Chapter 5
Data - Views

Views, a new feature introduced with Visual FoxPro 3.0, provide a way to take a new approach to many aspects of our applications. At the simplest, views represent a way of storing queries within the Visual FoxPro database container. They can, if desired, be used exclusively to provide the interface between forms, code and the data tables.

The term "view" is used interchangeably to identify two different but related things. The first is the entity that is stored in the Visual FoxPro database—that is, the SQL-Select statement and related properties that define the view— and the second is the temporary cursor that is created when the view is opened. There is virtually no difference between opening a view with USE <ViewName> and executing a query using the INTO CURSOR <cursorName> clause. The difference lies in the behavior of a cursor created using a simple SELECT…INTO statement when compared with one created by opening a view.

Views vs. queries

The following features compare and contrast views and queries:

- A view is opened just as a table is—by issuing the USE <ViewName> command; a query is executed by issuing the SELECT…FROM…INTO command in code, or by executing a query stored as a .QPR file. Opening a view requires that the database containing the view definition is open and is currently selected (which can be accomplished with the SET DATABASE TO command).
- Views do not support the TO, INTO ARRAY and INTO TABLE options of a SELECT-SQL statement. While the output from queries can be directed to a table or array, or to a text file, a printer or the screen, views direct their output only to cursors, and the cursor alias is the same as the name used to identify the view.
- Views can be constructed with runtime parameter variables. These parameter variables can be pre-populated, or they'll generate a runtime prompt for a value if the view is opened without establishing a value for the parameter prior to opening the view.
- Cursors resulting from opening views can be modified. A cursor resulting from an SQL-SELECT is opened as a read-only file.
- View fields share all of the properties belonging to table fields, including Caption, Comment, DefaultValue, DisplayClass, DisplayClassLibrary, Format, InputMask, RuleExpression, and RuleText.
- Views share some properties with tables, such as RuleExpression and RuleText.
- Changes made to a view-based cursor can (conditionally) be written back to the tables on which the view is based.

The last four items in the above list—the "writability" of views, the table-like properties for views and view fields, and the fact that changes made to views can affect the underlying tables—represent the real significance of views for application development.

A read-only view—that is, a view that is intended only to display, not update, our table data—is not much different from a query. A query can incorporate memory variables in its WHERE clause that can be assigned values programmatically, join tables, create "virtual" or "calculated" columns, and so on. The fact that it is more convenient to store views in the database instead of queries in code or .QPR files has caused queries to be used somewhat less often than was the case with versions of FoxPro prior to version 3. However, this doesn't differentiate the *function* of these views within an application—merely the way in which they are *packaged*.

The remainder of this chapter will focus on the utility of views for manipulating the data in our applications.

How views work in an application

The short answer to the question "How do we use views in our applications?" is "Just like we use tables." In every case, any technique, feature or syntax used with a table can also be used, 100 percent unchanged, with a view. Well, almost.

- If you want a form control to update a field in a table, you specify Alias.FieldName as the control's ControlSource property. If you're working with a view, you use (again) Alias.FieldName.
- If you want to programmatically change a field in a table, you use REPLACE <Alias.FieldName> WITH <value>. For a view, you still use REPLACE <Alias.FieldName> WITH <value>—no difference. In fact, most commands in the Xbase language, such as SCAN…ENDSCAN, SET FILTER, SCATTER/GATHER, COPY TO, and so on, work on views just as they do on tables.
- If you want to move to the next record, you use the SKIP command; to skip to the first record, use the LOCATE command; to skip to the last record, use the GOTO BOTTOM command. Same for views. If you want to change the order of the items in a table, you SET ORDER TO <index tag>. To change the order of items in a view, you SET ORDER TO <index tag>.
- When you're through editing records in a buffered table and want to commit the changes to the tables, you use the TableUpdate() function. When working with views, you use the TableUpdate() function.

With the last two items in the list above, we begin to get into a couple of areas in which working with views is a little different than working with tables, and you've probably noticed that we haven't made any reference to relations or parent/child tables.

We'll get to these aspects of working with views a little later. For now, the important thing to be aware of is that working with views differs *very little* from working directly against tables.

That little something extra

Note that, in the list presented above comparing views and queries, a view's fields share all the properties of table fields. This allows the view fields to operate interchangeably with table fields within an application. As mentioned in the previous section, all commands and functions or techniques that we might use with table fields will work just as well with view fields.

However, view fields have a whole stack of new properties that are not shared by table fields; and a bunch of properties are unique to views and not shared by tables. Most of these relate to the fact that we can update the tables on which the view is based. Some of these properties are useful only when the view is a *remote view*, that is, a view that draws records not directly from Visual FoxPro tables, but from an ODBC data source, usually an SQL database. SQL databases can include SQL Server or Oracle or an Access .MDB file or even VFP tables if you access them using ODBC. The other properties are of interest in all updateable views.

The first of these properties we need to be aware of is the SendUpdates property of the view. If .T., this indicates that changes made to the view can be sent back to the underlying table. In order for this to happen, VFP requires a critical piece of information: It must have some way of matching up a particular record in our view with the corresponding record in the underlying table. It does so by matching up (at minimum) the *key fields* that, individually or collectively, uniquely identify each record. In a properly designed database, each record in every table has a primary key value. This key can be determined by the value of a single field, or can be a "composite key" that is made up of the values of more than one field (see the previous chapter on indexes and keys). From this, it follows that we must use the KeyField property of the view fields to indicate those fields that represent the primary key value of the underlying table.

The next piece of information needed to make an updateable view "work" is which fields we want to be updated when we commit changes made to the view. This can include any number of the fields in the view, although if the view includes "calculated" or "virtual" fields, these exist only in the view, and cannot be marked as "updateable."

In the View Designer, these two properties are set by using the "Key" and "Updateable" columns, and the "Send SQL updates" on the "Update criteria" page.

Finally, we must specify exactly how modifications to the view are sent to the underlying tables. This involves two different properties: the WhereType property and the UpdateType property. You will notice option buttons to select these options are also on the "Update criteria" page of the view designer. In the view designer, these properties are set by setting the options for "SQL WHERE clause includes" and "Update Using."

The WhereType property specifies how a record in our view is "matched" with a record in the underlying table. For local views, we have three choices: use the key field(s) only, the key field(s) along with any modified fields, or the key field(s) and any updateable fields. There is a fourth choice for remote views that uses the key field(s) and the timestamp placed on the table by the database server.

From a practical standpoint, the WhereType is actually specifying the answer to the question, "How are we to detect an update conflict?" If the WhereType is set to "1–Key Fields Only," the update will be made if there is a record in the underlying table that has the same key value, *without regard to the fact that another user may have made changes to the record since the record was retrieved*. At the other extreme, if we specify a value of "2–Key and updatable" for the WhereType property, an update conflict will be reported (and the update will fail) if any

updateable field in the underlying table has changed between the time the view is opened and the time the user tries to commit his changes. In other words, Visual FoxPro looks for a matching record in the underlying table based on the key field(s) and all of the updatable fields. If one of these fields has changed, Visual FoxPro will *not* be able to find the matching record, and the update will fail. This most closely emulates the behavior of Visual FoxPro tables when working directly against the tables—if two users update the same record, the last one to attempt to save her changes gets an update conflict error.

An intermediate choice (and one of the coolest things about using views) is that if we specify a WhereType of "3–Key and Modified Fields," then Visual FoxPro examines only the fields that the user has changed, along with the key field, to find a matching record in the underlying tables. This means that if Bullwinkle modifies only field "A", and Rocky modifies only field "B" on the same record, and they both commit their changes, no update conflict occurs—both edits are saved without error.

Buffering and views

There are only two buffering options for views: optimistic row buffering (the default) and optimistic table buffering. There is no option for pessimistic buffering (views can't place a lock on the underlying tables while open) and there is no way to disable buffering for views. Since some type of data buffering is the norm for production applications, and since optimistic buffering is preferred in the majority of situations, this does not usually present a significant limitation. If a particular application requires pessimistic buffering, then views may not be appropriate.

Note: While views do not natively provide the ability to pessimistically lock a record, this does not mean that a situation in which an application requires pessimistic locking (the record is locked when it is opened for editing) cannot consider using views. Views can still be employed in this situation, but some other mechanism must be employed to determine whether a particular record can be edited, or if it is already in use by another user. This can be accomplished by using a "semaphore" locking scheme, in which a shared table is used to keep track of tables and records open for editing. This table can additionally store information concerning the user, and the date and time the record was locked. One of the slickest ways of implementing such a semaphore-locking scheme I've heard of is to actually get a lock on the semaphore table record. If another user needs to access the record and cannot lock the corresponding semaphore record, then the user is either locked out, or the record is opened "read only". The advantage of this approach is that if a user who has a lock on the record experiences a crash or a power failure, the server automatically removes the lock, making the record available to other users on the system without the need for an administrative utility to "unlock" the record.

When a query is executed in Visual FoxPro, all tables named in the query's FROM clause are automatically opened. This is also true when using local views. These tables are not buffered, so when a TableUpdate() is issued on a local view, the modifications are written immediately to the underlying table. While it is possible to explicitly open each of the underlying ta-

bles, either in code using the USE command or by placing them in a form's DataEnvironment and employing some kind of buffering scheme on them, there is no benefit in doing so. Indeed, this requires that a TableUpdate() be issued first on each view, and then a second time on each of the underlying tables in order to commit the changes the user has made to the view. As in running a query, the underlying tables remain open. Closing the cursor holding the result set does not close the underlying tables. Thus there is no difference (with one exception) between allowing a view to open its underlying tables automatically or by placing those tables in the DataEnvironment—tables added to the DataEnvironment use the form's buffering setting (none by default). The exception is when you're using the Default Data Session. The view will be closed when the DataEnvironment is destroyed (if AutoCloseTables is .T.), but the underlying tables are left open. If you add the tables to the DataEnvironment, *before adding the views*, they will be closed when the form is closed. Adding the tables *first* to get this behavior is important. The tables and views are opened in the order in which their associated cursor objects are added to the DataEnvironment. If you add the views first, the tables will be opened automatically, and will be left open when the form is destroyed. If the tables are opened before the views, then the DataEnvironment takes responsibility for closing them when the DataEnvironment is destroyed.

For more information on buffering, see Chapter 7.

Indexing views

Indexing is not a core technique to using views, but it can lend so much to your applications that it deserves some discussion.

Yes, you can index a view. After the view is opened, you simply execute INDEX ON <view_name> TAG <tag_name> and you're all set. You can do this in the Init() method of the form, or in any other method. However, I've found that the best place to perform this task is in the Init() of the DataEnvironment. This event occurs before any other control is initialized, so the views are indexed and ready for use by any control. If you're opening the views with the NoDataOnLoad property set to .T., then the views contain no records when the DataEnvironment.Init() fires, so the indexes are created almost instantaneously and are then automatically updated when you execute a REQUERY() on the view. Remember that the result set for a view is stored in a cursor. When a cursor is closed, the disk-based table associated with it is deleted from your disk, and likewise any .CDX files containing indexes.

Why would you want to index a view? Here are some reasons:

- To set a relation between two views at runtime. This isn't something that you need every day, but put this idea in your bag of tricks—it might come in handy.
- To allow the user to change the order of multiple records in a grid or list box.
- To enable incremental searching in a grid, or on a column other than the first column in a list box.

For whatever reason you decide to index a view, you need to be aware of some traps for the unwary involving indexing a view:

- You cannot index a table-buffered view. You must first set the buffering to row buffering, create the index, then set the buffering to table buffering. No big deal, but you just need to be aware of the limitation.

```
* DataEnvironment.Init()
CURSORSETPROP('buffering',3,<view_alias>) && row buffered
SELECT <view_Alias>
INDEX ON <expression> TAG <tag name>
CURSORSETPROP('buffering',5,<view_Alias>) && back to table buffered
```

- You cannot create more than one index tag on a read-only view (the ReadOnly property is .T.). The .CDX file gets flagged as ReadOnly also, and creating a second index tag attempts to write to the .CDX file.
- REQUERY() or SELECT stores the number of records in the result set to the _TALLY system memory variable. If TALK is set OFF, INDEX and REINDEX do not update _TALLY, and it is reset to 0. If you're running with TALK set OFF (the usual situation) and index a view after opening it, _TALLY will contain 0, without regard to how many records were in the result set. If you REQUERY() an indexed view, _TALLY will be also be reset to 0 because the table is REINDEX()ed after the SELECT is re-executed, resetting _TALLY to 0. In these situations you'll need to use RECCOUNT() or some other technique to determine how many records are in the result set.

Parameterized views

Let's assume we create a view on our 25,000-record customer table, named "v_customer", that uses the following query expression:

```
SELECT * FROM customer
```

All of the fields are updateable, and the primary key field is flagged as the key field.

Opening this query by issuing the **`USE v_customer`** command, or by placing this view into the DataEnvironment of a form and running the form, will retrieve all 25,000 client records. We can then locate the customer record we want to work with, make changes, and then issue a TableUpdate() command to commit the changes.

You're absolutely right if you're thinking that all of this doesn't sound very efficient. First, the query executes and grabs all 25,000 records. Then, we have an unindexed cursor in which we'll have to issue a non-optimized LOCATE command to find a particular customer record (SEEK() requires an index). This is not normally the way in which updateable views are employed in an application. Instead, the views are *parameterized*, that is, a filter condition is established in the view that determines which subset of records are retrieved, using a special memory variable that establishes the filter at runtime.

A view parameter is established by specifying a memory variable as the comparison value in a WHERE clause, and preceding the name of that memory variable with a question mark, as in the following example:

```
CREATE SQL VIEW v_customer AS ;
  SELECT * FROM customer WHERE cCust_ID = ?vp_cCust_ID
```

Many developers are familiar with specifying a memory variable in a WHERE clause. The difference between doing this and creating a parameterized view is that if the memory variable does not exist when the view is opened, an error will occur. On the other hand, in a parameterized view, if the memory variable does not exist, a dialog pops up, prompting you for the value of the variable. However, these types of views aren't usually used this way in a production application. The usual practice is to either open the view using the USE command, but with the NODATA clause, or more commonly, to place the view into the DataEnvironment of a form, and set the NoDataOnLoad property to .T.

Here are the effects of establishing a view parameter and opening the view in this way:

- No error is triggered because the memory variable specified in the WHERE clause doesn't exist.
- The dialog prompting the user for a parameter value is *not* triggered.
- The view is opened without any records, so it appears exactly like a newly defined table does before any records have been added.

After the view is opened using the NODATA keyword, a value can be assigned to the parameter variable and the REQUERY() function can be called. This will cause the SELECT command to be re-executed, retrieving all records that match the condition specified by the WHERE clause and the value of the parameter.

For example, look at the v_Employees view established in the Time and Billing sample database. All fields are selected, and the WHERE clause (as shown on the FILTER page in the view designer) is:

```
WHERE Employees.cEmployeeNumber = ?vp_cEmployeeNumber
```

Make sure that you SET STATUS BAR ON, and from the command window, issue the following commands:

```
USE v_Employees NODATA
vp_cEmployeeNumber = "5"
? REQUERY()
```

Note that after issuing the USE command, the status bar briefly showed "Selected 0 records in .02 seconds", and then showed "v_employees Record: None", indicating that we have a data entity with the alias "v_employees" open in the current work area, and that it contains no records. After assigning the character value of 5 to the memory variable vp_cEmployeeNumber, and calling the REQUERY() function, the value of 1 was displayed on the Visual FoxPro desktop, and status bar briefly displayed "Selected 1 record in .03 seconds" (your machine may execute the query faster or slower than mine does), then displayed "v_employees Record: 1/1". If you browse the result set, you will see that we've selected the record for Laura Callahan.

In general, the parameter for an updateable view will specify a value for one of three things:

- A primary key value—often for retrieving a parent table record

- A candidate key value—a non-surrogate alternative to a surrogate primary key
- A foreign key value—often for retrieving child table records

In the case of the v_Employees view, the view parameter specifies a value for the cEmployeeNumber field, a candidate key. We could just as easily establish a view parameter on the primary key value, which is an integer value found in the iEmployeeID field. However, this database uses *surrogate* keys as primary and foreign keys. Surrogate key values are system-generated and contain no business data. Surrogate keys are established strictly for maintaining a link between tables, and are seldom, if ever, revealed to the user. As a result of this, and the fact that the Employees table has no parent table, it is unlikely that the user would know the integer primary key value, but it's very likely to know the employee's employee number. Hence the logical decision to select records based on the candidate key value.

There is a little gotcha lurking with regard to parameterized views. It is always necessary to REQUERY() a parameterized view after setting the value of the parameter. Because of this, be aware that you should not flag a parameterized view as ReadOnly, even if the SendUpdates property of the view is .F. or if you have no intention of modifying it. The reason is that re-querying a view attempts to write to the table associated with the cursor, and a "Cannot update the cursor" error will result.

Note: When working with parameterized views, it's sometimes helpful to remember some "tricks" with regard to the WHERE clause.

1. BETWEEN(), which is Rushmore optimizable, can be used not just to select multiple records that fall within a range of values, but also to select records that match a single value. This is particularly useful when working with integer surrogate keys.

```
SELECT *;
       FROM <table> ;
       WHERE <primary key> BETWEEN ?vp_LowPK and ?vp_HIPk
```

If you set liLowPK and lnHIPk to the same value you get one record. If you instead set it to:

```
liLowPK = -2147483647
lnHIPk  =  2147483646
```

… you'd get all the records

2. *If exact is set OFF and you have a character key value or a filter condition applied to a character field, the empty string ("") will match all records.*
3. *If you prefer to not be dependent on the status of SET EXACT, or want to upsize to SQL Server (which behaves as if EXACT is set ON), you can use LIKE and "pad" a view parameter with "%" which is a wildcard character, analogous to the "*" used in file masks. For example:*

```
SELECT * ;
   FROM <table> ;
   WHERE UPPER(cLastName) LIKE ?vp_cLastName
```

If "SM%" is stored to vp_cLastName, the view will retrieve records for last names Smith, Smythe, Small, Smeed, and so on.

So what's the point?

If you haven't previously worked with views (or thought about them very much), you might be wondering what advantages views provide.

One very important advantage doesn't really apply to *local* views, that is, views that execute queries against tables in a native Visual FoxPro database. However, this advantage is of critical importance when using a *remote* view, which executes a query against a database server, such as SQL Server, Sybase or Oracle. With remote views, instead of your workstation examining 100,000 records on the server to determine which ones meet the filter conditions (as specified in the WHERE clause), the request is processed at the server, and only those records that meet the filter criteria are returned to the workstation. The end result is that network traffic is greatly reduced, which is a major benefit of using client/server architecture. The "searching" is done by the server when opening a remote view, but performed by the client when working with a local view. Note that, when opening a local view, all of the underlying tables are also opened to allow the workstation to determine which records meet the selection criteria.

However, many developers have discovered that there are other wonderful advantages in using local views instead of working directly against the Visual FoxPro tables. I first began using views in Visual FoxPro 3.0, simply as an exercise to check out this new feature. This "exercise" quickly evolved into my preferred method of interacting with data, so that I now use views exclusively for all database interactions. We'll come back to these specific issues later in the chapter. But for now, let me simply say that I have discovered that views are a much simpler and cleaner way to interact with data. They require a little more up-front design and implementation work, but the additional effort is leveraged mightily throughout the development process.

Views in action

Enough talk. Let's look at a form that actually uses views exclusively to interact with the database. You can examine the demonstration code by opening the CHAP5 project (Chap5.pjx) from the download files.

The form that illustrates use of views in data entry and maintenance is TIMECARD.SCX. Before proceeding further, let me stress that the sample form is "demoware." It isn't completely bullet-proof, and is not intended to be so. For instance, it's possible to add a new time card without associating it with any employee. With books, as with software, shipping is a feature, OK? Also, if you've been examining the tables and views up to this point and run with EXCLUSIVE set ON during development (as I think most of us do), remember to CLOSE TABLES ALL before trying to run the form, or you'll get "Error loading file—record number 7. cDmForm <or one of its members>. Loading form or the data environment: Error loading the data environment: Table is in use." Boy, do I *hate* that error—I must see it 20 times a day!

The form allows you to examine any time card for any employee, or create a new time card. The time card contains a reference to the employee and the date the time card was recorded, and is implemented as a single record in the table Time_Cards. The form also displays the child records stored in the Time_Card_Hours table, which represents the detail of the time

card. This detail includes the date worked, start and ending times, the project worked on, and the nature of the work performed, stored both as a code in an integer field and as free-form text in a memo field.

An existing time card is selected by first selecting an employee via a drop-down list, which in turn populates another drop-down list with a list of time cards on file for that employee, listed by date. Selecting one date causes a set of detail records to be displayed in a grid. The grid displays the project and work code for each detail record using a drop-down list. For cosmetic reasons, the memo field for each record is not displayed in the grid, but in an edit box immediately below the grid. The form has Add, Save, Cancel, Delete and Close command buttons.

Field rules prevent empty values for the line item start and end times, and a table rule prevents entering an end time before a start time. Violation of these rules will prevent saving a record, and the form gives appropriate error messages.

Adding a record allows selection of an employee using a drop-down list, entry of a date for the time card, and entry of line-item details. Line-item details are not required to save a record. AllowAddNew is set to .T. on the grid, allowing the user to add new records simply by pressing the down arrow while in the grid. Extraneous detail records (determined by seeing a newly appended record without a value for the date worked) are automatically TableRevert()ed prior to saving the record.

Before getting into the specifics of the example form, we should take a quick look at the code in the foundation form class that is used to create the form. Note that these data manipulation methods, while in use in production applications, are expanded upon and made more flexible by the data handling objects and techniques discussed in Chapter 7.

cDMForm methods

The form for this chapter's example is based on a foundation form class, cDMForm, designed to be data-aware, and capable of manipulation of updateable cursors. Most (though not all) methods of this form class are not in any way specific to the use of views. However, making a design decision to use views exclusively has allowed some simplification of these methods. For instance, you won't see any code checking to see if buffering is in effect for any cursor, because optimistic buffering is *always* active for a view. Likewise, it isn't necessary to check to see if we're working with a view or a table when using a function like CURSORGETPROP('SendUpdates'), which applies only to views.

Init()

The Init() event method is set up to accept an argument containing a reference to an application object, and to pass this object reference to the basic form class from which cDMForm is subclassed. More important to this example is the call to the classes' BuildCursorArray() method.

BuildCursorArray()

In order to write reusable code to manipulate the cursors that may be opened by a form, it is necessary to maintain a list of the cursors with which we need to be concerned. This means that when checking for changes to data, saving changes to data, discarding changes to data, and so on, we need to concern ourselves with a subset of all tables and views that might be open while a form is running. The Chapter 5 demonstration form TIMECARD.SCX has 11 tables and

views open while running. Only two of these are actually updateable views that the form is intended to manipulate.

Unlike many other container objects in Visual FoxPro, the DataEnvironment object does not have a collection property for its contained cursor objects. It's necessary to build and maintain our own collection of cursors. While we're at it, as suggested in the previous paragraph, we might as well create our collection with only those cursor objects that represent updateable views. This makes further checks for the updateability of a cursor in other methods unnecessary. BuildCursorArray() stores to a form array property, a reference to each Cursor object in the DataEnvironment that represents an updateable view.

When committing changes, the order in which the various cursors are updated is often very important. Consider the situation in which you are adding a new set of records that includes a parent record and a set of child records. If there is an insert trigger on the child table that prevents insertion of child records without a corresponding parent record, it will be necessary to commit the new parent record first, and then the child records. To enable this behavior, there must be some way, at design time, to specify this update order. The BuildCursorArray() method relies on a design decision to always have a numeric digit as the last one or two characters of the cursor object's name, and contains an ASSERT to enforce this. This allows the developer to specify, by way of the last characters of the cursor names, what order is to be used when updating the cursors. This makes it possible for BuildCursorArray() to sort the array of cursors in the order in which they are to be updated.

The view-specific part of this method is the check for whether the view is updateable, using the CURSORGETPROP('SendUpdates') function. There is also an interesting bit of code that stores a connection handle to the form if the view is a remote view. Limiting connections in a client/server application is highly desirable (it minimizes the chances of running out of resources on the server) so all remote views should be established using the SHARE clause. This, together with using an existing connection handle for any SQL pass-through commands, means that your entire application can run using only a single connection to the SQL server (see **Listing 5-1**).

***Listing 5-1**. The BuildCursorArray() method.*

```
* BuildCursorArray() Method
LOCAL lnCursors, ;
   lcReference, ;
   loReference, ;
   lcAlias, ;
   x, ;
   lcCursorName

LOCAL ARRAY laCursors[1]
* Create an array of all objects in the
* form's DataEnvironment. Since we will eventually
* be winnowing this list down to only the cursors
* we might as well start out calling the array laCursors
* and the number thereof lnCursors
lnCursors = AMEMBERS(laCursors, THISFORM.DATAENVIRONMENT, 2)
IF lnCursors = 0
   THISFORM.nCursors = 0
   RETURN
ENDIF
```

```
* Swap the name stored in each element
* with the object that the name refers to
FOR x = 1 TO lnCursors
   IF VARTYPE(laCursors[x]) = "L"
      EXIT
   ENDIF
   lcReference = "THISFORM.DataEnvironment." + laCursors[x]
   loReference = EVALUATE(lcReference)
   * just in case someone sneaks in a relation on
   * us, we'll get rid of it
   IF LOWER(loReference.BASECLASS) = "relation"
      ADEL(laCursors,x)
      lnCursors = lnCursors - 1
      x = x - 1
      LOOP
   ENDIF
   laCursors[x] = loReference
ENDFOR

* Eliminate from the array any references to
* cursor objects that represent an updateable
FOR x = 1 TO lnCursors
   IF VARTYPE(laCursors[x]) = "L"
      EXIT
   ENDIF
   lcAlias = laCursors[x].ALIAS
   IF x = 1 AND THISFORM.nConnectHandle = 0
      * Nothing has established a connection handle
      IF CURSORGETPROP("Sourcetype",lcAlias) = 2
         * This is a remote view, so we'll grab the
         * connection handle for this view so it's
         * available for the use of any SQL pass-through
         * commands we may need to execute
         THISFORM.nConnectHandle = CURSORGETPROP("ConnectHandle",lcAlias)
      ENDIF
   ENDIF
   IF CURSORGETPROP("SendUpdates",lcAlias) = .F. OR laCursors[x].READONLY
      * This cursor represents a read-only entity,
      * so we'll get rid of it
      ADEL(laCursors,x)
      lnCursors = lnCursors - 1
      x = x - 1
      LOOP
   ENDIF
ENDFOR
* Transfer the array of cursors and the
* number thereof to the form
THISFORM.nCursors = lnCursors
DIMENSION THISFORM.aCursors(THISFORM.nCursors, 2)
FOR x = 1 TO THISFORM.nCursors
   THISFORM.aCursors[x,1] = laCursors[x]
ENDFOR

* Store the 'number' of the cursor object
* to the second column of the form array property
FOR x = 1 TO THISFORM.nCursors
   lcCursorName = THISFORM.aCursors[x,1].NAME
   ASSERT ISDIGIT(RIGHT(lcCursorName,1))
```

```
   * If this assert fails, the developer didn't
   * follow the  practice of ending each cursor
   * name with a 1 or 2-digit numeric value
   IF ISDIGIT(LEFT(RIGHT(lcCursorName,2),1))
      THISFORM.aCursors[x,2] = VAL(RIGHT(lcCursorName,2))
   ELSE
      THISFORM.aCursors[x,2] = VAL(RIGHT(lcCursorName,1))
   ENDIF
ENDFOR
* Now, sort the aCursors array according to
* the 'number' of the cursor
ASORT(THISFORM.aCursors, 2)
```

Our technical editor has pointed out that the BuildCursorArray() method in Listing 5-1 makes an assumption that we won't be mixing local and remote views, hence the check for a connection handle only when encountering the first view. You'll need to modify this bit of code if you have a database that does indeed mix the two view types, because the first view encountered may not be remote.

Changed()

An important service that the form needs to provide is the ability to detect if the data being displayed by the form has been changed. This is needed when the user is in mid-edit and tries to close the form, or if the application tries to close the form programmatically. We want to allow the user to save his changes before the form is closed, discard the changes, or cancel whatever operation is closing the form. This whole mechanism for the cDMForm class is handled by the QueryUnload(), Close() and OkToClose() methods. However, they all rely on the Changed() method to determine the situation with regard to the form.

This method works by detecting "dirty" buffers, or data buffers in which some value has been changed since the data was retrieved from the disk where it is stored. To do this, it relies on the GETFLDSTATE() function. GETFLDSTATE() does a good job of distinguishing between the following four conditions:

- A newly appended record, unchanged
- An existing record, unchanged
- A newly appended record, changed
- An existing record, changed

A shortcoming of GETFLDSTATE(), and therefore a shortcoming of this method, is that it cannot distinguish between a value that was changed by the user but was later changed back to its original value, nor can it distinguish between a modification that was initiated by the user and a modification that was triggered in a newly appended record by a default value or the execution of a rule.

You will note the comment in this code that all views should be table buffered, even if you have only one record. This is because of the fact (discovered and documented by our able technical editor) that certain operations, contrary to expectation, will attempt to move the record pointer. If the view is row buffered, this will cause an automatic TableUpdate(), which is to be avoided at all costs. The update can fail, especially since the user hasn't finished with his data entry. And without the code written in the Save() method (discussed below), the user could be

faced with a cryptic Visual FoxPro-generated error dialog, rather than the user-friendly and informative message that the HandleError() method will present. However, the Changed() method is bracketed to handle either type of buffering, so it won't inadvertently trigger an update if the cursor, for some reason, isn't table buffered.

One significant advantage of using local views is that you have in your possession only a subset of records. This means that you can act (when appropriate) on all records, rather than trying to restrict any operation to a subset of the records of a table using SCAN FOR or filters or some other mechanism. This gives rise to a view-specific behavior of this method, which is that there is no limitation on the SCAN…ENDSCAN applied on table-buffered views. If this code were to be executed on a million-record, table-buffered table, this could have a noticeable impact on performance.

One might suggest, instead of relying on GETFLDSTATE() to detect changes, to use the GETNEXTMODIFIED() function instead. However, GETNEXTMODIFIED() considers a newly appended blank record to be a modified record, whether or not it contains any data, and it is still necessary to use GETFLDSTATE() to determine if the record can be discarded. Also, because of Visual FoxPro's near-light-speed handling of its native cursors, the SCAN…ENDSCAN executes in the blink of an eye, even with thousands of records to examine. Finally, the SCAN…ENDSCAN loop terminates immediately if it detects a changed record. See **Listing 5-2**.

***Listing 5-2**. The Changed() method.*

```
* Changed() method
LOCAL lcStatus, ;
   lcAlias , ;
   lcOldAlias, ;
   lnCurrentRecord, ;
   llRetVal, ;
   x
lcOldAlias = ALIAS()
* Loop through our collection
* of updateable cursors
FOR x = 1 TO THISFORM.nCursors
   lcAlias = THISFORM.aCursors[x,1].ALIAS
   * Check to see if the cursor is table buffered
   * (they should all be table buffered)
   IF CURSORGETPROP('buffering',lcAlias) = 5
      * Store the current record
      * Change to the work area of the cursor
      SELECT (lcAlias)
      * The following line deals successfully with empty cursors
      lnCurrentRecord = ;
IIF(EOF(lcAlias),RECCOUNT(lcAlias),RECNO(lcAlias))
      * Check all records to see if they've changed
      SCAN
         lcStatus = GETFLDSTATE(-1,lcAlias)
         * Note - GETFLDSTATE() returns .NULL.
         * if at EOF()
         IF !ISNULL(lcStatus) ;
               AND ("2" $ lcStatus ;
               OR "4" $ lcStatus)
            llRetVal = .T.
```

```
            * If this record has changed
            * proceed no further
            EXIT
         ENDIF
      ENDSCAN
      * Don't position the record pointer
      * if we're dealing with an empty cursor
      IF lnCurrentRecord > 0
         GOTO lnCurrentRecord
      ENDIF
      IF llRetVal = .T.
         EXIT
      ENDIF
   ELSE
      * Yadda, yadda, yadda....
      lcStatus = GETFLDSTATE(-1,lcAlias)
      IF !ISNULL(lcStatus) ;
            AND ("2" $ lcStatus ;
            OR "4" $ lcStatus)
         llRetVal = .T.
         EXIT
      ENDIF
   ENDIF
ENDFOR
IF ! EMPTY(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF
RETURN llRetVal
```

Cancel()

The Cancel() method (see **Listing 5-3**) allows you to execute the TABLEREVERT() function on all updateable cursors, and uses the cursors collection created by the BuildCursorArray() method. While it seems that many developers believe forms should always be in "edit mode," there are circumstances, and sometimes entire applications, that require the user to explicitly put the form into an "edit mode" before he can modify the data. Hence the inclusion of a reset of the form's lEditMode property in the Cancel() method. If interested, you can examine the lEditWatch property and the Refresh() method of the foundation control classes in the cContrls.VCX class library that is included in the CHAP5 project.

***Listing 5-3**. The Cancel() method.*

```
* Cancel() method
LOCAL lcAlias , ;
   x
* Loop through the cursors collection
* and execute a TableRevert() on each
FOR x = 1 TO THISFORM.nCursors
   lcAlias = THISFORM.aCursors[x,1].ALIAS
   TABLEREVERT(.T.,lcAlias)
ENDFOR
THISFORM.lEditMode = .F.
THISFORM.REFRESH()
```

Delete()

The foundation classes used in this example are based on a production application currently in development. In the past, it has been my practice to leave this method empty in the foundation class, and write form-specific delete method code in the form instance. Thus far, in the application we're developing, the assumptions inherent in this foundation class code have stood up well. The assumptions are:

- If only one table is updateable, it will be the table indicated by the InitialSelectedAlias property of the DataEnvironment object. If there is more than one updateable cursor, there will be a familial (parent/child/grandchild/great-grandchild) relationship between the updateable cursors, and the parent table will be the one indicated by the InitialSelectedAlias property of the DataEnvironment object.
- Deletion of a parent record will never require interaction with child records at the level of the form; that is, any interaction with child records is handled by the database via delete triggers.
- The interface standard for deletion is to immediately commit the change and "clear" the display of the current record, hence the call to the form's Save() and ClearCursors() methods. A confirmation ("do you really want to delete…") dialog is an option that can be implemented in the form instance, and can execute the foundation class Delete() method conditionally.

The foregoing assumptions (note the ASSERT in **Listing 5-4** below) allow the Delete() method to concern itself only with the alias specified in the InitialSelectedAlias property. The ClearCursors() method is empty at the level of the foundation cDMForm class, and is discussed later in the context of the Time Card form.

***Listing 5-4**. The Delete() method.*

```
* Delete() method
WITH THISFORM
   ASSERT ! EMPTY(.DATAENVIRONMENT.INITIALSELECTEDALIAS)
   SELECT (.DATAENVIRONMENT.INITIALSELECTEDALIAS)
   DELETE
   IF .SAVE()
     .ClearCursors()
     .REFRESH()
  ENDIF
ENDWITH
```

Save()

The Save() method does all the things that a good Save() method should:

- Provides a "hook" via a call to an empty method (BeforeSave()) that allows last-minute actions to be executed just before committing changes. Typical and form-specific actions that this method could perform are updating child records with foreign-key values, deleting extraneous records, modifying cursors based on calculated values, such as storing a total to a parent record based on a total of child record fields, and so on.

- Wraps all TableUpdate() commands in a transaction, ensuring all-or-nothing commitment of a multiple-table update.
- Calls the TableUpdate() function for each cursor in the cursors collection, in the order specified by the cursors collection.
- Executes an END TRANSACTION if all updates were successful.
- Executes a ROLLBACK if all updates were not successful, and stores all error information to a form array.
- Calls other form methods conditionally depending on success or failure of the updates: AfterSuccessfulSave() in the case of successful updates, and HandleError() and AfterFailedSave() if the saves fail.

Note: When an update is issued on a view, Visual FoxPro translates the TableUpdate() into an SQL update command, based on the view properties. If (as is usually the case) the WHERE type is specified as "Key and modified fields," then the UPDATE command has a WHERE clause that includes each of the modified fields listed, along with their old values. If the WhereType is specified as "Key and Updatable Fields", the UPDATE command will have a WHERE clause that includes all of the updateable fields. In Visual FoxPro 3.0 and 5.0, a list of fields in the WHERE clause that exceeded approximately 24 fields, the TableUpdate() will trigger an "SQL: Statement too long" error. This is because there is an internal limitation on the size of the WHERE clause. In my testing, I have not been able to consistently duplicate the 24-field limitation, but it seems to occur consistently within the range of 23 to 28 fields. Note that this limitation does not apply to remote views, nor does it apply to newly appended records.

I must admit that I've never triggered this error in a production application. However, it would be irresponsible for Microsoft to leave this situation unaddressed. With the release of Visual FoxPro 6, this limitation has been increased to 40 fields, and the VFP development team have given us yet another SYS() function, SYS(3055) that allows us to further increase this limit, albeit with some performance penalty. Should you run up against this scenario, you can look to SYS(3055) as a way of selectively changing the limit when necessary

***Listing 5-5**. The Save() method.*

```
* Save() method
LOCAL lcAlias , ;
   llSuccess, ;
   x
THISFORM.BeforeSave()
BEGIN TRANSACTION
* Use the cursors collection
FOR x = 1 TO THISFORM.nCursors
   * determine the cursor alias
   lcAlias = THISFORM.aCursors[x,1].ALIAS
   * Update all rows, don't force the update
   llSuccess = TABLEUPDATE(1,.F.,lcAlias)
   IF !llSuccess
      ROLLBACK
```

```
      AERROR(THISFORM.aErrorInfo)
      EXIT
   ENDIF
ENDFOR
IF llSuccess
   END TRANSACTION
   THISFORM.AfterSuccessfulSave()
   THISFORM.lEditMode = .F.
   THISFORM.REFRESH()
ELSE
   THISFORM.HandleError()
   THISFORM.AfterFailedSave()
ENDIF
RETURN llSuccess
```

uKeyValue property, uKeyValue_Assign() and Requery() methods

The vast majority of forms have a single key value that determines the set of records that are retrieved. A primary key value for a parent table is also the foreign key value for the child records. This value is often needed by various methods of the form; rather than passing it around as method arguments and storing it using local memory variables, it's much more convenient to store it to a form property. Because this key value can be of any data type, the property in the foundation class for storing this value is uKeyValue, and has "u" (for "unknown") as its initial character.

Forms that use parameterized views in their DataEnvironment will have the cursor's NoDataOnLoad property set to .T. This allows the view to open without having to first establish and assign a value to the view parameter variable. After the views are open and the form is up and running, it is then possible to programmatically establish the proper value for the view parameter, and then call the REQUERY() function to retrieve the records that the user wants to work with. In the foundation cDMForm class, there is an empty Requery() method that is populated with form-specific code in each instance of the form.

In versions of Visual FoxPro prior to version 6, a call to the form's Requery() method immediately followed the line that stored the key value to the form property. VFP 6.0 now has something called an ASSIGN method, which is triggered whenever its associated property is modified. The ASSIGN method can be used to change the value that actually gets assigned, prevent assignment of certain values, take some action depending on the old and new values of the property, or simply execute some code in response to the changed value. This last function is the one that is intended for the uKeyValue_Assign() method. Note that the default code for an ASSIGN method accepts a parameter that holds the new value for the property being modified, and the method then stores this value to the property. As you will see, in the form instance it is important to call the DODEFAULT() function and pass the new property value before calling the form's Requery() method, because the Requery() method will expect the new value to be stored in the uKeyValue property.

Before proceeding with a detailed discussion of the sample form, take a moment to go back over the preceding methods, and notice how little difference there is from the same methods that you might write to work with tables directly. The only new method is the Requery() method, which would likely be replaced by a Lookup() method in a table-based form.

The views used by the Time Card form

The Time Card form uses six views, as illustrated in **Table 1**.

***Table 1**. Views used by the Time Card form.*

View name	Parameterized	Updateable	Comments
V_Time_Cards	Yes	Yes	The view of the parent table being updated—one record by primary key value.
V_Time_Card_Hours	Yes	Yes	The view of the child table being updated—multiple records by foreign key.
V_EmployeeListRO	No	No	A list of all employees.
V_TimeCardsByEmpID_RO	Yes	No	A list of time cards on file for a particular employee.
V_SystemCodesRO	Yes	No	A list of codes—parameter specifies which type of code is retrieved.
V_ProjectListRO	No	No	A list of all current projects.

Some views are opened by being placed into the DataEnvironment of the form, while others are opened programmatically at runtime.

Views in the DataEnvironment

Opening TIMECARD.SCX in the form designer and opening the DataEnvironment, you will see four views. You can immediately tell that they're views because they all begin with "v_", my naming convention for views. You'll notice that two of the views end with the letters "RO". This is another convention that I employ to indicate that these views are "read only," that is, their SendUpdates property is set to .F. These views are present only for the purpose of populating two drop-down lists. v_EmployeeListRO supports a drop-down list (actually, two—cboEmployeeLookup and cboEmployee), and v_Time_Cards_ByEmpID_RO supports another—cboTimeCards.

The v_EmployeeListRO view is not parameterized. It simply pulls up a list of employee names (a virtual field concatenating first and last names) along with their last names (for sorting purposes), their employee number and their surrogate employee ID values. The last is important because the employee number is not stored in any other table other than the employee table. If we need to retrieve other sets of records based on a selected employee (and we do), we'll need the surrogate key, not the employee number. The drop-down list class that uses this view is cboEmployees and is stored in the CHAP5.VCX class library.

The important properties of the combo box class are:

- BoundColumn = 3
- ColumnCount = 2 (allows the Employee number to display when dropped down)
- RowSourceType = 2—Alias
- RowSource = "v_EmployeeListRO"

- Style = 2—drop-down list
- BoundTo = .T. (the bound column is an integer type)

There is also an ASSERT in the Init() to make sure that the developer remembered to add the view named in the RowSource to the DataEnvironment.

The v_TimeCardsByEmpID_RO view is parameterized, allowing it to retrieve only those records that correspond to a particular employee. The field being matched is iEmployeeID, and the view parameter is ?vp_iEmployeeID. I think you can detect another naming convention here, related to **V**iew **P**arameters. The fields in the view are simply the date of the time card, and the primary key value for that record, which is found in the iTimeCardID field. This is the foreign key value in the Time_Card_Hours table, and will be used to establish a view parameter for that view also.

The combo box that uses this view, cboTimeCards, is also stored in CHAP5.VCX. The following are the important properties for this combobox:

- BoundColumn = 2
- ColumnCount = 1
- RowSourceType = 2—Alias
- RowSource = "v_time_cardsByEmpID_RO"
- Value = 0
- Style = 2—drop-down list
- BoundTo = .T. (the bound column is an integer)

The other two views are the “meat” of the form. These are the two updateable views that are actually being manipulated. v_Time_Cards duplicates the structure of the Time_Cards table, and v_Time_Card_Hours duplicates the structure of the Time_Card_Hours table. All fields are updateable, and the primary key field for both views has a default value of NEWID(“<tableName>”).

You might be wondering why I’m establishing the primary key value in the view rather than at the database level. First, there’s no rule that says you can’t do both (which is the case here.) Unlike the behavior in VFP 3, a default value dirties the buffer. Thus, if it’s established in the view, the TableUpdate() doesn’t cause a second primary key value to be generated in the table; but it will accept and store the value generated in the view. However, the advantage with establishing the default value in the view (beyond just being able to do so) is that it’s then available to use in populating the foreign key value of any child table records that have been added. SQL Server has a feature whereby you can establish an integer column as an identity column, which is automatically incremented, and can be determined after a TableUpdate() using SQL pass-through to retrieve the @@identity value. However, if you plan on upsizing an application to client/server, or (the gods forbid!) you have to create an application that can work with either local *or* remote data, it’s a good idea to establish a method that won’t have to be bracketed or rewritten to work in a client/server environment. Establishing the PK value in the views works in either environment.

As long as we’re on the subject of primary key values, **Listing 5-6** shows the NEWID() function that is kept in the database’s stored procedures for calculating new PK values. Chapter 4 presented a GetKey() function that serves the same purpose. However, if the PK table is

stored in a remote database, we don't have access to the Xbase functions like SEEK() and LOCATE, so I came up with a NEWID() function that will again work seamlessly across a local data and client/server environment. It uses the same table structure for the NextKey table as is used in Chapter 4.

***Listing 5-6**. A NEWID() function that relies on views and SQL rather than Xbase.*

```
* NEWID() Function
FUNCTION NewID(tcTable)
ASSERT VARTYPE(tcTable) = "C" ;
   AND ! EMPTY(tcTable)
LOCAL lnRetVal, ;
   lcOldAlias, ;
   llSuccess
lcOldAlias = ALIAS()
lnRetVal = 0
vp_cTableName = UPPER(tcTable)
IF ! USED("v_NextKey")
   USE "time and billing!v_NextKey" IN 0
ELSE
   REQUERY("v_NextKey")
ENDIF
IF _TALLY = 1
   SELECT v_NextKey
   llSuccess = .F.
   DO WHILE ! llSuccess
      lnRetVal = v_NextKey.iNextKey
      REPLACE v_NextKey.iNextKey WITH v_NextKey.iNextKey + 1
      llSuccess = TABLEUPDATE(1)
      * If the TABLEUPDATE() fails, it means that some other
      * user grabbed the key value we were about to use
      * and replaced it with an incremented value
      * so we just grab the one the other user placed in
      * the NextKey table and try again
      IF ! llSuccess
         REQUERY("v_NextKey")
      ENDIF
   ENDDO
ENDIF
IF ! EMPTY(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF
RETURN lnRetVal
```

The v_Time_Cards view also has a default value of DATE() for its tDateEntered field, and there is a rule on the tDateWorked field of the v_Time_Card_Hours view. The rule is a function in the database's stored procedures, and is shown in **Listing 5-7**.

***Listing 5-7**. A field-level rule to facilitate data entry.*

```
FUNCTION time_card_date_rule()
   REPLACE tStart WITH tDateWorked + 9*3600
   REPLACE tEnd WITH tDateWorked + 16*3600
ENDFUNC
```

This rule simply "roughs in" a start time of 9:00 a.m. and an end time of 4:00 p.m. based on the date entered for the tDateWorked field. This illustrates one of the slickest aspects of using local views. You can use two entirely different sets of rules—one set for the tables in the database that enforce data integrity, and a second set in the views that facilitate data entry. Using a field or row-level rule on a view is often much easier than trying to write code in a grid's When(), AfterRowColChange(), BeforeRowColChange() and Delete() events, or in the LostFocus(), Valid() or InteractiveChange() events of its contained controls!

Runtime views

By running TIMECARD.SCX and opening the Data Sessions window, you can quickly see the tables and views that are in use by the form at runtime. You'll notice that the following views are not opened by the DataEnvironment:

- v_SystemCodesRO
- v_ProjectListRO

Listing 5-8 shows the commands that create these two views (this is just for illustration; both views can be created in the View Designer).

Listing 5-8*. Read-only views for project and system codes picklists.*

```
CREATE SQL VIEW v_ProjectListRO AS ;
   SELECT Projects.cprojectname, ;
         Projects.iprojectid;
      FROM "time and billing!projects";
      ORDER BY Projects.cprojectname

CREATE SQL VIEW v_SystemCodesRO AS ;
   SELECT Systemcodes.cdescription, ;
         Systemcodes.icode_id;
      FROM "time and billing!systemcodes";
      WHERE Systemcodes.ctype = ?vp_cType;
      ORDER BY Systemcodes.cdescription
```

The drop-down-list picklists that use these views work a little differently than the two discussed above. First, they are a little more "encapsulated." They use a RowSourceType of 5–Array, and have their own array property to hold the contents of the two views above. Also, they open the views if they're not already open, so the views they use don't have to be added to the DataEnvironment. They work by calling their Requery() method from their Init() event. This is also convenient if these picklists need to be refreshed as a result of opening another form, adding a new project or code, and then returning to the time card form. A call to the Requery() methods of these two controls does the trick. **Listing 5-9** shows the code in the cboProjects class:

Listing 5-9*. The Requery() method of the object cboProjects.*

```
LOCAL lcOldAlias ;
   lnTally ;
   lnRowCount
```

```
lcOldAlias = ALIAS()
* Make sure the database is open
* and currently active, otherwise
* the view can't be found
IF !DBUSED("time and billing")
   OPEN DATABASE "time and billing"
ENDIF
SET DATABASE TO "time and billing"
IF ! USED("v_ProjectListRO")
   USE v_ProjectListRO NODATA IN 0
ENDIF
* Make sure we have the latest version
REQUERY("v_ProjectListRO")
lnTally = _TALLY
IF lnTally = 0
   DIMENSION THIS.alist[1,2]
   STORE "" TO THIS.alist
ELSE
   DIMENSION THIS.alist[lnTally,2]
ENDIF
SELECT v_ProjectListRO
lnRowCount = 1
SCAN
   THIS.alist[lnRowCount,1] = v_ProjectListRO.cProjectName
   THIS.alist[lnRowCount,2] = v_ProjectListRO.iProjectID
   lnRowCount = lnRowCount + 1
ENDSCAN

IF !EMPTY(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF
```

Note: If you're used to saving and restoring environmental settings in your code, you may wonder about the SET DATABASE command. You need to be concerned about the currently selected database when a) programmatically opening a view, b) calling a function/procedure in the stored procedures, or c) retrieving a property using DBGETPROP(). If you don't SET DATABASE TO prior to issuing any of those commands, you'll get hosed eventually, even if you only have one database opened, because it isn't necessarily currently selected. Thus, the proper defensive programming practice is not to save and restore the currently selected database, but to make sure that you have the database (or the correct database, if you are dealing with more than one) currently selected before issuing a command that works on the currently selected database. Not having an open database selected as the current database poses no problem in other situations.

Similarly, with private data sessions, bound form controls, and the addition of alias clauses to most commands and functions that act on a cursor, the need to save and restore the current work area is greatly reduced, even though many of us—out of habit—continue to do so. I have adopted the practice of never using a command or function without explicitly specifying an alias for the appropriate work area when the command or function supports an alias or work area argument. As a result, as

with SET DATABASE, I program defensively by never assuming the state of the environment, and always select the proper work area for those few commands (like APPEND FROM or LOCATE or SCAN…ENDSCAN) that do not accept an argument to specify an alias or work area.

As you can see from the code in Listing 5-9, the cboProjects control has two columns (but only displays the first), with the names of the projects in the first column, and the second with the surrogate primary key value for each project.

CHAP5.VCX contains two other combo boxes that are both subclassed from cboSystemCodes. cboSystemCodes uses v_SystemCodesRO, which is parameterized as shown in Listing 5-8. The SystemCodes table is an "overloaded" table. If you browse it, you'll see that it contains two different kinds of codes: work codes and expense codes. cboSystemCodes calls its Requery() method just as does cboProjects. However, its Requery() method has to establish a view parameter to retrieve only one type of code from the SystemCodes table. This value is stored in the cType property of cboSystemCodes. The Requery() method for cboSystemCodes is shown in **Listing 5-10**.

Listing 5-10*. The Requery() method for cboSystemCodes.*

```
LOCAL lcOldAlias ;
   lnTally ;
   lnRowCount

lcOldAlias = ALIAS()

IF !DBUSED("time and billing")
   OPEN DATABASE "time and billing"
ENDIF
SET DATABASE TO "time and billing"
IF !USED("v_SystemCodesRO")
   USE v_SystemCodesRO NODATA IN 0
ENDIF
* Here's where we establish the view
* parameter
vp_cType = THIS.cType
REQUERY("v_SystemCodesRO")
lnTally = _TALLY
IF lnTally = 0
   DIMENSION THIS.alist[1,2]
   STORE "" TO THIS.alist
ELSE
   DIMENSION THIS.alist[lnTally,2]
ENDIF
SELECT v_SystemCodesRO
lnRowCount = 1
SCAN
   THIS.alist[lnRowCount,1] = v_SystemCodesRO.cDescription
   THIS.alist[lnRowCount,2] = v_SystemCodesRO.iCode_ID
   lnRowCount = lnRowCount + 1
ENDSCAN

IF !EMPTY(lcOldAlias)
   SELECT (lcOldAlias)
ENDIF
```

The two classes subclassed from cboSystemCodes are cboWorkCodes and cboExpenseCodes, and each has its cType property set accordingly.

Note: Our ever-vigilant technical editor was wondering about my use of SCAN…ENDSCAN to transfer information from cursors into the arrays used by the combo box picklists. One of the powerful aspects of views is that they are data-source independent. All views are stored as a local cursor. Thus, a view is a view as far as VFP is concerned. It doesn't matter whether the view is drawing from a VFP database, a FoxPro 2.x database, an Access .MDB file or a SQL Server database. As long as the view definition is what the code expects it to be, it doesn't care (or know, usually) where the data actually comes from. However, unlike the situation where we know that the data will only come from a Visual FoxPro table, and can use a SELECT…INTO ARRAY, views cannot be directed into arrays. Even if we use ODBC and SQL pass-through, the results of a query executed with SQLEXEC() are placed into a cursor. By using a view and SCAN…ENDSCAN to transfer view contents to the array, we can freely change the source of the data without having to change the code.Depending on the situation, COPY TO ARRAY can be used, but SCAN…ENDSCAN provides a little more flexibility to concatenate or otherwise modify the cursor contents. For a lot more information on these techniques, refer to Chapter 11

The grid

The grid in the time card form contains an instance of cboProjects and an instance of cboWorkCodes, so the description of the codes is displayed, rather than the code itself. To allow these codes to be displayed all the time, the Sparse property of the columns is set to .F. For cosmetic purposes, the SpecialEffect property of the lists is set to "1–Plain" and the BorderStyle is set to "0–None".

The old swaperoo

When the form is first run, two drop-down-list-style picklists are visible at the top of the form—one for employees and the other for time cards by date. When the Add() button is clicked, the employee drop-down list is made invisible, and another is made visible, in the same position and identical to the first. The difference is that the "add mode" drop-down list is bound to the iEmployeeID field of the v_Time_Cards view, and the "edit mode" drop-down list is not. The cboEmployeeLookup control is used only for specifying an employee whose time cards you want to edit. The cboEmployee control is used to modify the employee ID of a new record.

Also, when the form goes into "add mode," the drop-down list that shows time cards for specific dates for the selected employee is made invisible, and a text box bound to the v_Time_Cards.tDateEntered field is made visible. The process of toggling these four controls is performed by the Add() method, the Cancel() method and the AfterSuccessfulSave() method.

Cut to the chase

Now that we've gotten the mundane stuff out of the way, let's concentrate on the real work of the form.

In order to work with views, a form performs two basic functions that distinguish them from forms working directly against Visual FoxPro tables. It stores user-determined key values to view parameters and calls the REQUERY() function on each updateable view. Beyond that, there isn't much that is really different from how a form would handle the tables directly.

In practice, the only features of a view-form that are different from a table-form are how it accesses an existing record, and that it follows a slightly different procedure before adding new records.

Selecting a time card—accessing an existing record

To select a time card for viewing or editing, the user first selects an employee. The InteractiveChange of the cboEmployeeLookup control stores its value to the form's iEmployeeID property. An iEmployeeID_Assign method is thus triggered. This method, shown in part in **Listing 5-11**, establishes the view parameter and executes the REQUERY() on the v_Time_CardsByEmpID_RO view and calls cboTimeCards.Requery() to update it with the new list of time cards for the selected employee.

Listing 5-11*. Part of the iEmployeeID_Assign method.*

```
vp_iEmployeeID = ThisForm.iEmployeeID
REQUERY("v_time_CardsByEmpID_RO")
ThisForm.cboTimeCards.Requery()
```

Control returns to the InteractiveChange event method of cboEmployeeLookup, which then sets the value of cboTimeCards to 0, to reflect that none of the time cards for this employee are selected. At this point, the list property of cboTimeCards has been populated with the records from v_Time_CardsByEmpID_RO, which is a list of all the time cards, ordered by date, that are on file for the selected employee.

The act of storing 0 to the value property of cboTimeCards causes its ProgrammaticChange event to fire, which calls the method code for the InteractiveChange event. This is primarily for the purpose of "clearing" the display when the user is already viewing a time card and wants to view a different time card. We'll see how this is accomplished in a moment.

The next thing the user does, after selecting an employee, is select which time card he wishes to view by making a selection from the cboTimeCards drop-down list. The InteractiveChange() event method of cboTimeCards stores the drop-down's value (the iTimeCardID of the selected time card) to the uKeyValue property of the form. There is an assign method associated with the uKeyValue property. This method calls the form's Requery() method. The code for the uKeyValue_Assign and the Requery() method are shown in **Listing 5-12**.

Listing 5-12*. The uKeyValue_Assign() and Requery() methods.*

```
* uKeyValue_Assign
LPARAMETERS vnewval
DoDefault(vnewval) && Stores the iTimeCardID value to the form's uKeyValue
property
ThisForm.Requery()

* Requery() Method
vp_iTimeCardID = ThisForm.uKeyValue
REQUERY("v_Time_Card_Hours")
```

```
REQUERY("v_Time_Cards")
ThisForm.Refresh()
```

As you can see from Listing 5-12, four lines of code take the form's uKeyValue property (which is a primary key value for the Time_Cards table, and a foreign key value for the Time_Card_Hours table) and retrieve the corresponding records from each table by calling the REQUERY() function for each view, then refreshing the form.

Adding a new time card

Adding a new time card requires two steps. The first is "blanking," "clearing" or "purging" the cursors. This means requerying the updateable views in such a way that they contain no records—they're ready to have a new, empty record appended. This is what differentiates adding records in a form that uses views from one that works with tables. The second step is the same step necessary when working with tables—appending blank records to the cursors, ready to accept the data that the user wants to enter.

Clearing cursors is important, because when working with views it is imperative that you never have records "in hand," other than those you intend to work with. Simply doing an APPEND BLANK on v_Time_Cards when viewing another record would give you two records—one for the new one you are adding, and another for the one you were just viewing. Because of this, when adding a new record (or deleting an existing record), it is necessary to do something that will result in a RECCOUNT() of 0 for all updateable views.

Logically, simply executing a query using a view parameter value that has no corresponding records in the table will do the trick. If you're using integer surrogate keys, and are retrieving records by the key values (and have a rule that says the primary keys can't be 0) then you're all set. Storing a 0 to the view parameters and calling REQUERY() will have the desired result—views with empty result sets.

If, on the other hand, you're using character surrogate keys, things become a little trickier. If you're doing the filter comparison using "=" and EXACT is set OFF, or if you use the LIKE operator, then you have to ensure that the character string that you specify does not, and will never, exist in the table, nor will it ever appear as a subset of the key values in the table. If you are using a base 62 scheme (which uses all uppercase and lowercase letters as well as the digits 0–9), you can never guarantee that you won't someday match an existing record. If you run with EXACT ON, or use the "==" operator, you can clear the cursor by using the empty string for the view parameter.

If the view uses a non-surrogate character field for the view parameter comparison, it may be easy to select an expression that will never show up in the table. For instance, if the filter comparison is being performed on the cInvoiceNo field, and the invoice numbers are six characters consisting only of the digits 0–9, then you can use a parameter value of "XXXXXX" (or any other string of alpha characters) to clear the cursors.

In the case of the Time Card form, the key values are indeed integers, and there is a rule on the primary key field prohibiting 0 values. Storing a value of 0 to the view parameters for both v_Time_Cards and v_Time_Card_Hours and requerying both views will yield an empty result set for both views.

Recognizing the need for this functionality, the cDMForm contains an empty ClearCursors() method. In the Time Card form, this method has only a single line of code:

```
ThisForm.cboEmployeeLookup.Value = 0
```

Executing this line of code will initiate the following cascade of events, courtesy of control events and the new assign methods. Note that nothing in this cascade of events is anything different than what has already been coded to allow the user to display an existing time card. The user cannot (by selecting an employee from the list) select an iEmployeeID value of 0, so the ClearCursors() method takes care of this:

- Changing the value property of cboEmployeeLookup to 0 triggers CboEmployee-Lookup.ProgrammaticChange().
- CboEmployeeLookup.ProgrammaticChange() calls CboEmployee-Lookup.InteractiveChange().
- CboEmployeeLookup.InteractiveChange() stores 0 to the form's iEmployeeID property.
- Storing 0 to the form's iEmployeeID property triggers the iEmployeeID_Assign method.
- The iEmployeeID_Assign method stores 0 to the view parameter and calls REQUERY() for v_Time_CardsByEmpID_RO, which clears that view.
- CboEmployeeLookup.InteractiveChange() also stores 0 to cboTimeCards.Value, which triggers cboTimeCards.ProgrammaticChange().
- cboTimeCards.ProgrammaticChange() calls cboTimeCards.InteractiveChange().
- cboTimeCards.InteractiveChange() stores 0 to the form's uKeyValue property.
- Storing 0 to the form's uKeyValue property triggers the uKeyValue_Assign() method.
- uKeyValue_Assign() calls the form's Requery() method, which calls REQUERY() for v_Time_Cards and v_Time_Card_Hours, which clears both views.

Before leaving the example form, I'd like to call your attention to the BeforeSave() method, which does something that might look a bit unusual in the case of a deleted time-card record. This database has a delete trigger on the Time_Cards table, which causes all related child records in Time_Card_Hours to be deleted if the Time_Cards record is deleted. If the user has modified any records in v_Time_Card_Hours before deleting the time card, the update of the v_Time_Cards view (which, you'll recall, occurs first because the cursor name for this view ends in "1") will delete the related Time_Card_Hours records. Then, the update of the v_Time_Card_Hours view takes place, and it will detect that the underlying tables have been changed (they've been deleted!) and the update will fail because of an update conflict.

To avoid this, the BeforeSave() method checks to see if the parent record in v_Time_Cards has been deleted, and if so, it reverts any changes made to the child view.

The View/Query Designer

Within a typical application, views will fall into one of two groups: one group of views designed as updateable views that are used to modify the data, and another group designed as *non*-updateable views, to support reports, on-screen inquires, picklists and validation queries. In general, updateable views are almost always parameterized and usually incorporate fields from only a single table.

Although you can feel extremely clever constructing a single view that joins three tables and performs updates on all three, I've found that the KISS (Keep It Simple, Stupid!) principle applies as well to the creation of updateable views as it does to almost any other aspect of ap-

plication development. When creating updateable views, adhere to the following three rules of thumb:

1. The SQL property of the view should look like this: SELECT * FROM <table_name> WHERE <field_name> = <view_parameter> — note only one table, no joins.
2. A single field should be flagged as the key field. If you're using integer surrogate keys, you're home free.
3. All fields should be flagged as updateable.

Save your cleverness for adding some useful "virtual" fields to your updateable views or creating rules that facilitate data entry … and save a *lot* of cleverness for those thorny reporting requirements that the client has in the specifications!

From the previous couple of paragraphs, you can see that the View Designer does a good job meeting all your needs for creating updateable views. But the View Designer really falls down in some of the complex queries required for some lists and reports. Let's take a moment to understand why the View Designer is so limited in this area.

There are two types of join syntax that are permitted under the ANSI '92 SQL standard. Many have come to describe the two types of syntax as the "nested" and the "sequential" syntax. The following listings show only the FROM clause of a SELECT command to illustrate the differences.

***Listing 5-13**. An example of "nested" join syntax.*

```
1.  FROM <table_1> ;
2.     JOIN <table_2> ;
3.        JOIN <table_3> ;
4.           JOIN <table_4> ;
5.           ON <table_4_expression> = <table_3_expression> ;
6.        ON <table_3_expression> = <table_2_expression> ;
7.     ON <table_2_expression> = <table_1_expression> ;
```

In Listing 5-13, line 6 shows an expression joining table_3 with table_2, but it could just as well join table_4 with table_3, or any other table within the "nest."

***Listing 5-14**. An example of "sequential" join syntax.*

```
1.  FROM <table_1> JOIN <table_2> ;
2.        ON <table_1_expression> = <table_2_expression> ;
3.     JOIN <table_3> ;
4.        ON <table_3_expression> = <table_2_expression> ;
5.     JOIN <table_4> ;
6.        ON <table_4_expression> = <table_3_expression>
```

In Listing 5-14, line 6 could join table_4 with table_3, table_2 or table_1; this fact seems readily apparent and quite intuitive. In general, I strongly believe that anyone who tries to create a complex query using the "nested" syntax, as shown in Listing 5-13, is a confirmed maso-

chist, and should be kept away form sharp objects. The reason becomes apparent when trying to create a join as illustrated in **Figure 1**.

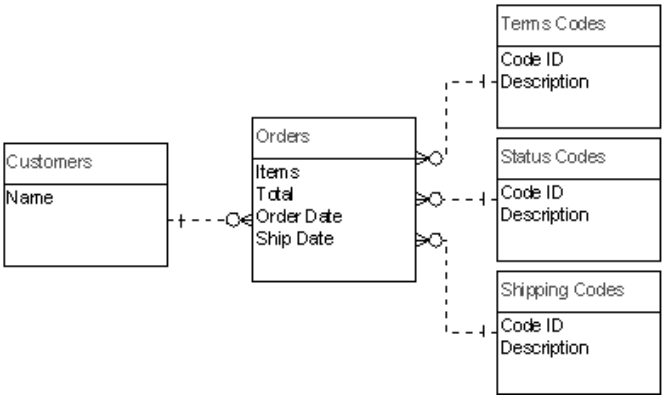

***Figure 1**. A join scenario suspected to be impossible using the "nested" join syntax.*

Figure 1 illustrates a view that has, as a requirement, the customer name, order items, total amount, order date and ship date, as well as the description associated with the terms code, status code, and the shipping code on each order. Using the "sequential" syntax, it's a walk in the park. Using the "nested" syntax … well, call me when you have it done—meanwhile, I'm going on a two-month vacation.

Even if we simplify the query illustrated in Figure 1 by eliminating the need for descriptions from two of the three code tables, the nested syntax is still somewhat cumbersome, requiring us to be very careful with the *order* in which the joins are made to get the desired results. Matters get even worse when you introduce outer joins into the equation. To be fair, there might be some unique capabilities inherent in the nested syntax, but I have yet to discover them.

So what does this have to do with the Visual FoxPro View Designer? The view designer stores the join conditions using the nested syntax. As a result, it becomes either very difficult to use, or downright useless when creating a complex view.

All is not lost, however. Most experienced developers include a file that ends up being called something like VIEWSCRIPT.PRG (see **Listing 5-15**) in which these complex views can be maintained in code. If you're not already comfortable with writing SQL code, I strongly encourage you to develop that skill as soon as possible. Many skills we acquire eventually become obsolete. However, I think the ability to easily write SQL query syntax is one that really has some "legs," and will be useful for many years to come.

The big drawback in maintaining a view in code is that we have no easy way to set the properties for the various view fields, such as which field is the key field, what fields are updateable, and what UpdateType and WhereType are to be used. Establishing these rules in code requires a long sequence of DBSETPROP() commands. However, as mentioned above, most (if not all) of the updateable views you will be using involve no joins, and as a result can be handled very nicely using the View Designer. The views that you will most likely be creating and

maintaining in code are used for reports and for creating on-screen queries and form picklists. This type of view is not an updateable view, but a "read–only" view, as shown in the Chapter 5 example form, and exists only to conveniently store a scheme for extracting and presenting some information from the database. A view is by default *not* updateable. So creating a complex view in code doesn't have to do anything other than specify the name under which the view is stored, and the SELECT statement with the view parameters (if any). Using the example illustrated in Figure 1, you might see a developer use the code in Listing 5-15 to support this particular report.

***Listing 5-15**. An example of a "read-only" view that would be maintained in code in a VIEWSCRIPT.PRG file.*

```
PROCEDURE Create_V_Order_Status_RO
   CREATE SQL VIEW v_Order_Status_RO AS ;
      SELECT customers.cName, ;
            orders.iItems, ;
            orders.dordered, ;
            orders.dshipped, ;
            orders.ytotal, ;
            termscodes.cDescription AS terms, ;
            statusCodes.cDescription AS status, ;
            shippingCodes.cDescription AS via, ;
         FROM customers JOIN orders ;
               ON customers.cCust_ID = orders.cCust_ID ;
            JOIN systemcodes termsCodes;
               ON orders.iTerms_Ref = termsCodes.iCode_ID ;
            JOIN systemcodes statusCodes ;
               ON orders.iStatus_Ref = statusCodes.iCode_ID ;
            JOIN systemcodes shippingCodes ;
               On orders.iShip_Ref = shippingCodes.iCode_ID ;
         WHERE orders.cCust_ID = ?vp_cCust_ID
ENDPROC
```

Once this procedure is written, it can be executed as shown in **Listing 5-16**.

***Listing 5-16**. Executing the view-maintenance routine in Listing 5-15.*

```
IF NOT DBUSED("<target_database>")
   OPEN DATABASE <target_database>
ENDIF
SET DATABASE TO <target_database>
SET PROCEDURE TO viewscript ADDITIVE
DO create_v_order_status_RO
```

However, my preferred method is to open the target database and make it the currently selected database with SET DATABASE (this is important!). Then, open the VIEWSCRIPT.PRG in the Visual FoxPro editor, highlight everything *within* the procedure, right-click with the mouse and choose "Execute selection" from the shortcut menu. This is particularly convenient when it's necessary to modify the view. You modify, select and run the CREATE SQL VIEW command, then close and save the VIEWSCRIPT.PRG.

The case for updateable views

Let me clearly state that I in no way want to try to convince anyone that views are the only way, or even the "right" way, to interact with data. There is no principle, or best practice, or even a rule of thumb that will dictate when/if you should use updateable views. There is no rule (or even a school of thought) that says you're a dummy if you don't use views, and there is certainly no rule that says you can't use a mixture of updateable views and direct table manipulation. However, in my life as a developer, certain things have come down the pike that have resonated with my soul in some way—things that make me say to myself, "This is Right; This is Good." Procedural programming, naming conventions, black-box routines, reusable code and object-oriented programming were all concepts that "clicked" when I began to understand how they worked. Each new approach or idea brought me a big step closer to some kind of programmer's Nirvana, where the DoWhatIMean() function is a reality! Once I started using updateable views, I experienced the same feeling of "Wow" that I did with some of the earlier concepts. I hope that more folks give views a spin, and find themselves closer to Programming Perfection as I did.

Why would you exclusively use views?

First, I've found that working with views is a very easy way to interact with data. The whole process has a very "clean" feel to it. This is not a factor to be discounted—how much of what we do in our development practices do we do simply because "it feels right"?

There are no indexes, relations or filters that have to be maintained at runtime to ensure that we're only modifying the data that we intend to modify. With the exception of occasionally restoring the record pointer to its original position after moving it when doing some kind of processing, and checking for an EOF() condition when appropriate, the record pointer can be ignored. Views can be indexed if it's necessary to allow the user to control the order in which records are displayed, and the index order can be changed without concern for "breaking" an established relation.

Then there are field-level rules. Work with tables and you really don't want to enforce field-level rules; violating a rule means the user absolutely *cannot* leave a field or control until the rule is satisfied. Likewise, a row-level rule must be satisfied before moving the record pointer to another record (See Chapter 6 for more information on rules). With views, field-level and row-level rules can be used freely because they aren't evaluated until the TableUpdate() is executed.

While on the subject of rules, we can establish two completely different sets of rules: one at the table level to enforce data integrity, and one at the view level to facilitate data entry. I've spent days trying to accomplish the most god-awful things using the myriad of grid events and other form controls, that were suddenly a piece of cake when I called a stored procedure from a view field- or row-level rule.(Refer to Chapters 11 and 12 for some important views on using Grids).

Server-based SQL databases are increasingly popular, and being able to interact with them effectively is a valuable skill. Acquiring this skill requires that we begin to think entirely within the constraints of SQL commands. You can't perform a SEEK on a SQL Server table to validate a user's input. However, you can create a lookup table on the fly using a view, index that view, and use that for validation. You can also use a view and dispense with Xbase syntax almost entirely, as shown in **Listing 5-17**.

Listing 5-17*. A sample data-validation function using a view.*

```
FUNCTION ValidatePartNumber(tcPartNumber)
   vp_cPartNumber = tcPartNumber
   IF ! USED("v_ValidatePart")
      USE v_ValidatePart IN 0
   ELSE
      REQUERY("v_ValidatePart")
   ENDIF
   IF _TALLY = 0
      llReturnValue = .F.
   ELSE
      llReturnValue = .T.
   ENDIF
   RETURN llReturnValue
ENDFUNC
```

Note that the foregoing function works equally well with remote views as it does with local views.

Designing Visual FoxPro applications using views exclusively helps you to begin thinking more in terms of pure SQL, which not only influences how you implement the application, but with how you design the database!

Jim : Steve and I agree about a lot of things, but exclusive use of views isn't one of them. Views, both local and remote, are very powerful tools and they should not be dismissed from our collection of weapons. However, exclusively doing anything in application development leaves the possibility of missing a mechanism of solving a problem simply because it doesn't fit our exclusive vision of things.

The design of the data access mechanisms should be part of the analysis and design of the overall application, and it should consider all of the issues involved. Using views for data access exclusively is one of many options available to the developer, and it should be considered as such: one of the possibilities.

Data-aware form classes should be capable of handling views and tables in the data for the form. Not doing this simply creates a form class that fails under certain conditions.

There are times when the extra overhead of views is not necessary—for example, in a small departmental system where using tables directly is fine. Also, using views can seriously complicate development of real-time systems. The level of indirection that is the very power of a view causes an inherent problem in real-time applications.

So, I say, keep things in perspective. Views are powerful and they offer certain advantages. However, nothing is *just* advantages; all things also carry disadvantages. To blindly go forward without acknowledgement of the disadvantages of a certain path is giving those disadvantages absolute power over your work.

Now that I've said all of this, I have to agree with Steve that, in most cases, using views exclusively for data management is a very good approach to development.

Steve replies:

I agree with Jim's points in general. However, my experience is that the "extra overhead" that Jim speaks of isn't really worth worrying about. In fact, making a design decision to use views lends a simplicity and clarity to the development process that is really quite surprising once you

get involved with it. Believe me when I say that I'm probably one of the world's laziest developers, and if something is a lot of extra work or headache, I'm quick to dump it. I agree that few things in life consist *only* of advantages, but updateable views are one of the things that seem a little thin on the downside.

Performance considerations

A concern I frequently hear from developers when I talk about using views is that using views will slow down the application. Consider that Visual FoxPro is (and long has been) the fastest database management tool in the known universe. Views are simply one way of retrieving data, and retrieving data is what FoxPro does best, right?

Just to provide a couple of examples, I have a view in a production application that uses a complex series of outer joins to extract 1163 records from four tables containing 51, 117, 1223 and 1229 records. This view executes (starting with all tables closed) in 0.45 seconds on a 200 MHz laptop with a run-of-the-mill IDE drive. I also have a view that extracts 5100 records from a single table of 126,000 records in 0.43 seconds on the same machine. Remember that virtually every updateable view you are likely to use will be indexed on the primary or foreign key, and that this is the field that is used in the WHERE clause in the query. Rushmore kicks in every time, and you get the records fast-fast-fast!

If you take the time to give views a try, you'll find just how easy they are to use, and how in many ways they'll make your life much easier. They do require some up-front time and effort to get comfortable with the ideas and techniques, and perhaps a little more design work at the database level, but it's effort that pays off handsomely in the long run.