CHAPTER 3

Building a Visual FoxPro Application for SQL Server

Even if you haven’t started using three-tier data access in FoxPro, you certainly have heard of it. What’s the big deal? Is this something that you need to learn? In this chapter, you’ll discover that

• You absolutely need to learn three-tier data access.
• It’s easy.
• The library described in this chapter can be used with your very next project.

In this chapter, we’ll build a data access layer to communicate with either DBFs or SQL Server. And we’ll build it in such a way that there is absolutely no code to change when you move from DBFs to SQL tables. We’ll even include an upsizing wizard to migrate the data for you. We’ll talk about the things that you don’t want to do in SQL if you want to simplify programming (always a good thing). We’ll use a data access layer, which gives you the ability to use DBFs, SQL Server, a WebConnection XML server, or XML Web services built in Visual FoxPro 8, the best upgrade yet. The code for this chapter is written to be compatible with Visual FoxPro 7, but in subsequent chapters we’ll add features only available in versions 8 and higher. It might surprise Microsoft, but not everyone has the latest version of their languages.

Why Three-Tier?
Three-tier is a variant of n-tier: A calls B calls C, and so on. Each one does a part of the task. With server farms and ASP

IN THIS CHAPTER

• Why Three-Tier?
• Creating the SQL Database and Loading Your Tables
• Writing the Sample Application
• The Form Template
• A Search Form Template
• The Data Tier
• What’s Next?
applications, there can be several data tiers, a page generation tier, and so forth. But for our purposes, three-tier is generally sufficient. In the usual three-tier diagrams (which we’ll dispense with here), A is your form, B is a data access layer, and C is the place where the data is stored—usually DBFs in our world, but that’s changing, and that’s where the data access layer comes in.

In traditional FoxPro applications, our forms contain the code that gets and stores data. Our code snippets are full of SEEK and REPLACE commands. The problem arises when our client decides that they’re tired of kicking everyone out of the application and rebuilding the indexes, or redoing a 400-file backup that just failed because a user didn’t close the CONTROL file, or watching an APPEND BLANK take 30 seconds because the table has 900,000 records and the index is 9MB. DBFs are great, but they do have drawbacks. And the solution is spelled S-Q-L.

SQL has numerous benefits. When you back up a SQL database, you’re backing up a single file. Backup can be run while users are in the system. And RESTORE is also a one-line command.

Security is another issue with FoxPro tables. Anyone with access to the DBF directory on the server can see your FoxPro tables. SQL Server, on the other hand, has complex security built in. So you can decide who does what. In today’s increasingly risky environment, users can and will demand improved security. SQL Server is a good way to accomplish it.

So your client is sold. Install SQL Server. You can run SQL Server on your development machine just fine; in fact, it’s a great idea. Be sure to install the Developer Edition, which has a Management Console. If all you have is MSDE, it will work fine, but you have to create the database, indexes, and logins programmatically, and it’s just a little harder to learn some SQL tasks nonvisually.

SQL Server runs as a service. It “listens” for requests from workstations, does what is asked of it, and sends any result set back to the workstation. There is no index traffic because the indexes don’t come back with the results. Most of the slowdown you might have experienced in FoxPro apps on a LAN are due to network traffic, so solving the slowdown problem can be sufficient motivation for migrating to SQL Server.

So you’ve installed SQL Server, and you need to migrate your application to use SQL Server tables. First, you have to migrate the data. There are several ways to do this, and all of them have problems. You can use the SQL Upsizing Wizard, but the resulting SQL tables can cause serious programming headaches. There’s a DTS utility that is installed when you load SQL Server, and if you like writing your data recoding routines in Basic, go right ahead. I prefer FoxPro. So your best bet is to write your own data migration program. I’ve included one in the code for this chapter.

What’s wrong with the Upsizing Wizard? For one thing, it defaults to permitting NULLs as values in uninitialized fields. If you’ve never run into NULLs, consider yourself lucky. Statisticians need to know whether a value of zero is a reported value or simply someone
who didn’t answer the question—for example, What is your age? You can’t calculate average age by summing ages and dividing by zero if half of the respondents didn’t want to answer. So you have to know which are missing values. SQL Server allows for missing values, and in fact defaults to them. But they nearly double the programming burden. ASP code goes bonkers with nulls. So unless you really, truly care about missing values, you absolutely don’t want to use NULLs. That’s why the preferred way to declare a column in T-SQL is

Age Integer NOT NULL DEFAULT 0,...

I strongly urge you to include NOT NULL in your column declarations and to supply a default value.

Secondly, SQL has reserved words, which have to be enclosed in square brackets if you use them as field names. I don’t use them if I have my way. But we often have to support legacy applications, and that means using two systems in parallel for at least a while. So we’ll want to enclose reserved word column names in square brackets while building the table definitions. The data conversion program provides for you to furnish a list of SQL keywords that you’ve used as field names. I’ve seeded the list with the usual suspects (for example, see line 21 of the LoadSQLTables.PRG file in Listing 3.2). Add other field names that you’ve used in your tables if SQL complains about them during the table creation process.

Finally, in order to make updating records easy, it’s a good idea to provide a unique integer key as the primary key for each table, especially if you have another text field that you’ve been using as a key. The reason is that unique keys are essential if you want to make the coding of updates simple, so every table has to have one. In the FoxPro world we’ve developed the bad habit of using a compound key (for example, PONum+LineNum for the purchase order items file), which is simply a nightmare to code in some generic fashion.

SQL Server has an autoincrementing feature called IDENTITY. For example, you can declare an Integer field named MyKey and set IDENTITY(1,1) (begin with 1 and increment by 1), and every time you insert a record a new key value will appear. The problem is that if you need instant access to that value, you need to add a SELECT @@IDENTITY command after the INSERT command—for example, after adding an Invoice header and before inserting the related Invoice Detail Lines in order to provide the header key in the detail records for subsequent JOINs. And there are other reasons. If you have an IDENTITY field, you must not include its name in any INSERT commands. So your CommandBuilder code has to know to skip the key field if it’s an IDENTITY field. You get the picture. Identity fields are more trouble than they’re worth. So we’ll do our own primary key generation instead of using SQL’s IDENTITY feature.

To get a head start, open each of your application’s tables and create a primary key field if it doesn’t already have one (child tables are good candidates). Use the MODIFY STRUCTURE command, add the PKFIELD column name (you can use PKFIELD as the PRIMARY KEY column for every table if you want to), and then use this to add unique keys:
REPLACE ALL "pkfield" WITH RECN0()

or

REPLACE ALL "pkfield" WITH TRANSFORM(RECN0(),"&L #######")

for character fields.

When you're done, do a SELECT MAX(KeyFieldName) FROM (TableName) for each of your DBFs, and make sure that these are the values that appear in the table called Keys that you'll find in the zip file for this project. This table tells the application what the last primary key value used was in each of the tables in the application. This table exists both for DBF-based and for SQL-based systems. It's a loose end, and a commercial application that used this technique would need to include a goof-proof way to set these keys after migrating the data and before going live. You'll have to do it manually. Or you can write a little utility routine as an exercise. (Hint: You'll need a list of table names and their primary key fields in order to automate the process.)

Getting Our Test Data Ready

The FlatFileForm and DataTier classes will allow us to quickly create forms that work with both a DBF and a SQL table. But in order to test them, we'll need to have the same tables in both formats. Luckily, there's an easy way to accomplish this.

FoxPro ships with a sample data directory containing some good examples. Type the following in the command window:

USE C:\PROGRAM FILES\MICROSOFT VISUAL FOXPRO 8\SAMPLES\DATA\Customer
COPY TO CUSTOMER

USE C:\PROGRAM FILES\MICROSOFT VISUAL FOXPRO 8\SAMPLES\DATA\EMPLOYEE
COPY TO EMPLOYEE

When you have your two tables, you should add integer keys to the tables using MODIFY STRUCTURE. In this case, use PKFIELD (Integer) for both tables, and make it the first field in the table. Before leaving the schema designer, add PKFIELD as an index tag, or just type the following in the command window:

USE CUSTOMER EXCLUSIVE
INDEX ON PKFIELD TAG PKFIELD
USE CUSTOMER EXCLUSIVE
INDEX ON PKFIELD TAG PKFIELD

Finally, you can set the database name in the SQL ConnectionString and run the LoadSqlTables program to load your tables.

If you want to use your own tables from your own database, you can copy them to DBFs very easily, using the procedure shown in Listing 3.1; substitute your names for your database, userID, and password.
LISTING 3.1 Procedure to Copy a SQL Table to a DBF

PROCEDURE SQLToDBF
PARAMETERS TableName
ConnStr = [Driver={SQL Server};Server=(local);Database=(Name);UID=X;PWD=Y;]
Handle = SQLStringConnect( ConnStr )
SQLExec ( Handle, [SELECT * FROM ] + TableName )
COPY TO ( TableName )
MessageBox ( [Done], 64, [Table ] + TableName + [ copied from SQL to DBF], 1000 )
ENDPROC

Sample usage:
SQLToDBF ( [Customers] )

Creating the SQL Database and Loading Your Tables

We’re ready to build the SQL database and load the data. The easy way is to open the Enterprise Manager and create the MDF and LDF files yourself, estimating the initial size of each. Or you can just use a CREATE DATABASE command and let SQL start you out with about half a megabyte each. You can specify where to put these files, but the default, which is probably Program Files\Microsoft SQL Server\MSSQL\Data, is usually best. You can also use three lines of code in the command window to create the database:

Handle = SQLStringConnect ( ;
"driver={SQL Server};server=(local);database=Master;pwd=sa;uid=;"
Result = SQLExec ( Handle, 'CREATE DATABASE MYDATABASE' )
SQLDisconnect(0)

I also strongly urge you to create a userID and password that takes you straight to your database. Again, you can do this in Enterprise Manager or in code, but doing so visually is easier. Use the Security tab to add a login, giving (for now) full rights to the userID. Let the DBAs worry about fine-tuning security. They want to keep people out; we want to let them in.

As you saw earlier, whenever you want to send a command to SQL Server, you’ll need a handle, which is always a positive integer, for example, 1, 2, 3, and so on.

The following connection string gets you a handle:

Handle = SQLStringConnect ( ;
"driver={SQL Server};server=(local);database=MyDatabase;pwd=sa;uid=;"

To close any and all open handles, use SQLDisconnect(0).

Now we’re ready. The program shown in Listing 3.2 will ask you where your DBFs are and load them to the named database on SQL Server.
* Purpose....: Creates a duplicate of each DBF
*            : from your data directory in SQL Server
*              and copies the DBF's records to the SQL table.
*              The program puts brackets around named reserved words.
* If you get an error indicating illegal use of a reserved word, add it here:

SET TALK OFF
CLEAR
CLOSE ALL
SET STRICTDATE TO 0
SET SAFETY OFF
SET EXCLUSIVE ON
SET DATE AMERICAN
SET CONFIRM ON

ConnStr = [Driver={SQL Server};Server=(local);UID=sa;PWD=;Database=MyDatabase;]
Handle = SQLSTRINGCONNECT( ConnStr )
IF Handle < 1
    MESSAGEBOX('Unable to connect to SQL' + CHR(13) + ConnStr, 16 )
    RETURN
ENDIF

ReservedWords = ;
[,DESC,DATE,RESERVED,PRINT,ID,VIEW,BY,DEFAULT,CURRENT,KEY,ORDER,CHECK,FROM,TO,]

DataPath = GETDIR("Where are your DBFs?")
IF LASTKEY() = 27  && Escape was pressed
    RETURN
ENDIF
IF NOT EMPTY ( DataPath )
    SET PATH TO &DataPath
ENDIF
ADIR( laDBFS, ( DataPath + [*.*DBF] ) )
ASORT(laDBFS,1)
* Load each of the tables to SQL
FOR I = 1 TO ALEN(laDBFS,1)
    USE ( laDBFS(I,1))
    _VFP.Caption = "Loading " + ALIAS()
    LoadOneTable()
ENDFOR

SQLDISCONNECT(0)
PROCEDURE LoadOneTable
LOCAL I
cRecCount = TRANSFORM(RECCOUNT())
cmd  = [DROP TABLE ] + ALIAS()
SQLEXEC( Handle, Cmd )
* Skip tables we don’t want to load
IF ALIAS() $ [COREMETA/DBCXREG/SDTMeta/SDTUSER/FOXUSER/]
* skip system tables, add yours here.
  ? [Skipping ] + ALIAS()
  RETURN
ENDIF
CreateTable() && see below

SCAN
  WAIT WINDOW [Loading record ] + TRANSFORM(RECNO()) + [ ] + cRecCount NOWAIT
  Cmd  = [INSERT INTO ] + ALIAS() + [ VALUES ( ]
  FOR I = 1 TO FCOUNT()
    fld  = FIELD(I)
    IF TYPE(Fld) = [G]
      LOOP
    ENDIF
    dta  = &Fld
    typ  = VARTYPE(dta)
    cdta = ALLTRIM(TRANSFORM(dta))
    cdta = CHRTRAN ( cdta, CHR(39),CHR(146) ) && remove any single quotes
    DO CASE
      CASE Typ $ [CM]
        Cmd = Cmd + ['] + cDta + ['] + [, ]
      CASE Typ $ [IN]
        Cmd = Cmd + cDta + [, ]
      CASE Typ = [D]
        IF cDta = [/ ]
          cDta = [ ]
        ENDIF
        Cmd = Cmd + ['] + cDta + ['] + [, ]
      CASE Typ = [T]
        IF cDta = [/ ]
          cDta = [ ]
        ENDIF
      ELSE
**LISTING 3.2 Continued**

```
Cmd = Cmd + '[' + cDta + ']' + [ , ]
CASE Typ = [L]
    Cmd = Cmd + IIF('F'Scdta, [0], [1]) + [ , ]
CASE Typ $ [Y]
    Cmd = Cmd
ENDCASE
ENDFOR
Cmd = LEFT(Cmd, LEN(cmd)-2) + [ )]
lr = SQLEXEC( Handle, Cmd )
IF lr < 0
    ? [Error: ] + Cmd
    SUSPEND
ENDIF
ENDSCAN
WAIT CLEAR
PROCEDURE CreateTable
LOCAL J
Cmd = [CREATE TABLE ] + ALIAS() + [ ( ]
AFIELDS(laFlds)
FOR J = 1 TO ALEN(laFlds,1)
    IF laFlds(J,2) = [G]
        LOOP
    ENDIF
FldName = laFlds(J,1)
    IF [,] + FldName + [,] $ ReservedWords
        FldName = "[" + FldName + "]"
    ENDIF
Cmd = Cmd + FldName + [ ]
DO CASE
    CASE laFlds(J,2) = [C]
        Cmd = Cmd + [Char() + TRANSFORM(laFlds(J,3)) ;
                        + [ ] NOT NULL DEFAULT ' ', ]
    CASE laFlds(J,2) = [I]
        Cmd = Cmd + [Integer NOT NULL DEFAULT 0, ]
    CASE laFlds(J,2) = [M]
        Cmd = Cmd + [Text NOT NULL DEFAULT ' ', ]
    CASE laFlds(J,2) = [N]
        N = TRANSFORM(laFlds(J,3))
        D = TRANSFORM(laFlds(J,4))
        Cmd = Cmd + [Numeric() + N + [,] + D + [ ] NOT NULL DEFAULT 0, ]
    CASE laFlds(J,2) $ [TD]
```
LISTING 3.2  Continued

    Cmd = Cmd + [SmallDateTime NOT NULL DEFAULT '', ]
    CASE laFlds(J,2) = [L]
        Cmd = Cmd + [Bit NOT NULL DEFAULT 0, ]
    END_CASE
    ENDFOR
    Cmd = LEFT(Cmd,LEN(cmd)-2) + [ )]
    lr = SQLEXEC( Handle, Cmd )
    IF lr < 0
        _ClipText = Cmd
        ? [Couldn't create table ] + ALIAS()
        MESSAGEBOX( Cmd )
        SUSPEND
    ENDIF
    ? [Created ] + ALIAS()
ENDPROC

For each DBF that's not in the “Skip These Tables” list at the top of the LoadOneTable routine, the program issues a DROP TABLE "Name" command, followed by a CREATE TABLE command, which it builds. It then scans all records in the DBF and creates and executes an INSERT statement for each record. Users are often amazed at how fast this loads their data. I’m not. It’s FoxPro.

Writing the Sample Application

Now, you’re ready to write your sample application. Begin by creating a project called Chapter3:

MODIFY PROJECT Chapter3

Leave it open because if you close a FoxPro project before you’ve added anything to it, it asks you if you want to delete the empty project file, and that’s just one more thing to deal with.

My MAIN.PRG will create a few settings, put a title on the screen, instantiate my DataTier object, install the menu, and initiate the event loop (see Listing 3.3).

LISTING 3.3  MAIN.PRG

* Purpose.....: MAIN program for application

CLEAR ALL
CLOSE ALL
CLEAR
CLOSE ALL
LISTING 3.3  Continued

SET TALK OFF
SET CONFIRM ON
SET MULTILOCKS ON
SET CENTURY ON
SET EXCLUSIVE ON
SET SAFETY OFF
SET DELETED ON
SET STRICTDATE TO 0

WITH _Screen
  .AddObject ( [Title1], [Title], 0, 0 )
  .AddObject ( [Title2], [Title], 3, 3 )
  .Title2.ForeColor = RGB ( 255, 0, 0 )
ENDWITH

ON ERROR DO ErrTrap WITH LINENO(), PROGRAM(), MESSAGE(), MESSAGE(1)

DO MENU.MPR

oDataTier = NEWOBJECT ( [DataTier], [DataTier.PRG] )
oDataTier.AccessMethod = [DBF]  && Required to execute Assign method code

IF NOT EMPTY ( oDataTier.AccessMethod )
  READ EVENTS
ENDIF

ON ERROR

SET PROCEDURE TO
SET CLASSLIB TO

SET SYSMENU TO DEFAULT

WITH _Screen
  .RemoveObject ( [Title1] )
  .RemoveObject ( [Title2] )
ENDWITH

DEFINE CLASS Title AS Label
Visible   = .T.
BackStyle = 0
FontName  = [Times New Roman]
PROCEDURE Init
LPARAMETERS nTop, nLeft
THIS.Top = _Screen.Height - 100
IF PCount() > 0
   THIS.Top = THIS.Top - nTop
   THIS.Left = THIS.Left - nLeft
   THIS.ForeColor = RGB(255,0,0)
ENDPROC

ENDDEFINE

PROCEDURE ErrTrap
LPARAMETERS nLine, cProg, cMessage, cMessage1
OnError = ON("Error")
ON ERROR
IF NOT FILE( [ERRORS.DBF] )
   CREATE TABLE ERRORS (   
      Date     Date,         
      Time     Char(5),      
      LineNum  Integer,      
      ProgName Char(30),     
      Msg      Char(240),    
      CodeLine Char(240)     
   )
ENDIF
IF NOT USED ( [Errors] )
   USE ERRORS IN 0
ENDIF
SELECT Errors
INSERT INTO Errors VALUES (   
   DATE(), LEFT(TIME(),5), nLine, cProg, cMessage, cMessage1 )
USE IN Errors
cStr = [Error at line ] + TRANSFORM(nLine) + [ of ] + cProg + [ : ] + CHR(13) 
   + cMessage + CHR(13) + [Code that caused the error: ] + CHR(13) + cMessage1
IF MESSAGEBOX( cStr, 292, [Continue] ) <> 6
   SET SYSMENU TO DEFAULT
LISTING 3.3  Continued

    IF TYPE ( [_Screen.Title1] ) <> [U]
        _Screen.RemoveObject ( [Title2] )
        _Screen.RemoveObject ( [Title1] )
    ENDIF
CLOSE ALL
RELEASE ALL
CANCEL
ELSE
    ON ERROR &OnError
ENDIF

Don’t try to run it yet because that reference on line 25 to MENU.MPR refers to a menu that we haven’t yet created. The menu is where requests to show forms are located, as well as the all-important CLEAR EVENTS command that will end the event loop and allow the program to march onward to its eventual self-destruction.

Next I’ll add a menu. I want to be able to switch between DBF and SQL access at will, so that I can verify that my program works exactly the same way with both data stores. So my menu (in which I’ve already added calls to a couple of screens that we’ll build in this chapter) looks like Listing 3.4.

LISTING 3.4  Menu.MPR

    SET SYSMENU TO
    SET SYSMENU AUTOMATIC

    DEFINE PAD FilePad       OF _MSYSMENU PROMPT "File"
    DEFINE PAD TablePad      OF _MSYSMENU PROMPT "Tables"
    DEFINE PAD DataPad       OF _MSYSMENU PROMPT "Change data access"
    ON PAD FilePad           OF _MSYSMENU ACTIVATE POPUP file
    ON PAD TablePad          OF _MSYSMENU ACTIVATE POPUP tables
    ON SELECTION PAD DataPad OF _MSYSMENU DO ChangeAccess IN MENU.MPR

    DEFINE POPUP file MARGIN RELATIVE
    DEFINE BAR 1 OF file PROMPT "E\\<xit"
    ON SELECTION BAR 1 OF file CLEAR EVENTS

    DEFINE POPUP tables MARGIN RELATIVE SHADOW COLOR SCHEME 4
    DEFINE BAR 1 OF tables PROMPT "Customers" SKIP FOR WEXIST([frmCustomer])
    DEFINE BAR 2 OF tables PROMPT "Employees" SKIP FOR WEXIST([Employees] )
LISTING 3.4  Continued

ON SELECTION BAR 1 OF tables do form frmCustomer
ON SELECTION BAR 2 OF tables do form Employee

PROCEDURE ChangeAccess
  oDataTier.AccessMethod = :
    UPPER(INPUTBOX("Data access method", "DBF/SQL/XML/WC", "DBF" ))

What’s oDataTier? That’s next. The DataTier controls everything about access to data. So the first thing we do is add a property called AccessMethod, use its Assign method to trap and validate the assignment, and then do whatever needs to be done based on the method chosen. More about that later. Type

BUILD EXE Chapter3 FROM Chapter3

or use the project’s Build button. Then use DO Chapter3, or press either the Ctrl+D short-cut key or Alt+P, D to activate the menu and bring up the Do dialog screen and select Chapter3.exe.

Figure 3.1 shows the application’s main screen. That third menu selection should look interesting. By the end of this chapter, you’ll be able to click on it and try your forms using either data source. The source code is downloadable from the Sams Web site, www.samspublishing.com. Also, readers can download this or any other source code from my site, www.LesPinter.com.

FIGURE 3.1  The main application screen.

We’ve already done something that would have stopped Visual Basic .NET in its tracks. The DO FormName references to the two forms that don’t yet exist would be compiler errors in .NET. As far as FoxPro knows, the forms are right there in the application directory, or somewhere on the path; we simply have our reasons for excluding the forms from the build. Unlike Visual Basic .NET, FoxPro trusts us to come up with the forms by the time we ask for them.
The Form Template

The basic form template for this application is a flat file form. Most, if not all, applications have one or more of these, and some apps have dozens. So it’s immediately useful, while still possessing sufficient simplicity to make the code easy to understand. When you catch on to the principles, more complex templates and data access methods should be easy to build.

What’s a little tricky is understanding how form templates and the data tier interact. During the Depression, my father said that a humorous way of dismissing the difficult situation at hand was to say “If we had some ham we could have ham and eggs. If we had some eggs...” The form template and the data tier are each written with the other’s function in mind. The reusable components go in the data tier, and the calls to use these components go in the template.

For example, Listing 3.5 shows the class code for my FlatFileForm template. I’ll comment it as we go.

LISTING 3.5 The FlatFileForm Class

```
MainTable = .F.     && Name of the main table for the form
KeyField = .F.     && Name of the Primary Key for the Main Table
KeyValue = .F.     && Value in the KeyField of the current record
Beforeadd = .F.     && Record number before adding (used when Add is canceled)
SearchForm = .F.     && Name of a "pick one" form called by the Find button
Adding = .F.     && True at 'Save' time if 'Add' button was clicked
Inputfields = "MYTEXT,MYCHECK,MYEDIT,MYCOMBO,MYSPIN,MYDATE"
* Classes to enable/disable
Beforeadd = 0 && Record pointer value before an add or edit begins
PROCEDURE Buttons && Turns form buttons on or off as needed
LPARAMETERS OnOff
WITH THISFORM
  .cmdAdd.Enabled = OnOff
  .cmdFind.Enabled = OnOff
  .cmdClose.Enabled = OnOff
  .cmdEdit.Enabled = OnOff AND RECCOUNT() > 0
  .cmdDelete.Enabled = OnOff AND RECCOUNT() > 0
  .cmdSave.Enabled = NOT OnOff
  .cmdCancel.Enabled = NOT OnOff
  .cmdClose.Cancel = OnOff
  .cmdCancel.Cancel = NOT OnOff
ENDWITH
ENDPROC

PROCEDURE Inputs && Enables/Disables form controls
```
LISTING 3.5  Continued

LPARAMETERS OnOff
WITH THISFORM
FOR EACH Ctrl IN .Controls
    IF UPPER ( Ctrl.Class ) $ UPPER (.InputFields )
        Ctrl.Enabled = OnOff
    ENDIF
ENDFOR
.Buttons ( NOT OnOff )
ENDWITH
ENDPROC

PROCEDURE Load
*  Runs when an instance of this form class is instantiated
WITH THISFORM
IF EMPTY (.MainTable )
    MESSAGEBOX( [No main table specified], 16, [Programmer error], 2000 )
    RETURN .F.
ENDIF
oDataTier.CreateCursor ( .MainTable, .Keyfield )
ENDWITH
ENDPROC

PROCEDURE Init        && Runs after buttons have been instantiated
THISFORM.Buttons ( .T. )
ENDPROC

PROCEDURE Unload      && Closes table or cursor opened by this form
WITH THISFORM
IF USED   (.MainTable )
    USE IN (.MainTable )
ENDIF
ENDWITH
ENDPROC

PROCEDURE cmdAdd.Click        && Adds a new record, autopopulating the key field
WITH THISFORM
cNextKey = oDataTier.GetNextKeyValue (.MainTable )
SELECT (.MainTable )
 .BeforeAdd = RECNO()
CURSORSETPROP( [Buffering], 3 )
APPEND BLANK
IF TYPE (.KeyField ) <> [C]
LISTING 3.5  Continued

    cNextKey = VAL ( cNextKey )
    ENDIF
    REPLACE ( .Keyfield ) WITH cNextKey
    .Refresh
    .Inputs { .T. }
    .Adding = .T.
    ENDWITH
ENDPROC

PROCEDURE cmdEdit.Click      && Initiates an edit of the current record
WITH THISFORM
    SELECT ( .MainTable )
    .BeforeAdd = RECNO()
    CURSORSETPROP( [Buffering], 3 )
    .Inputs { .T. }
    .Adding = .F.
    ENDWITH
ENDPROC

PROCEDURE cmdDelete.Click    && Deletes the current record
WITH THISFORM
    IF MESSAGEBOX( [Delete this record?], 292, _VFP.Caption ) = 6
        oDataTier.DeleteRecord ( .MainTable, .KeyField )
        DELETE NEXT 1
        GO TOP
        .Refresh
    ENDIF
    ENDWITH
ENDPROC

PROCEDURE cmdSave.Click
    *  Saves data in local cursor and then remotely (if not DBF)
WITH THISFORM
    SELECT ( .MainTable )
    TABLEUPDATE(.T.)
    CURSORSETPROP( [Buffering], 1 )
    .Inputs { .F. }
    oDataTier.SaveRecord( .MainTable, .KeyField, .Adding )
    ENDWITH
ENDPROC

PROCEDURE cmdCancel.Click    && Cancels an Edit or Add
LISTING 3.5  Continued

WITH THISFORM
SELECT (.MainTable )
TABLEREVERT(.T.)
CURSORSETPROP( [Buffering], 1 )
 Inputs (.F. )
IF BETWEEN (.BeforeAdd, 1 , RECCOUNT() )
  GO (.BeforeAdd )
ENDIF
.Refresh
ENDWITH
ENDPROC

PROCEDURE cmdFind.Click
  * If they're using DBF and no search form is defined, use BROWSE
WITH THISFORM
IF EMPTY (.SearchForm ) AND oDataTier,AccessMethod = [DBF]
  SELECT (.MainTable )
  .BeforeAdd = RECNO()
  ON KEY LABEL ENTER KEYBOARD CHR(23)
  ON KEY LABEL RIGHTCLICK KEYBOARD CHR(23)
  BROWSE NOAPPEND NOEDIT NODELETE
  ON KEY LABEL ENTER
  ON KEY LABEL RIGHTCLICK
  IF LASTKEY() = 27
    IF BETWEEN (.BeforeAdd, 1 , RECCOUNT() )
      GO (.BeforeAdd )
    ENDIF
  ENDIF
ELSE
  DO FORM (.SearchForm ) TO RetVal
  IF NOT EMPTY (RetVal )
    oDataTier.GetOneRecord (.MainTable , .KeyField , RetVal )
    .Refresh
    .Buttons (.T. )
  ENDIF
ENDIF
ENDWITH
ENDPROC

PROCEDURE cmdClose.Click  && Da Svidaniya
THISFORM.Release
ENDPROC
How to Use the FlatFileForm Template

To use this template, you do the following eight steps:

1. Set Field Mappings to use Pinter.VCX and the appropriate controls (MyText, MyChk, and so on) for the data types you use in your tables.

2. Type the following in the command window:
   
   CREATE FORM 'name' AS FlatFileForm FROM Pinter;

3. Open the form, add the DBF that's the MainTable to the Data Environment, drag and drop the fields to the form, and remove the table from the Data Environment.

4. Click on Tab Order and pick Rows, and then move the cmdAdd and cmdEdit objects to the top of the Tab Order list.

5. Set the form's MainTable and KeyField properties to the table name and key field name, respectively.

6. Remove any fields that you don't want users to enter or edit, like CreateDate or RecordID. (If you prefer, you can browse your SCX, find the controls you want to make noneditable, and change the value in the Class field of the SCX to noedit, which is a disabled TextBox control in pinter.vcx. Because it doesn't appear in the InputFields property list of editable fields, it never gets enabled.)

7. If you want the KeyField value to appear on the screen, add a label somewhere on the form, and add this code in the form's Refresh method:

   ```
   THISFORM.txtKeyField.Caption = ;
   TRANSFORM ( EVALUATE ( THISFORM.MainTable + .[.];
   + THISFORM.KeyField ))
   ```

8. Add a DO FORM "name" bar to the menu. Recompile and run it, and it ought to work, except for the search form, which is next.

I've followed my instructions and built the little Customer form shown in Figure 3.2 in about 90 seconds.

That's a pretty simple form template, and you might wonder if it really does everything you need. It does if you include the data tier and the search form.
A Search Form Template

I didn’t include any navigation in the FlatFileForm template because Next/Previous/First/Last record is an artifact of the xBASE world. We could provide those four features with a single line of code each (not counting checking for BOF()/EOF() conditions). But users typically don’t care about the previous or next record. They want to see a candidate list, point to the one they want, and click. Besides, it’s nearly impossible if you use SQL Server, and that’s what we’re going to do next.

I’ve included a class called EasySearch. It allows you to add a search form with up to 4 searchable fields (and it’s easy to extend that to 8 or 10 if you need them), to let users filter records and pick one, and to return the key value, from a DBF, SQL, or a Web service, with absolutely no coding in the form itself. You simply fill in three or four properties, name the input fields that you put on the search form with names SEARCH1, SEARCH2, SEARCH3, and SEARCH4, set the form’s tab order to Row order, and you’re done. The code for this class is shown in Listing 3.6.

LISTING 3.6 The EasySearch Class

```plaintext
DEFINE CLASS EasySearch AS modalform

tablename = ([]))  && Table name to search
colwidths = ([]))  && Comma-delimited list of the relative widths
colnames  = ([]))  && Comma-delimited list of field names
orderby   = ([]))  && "Order by" column name
colheadings = ([]))  && Comma-delimited list if you don't want to use
                 * field names as headings
keyfield  = ([]))  && Name of key field value to return
```

FIGURE 3.2 A sample customer form based on the FlatFileForm class.
LISTING 3.6  Continued

PROCEDURE Init
WITH THISFORM
   .Caption = [Search form - ] + .Name + [ (Main Table: ] ;
      + TRIM(.TableName)+[]) Data access: ] + .Access
NumWords = GETWORDCOUNT(.ColNames,[,])
IF NumWords > 4
   MESSAGEBOX( [This class only supports a maximum of 4 fields, sorry], ;
      16, _VFP.Caption )
   RETURN .F.
ENDIF
FOR I = 1 TO NumWords
   .Field(I)   = GETWORDNUM(.ColNames, I,[,])
   .Heading(I) = GETWORDNUM(.ColHeadings, I,[,])
   .ColWidth(I)= GETWORDNUM(.ColWidths, I,[,])
ENDFOR
WITH .Grid1
   .ColumnCount        = NumWords
   .RecordSource        = THISFORM.ViewName
   .RecordSourceType    = 1
   GridWidth = 0
FOR I = 1 TO NumWords
   GridWidth = GridWidth         +  VAL( THISFORM.ColWidth(I) )
   FldName   = THISFORM.ViewName + [.] + THISFORM.Field   (I)
   .Columns(I).ControlSource      = FldName
ENDFOR
Multiplier = ( THIS.Width / GridWidth ) * .90      && "Fudge" factor
FOR I = 1 TO NumWords
   .Columns(I).Width = VAL( THISFORM.ColWidth(I) ) * Multiplier
ENDFOR
 .Refresh
ENDWITH
* Look for any controls named SEARCHn (n = 1, 2, ...)
FOR I = 1 TO .ControlCount
   Ctrl = .Controls(I)
   IF UPPER(Ctrl.Name) = [MYLABEL]   && That is, if it starts with "MyLabel"
      Sub = RIGHT(Ctrl.Name,1)       && Determine the index
      IF TYPE([THISFORM.Search]+Sub)=[O]  && A search field #"Sub" exists
         Ctrl.Visible = .T.
         Ctrl.Enabled = .T.
         Ctrl.Caption = .Heading(VAL(Sub))
      .SearchFieldCount = MAX ( VAL(Sub), .SearchFieldCount )
PROCEDURE Load
WITH THISFORM
IF EMPTY (.TableName )
    MESSAGEBOX( [Table name not entered], 16, _VFP.Caption )
    RETURN .F.
ENDIF
IF EMPTY (.ColNames )
    Msg = [ColNames property not filled in.]
    MESSAGEBOX( Msg, 16, _VFP.Caption )
    RETURN .F.
ENDIF
IF EMPTY (.ColWidths )
    .ColWidths = [1,1,1,1,1]
ENDIF
IF EMPTY (.ColHeadings )
    .ColHeadings = .ColNames
ENDIF .Access = oDataTier.AccessMethod
.ViewName = [View] + .TableName
oDataTier.CreateView (.TableName )
ENDWITH
ENDPROC

PROCEDURE Unload
WITH THISFORM
IF USED (.ViewName )
    USE IN (.ViewName )
ENDIF
RETURN .ReturnValue
ENDWITH
ENDPROC

PROCEDURE cmdShowMatches.Click
WITH THISFORM
Fuzzy = IIF ( THISFORM.Fuzzy.Value = .T., [%], [ ] )
LISTING 3.6  Continued

STORE [] TO Expr1,Expr2,Expr3,Expr4
FOR I = 1 TO .SearchFieldCount
    Fld = [THISFORM.Search] + TRANSFORM(I) + [.Value]
    IF NOT EMPTY ( &Fld )
        LDD = IIF ( VARTYPE( &Fld) = [D],
                    IIF (.Access = [DBF],[{|},{ }],);
                    IIF(VARTYPE( &Fld) = [C],[{},{ }])
        RDD = IIF ( VARTYPE( &Fld) = [D],
                    IIF (.Access = [DBF],[{|},{ }],);
                    IIF(VARTYPE( &Fld) = [C],[{},{ }])
        Cmp = IIF ( VARTYPE( &Fld) = [C], [ LIKE ],[ = ]
        Pfx = IIF ( VARTYPE( &Fld) = [C], Fuzzy, []
        Sfx = IIF ( VARTYPE( &Fld) = [C], [%], []
        Exp = [Expr]  + TRANSFORM(I)
        &Exp = [ AND UPPER(] + .Field(I) + []) + Cmp ;
                    + LDD + Pfx + UPPER(ALLTRIM(TRANSFORM(EVALUATE(Fld)))) + Sfx + RDD
    ENDIF
ENDFOR
lcExpr = Expr1 + Expr2 + Expr3 + Expr4
IF NOT EMPTY ( lcExpr )
    lcExpr = [ WHERE ] + SUBSTR ( lcExpr, 6 )
ENDIF
lcOrder = IIF(EMPTY(.OrderBy),[],[ ORDER BY ] ;
                    + ALLTRIM(STRTRAN(.OrderBy,[ORDER BY],[()])))
Cmd = [SELECT * FROM ] + .TableName + lcExpr + lcOrder
oDataTier.SelectCmdToSQLResult ( Cmd )
SELECT ( .ViewName )
ZAP
APPEND FROM DBF([SQLResult])
GO TOP
/Grid1.Refresh
IF RECCOUNT() > 0
    .cmdSelect.Enabled = .T.
    .Grid1.Column1.Alignment = 0
    .Caption = [Search Form - ] + PROPER(.Name) ;
                    + [ () + TRANSFORM(RECCOUNT()) + [ matches]]
ELSE
    .Caption = [Search Form - ] + PROPER(.Name)
LISTING 3.6  Continued

MESSAGEBOX( "No records matched" )
.cmdSelect.Enabled = .F.
ENDIF
KEYBOARD {{BackTab}{{BackTab}{{BackTab}{{BackTab}{{BackTab}}}
ENDWITH
ENDPROC

PROCEDURE cmdClear.Click
WITH THISFORM
FOR I = 1 TO .SearchFieldCount
  Fld = [THISFORM.Search] + TRANSFORM(I) + ['.Value']
  IF VARTYPE ( &Fld ) <> [U]
    lVal = IIF ( VARTYPE( &Fld) = [C], [],
              IIF ( VARTYPE( &Fld) = [D], {/},
              IIF ( VARTYPE( &Fld) = [L], .F.,
              IIF ( VARTYPE( &Fld) $ [IN], 0, [?])))
    &Fld  = lVal
  ENDIF
ENDFOR
ENDWITH
ENDPROC

PROCEDURE cmdSelect.Click
WITH THISFORM
lcStrValue = TRANSFORM(EVALUATE(.KeyField))
 .ReturnValue = lcStrValue
 .Release
ENDWITH
ENDPROC

PROCEDURE cmdCancel.Click
WITH THISFORM
 .ReturnValue = []
 .Release
ENDWITH
ENDPROC

ENDDEFINE
How to Use the EasySearch Template

Here's an example of how to use this template in five easy steps:

1. Type this line in the command window:
   ```
   CREATE FORM FindCust AS EasySearch FROM Pinter
   ```

2. Add two text boxes and a StatesList combo box. Name these three controls
   `Search1`, `Search2`, and `Search3`.

3. Set the Tab Order to Rows on the search form.

4. Set the `MainTable` property to Customers, the `KeyField` property to `CustomerID`, and
   the `ColumnList` property to `[CompanyName, ContactName, Phone].`

5. Enter `FindCust` as the `SearchForm` property value in your Customers form.

Figure 3.3 shows the screen for the FindCust form.

![The FindCust form.](image)

FIGURE 3.3 The FindCust form.

The Data Tier

But there's still a missing piece—the DataTier class library that's the subject of this
chapter. Now that you've seen all of the places where its methods are called, you should
be able to match up the calls from Listing 3.6 with the methods in Listing 3.7 and see
how they fit together.

The data tier is a class used to instantiate an object called `oDataTier` in `MAIN.PRG.`
Thereafter, any time we need to send data to a data store, we call a method on the
`oDataTier` object.
It's important to note that in this methodology, you either open a DBF or a CURSOR for each table object. If you initiate either an Add or an Edit, you set buffering mode to 3, change something, and then call TableUpdate() and set Buffering back to 1. (If your user cancels, you refresh the screen with whatever was there before—the original, unchanged record. TableRevert() does that, as long as you add a little code to go back to the record you were pointing to before the APPEND BLANK step.) If you were using FoxPro, you'd be home now.

However, if you're using SQL Server or an XML Web Service, you're only halfway there. You've done the equivalent in .NET of saving changes to a dataset. Now you need to use the saved data to update the data store, which might be on SQL Server or in Timbuktu. So we still need the TableUpdate() and TableRevert() function calls in the Save and Cancel buttons. We just have to add one more call to the DataTier object to see what else to do. If we're using DBFs, as you'll see, it simply excuses itself and returns. Similarly, in the form's Load event, we call the CreateCursor method of this object, which either creates a cursor, or, in the case of DBF access, opens the appropriate table and sets the index tag.

The proper way to read this code is to find the place in the form template code where each routine is being called, and then see what the DataTier routine does after the template code runs. It's the combination of the template code and the DataTier code that works the magic. Listing 3.7 shows the DataTier code.

LISTING 3.7  DataTier.PRG

DEFINE CLASS DataTier AS Custom
AccessMethod = []
* Any attempt to assign a value to this property will be trapped
* by the "setter" method AccessMethod_Assign.
ConnectionString = ;
              [Driver={SQL Server};Server=(local);Database=Mydatabase;UID=sa;PWD=;]
Handle       = 0

* Betcha didn't know you could write your own Assign methods...

PROCEDURE AccessMethod_Assign
PARAMETERS AM
DO CASE
CASE AM = [DBF]
      THIS.AccessMethod = [DBF]  && FoxPro tables
CASE AM = [SQL]
      THIS.AccessMethod = [SQL]  && MS Sql Server
      THIS.GetHandle
CASE AM = [XML]
      THIS.AccessMethod = [XML]  && FoxPro XMLAdapter
CASE AM = [WC]
LISTING 3.7  Continued

    THIS.AccessMethod = [WC] & WebConnection server
    OTHERWISE
        MESSAGEBOX( Incorrect access method ] + AM, 16, Setter error )
        THIS.AccessMethod = []
    ENDCASE
    _VFP.Caption = [Data access method: ] + THIS.AccessMethod
ENDPROC

* CreateCursor actually opens the DBF if AccessMethod is DBF;
* otherwise it uses a structure returned from the data
* store to create a cursor that is bound to the screen controls:

PROCEDURE CreateCursor
LPARAMETERS pTable, pKeyField
IF THIS.AccessMethod = [DBF]
    IF NOT USED ( pTable )
        SELECT 0
        USE ( pTable ) ALIAS ( pTable )
    ENDIF
    SELECT ( pTable )
    IF NOT EMPTY ( pKeyField )
        SET ORDER TO TAG ( pKeyField )
    ENDIF
    RETURN
ENDIF
Cmd = [SELECT * FROM ] + pTable + [ WHERE 1=2]
DO CASE
    CASE THIS.AccessMethod = [SQL]
        SQLEXEC( THIS.Handle, Cmd )
        AFIELDS ( laFlds )
        USE
        CREATE CURSOR ( pTable ) FROM ARRAY laFlds
    CASE THIS.AccessMethod = [XML]
    CASE THIS.AccessMethod = [WC]
ENDCASE

* GetHandle is called in the Assign method of the AccessMethod
* property earlier in this listing (two procedures back).

PROCEDURE GetHandle
IF THIS.AccessMethod = [SQL]
LISTING 3.7  Continued

IF THIS.Handle > 0
    RETURN
ENDIF

THIS.Handle = SQLSTRINGCONNECT( THIS.ConnectionString )
IF THIS.Handle < 1
    MESSAGEBOX( [Unable to connect], 16, [SQL Connection error], 2000 )
ENDIF
ELSE
    Msg = [A SQL connection was requested, but access method is ] ;
    + THIS.AccessMethod
    MESSAGEBOX( Msg, 16, [SQL Connection error], 2000 )
    THIS.AccessMethod = []
ENDIF
RETURN

PROCEDURE GetMatchingRecords
LPARAMETERS pTable, pFields, pExpr
pFields = IIF ( EMPTY ( pFields ), [], pFields )
pExpr   = IIF ( EMPTY ( pExpr ), [],
    [ WHERE ] + STRTRAN ( UPPER ( ALLTRIM ( pExpr ) ), [WHERE ], [] )
)cExpr   = [SELECT ] + pFields + [ FROM ] + pTable + pExpr
IF NOT USED ( pTable )
    RetVal = THIS.CreateCursor ( pTable )
ENDIF
DO CASE
    CASE THIS.AccessMethod = [DBF]
        &cExpr
    CASE THIS.AccessMethod = [SQL]
        THIS.GetHandle()
        IF THIS.Handle < 1
            RETURN
        ENDIF
        lr = SQLExec ( THIS.Handle, cExpr )
        IF lr >= 0
            THIS.FillCursor()
        ELSE
            Msg = [Unable to return records] + CHR(13) + cExpr
            MESSAGEBOX( Msg, 16, [SQL error] )
        ENDIF
ENDCASE
ENDPROC
In my EasySearch template, I open up a new cursor, the name of which is “View” followed by the name of the table that the data is coming from. So it’s not a view, just a cursor name:

```vbnet
PROCEDURE CreateView
LPARAMETERS pTable
IF NOT USED( pTable )
    MESSAGEBOX( [Can’t find cursor ] + pTable, 16, [Error creating view], 2000 )
    RETURN
ENDIF
SELECT ( pTable )
AFIELDS( laFlds )
SELECT 0
CREATE CURSOR ( [View] + pTable ) FROM ARRAY laFlds
ENDFUNC
```

GetOneRecord is called with all types of data stores. If we’re using a DBF, a LOCATE command that refers to the current index tag is Rushmore-optimized and very fast. With SQL, it’s also Rushmore-optimized, because SQL 2000 uses Rushmore—with not a single mention of where it came from. But we know...

```vbnet
PROCEDURE GetOneRecord
LPARAMETERS pTable, pKeyField, pKeyValue
SELECT ( pTable )
Dlm = IIF ( TYPE ( pKeyField ) = [C], [', ['] ] )
IF THIS.AccessMethod = [DBF]
    cExpr = [LOCATE FOR ] + pKeyField + [=] + Dlm + TRANSFORM ( pKeyValue ) + Dlm
ELSE
    cExpr = [SELECT * FROM ] + pTable ;
        + [ WHERE ] + pKeyField + [=] + Dlm + TRANSFORM ( pKeyValue ) + Dlm
ENDIF
DO CASE
    CASE THIS.AccessMethod = [DBF]
        &cExpr
    CASE THIS.AccessMethod = [SQL]
        lr = SQLExec ( THIS.Handle, cExpr )
        IF lr >= 0
            THIS.FillCursor( pTable )
        ELSE
            Msg = [Unable to return record] + CHR(13) + cExpr
            MESSAGEBOX( Msg, 16, [SQL error] )
        ENDIF
    CASE THIS.AccessMethod = [XML]
    CASE THIS.AccessMethod = [WC]
```
FillCursor is analogous to the Fill method of .NET's DataAdapters. FoxPro's ZAP is like the .NET DataAdapter.Clear method. By appending FROM DBF(CursorName) into the named cursor, we don't break the data binding with the form's controls:

```plaintext
PROCEDURE FillCursor
LPARAMETERS pTable
IF THIS.AccessMethod = [DBF]
     RETURN
ENDIF
SELECT ( pTable )
ZAP
APPEND FROM DBF ( [SQLResult] )
USE IN SQLResult
GO TOP
ENDPROC

I'd like to think that all primary keys were going to be integers, but legacy systems sometimes have character keys. That's what the check for delimiters is in the DeleteRecord routine:

```plaintext
PROCEDURE DeleteRecord
LPARAMETERS pTable, pKeyField
IF THIS.AccessMethod = [DBF]
     RETURN
ENDIF
KeyId = EVALUATE ( pTable + [.] + pKeyField )
Dlm = IIF ( TYPE ( pKeyField ) = [C], [''], [] )
DO CASE
    CASE THIS.AccessMethod = [SQL]
        cExpr = [DELETE ] + pTable + [ WHERE ] + pKeyField + [=];
        + Dlm + TRANSFORM ( m.KeyValue ) + Dlm
        lr = SQLExec ( THIS.Handle, cExpr )
        IF lr < 0
            Msg = [Unable to delete record] + CHR(13) + cExpr
            MESSAGEBOX ( Msg, 16, [SQL error] )
        ENDIF
    CASE THIS.AccessMethod = [XML]
    CASE THIS.AccessMethod = [WC]
ENDCASE
ENDPROC
```
The `SaveRecord` routine either does an `INSERT` or an `UPDATE` depending on whether the user was adding or not:

```
PROCEDURE SaveRecord
PARAMETERS pTable, pKeyField, pAdding
IF THIS.AccessMethod = [DBF]
    RETURN
ENDIF
IF pAdding
    THIS.InsertRecord ( pTable, pKeyField )
ELSE
    THIS.UpdateRecord ( pTable, pKeyField )
ENDIF
ENDPROC
```

The `InsertRecord` and `UpdateRecord` routines call corresponding functions that build SQL `INSERT` or `UPDATE` commands, in much the same way that .NET does. I store the resulting string to `_ClipText` so that I can open up Query Analyzer and execute the command there manually to see what went wrong. It's the fastest way to debug your generated SQL code. Finally, I use `SQLExec()` to execute the command. `SQLExec()` returns -1 if there's a problem.

```
PROCEDURE InsertRecord
PARAMETERS pTable, pKeyField
    cExpr = THIS.BuildInsertCommand ( pTable, pKeyField )
    _ClipText = cExpr
    DO CASE
        CASE THIS.AccessMethod = [SQL]
            lr = SQLExec ( THIS.Handle, cExpr )
            IF lr < 0
                msg = [Unable to insert record; command follows:] + CHR(13) + cExpr
                MESSAGEBOX( Msg, 16, [SQL error] )
            ENDIF
        CASE THIS.AccessMethod = [XML]
        CASE THIS.AccessMethod = [WC]
        ENDCASE
    ENDFUNC
```

```
PROCEDURE UpdateRecord
PARAMETERS pTable, pKeyField
    cExpr = THIS.BuildUpdateCommand ( pTable, pKeyField )
    _ClipText = cExpr
    DO CASE
        CASE THIS.AccessMethod = [SQL]
```
lr = SQLExec ( THIS.Handle, cExpr )
   IF lr < 0
      msg = [Unable to update record; command follows:] + CHR(13) + cExpr
      MESSAGEBOX( Msg, 16, [SQL error] )
   ENDIF
   CASE THIS.AccessMethod = [XML]
   CASE THIS.AccessMethod = [WC]
   ENDCASE
ENDFUNC

FUNCTION BuildInsertCommand
PARAMETERS pTable, pKeyField
Cmd = [INSERT ] + pTable + [ ( ]
FOR I = 1 TO FCOUNT()
   Fld = UPPER(FIELD(I))
   IF TYPE ( Fld ) = [G]
      LOOP
   ENDIF
   Cmd = Cmd + Fld + [ , ]
ENDFOR
Cmd = LEFT(Cmd,LEN(Cmd)-2) + [ ) VALUES ( ]
FOR I = 1 TO FCOUNT()
   Fld = FIELD(I)
   IF TYPE ( Fld ) = [G]
      LOOP
   ENDIF
   Dta = ALLTRIM(TRANSFORM ( &Fld ))
   Dta = CHRTRAN ( Dta, CHR(39), CHR(146) )
   get rid of single quotes in the data
   Dta = IIF ( Dta = [ / /], [ ], Dta )
   Dta = IIF ( Dta = [.F.], [0], Dta )
   Dta = IIF ( Dta = [.T.], [1], Dta )
   Dlm = IIF ( TYPE ( Fld ) $ [CM],[',];
            IIF ( TYPE ( Fld ) $ [DT],[',];
            IIF ( TYPE ( Fld ) $ [IN],[], [ ]))
   Cmd = Cmd + Dlm + Dta + Dlm + [ , ]
ENDFOR
Cmd = LEFT ( Cmd, LEN(Cmd) -2 ) + [ ] && Remove ' , " add '"
RETURN Cmd
ENDFUNC

FUNCTION BuildUpdateCommand
PARAMETERS pTable, pKeyField
Cmd = [UPDATE ] + pTable + [ SET ]
FOR I = 1 TO FCOUNT()
    Fld = UPPER(FIELD(I))
    IF Fld = UPPER(pKeyField)
        LOOP
    ENDIF
    IF TYPE ( Fld ) = [G]
        LOOP
    ENDIF
    Dta = ALLTRIM(TRANSFORM ( &Fld ))
    IF Dta = [.NULL.]
        DO CASE
            CASE TYPE ( Fld ) $ [CMDT]
                Dta = []
            CASE TYPE ( Fld ) $ [INL]
                Dta = [0]
        ENDCASE
    ENDIF
    Dta = CHRTRAN ( Dta, CHR(39), CHR(146) )
    * get rid of single quotes in the data
    Dta = IIF ( Dta = [ / ] , [], Dta )
    Dta = IIF ( Dta = [.F.] , [0], Dta )
    Dta = IIF ( Dta = [.T.] , [1], Dta )
    Dlm = IIF ( TYPE ( Fld ) $ [CM],[', ];
            IIF ( TYPE ( Fld ) $ [DT],[],' );
            IIF ( TYPE ( Fld ) $ [IN],[], [ ] ) )
    Cmd = Cmd + Fld + [=] + Dlm + Dta + Dlm + [, ]
ENDFOR
Dlm = IIF ( TYPE ( pKeyField ) = [C], [', ], [ ] )
Cmd = LEFT ( Cmd, LEN(Cmd) - 2 )
    ; + [ WHERE ] + pKeyField + [=] ;
    + + Dlm + TRANSFORM(EVALUATE(pKeyField)) + Dlm
RETURN Cmd
ENDFUNC

Sometimes I need to return a cursor that I'll use for my own purposes, for example, to load a combo box. I use the default name SQLResult. I only name it here because if I use FoxPro's SELECT, it returns the cursor into a BROWSE by default, and I need to ensure that the cursor name will be SQLResult when I return from this procedure:

PROCEDURE SelectCmdToSQLResult
LPARAMETERS pExpr
DO CASE
    CASE THIS.AccessMethod = [DBF]
When I add a new record, whether I use DBFS or SQL, I'm responsible for inserting a unique value into the table. So I maintain a table of table names and the last key used. If you create this table manually, be sure to update the LastKeyVal field manually before going live, or you'll get thousands of duplicate keys.

FUNCTION GetNextKeyValue
LPARAMETERS pTable
EXTERNAL ARRAY laVal
pTable = UPPER ( pTable )
DO CASE

CASE THIS.AccessMethod = [DBF]
    IF NOT FILE ( [Keys.DBF] )
        CREATE TABLE Keys ( TableName Char(20), LastKeyVal Integer )
    ENDIF
    IF NOT USED ( [Keys] )
        USE Keys IN 0
    ENDIF
    SELECT Keys
    LOCATE FOR TableName = pTable
    IF NOT FOUND()
        INSERT INTO Keys VALUES ( pTable, 0 )
    ENDIF
    Cmd = [UPDATE Keys SET LastKeyVal=LastKeyVal + 1 ]
    + [ WHERE TableName='] + pTable + [']
&Cmd

CASE THIS.AccessMethod = [SQL]
    THIS.GetHandle()
    IF THIS.Handle < 1
        RETURN
    ENDIF
    lr = SQLExec ( THIS.Handle, pExpr )
    IF lr < 0
        Msg = [Unable to return records] + CHR(13) + cExpr
        MESSAGEBOX( Msg, 16, [SQL error] )
    ENDIF

CASE THIS.AccessMethod = [XML]
CASE THIS.AccessMethod = [WC]
ENDCASE
ENDFUNC
CHAPTER 3  Building a Visual FoxPro Application for SQL Server

```foxpro
Cmd = [SELECT LastKeyVal FROM Keys WHERE TableName = ']
+ pTable + [' INTO ARRAY laVal
&Cmd
USE IN Keys
RETURN TRANSFORM(laVal(1))

CASE THIS.AccessMethod = [SQL]

Cmd = [SELECT Name FROM SysObjects WHERE Name='KEYS' AND Type='U'
lr = SQLEXEC( THIS.Handle, Cmd )
IF lr < 0
  MESSAGEBOX( "SQL Error:"+ CHR(13) + Cmd, 16 )
ENDIF
IF RECCOUNT([SQLResult]) = 0
  Cmd = [CREATE TABLE Keys ( TableName Char(20), LastKeyVal Integer )]
  SQLEXEC( THIS.Handle, Cmd )
ENDIF
Cmd = [SELECT LastKeyVal FROM Keys WHERE TableName=']
+ pTable + ['
lr = SQLEXEC( THIS.Handle, Cmd )
IF lr < 0
  MESSAGEBOX( "SQL Error:"+ CHR(13) + Cmd, 16 )
ENDIF
IF RECCOUNT([SQLResult]) = 0
  Cmd = [INSERT INTO Keys VALUES (']
+ pTable + [' , 0 )]
  lr = SQLEXEC( THIS.Handle, Cmd )
  IF lr < 0
    MESSAGEBOX( 'SQL Error:'+ CHR(13) + Cmd, 16 )
  ENDIF
ENDIF
Cmd = [UPDATE Keys SET LastKeyVal=LastKeyVal + 1 WHERE TableName=']
+ pTable + ['
lr = SQLEXEC( THIS.Handle, Cmd )
IF lr < 0
  MESSAGEBOX( "SQL Error:"+ CHR(13) + Cmd, 16 )
ENDIF
Cmd = [SELECT LastKeyVal FROM Keys WHERE TableName=']
+ pTable + ['
lr = SQLEXEC( THIS.Handle, Cmd )
IF lr < 0
  MESSAGEBOX( "SQL Error:"+ CHR(13) + Cmd, 16 )
ENDIF
nLastKeyVal = TRANSFORM(SQLResult.LastKeyVal)
USE IN SQLResult
RETURN TRANSFORM(nLastKeyVal)
```

04 0672326493  CH03  4/14/04  10:49 AM  Page 132
CASE THIS.AccessMethod = [WC]
CASE THIS.AccessMethod = [XML]
ENDCASE
ENDDDEFINE

Running the Application

Type

BUILD EXE Chapter3 FROM Chapter3

or use the Build button in the Project Manager. When you have an executable, run it and try each of the forms, as well as the search screens. It works pretty nicely, as we're used to seeing with DBF tables.

Now, select Change Data Source from the menu and enter SQL, as shown in Figure 3.4.

![Figure 3.4](image)

My application

FIGURE 3.4 Changing the data access method.

Now, open the Customer form. There's no data! That's standard for a SQL application because until the user requests a record, they don't have one yet. So click on Find, enter some search criteria (actually, to return all records, just leave them all blank), click on Show Matches, and select one. The search form disappears, and you've got data! Select Edit, and then change something and save it. Reload the page to verify that it worked. Add a record, save it, and see if the search form finds it.
Nothing special happens, except that you just built a FoxPro application that works with either DBFs or SQL without writing a single line of code in any of the forms.

**What’s Next?**

In the next chapter, we’ll build the same application in Visual Basic .NET. That way, you’ll see for yourself why I suggested earlier that in .NET, local data access is harder than it is in FoxPro, but SQL access is no harder than local data access.