This paper was originally presented at the Southwest Fox conference in Mesa, Arizona in October, 2007. http://www.swfox.net

Framework Fundamentals

Session Number 8

Rick Borup
Information Technology Associates
701 Devonshire Drive, Suite 127
Champaign IL 61820
Voice:217.359.0918
Email: rborup@ita-software.com

In the same way design patterns provide a structure to help you solve common design problems, frameworks provide a structure to help you build a complete software application. Most applications share a number of things in common: menus, forms, toolbars, methods for rendering reports, retrieving and updating data, etc. A framework provides a standardized structure for these components and a standardized approach to managing the interactions between them. Whether you intend to use a full-featured commercial framework or just want to create a simple one of your own, understanding what a framework is and what it's intended to do will help you become a better developer.

Introduction

What this session is

This session explores the fundamental concepts involved in building a framework for Visual FoxPro applications. It is part of the "Reviewing VFP Fundamentals" track of the conference, and is therefore geared toward entry-level programmers or those new to Visual FoxPro. However, experienced Visual FoxPro developers who have never used a framework may find the material useful as well.

What this session is not

Although this session shows you how to build a simple framework for Visual FoxPro applications, it is not a comprehensive tutorial on building your own framework nor is it an introduction to or overview of any of the commercial frameworks for Visual FoxPro. The material in this session is not intended to be representative of any commercial framework, and the techniques presented here may or may not be similar to those used by any of the commercial frameworks. If you are interested in information about a particular commercial framework for Visual FoxPro application development, I've included some links in the Resources section at the end of this paper that you may find helpful.

What is a framework

A framework for software application development is like a bucket of parts that can be fit together to form a whole. Each part is designed to fulfill a particular purpose and to interact with other parts in a known manner. Although specific to their framework, the parts of any given framework are generic enough that they can be put together in a wide variety of ways to create many different types of applications.

Most everyone is familiar with LEGOs, the popular children's construction toy. The little red LEGO bricks and other parts can be found in the play rooms and living rooms of homes around the world. One reason they're so popular is that they're fun, easy to work with, and can be used to build a wide variety of toy structures. Adults love them, too. If you've ever been to LEGOLAND® you know what I mean—there you can find an amazing variety of intricate and often life-sized structures built from tens or hundreds of thousands of small LEGO parts.

You can think of a software application framework as analogous to a bin full of LEGO parts. Each part performs only one function, but can be connected to other parts to create large and complicated structures. In other words, a brick is always a brick but it can become part of a wall or part of a spaceship. A software framework is similar in that there are a variety of parts, each of which is designed to perform essentially one function but which can employed for specific purposes and combined with other parts to create entire applications.

Visual FoxPro applications typically consists of a set of class libraries and an underlying conceptual structure for tying them together into a working application. Like applications created with other development tools, Visual FoxPro applications typically need menus, forms, toolbars, reports, data access capabilities, and so on. Correspondingly, a Visual FoxPro application framework typically has classes for creating, managing, and coordinating the interaction between

the various controls, menus, forms, toolbars, reports, data access components, etc. used in any given application.

Although Visual FoxPro is an object-oriented language, it has procedural roots. Therefore it's not uncommon for a framework to have some procedural bits as well as class libraries. Regardless of whether the pieces are classes or procedures, within any given framework they are designed according to some conceptual structure that governs how they work with one another. This is analogous to the size and shape of the little round knobs on top of a LEGO brick, which are designed to align with and fit precisely into the holes on the underside of other LEGO bricks.

While they can be used to build just about anything, things built with LEGOs have a particular look and feel that's easily distinguishable from things built with other types of construction toys. For example, it's easy to distinguish a toy house built with LEGOs from one built with Tinker Toys or Lincoln Logs®. In the same way, software applications built with any given framework tend to have a similar look and feel to their user interface, so it's generally easy to distinguish applications built with one framework from applications built with another.

Why do I need a framework

The short answer to the question "Why do I need a framework" is, you don't *need* a framework. The better answer is that using a framework frees you from the drudgery of creating and hooking up the basic plumbing for every application you build, thus allowing you focus your attention on the business solution instead of the mechanics. A framework enables you to build applications more quickly and more reliably.

If you use a commercial application framework you will spend a good deal of time learning it, but once you become adept at using it you can be more productive than if you build each application from scratch. If you decide to build your own framework you will spend a good deal of time designing, building, and enhancing it, but in the end you will have a framework that does things exactly they way you want it to. You will also have a very thorough understanding of how it works because you built it from the ground up. Either way, once you have a good framework and know how to use it, you can deliver solid, working applications in a shorter time than without one.

If you build a lot of applications without using a framework, you will tend to find yourself doing many of the structural things for each application in the same way every time. This only makes sense: If you've solved a problem once, why invent a different solution the next time? Because of this, you will over time tend to create a *de facto* framework of your own, even if you don't consciously set out to do so.

Value of a framework for beginners

It's reasonable to ask, should a beginning Visual FoxPro programmer attempt to learn a framework right away? I think for the beginner, it's a double-edged sword: On one hand, using a framework most likely makes it possible for the new programmer to deliver a working app more quickly than would otherwise be possible. On the other hand, if a new programmer learns a framework at the same time he or she is learning Visual FoxPro itself, it may be difficult to distinguish what VFP can do natively from how the framework does it. The "aha!" moment may

not come until later, when the programmer is exposed to another way of implementing a solution and realizes the framework they first learned provided *one* way of doing it but not necessarily the *only* way. The ideal scenario is probably for the beginning programmer to gain a fairly comprehensive understanding of Visual FoxPro core concepts and a working familiarity with the language before attempting to learn a full-fledged commercial framework.

In researching this session, I came across the following comments about frameworks in the *VFP Rookie Mistakes* topic on the FoxPro Wiki.[1] I think it's well written I'd like to quote it here verbatim. Several people contributed to this topic on the Wiki; some of them left their names, but unfortunately this particular section is un-attributed so I can't give the author the credit he or she deserves.

> "Study a framework. It will show you how things can be done, as well as how to avoid the pitfalls of VFP. Visual Maxframe has a shareware version on their web site that is a worthwhile study. Also, the original Codebook is freely available and worth a look. You may decide that using a framework is worthwhile. It usually is, except for the simplest of projects.
>
> "Don't expect to be successful with any framework until you've first learned a considerable amount about OOP, data buffering, the database container and the VFP designers. With or without a framework you need to have a fair amount of VFP knowledge before you can be successful.
>
> "Hold the chainsaw by the correct end. Frameworks took years to design, build and refine, and will take time to master. Use the framework as it was intended. On your early outings, do things the way the framework expects you to, or you'll be treading uncharted territory where side-effects can make the framework your adversary. As you learn the intricacies of the tool, you can start to cautiously experiment with going outside the lines."

Concepts for framework development

This section introduces what a framework should be expected to do and what it should not be expected to do. It then looks in some detail at the concept of Visual FoxPro classes and subclasses, which are the fundamental building blocks of a framework. It wraps up with a discussion of what belongs in an application's main program, which is a precursor to building a framework.

What should a framework be expected to do

A software framework should be expected to provide all of the classes, procedures, templates, and other bits necessary to construct a working application. A framework might also include an application generator the developer can run to create a skeleton application, or base application, to use as the starting point for every application. For Visual FoxPro, this skeleton app takes the form of a VFP project file referencing all of the necessary parts to build a functioning app, but without any of the application-specific bits such as data, forms, custom menus, reports, and so on.

[1] http://fox.wikis.com/wc.dll?Wiki~VFPRookieMistakes

Ideally, a framework should also provide tools making it easy for the developer to add application-specific resources to the project, and to enable the developer to easily subclass the framework's base classes to implement enhancements and customized behaviors.

Finally, no framework should be considered complete without full documentation explaining the design concepts employed by the framework designers and documenting the structure of and interrelationship between the classes, procedures, and other parts comprising the framework.

What should a framework not be expected to do

A framework should not be expected to create a working application for a specific purpose without modification by the developer. Frameworks are by definition generic. If a framework were designed to create only one specific type of application, that framework's usefulness would be severely limited.

A developer using a framework needs to learn how to use the framework as a tool for creating individual applications for specific purposes. The framework frees the developer from having to construct and hook up the basic plumbing for every application from scratch, thereby enabling the developer to spend time more productively by concentrating on the tasks required to customize and enhance the app for each specific purpose. However, a framework does not relieve the developer of the responsibility to design a good solution and to put the pieces together in the best way possible to meet the user's needs.

Classes and subclasses

In Visual FoxPro, as in any object-oriented programming language, the fundamental building block is the class. The beauty of classes, of course, is that their essential functionality is encapsulated in a base class whose behavior can be modified or extended in subclasses.

Visual FoxPro ships with several base classes that can be used “as is” to add functionality to an application. Some of these are visual classes, such as the Form class and the classes for controls used on a form. Visual classes have a visual representation in an application at runtime. Other VFP base classes, such as the Collection class and the Session Object class, are non-visual classes. Non-visual classes do not have a visual representation at runtime.

In addition to being considered visual or non-visual, the VFP base classes are also classified as either container classes or control classes. Container classes create objects that can contain other objects. For this reason, container classes have an AddObject method control classes lack.

With the exception of the Column, Header, and Empty classes, all of the Visual FoxPro base classes can be subclassed. While it is certainly possible to create application objects directly from the VFP base classes, the best practice is to create subclasses of the VFP base classes and use those subclasses as the base classes from which any further subclasses are derived. Sample code for doing this is shown later, in the section called "Building your own framework."

A framework typically includes a set of framework-level base classes along with several subclasses, each of which delivers increasingly specific functionality as you move down the class hierarchy. For example, at the framework base class level there might be class library containing subclasses of all of the Visual FoxPro controls, derived directly from the Visual FoxPro base

classes. Some of these framework-level base class controls might in turn have subclasses of their own in order to implement some specific functionality, such as a read-only text box or a label that serves as a hyperlink. A commercial framework typically supplies a wide variety of classes for all sorts of objects the developer can use to create a fully functional application.

Visual class libraries

In Visual FoxPro, class definitions can be stored either in a visual class library (.vcx file) or in a program (.prg file). Each has its advantages and disadvantages, and which you use often comes down to a matter of personal preference. Most likely you will use both, at different times and for different types of classes.

Class libraries are created with the CREATE CLASSLIB command. The CREATE CLASSLIB command takes only one argument, which is the name of the class library to be created. If the name specified does not have a file name extension, Visual FoxPro uses the default .vcx. For example, the following statement creates a visual class library named myClasslib.vcx in the current directory.

```
CREATE CLASSLIB myClasslib
```

Class definitions in a visual class library can be created with the CREATE CLASS command. The syntax of the CREATE CLASS command provides for the name of the class being created, the class library in which the class is to be stored, the class from which the new class is derived (the *parent class*), and, if the parent class is not a VFP base class, the class library in which the parent class definition is stored. For example, the following command creates a new class named myTextbox whose parent class in the VFP textbox base class, and stores it in a visual class library named myClasslib.vcx.

```
CREATE CLASS myTextbox of myClasslib as textbox
```

Like the CREATE CLASSLIB command, the CREATE CLASS command can be run from a program or from the command window. Either way, the CREATE CLASS command opens the Class Designer window, which you can use to modify the properties and methods of the newly created class.

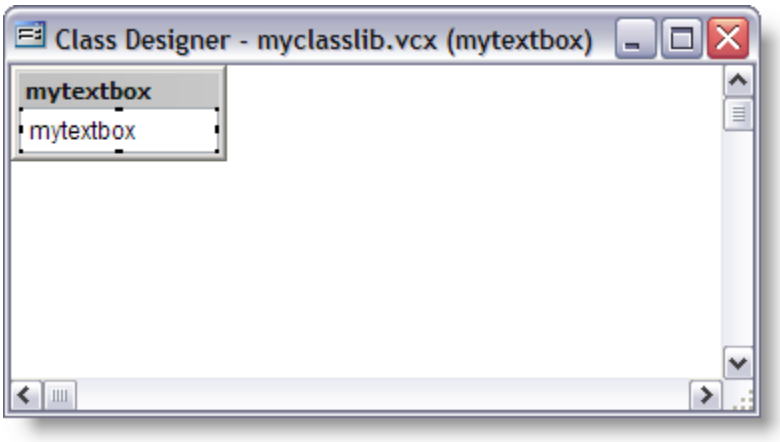

Figure 1: The Class Designer window is used in conjunction with the Properties sheet to modify the properties and methods of a class.

You can also create a new class from the Class Browser by clicking the New Class button on the toolbar, or by issuing the CREATE CLASS command with no arguments. In both cases, VFP displays the New Class dialog in which you can specify the class name, class library, parent class, and (if necessary) the parent class library.

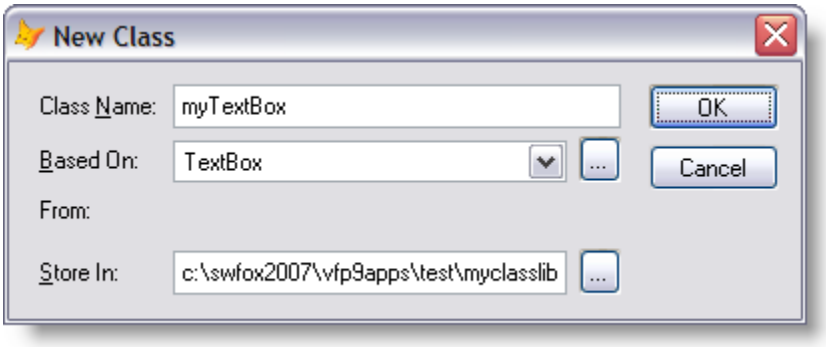

Figure 2: The New Class dialog enables you to create a new class by specifying its name, class library, parent class, and parent class library.

One possible source of confusion when dealing with visual class libraries arises from what some might consider an ambiguity in the use of the word *visual*. In the term *visual class library*, the word *visual* can be thought of as referring more to the nature of the class library than to the nature of the classes stored within. While visual class libraries can only store classes that can be modified visually, some non-visual classes can be modified visually and can therefore be stored in a visual class library. The Collection class is one example. The following code works just fine, even though the Collection class is a "non-visual" class in that it does not have a visual representation at runtime.

```
CREATE CLASS myCollection OF myClasslib as Collection
```

Not all Visual FoxPro classes can be modified visually. Two that cannot are the Exception class and the Session class. Because they cannot be modified visually, subclasses derived from either of these two classes cannot be stored in a visual class library.

A typical visual class library contains many individual class definitions. While they are a convenient way to organize classes, visual class libraries do have some disadvantages. For one thing, a visual class library is a two physical files on disk, a .vcx and a corresponding .vct. For another, when you use a class from a visual class library in a project, the entire class library is included in your project (and hence in your compiled application) even if you use only one class from that library. With large class libraries, this could be a consideration if the size of the compiled EXE is of concern.

PRG-based classes

Class definitions can also be stored in program (.prg) files. This is accomplished with the DEFINE CLASS command. A single program file can contain several class definitions, or it may contain only one. A program file containing one or more classes definitions can be referred to as a class library, although that term is generally used as shorthand for a visual class library. Classes defined with a DEFINE CLASS statement in a program file are referred to as prg-based, or prog-based, classes.

The DEFINE CLASS statement can be used to subclass any Visual FoxPro base class that VFP allows to be subclassed. This includes the VFP base classes that cannot be modified visually. While prg-based classes may typically be used for subclassing non-visual classes, you can use DEFINE CLASS to subclass any type of class. The choice depends largely on whether you prefer to work with the subclass programmatically rather than visually. For example, the application object used in the sample framework for this session is a prg-based class derived from the VFP Custom base class. Because it is a non-visual class, I prefer to work it in the VFP editor instead of in the class designer.

DEFINE CLASS statement is quite a powerful command. It enables you not only to create a class, but also to specify class properties and methods, among other things. This includes the ability to modify properties and methods of the parent class as well as to add new properties and methods that are unique to the subclass.

The code in Listing 1 illustrates how DEFINE CLASS works. It creates a subclass of the VFP textbox base class, modifying the values of some of the textbox's native properties and methods. Specifically, it sets the BackColor to pale yellow, the width to 150, the height to 21, and adds a new property named *iseditable* with an initial value of True. Finally, it overrides the When event base class code to return the value of the iseditable property.

Listing 1: The DEFINE CLASS command creates a class definition including properties and methods.

```
DEFINE CLASS myTextBox as Textbox
*  Base class properties
BackColor = 12648447  && pale yellow
Width = 150
Height = 21
*  Custom property
iseditable = .T.
*  When() event method code
FUNCTION When() as Logical
RETURN this.iseditable
ENDFUNC
ENDDEFINE && myTextBox
```

Did you know you can use Class Browser with prg-based classes as well as visual class libraries? In fact, the Class Browser works with visual class libraries (.vcx), forms (.scx), program files (.prg), and even project files (.pjx). If a single program file contains more than one class definition, the Class Browser shows all of them in the tree view just as it does when displaying a visual class library. If you right-click on a prg-based class in the Class Browser and choose Modify from the popup menu, VFP opens the .prg file in the code editor.

What belongs in main.prg

Every Visual FoxPro project must have a main file. The main file defines the starting point for the application, in other words the code that runs first. The main file can be either a program or a form, but it is customary and generally more useful to use a program (a .prg file).

The main file is set in the Project Manager by right-clicking on it and choosing "Set as main" from the popup menu. After the main file has been set, its name is shown in bold in the Project Manager tree view. A project can have only one main file.

What you put in the main program file depends on what you want it to do and to some extent on what kind of app it is. Most frameworks implement an application object to serve as the highest level object in the application's object hierarchy at runtime. In this case, about the only thing the main program has to do is to instantiate the application object and tell the application to run. The concept of an application object is covered a bit later. For starters, it's easier to illustrate the basic concepts of a main program with examples for an app that does not use an application object.

If your app begins (as many do) by showing a form in the main Visual FoxPro screen, then about the simplest main program you can have is the one shown in Listing 2.

Listing 2. A simple main program

```
DO FORM myForm NAME ofrmMyForm
READ EVENTS
RETURN
```

The READ EVENTS command tells VFP to start event processing, in other words to begin responding to events initiated by the user. One of the most common mistakes made by beginning VFP programmers is forgetting to issue READ EVENTS. If you don't include it, your form flashes briefly on the screen and the application terminates immediately.

If your app runs as a top-level form (using Form.ShowWindow = 2) and you don't want the Visual FoxPro screen to be visible behind it, you can add a line of code at the top of the main program to make the main VFP screen invisible, as shown in Listing 3.

Listing 3. A simple main program for an application with a top-level form

```
_SCREEN.VISIBLE = .F.
DO FORM myForm NAME ofrmMyForm  && A top-level form
READ EVENTS
RETURN
```

This works, but hiding the main VFP screen in this manner is not an optimal solution because the screen may visibly flash on and off before your form appears. A better way to hide the main VFP screen is to create an application-specific configuration file (config.fpw) and use SCREEN=OFF.

At shutdown time, a VFP application must issue a CLEAR EVENTS to terminate the processing started by READ EVENTS. When CLEAR EVENTS is executed, the event loop is terminated and processing continues with the statement after READ EVENTS. In general, the statement(s) following READ EVENTS should run whatever code is necessary to clean up and shut down the application gracefully.

You may have noticed in Listings 2 and 3 that the next statement after READ EVENTS is RETURN. The use of an explicit RETURN is optional, but I recommend including it as the last statement in the main program. Some examples (and even some real-life applications) use the QUIT statement at the end of the main program. The disadvantage of using QUIT comes into play if you run the application from within the Visual FoxPro IDE, which for many developers is common practice during the coding and testing cycle. If the application's main program ends with a QUIT statement, Visual FoxPro itself terminates when the application terminates, which is usually not the desired behavior during development. Using a RETURN statement instead of QUIT avoids this problem. In the development environment, RETURN returns control to the VFP IDE. In the runtime environment, there is no higher level program in the call stack to return to, so RETURN does the same thing as QUIT.

In an application with only one form, the app should most likely terminate when the form is closed. A logical place for CLEAR EVENTS in this case is in the form's Unload method, which is the last method to run before the form is released. When the user closes the form, the Unload event fires and the CLEAR EVENTS command runs. This returns control to the main program, which does whatever cleanup code may be necessary and then terminates.

In a real application you are most likely going to have more than one way to terminate the application. For example, there might be an Exit item on the File menu as well as a way to terminate the app from a form or toolbar. It's also possible your application could be told to terminate by events outside of the application itself, for example if the user shuts down Windows while the app is still running.

You can use an ON SHUTDOWN command to ensure that the CLEAR EVENTS command always gets executed when the app is told to terminate. The ON SHUTDOWN command defines the code to be run when the application is told to shut down. The ON SHUTDOWN command code can be stored in a procedure, a method, or even a separate program file. Because it is invoked whenever the application is told to shut down, the ON SHUTDOWN command code is the logical place to put the CLEAR EVENTS statement. It also provides a place where you can check for and handle conditions that would prevent gracefully shutting down the app, such as open forms, processes still pending completion, or uncommitted database updates.

The ON SHUTDOWN code can be as simple as that shown in Listing 4.

Listing 4. A generic shutdown method

```
ON SHUTDOWN    && Release the current ON SHUTDOWN command (this program).
CLEAR EVENTS   && Stops the processing started with READ EVENTS.
```

The comments in Listing 4 indicate the purpose of each of the two lines. The reason you need an ON SHUTDOWN with no command after it in your ON SHUTDOWN method is that if you don’t, you'll be unable to terminate the application because it's effectively in an infinite loop, re-entering the ON SHUTDOWN code every time you tell it to quit.

If you use an ON SHUTDOWN program like the one shown in Listing 4, you can call it from any place in your application where you want to terminate the application. Thus in the example of an application with only one form, the form's Unload method could contain DO myShutdown.prg instead of CLEAR EVENTS.

As a safety net, you can put a global ON SHUTDOWN command in the main program to trap all requests to terminate the app and ensure the ON SHUTDOWN program is run regardless of where the termination request originates. Listing 5 shows a modified main program to implement this idea.

Listing 5. A simple main program with a global ON SHUTDOWN command

```
ON SHUTDOWN DO myShutdown.prg
DO FORM myForm NAME ofrmMyForm
READ EVENTS
RETURN
```

Using ON SHUTDOWN code also avoids the "Cannot Quit Visual FoxPro" condition that arises if Visual FoxPro is told to shut down, for example by closing the main VFP screen, while an event

loop is still in effect. If this condition occurs during development the developer can handle it by canceling the program, but in the runtime environment the end user has no way to terminate the app short of killing the process from the task manager.

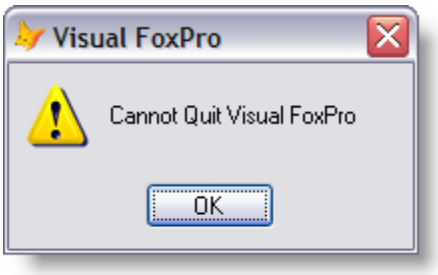

Figure 3: The "Cannot Quit Visual FoxPro" condition occurs if you try to shut down VFP while an event loop is still active.

In a real application, you are most likely going to want to do a lot of other things before opening the first form. For example, you may want to set up the runtime environment by issuing the appropriate SET statements. In a simple app with no application object, the main program is a good place to put these commands. Similarly, you might want to clean up the runtime environment by issuing a CLOSE ALL and related commands prior to terminating the app. The code in Listing 6 illustrates a slightly more complete main program that implements these ideas.

Listing 6. A somewhat more complete main program

```
SET TALK OFF
SET EXCLUSIVE ON
SET DELETED ON
ON SHUTDOWN DO myShutdown.prg
DO FORM myForm NAME ofrmMyForm
READ EVENTS
CLOSE ALL
RETURN
```

Other tasks commonly performed by the main program but not shown in Listing 6 include showing a splash screen, opening the database, setting the path, setting up a global error handler, putting up the application's main menu, setting the screen icon, caption, and background (if any), launching the security process for user login (if any), and so on.

If you decide to use an application object, much of the code that might otherwise be in the main program usually gets placed in a method or methods of the application object or one of its subordinate objects. When this is the case, you still need a main program as the application's starting point, but it need do nothing more than instantiate the application object and call whatever method is used to start up the app. The start-up code is often placed either in the application object's Show method or in a custom method named Run, Start, or whatever else you want to call it. A sample main program for an application that uses an application object is shown in Listing 7.

Listing 7. A sample main program for an app that uses an application object

```
PUBLIC oApp
oApp = NEWOBJECT( "myAppClass", "myAppClass.prg")
oApp.Run()
```

```
RETURN
```

When an application object is used, the ON SHUTDOWN code is usually placed in a method of the application object and the ON SHUTDOWN command is something like this:

```
ON SHUTDOWN oApp.Shutdown()
```

The VFP application wizard

No discussion of framework fundamentals for Visual FoxPro would be complete without some mention of VFP's own built-in framework. While you might not necessarily build a production application with this framework, it's nevertheless instructional to know it's there and to learn how it works.

The Visual FoxPro built-in framework is a combination of two tools, the Application Wizard and the Application Builder. The Application Wizard generates a generic, skeleton project from the framework's classes and other resources. The Application Builder can then be used to add application-specific elements to the generic project to create a useful application.

The Application Wizard is launched from the main menu by choosing Tools | Wizards | Application. The Wizard prompts you for a project name, as shown in Figure 4, and creates a project file of that name in the specified location.

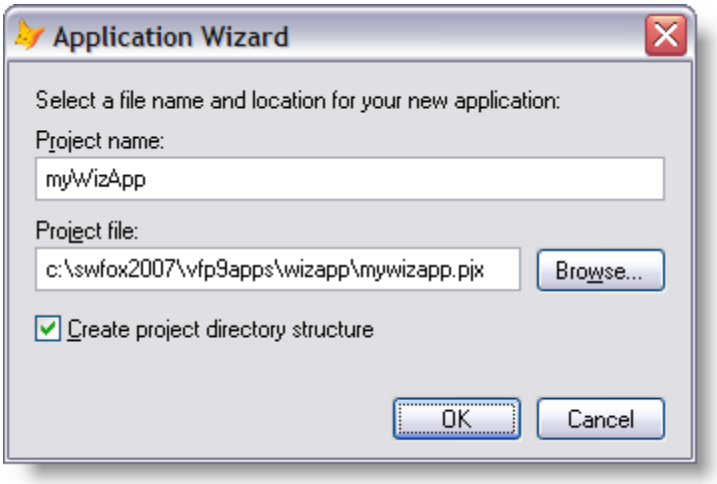

Figure 4: The Application Wizard prompts you for a project name and location.

If you mark the 'Create project directory structure' check box, the Wizard creates a standard set of subdirectories such as Data, Forms, Help, Include, and so on, and places the generated files into the appropriate subdirectory. If you do not mark this check box, all the generated files are placed in the project's root directory.

It takes several seconds for the Application Wizard to run to completion. When it finishes, it launches the Application Builder, shown in Figure 5. The Application Builder is a tool designed to help you add data, forms, and reports to the application.

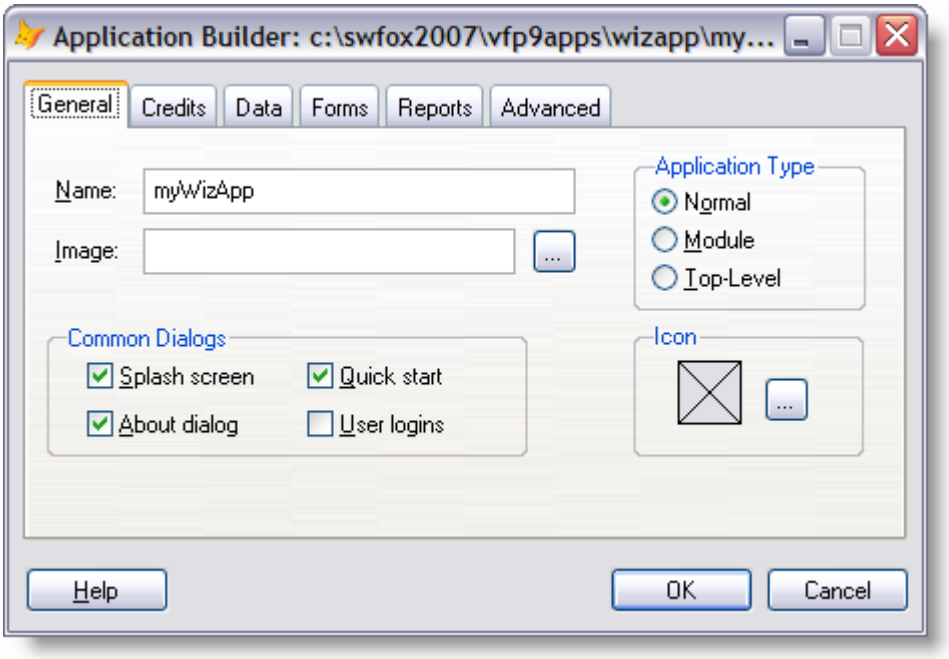

Figure 5: The Application Builder is a tool for setting project attributes and adding new elements such as data, forms, and reports.

You do not need to run the Application Builder at this point. If you close it and stop here, you can add new or existing databases, forms, reports, and other resources to the project via the Project Manager in the conventional manner. If you want to run the Application Builder later on, you can launch it from the VFP main menu via Tools | Wizards | All Wizards | Application Builder.

The project generated by the Application Wizard is a skeleton application comprising a main program, an application class library, a couple of menus, a configuration file, several header (#include) files, and other components. The Wizard also brings in references to several of the Visual FoxPro foundation classes, which are located in the ffc subdirectory under the VFP home directory.

Where applicable, the Application Wizard incorporates the name of the application into the names of the files it generates. For example, if you create an application named myWizApp, as shown in Figure 5, then the main program becomes myWizApp_app.prg, the application class library is called myWizApp_app.vcx, and so on.

The main program generated by the VFP application wizard is considerably different from the simple examples shown in Listings 2, 3, and 6. The downloads for this session include the generated code for an Application Wizard app named myWizApp. The main program, named myWizApp_app.prg, can be found in the myWizApp\Progs\ directory.

Although the Visual FoxPro documentation refers to the skeleton application generated by the Application Wizard as a framework, in my view that is not really the framework. The framework is the underlying classes and other resources from which the skeleton application is derived. In that sense, the skeleton app is more an instance of the framework than the framework itself.

Although it does nothing useful out of the box, you can compile the skeleton application into a executable and run it (see Figure 6). In actual practice you would of course need to add the data, forms, reports, and so on that are required to make a useful application. The purpose of the skeleton app generated by the Application Wizard, like that generated by any framework, is to give you a starting point.

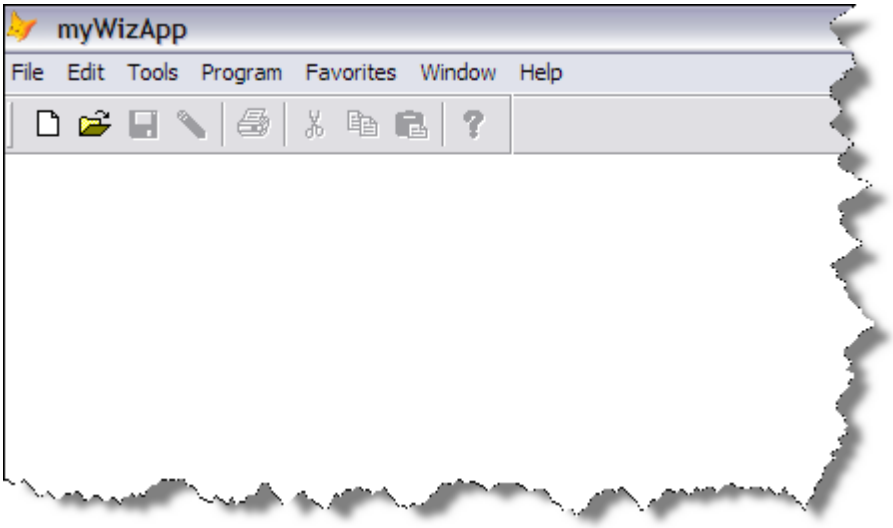

Figure 6: The skeleton application generated by the VFP Application Wizard can be compiled into an executable and run, although it does nothing useful until you add actual data, forms, and reports.

If you want to learn more about the Visual FoxPro Application Wizard, not only can you generate and experiment with an app of your own but you can also explore the source code for the Application Wizard itself. The source code is included in XSource.zip, which is distributed with VFP and can be found in the Tools\XSource subdirectory under the VFP home directory.

To be sure you have the latest version of the XSource files, download XSource.zip for Visual FoxPro 9.0 SP1 from Microsoft using the link to "XSource for Visual FoxPro 9.0 SP1" on the VFP Product Updates page at http://msdn2.microsoft.com/en-us/vfoxpro/bb190232.aspx. This release of the XSource files, which was not included in the VFP 9.0 SP1 download, also comes with an updated license permitting not only usage and modification but also distribution of the source code. See the January 2006 Letter from the Editor at http://msdn2.microsoft.com/en-us/vfoxpro/bb190239.aspx for more information on the new "permissive" license for this code.

Extract the contents of XSource.zip into Tools\XSource, retaining the folder names associated with each individual file within the zip file. After you do this, the XSource source code files are grouped into subdirectories under Tools\XSource\VFPSource. The project file for the Application Wizard is in Tools\XSource\VFPSource\Wizards\wzapp. Open the *wzapp* project file to explore the source code for the Application Wizard.

Building your own framework

When you first set out to create your own framework, don't try to do too much. Your goal is not to create something with all the features of a commercial framework. Your goal is to learn by

doing, and you can learn a lot by creating a simple framework that implements some of the basic concepts.

Chances are you'll never develop anything close to what a fully featured commercial framework does. Why would you want to, when there are some excellent commercial frameworks available at reasonable prices? But until you understand basic framework concepts, you cannot really understand or appreciate everything a commercial framework does for you.

The rest of this session is devoted to helping you learn basic framework concepts by developing a simple yet fully functional framework and seeing how it's used to create a working application. The sample framework for this session is called "My Framework". To help identify them, all of the framework's classes, class libraries, and program files begin with the prefix *my*, and I'll refer to the framework itself as *myFramework*. All of the code for myFramework and for the sample application are included in the session downloads.

Creating the framework base classes

The first step is to create a visual class library (vcx) containing a set of subclasses derived directly from the Visual FoxPro base classes. Because these first-level subclasses are direct descendents of the VFP base classes, they are sometimes referred to as "one-off" classes, meaning they're one level removed from the base classes.

Although you can do it by hand, it's easy to write a program to create a class library and populate it with subclasses derived directly from the VFP base classes. A sample program for this purpose is shown in Listing 8. The code creates a class library named myBaseCtrl.vcx for the base controls, and another class library named myBaseForm.vcx for the base form and base toolbar classes. The class libraries are created in a sub-directory called Classes underneath wherever the program file itself is located.

It's not necessary to separate controls and forms into separate class libraries, but some type of organization is helpful. You can certainly choose a different arrangement if you prefer. Note that not all the VFP base controls are subclassed in this example. The code in Listing 8 is included in the session downloads in the myFramework directory.

When you open the session download code in Visual FoxPro on your own machine, the indented lines will be aligned as intended if you set the Tab size to 3 (under Tools | Options | IDE).

Listing 8: This program creates class libraries containing one-off subclasses derived directly from the Visual FoxPro base classes.

```
*============================================================================
* Program:      CREATEBASECLASSES.PRG
* Author:       Rick Borup
* Date Written: 09/05/2007
* Copyright:    (c) 2007 Information Technology Associates
*                   All rights reserved.
* Compiler:     Visual FoxPro 09.00.0000.3504 for Windows
* Abstract:     Create framework-level base classes from VFP base classes.
```

```
* Environment in:
* Environment out:
* Parameters:
* Returns:
* Changes:
*=============================================================================
*  Framework base controls
#DEFINE _BaseLibrary ".\Classes\myBaseCtrl.vcx"
CREATE CLASSLIB _BaseLibrary
CREATE CLASS chkBase OF _BaseLibrary as CheckBox NOWAIT
CREATE CLASS cboBase OF _BaseLibrary as ComboBox NOWAIT
CREATE CLASS cmdBase OF _BaseLibrary as CommandButton NOWAIT
CREATE CLASS cmgBase OF _BaseLibrary as CommandGroup NOWAIT
CREATE CLASS cntBase OF _BaseLibrary as Container NOWAIT
CREATE CLASS ctlBase OF _BaseLibrary as Control NOWAIT
CREATE CLASS edtBase OF _BaseLibrary as EditBox NOWAIT
CREATE CLASS grdBase OF _BaseLibrary as Grid NOWAIT
CREATE CLASS imgBase OF _BaseLibrary as Image NOWAIT
CREATE CLASS lblBase OF _BaseLibrary as Label NOWAIT
CREATE CLASS linBase OF _BaseLibrary as Line NOWAIT
CREATE CLASS lstBase OF _BaseLibrary as ListBox NOWAIT
CREATE CLASS opgBase OF _BaseLibrary as OptionGroup NOWAIT
CREATE CLASS optBase OF _BaseLibrary as OptionButton NOWAIT
CREATE CLASS pgfBase OF _BaseLibrary as PageFrame NOWAIT
CREATE CLASS sepBase OF _BaseLibrary as Separator NOWAIT
CREATE CLASS shpBase OF _BaseLibrary as Shape NOWAIT
CREATE CLASS spnBase OF _BaseLibrary as Spinner NOWAIT
CREATE CLASS txtBase OF _BaseLibrary as TextBox NOWAIT
CREATE CLASS tmrBase OF _BaseLibrary as Timer NOWAIT

*  Framework base form and toolbar
#UNDEF  _BaseLibrary
#DEFINE _BaseLibrary ".\Classes\myBaseForm.vcx"
CREATE CLASSLIB _BaseLibrary
CREATE CLASS frmBase OF _BaseLibrary as Form NOWAIT
CREATE CLASS tbrBase OF _BaseLibrary as ToolBar NOWAIT
```

Each of the CREATE CLASS statements opens a class designer window. The NOWAIT statement tells VFP to continue with the next line of code without waiting for the class designer window to be closed. Therefore when this program finishes running, there will be several class designer windows open in the VFP IDE. Because you do not need to make any changes to these base classes at this point, just close each of the class designer windows. (Tip: Using Ctrl+W to close a lot of cascaded windows is much faster than using the mouse.)

Creating the framework subclasses

The next step in constructing the framework's class structure is to create some useful subclasses from the framework base classes. In order to keep things simple, only a few subclasses are presented here. They are meant to be representative of what in a real framework would be a wide variety of classes designed for all sorts of uses within an application.

The subclasses in this example are all stored in the same class libraries as the framework base classes from which they're derived. In a real framework, with dozens or perhaps even hundreds of classes, it would certainly be acceptable if not preferable design to create a separate class library or libraries for the framework subclasses.

The subclasses created for the sample framework are as follows:

- an "OK" command button with its caption set to OK and its Default property set to True;
- a "Cancel" command button with its caption set to Cancel and its Cancel property set to True;
- a display-only text box with its back color set to cyan and its When method set to always return False;
- a framework-aware base form with code in its Destroy method to communicate with the framework's form manager object and code in its Show method to restore its size if the form is minimized when launched;
- a subclass of the framework-aware base class with an OK button; and
- a subclass of the framework-aware base class with OK and Cancel buttons.

The code in Listing 9 is intended for demonstration purposes only. In actual practice you would most likely create these subclasses manually via the Class Designer instead of using code. The purpose of including this code here is to illustrate the properties and methods that are changed in the subclasses. The code in Listing 9 is included in the session downloads.

Listing 9: This program creates some useful subclasses of the framework base classes.

```
*=============================================================================
* Program:      CREATESUBCLASSES.PRG
* Author:       Rick Borup
* Date Written: 09/05/2007
* Copyright:    (c) 2007 Information Technology Associates
*                   All rights reserved.
* Compiler:     Visual FoxPro 09.00.0000.3504 for Windows
* Abstract:     Create framework-level subclasses.
* Environment in:
* Environment out:
* Parameters:
* Returns:
* Changes:
*=============================================================================
#DEFINE _BaseLibrary ".\Classes\myBaseCtrl.vcx"

*     "OK" command button
CREATE CLASS cmdBaseOK OF _BaseLibrary as cmdBase FROM _BaseLibrary NOWAIT
TEXT TO lcMethodCode NOSHOW
*     Method: cmdOK.Click()
thisform.Release()
ENDTEXT
_cliptext = lcMethodCode
MODIFY CLASS cmdBaseOK OF _BaseLibrary METHOD Click
***   Paste method code
***   Set Caption to \<OK
***   Set Default to .T.

*     "Cancel" command button
CREATE CLASS cmdBaseCancel OF _BaseLibrary as cmdBase FROM _BaseLibrary NOWAIT
TEXT TO lcMethodCode NOSHOW
*     Method: cmdCancel.Click()
```

```
thisform.Release()
ENDTEXT
_cliptext = lcMethodCode
MODIFY CLASS cmdBaseCancel OF _BaseLibrary METHOD Click
***   Paste method code
***   Set Caption to \<Cancel
***   Set Cancel to .T.

*     Display-only textbox
CREATE CLASS txtBaseDisplayOnly OF _BaseLibrary as txtBase FROM _BaseLibrary
       NOWAIT
TEXT TO lcMethodCode NOSHOW
*     Method: txtBaseDisplayOnly.When()
RETURN .F.
ENDTEXT
_cliptext = lcMethodCode
MODIFY CLASS txtBaseDisplayOnly OF _BaseLibrary METHOD When
***   Paste method code
***   Set BackColor to 128,255,255 (cyan)

DO ( _Browser) WITH ( _BaseLibrary)

#UNDEF  _BaseLibrary
#DEFINE _BaseLibrary ".\Classes\myBaseForm.vcx"

*     A framework-aware base form
CREATE CLASS frmBaseAppAware OF _BaseLibrary as frmBase FROM _BaseLibrary
       NOWAIT
TEXT TO lcMethodCode NOSHOW
*     Method: frmBaseAppAware.Destroy()
IF TYPE( "oApp.oFormMgr") = "O" AND !ISNULL(oApp.oFormMgr)
   IF PEMSTATUS(oApp.oFormMgr, "ReleaseForm", 5) = .T. AND ;
      PEMSTATUS(oApp.oFormMgr, "ReleaseForm", 3) = "Method"
      oApp.oFormMgr.ReleaseForm( thisform.name)
   ENDIF
ENDIF
ENDTEXT
_cliptext = lcMethodCode
MODIFY CLASS frmBaseAppAware OF _BaseLibrary METHOD Destroy
***   Paste method code

TEXT TO lcMethodCode NOSHOW
*  Method: frmBaseAppAware.Show()
LPARAMETERS nStyle
WITH thisform
   IF .WindowState = 1        && If form is minimized,
      .WindowState = 0        && set it to normal.
   ENDIF
ENDWITH
ENDTEXT
_cliptext = lcMethodCode
MODIFY CLASS frmBaseAppAware OF _BaseLibrary METHOD Show
***   Paste method code

*     A framework-aware subclass form with an OK button
CREATE CLASS frmBaseOK OF _BaseLibrary as frmBaseAppAware FROM
       _BaseLibrary NOWAIT
***   Add OK button from myBaseCtrl.vcx
```

```
*     A framework-aware subclass form with OK and Cancel buttons
CREATE CLASS frmBaseOKCancel OF _BaseLibrary as frmBaseAppAware FROM
       _BaseLibrary NOWAIT
***   Add OK and Cancel buttons from myBaseCtrl.vcx

*     View classes in class browser
DO ( _Browser) WITH ( _BaseLibrary)

RETURN
```

Now that the framework-level classes and sub-classes for forms and controls have been created, we can turn our attention to the non-visual classes for the application object, the forms manager, the menu manager, and the reports manager.

Creating the application object class

When an application is first launched, several startup tasks typically have to be done before the user can begin interacting with the application. For example, the runtime environment needs to be set up, a splash screen might be displayed, a database may need to be opened, the main menu needs to be put up, and an initial form may need to be opened. Remembering the section about what belongs in the main program, you know that these things can be done in main.prg or they can be assigned to the application object or one of its delegates. In order to stick with an object-oriented design as much as possible, the sample framework for this session uses the second approach of assigning the application initialization tasks in the application object. As shown in Listing 10, the main program for the sample framework does nothing more than instantiate the application object.

Listing 10: The main program for the sample framework does nothing more than instantiate the application object and call its Run method.

```
PUBLIC oApp
oApp = NEWOBJECT( "myAppCls", "myAppCls.prg")
oApp.Run()
RETURN
```

As you can tell from the main program, the application object for myFramework is a prg-based class called myAppCls defined in a program file named myAppCls.prg. After instantiating the application object, the main program calls its Run method. The Run method, which is shown in Listing 11, is responsible for doing all of the tasks necessary to initialize the application and for starting the event loop by issuing READ EVENTS.

Listing 11: The application object's Run method is responsible for initializing the application and for starting the event loop.

```
PROCEDURE Run() as VOID
ON SHUTDOWN oApp.Shutdown() && Use "oApp.Shutdown()", not "this.Shutdown()".
this.InitializeApp()
ON ERROR oApp.oErrorHandler.HandleError( ERROR(), PROGRAM(), LINENO(),
         MESSAGE(), .T.)
_SCREEN.Visible = .T.
READ EVENTS
ENDPROC && Run
```

In order to keep the size of the Run method to a minimum, most of the detail tasks required to initialize the application are delegated to the InitializeApp method. As you can see from Listing

11, the Run method sets up the ON SHUTDOWN code, established an error handler via the ON ERROR command, makes the screen visible, and issued READ EVENTS.

At runtime, shutdown requests and errors typically occur in code running outside of the application object, for example in forms, controls, or methods of other objects. Therefore the ON SHUTDOWN and ON ERROR commands issued by the application object must be functional regardless of where they are invoked. That's why, in those two commands, the Run method refers to the application object by name (oApp) instead of using "this". At runtime, "this" does not always refer to the application object, but oApp always does.

The primary responsibility of the Shutdown method is to issue CLEAR EVENTS to terminate the event loop. The Shutdown method in myFramework is also set up to ask the user if they want to terminate the application, if the value of the application object's lPromptOnExit property is True. The idea here is that the user would have a way, perhaps via a check box in an Options dialog, to set the value of lPromptOnExit True or False according to their personal preference. The Shutdown method code is shown in Listing 12.

Listing 12: The application object's Shutdown method is responsible for terminating the event loop by issuing CLEAR EVENTS.

```
PROCEDURE Shutdown() as VOID
IF THIS.lPromptOnExit = .T.
   IF MESSAGEBOX( "Do you want to quit now?", ;
                  MB_YESNO + MB_ICONQUESTION, "Exit Application") <> IDYES
      RETURN
   ENDIF
ENDIF
ON SHUTDOWN  && Clear ON SHUTDOWN command to avoid infinite loop
this.CleanUp()
CLEAR EVENTS
ENDPROC && Shutdown
```

As you can see, the Shutdown method in myFramework calls a method named CleanUp, which as its name implies is responsible for cleaning up the application environment before shutting down. If the application is running in development mode—that is, if the developer is running it in the VFP IDE during testing and debugging—the CleanUp method calls a method named DevEOJ which does a few additional things to restore the developer's environment before shutting down the app. You can see the CleanUp, DevEOJ, and all the other mthods of the sample application object in the session downloads. In actual practice, there are probably going to be more things you want to do in CleanUp and DevEOJ than are shown in the sample code.

The framework-level application class code also contains several abstract methods designed to be implemented in the application subclass. These include the CreateAppObjects method that instantiates the framework manager classes for forms, menus, and reports. This code needs to be implemented in the application subclass because these objects are instantiated from application-level subclasses whose names are set up at runtime.

The code for myAppCls.prg is too long to reproduce here, but is included in its entirety in the session downloads so you can inspect it and learn how it operates.

Constructing a forms manager class

The primary purpose of a forms manager class is to facilitate the instantiation, display, and release of form objects. A forms manager may also be used to coordinate these events with other objects in the framework that need to aware that a form has been instantiated, released, or has changed state.

In Visual FoxPro, forms can be defined as form files (.scx) or as classes in a visual class library (.vcx). Forms defined in form files are instantiated with the DO FORM command, while forms defined in a class library are instantiated with CREATEOBJECT or NEWOBJECT. A forms manager can encapsulate both behaviors and expose them to the application through a single interface such as a DoForm method.

The forms manager class for the sample framework used in this session does just that. The code is too long to reproduce in the body of this paper, but the entire forms manager class is included in the session downloads as myFormMgr.prg.

The forms manager uses a forms collection to keep track of which forms are in use at any given time. Each entry in the forms collection holds information about one particular form. When the form manager's DoForm method is called, it checks to see if the form is already running. If not, the forms manager instantiates the form and adds it to the forms collection by calling the AddToFormsCollection method. If the form is already running, the form manager calls the form object's Show method. In this framework, only one instance of a given form is allowed to run at the same time.

In the sample framework for this session, the forms collection is stored in an array property of the forms manager class.[2] Each row of the array holds five pieces of information about the form: the form's name, its object name, whether it is .scx based or .vcx based, how many parameters are accommodated in its Init method, and the menu bar name associated with the form on the Window pad of the application's main menu.

In this sample framework, forms that interact with the forms manager need to be registered with the application by the developer. An embedded table named FormList.dbf is used for this purpose. As shown in Table 1, the FormList table has columns for each of the five elements of information in the forms collection.

Table 1: The FormList.DBF table stores information about registered forms in the application.

Field Name	Type	Size	Description
cFormName	C	50	Name of the form (e.g., frmCustomers)
cObjName	C	50	Object name of the form at runtime (e.g., ofrmCustomers)
cBarName	C	20	Name to appear on the Window menu pad (e.g., Customers)
nNbrParms	N	2,0	Number of parameters accommodated in the form's Init method
lClassForm	L	1	True for .vcx-based forms, False for .scx-based forms

[2] If you are using VFP 8.0 or later, a collection object might be a better choice than an array. The forms manager class used in the sample framework for this session is derived from a class I created long before the collection class was introduced in VFP 8.0.

The developer registers a form by adding a row with the appropriate values to the FormList table. Figure 7 shows the FormList table entry for a sample form named frmDummy. The form's object name is ofrmDummy, its menu bar name is Dummy Form, it takes zero parameters, and it is not a class-based form.

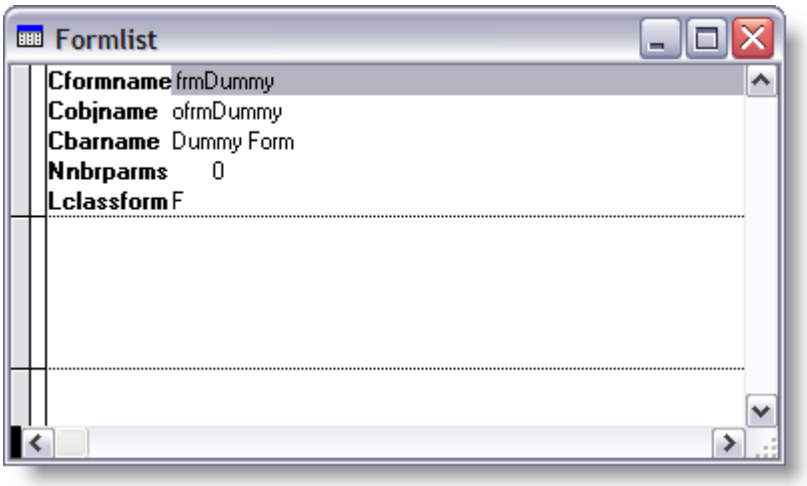

Figure 7: The FormList table entry for a sample form named frmDummy.

The forms manager class also has a ReleaseForm method. The purpose of the ReleaseForm method is to remove the form from the forms collection, which is done by a call to the form manager's RemoveFromFormsCollection method.

Even if you place a command button prominently labeled "Close" on a form, users won't always use it to close the form. A form can also be closed in other ways, such as by clicking the standard Windows Close button on the form's title bar. For this reason, you can't rely on code in a command button or any other individual control to notify the forms manager object that the form is being released. Instead, that responsibility belongs to the form itself. That's why the Destroy event method of the framework-aware forms class used in this framework includes a call to the form manager's ReleaseForm method, as shown earlier in Listing 9.

The forms manager also handles other functions such as a method to check whether a form is registered in the FormList table, a method to close all open forms (useful when the application is being shut down), and methods to interact with the menu manager class in order to add the name of a registered form to the Window menu pad when the form is instantiated and remove it when the form is released.

Constructing a menu manager class

Although Visual FoxPro became an object-oriented development tool when VFP 3.0 superseded FoxPro 2.x, and although VFP has continued to evolve in that direction ever since, menus remain essentially unchanged and are still purely procedural. Over the years, developers have come up with several approaches to improve on the native VFP Menu Designer in attempts to move menu design in a more data-driven and object-oriented direction. One such current project is the VFPX Menu Project, available for download from CodePlex at http://www.codeplex.com/VFPX.

Unless you choose to incorporate a replacement approach to menus in your framework, you're stuck with the native VFP approach to designing menus and programmatically manipulating them at runtime. Because menus are not classes, you can't subclass them to implement specific behaviors. For the framework designer, this means supplying a basic menu and copying it (or creating it from code) into the generated project for every application.

What sorts of things should a framework menu manager be expected to accomplish? One useful function is to add the name of a form to the Window menu pad when the form is opened, and remove it when the form is closed. In the sample framework for this session, this functionality is implemented by a coordinated effort between the forms manager and the menu manager.

When the forms manager adds a form to its forms collection, it makes a call to the menu manager's AddToWindowPad method. Using parameters, the forms manager passes the menu manager the information necessary for the menu manager to set up a bar on the Window menu for the newly opened form. Similarly, when the forms manager removes a form from its forms collection, it makes a call to the menu manager's RemoveFromMenuPad method.

Another function the menu manager can handle is removing bars or pads from the main menu under certain conditions. For example, I like to have a Developer menu pad available when I'm testing and debugging compiled apps from with the Visual FoxPro IDE. The Developer menu gives me access to tools such as the Debugger, the Class Browser, the Data Session window, and the Command Window while the application is running. Of course, these tools aren't available to the end user so the Developer pad should be removed if the app is not running in the VFP IDE.

Similarly, there may be individual bars on a menu pad that are available to some users but not all users, perhaps depending on their access level. One example might be a menu item giving the user the ability to add new users or edit other existing users' passwords or permissions. In my opinion, if a user does not have access to the functionality provided via a certain menu bar it's preferable to remove that bar entirely rather than merely to disable it. A disabled menu bar still shows on the menu pad and serves to let the unauthorized user know there is a function they can't get to, whereas removing the bar hides the existence of that functionality from the unauthorized user. The menu manager can provide a method to remove a bar from a menu pad based on its bar name.

Although working with native VFP menu pads and bars is not as intuitive as it could be, the code for the menu manager class is fairly straightforward. The menu manager class for the sample framework is included in the session downloads as myMenuMgr.prg.

Constructing a reports manager class

The primary job of the reports manager class is to encapsulate the logic necessary to print and/or preview a report. In the same way the forms manager provides a DoForm method to hide the differences between instantiating a vcx-based from and an scx-based form from the rest of the application, the reports manager in this sample framework provides a ShowReport method that takes care of rendering a report in the manner specified by the user.

In the sample framework, the report manager's ShowReport method takes three parameters: the name of the report, a flag indicating whether or not to prompt the user to print after preview, and a flag indicating whether to run the report as a summary or not. In addition, the ShowReport

method checks the application object's lPrintPreview property to know whether or not to show the report in the preview window or send it directly to the printer. Finally, and this is a design decision you may or may want to use in your own apps, the ShowReport method always prompts the user for the desired printer before printing.

The report manager class also includes a NoReport method to gracefully handle the situation where there is nothing to report, for example if the cursor or table driving the report is empty. In the sample framework this is a simple message box telling the user there is nothing to report. In the design of the sample framework, responsibility for calling the NoReport method instead of the ShowReport method rests with the calling object.

The report manager class in the sample framework also includes a way to set an error message in a property of the class and return it to a calling object. Although not implemented in the sample application, this is an example of a technique that's useful in any class that may or may not be allowed to communicate via a user interface element such as a message box. For example, in a Web application reports could be generated as PDF files instead of to the screen or printer, and any errors would have to be handled by the calling method so it could notify the user via a response to their browser.

The reports manager class for the sample framework is included in the session downloads as myRptMgr.prg.

How to tie it all together

An application class is commonly used as the highest level class in the framework hierarchy. At runtime, the application object is the glue that ties the rest of the objects together into a working application. In addition to many other responsibilities, the application object creates instances of the application-level subclasses derived from the framework-level base classes.

In order to preserve the class hierarchy with the application object at the top, references to all objects created by the application object are held in properties of the application object rather than being created at the top level. For example, the application object's oFormMgr property holds a reference to the forms manager object and its oMenuMgr property holds a reference to the menu manager object.

Because the application object is the first to be instantiated and the last to be released, the references to the application-level delegate objects persist from the time the objects are instantiated at application startup until the application releases them when it terminates. The application object name (oApp) is defined as public in the main program in order to keep it in scope throughout the entire application. This means code anywhere in the application can call a method on oApp or reference any of the framework-class instances such as oApp.oFormsMgr, oApp.oMenuMgr, and so on. Although the use of public variables is generally discouraged, it's customary to define the application object as public for this reason. In the sample framework, oApp is the only public variable.

Creating the application class libraries

Once the framework base classes and subclasses have been created, they could theoretically be used directly in an application. However, it's highly advisable to create another layer of abstraction between the framework classes and the classes used by an application based on that framework. This is done by subclassing the framework classes into application-level class libraries.

Subclassing the framework base classes into application-specific class libraries allows the developer to modify the appearance and behavior of the application-level classes for one application separately from any modifications that might be made for another application. For example, Application A might use a cyan background for the read-only text box, while Application B might use a pale yellow background for the same control. This type of change, of course, should be made at the application level rather than the framework level.

The application-level classes for all applications are subclassed from the framework classes, but every application derived from the framework gets its own application-level class libraries. In addition to ensuring that changes made to one application don't affect another one, this also preserves the class hierarchy back to the framework base classes and ensures that any changes to the framework classes are inherited by each application.

Creating the application-level class libraries from the framework class libraries is a task for the framework's application generator. I did not build a complete application generator for the sample framework in this session, but the code in Listing 13 shows one way the application-level class libraries can be programmatically generated from the framework class libraries.

In the design of this framework, each application is assigned a one to three character prefix. For example, I chose the prefix "swf" (short for Southwest Fox) for the sample application for this session. The code in Listing 13 expects this prefix as a parameter, and prompts for the directory in which the new class library is to be created.

The code to create the application-level subclasses from the framework classes is essentially identical to the code to create the framework classes from the VFP base classes. However, because the application-level classes are derived from the framework base classes instead of the VFP base classes, each CREATE CLASS statement must include a FROM clause identifying the class library where the parent class is located. In addition, the name of each class and class library at the application level begins with the application's prefix. For example, the application-level class library for controls is named swf_Controls.vcx, and the application-level text box base class is named swf_textbox. This naming convention is arbitrary, but using a naming convetion like this helps you remember which class you're working with during development.

Listing 13 shows the beginning part of the code to generate the application-level subclasses. The program is too long to reproduce all of it here, but the entire program is included in the session downloads.

Listing 13: This program creates the application-level subclasses derived from the framework base classes.

```
*==============================================================================
* Program:      CREATEAPPCLASSES.PRG
* Author:       Rick Borup
* Date Written: 09/05/2007
* Copyright:    (c) 2007 Information Technology Associates
```

```
*                   All rights reserved.
* Compiler:     Visual FoxPro 09.00.0000.3504 for Windows
* Abstract:     Create application-level classes from the framework classes.
* Environment in:
* Environment out:
* Parameters:
* Returns:
* Changes:
*============================================================================
#INCLUDE foxpro.h

LPARAMETERS tcAppPrefix
IF VARTYPE( tcAppPrefix) <> "C" OR ;
   NOT BETWEEN( LEN( ALLTRIM( tcAppPrefix)), 1, 3) OR ;
   NOT ISALPHA( tcAppPrefix)
   MESSAGEBOX( "The application's prefix must be 1 to 3 alpha characters.", ;
               mb_iconstop, "Create application-level base classes")
   RETURN
ENDIF
Local lcAppPrefix
lcAppPrefix = LOWER( ALLTRIM( tcAppPrefix)) + "_"

LOCAL lcAppDir, lcAppClassDir
lcAppDir = GETDIR("", "Please select the new application's root directory", ;
           "Create Application Classes", 64)
IF EMPTY( lcAppDir)
   MESSAGEBOX( "App directory must be specified.", ;
               mb_iconstop, "Create application-level base classes")
   RETURN
ENDIF
lcAppClassDir = ADDBS( lcAppDir) + "Classes"

*  Application-level control classes
#DEFINE _BaseLibrary ".\Classes\myBaseCtrl.vcx"
#DEFINE _AppLibrary ADDBS( lcAppClassDir) + lcAppPrefix + "Controls.vcx"

CREATE CLASSLIB _AppLibrary
CREATE CLASS &lcAppPrefix.Checkbox OF _AppLibrary as chkBase ;
   FROM _BaseLibrary NOWAIT
CREATE CLASS &lcAppPrefix.ComboBox OF _AppLibrary as cboBase ;
   FROM _BaseLibrary NOWAIT
CREATE CLASS &lcAppPrefix.CommandButton OF _AppLibrary as cmdBase ;
   FROM _BaseLibrary NOWAIT

*** And so on for the rest of the classes.
```

This code is intended to be run by the developer (or the application generator) once for each application being generated from the framework. As with the code that was used to generate the framework base classes, you will need to close the class designer windows from the IDE after this code finishes running.

The base application

A framework typically includes an application generator. The developer runs the application generator once for each new application. The application generator creates a base application for the new project. The base application, which is created in a directory of the developer's choice, consists of a VFP project file along with subclasses or copies of the resources furnished by the

framework. As noted earlier in the section on the VFP application wizard, the base application can generally be compiled into a working executable, although it does nothing useful until the developer adds the application-specific components to create a solution for a particular need.

The sample framework presented in this session does not include an application generator. Instead, the base application is included in its entirety in the session downloads as SampleApp. You can open the project file SampleApp.pjx and compile the application yourself, or you can run SampleApp.exe which has already been compiled for you. Like all examples in this session, the base application was written and compiled in VFP 9.0 SP1.

With the base application open in the VFP Project Manager, it's easy to see how the framework-level base classes and the application-level subclasses come into play. The value of using a naming convention that includes a prefix in front of each element's name becomes apparent here. In Figures 8 and 9, for example, you can easily see which class libraries, classes and program files belong to the framework and which belong to the application: Everything with a "my" prefix is the part of the framework, while everything with an "swf" prefix is a subclass belonging to the application.

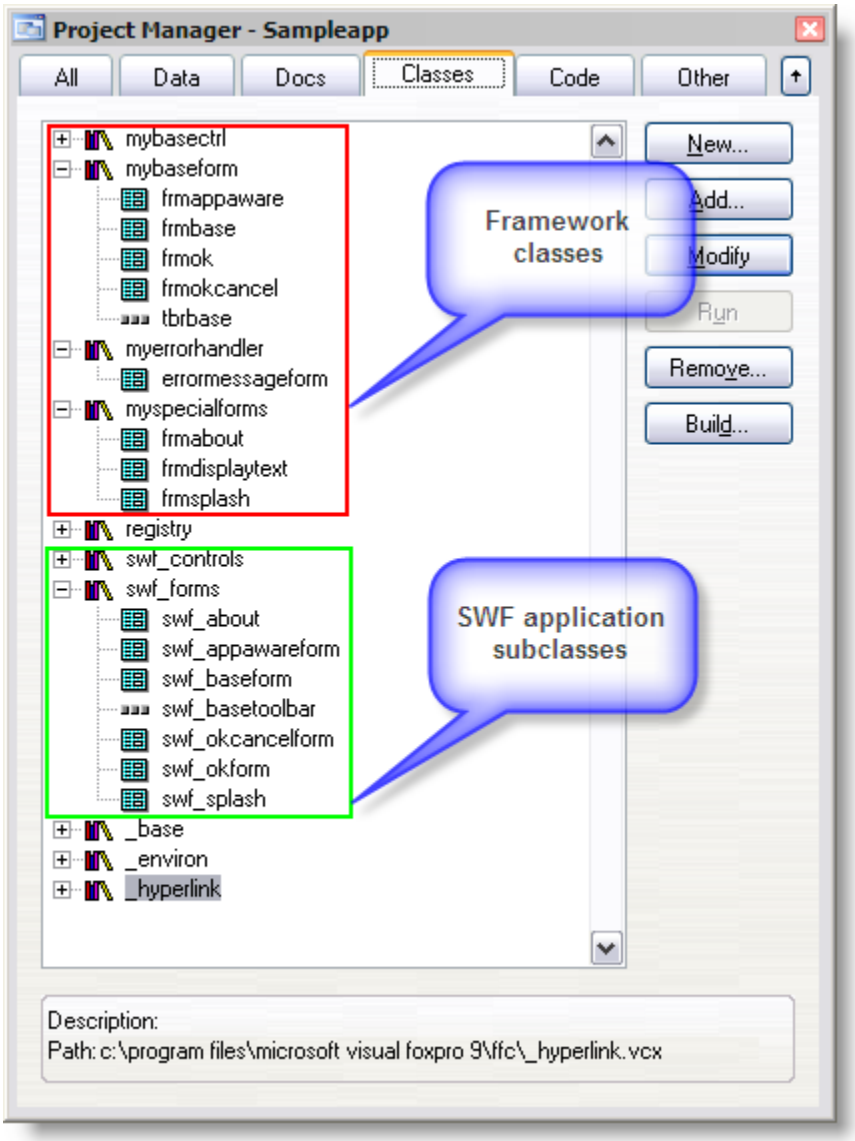

Figure 8: Classes and class libraries prefixed with "my" belong to the framework, while those prefixed with "swf" belong to the application. The ones without prefixes are VFP foundation classes.

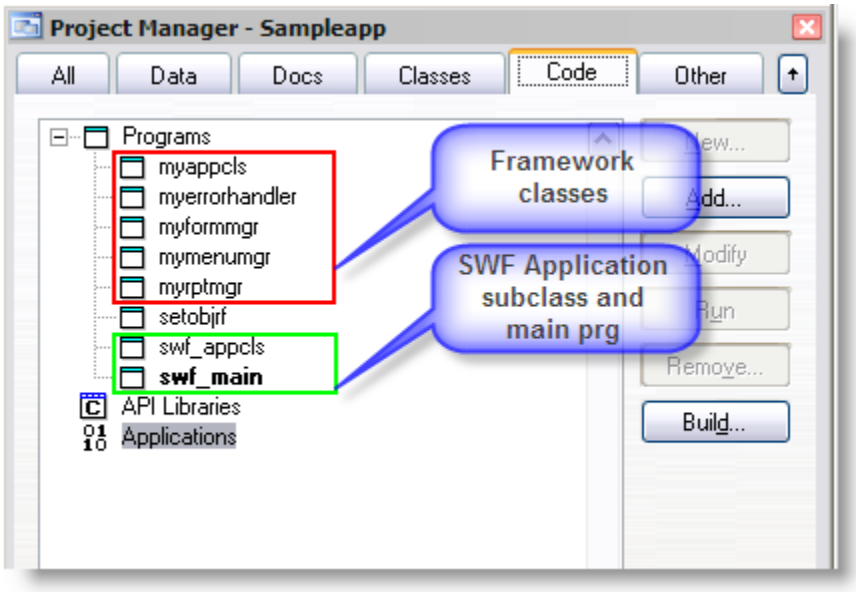

Figure 9: Prg-based classes prefixed with "my" belong to the framework, while those prefixed with "swf" belong to the application. The one without a prefix (setobjrf) is a VFP foundation class.

Customizing the base application

Once the base application has been generated you can begin to customize it for the particular solution you're creating. In this sample framework, there are five things you should do to give the application its own identity. These are:

1. Create an icon for the screen's title bar;
2. Create an image for the Splash and About forms;
3. If desired, override any default values such as print preview and prompt on exit in the InitProps method of the application object subclass; and
4. Fill in the information in the application control table appinfo.dbf (see below for details);
5. Compile the application into an executable (.exe) with a version number.

After doing these five things, you have a base application that's ready to run. Then it's on to the real work of adding the forms, custom menu items, reports, data, and other elements necessary to complete the solution.

The sample framework uses a free table called appinfo.dbf to hold many of the variable values unique to each particular application. This is simply a convenient way for the developer to store these values. Because they are always required and are never changed by the user, the appinfo table is included in the EXE rather than being deployed as a separate file.

The application object reads the values from the appinfo table and stores them in properties at application startup time – see method GetAppInfo in framework class myAppCls.prg. Other methods and objects in the application can then reference these properties as necessary. For example, the Splash and About forms pull some of the information they display from properties of the application object.

Figure 10 shows the values in the appinfo table for the sample app, along with annotations indicating what some of them are used for. See the source code for the sample application for more detail.

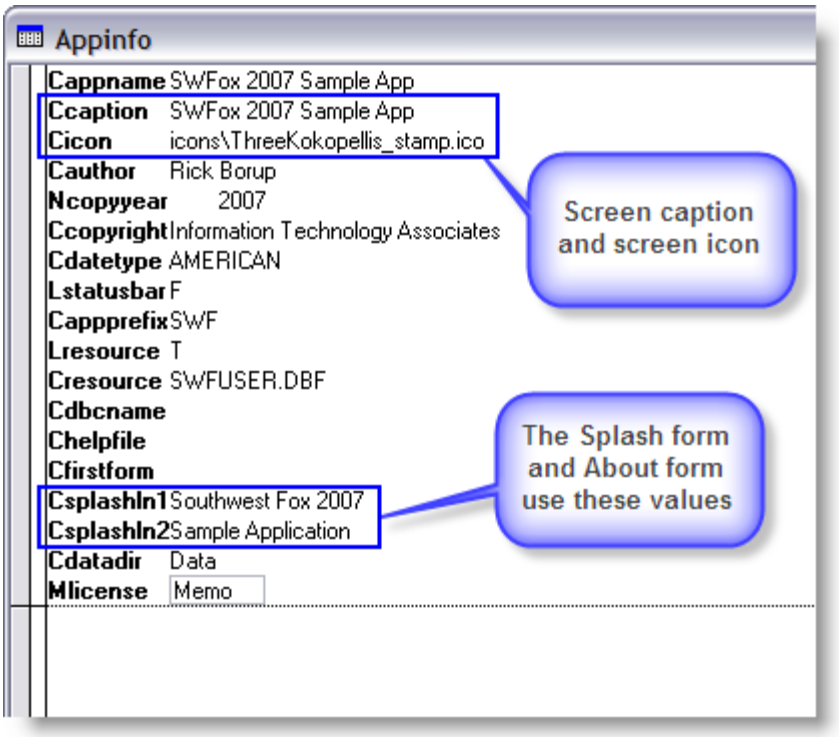

Figure 10: The appinfo table stores values specific to the application.

Figure 11 shows the sample application with the About form showing. At this point, nothing has been changed in the base app except for the five things referenced above. Note the application already has an identity (title, icon, version number, build date, etc.) and already has a "finished" appearance even before a single form, report, database, or other element of the actual solution has been added.

Figure 11: After just a bit of customization, the base application already has a finished appearance.

Adding forms, reports, and data

The final step is to add the forms, reports, data, and other elements necessary to create a real solution from the base application. The beauty of using a framework is that it takes only a few minutes to create the base application and establish its unique identity, as show in Figure 11. The developer can now focus on designing and building the actual solution desired by the client, using the classes and capabilities provided by the framework as the foundation. Commercial frameworks typically include wizards and builders to make the job even easier.

In the sample application, I added a few solution-specific elements to illustrate the point. The sample application uses the Customers table from Northwind database included with Visual FoxPro. The Northwind database is located in the Samples subdirectory under the Visual FoxPro home directory on your computer.

I built a simple form to display the Northwind Customers table in a grid. The form has a print button that generates a report of the Customers data. Following are the steps involved in creating the form and the report, and hooking them into the application.

Create an app-aware form

The first step is to create a form to show the data in the Customers table. Because this is a non-modal form, it should show up on the Window menu pad when it's open and should therefore be based on the "application aware" form subclass. Assuming the current default directory is the root directory for the sample application project, the customers form can be created with the following command:

```
CREATE FORM forms\frmCustomers as swf_appawareform FROM classes\swf_forms.vcx
```

VFP automatically opens the new form in the Forms Designer so you can modify its properties and methods. The custom property values for the Customers form are shown in Figure 12.

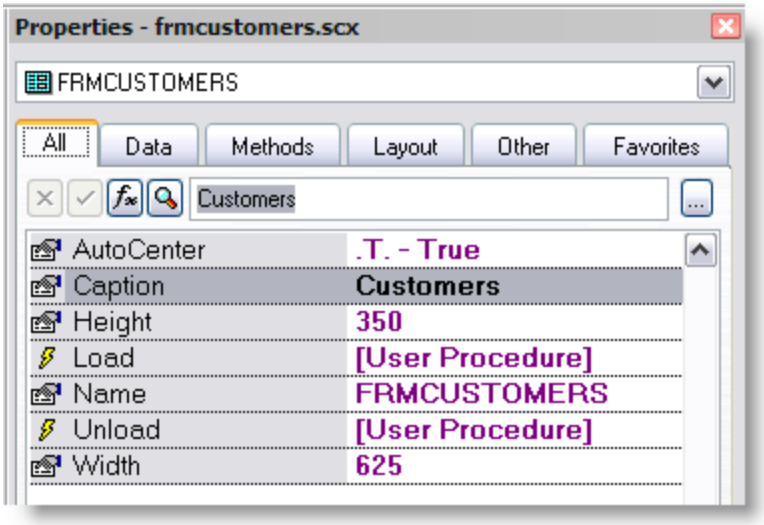

Figure 12: Name the customers form "frmCustomers" and set other custom properties as shown.

Because there are so many types of data sources and so many ways to interact with data in a VFP app, a full-fledged framework would typically come with a rich set of classes to enable data access. To keep things simple, the sample framework uses the Northwind database as local data, which is opened in the OpenDBC method of the application object at startup time. Therefore the Load event method of the Customers form simply needs to open the Customers table of the Northwind database.

Listing 14: The Load event method of the Customers form opens the Northwind Customers table.

```
IF NOT USED( "Customers")
   USE customers IN 0 AGAIN ALIAS customers SHARED
ENDIF
```

To display the data, add a grid to the form using the swf_grid class from the swf_Controls.vcx class library. Set the ControlSource property of the grid columns to the desired columns of the Customers table, and set the grid's Header captions accordingly.

Below the grid, add a command button based on the swf_CommandButton class and set its caption to "Print". In the button's Click event method, hook into the framework by calling the report manager's ShowReport method and passing the name of the desired report:

```
oApp.oRptMgr.ShowReport( "rptCustomers")
```

The finished form is shown in Figure 13.

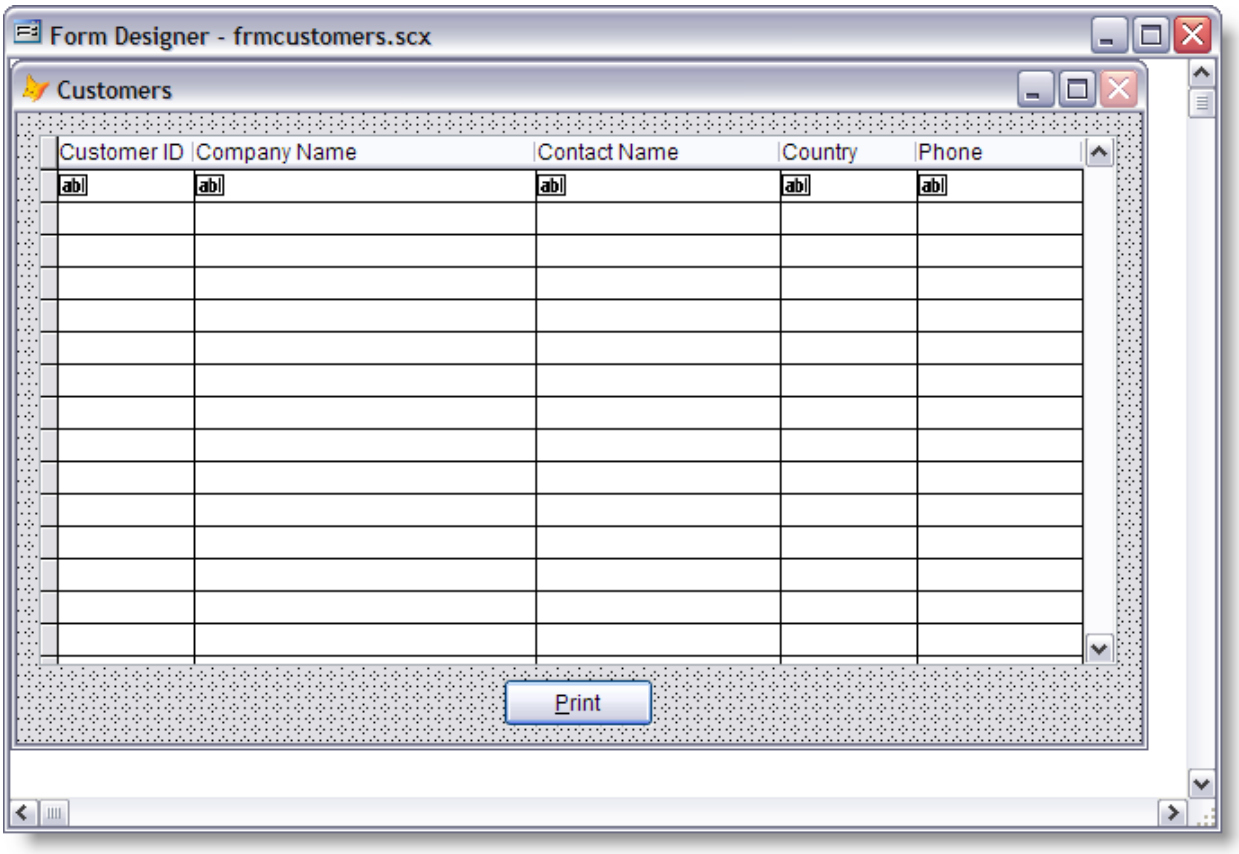

Figure 13: The Customers form is built from application-level subclasses, which hooks into the framework.

Register the form with the framework

In this sample framework, all forms that interact with the forms manager need to be registered in the FormList table. The Customers form is registered by inserting a row into the FormList table, as shown in Figure 14. Note that the form name registered in the FormList table (frmCustomers, in this case) must be the same as the Name property of the form and, for .scx-based forms, the same as the name of the .scx file.

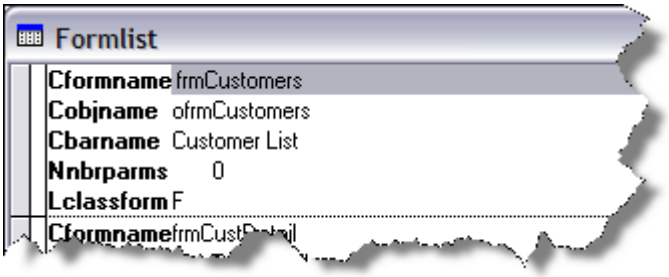

Figure 14: The Customers form is registered with the framework via a row in the formlist table.

The value of the bar name column (cBarName) is the name added to the Window pad when the form is open.

Add a menu item

The user needs a way to launch the Customers form. In the sample application, I edited the main menu and added an item named Open to the File menu. The Open menu has a submenu named Customers, which runs a procedure to hook into the framework's forms manager. The menu code is shown in Listing 15

Listing 15: Code in the Customers menu bar procedure launches the Customers form using the framework's forms manager.

```
oApp.oFormMgr.DoForm( "frmCustomers")
```

Note that this same code works whether frmCustomers is a .vcx-based or .scx-based form. The forms manager object looks up frmCustomers in the FormList table and uses the value of the lClassForm property to instantiate the form in the proper way.

Create a report

The report for the sample app is just a bare-bones report I created using VFP's Quick Report dialog. In a real application you would most likely spend time creating a much more useful and attractive report than this. However, regardless of how simple or how complex, all reports are launched in the same way by using the report manager's ShowReport method.

Note that the sample framework does not implement any of the functionality offered by the enhancements to the report writer in VFP 9.0. A real framework would typically provide methods to take advantage of these great new features.

Build the app and go!

That's all there is to it: With a few simple steps, you've added a form and a report to the sample application and hooked them into the framework using the forms manager and the report manager. You're now ready to build the app and run it.

At runtime, other useful aspects of the framework becomes apparent. For example, when you open the Customers form it's automatically added to the Window menu using the bar name specified in the formlist table, as show in Figure 15.

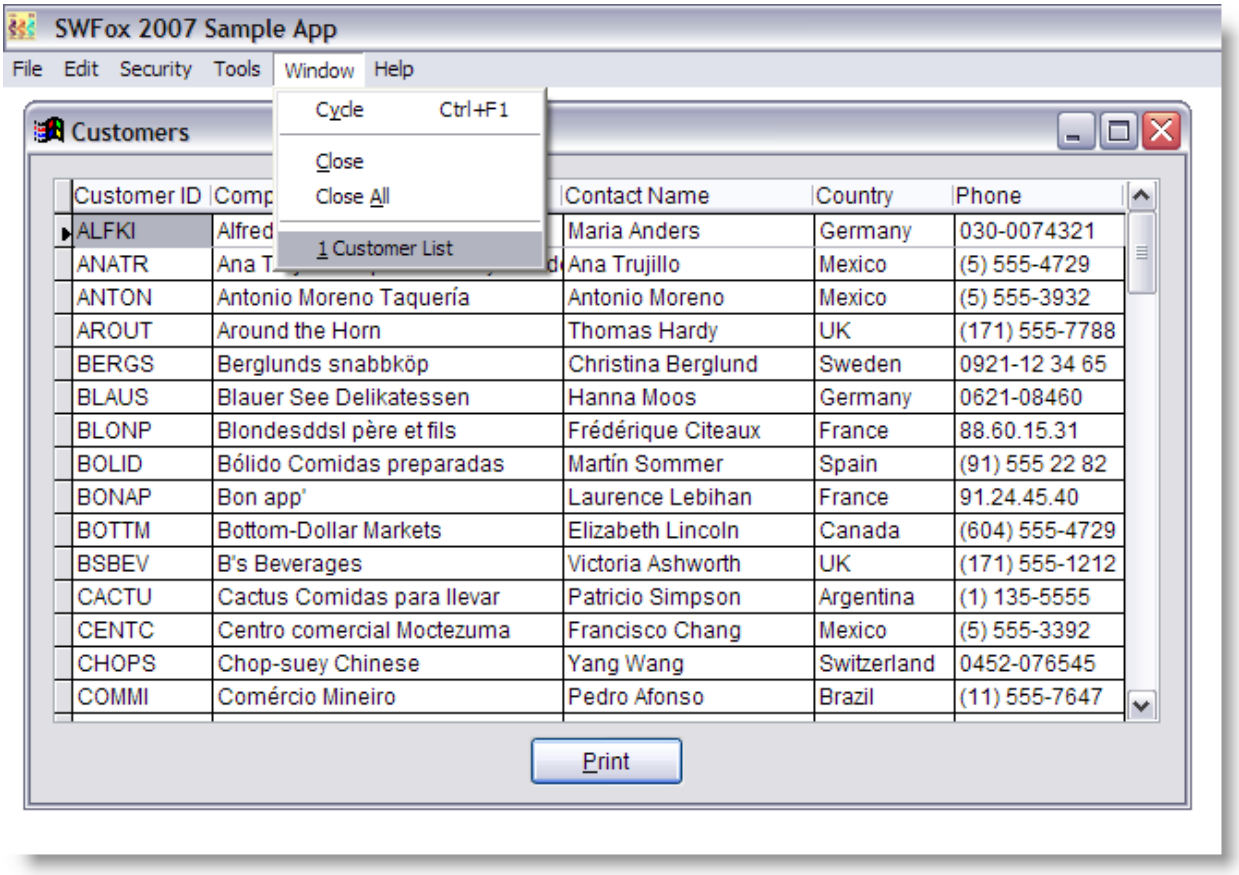

Figure 15: When the Customers form is open, the framework adds its name to the Window menu pad.

Note that *Customer List* is preceded by the number 1 on the Window menu pad. This number is added automatically by the framework based on how many registered forms are open and the order in which they were opened. When a second form is opened it is assigned number 2, and so on.

To illustrate this, I created a second form called Customer Detail. In the sample app, you can open the Customer Detail form by double-clicking on a customer ID in the Customer List form. As you can see in Figure 16, the Customer Detail form is assigned number 2 in the list of open forms on the Window pad.

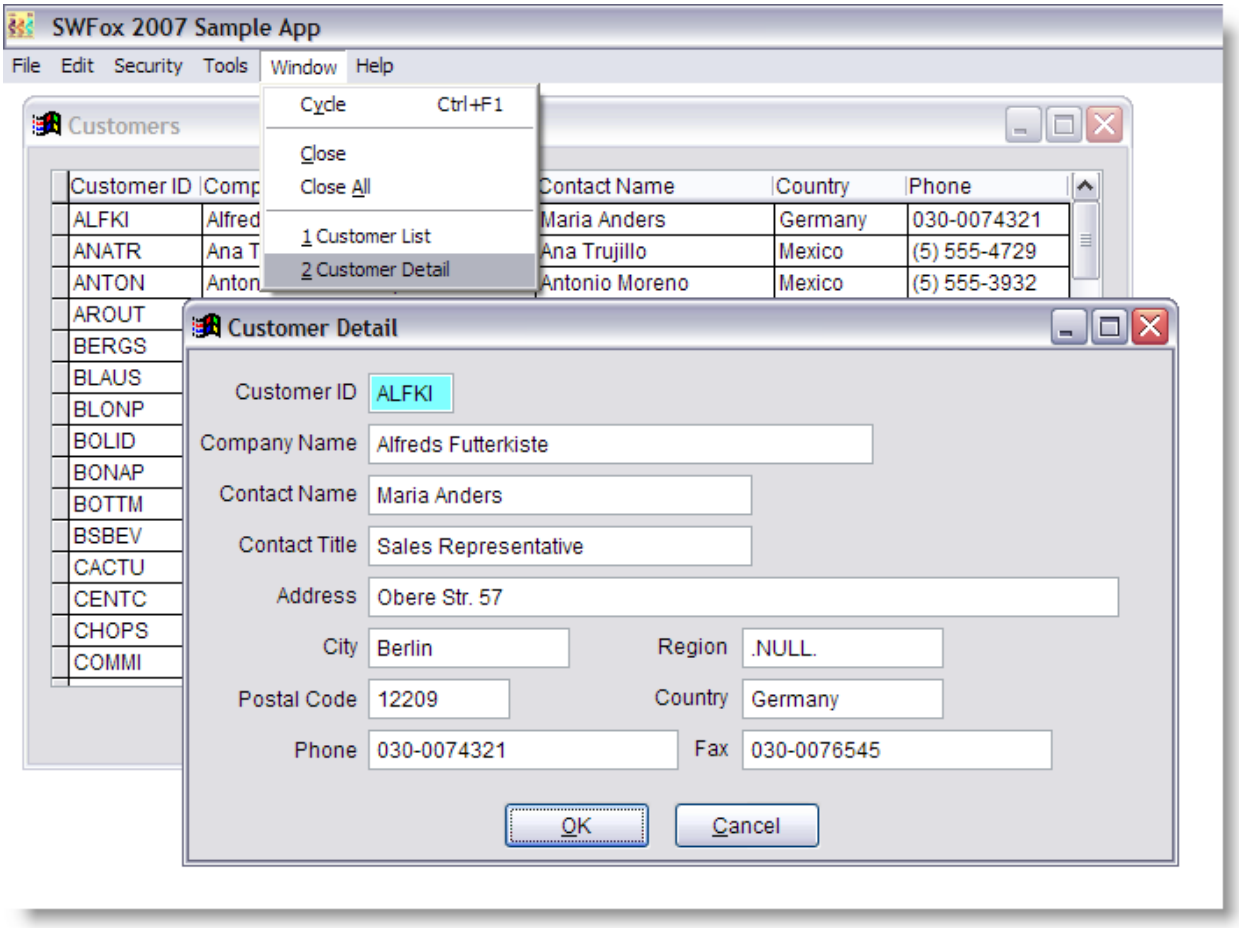

Figure 16: The Customer Detail form is the second form to be opened so the framework assigns it number 2 in the Window menu pad.

The Customer Detail form also illustrates the use of the framework's display-only text box class, which is used for the Customer ID. The cyan back color is a visual cue to the user that the value in the text box is read-only. The other controls on the form are based on the standard application-level subclasses in the swf_controls.vcx class library.

For demo purposes, the OK and Cancel buttons on the Customer Detail form do nothing but release the form. Because the text boxes on this form are sourced directly from the Northwind Customers table, any changes you make in the form will be written to the Northwind database regardless of whether you click OK or Cancel.

When a registered form is closed, its name is removed from the Window menu pad and the remaining forms are renumbered. For example, if the Customer List form is closed while the Customer Detail form is still open, the Customer Detail form reverts to number 1 in the list. All of this functionality is automatically supplied by the framework for forms based on the AppAware subclass.

Other helpful framework classes

The classes included in the sample framework represent some of the essential functionality typically provided by any framework. One of the biggest pieces I've left out of the sample framework is any kind of data handling classes. Because of the diversity of data sources and the many ways VFP can access data, that topic merits its own session, if not an entire book, all by itself. Another major functional piece typically provided by a framework, particularly in an n-tier design, is the concept of a middle tier business object which coordinates with the user interface on one side and with the data handling layer on the other.

Frameworks can also include a wide variety of other classes and methods to help the developer incorporate commonly used functionality in an application. A few examples are classes to facilitate office automation, a set of progress bar classes for various uses, a security module to control usernames and passwords and enable different levels of access to the application's functions by user, and the ability to run external applications for example by using ShellExecute.

Final Thoughts

A good framework is a valuable tool. Learning how to use one helps you build applications quickly and easily. A commercial framework typically requires a significant learning curve but pays off with a rich variety of classes and utilities to use when building applications. Creating a basic framework of your own is a useful learning experience, as I hope this session has shown.

If you're inspired to go on and create a more complete framework of your own, great. On the other hand, if you're more inclined to explore what the commercial frameworks have to offer, I hope this session has helped you understand what a framework is all about and what kinds of things you can expect it to do.

Resources

Following is a list of some commercial frameworks for Visual FoxPro. The lists are in alphabetical order, with no bias toward any particular one of them intended or implied.

Commercial frameworks for Visual FoxPro

Codebook Framework for Visual FoxPro. Open Source. http://sourceforge.net/projects/codebook

Mere Mortals VFP Framework. Oak Leaf Enterprises. http://www.oakleafsd.com

Visual FoxExpress®. F1 Technologies®. http://www.f1tech.com/VFE

Visual MaxFrame Professional. Visionpace. http://www.visionpace.com/vmpsite

Visual ProMatrix. ProMatrix Corporation. http://www.promatrix.com

Web-oriented frameworks

Active FoxPro Pages. ProLib Software GmbH (Germany). http://www.afpages.de

ActiveVFP. dotComSolution. http://www.activevfp.com

West Wind Web Connection. West Wind Technologies. http://www.west-wind.com

Acknowledgments and Copyrights