Chapter 16
Customizing Your Development Environment

It never ceases to amaze me to walk into a development shop and see how the developers work. The most mind-numbing experience is to watch a developer start up Visual FoxPro, and then issue three or four attempts of the "CD <current app dev directory>" command as his first step—because VFP's default location is still set somewhere deep in the bowels of Program Files. Think of all the time you spend building applications to make other people more productive. Doesn't it make sense to tune your own work environment? In this chapter, I'll show you a number of techniques that you can use to make your daily work easier and more bug-proof.

The purpose of this chapter is to introduce to you some ideas and concepts on how to tune your development environment. But before I start, let me warn you that the methodology presented here is just one means to an end—one way of accomplishing three important goals.

The first goal is to customize the environment so that it works well for you and your style. Different developers have different needs—one of you is probably going to spend the next two years on a single app, while another will work on pieces of over a dozen applications. Some of you are going to stay strictly LAN-based for the foreseeable future; others are heading full barrel into Web-based and distributed development. No one style is going to work the same for everyone.

The second goal is to optimize your development environment. Do you really want to go through four steps every 10 minutes when you could simply tap a hot spot on your computer screen or press a hotkey? Just like "Quick Carl" in those old Marathon Bar advertisements, I like to do everything "fast-fast-fast"—and I hate waiting for anything on my computer as well. Ever get up and walk into another room to get something, and by the time you're in that room, you've forgotten what it was you were going to get? Application development is based on ideas, and as your brain is cranking out the idea, the trick is to keep the response time or turnaround time of the machine shorter than your attention span.

The third, and most subtle, goal is to set up shop so that you are less likely to create bugs. Some of those attributes have nothing to do with the bits and bytes on your hard disk—a big work desk, a big monitor, plenty of hard disk space, an office where you can close the door and shut out the noise so you can concentrate—they're all part of the package. But there are lots of things you can do while configuring Visual FoxPro and structuring your application development directory to prevent you from doing "stupid" things. I've long avoided keeping an open beverage on my desk; I feel an open mug of liquid perched above my system unit, ready and waiting for me to knock it over, is an accident waiting to happen, and I think I'll try to avoid that one. Similarly, how can you construct your development environment so as to avoid those "accidents waiting to happen"?

All too often I see directories for all of a developer's custom apps located under the Visual FoxPro directory, right next to "\API", "\DISTRIB.SRC" and "\FFC." Or if they've made the

conceptual leap of making separate directory entries from each application, perhaps even all stored in a parent "applications" directory, the entire application is stored in one directory—source code, data, common libraries, third-party utilities, and so on. It's a recipe for disaster if there ever was one. It's all too easy, for instance, to take a copy of the application to the customer's site and accidentally overwrite their live data with your copy of the table that contains entries for Bugs Bunny and Jeri Ryan. I imagine it would be pretty embarrassing to have to hunt down the network administrator to find out how recent their last backup is after blowing away 170 megabytes of data.

You're certainly free to pick and choose from the ideas I'll present here and determine what works best for you. I'll present problems I've run into in the past, mention some problems that you'll likely run into in the future, and present some strategies and techniques for dealing with those issues. And I'd love to hear from you about tricks and tips you've developed to tune your own environment.

To get started, then, I'm going to begin at the beginning. I brushed over installation in Chapter 1 because you needed to get into the meat of VFP right away. Now that you're comfortable with data, programs, and classes, and have even possibly built a couple sample applications, it's time to set up things correctly.

Machine setup and installation

I think it's a reasonable assumption that, at the end of the 20th century, every database developer on the planet should be developing on a LAN, or at least have access to one on a regular basis. However, I also think there are significant benefits to doing your primary development on a workstation. As a result, I'm going to be a little schizophrenic in my suggestions.

First of all, you have to install Visual FoxPro on your local workstation. Unlike older programs, where you could get away with an install on the server and then run the program from that box, current Windows programs are tightly integrated with the workstation upon which they're running. (This also goes for applications you build for others—they have to install your application on each user's machine.)

Before you go any further in the installation process, consider this: What should your computer look like? I'm going to assume that you keep all of your "data" on a server of some sort; if you don't, you can just substitute a drive on your local machine for the server. I think it makes sense to have at least two drives available on your local machine. One will be for your shrink-wrap software, and the other is for your data.

Figure you're going to have to reinstall Windows sooner or later (probably sooner)—don't be one of those sorry goofs who's got 800 megabytes of data buried in the depths of their various C-drive directories. I spent a couple of hours on the phone with a frantic acquaintance after she had watched her machine disintegrate before her eyes during the Office 2000 beta. Turns out she had just taken the computer out of the box and started working with it. One C drive—that's all. Had installed beta after beta on the machine over a period of years, trusting that "uninstall" actually would. She had literally thousands of documents stored in My Documents, and thousands more data files scattered throughout the rest of her drive. It took her a week to back up those files to a hastily purchased ZIP drive before she could get her machine working again.

As you pull out the two or three dozen CDs (well, it seems like that many! Remember when software used to come on a single CD?) that make up Visual Studio, you'll realize that the 2-gigabyte partition that your C drive was created in no longer seems sufficient.

You have two choices. If you have the luxury of creating a larger C drive partition, 4 to 6 gigabytes should be enough for most developers. (I still remember being able to boot and run my primary development environment from a 1.2-megabyte floppy!)

If you're stuck with a 2-gigabyte C drive, as I am (I got a new box a few months ago and I just couldn't bear to do another FDISK and NT reinstall this year), the alternative is to do minimal installs of everything you possibly can. I do a complete install of VFP and VB, bail on the installs of other Visual Studio programs (C++ takes 500 meg, and I'm not about to become a C++ junkie), and then selectively pick and choose among options in Office and other programs. In addition, I've located MSDN on my 2-gigabyte D drive—allowing me to load every help file I could find on the drive with some room to spare.

In either case, the primary philosophy I recommend is to keep everything except shrink-wrap off of your C drive. There are two kinds of developers—those who can FDISK and reinstall NT without turning the monitor on, and those who are going to learn to.

As a result, I have a binder with step-by-step instructions on how to set up each machine in my office: the equipment inside the box (for example, weird video cards), the programs that had to be loaded, the location of the programs (do you have 1700 CDs lying around your office, too?), and the various settings (such as IP addresses) needed at each point during the install.

Because each box that shows up ends up being configured a little differently (ZIP drives, CD-ROM or DVD drives, video cards, adapters, and so on), I also have a directory on my server that contains a copy of every driver and program that needs to be installed on each machine. That way I can keep straight the Diamond drivers for my Tahoe machine and the STB drivers for my Indy machine. (I also keep a copy of these directories on the local drives just in case the install—or reinstall—hoses the network connection.)

I'll address the rest of your setup—directories for data and whatnot—shortly.

Startup configuration

The first time you click on the Visual FoxPro icon, you'll be presented with the Welcome screen. Just about every developer I know immediately checks the "Don't ever, ever, ever show this screen again" check box in the lower left corner of the form. If you're one of those people but want to run the form again, you can launch the VFP6STRT.APP program in VFP's home directory.

By default, VFP 6.0 is installed the C:\Program Files\Microsoft Visual Studio\VFP98 directory.

Default directory

As I mentioned earlier, you don't ever, ever want to start Fox in this directory, because sooner or later you'll end up creating files in that directory. You can control the VFP startup location through two different mechanisms.

The first is to simply change the "Start in" location in the desktop icon's Properties dialog. (Right-click on the icon on the desktop and select the Properties menu command.) Unfortunately, there's no "browse" button, so you have to manually enter the location yourself.

By the way, if you've enabled the Windows Quick Launch toolbar in the Start menu task bar, you can drag a copy of your VFP desktop icon to it. (Right-click in the task bar, and select the Toolbars, Quick Launch menu option to enable it.) I do this for the half-dozen or so applications that I use all the time, instead of having to close or minimize all open windows to get to my desktop icons. However, once you copy the desktop icon to the Quick Launch toolbar, the values in the Properties window take on a life of their own. Specifically, if you change the "Start in" property in the desktop icon's Properties window, that change is not reflected in the Quick Launch icon's Properties window. This drove me nuts for a couple of days until I realized I was launching VFP from both icons at different times.

By the way, the default startup location isn't the only thing you might want to change. If you've got more than one version of Fox (any version) installed on your machine, you might find it confusing to have multiple icons on your desktop. That's why I've kept creating new icons for each new release of FoxPro since FoxPro 2.0 was released sometime in the early 17th century. You can find icons for each version since 2.6 in the CH16 source code downloads for this book.

Okay, enough whimsy. You can also change VFP's default directory in the File Locations tab of the Tools, Options dialog, as shown in **Figure 16.1**.

Entering a value in the Tools, Options dialog will override any value you enter in the Properties window of the VFP desktop or Quick Launch icon.

However, just because you can now automatically load VFP and have it start in a different default location doesn't mean that you'll never return to VFP's home directory. There are lots and lots of goodies in the various subdirectories—a rich variety of samples and useful programs, as well as the wonderfully robust Fox Foundation Classes (FFC). So it's entirely possible that you might quickly get sick and tired of trying to change back to this directory by navigating through the various Open dialogs in Windows. If you're in VFP and you've changed VFP's default directory location, you can switch back to VFP's home on C with the HOME() function. For example, typing:

```
cd home()
```

in the Command window will perform the equivalent of:

```
cd "C:\Program Files\Microsoft Visual Studio\VFP98"
```

or even:

```
set default to "C:\Program Files\Microsoft Visual Studio\VFP98"
```

Remember to enclose these fully qualified paths in quotes, because they contain spaces.

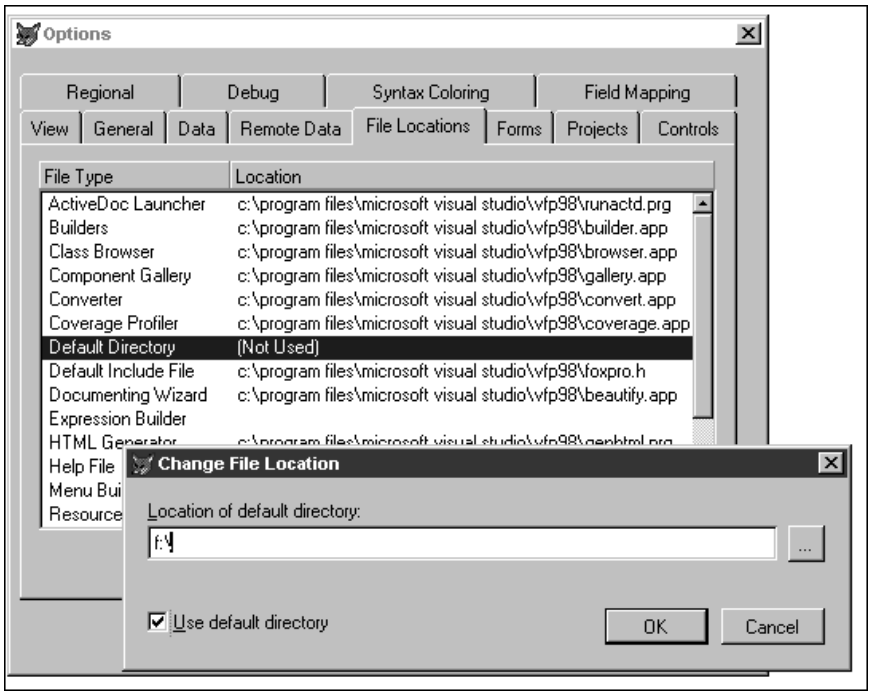


***Figure 16.1**. To change Visual FoxPro's default directory, check the Use default directory check box and then enter or choose a directory.*

Startup files

When Visual FoxPro starts, it looks in a variety of places for information on how to set itself up. About a thousand places, I think. You just saw one attribute of VFP's setup data—the location of the default directory. The files of interest are the Windows Registry, CONFIG.FPW, FoxUser.DBF, and FoxPro.INI. Here's the deal on each of them.

The Windows Registry

In the olden days of Windows 3.1, configuration data was stored in text files that had the extension of INI (for "initialization"). There were three problems with these files: there were lots of them, they could be edited by civilians, and it was difficult and/or impossible to manage them remotely. Thus, Microsoft came out with a binary data store called The Registry. It provides a single location for all Windows-related configuration data, and requires a special tool, RegEdit, to get into it. It can also be accessed remotely, if you really know what you're doing. All settings in the Tools, Options dialog are stored in the Windows Registry.

CONFIG.FPW

This text file stores the settings of a number of Visual FoxPro SET commands as well as several other options that can't be controlled from anywhere else. The default CONFIG.FPW file is located (where else?) in the Visual FoxPro home directory.

However, you can tell VFP to use a CONFIG.FPW file located elsewhere. The answer, however, is not that straightforward. Because of the variety of environments that developers work in, and the types of applications Fox developers have built over the years, there are a considerable number of places where you can do so, and there is a sequence of steps that VFP follows to determine which CONFIG.FPW file to use.

First, you can use a command-line switch (in the Target property of a desktop shortcut) to specify a particular config file. In fact, you don't even have to name it CONFIG.FPW! See the following section on startup switches for an example or two.

Next, VFP checks the contents of the DOS environment variable FOXPROWCFG (remember DOS?) for the fully qualified path of a file. Again, you don't have to name the config file CONFIG.FPW. If neither of these two choices applies, Visual FoxPro checks the default directory and then the DOS path (if one has been established) for a file specifically named CONFIG.FPW.

You can determine the current config file in use with the SYS(2019) command. If no config file was used, this function returns an empty string.

When you ship an application with an .APP or .EXE file and the VFP runtime, the config file business requires an additional step. You can include your own config file in your app (you have to actually bind it into the app); doing so overrides all of the other steps. If you don't include a config file, VFP follows the normal route of searching for the config file.

FoxUser.DBF

There is a .DBF (and associated .FPT) called FoxUser that can be used to store preferences, such as the positions and sizes of all windows in your development environment and preferences for editing windows. Each time VFP closes down normally, the last settings of those preferences are saved to individual records in this .DBF. That's why, if you move your Command window around, it shows up in the same place when you fire VFP back up.

This .DBF is called the resource file, and you can turn it on or off in a couple of ways. First, you can issue the commands SET RESOURCE ON/OFF in the Command window. Like the config file, FOXUSER is normally stored in the VFP home directory but can be located in a different directory. Again, like config, there are several mechanisms to determine which resource file to use. You can use the Resource file entry in the File Locations tab of the Tools, Options dialog to store the location in the Windows Registry. You can also put an entry in your config file, like so:

```
RESOURCE = <name of file>
```

Doing so overrides the setting in the Registry. And last and least, you can SET RESOURCE TO <name of file> interactively.

DEFAULT.FKY

You can create keyboard macros to automate the execution of special tasks. These macros can be stored in a file of your own choosing, but Visual FoxPro will automatically load any keyboard macros found in a file called DEFAULT.FKY located in the VFP home directory.

FoxPro.INI

FoxPro for Windows and Visual FoxPro 3.0 both used a file called FoxPro.INI to store settings such as the various startup window positions. These settings are now stored in the Registry. This .INI file was located in the Windows directory. In my brief testing I couldn't tell if it still works, but even if it did, you shouldn't use it. Try to keep everything in as few places as possible—which means the Registry in this case.

How to use the startup files

So now that you know what they are, how should you go about using them? First of all, you can't do much about the stuff that's stored in the Registry.

Remember my rule about keeping as little info on drive C as possible? Well, you can't move the registry off of C (and if you figured a way to do so, it would probably break Windows six ways from Sunday because of its tight coupling with the operating system). That's why I don't keep a config file, a resource file, or anything else on my C drive anymore. Nor do I use DOS environment variables, because by definition they're "part" of C as well.

Instead, I make a few small changes to VFP's File Locations in Tools, Options, and point to files on another drive. Then I configure the daylights out of those files. If I have to rebuild a machine (or a new version of Fox comes along, possibly ignoring or wiping out all of the files in my previous VFP home directory), I just point to the files on the other drive, and I'm all set.

I keep all of these files (as well as a host of others) in a directory called "DeveloperUtilities" on my main VFP development drive. Actually, there are several of these directories, one for each version of Fox that I'm using, because utilities for one version often won't work in another.

Here are the settings I change: Default Directory, Menu Builder, Resource File, Search Path, and Startup Program.

Default Directory

I've already addressed this, but to be complete I wanted to make it clear that I change the default directory here, not in the Properties dialog of the shortcut.

Menu Builder

This points to Andrew Ross MacNeill's GENMENUX program that I discussed in Chapter 7, "Building Menus." This program is stored in my Developer Utilities directory.

Resource File

I keep my FoxUser file in my Developer Utilities directory as well. Remember that FoxUser consists of both .DBF and .FPT files.

Search Path

The VFP search path is much like the old DOS search path—it's where VFP will look for files if they're not in the current directory. This includes programs, forms, and even databases and tables! This is particularly handy because, after all, what good are a bunch of cool utilities in your Developer Utilities directory if you can't get to them easily? I keep my Developer Utilities directory in my search path at all times.

Startup Program

Here's where all the real work happens. You can have Visual FoxPro automatically run a VFP program when it starts. That's what the Welcome to Visual FoxPro splash screen really is—just a VFP program.

But before I get into the details of what your startup program might do, I should mention that, as with all things FoxPro, there are a bunch of ways to specify the startup program. This entry in the Tools, Options dialog is one, of course, but you can also make one of two entries in your config file:

```
_STARTUP = <program name>
COMMAND= <command>
```

For example:

```
command = do \MYPROG3.PRG
_startup = "\MYPROG2.PRG"
```

Note that you need to specify a complete command with COMMAND—not just the name of a program to execute. If you specify programs in both, both will run: the program attached to _STARTUP first, and then the one attached to COMMAND. The program assigned to _STARTUP in your config file updates the value you enter in the Tools, Options dialog.

I personally just use the entry in Tools, Options, and point to my startup program in my Developer Utilities directory like the rest of the settings I've mentioned in this section. So what could you do in a startup program?

I use mine to launch a couple of applications that I use all the time, add a couple of menu pads to the standard VFP menu, open a couple of windows, and then place focus on the Command window when I'm all done. Here's the code:

```
* gofoxgo.prg
* start with a good menu
set sysmenu to default
* open the class browser
do (_browser)
* (other programs you want to load can go here too)
* run my custom menu
do GOFOXGO.MPR
* change the title of the VFP window to show what version
* of VFP I'm running as well as the current drive and directory
modify window screen title "We're developing in " ;
   + version() + "  (" + sys(5) + sys(2003) + ")"
* open up the Data Session window
set
* place focus back to the Command window
```

```
keyboard "{CTRL+F2}"
* get rid of anything that was left laying around on the VFP desktop
clear
```

My tech editor has made a number of other suggestions as well. First, SET ASSERTS ON should be the default during development. This is needed because there isn't a Tools, Options entry for this setting. SYS(3050, 1, 24000000) sets the maximum amount of RAM that VFP can use to 24 MB. Otherwise VFP will try to chew up all the memory on the machine, which can then cause problems if you try to open many other apps at the same time. And SET SYSMENU SAVE will save all these settings so that issuing a SET SYSMENU TO DEFAULT command won't blow it all away.

As you can see, this code is located in a program named GOFOXGO, and this .PRG is located, of course, in the Developer Utilities directory. The GOFOXGO menu, referenced about halfway through the program, actually adds two menu pads to the VFP system menu instead of replacing it. The pads are shown in **Figure 16.2**.

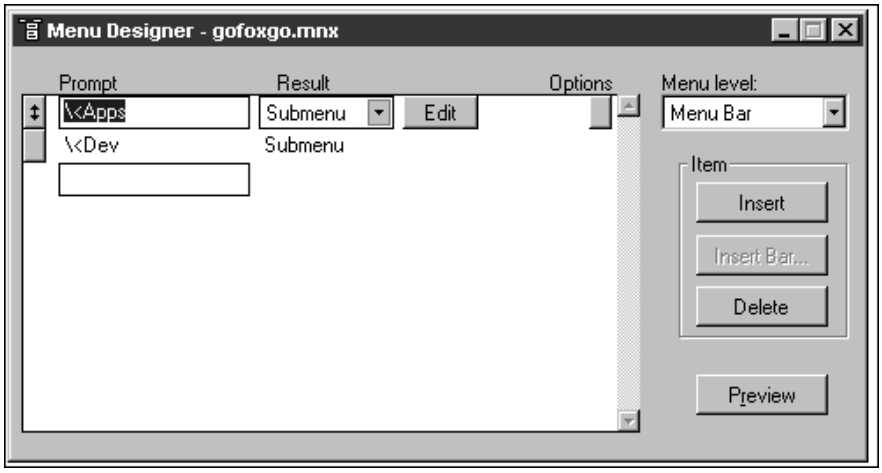


***Figure 16.2**. The GOFOXGO menu adds an "Apps" and a "Dev" menu to the VFP system menu.*

The Apps menu pad displays a list of applications I'm currently working on. Selecting any of the applications changes the current directory to the directory that contains the source code for that project. The Dev menu pad displays a variety of developer utilities and helpful shortcuts.

To add these two pads to the existing menu, select the View, General Options menu command, and select the Append option button, as shown in **Figure 16.3**. If you don't, you'll end up with a menu that has only two menu pads, and the rest of the Visual FoxPro menu will have disappeared.

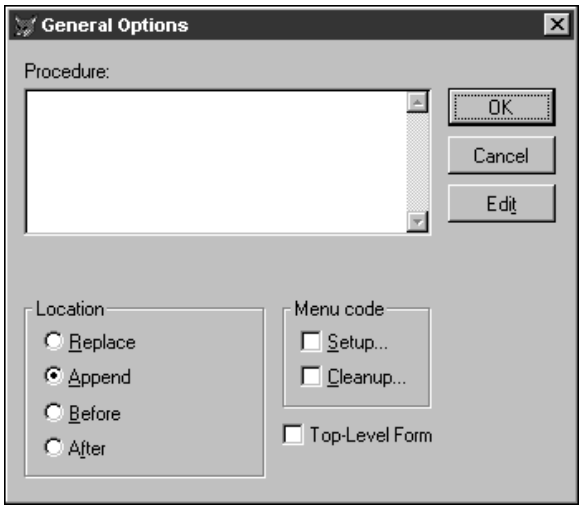


***Figure 16.3**. Be sure to select the Append option button to include additional menu pads on the standard VFP menu.*

In order to keep the code for the menu as stable as possible, each menu command runs the same program, passing the name of the menu bar as a parameter:

```
do l_gofox in GOFOXGO with prompt()
```

The L_GOFOX routine, contained as a procedure in GOFOXGO.PRG, looks like this, in part:

```
*************************************************************************
function l_gofox
*************************************************************************
lparameter m.tcNaApp
m.cPathPrefix = "f:"
m.cPath60LAN ;
   = m.cPathPrefix + "\devuti60; " + m.cPathPrefix + "\common60LAN"
m.cPath60CS ;
   = m.cPathPrefix + "\devuti60; " + m.cPathPrefix + "\common60CS"

do case
 case upper(m.tcNaApp) = "BC"
  set path to &cPath60LAN
  set default to m.cPathPrefix + "\bc\t12\source"
 case upper(m.tcNaCust) = "WEAC"
  set path to &cPath60CS
  set default to m.cPathPrefix + "\weac\mid2\source
```

```
 case upper(m.tcNaCust) = "WHERE AM I?"
  modify window screen title "We're developing in " ;
    + version() + "  (" + sys(5) + sys(2003) + ")"
 case upper(m.tcNaCust) = "DUPLICATE CURRENT RECORD"
  scatter memo memvar
  append blank
  gather memo memvar
 case upper(m.tcNaCust) = "DEVHELP"
  do DEVHELP
 case upper(m.tcNaCust) = "SWITCH TO DEVUTILS DIRECTORY"
  set path to m.cPathPrefix + "\devuti60; " + m.cPathPrefix + ""\common60"
  set default to m.cPathPrefix + "\devuti60"
 other
  wait wind "You goofball! Better clean up the menu!"
 endcase

* now change the window title to reflect our current location
modify window screen title "We're developing in " ;
  + version() + "  (" + sys(5) + sys(2003) + ")"
clear
clear program
return .t.
```

Because I might be using different class libraries for different projects, the first thing that the L_GOFOX function does is create extended paths for the locations of the class libraries. COMMON60LAN contains one set of class libraries and is used for one group of applications, while COMMON60CS contains a different set of class libraries and is used, obviously, for a different set of apps.

Then the program processes each menu option passed to it in a large CASE statement. Most of these CASE segments will just set the path and change the default directory, but a few will perform different kinds of tasks. For example, under the Dev menu pad, I have a command called “Duplicate current record”—and all it does is scatter memory variables from the current record, add a new record, and then gather the recently scattered memvars into the new record. Simple, but it sure saves a lot of time. Another menu command under the Dev menu pad automatically switches to my Developer Utilities directory.

After any of these routines is run, I’ll update the window title. It sounds silly, but I’ve found it really useful to see the name of the current directory in the title at all times. As soon as the phone rings, my concentration is shot.

A word about the help file

Another file location that many people like to alter is the Help file. If you’ve got the *Hacker’s Guide to VFP* (which you should—it can be purchased online at www.hentzenwerke.com), you can copy the HACKFOX.CHM file to, say, the Developer Utilities directory on your machine, and then point to it instead of to FoxHelp. I actually recommend not doing so, however. Instead, point to MSDN help in the Help File location in Tools, Options, and then either place shortcuts on your desktop/QuickLaunch toolbar, or add menu options to the native Fox menu to point to the Hacker’s Guide .CHM (as well as any other .CHM files you want to regularly access). The reason is that you’ll often find yourself needing information that is not Fox-specific. If you call up MSDN help inside Fox, you can access anything—not just Fox.

If you've gone ahead and changed the Help File location and now want to change it back, you might get a little frustrated trying to figure out where to change it back to. First of all, help files are no longer stored in the Visual FoxPro home directory; instead, everything is stored in the MSDN directory. Typically, that's under the Program Files\Microsoft Visual Studio directory (although it may vary if you've installed MSDN somewhere else). Second, "help" actually consists of several hundred .CHM (and related) files deep inside the MSDN directory. Which file is the "main" file? It's not even a .CHM file—it's a custom viewer that Microsoft uses to provide additional functionality, like the menu at the top of the MSDN help window. The name of the file is MSDNVS6A.COL, and it's (as of this writing) located with the rest of the .CHM files—in the subdirectory MSDN98\98VSA\1033.

You might be thinking that you could create a shortcut to the MSDNVS6A.COL file, and keep VFP pointed to HACKFOX.CHM, but, unfortunately, shortcuts to MSDNVS6A.COL generate an error.

Startup switches

You can include "command-line switches"—additional characters after the name of the program in a shortcut's Properties dialog—to control how Visual FoxPro will load.

Use the –A command-line switch to start FoxPro without reading the current CONFIG.FPW and Registry switches. This is handy if one of those files (particularly the Registry!) has become corrupt, and if you can't load VFP any other way. Example:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE -A
```

Use –R to re-establish the associations that Visual FoxPro has with file extensions in Explorer and File Manager. This is a lot easier than having to edit each file association manually:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE -R
```

Use the –C switch to control which config file VFP will use during startup:

```
C:\Program Files\Microsoft Visual Studio\VFP98\VFP6.EXE –CF:\MYCONFIG.FPW
```

If you find yourself using any of these switches frequently, you might consider creating multiple VFP startup icons on the desktop, each with its own VFP configuration and startup switches.

Developer vs. user requirements

Your needs as a developer and the needs of your users while they are using your application are considerably different. You'll often find yourself needing to perform functions for testing and debugging—while your application is running—that you don't want your users to have access to. For this reason, it's a good idea to have a separate menu pad in your application that provides access to a variety of "developer-only" commands.

These developer-only commands might include those shown in **Figure 16.4**.

***Figure 16.4**. A typical developer-only drop-down menu.*

The first five commands allow the developer to open any of the debugging windows. The next three are windows I find handy—the View window (now known as the Data Session window) is particularly useful while an app is running because it allows you to investigate what's happening in each of the open data sessions. The next batch simply lets you halt and continue execution of your program. And the last one opens up the Event Log—the file that all VFP errors are saved to.

Of course, what you put on your Developer menu is limited only by your imagination. For example, my tech editor also adds these menu options:

- Object under Mouse (F12). Performs "o=sys(1270)". This creates an object reference, called "o", to whatever is under the mouse when F12 is pressed. You can bring up the Watch window, enter "o", and then view or edit properties of the object through the Watch window.
- Release O—Performs "release o". This just releases the reference created in Object under Mouse (F12).
- Toggle Application Timer—Performs "oApp.oAppTimer.Enabled = !oApp.oAppTimer.Enabled". Doug uses an application-level timer to refresh the application toolbar, but it's a nuisance when tracing code, so this allows the timer to be turned off and back on again during tracing.

I also add an extra menu command to the bottom of the Forms menu. Users see only one "Exit to Windows" command, while developers can choose "Exit to VFP" as well. In Chapter 15 I also described a text box class that displays on a form only when a developer is using it.

Furthermore, you will need the ability to run your application in a couple of modes—not only as a developer, but also as a user, to mimic exactly what users will see. Not only will they

get different menu options, but their screens will look different (because they're not seeing any debugging information you've included), and you might even want to run against a different data set.

It is often necessary to be able to run the application you are creating as if you were a user. Unfortunately, it's not very convenient to build a brand new executable and copy that file onto a machine that has only the VFP runtime on it.

And it's not unlikely that you'll also be called to a customer's site and asked to debug a running application—it's nice to be able to "set developer on" while you're running the application but not let your users have that same functionality.

Finally, I've found it useful to provide secret "back doors" into functions for specific individuals, such as myself, regardless of who has logged on to the application.

The requirements, then, are such:

- Ability to run as a developer or a customer
- Ability to simulate the VFP development environment or the customer's environment
- Ability to run as a specific user, regardless of logon capabilities.

I've found the easiest way to do this is to create three files in the root directory of the drive that the application is running on, and then do a quick test for those files in the startup program of the application. The existence of those files sets application object properties that the rest of the application can read to determine behavior. For example, here's a code segment that you could use at the beginning of your startup program:

```
lparameters m.gcMethod, m.gcLocation, m.gcByWho
do case
case pcount() < 1
 m.gcByWho    = iif( file("\fpdev.whil"), "WHIL", "NONE")
 m.gcLocation = iif( file("\fplocation.hw"), "HW", "NONE")
 m.gcMethod   = iif( file("\fpmethod.dev"), "DEV", "NONE")
case pcount() < 2
 m.gcByWho    = iif( file("\fpdev.whil"), "WHIL", "NONE")
 m.gcLocation = iif( file("\fplocation.hw"), "HW", "NONE")
case pcount() < 3
 m.gcByWho    = iif( file("\fpdev.whil"), "WHIL", "NONE")
endcase
```

You'll notice that this code allows the person running the program to specify these values in one of two ways: either by providing fields named a certain way in the current drive's root, or by passing parameters to the main program. In either case, default values (which mimic a customer's environment) are provided in case no parameters were passed or no files existed.

Each of these memory variables—m.gcMethod, m.gcLocation, and m.gcByWho—is converted into an application object property with the following code as soon as the application object is instantiated:

```
oApp.cMethod = m.gcMethod
oApp.cLocation = m.gcLocation
oApp.cByWho = m.gcByWho
release m.gcMethod, m.gcLocation, m.gcByWho
```

The Method parameter can be either “DEV” or “CUST”. If the parameter is “DEV”, the Developer menu pad appears, the “Exit to VFP” menu command appears, and all developer debugging controls appear.

The Location parameter can be either “HW” or “CUST”. This value is used to configure the environment’s data directories and other environmental options. The purpose is to allow the developer to simulate, as close as possible, the customer’s environment. I’ve not run across a situation where I’ve set this option to “HW” at a customer’s site.

The ByWho parameter is generally used for special cases. Typically, the developer would include a special branch of code that tested for this variable being equal to a value, like so:

```
if oApp.cByWho = "FPDEV.HERMAN"
 <code that only Herman wants to run>
endif
```

The nice thing about this one is that it’s easy to expand its functionality to multiple developers, simply by using additional extensions for the FPDEV file name.

Finally, I should note that the contents of these files, FPMETHOD, FPLOCATION, and FPDEV, are irrelevant—they merely represent the existence of the file itself. This means you can quickly and easily create these files anywhere, simply by using a text editor or copying another file and renaming it.

Configuring development application directory structures

A little bit of history, or “how we got here”

When I started developing database applications, “state of the art” meant you had a second 360K disk drive and 256K of RAM. As a result, you were forced to keep your programs and data in the same location. DOS didn’t even know about directories, much less hard disks! Of course, the data sets being handled at this time weren’t that big—and a key component of any specification was the delineation of the record size and expected number of records.

Human beings being subject to inertia like anything else, when that monster 10 MB hard disk came along, developers generally didn’t see any reason to change the tried-and-true method of placing data and source code in the same directory. In fact, there was a good reason for not doing so: a lot of code would have to be rewritten to handle this situation, and who was going to pay for that? Pretty hard to explain to a customer that they were going to have to shell out extra bucks for no perceivable increase in functionality.

Suddenly—overnight, it seemed—applications were running on 80386 processor-based computers and, with the speed capabilities of FoxPro, data sets of 25, 50, 100 MB or more of data were within the grasp of the developer!

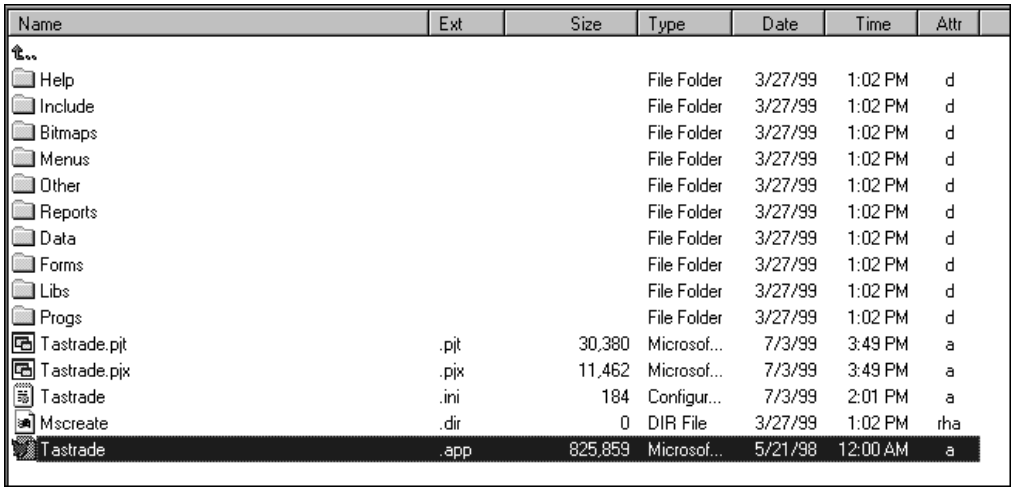
This situation started to present a couple of problems. First was the issue of accidentally overwriting customer data when you delivered an update to a system. Sure, you could be real careful when updating the files, taking care not to stomp on any .DBFs. But you only had to goof up once. Or you could just take the files that had been changed since the last update. But that meant keeping track of update dates, and even then, it was possible to have dozens or hundreds of files that had changed in a large system. And if you were continually bringing out

incremental changes, how did you deal with the issue of discarded and outdated files? You didn't dare erase all the files, so when REPORT5.PRG was replaced by REPORT5A.PRG, you just brought REPORT5A.PRG along and left REPORT5.PRG to gather dust.

As I'm writing this book, my company inherited an application that has more than 600 files in one directory. (And we found out that they had just moved this application from the root directory into its own subdirectory not too long ago.) The first job is to clean out the junk, and it looks like we'll get down to fewer than 100 files by the time we're done. Imagine the disk space savings, not to mention the boost in productivity when doing additional maintenance on the system!

With the advent of FoxPro 2.0, the number of source code files required for an application nearly doubled. Instead of having a single .PRG (and its compiled .FXP) for a screen, two additional files were needed—the .SCX and the .SCT—and then two more: the generated screen .SPR (and its compiled .SPX). Menus presented the same situation with the .MNX/.MNT/.MPR/.MPX combination, and each report generated through the report writer required two files—an .FRX and an .FRT. Now throw in a few dozen .DBF, .CDX, and .FPT files, and suddenly even simple applications were back to having hundreds of files in one directory.

Fox Software suggested a solution that's been carried forward by Microsoft: Use separate directories for each type of source code, so that your directory structure for Visual FoxPro looks like **Figure 16.5**.



Name	Ext	Size	Type	Date	Time	Attr
..						
Help			File Folder	3/27/99	1:02 PM	d
Include			File Folder	3/27/99	1:02 PM	d
Bitmaps			File Folder	3/27/99	1:02 PM	d
Menus			File Folder	3/27/99	1:02 PM	d
Other			File Folder	3/27/99	1:02 PM	d
Reports			File Folder	3/27/99	1:02 PM	d
Data			File Folder	3/27/99	1:02 PM	d
Forms			File Folder	3/27/99	1:02 PM	d
Libs			File Folder	3/27/99	1:02 PM	d
Progs			File Folder	3/27/99	1:02 PM	d
Tastrade.pjt	.pjt	30,380	Microsof...	7/3/99	3:49 PM	a
Tastrade.pjx	.pjx	11,462	Microsof...	7/3/99	3:49 PM	a
Tastrade	.ini	184	Configur...	7/3/99	2:01 PM	a
Mscreate	.dir	0	DIR File	3/27/99	1:02 PM	rha
Tastrade	.app	825,859	Microsof...	5/21/98	12:00 AM	a

***Figure 16.5**. The standard Visual FoxPro directory structure contains nearly a dozen subdirectories.*

Well, I have some problems with this layout. First, I can never keep the names of all of these directories straight. I valiantly struggled with "Why did they spell out REPORTS but abbreviate PROGS?" "Did I call it PROGRAMS or PROGS?" and "Was that in REPS or REPORTS?" And looking through previous versions of FoxPro as well as the default

installations of various third-party products, it appears that I wasn't the only person who couldn't be consistent.

Second, I found it ridiculous to have an entire directory for a total of four menu files, or two INCLUDE files, a dozen bitmaps, and so on. For every application you create, it's just that much more work to create and maintain those directories.

Third, and this is more of a real problem than the previous ones, the FORMS/MENUS/PROGS/REPORTS/etc. structure doesn't make any provision for handling libraries and other files used across multiple applications, nor does it provide for a mechanism to deal with data dictionary extensions or other third-party products that require hooks into the app. And it's positively hostile in terms of having to move an app into production and then shuttle changes from the development environment into the production arena.

Finally, you have to either build your application to an .APP or an .EXE in order to run it—which is a real problem if you're making incremental tweaks (remember that attention span issue)—or include all of the subdirectories in the VFP path in order to run from the source code.

Here's what I've come up with as an alternative.

Developer environment requirements

All but the most trivial of applications will require a number of file types.

The first, pretty obviously, is source code. These are the custom programs, forms, menus, and other files that we have generated and that make up the application. The key distinguishing characteristic is that these files are custom built for this application and they cannot be modified by the user. Note that reports may not fall into this group!

The next group of files make up the common libraries that support all of your applications. In previous versions of FoxPro, these were just Procedure files, but with Visual FoxPro, they include your generic class libraries. Their distinguishing feature is that they are files you created and they're used across applications.

The third group of files include third-party libraries and tools. Smart developers will not try to invent everything themselves, but rather will rely on third-party tools and utilities whenever possible. A number of tools and utilities provide incredible leverage for minimal cost. It's one of the seven wonders of the modern world that a developer would rather spend a couple of months trying to write a barely functional ad-hoc reporting tool when there are several powerful commercial products that provide orders of magnitude more functionality for the cost of a morning of programming time.

Where to keep these third-party tools? I've found it productive to keep them in their own directory instead of throwing them in with the common code. The primary reason is that I tend to update my common code fairly often, but I only add to the third-party libraries every couple months or so.

While I'm on the topic of tools, there's another set of tools that don't become part of an application: "Developer tools" include all those homegrown utilities you've got that make your life easier, as well as commercial products that aid in development. I keep these in the Developer Utilities directory that I mentioned before. I'll cover a few of these in the next section of this chapter.

Next are the data files. These are the tables that hold the user's information. The distinguishing characteristic of these files is that the user has complete control over what goes

into the tables (with, of course, the exception of the key and audit fields). Once a data set has been delivered to the customer, the developer doesn't touch it again unless the structure of the table has changed. (And even then, you should probably provide a utility for automatic update instead of having to depend on human frailty and do it manually.) However, don't think of data as one set of files.

You need to keep multiple sets of data in separate directories so that you can jump from one data set to another. As you get involved in applications that are "big-time," you're not going to want your users practicing on live data. Instead, give them the ability to move between data sets—one set for training and another for actual use. This same mechanism can then be used to run the same application for multiple businesses, and for you, to switch between test and production data sets.

Finally, there are more files that aren't as easily categorized. One type is data files that the user controls but that aren't specific to a single data set. Examples include a table of authorized users or a table that contains user-defined help. Another type is those data files that are created and updated indirectly. For example, many robust applications contain tables that track user logins, application errors, and, in some instances, an audit trail of every change to data. These are also application-specific as opposed to data-set-specific, but the user has no control over them, and, in many cases, isn't even aware that they exist.

A third type of files is the metadata for the application—the data dictionary files. Contrary to popular belief, the database (.DBC) does not make up the entire data dictionary, and you need additional files in your application. Of course, these are application-specific but not data-specific because all of the data sets had better be based on the same metadata!

This has suddenly gotten awfully complex. Perhaps you should just go back to throwing everything back into one directory and then buy a really fast disk controller so the machine doesn't choke on 1400 files in one directory. Well, maybe not...

But how should these files be grouped? To figure this out, I asked two questions: first, to what area of development do the files belong? As you've noticed, three sets of files are application independent, a couple of other sets of files are application-specific but data-independent, and still others are data-specific. Second, who updates the files—the developer or the customer? Because one of the primary problems is updating the production site with modifications, it makes sense to group the files so that they are divided into "updated by developer" and "updated by customer" categories. Here's how we'll do it.

Drive divisions

Readers of this book will have so much disparity in their machine configurations that it's basically a waste of time to suggest a single methodology of setting up the drive and directory structure and then expect everyone to follow it. However, I'll present one mechanism that can be used on a single machine—even a notebook—but can easily be applied and extended to deal with a network environment. The key point to understand is what I'm trying to accomplish; you can then adapt my techniques or use new ones to apply to your environment.

I have a pet peeve about looking at the root directory of a drive and seeing dozens or hundreds of directories. Not only do the miniature directory list boxes inside most Windows apps make a large number of directories inconvenient to see, but mentally, I can't keep track of more than 15 or 20 entities on the same level. I'd rather break them down into hierarchies.

As a result, I segregate software on a machine into at least three separate locations—either drives or directories depending on the hardware of the box:

- The first drive or directory contains the operating system and all shrink-wrap software. As I've mentioned, you should be prepared to reformat that drive often, so keep as little custom stuff on it as you can.
- The second drive or directory contains nothing but custom FoxPro applications and their support files (like third-party tools). If you develop on multiple platforms, you might still want to keep all of your custom development separated from all the other garbage that ends up on your hard disk.

  This drive, again, is fairly sparse in terms of the number of directories in the root. The only entries in the root are single directories for each customer, plus directories for application-independent files (common files, developer utilities, etc.) as discussed above. Depending on the size of your shop or of individual applications, you might want to split this across multiple drives on a network.
- Finally, the third drive or directory is used for "stuff that didn't fit anywhere else." This includes all internal data, home directories for each user on the network, general reference libraries (like online CD-ROMs), and so on.

Root directory contents

Now it's time to break down the custom apps directory into more detail. As mentioned, there's a directory for each customer, and each project for a customer has its own directory under the customer directory. Before I get into customer directories in any more detail, however, I have to finish up with everything else that's in the root.

The root directory contains three more directories that are application-independent. The first is COMMON60, which contains all common code and libraries that you've written for Visual FoxPro. (Actually, if you went snooping around my network, you'd also find COMMON10, COMMON20, COMMON25, COMMON26, COMMON30 and COMMON50 for the various versions of FoxPro.) The distinguishing characteristic of these files is that you have written them yourself, and they are used across multiple applications.

The second non-customer directory is THIRDPARTY60, which contains all third-party utilities, tools, and libraries that you would use in your application. Many times a third-party utility comes with dozens of files—source code as well as distribution files, sample files, documentation, and other goodies. This directory does not contain all of that—those full-blown installs are contained in their own directories on the Shrinkwrap drive under a directory named something like "FoxPro Utilities." Rather, this directory contains just the minimum files needed to run the tool.

And just like COMMON, you'll actually find directories named THIRD10, THIRD20, THIRD25, and so on, because typically the libraries don't run across multiple versions of FoxPro.

The third root-level directory is called DEVELOPER UTILITIES 60, and again, there are corresponding directories for earlier versions of FoxPro. This directory contains tools and utilities for internal use. Because these files are never taken off-site, I keep the internally

written tools as well as commercial products in the same place and use a naming convention to keep them straight.

Application directory contents

As described before, each customer has its own directory off of the root directory, and each project for a customer has its own directory underneath the customer's directory.

So what's in the directory for a project? Just a handful, really. See **Figure 16.6**.

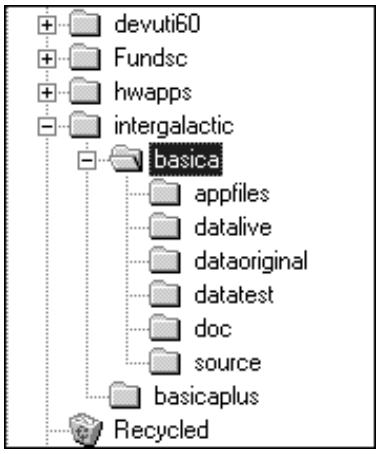


***Figure 16.6**. The Intergalactic Corporation has two applications: Books and Software Inventory Control Application, and Books and Software Inventory Control Application–Plus.*

At first glance, it might seem that I've just replaced the multiplicity of directories used by Microsoft's Visual FoxPro install with my own—equally voluminous, and clearly more confusing—set. However, a brief explanation will set all right again.

Doc

I know—it's a radical idea, but I keep all of the documentation for a project. Things like correspondence, specifications, data diagrams, and so on, all end up in this directory. Occasionally you'll have documentation that is specific to a customer, but not to any specific project—in those cases, I create a generic DOC directory on the same level as each project.

Source

SOURCE, obviously, contains source code. All of it. Into this directory goes the project, every custom menu, program, form, and class library file (classes that are specific to this project), as well as any other types of files (like bitmaps) for this application. I've already whined about my dislike for multiple source code directories, and you're welcome to create separate directories if you like; you can still use the rest of this structure with very little modification.

One question that has already come up has dealt with the number of files that end up in SOURCE in a big application, and the resultant drag on performance. Actually, however, there really isn't a drag. I've built applications with more than 1000 source code files on a 266 MHz PII and really haven't noticed much of a drag. Applications delivered to customers are compiled to a small set of .EXEs and .APPs, and they're placed above the SOURCE directory, so the effects at the customer's site (where they probably have lower-powered machines) aren't noticeable either.

Appfiles

APPFILES contains all application-specific files for the system. This consists of the data dictionary and any other files that are application-specific, whether or not the user will be able to modify them—either intentionally (such as a user-defined help file or the users table) or unintentionally (those files that the system will update behind the scenes, such as the logins table and the error-tracking table). The key attribute of each of these files is that they are data-independent—the same information applies regardless of which data set the user is working with. I used to keep metadata files in a separate directory, but Visual FoxPro's architecture with the .DBC has made that unnecessary.

Why? Note that the .DBC file for a dataset does not belong in the APPFILES directory. Due to the nature of how Visual FoxPro handles the database, the best design decision is to have a separate .DBC for each set of data. It's not that bad; because the .DBC stores the relative path of its tables (if they're all on the same drive), you could create a .DBC, add the tables, and then just copy the whole structure to another directory on the same level. When your application runs, it will switch to the appropriate data directory and open the database.

The only glitch is making sure that forms that have been tied to a specific data set will recognize the current data set, but I've already discussed the code in the BeforeOpenTables event of a form's Data Environment required to make this happen (Chapter 15, if you skipped over it.) It's probably good to mention a second time, though, that when you build your forms, the database you use initially will be bound to the form. For this reason, I always bind all forms to the database in the DataOriginal directory. (More on this when I cover the data directories.)

What types of files might you have in APPFILES?

- A_USER contains one record for each user of the system. The record contains the user's full name, login name, password (encrypted, of course), permission level, a flag for whether or not they're allowed to currently access the system, and fields for storing individual user preferences.
- A_PROC contains one record for each time a user logs in or out of the system. The record contains the user's login name, the date and time logged in, and the date and time logged out. This file can be used for three purposes. First, you can monitor access to the system—seeing who is using the system and when. This can be useful information both for security problems as well as to track down bugs that might be user-related. Second, if you suspect users are not exiting the application properly, simply turning their computers off ("Oh, no, I never turn my computer off without logging out first!"), you can track them down by looking for entries that have a login but no matching logout. And finally, you can do a quick query against this file to see

who might be logged in. Note that this is not foolproof, because a user who just shut their computer off will still appear to be logged in.

Note that a single record is used to record the matching login and logout. It's simple to write a routine that will locate the last record for the current user that doesn't have a logged-out date and time, and update those fields during their logout procedure.

- A_EVENT contains one record for each instance that the application's error handlers have been fired. The record contains the number and the description of the error, the line number (and line, if available) of source code that was executing at the time, the login name of the user, and the date and time the error occurred. The associated memo file contains a text file that lists the status of the application and environment at the time of the error.
- A_HELP is the compiled HTML Help file for the system. See Chapter 25 for information on building HTML help files.

Data directories

Finally, I keep at least two (and usually three) data directories with a project. I reverse the names ("DataOriginal" instead of "OriginalData") so that all of the data directories show up in proximity in Explorer. DataOriginal contains the dataset I use to build the application's forms and classes. In other words, when I create a form, I use DataOriginal as the source for the tables. DataOriginal actually contains no data, which serves an important purpose. I get to make sure that the application will run even without records in the tables—and that is often a handy test. It's so embarrassing to build and test an application, and then, as the last step, zap all of the tables and ship the system—only to have customers call with bug reports because the app can't handle empty tables.

Each data set contains the .DBC as well as a few specific files—IT.DBF contains the last used key value for each table in the data set as well as other values that are automatically incremented by the system, and ITLOOK is the standard lookup table. This table is not considered to be data-independent because its values might be specific to a particular data set: A user in the training data set might create a bunch of garbage values that shouldn't be accessible by the production data; or multiple businesses might have different values in their lookup tables.

Build directory contents

Developing apps is one thing—with Visual Studio 6, distribution is a lot more complicated. It's not enough to simply create an .EXE and ship it to your customer with a couple of runtime files. When you ship a VFP application, you'll need to build a complete set of distribution disks, complete with a standard Windows setup program.

I mentioned earlier that each customer directory contains separate directories for each project. However, because of this involved distribution process, you'll need a second set of project directories to hold your distribution files, as shown in **Figure 16.7**.

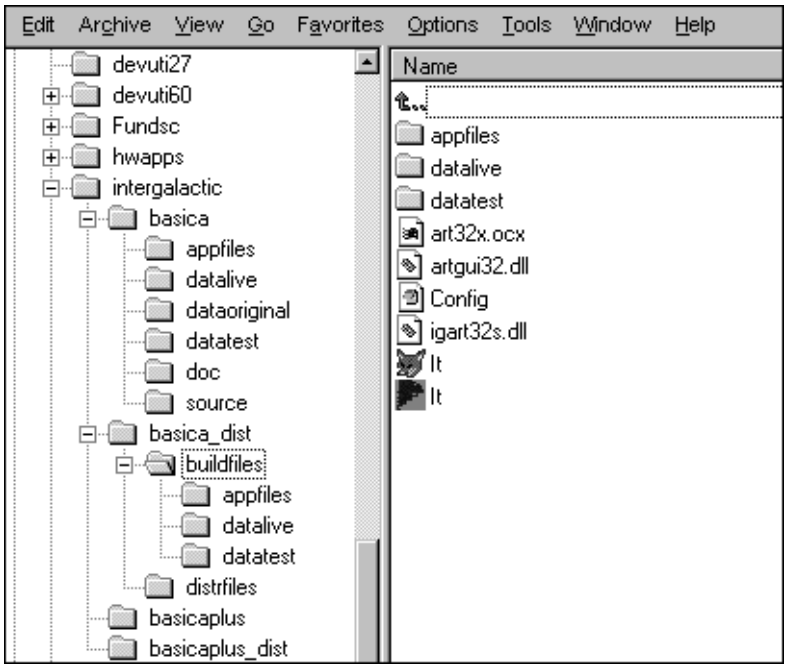


***Figure 16.7**. The distribution directory for the Books and Software Inventory Control Application project has subdirectories for storing files to build from and for storing the resulting distribution files.*

The first subdirectory under the project's distribution directory is BUILDFILES. It is used to store a copy of the application from which the distribution disks will be built. This directory will hold the application as the end user will see it; thus, you'll have an APPFILES directory, and a test and a live data directory (but not DataOriginal). You'll also want the .EXE you built for your application, an icon file, and, optionally, a config file. You don't need to include the VFP runtime—the Visual FoxPro setup wizard will automatically grab it from the Windows system directory.

Finally, you'll need a target for the setup wizard—where it will put the distribution files for your application. I use the DISTRFILES directory under the project's distribution directory as a handy place to keep them organized. If you like, you could even create multiple subdirectories under the DISTRFILES directory for each build you produce.

Developer utilities

I'm of two minds while writing this section. One part of me wants to share a bunch of cool utilities that I think you'll find useful. The other part of me knows you all too well—and I know you're thinking, "Building my own utilities is half the fun of being a developer!" So I'm going to take the middle road and list a few simple tools I've found handy, and then describe a few more that you might want to build yourself.

I've divided this section into two pieces: those that I wrote myself, and those that I found elsewhere.

My own stuff

Part of the thrill of being a developer is writing your own utilities in response to needs that come up during your day-to-day coding. Here are a couple of mine and Doug's. They're included with the source code files for this book at www.hentzenwerke.com.

WW

I'm forever using a Wait window or a message box to quickly display the value of some memory variable or contents of a field. The problem is that you need to convert the value to a string in order for it to display properly—and, if that's not hard enough to type, you don't always know what the data type of the value is. (By doing this in the first place, you already demonstrate that you don't know what the value is—it's likely that you don't even know what the data type of the value is, right?)

This little function—only two characters long—takes any type of data and displays both the type of data and the value itself in a Wait Window box in the center of the screen. You can pass a ",1" parameter to WW to display the value in the upper right corner of the screen, where Wait Window displays by default, or a ",2" parameter to display the value in the upper left corner:

```
m.lcX = "FJFJFJFJFJFJFJFJFJFJ"
? ww(m.lcx,2)
```

The parameters correspond to the quadrant numbers you learned about in high school geometry. See **Figure 16.8.**

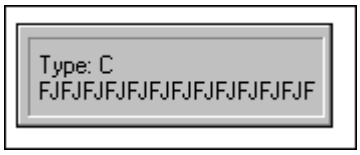


Figure 16.8*. "WW" will produce a Wait window that displays the data type and value of any variable or field name passed to it.*

Array Browser

You can use the Debug window to examine values of an array. But if there are more than a few values in a two-dimensional array, the display is hard to use because you just get a single long list of every array element. I wrote the Array Browser many moons ago in order to display an array in a visual row-and-column format.

You can call it three ways. If you simply call the routine, you'll be prompted for an array name:

```
=ab6()
```

If you pass the name of an array, a form will be opened with the array displayed in one of them-thar ole-fashioned browse winders:

```
=ab6("aMyArray")
```

DevHelp

Just about every developer I know has about 1500 Post-It Notes stuck around the edges of their monitor, reminding them how to do this and that. The problem with this system is that it's not multi-user. So I created a little table on one of the network drives that everyone could access—and in it, I started loading all sorts of tips and how-to's that had been stored on layers of yellow stickies. Then I wrote a little routine that opened this table from the Dev menu pad (which I discussed in the "Startup program" section earlier in this chapter).

OpenAll

Are you as lazy as me? For some odd reason, I find myself constantly opening a bunch of tables in a database. And it's a real pain:

```
Use DBF1 in 0
Use DBF2 in 0
Use DBF3 in 0
Use DBF4 in 0
Use DBF5 in 0
Use DBF6 in 0
… (all the way till)
use DBF322 in 0
```

So I wrote this little program that digs out the names of all the .DBF files in the current directory, and opens each one of them.

Z

You know when you're trying to track down a really nasty bug? And you've tried everything? What's the very last thing you do before quitting FoxPro, turning your machine off, and then starting back up again? You go through a series of commands that should shut down everything in sight:

```
close all
clear all
clear
on error
on escape
```

and so on. Well, in order to save a few keystrokes (and the last few strands of hair on my head), I wrote a program that does all this—and much more—to clean up the environment and set everything back to where VFP had it when it was last loaded. And because it's now the last thing I do, I named the program "Z". Then, after any kind of error during my programming or testing, I simply:

```
do z
```

and everything is set back to the way it should be.

HackCX

Once you get comfortable with the idea that forms and class libraries are simply tables—and, even better, that there's some rhyme and reason to how they're organized—it's just a short warp jump to realizing that you can open and modify them yourself.

For example, suppose you've got this complex data-entry form with a couple of page frames and dozens of other controls on it. And suddenly it hits you, when you're wondering why the three text boxes on the third tab aren't behaving like you were expecting, that those came from VFP's base classes, not from your own class library.

After you've recovered from the slap upside the head you gave yourself, you wonder how you're going to fix the error without going through every control, one by one, to determine whether it's from the proper class. Then it dawns on you—you can just open the .SCX and change the Class field from "textbox" to "hwtxt" and Class Location field from "" to "hwctrl62.vcx".

And *then*, after you've done this a few times, you realize that it's really clunky—having to open the form or class library as a table, then browse and resize the window, and then open a half-dozen or more memo fields to get at the data you want. Wouldn't it be better to have a nice user interface to do this?

HackCX is a simple form (and associated class library) that does just that: opens an .SCX or .VCX, allows you to scroll through every record, and allows you to see the contents of and make changes to an object's class, class location, base class, object name, and parent. See **Figure 16.9**. You can even see and modify the properties and methods. However, because the compiled code for those items is stored in the Objcode field, you'll want to make sure to use the Wipe Out Objcode button—in effect deleting the compiled code from the form or class library, and thus forcing VFP to recompile the next time the form or class is used.

***Figure 16.9**. HackCX allows you to view and edit multiple memo fields in an .SCX or .VCX.*

NIHBSM (Not Invented Here But Still Marvelous)

.CHM files

The first thing that should be in your Developer Utilities directory is the complete set of .CHM files from the books that I publish. (What? You mean you don't have all of them already? I'll wait right here while you go place your order for the rest … really, it's no trouble … go ahead, do it now, before you forget *again*!) My Dev menu pad has shortcuts to all of the .CHM files, as shown in **Figure 16.10**.

In the CASE statement in GOFOXGO, then, I make a parameterized call to a generic help function:

```
=l_help("HACKFOX")
```

The generic help function looks like this:

```
*************************************************************************
func l_Help
*************************************************************************
lparameters m.tcNameCHM
m.lcCurrentHelp = set("Help",1)
set help to (m.tcNameCHM)
help
if !empty(m.lcCurrentHelp)
 set help to (m.lcCurrentHelp)
endif
return .t.
```

This way, you don't blow away the existing Visual Studio help reference but you still have one-click access to any of the .CHM files.

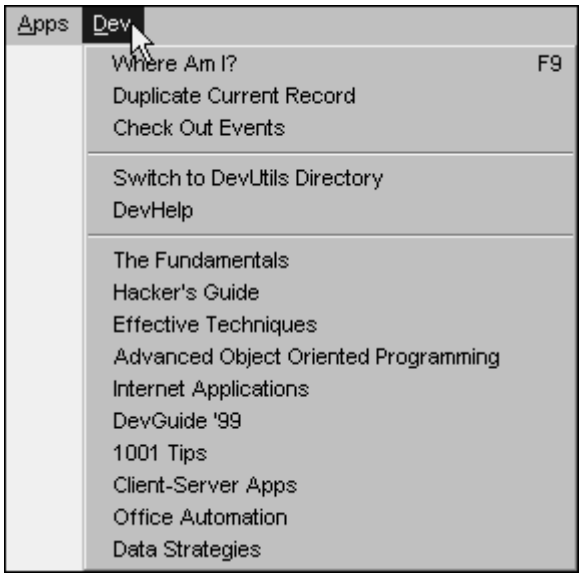


***Figure 16.10**. My Dev menu pad includes commands to launch all of the Essentials .CHM files.*

SoftServ's Command Window Emulator

SSCommand was designed to help in situations where you have a VFP application running (compiled) at a client site, and the client does not own Visual FoxPro. It emulates the VFP Command window (see **Figure 16.11**), allowing you to perform virtually any FoxPro command. It frees you as a developer from having to tote your notebook computer when you have to make on-site data maintenance or report modifications. It also makes life much easier if you use remote dial-in software for application maintenance.

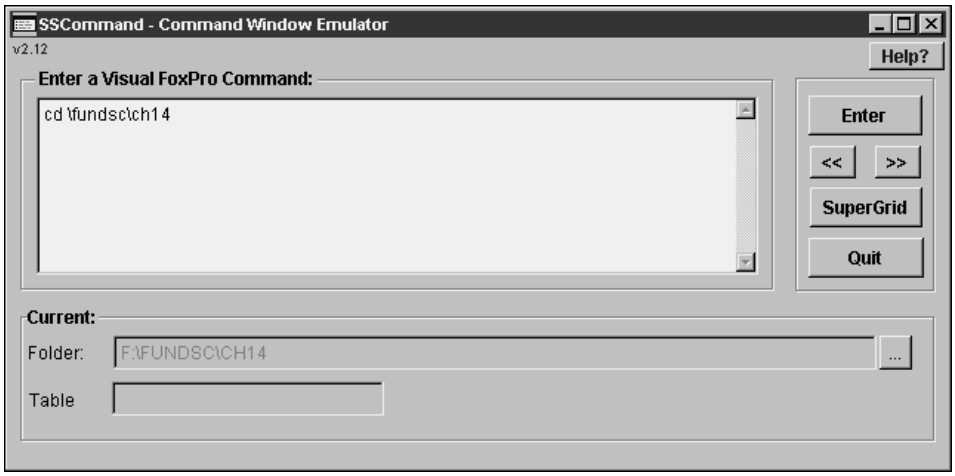


***Figure 16.11**. SSCommand from SoftServ.*

Like the Command window, the Command Window Emulator maintains a list of your previous commands. It also allows for multi-line commands using the semicolon for line continuance. You can browse a table, modify a table's structure, modify a report, run a report, and so on. The only thing you *can't* do is compile source code.

In addition to providing all the regular functionality of the Command window, they've created one dandy data-manipulation tool. It's called SuperGrid. See **Figure 16.12**. This is a browse-like grid that has many features, including "on-the-fly" column indexing, actual field name column headers (remember the old days?), column partial-value searching, column locate-value searching, compound index creation, and so on.

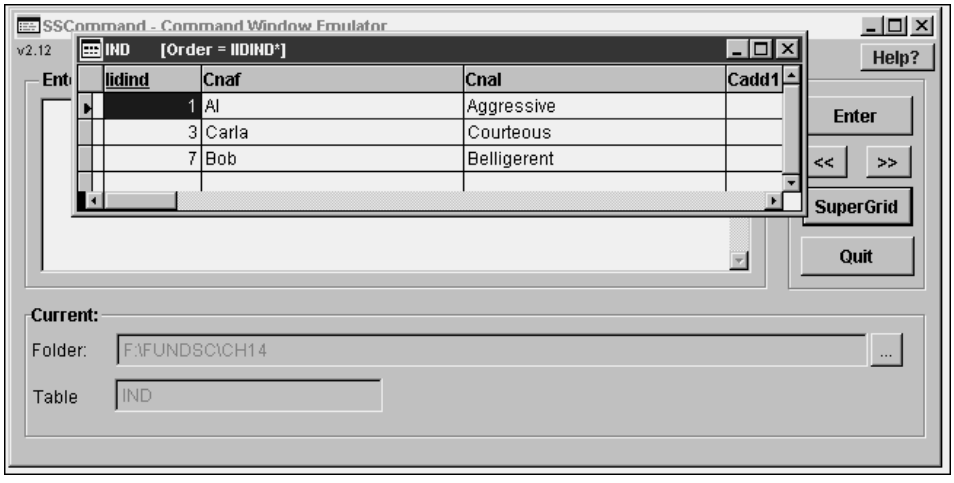


***Figure 16.12**. SuperGrid from SoftServ.*

Eraser

Okay, so this is silly, moronic, and makes no sense. But so is going to work on Fridays. I found it on one of the CompuServe forums a hundred years ago, and have always kept it around for idiot relief. See **Figure 16.13**.

When you run Eraser, it fills your VFP screen with happy faces, and gives you a timer and an "eraser" (a cursor about four characters wide). As soon as you start moving the eraser (to erase the happy faces, don'tcha know…), the timer starts running. The goal, of course, is to erase all of the happy faces in the shortest amount of time possible.

You'll want to close all of the windows in VFP first, and then run Eraser from the Program, Do menu command.

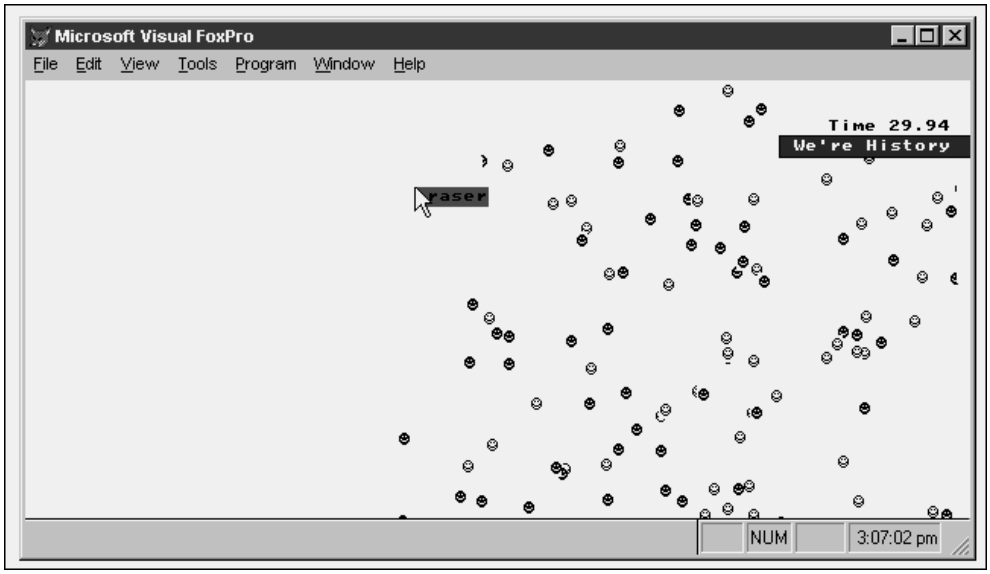


***Figure 16.13**. The Eraser game is just plain silly.*

Third-party applications

By now, hopefully you're all fired up and have a ton of ideas for your first Visual FoxPro application. It's tempting to find a quiet room, fill the refrigerator with Jolt cola, and lock yourself in until the app is finished. But you shouldn't do it alone. In fact, the ugly truth is that you can't do it alone anymore. Visual FoxPro is too big and too complex for most people to master in a short amount of time. And while you could spend months and years learning the product, chances are you need to get applications up and running now! (If not sooner.)

Well, there's no need to reinvent the wheel or to do everything yourself. There are a number of third-party products that provide various mechanisms and tools to help you develop applications, and I'm going to steer you toward 17 of the very best. In each case, the third party has invested thousands of hours of development into creating robust, feature-rich products that are used throughout the world. For a couple hundred bucks, there's no way you could match the

functionality and reliability that these products offer. I've known the principals of each of these firms for years—and they're all individuals of the highest caliber. They stand behind their work, and the rest of the industry would do well do emulate their business practices. And no, I'm not getting commissions or anything. (In fact, I think I owe a couple of them a beverage or two the next time I see them...) It's just that I think these are all great products and they deserve your consideration.

And by the way, just because I mention only these doesn't mean that everything else out there is junk. These are important tools that every new Visual FoxPro developer should be aware of. There are dozens more tailored for specific needs or certain functionality, but they aren't as universally applicable.

I've included a brief description of each (in most cases, lifted directly from the company's marketing literature) to get you started. Contact them at the sites below and they'll be happy to send you piles of literature and demo disks, and also to answer any questions you have.

CodeMine

CodeMine is a high-performance, object-oriented application framework for Visual FoxPro 5.0 and 6.0. CodeMine includes a comprehensive set of integrated class libraries and development environment extensions for the ultimate in Rapid Application Development. See **Figure 16.14**.

www.codemine.com

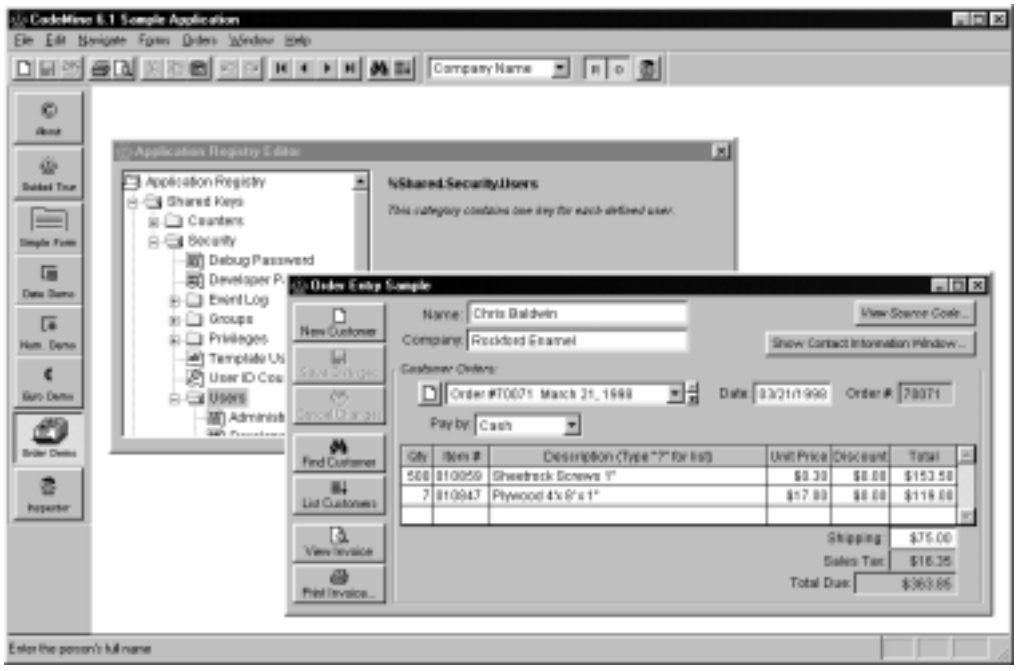


***Figure 16.14**. The sample application for CodeMine.*

DBI Technologies ActiveX controls

DBI has been at the forefront of producing ActiveX controls that work in Visual FoxPro. Many ActiveX control vendors concentrate their efforts on the Visual Basic market, and thus many controls have difficulty running properly in VFP. DBI is one of the few vendors that has focused on the VFP market, making sure their controls behave properly in both VB and VFP. See **Figure 16.15**.

www.dbi-tech.com

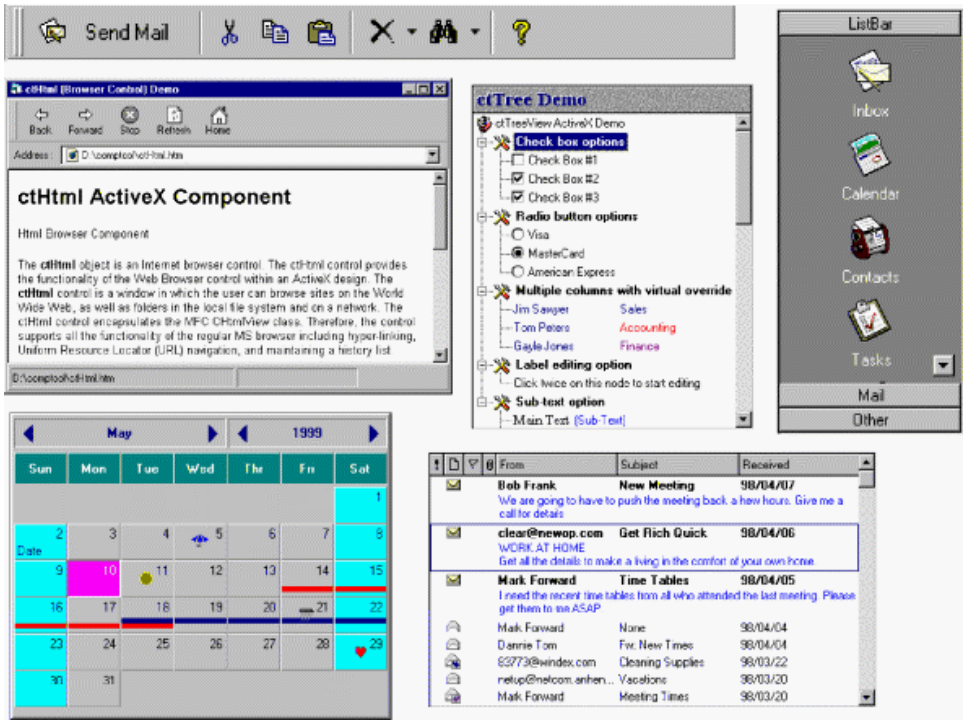


***Figure 16.15**. The Component Toolbar from DBI.*

FoxAudit

FoxAudit, from Take Note Consulting, allows developers to add complete, automatic, client-server–like, transaction logging and audit trail support to their applications. It is implemented as a Visual FoxPro class that captures information each time a user inserts, updates, or deletes records in an application. The details of each update are stored in a transaction log table. See **Figure 16.16**.

www.takenote.com

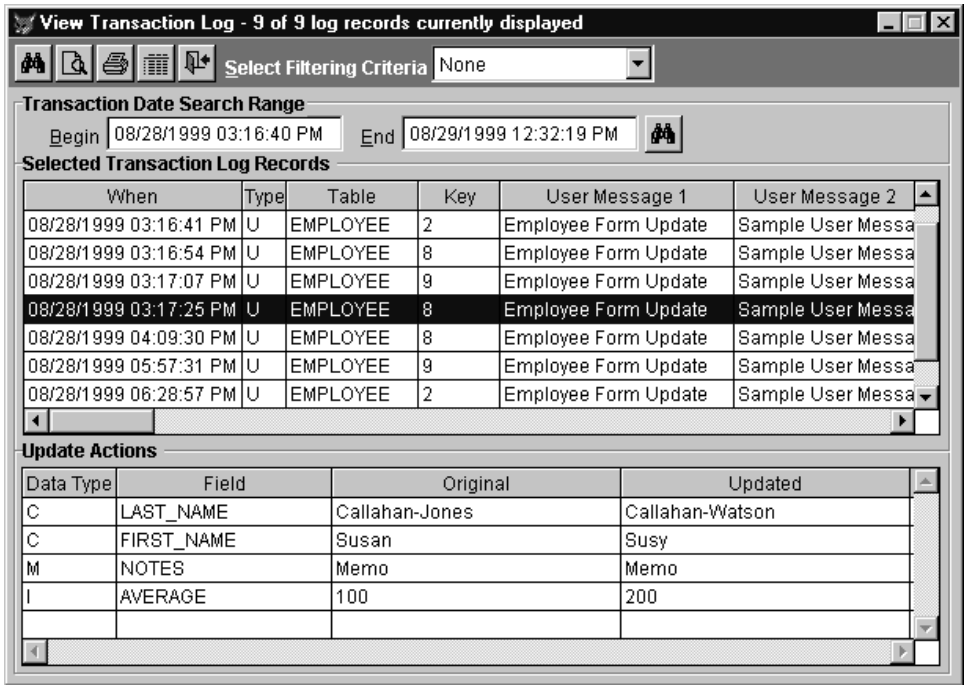


***Figure 16.16**. Viewing a transaction log using FoxAudit.*

Foxfire!

It's the rare application that doesn't require output, and it's even rarer when you find a user who doesn't want to change "one little thing" on the reports you've put together. Many developers decide they want to create a generic ad-hoc report writer that will allow users to create and store their own reports. A few months later, the developer emerges from his office, dark circles under his eyes, muttering, "If only I could get this one thing to work, I think I'd be done!" Foxfire!, from Micromega Systems, provides this functionality and more. See **Figure 16.17**.

www.micromegasystems.com

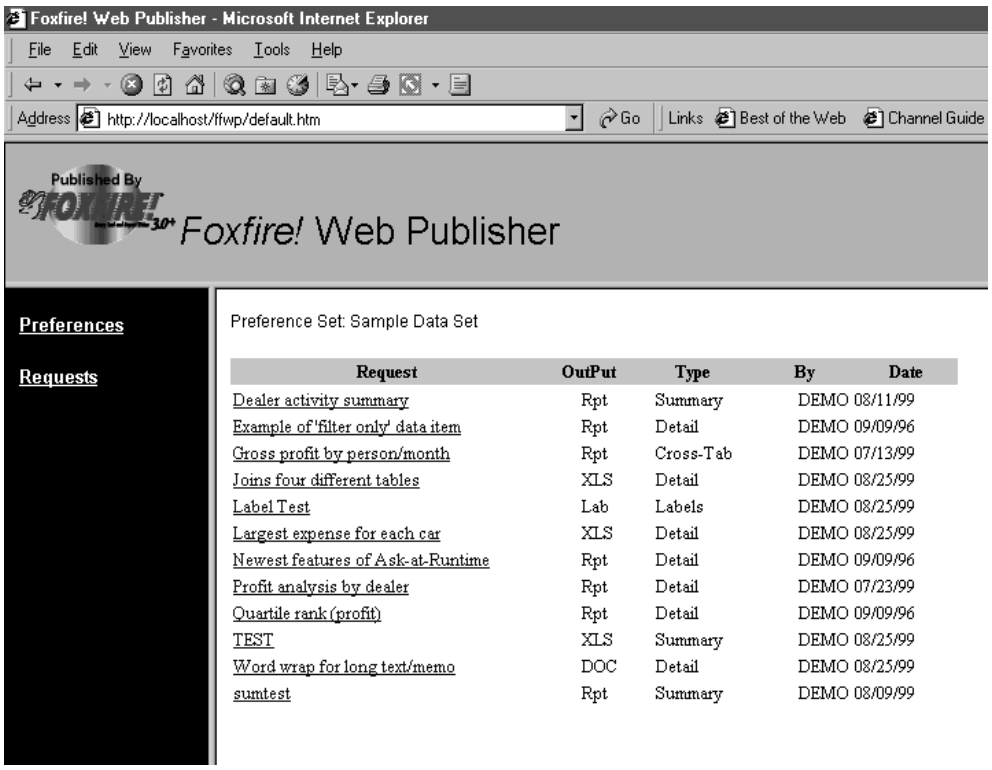


***Figure 16.17**. Foxfire! from Micromega Systems.*

HTML Help Builder

West Wind HTML Help Builder offers a new way to build developer help files. Most help file–generation software focuses heavily on the visual aspects of help generation, so you end up using a word processor like Word with macros to generate your content. That works well for end-user documentation—but what about developer documentation? Developer documentation tends to be much more structured and uses repetitive data entry that's often easier to accomplish via structured data input. With this structured approach it's easy to integrate the help file creation into the development process, both for creating end-user developer documentation as well as building internal documentation and references. See **Figure 16.18**.

Help Builder also focuses on laying out the structure of the help file. All while you're working with Help Builder you're creating a layout that matches the layout of the HTML Help file and you can see the project take shape as you work. All views are live so you can see any text you create immediately as HTML, and the project structure is at any point visible in real time. All topics inside the viewer are also live, so you can click and move around the help file just like you can in the compiled HTML Help file. Finally, when you're ready to build your

output, Help Builder allows you to create both stand-alone HTML documents or an HTML Help output file. This way you can use the output both for inclusion with your application as well as display the output on the Web without any special viewers.

www.west-wind.com

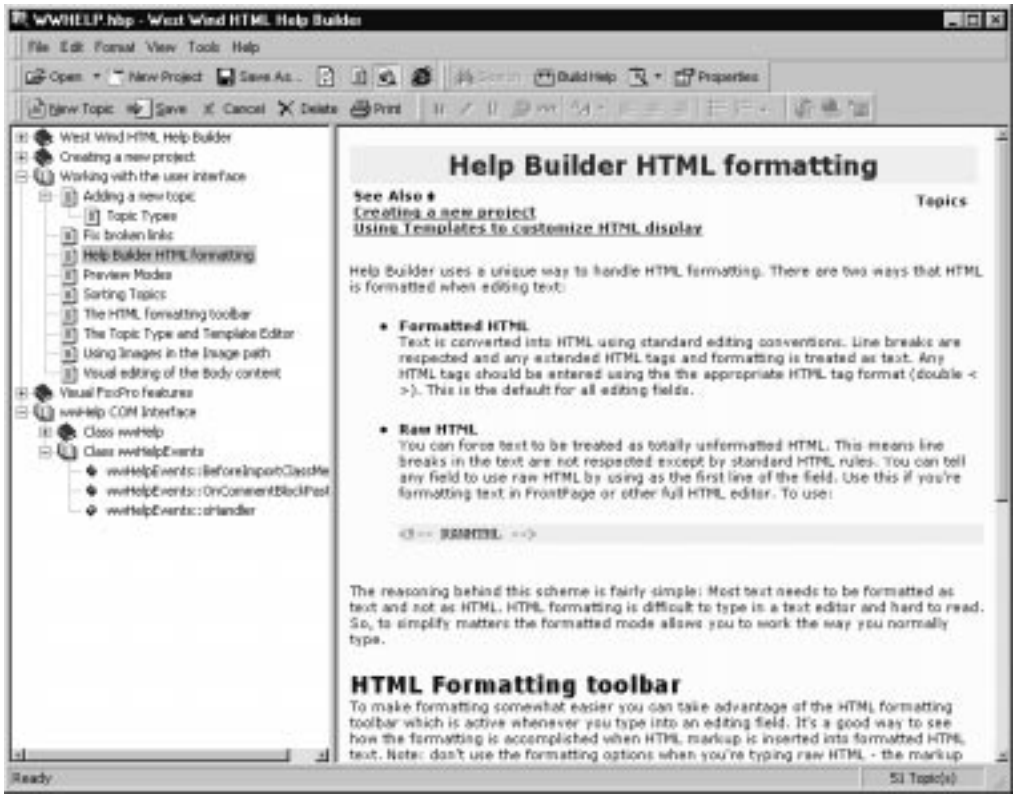


***Figure 16.18**. HTML Help Builder from West Wind Technologies.*

INTL

Steven Black's INTL Toolkit for Visual FoxPro 6 works either at run-time or at compile-time. Run-time localization gives you one executable for the world, with resources bound at run-time. INTL for VFP 6 has the following run-time capabilities:

- Interface strings
- Fonts
- Images
- Data sources

- Currencies
- Right-To-Left writing systems
- Dynamic dialogs

Compile-time localization, which is new to INTL for VFP, gives you one executable for each locale, with resources bound when you build your application. INTL for VFP 6 has the following compile-time capabilities:

- Interface strings
- Fonts
- Images
- Data sources
- Dynamic dialogs

If you do multi-lingual work, you must have this product!

www.stevenblack.com

Mail Manager

Mail Manager, by Classy Components, is a MAPI-compliant component that allows you to easily integrate e-mail into your applications. When coupled with QBF Builder (see below) or with a compliant result set, it will allow you to send an e-mail to a queried result set of recipients—for example, "Send a welcome e-mail to all new customers." See **Figure 16.19**.

www.classycomponents.com

Mere Mortals Framework

The Codebook methodology has been popular among developers for several years. Kevin McNeish has taken the complexity out of it and created an application framework that, as its name suggests, "mere mortals" can use. The Mere Mortals Framework, from Oak Leaf Enterprises, is a robust tool for rapidly developing flexible and adaptable applications. It helps you understand and use new technologies while providing a clear migration path that protects your investment in existing applications. See **Figure 16.20**.

With Mere Mortals you can create applications that anticipate changes in the user's requirements and the software industry. The Framework, along with extensive documentation, guides and educates you throughout the development process of creating true three-tier, client-server applications that are easily adapted to run over the Internet. Mere Mortals also takes advantage of the Visual FoxPro Component Gallery by integrating and enhancing the Foundation Classes.

www.oakleafsd.com

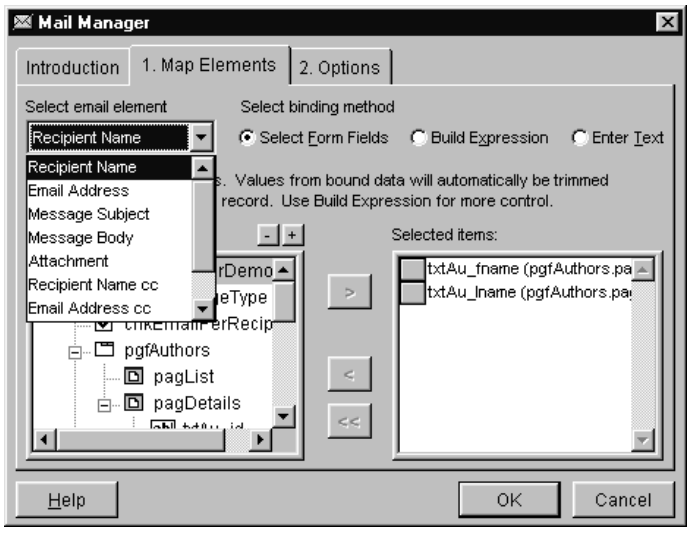


***Figure 16.19**. The Mail Manager interface from Classy Components.*

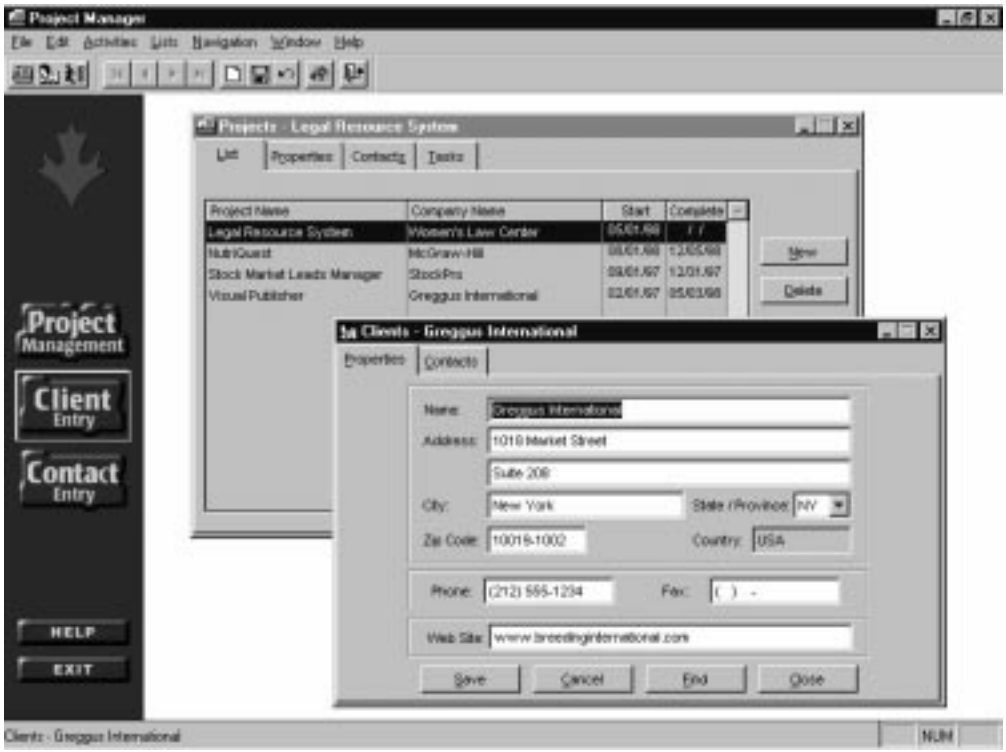


***Figure 16.20**. Mere Mortals Framework from Oak Leaf.*

QBF Builder

QBF Builder, by the same folks who brought you Mail Manager, is a control that you can drop on any data-entry form to provide "query by form" capabilities. It works with tables and local/remote views. You can save and rerun query definitions, and access advanced criteria ("contains", "inlist", "like", etc.) via a context menu on the query form. See **Figure 16.21**.

www.classycomponents.com

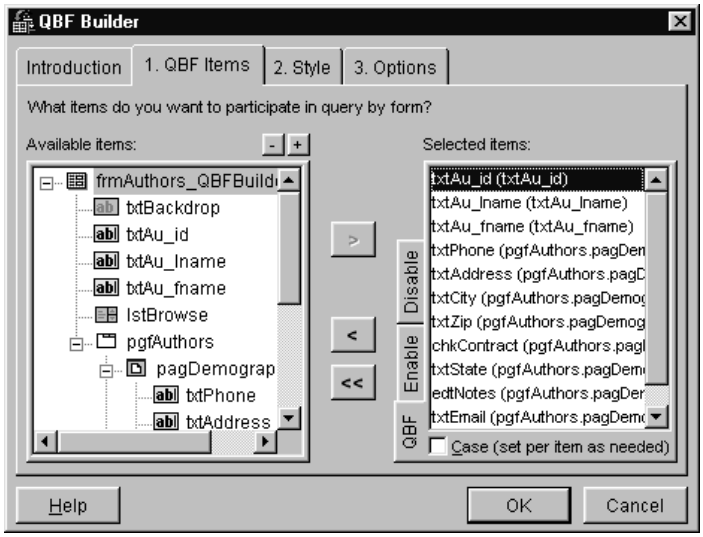


***Figure 16.21**. QBF Builder from Classy Components.*

Stonefield Database Toolkit

Visual FoxPro provides something FoxPro developers have needed for years—a built-in data dictionary. Visual FoxPro's data dictionary provides table and field validation, field captions, triggers, even table and field comments. See **Figure 16.22**.

Unfortunately, many things are missing from the data dictionary, such as structural information necessary to create or update table structures at client sites, and useful information such as captions for tables and indexes. Also, the tools Visual FoxPro provides to manage the database container itself are less robust than you'd expect. For example, altering the structure of a table breaks views based on that table. Table and field-name changes aren't propagated throughout the database, resulting in orphaned database objects. Moving a table to a different directory or changing the .DBF name causes the database to lose track of the table.

Stonefield Database Toolkit (SDT) overcomes these limitations and many others we've found in Visual FoxPro, and provides additional functionality that serious application developers need. There are three aspects to SDT:

- It enhances the tools Visual FoxPro provides to manage the database container.

- It provides the ability to define extended properties for database objects and set or obtain the values of these properties at run-time.
- It includes a class library you can add to your applications to provide database management functions at run-time, including recreating indexes, repairing corrupted table headers, and updating table structures at client sites.

www.stonefield.com

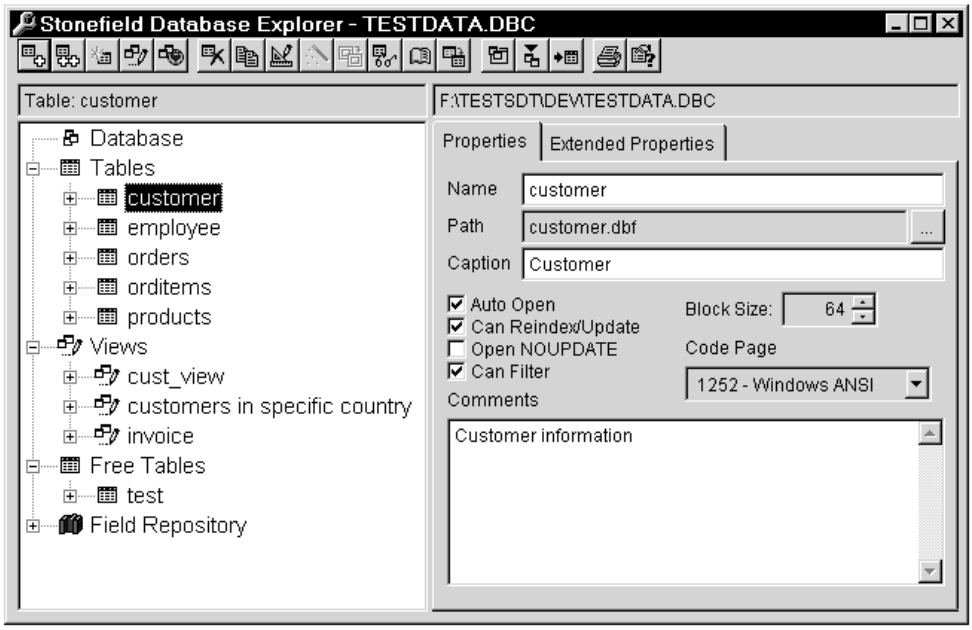


***Figure 16.22**. Stonefield Database Toolkit from Stonefield Systems.*

Stonefield Query

Stonefield Query is a powerful end-user query-builder tool. Users can specify fields and operators from multiple tables using English descriptions rather than cryptic names and symbols, without knowing about complex stuff like join conditions. See **Figure 16.23**.

It's easy to implement as a developer: simply drop an SFQuery object on a form, set some properties, and call the Show() method to display the Filter dialog. You can use the cFilter property to set a filter or as the WHERE clause in a SQL SELECT statement, or use the DoQuery() method to perform the SQL SELECT.

www.stonefield.com

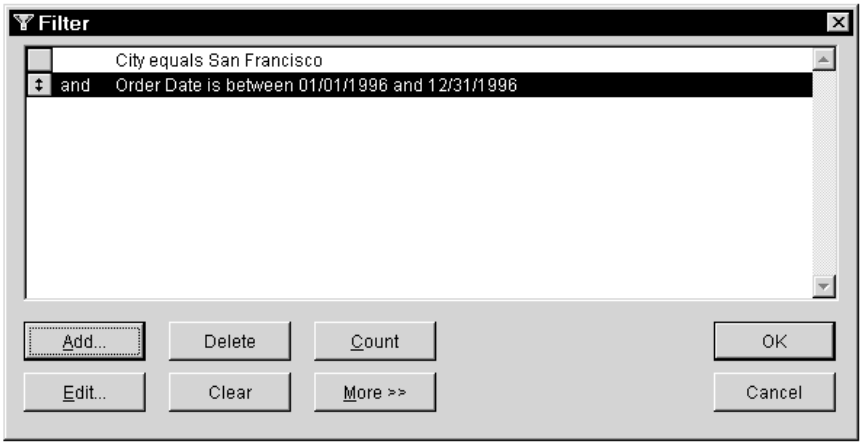


***Figure 16.23**. Stonefield Query from Stonefield Systems.*

Stonefield Reports

Stonefield Reports is an end-user report manager/writer. It sports a simple "wizard" interface—after selecting a report from the TreeView list of reports, your users can select the sort order, enter filter conditions, and select where the output should go (printer, disk file, spreadsheet, screen preview, and so on). See **Figure 16.24**.

In addition to running predefined reports, you can teach your users how to create their own "quick" reports in just minutes. They simply select which fields to report on from the list of available fields (full-text descriptions, of course), and they're done! For finer control, they can select from a "Field Properties" dialog how each field should appear, including column heading, grouping, and totaling.

www.stonefield.com

Visual FoxExpress

The Visual FoxExpress Framework, from F1 Technologies, is the first complete n-tier solution for Visual FoxPro that's suitable for all types of application development. It's a framework that's lean enough to deliver the performance your applications require and flexible enough to allow you to easily incorporate the complex business rules they need. It separates user interface, business rules, and data. It's possible to create user interfaces that are not directly bound to data at all. VFE apps can be deployed as COM or DCOM objects with no user interface, accessible from other front-ends such as ASP or DHTML Web pages, or built with other tools such as Visual Basic. FoxExpress generates 100% Visual FoxPro code, and if you need to get applications up and running quickly, you should have this in your bag of tricks. See **Figure 16.25**.

www.f1tech.com

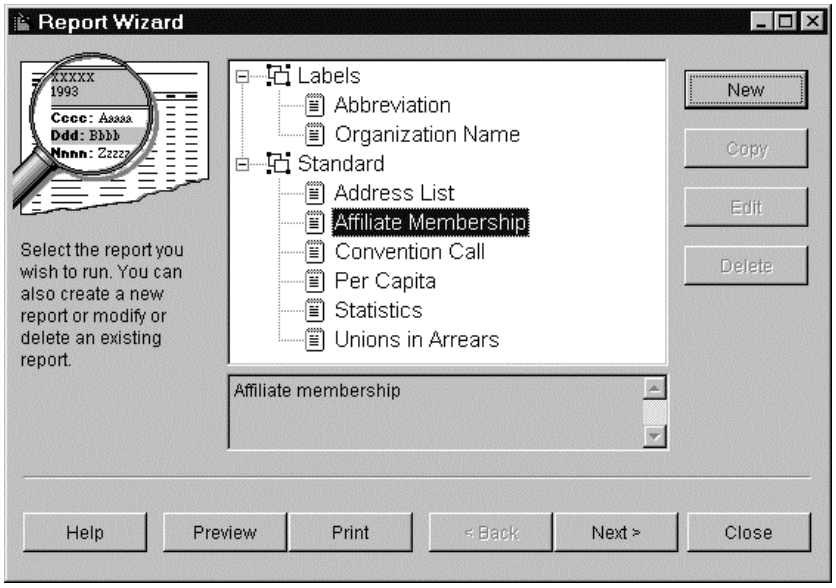


***Figure 16.24**. Stonefield Reports from Stonefield Systems.*

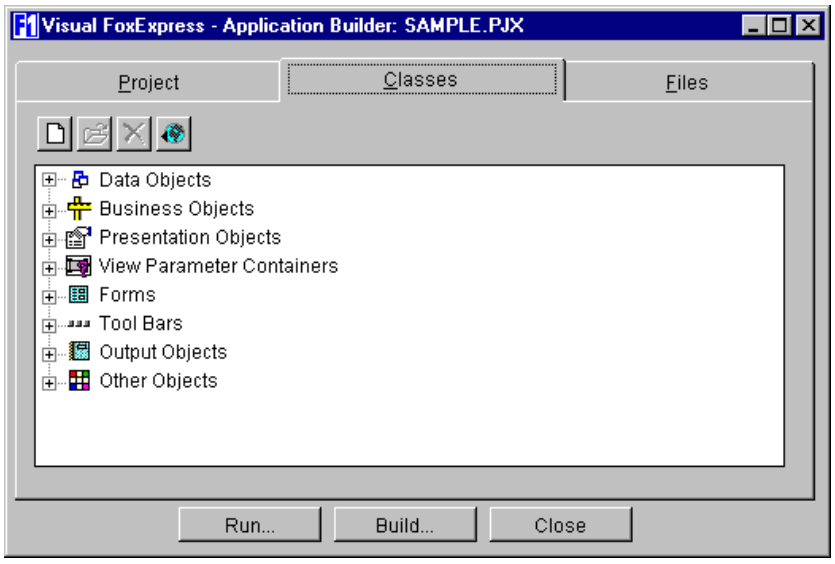


***Figure 16.25**. Visual FoxExpress from F1 Technologies.*

Visual MaxFrame Professional

Designed and produced by VFP expert Drew Speedie, VMP is an application framework for VFP and is now in its fourth version. It includes complete source code, extensive documentation, and a sample application. Version 4.0 includes client-server services, an optional security system, new and enhanced developer tools, as well as a wealth of improvements through the product. See. **Figure 16.26**

www.maxlink.com/vmpmain.htm

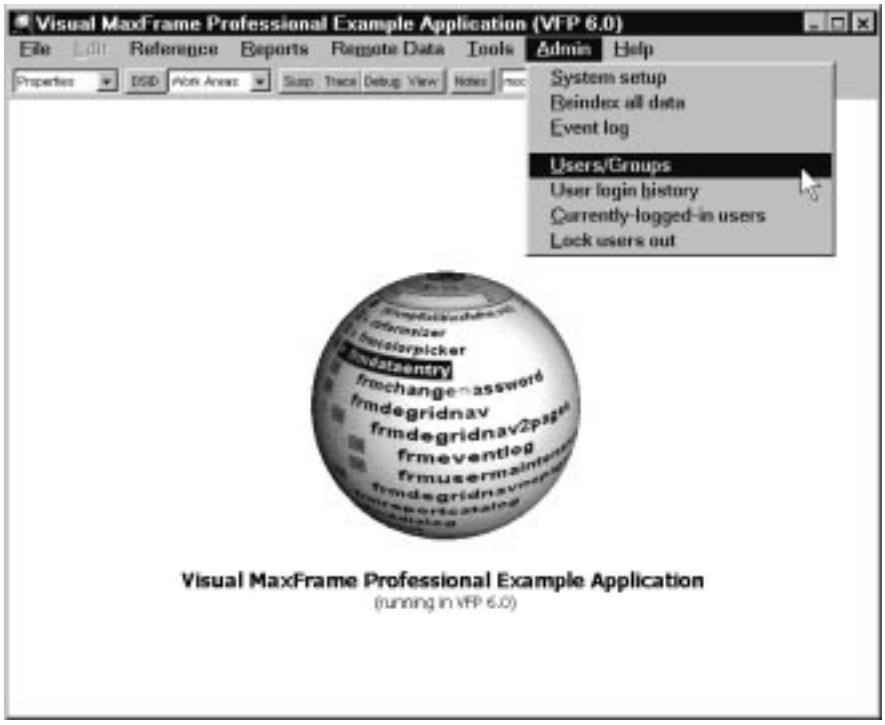


***Figure 16.26**. Visual MaxFrame Professional from Metamor Worldwide.*

Visual Web Builder

Are you ready to build Web- and HTML-based application the easy way? Need to plug HTML functionality into existing desktop applications for a rich user interface? Need to build back-end Web applications? Then check out Visual WebBuilder from West Wind Technologies and get ready to jump into the world of visual Web development using HTML, scripting, and Visual FoxPro code in a highly visual and interactive environment. See **Figure 16.27**.

Visual WebBuilder leverages the power of Visual FoxPro both internally in the framework as well as extending this power to your applications—you can run full FoxPro code inside

FoxPro classes, program files, and scripted HTML pages that mix HTML and FoxPro code and expressions.

A Visual Component Editor allows you to build small components that can interact with each other. Take code reuse to a new level by creating small, functional components that can be plugged into other components or be directly accessed from Web pages. Whether you use a Visual FoxPro back-end tool like West Wind Web Connection or Active Server Pages, Visual WebBuilder has you covered.

www.west-wind.com

***Figure 16.27**. Visual WebBuilder from West Wind Technologies.*

West Wind Web Connection

Rick Strahl's West Wind Web Connection is the premier tool for building Web applications with Visual FoxPro. See **Figure 16.28**.

Web Connection provides a scalable Web platform for building fast, powerful and flexible applications with ease. Take your pick between code-based HTML creation or FoxPro-based HTML scripting, dynamic form rendering or PDF Report generation to create your Web output. The supplied classes and tools greatly simplify receiving Web server requests, using the provided information and generating the final dynamic HTML and other data content like XML or raw data to return to the client.

www.west-wind.com

***Figure 16.28**. Web Connection from West Wind Technologies.*

xCase

xCase for Fox, from Resolution, Ltd., provides powerful database-design features that completely replace and greatly enhance VFP's native Database Designer. A highly sophisticated visual interface allows you to accurately design your database by capturing every detail about your customer's business information world. Then it produces high-quality diagrams, enabling you to present your design with different views and at various levels of detail. xCase will also assist you in generating all database-related code: Data Definition

Language code to build the database, triggers and stored procedures to safeguard data integrity, views and queries to extract data, and object-oriented code to provide all of the important metadata for the front end of your application. See **Figure 16.29**.

www.xcase.com

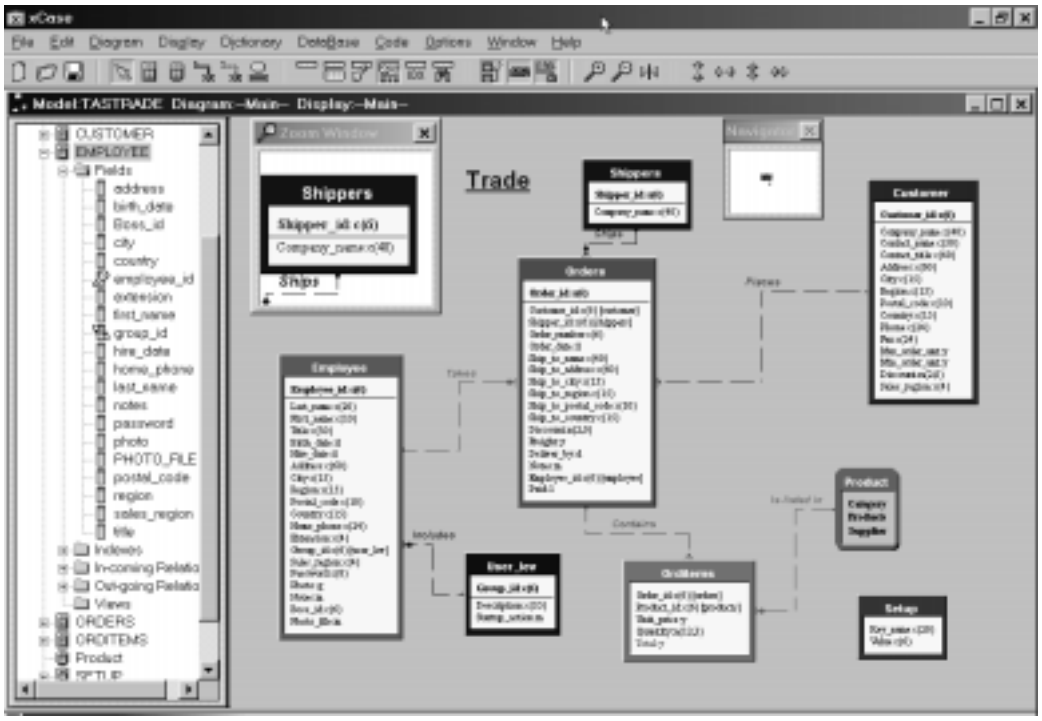


***Figure 16.29**. xCase from Resolution, Ltd.*