allow you to much more easily find a match with "A.C. Coates," which you'd standardize as follows:

ContactTitle	""
ContactFirstname	"A"
ContactInitials	"C"
ContactSurname	"Coates"
ContactPostNom	""

Break data down into the smallest units you have. For example, split names into their components, and addresses into number, street name, city, state, and postal code.

Use standard abbreviations for common components. For example, the USPS has a comprehensive list of street types ("Street," "Lane," and so forth), common variations ("St," "Str," "Ln," "Lne," and so on), and their preferred abbreviation. Use either the standard full title or the standard abbreviation.

Time spent on standardization is rarely wasted. Extract a set of data from your target to develop your standardization algorithm. Extract another (completely separate) set of data from your target to test the algorithm after you're satisfied with the algorithm. Having this second set will often reveal exceptions or expose inaccuracies that you might not have considered. It's important to have this verification set of data as well as a development set.

For anything but the most trivial data sets, standardization is a computationally intensive task. Be prepared for the standardization processing to take a significant amount of time. Standardization routines I've

developed have turned into 100+ case statements and have taken in the order of one quarter of a second per record to process. While this isn't a lot if you're only processing a thousand or so records (although there's still time to make a cup of coffee), 250,000 records will take more than 17 hours.

As a final standardization remark, I should note that there are commercially available standardization packages, as well as mailing houses that will take your data and return it appropriately compartmentalized. You might consider using these services if you have a single job and can't justify developing your own routine or if you have so much data that you just don't have the resources to handle it.

What to match

Once you've got your data in a standard format, you need to decide what constitutes a match. Is the A. Coates living in Sydney the same as the Anthony Coats from Maroubra in New South Wales? The answer to that question is "possibly." What the probability is of a match is something you need to determine.

If you have a match on a unique identifying number like the U.S. social security number, then the probability of a match is quite high. If you don't have the "smoking gun," then you need to use other means to make the case for a match or otherwise. Possible fields for matching are:

- Address (but don't forget that families often live in the same house and have pretty similar names)
- Date of birth / age
- Name
- Geographical location
- Company name
- Phone numbers

Phonetic matching

One common matching option is to compare the phonetic values of strings such as people's names or street names. There are several algorithms available that assign a value to a string based on how it sounds. Using these techniques, you'll be able to find duplicates that might not be spelled exactly the same way, such as "Smith" and "Smythe." The algorithms I've used to a greater or lesser extent are SOUNDEX and NYSIIS. Each of these has its pros and cons. FoxPro also includes an algorithm called DIFFERENCE() that compares two strings. Another algorithm I found while researching this article is called Metaphone. I've never used it in anger, and I've not been able to find the source for it, but it appears to produce a code that's similar, but not identical, to the NYSIIS algorithm.

Phonetic algorithms basically work by suppressing the vowel information (because it's unreliable) and giving the same code to letters or groups of letters that sound the same (for example, "PH" sounds like "F," so you give them both the same code). You can use the code generated to find matching names in a table; for example, the following will display a browse window with all records with surnames including SMITH, SMYTHE, SCHMIDT, SMYTH, and SCHMIT:

```
BROWSE FOR SOUNDEX(surname) = SOUNDEX("SMITH")
```

SOUNDEX

SOUNDEX is a phonetic coding algorithm that ignores many of the unreliable components of names, but by doing so reports more matches. The rules for coding a name are (from Newcombe):

1. The first letter of the name is used in its un-coded form to serve as the prefix character of the code. (The rest of the code is numerical.)
2. Thereafter, W and H are ignored entirely.
3. A, E, I, O, U, and Y aren't assigned a code number, but they do serve as "separators" (see Step 5).
4. Other letters of the name are converted to a numerical equivalent:

   1 B, P, F, V
   3 D, T
   4 L
   5 M, N
   6 R
   2 All other consonants (C, G, J, K, Q, S, X, Z)

5. There are two exceptions: a) letters that follow prefix letters that would, if coded, have the same numerical code, are ignored in all cases unless a "separator" (see Step 3) precedes them; and b) the second letter of any pair of consonants having the same code number is likewise ignored (unless there's a "separator" between them in the name).
6. The final SOUNDEX code consists of the prefix letter plus three numerical characters. Longer codes are truncated to this length, and shorter codes are extended to it by adding zeros.

Examples of names with the same SOUNDEX code are shown in **Table 1**.

Table 1. Names with the same SOUNDEX code.

Name	Code
ANDERSON, ANDERSEN	A 536
BERGMANS, BRIGHAM	B 625
BIRK, BERQUE, BIRCK	B 620
FISHER, FISCHER	F 260
LAVOIE, LEVOY	L 100
LLWELLYN	L 450

SOUNDEX has an immediate attraction for FoxPro (and SQL Server) developers. It's implemented as a native language function. Issuing the following from the command window will display the code S530—no further programming is necessary:

```
? SOUNDEX('SCHMIDT')
```

Compare this with the monstrosity that's NYSIIS.PRG (see this month's Subscriber Downloads at www.pinpub.com/foxtalk). Having the function built-in also has significant speed advantages.

NYSIIS

NYSIIS differs from SOUNDEX in that it retains information about the position of vowels in the encoded word by converting all vowels to the letter A. It also returns a purely alpha code (no numeric components). NYSIIS isn't part of the native FoxPro command set, nor does it seem to have been implemented by any third-party utility developers. I've coded the algorithm in the FoxPro procedure shown in and included in this month's Subscriber Downloads, but the high-level pseudo-code for this algorithm is shown in **Listing 1**.

Listing 1. High-level pseudo-code for the NYSIIS algorithm.

```
* Program....: NYSIIS.PRG
* Version....: 1.0
* Author.....: Andrew Coates
* Date.......: March 1, 1999
* Notice.....:
* Compiler...: Visual FoxPro 6 for Windows
* Abstract...: NYSIIS phonetic encoding algorithm
* Taken from Newcombe 1988 pp182-183
* rule number and lettering as per Newcombe
* Changes....:

*!* 1. Change the first letter(s) of the name
*!* 2. Change the last letter(s) of the name
*!* 3. First character of the NYSIIS code is the
*!*    first character of the name
*!* 4. Set the pointer to the second letter
*!*    of the name
*!* 5. Change the current letter(s) of the name
*!* 6. Add a letter to the code
*!* 7. Change the last character of the NYSIIS code
*!* 8. Change the first character of the NYSIIS code
```

NYSIIS has a disadvantage in our rapidly shrinking, multicultural world of being fairly Anglo in its phonetic coding. If the names you're matching have non-Anglo origins, it would probably be better to use a different algorithm—for example, SOUNDEX.

Phonetic matching example

To demonstrate the use of phonetic matching, I've used the customer table in the testdata database in the sample data that comes with VFP. The code is shown in **Listing 2**.

Listing 2. Phonetic matching example.

```
* Program....: PHONETICS.PRG
* Version....: 1.0
* Author.....: Andrew Coates
* Date.......: March 1, 1999
* Notice.....:
* Compiler...: Visual FoxPro 6
* Abstract...: Extracts surnames from
*  the testdata!customer table and does
*  phonetic comparisons.
*  NB - assumes that NYSIIS.PRG is in the path
* Changes....:

clos data all
open data (home(2) + 'data\testdata.dbc')

* break the contacts' names apart
select cust_id, ;
  padr(substr(contact, at(' ',contact) + 1), 30) ;
  as Surname ;
  from customer ;
  into cursor names


* get the codes for each contact
select *, ;
  nysiis(surname) as NYSIIS, ;
  soundex(surname) as SNDX ;
  from names ;
  into cursor codes

* get a list of all the nysiis codes that appear
* more than once
select nysiis, count(*) as tot ;
  from codes ;
  group by nysiis ;
  having tot > 1 ;
  into cursor multinysiis

* get a list of all the soundex codes that appear
* more than once
select sndx, count(*) as tot ;
  from codes ;
  group by sndx ;
  having tot > 1 ;
  into cursor multisndx

* generate a list of customers with phonetically
* matching surnames
select names.surname, codes.sndx ;
  from names ;
  inner join codes on ;
    names.cust_id  = codes.cust_id ;
  inner join multisndx on ;
    codes.sndx = multisndx.sndx ;
  order by codes.sndx ;
  into cursor sndx

select names.surname, codes.nysiis ;
  from names ;
  inner join codes on ;
    names.cust_id  = codes.cust_id ;
  inner join multinysiis on ;
    codes.nysiis = multinysiis.nysiis ;
  order by codes.nysiis ;
  into cursor nysiis
```

First, I extract the surnames from the customer name field using the assumption that the surname starts immediately after the first space character in the name field. Note that this assumption isn't valid for all cases—like the names "José Pedro Freyre" and "Isabel de Castro"—but I've ignored that problem here.

Next, I calculate the NYSIIS and SOUNDEX codes for the surnames and store them with the customer ID. Then I

find all of the codes that appear more than once (potential matches), and finally I generate a cursor with the surname and code for each potential match. The results for SOUNDEX and NYSIIS are shown in **Table 2** and **Table 3**, respectively.

Table 2. Potential SOUNDEX matches from the customer table.

Surname	SOUNDEX
Ashworth	A263
Accorti	A263
Berglund	B624
Bergulfsen	B624
Crowther	C636
Cartrain	C636
Moreno	M650
Moroni	M650
Pereira	P660
Perrier	P660
Wilson	W425
Wang	W520
Wong	W520

Table 3. Potential NYSIIS matches from the customer table.

Surname	NYSIIS
Moreno	MARAN
Moroni	MARAN
Pereira	PARAR
Perrier	PARAR
Wilson	WALSAN
Wang	WANG
Wong	WANG

By comparing Tables 2 and 3, you'll notice that SOUNDEX seems to match more names than does NYSIIS. The additional matches generated by SOUNDEX in this case don't seem to be good matches, but that's not always the case. You need to assess your data (perhaps by pulling a sample of 100 or so matches and calculating the hit rate).

Automating matching

It's possible to assign a value based on a match on some or all of the fields in your table and then use the "matching rank" to automatically determine whether records match. You could set up a system like the one shown in **Table 4**.

Using this system, you could calculate a matching rank for each record compared with each other record. You could set a threshold value—say, 200—above which you're sure that you've got a match, and another—say, 100—below which you're sure you haven't (the actual numbers you use will depend on your data). The question is what to do with the middle range. Generally, they have to be reviewed by a human.

Another problem you might come across with this system is multiple possible matches. You need to decide how to handle these. Perhaps present the top n possible matches, or perhaps present all of them. You could decide to present any match with a rank of at least 50 percent of the highest possible match. What happens if you have a definite match and a possible match? In this case, there's a possibility that there's a duplicate in the table you're matching. It's worth de-duplicating before you match to try to reduce this problem as much as possible.

You can write a program to automate the matching process. **Listing 3** shows some pseudo-code for such a program.

Listing 3. Pseudo-code for automatic matching process.

```
Open Tables
For each record in table1
  For each record in table2
    Get matching rank for this record combination
    Do case
    Case matching rank < lower threshold
      No match, just go to next record
    Case matching rank > upper threshold
      Definite match - write IDs to matched record
      table
    Otherwise
      Possible match - write IDs and rank to possible
      match table
    End case
  End for && table2
End for && table1
Deal with the possibility of multiple possible matches
```

Table 4. Sample "matching rank" system.

Field	Match	Rank
Address	All fields identical	100
	Street name SOUNDEX match and Suburb and State match	80
	Suburb and State match	60
	Address 1 within 50 km of address 2	30
	Country doesn't match	-20
	Any other address configuration	0
Name	All Fields identical	100
	Surname identical, first letter of first name matches	50
	Surname SOUNDEX matches, first name matches	40
	No match on any field	-100
Date of Birth	Identical	80
	Any 1 component (day, month, or year) +/-1	30
	Within 18 months	10
	Difference between 5 and 10 years	-30
	Difference > 10 years	-80
	Any date of birth configuration	0
Gender	Matches	0
	No Match	-100

Avoiding duplicates during data entry

Prevention is better than cure, and if you have control of the data capture phase of your operation, you can apply the matching algorithms suggested previously to the data as it's being keyed. Tell the user if you think they're entering duplicate data, and you'll eliminate the need for costly and time-consuming de-duplication later.

Conclusion

This month, I've probably presented more questions than answers, but that's often the way when describing a fuzzy operation such as matching. I hope that I've been able to point out some of the things you can do to find matching records and avoid duplication in your data sets. Matching is an art, and one you need to practice to perfect. Criteria for determining matches change depending on the data set, and you need to tweak your matching processes accordingly.

Next month, I'll deviate a little from the Data Bus concept and introduce a "cool tool" you can use for sending messages between applications or instances of the same application across a TCP/IP network. I'll show you how to build a simple license manager and a chat server. ▲

DOWNLOAD **05COATSC.ZIP at www.pinpub.com/foxtalk**

Andrew Coates is a director of Civil Solutions, a PC development consultancy in the Olympic City, Sydney, Australia. Andrew specializes in PC database applications, particularly integrating components and visualizing spatial data. a.coates@civilsolutions.com.au.

References

I got the SOUNDEX and NYSIIS algorithms and several other concepts from the *Handbook of Record Linkage*, Howard B. Newcombe, Oxford University Press, 1988.

12-Step Program . . .

Continued from page 8

```
      .SetAll("Enabled", .F., "textbox")
      .SetAll("Enabled", .F., "_commandbutton")
      .txtCustomerID.Enabled = .T.
      .txtCustomerID.Setfocus()
    ELSE && In find mode, so Retrieve
      .SetAll("Enabled", .T., "textbox")
      .SetAll("Enabled", .T., "_commandbutton")
      this.Caption = "\<Find"
      .txtCustomerID.Enabled = .F.
      cCustomerID = .txtCustomerID.Value
      Requery()
      thisform.lFindMode = .F.
    ENDIF
    .cmdfind.Enabled = .T.
    .txtOrderID.Enabled = .F.
    .Refresh()
    .Lockscreen = .F.
  ENDWITH
ENDIF
ENDPROC
```

Acknowledgments

I've leaned on a lot of people during this VFP to SQL Server migration—you will, too. So I'd like to thank the following for their contributions: Brent Vollrath and Terry Weiss of Micro Endeavors; Richard Berry, Phu Luong, and the rest of the Visual Garpac Development Team; Roger Nettler and Stan York of Programmed Solutions, Inc.; and Bill Martiner, an independent consultant and a SQL Server guru.

Conclusion

I hope you won't let all of these issues deter you from migrating to SQL Server. Yes, it's a lot of work. But it's more tedious than it is complicated. I hope this article saves you a lot of time by making you aware of many of the issues before you even begin.

As far as having one set of code access a VFP database as well as client/server databases, I don't recommend it for the long term. There's a little too much conditional code required and duplication of tools that must be written to warrant it. In the end, you'll probably end up with a system that's inefficient against all back ends, although you might get away with it if you throw enough hardware at the situation. I'd use local views only if I were definitely planning to upsize to SQL Server, Oracle, and so on. After the prototyping stage, I recommend that you gradually migrate your code to work remotely only.

SQL Server 7.0's ability to work on Windows 95/98 and NT and a host of new features make moving from VFP a much more seamless process. It can easily serve as your only back end, while using the VFP data engine for crunching data brought from the server. Together, they make a perfect team.

I welcome hearing from you about your experiences, trials, workarounds, and successes. Good luck. ▲

DOWNLOAD **05FALINO.ZIP at www.pinpub.com/foxtalk**

Jim Falino has been happily developing in the Fox products since 1991, beginning with FoxBASE+. He's a Microsoft Certified Professional in Visual FoxPro and the vice president of the Professional Association of Database Developers (PADD) of the New York Metro Area. For the past three years he has been a project manager, leading the development of a very large client/server apparel-manufacturing application for the GARPAC Corporation using Visual FoxPro as a front end to any ODBC-compliant SQL back end. jim@garpac.com.

VLT on Rye, Hold the Mustard

Paul Maskens and Andy Kramek

MAC WIN 3.0 5.0 6.0

Very Large Tables are possible in Visual FoxPro. Here's one approach to handling the problem of explosive growth in data volumes . . .

Paul: As you know, Andy, we've been experiencing a 50 percent growth per month in data volumes recently. Compound growth, of course. Not surprisingly, we're running into the 2GB per table limitation in VFP. Of course, the ideal answer to that is to move to a client/server system and store the large data sets on a back-end system. But that means rewriting large portions of the existing code. Plus, we'd have to set up the database server and learn how to administrate it, how to write queries for its dialect of SQL, and just what limitations there are in the server.

Andy: Well, you're certainly right about setting up a back-end server. It's not just a question of "upsizing" your data—there's a whole raft of issues to be investigated and addressed. It's much more complex than moving FP data from your local machine to a file server. The other restrictions are because of deadlines, right? Can you just use views to get manageable subsets?

Paul: Does a remote view suffer from the 2GB limit?

Andy: Actually, I don't know, but I'd think so. The 2GB limit is a by-product of the way locks are applied (VFP uses memory addresses beginning at 2GB to lock records—if your file exceeds 2GB, this would corrupt the locking mechanism). Since a view will create a disk file (albeit a temporary one), I'd expect the limit to apply.

Paul: Exactly! Anyway, the existing programs are written to use the FoxPro tables—for example, production of billing data requires a SCAN so that every invoice line can be processed. This system has just grown from a small app that used to deal with a few thousand—not millions—of records! While the server DBA experts are setting up the back-end system, we need to carry on.

Andy: Okay, then, I guess you have stay with a FoxPro solution to the problem. What are you thinking?

Paul: Well, I'm thinking of partitioning the data. That should be simple. For example, take a billing system with BillHeader and BillDetail tables. Just add an extra column to the header table, indicating *which* detail file contains the records for that bill. Then you can use multiple tables for the bill details, so long as the detail lines for one header line aren't split across more than one table.

Andy: It's a little more than "just" adding a column, isn't it? For example, a SQL select couldn't just join BillHeader and BillDetail anymore; and, yes, you really would be in trouble if your header file exceeded 2GB!

Paul: But if you take a typical enquiry screen, you'll first look up the header information and then extract a subset of detail information. For example, you might want all of the bills for a customer to be displayed, showing just the header information. Then you can drill down and view a particular bill, or print it. That can use the header table for summary information, then the particular bill detail table necessary for the line item retrieval.

Andy: So how are you going to partition the data? How do you decide what range of keys you're going to assign to each table?

Paul: Well, in this particular case, the requirement is simple—size alone. There's no easy way of partitioning the data by customer number yet ensuring that each partition won't exceed 2GB. The partition boundaries would potentially need to be adjusted monthly.

Andy: True. Don't forget that this is also a problem in both Oracle and SQL Server, where your overall database size is pre-defined and therefore limited. When your total data store exceeds that limit, you have to stop, back up, extend the database, and then import your data from the backup. This isn't a trivial operation, especially when you need the system up and running at all times.

Paul: Well, remember, for now I'm just looking for *a* way to handle very large tables in FoxPro. Not necessarily the *best* way—just one that can be implemented in the minimum time to meet this one specific need. I'm not interested in partitioning in the same way as Oracle implements it, with set boundaries for each partition.

Andy: In which case, partitioning on physical size sounds like the way to go. Do you have a strategy for doing that already, or do I have to exercise my brain?

Paul: I've already put together some basic requirements—here they are:

- Store more than 2GB of data partitioned among VFP tables.

- Provide an interface to ensure a number of records can fit in one physical file.
- Provide a single alias no matter which partition is in use.
- Allow insert or append/replace into the underlying table.
- Provide interface to return partition number.

Remember, all I'm interested in is writing blocks of detail records to the very large table and knowing which partition they went into so that I can store that in the header file.

Andy: First, can I assume that you're looking to create a class to do this? In which case, I also assume that by "interface" you're referring to the Public Interface rather than to a UI?

Paul: Yes, I expected you to assume we were going to define a class because we *always* (well, almost always) end up talking about class definitions. I should have been clearer.

Andy: So we're looking at a non-visual class, and we need to define the responsibilities to fulfill that specification. Then we can go on to look at the methods and properties to implement them.

Paul: Okay, then—you start <g>.

Andy: Let's start by considering writing data. What does this class need to do in order to write data? It needs to: 1) Ensure that the specified table is open; 2) Ensure that the data to be written doesn't cause the currently selected partition to exceed 2GB; 3) Write the data; and 4) Close the table.

Paul: Hang on a minute, those responsibilities could be implemented in many different ways! They don't necessarily result in functionality that meets my requirements.

Andy: No, but they do define the responsibilities of a Very Large Table root class that's generic and could provide the basis for a range of subclasses, giving you much more functionality than you need right now—your requirements will change, you can be sure of that <g>.

Paul: I think I'm expecting your high-level (head in the clouds) responsibilities to meet my specific implementation. I can think of quite a few lower-level responsibilities that appear to be missing. How about: 5) Open the first partition; 6) Open the next partition; 7) Calculate the size of the data to be written; 8) Calculate the size of the current partition; and 9) Provide the number of currently open partitions.

Andy: I'd say that those are certainly key methods that are needed to implement the responsibilities that we've defined. I'm not sure that they're responsibilities of the class.

Paul: Well, I think I'm talking about responsibilities—they may well *be* methods in this implementation, but I'm still thinking of responsibilities and not methods at this point. For example, these could easily be delegated to another class (like your data classes), and they might well not be responsibilities of *this* class at all. But they're responsibilities that are required by the higher-level responsibilities and might well be implemented by collaborations between classes.

Andy: Okay, in that case, I absolutely agree with you. As we always say, this lower level is where you should be applying the must/should/could tests.

Paul: Go on, then . . .

Andy: Let's take opening the file. At the generic level, we have to make a decision. It has to be given the stub of the tablename (for example, BillDetail) and the alias under which the partitions should be opened. That's an absolute must. It must then check to see whether that alias name is already in use. It could fail because it's in use, or it could succeed because the alias is already open.

Paul: We need to define *the* behavior that we want, then; otherwise, we'll have an infinite number of subclasses that all do different things. It'll be unusable and unmaintainable—which never stopped anybody from doing it, right?

I decree that the behavior I want is that the alias must *not* already be in use. If it's in use, there's an error and we can't proceed. Raise an error 9000, "It ain't worked, pal."

Andy: Gasp! A design decision! It must open the first partition using the stub table name plus however we identify the individual partition tables—by the way, can we call these individual tables "extents," please?

Paul: Yes, you identify an extent by *LTRIM(STR(extent_number))*, so "BillDetail Extent 1" would be in *BillDetail1.dbf* and Extent 25 in *BillDetail25.dbf*. I don't think making it fixed-width leading 0 padded makes any kind of sense.

I assume that we're going to handle "normal" errors like file not found, memo file missing, index doesn't match table, and so on as a matter of course—one's error handler handles those things.

Andy: The next thing, I suppose, is to write some data. That raises another design decision (you really didn't specify this clearly, did you?)—namely, should we have to specify *which* extent the data is placed into, or do we allow the class to decide? Or do we care?

Paul: Hmm, tricky. My hidden assumption is to use the

extents in order, starting at extent 1 and working through. Many extents might exist already and some could be empty, or only extent 1 might exist.

Andy: So shouldn't it open at the last logical (highest numbered) extent that contains data and can accept more data? If there are no extents that can accept more data, should it create a new one?

Paul: You bet; yes, it should. I reverse that previous decision, and the requirement that it start at the first partition!

Andy: <lol> I thought getting a decision was too good to be true.

Paul: Oh, and if no extent is available, create a new one <sulk>.

Andy: Right. That's the responsibilities for opening the table taken care of—we have:

1. Check that the specified alias isn't already in use (raise error if it is).
2. Open, as the specified alias, the highest numbered extent that contains data and that's less than 2GB. Create a new extent if none is available.

   How do we know how much data we want to add?

Paul: I suppose it depends on how we're going to pass the data to be inserted to the VLT class. All you need to know is the number of rows to be inserted, since you can calculate *RECSIZE()*nROWS* to find the size of the data. So we need to decide how to pass the rows—in an array, a named cursor, a source table and a WHERE clause . . . there are so many ways to skin the Fox. I think we should place the data to be inserted into a special cursor.

Andy: Remember that this cursor can't be the result of a SELECT statement that produces a filtered view. You'd have to use the NOFILTER clause.

Paul: Of course! More importantly, the "transfer" cursor can't exceed 2GB, either! Then we simply pass the name of the cursor.

Andy: Good! We get the number of rows from the cursor and calculate the amount of space required by multiplying by the RECSIZE() of the extent. To actually determine whether there's enough room, we need the true extent size, which is HEADER() +1 + (RECCOUNT() * RECSIZE()). If this value plus the amount of data is more than 2GB, then we need to open (or create) the next logical extent and use that instead.

Of course, if some fool uses a transfer cursor with a smaller record size and has enough records to exceed 2GB in the extent, then this is also an error and we shouldn't go into an infinite loop. Filling the hard disk with empty extents isn't a good idea!

Paul: Oh, by the way, I want to make the maximum extent size configurable, so we'll just add a property. Then, if for performance reasons I want to use 10x 1G files or 20x 0.5G files, I can make the class do what I want. Now, having gotten the size of the data and determined that we can write it, we need a method to know where it was written to.

Andy: How about if the method that writes the data returned the extent number it wrote to, and -1 if it failed?

Paul: Yeah, that'd work! You've combined what I had as two requirements into one implementation.

Andy: Just as an aside, I assume that the write method will test for BufferMode and handle the TableUpdate appropriately.

Paul: Of course, either we use your data classes and call TryUpdate(), or, well, it's an exercise for the reader, isn't it? So our third responsibility can be defined as determining the size of the data to be written and adding it to the current extent if there's room, or using the next logical extent (creating one if necessary) if there isn't. The return value must be either the extent number that was written to, or -1 if the write failed.

The fourth, and last, responsibility is to close the table—which is really easy—USE IN *aliasname* because the extent is always opened in the same alias name.

Andy: So we only have four responsibilities in the class after all. However, all this does is write data. How do you propose to get the data back out of these tables?

Paul: Actually, I'd already thought of a few more useful things—which means more requirements:

- An exposed interface like SKIP to move to the next record, crossing partition boundaries transparently.
- An exposed interface like GO TOP | BOTTOM across all partitions.
- An interface to execute a SQL statement across all partitions, returning a union of results.

Andy: I told you your requirements would change <g>. We haven't even finished defining the class, and you're extending it!

Paul: Well, if you want to get really fancy:

- Given a Very Large Table name (for example, "Customer")
- Hold control data for partitioning in a special associated table (for example, named "CustomerVLT")

- Extent Tables "Customer1" through "Customer9999" hold the actual data
- Provide an interface to define the column used for partitioning and values per partition

Andy: I'm not sure I see what you're driving at here.

Paul: Well, you can optimize a select for a partition column value (or range)—for example, selecting only from the associated table. You can define value-based or space-based partition criteria, making it all the more flexible.

Andy: That's a good thought. I like the idea, though implementing it could prove to be a bigger job than we can comfortably manage within the confines of the column. What we do have here isn't a complete solution to partitioning, but it hits enough of the high points, so that could form the basis of a real solution.

Paul: If any readers decide to produce their own VLT class, please let us know, or even better, why not share it on CompuServe? Post it (in ZIP file format, please) in GO VFOX Library Section 3, prefix the description with "FH:" so it sits with the "FreeHelp" files.

*[**Paul:** I'd like to acknowledge the assistance of Des Bandoo in producing this; not only did he comment on the finished article, but he also developed the production version of our VLT classes (sorry, we can't share those). Just like Andy, he had to work with my changes of mind, and he produced a great solution that was the spur for writing this article. There you are, Des—your name's in print now!]* ▲

Paul Maskens is a VFP specialist and FoxPro MVP who works as programming manager for Euphony Communications Ltd. He's based in Oxford, England. pmaskens@compuserve.com.

Andy Kramek is an old FoxPro developer, FoxPro MVP, independent contractor, and occasional author, currently of no fixed abode, working in Buffalo, NY. andykr@compuserve.com.

Downloads May Subscriber Downloads

- 05FALINO.ZIP—Source code described in Jim Falino's article, "Turn Your VFP App Client/Server: A 12-Step Program, Part 2."
- 05DONNIC.ZIP—Source code described in Jeff Donnici's article, "Best Practices: Seeing Patterns: The Singleton."
- 05COATSC.ZIP—Source code described in Andrew Coates' article, "Matchmaker, Mitchmoker . . . Is That a Match? Or De-mystifying De-duplication."

Extended Articles

- 05BAKER.HTM—"Supercharged Date Entry!" Are both you and your clients tiring of the same old date entry blues—month-month slash day-day slash year-year? It's 1999, and our programming language of choice is Visual FoxPro, not COBOL. Maybe there's a newer, better way for date entry—a Visual way. Jeff Baker explains.
- 05BAKER.ZIP—Source code described in Jeff Baker's article, "Supercharged Date Entry!"
- 05BOOTH1.HTM—"Where is That Control?" Containership: We've all heard the word. "Visual FoxPro has a very good containership model." What is containership, and why do we care about it? Using containership to our advantage will allow us to create classes that are reusable and flexible. Jim Booth shows you how.
- 05BOOTH2.HTM—"Buffering, the Vampire Slayer: The Continuing Story." Last month, you saw an introductory discussion of the buffering technology in Visual FoxPro. Jim Booth reviewed the various buffer modes and presented a comparison of the FoxPro 2.x methodology of indirect edits and VFP buffering. This month, he discusses the TableUpdate and TableRevert functions and explains how they're used with data buffering.
- 05ZIMMEL.HTM—"Help Eliminate DBF Corruption with a Posting Engine." Steve Zimmelman describes a long-term solution to problems with the DBF file—forcing the multi-user application to behave like a single-user system by developing what's become known as a *posting engine*.
- 05ZIMMEL.ZIP—Source code described in Steve Zimmelman's article, "Help Eliminate DBF Corruption with a Posting Engine."
- 05HENTZE.HTM—"Visual Basic for Dataheads: Creating the User Interface: VB Forms and Controls." Now that Whil Hentzen has covered the language, you should have a good foundation under your belt. It's time to look at the tools that we'll use to create the most interesting part of the application—from the user's perspective, that is. Much like an automobile and an 18-wheeler, some of the tools and techniques will be familiar to experienced VFP developers, but other techniques and components are either brand new or just simply different. Let's explore.

FoxTalk Subscription Information: 1-800-788-1900 or http://www.pinpub.com

Subscription rates:

United States: One year (12 issues): $179; two years (24 issues): $259
Canada:* One year: $194; two years: $289
Other:* One year: $199; two years: $299

Single issue rate: $17.50 ($20 in Canada; $22.50 outside North America)*

European newsletter orders:
Tomalin Associates, Unit 22, The Bardfield Centre,
Braintree Road, Great Bardfield,
Essex CM7 4SL, United Kingdom.
Phone: +44 1371 811299. Fax: +44 1371 811283.
E-mail: 100126.1003@compuserve.com.

Australian newsletter orders:
Ashpoint Pty., Ltd., 9 Arthur Street,
Dover Heights, N.S.W. 2030, Australia.
Phone: +61 2-9371-7399. Fax: +61 2-9371-0180.
E-mail: sales@ashpoint.com.au
Internet: http://www.ashpoint.com.au

* Funds must be in U.S. currency.

Editor Whil Hentzen; **Publisher** Robert Williford;
Vice President/General Manager Connie Austin;
Managing Editor Heidi Frost; **Copy Editor** Farion Grove

Direct all editorial, advertising, or subscription-related questions to Pinnacle Publishing, Inc.:
1-800-788-1900 or 770-565-1763
Fax: 770-565-8232

Pinnacle Publishing, Inc.
PO Box 72255
Marietta, GA 30007-2255

E-mail: **foxtalk@pinpub.com**

Pinnacle Web Site: **http://www.pinpub.com**

FoxPro technical support:
Call Microsoft at 425-635-7191 (Windows)
or 425-635-7192 (Macintosh)

FoxTalk (ISSN 1042-6302) is published monthly (12 times per year) by Pinnacle Publishing, Inc., 1503 Johnson Ferry Road, Suite 100, Marietta, GA 30062. The subscription price of domestic subscriptions is: 12 issues, $179; 24 issues, $259. **POSTMASTER:** Send address changes to *FoxTalk*, PO Box 72255, Marietta, GA 30007-2255.

The Subscriber Downloads portion of the *FoxTalk* Web site is available to paid subscribers only. To access the files, go to www.pinpub.com/foxtalk, click on "Subscriber Downloads," select the file(s) you want from this issue, and enter the user name and password at right when prompted.

User name: imagine
Password: jolly