



Creating Dynamic Object-Oriented Reports and Menus

Doug Hennig

Stonefield Software Inc.

Email: dhennig@stonefield.com

Corporate Web site: www.stonefieldquery.com

Personal Web site : www.DougHennig.com

Blog: DougHennig.BlogSpot.com

Twitter: [DougHennig](https://twitter.com/DougHennig)

Reports and menus are two components in VFP for which there is no object-oriented access. And yet an OOP approach to both would make them much easier to work with, especially if you want to dynamically create or make changes to them at runtime. I created object-oriented wrappers for both reports and menus and have used them successfully for many years. In this session, I'll show you how to use these wrappers to provide cool features to your applications, such as dynamic menus that provide control for security purposes and dynamic reports that output the contents of a grid or form.

Introduction

Although VFP is a great object-oriented development language, there are two glaring areas of the product that have never been objectified: reports and menus. As they are out of the box, neither can be easily changed dynamically at runtime nor generated programmatically. As we will see in this document, object-oriented wrappers for menus and reports allow you to do many useful things that are difficult to do otherwise.

Object-oriented menus

Menus in VFP are defined using commands like `DEFINE MENU`, `DEFINE PAD`, `DEFINE BAR`, and so on. We'll look at these commands in the next section. However, most developers don't use these commands. Instead, they use the Menu Designer to design a menu and store the design in an MNX file. The Generate function in the Menu Designer and the Build function in the Project Manager generate an MPR file from an MNX file. An MPR is essentially a PRG file that creates a menu using the various `DEFINE` commands.

Having a hard-coded generated MPR file that defines an application's menu is far from ideal. Here are a few reasons:

- You have to use kludges to add or remove bars dynamically such as having some bars or pads not appear for certain users.
- It's difficult to create a localized menu or allow the prompts for menu items to vary under certain circumstances.
- Menu bars aren't reusable, so you have to replicate menus between applications.

The object-oriented menu classes we'll look at in this document are a VFPX project, available at <https://github.com/VFPX/OOPMenu>.

How menus work in VFP

Before we look at the classes that implement an object-oriented menu, let's review how menus are handled in VFP. A menu bar appears below the title bar of an application. Although you can define a menu bar using `DEFINE MENU`, most developers use the VFP system menu bar, `_MSYSMENU`, because it's easier and that's what code generated by the Menu Designer does.

A menu bar consists of pads. A typical application has File, Edit, and Help pads at a minimum, but Tools, View, and Window pads are also common. Pads are created with the `DEFINE PAD` command. A pad has a prompt, a hot key (usually an Alt-key combination), a `SKIP FOR` clause (which, when it evaluates to `.T.`, disables the pad), and a message that appears in the status bar when the pad is highlighted. Pads can have other properties, such as font and color, but these are rarely used. You can specify what should happen when a pad is selected, but the standard behavior is to display a popup.

As with pads, popups can have a number of properties specified with the `DEFINE POPUP` command, but the two that are typically used are `MARGIN`, indicating that extra space is

available at the left margin of the popup so images or mark characters can be displayed, and RELATIVE, which means that items in the popup appear in the order in which they're defined.

Popups consist of bars, the items a user can select from the popup. A bar is created with the DEFINE BAR command, and has a number of properties you can specify, including the prompt, hot key and the text representing the hot key that should appear to the right of the prompt, a SKIP FOR clause, a message that appears in the status bar when the bar is highlighted, the name of a file containing the image to display to the left of the bar's prompt or the name of a system menu bar whose image should be used for the bar, whether the bar should appear inverted, and whether the bar should be an MRU (most recently used) bar (it appears as a chevron pointing downward). As with pads, other bar properties such as font and color are rarely used. You specify what should happen when the user selects a bar with the ON SELECTION BAR command.

Now let's look at the classes that implement object-oriented menus. As I mentioned earlier, some of the options VFP provides for menus are rarely if ever used. I decided when I created these classes that I'd ignore options I never use or those that don't follow Window standards (many of these options are carry-overs from the DOS days). All the classes discussed in this article are located in SFMenu.vcx.

SFMenu

This class represents the menu bar. Like all classes in SFMenu.vcx except SFShortcutMenu, which I'll discuss later, it's a subclass of SFMenuBase, based on Collection. SFMenu has only a few custom properties:

- `cFormName`: specifies the name of a top-level form in which to put the menu. Leave this property blank to put the menu in the system menu bar.
- `cInstanceName`: contains the name of the variable this class is instantiated into (we'll see why we need this later). Although it's public, it has an assign method that makes it read-only to everything except methods of this class. When you instantiate SFMenu, pass the name of the variable as a parameter; the Init method sets `cInstanceName` to the parameter value. For example:

```
loMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'loMenu')
```

- `cMenuName`: the name of the menu. The default is `_msysmenu`.
- `lAddToSystemMenu`: .T. to add to the VFP system menu (that is, this is a partial menu that supplements the VFP system menu) or .F. (the default) to be a standalone menu.

The AddPad method adds a new pad to the menu. Pass the class for the pad, the library the class is contained in, and the name to assign to the pad. If you want the default class for pads (SFPad), you can just pass the name as the only parameter. My convention is to use the prompt with "Pad" as the suffix for the name (for example, "FilePad"). AddPad instantiates the specified class as a member of SFMenu and is also added to the collection

so pads can easily be enumerated. Here's an example that calls this method, specifying a subclass of SFPad named SFEditPad:

```
loMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')
```

Here's an example that use the default SFPad class:

```
loMenu.AddPad('FilePad')
```

You can now reference the new pads as loMenu.EditPad and loMenu.FilePad.

The Show method displays the menu. If no pads have been added to the menu, Show calls the DefineMenu method. That method is an abstract method in this class, but in a subclass, you could put a series of AddPad calls that add the desired pads to the menu; this makes the menu subclass self-contained (no outside code has to add pads to the menu). The code then calls the Show method of each pad in the menu and turns on menu handling. The code for Show is shown in **Listing 1**.

Listing 1. The Show method of SFMenu.

```
local lnI, ;
  loPad
with This

* If we're putting the menu in a top-level form, set cMenuName and define the
* menu in the specified form.

  if empty(.cFormName)
    push menu _MSYSMENU
    if not .lAddToSystemMenu
      set sysmenu to
    endif not .lAddToSystemMenu
  else
    if .cMenuName = '_msysmenu'
      .cMenuName = sys(2015)
    endif .cMenuName = '_msysmenu'
    define menu (.cMenuName) bar in (.cFormName)
  endif empty(.cFormName)

* Define the menu if it hasn't already been.

  if .Count = 0
    .DefineMenu()
  endif .Count = 0

* Show all the pads.

  for lnI = 1 to .Count
    loPad = .Item(lnI)
    loPad.Show()
  next lnI

* Display the menu.
```

```
    set sysmenu automatic
    activate menu (.cMenuName) nowait
endwith
```

The Refresh method refreshes the menu by activating it again; this is useful after you've displayed the menu, and then changed some of the bars or pads in the menu. The ReleaseMembers method, called when the object is destroyed (this is actually defined in SFCollection), releases all the pads and restores the VFP system menu bar if cMenuName is _msysmenu.

SFPad

SFPad is the parent class for all pads in a menu. Set the following properties:

- cCaption: set it to the prompt for the pad, such as "File."
- cKey: set it to the hot key for the pad, such as "Alt+F." If you don't set cKey, it defaults to Alt plus the letter following "\<" in the caption or the first letter of the caption if there is no "\<." If you don't want that behavior, set lAutoAssignHotKey to .F.
- cStatusBarText: set it to the message to display in the status bar, such as "Performs file functions."
- cSkipFor: set it to an expression that, when it evaluates to .T., disables the pad.
- lVisible: set it to .F. if the pad shouldn't currently be visible; you can later set it to .T. to display the pad.
- cPadPosition: set it to "before" or "after" another pad to define the pad's location in the menu.

Notice that in VFP menus, there's no direct relationship between pads and bars. Instead, bars belong to a popup and a popup is associated with a pad. I decided to simplify this in my design; I really couldn't see the need to expose popups at all (if you think about it, you'll realize that's how the VFP Menu Designer works too). So, bars belong to pads in this design.

To add a bar to a pad, call the AddBar method, passing the class for the bar, the library the class is contained in, the name to assign to the bar (I usually concatenate the pad prompt, the bar prompt, and "Bar" for the name, such as "FileExitBar"), and optionally the bar number (if you don't pass this, AddBar automatically assigns the next available bar number to it). As with the AddPad method of SFMenu, if you want to use the default class for bars, SFBar, skip the first two parameters and pass the name as the first one. AddBar instantiates the class as a member of SFPad and is also added to the collection so bars can easily be enumerated. The Init method of the bar class is passed the bar number, the name of the popup, the name of the bar, and .T. if the bar number was specified or .F. if AddBar assigned it. Here's an example that calls AddBar:

```
loMenu.FilePad.AddBar('SFBar', 'SFMenu.vcx', 'FileOpenBar')
```

Since that statement uses SFBar as the class, it can be simplified to:

```
loMenu.FilePad.AddBar('FileOpenBar')
```

You can now reference the new bar as loMenu.FilePad.FileOpenBar. To add a separator bar, call the AddSeparatorBar method (it doesn't accept any parameters).

The Show method displays the pad, its popup, and the bars in the popup. If the pad hasn't been defined yet (the protected lDefined property is .F.), Show calls the Define method to define the pad. If it has been defined but was then released, it calls DefinePopup to define the popup then DefinePad to define the pad (Define also calls both of those methods but performs other tasks as well). Define also calls AddBars. That method is an abstract method in this class, but in a subclass, you could put a series of AddBar calls that add the desired bars to the pad; this makes the pad subclass self-contained. The Show method then calls the either Show or Hide method of each bar in the pad (Hide if the bar is inverted, Show if not). Since this method is called from the Show method of SFMenu, you probably won't have to call this method directly unless you call the Hide method (discussed next). The code for Show is shown in **Listing 2**.

Listing 2. The Show method of SFPad.

```
local lnI, ;
      loBar
with This
      do case

* If the pad has never been defined, call Define.

          case not .lDefined
              .Define()

* If the pad has been defined but then released, call DefinePopup and
* DefinePad.

          case not .lPadDefined
              .DefinePopup()
              .DefinePad()
          endcase
          for lnI = 1 to .Count
              loBar = .Item(lnI)
              if loBar.lInvert
                  loBar.Hide()
              else
                  loBar.Show()
              endif loBar.lInvert
          next lnI
      endwith
```

The Hide method releases the pad and popup that underlay this class, and sets the lVisible and lPadDefined properties to .F. You can call this method to make a pad disappear, and then later call Show to make it reappear. Alternatively, you can set lVisible to .F. and then

later .T.; this property has an assign method that calls either Hide or Show, depending on the value you're setting it to.

The Refresh method refreshes the pad. Call this method to refresh the entire pad when changes have been made to things like pad and bar captions. You can also call the Refresh method of the menu or a bar to refresh the entire menu or just one bar.

The ReleaseMembers method, called when the object is destroyed, calls Hide to destroy the pad and popup.

SFEditPad is a subclass of SFPad. Its AddBars method adds bars for Undo, Redo, Cut, Copy, Paste, Clear, and Select All functions. I figured since every application has one of these pads, why bother reinventing the wheel each time?

Use SFPadCommand if you want a pad that does something when clicked, such as an Exit function that exits the application. I don't recommend this because users expect a dropdown of choice when they click a pad, but if you really want it, it's available. It has the same properties as SFBar (discussed next), such as cOnClickCommand that specifies the code to execute when the pad is selected.

SFBar

SFBar is the parent class for all bars in a menu. Set the following properties:

- cCaption: set it to the prompt for the bar, such as "Open..."
- cKey and cKeyText: set these to the hot key and the text for it, such as "Ctrl+F" for both. If you don't set cKey but set lAutoAssignHotKey to .T. (the default is .F.), the hot key defaults to Ctrl plus the letter following "\<" in the caption.
- cStatusBarText: set it to the message to display in the status bar, such as "Open a document."
- cSystemBar: if you want to use a VFP system bar, set cSystemBar to the name of the bar. For example, the SFHelpTopicsBar subclass has cSystemBar set to "_MST_HPSCHE" so it automatically has the functionality of that system bar.
- cSkipFor: to conditionally disable the bar, you can either set cSkipFor to an expression or put code in the Allow method that returns .T. if the user is allowed to select the bar. You can also set lEnabled to .F. to unconditionally disable the bar.
- lVisible: set it to .F. if the bar shouldn't currently be visible; you can later set it to .T. to display the bar.
- lMarked: set it to .T. to display a mark, such as a checkmark, beside the bar.

There are five ways you can define what happens when the user selects the bar:

- Set cActiveFormMethod to the name of the method of _screen.ActiveForm to call (such as "Find;," you can specify parameters in parentheses if necessary).

- Set `cAppObjectMethod` to the name of the method of an application object to call if you use such as object and it's instantiated into a global variable whose name is in the `cAppObjectName` property (the default is "oApp").
- Set `cOnClickCommand` to the VFP command or code to execute.
- Put the code to execute in the `Click` method of a subclass of `SFBar`.
- Set the `oImplementation` property to an implementation object that specifies when the bar is visible and enabled and what code to execute when it's selected. Implementation objects are discussed in the next section.

Bars can have images: either set `cPictureFile` to the name of the graphic file to use or `cPictureResource` to the name of the VFP system bar whose graphic you want to use (for example, "_MFI_NEW" to use the image of the File New bar).

You likely won't need to call any `SFBar` methods directly. `Show` displays the bar by first calling the protected `FindBarPosition` method if necessary (if the bar number was specified rather than automatically assigned or if the bar is inverted, it may need to be placed between other bars, so `FindBarPosition` figures out where it should go), and then calling the protected `Define` method to set up and execute the `DEFINE BAR` and `ON SELECTION BAR` commands. The `Hide` method releases the bar so it disappears from the menu.

To create a submenu, call the `AddBar` method of an `SFBar` object, passing the same parameters as you do to the `AddBar` method of `SFPad`. You can also call `AddSeparatorBar` to add a separator bar. Like `SFPad`, `SFBar` has an `AddBars` method that's abstract in this class; however, in a subclass you could put a series of `AddBar` calls that add the desired bars to the submenu for the bar; this makes the bar subclass self-contained.

By the way, the reason we need to know the name of the variable `SFMenu` is instantiated into is the `ON SELECTION BAR` command. We can't use code like "This.Click", because "This" isn't available in the context of a menu selection. Instead, we need to use something like "loMenu.FilePad.FileOpenBar.Click". We can get "FilePad" from `This.Parent.Name` and "FileOpenBar" from `This.Name`, but we can't get "loMenu" from any native property. Thus, we need to store the name of the variable the menu was instantiated into in `SFMenu.cInstanceName`. Then, the entire path to the command to execute can be determined with:

```
This.Parent.Parent.cInstanceName + '.' + ;  
    This.Parent.Name + '.' + This.Name + '.Click()'
```

I created a few commonly used subclasses of `SFBar`. `SFHelpTopicsBar` (discussed earlier) provides a Help Topics bar. `SFSeparatorBar` is used when you call `SFPad.AddSeparatorBar`; it simply has `cCaption` set to "\-". `SFMRUBar` is the class used for an MRU bar when you set `SFPad.IMRU` to `.T`.

Implementation objects

Implementation objects are subclasses of SFMenuFunction (which is based on Custom) that perform some action when a menu item or toolbar button is selected. The advantage of using an implementation object rather than filling in one of the properties that specify the action (such as cOnClickCommand) is separating the display of the menu from its functionality. This allows you to vary the implementation (change the code in the Execute method of the implementation class or specify a different class) without changing the appearance of the menu item. It also allows you to coordinate menu items and toolbars without duplicating code, as both items execute the code in the implementation object and use the IEnabled and IVisible properties of this object to determine whether the bar and toolbar button are enabled and visible.

If you want the implementation object to only control the enabled and visible status of menu bars and toolbar buttons, set INoExecute to .T.; in that case, what code to execute works as it would without an implementation object.

Dynamic menus

Taking inspiration from Matt Slay's Dynamic Form VFPX project, I created a subclass of SFMenu called SFDynamicMenu (in SFDynamicMenu.vcx) that creates a menu from text content. The idea is that the menu can easily be created by simply laying it out in text rather than in code or using the Menu Designer, as shown in **Listing 3**.

Listing 3. The content for the menu as text.

```
&File                Name = FilePad
  &New...             cOnClickCommand = messagebox('You chose New')
                    | cPictureFile = newxpsmall.bmp
                    | Name = FileNew
  &Open...            cOnClickCommand = messagebox('You chose Open')
                    | cPictureFile = openxpsmall.bmp
                    | Name = FileOpen
  &Print              cPictureFile = printxpsmall.bmp
                    | Name = FilePrint
    &Invoice           cOnClickCommand = messagebox('You chose Invoice')
                    | Name = FilePrintInvoice
    &Customer List     cOnClickCommand = messagebox('You chose Customer List')
                    | Name = FilePrintCustomerList
  -----
  E&xit              cOnClickCommand = oMenu.Release()
                    | Name = FileExit
&Edit                Class = SFEditPad | Library = SFMenu.vcx
```

There are a few things to note about this content:

- Lines which are not indented (that is, preceded with a tab) represent pads.
- Lines that have one indent level (one tab at the start of the line) represent bars in a pad.
- Lines that have more than indent level represent bars in a submenu, in another bar.

- The first thing on a line (after any tabs) is the caption of the item. You can use “&” as a replacement for “\<” to indicate the hotkey in a caption. Specify “-” (more than one is allowed) for the caption to represent a separator bar.
- One or more tabs separate the caption from the properties for the item. Properties are specified as “property name = propertyvalue;” for example, “cPictureFile = MyImage.bmp.” Properties are separated with “|,” although that’s a property of SFDynamicMenu (cDelimiter) you can change if you need to use “|” for some other purpose, such as in the caption.
- Item definitions can span multiple lines for readability. Place the “|” (or whatever you set cDelimiter to) as the first non-tab character of the line to indicate this is part of the previous line.

To use SFDynamicMenu, create the content for the menu, either in a file or in a string, instantiate SFDynamicMenu and pass the name of the file to the Init method or set the cMenuDefinition property after instantiation to the content string. Then call Show as you normally would for a menu. For example, this specifies using Menu.txt, shown in Listing 3, as the content for the menu:

```
oMenu = newobject('SFDynamicMenu', 'SFDynamicMenu.vcx', '', 'oMenu', 'Menu.txt')
oMenu.Show()
```

If you change the content of the menu (either changing the content of the file or the layout in cMenuDefinition) and call the Refresh method, the menu changes immediately. You might even allow your end-users to customize the menus for themselves by simply updating the content. You could, of course, implement this idea using XML or a table rather than text content.

Examples

Let’s look at several examples of menus created with these classes. The first, DemoMenu1.prg that accompanies this document, defines a menu programmatically using the base menu classes rather than subclasses (one subclass, SFEditPad, is used). It instantiates SFMenu into oMenu, which it declares public so it doesn’t go out of scope when the program ends. The code adds three pads, File, Edit, and Exit to the menu. It adds New, Open, Print, and Exit bars to the File pad; the first two simply display a messagebox when you select them, Print displays a submenu, and Exit terminates the demo menu. The Exit pad has no functions under it; it acts like a command. Run DemoMenu1.prg (**Listing 4**), try out the menu items, then either choose Exit from the File menu, click the Exit pad, or type oMenu.Release() in the Command window to restore the VFP system menu.

Listing 4. DemoMenu1.prg demonstrates using OOP menus.

```
public oMenu
oMenu = newobject('SFMenu', 'SFMenu.vcx', '', 'oMenu')

* Create the File pad.

oMenu.AddPad('SFPad', 'SFMenu.vcx', 'FilePad')
```

```
with oMenu.FilePad
    .cCaption      = '\<File'
    .cKey          = 'ALT+F'
    .cStatusBarText = 'Performs file functions'

    .AddBar('SFBar', 'SFMenu.vcx', 'FileNew')
    with .FileNew
        .cCaption      = '\<New...'
        .cKey          = 'CTRL+N'
        .cKeyText      = 'Ctrl+N'
        .cStatusBarText = 'Create a file'
        .cPictureFile  = 'newxpsmall.bmp'
        .cOnClickCommand = [messagebox('You chose File, New')]
    endwhile

    .AddBar('SFBar', 'SFMenu.vcx', 'FileOpen')
    with .FileOpen
        .cCaption      = '\<Open...'
        .cKey          = 'CTRL+O'
        .cKeyText      = 'Ctrl+O'
        .cStatusBarText = 'Open a file'
        .cPictureFile  = 'openxpsmall.bmp'
        .cOnClickCommand = [messagebox('You chose File, Open')]
    endwhile

    .AddSeparatorBar()

    .AddBar('FilePrint')
    with .FilePrint
        .lAutoAssignHotKey = .T.
        .cCaption          = '\<Print'
        .cStatusBarText    = 'Print something'
        .cPictureFile      = 'printxpsmall.bmp'

        .AddBar('FilePrintInvoice')
        with .FilePrintInvoice
            .cCaption      = '\<Invoice'
            .cStatusBarText = 'Print invoice'
            .cOnClickCommand = [messagebox('You chose File, Print, Invoice')]
        endwhile

        .AddBar('FilePrintCustomerList')
        with .FilePrintCustomerList
            .cCaption      = '\<Customer List'
            .cStatusBarText = 'Print customer list'
            .cOnClickCommand = [messagebox('You chose File, Print, Customer List')]
        endwhile
    endwhile

    .AddBar('SFBar', 'SFMenu.vcx', 'FileExit')
    with .FileExit
        .cCaption      = 'E\<xit'
        .cStatusBarText = 'Restore the VFP menu'
        .cOnClickCommand = 'oMenu.Release()'
    endwhile
```

```
endwith

* Create the edit pad.

oMenu.AddPad('SFEditPad', 'SFMenu.vcx', 'EditPad')

* Create an Exit pad that executes code.

oMenu.AddPad('SFPadCommand', 'SFMenu.vcx', 'ExitPad')
with oMenu.ExitPad
    .cCaption          = 'E\<xit'
    .cStatusBarText    = 'Restore the VFP menu'
    .cOnClickCommand  = 'oMenu.Release()'
    .cKey              = 'ALT+X'
endwith

* Display the menu.

oMenu.Show()
```

Although the menu it displays is exactly the same as the previous example, DemoMenu2.prg is much simpler:

```
public oMenu
oMenu = newobject('MyMenu', 'MyMenu.vcx', '', 'oMenu')
oMenu.Show()
```

That's because all of the menu, pad, and bar definitions are done in subclasses of SFMenu, SFPad, and SFBar. The DefineMenu method of the SFMenu subclass MyMenu adds the MyFilePad and SFEditPad classes to the menu. The AddBars method of the SFPad subclass MyFilePad adds the MyFileNewBar, MyFileOpenBar, and MyFileExitBar classes to the File pad. MyFileNewBar, MyFileOpenBar, and MyFileExitBar are subclasses of SFBar that specify the desired properties and behavior of these bars. Essentially, these subclasses simply do in the Class Designer what DemoMenu1.prg does in code. The benefit of using subclasses, though, is that they're now reusable (not that I'd want to use these particular bars anywhere but a demo, but you get the idea).

The third example, DemoMenu4.prg, is similar to DemoMenu1.prg, but it adds another bar to the File menu: "Change to French." Choosing this bar runs ChangeToFrench.prg, which changes the captions of the File pad and its bars to English to French and back again if called a second time. Note that it calls the Refresh method of the pad to refresh it and its bars.

The last example, Testform.scx (**Figure 2**), shows both how to create a menu that resides in a top-level form rather than _SCREEN and how to use implementation objects (it also shows shortcut menu, but I'll discuss that later). To have the menu appear in the form, the form's Init method instantiates an SFMenu object and sets its cFormName property to the form name. MyMenu.vcx has two subclasses of SFMenuFunction: MyNewFunction and MyOpenFunction. The Execute methods simply display a message about the function. SFToolBarButton in SFToolbar.vcx has an oImplementation property. If this property

contain an implementation object, its Click method calls the Execute method of that object. Also, the button binds to the IVisible and IEnabled properties of the object, setting its own Visible and Enabled properties accordingly. TestForm.scx sets the oImplementation property of the two toolbar buttons in the form to instances of MyNewFunction and MyOpenFunction. The menu has functions that also use these implementation objects. So, choosing Open from the menu or clicking the Open button in the toolbar execute the same code and turning off the Enable Open Function checkbox in the form disables both the menu function and the toolbar button without using any additional code.

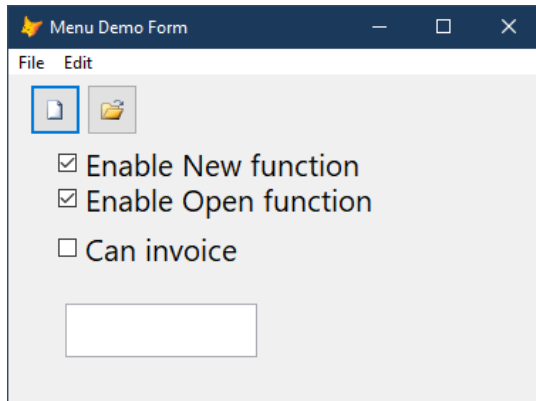


Figure 1. Testform.scx shows displaying menus in top-level forms and using implementation objects to coordinate menus and toolbars.

Converting existing menus

You're probably thinking, "this sounds great but I already have an existing MNX-based menu and it'd be a lot of work to use these menu objects instead." Actually, no. Here's what's involved:

- Use the ProcessMNX and MenuClassGenerator classes in ConvertMNX.prg to create a PRG that uses menu objects to reproduce your MNX menu. ProcessMNX reads an MNX file and generates a filled SFMenu object while MenuClassGenerator takes a filled SFMenu object and creates code to generate that menu, with the menu, each pad, and each bar being its own class (subclasses of SFMenu, SFPad, and SFBar). Here's some code that converts MainMenu.mnx into Menu.prg:

```
loProcess = newobject('ProcessMNX', 'ConvertMNX.prg')
loMenu    = loProcess.ProcessMenu('mainmenu.mnx')
loGenerator = newobject('MenuClassGenerator', 'ConvertMNX.prg')
loGenerator.cMenuName = 'MainMenu'
lcCode = loGenerator.GenerateMenu(loMenu)
strtofile(loGenerator.cCodeOutput, 'menu.prg')
```

Listing 5 shows some partial code of what the classes look like.

Listing 5. Part of Menu.prg showing what the classes look like.

```
*=====
define class MainMenu as SFMenu of SFMenu.vcx
*=====
```

```

    procedure DefineMenu
        with This
            .AddPad('msm_file', '', 'msm_file')
            .AddPad('msm_edit', '', 'msm_edit')
            .AddPad('msm_tools', '', 'msm_tools')
            .AddPad('msm_sysm', '', 'msm_sysm')
        endwhile
    endproc
enddefine

*=====
define class msm_file as SFPad of SFMenu.vcx
*=====
    cCaption = [\<File]
    cKey = [ALT+F]
    cPadPosition = [before msm_edit]
    cPopupMenu = [_mfile]
    cStatusBarText = [Performs file tasks]
    Comment = [This is the File pad]

    procedure AddBars
        with This
            .AddBar('msm_fileBar1', '', 'msm_fileBar1')
            .AddBar('msm_fileBar2', '', 'msm_fileBar2')
            .AddSeparatorBar()
            .AddBar('msm_fileBar4', '', 'msm_fileBar4')
            .AddBar('msm_fileBar5', '', 'msm_fileBar5')
            .AddSeparatorBar()
            .AddBar('mfi_pgset', '', 'mfi_pgset')
            .AddBar('mfi_prevu', '', 'mfi_prevu')
            .AddSeparatorBar()
            .AddBar('msm_fileBar10', '', 'msm_fileBar10')
        endwhile
    endproc
enddefine

*=====
define class msm_fileBar1 as SFBar of SFMenu.vcx
*=====
    cCaption = [\<New...]
    cKey = [CTRL+N]
    cKeyText = [Ctrl+N]
    cOnClickCommand = [ShowMenuChoice('New')]
    cPictureFile = [newxpsmall.bmp]
    cStatusBarText = [Creates a new file]
enddefine

```

Note that pad class names come from the MNX file while bar class names are the pad name suffixed with “Bar” and the bar number. You may wish to replace those names with more meaningful ones, such as, in the code above, replacing “msm_file” with “FilePad” and “msm_fileBar1” with “FileNewBar.” That makes the code easier to read and allows you to reference a bar using code like oMenu.FilePad.FileNewBar if you want to change something about it at runtime.

- Replace your DO MAINMENU.MPR with two lines of code:

```
oMenu = newobject('MainMenu', 'menu.prg', '', 'oMenu')
oMenu.Show()
```

oMenu must be global so it lives throughout the application.

Shortcut menus

What about shortcut menus? SFMenu.vcx covers those too with the SFMenuShortcutMenu class (there's also an older SFShortcutMenu class for backward compatibility but it's been superseded by SFMenuShortcutMenu).

SFMenuShortcutMenu, which is based on SFMenuBase, only has a couple of custom properties: nCol and nRow, the column and row for the menu, respectively. Leave them at their default of 0 to use the default position for the menu, which is at the mouse pointer.

The mostly commonly used methods are AddBar, which adds a bar to the menu, AddSeparatorBar, which adds a separator bar, and Show, which displays the menu. AddBar accepts the following parameters:

- tcPrompt: the caption for the bar. Optionally, you can pass an SFBar object to use that bar, in which case the other parameters except tnElementNumber are ignored.
- tcOnSelection: the command to execute when the bar is chosen.
- tuDisabled: an expression that if it evaluates to .T. disables the bar (in other words, its "skip for" expression).
- tcImage: an image file or system resource (starting with "_") to use as the image for the bar.
- tcSystemBar: the name of a system bar to use for this bar.
- tnElementNumber: the bar number (optional: if not passed, the next available bar number is used).

We'll see some examples shortly.

To make it easier to add a shortcut menu to any object, I added the following to almost all of my base classes (at least the visible ones):

- RightClick: calls This.ShowMenu().
- oHook: a new property that in practice I rarely use but intended to extend the menu in a subclass.
- oMenu: a new property that holds a reference to an SFMenuShortcutMenu instance.
- lUseFormShortcutMenu: a new property that if .T. (the default is .F.) adds the form's shortcut menu bars (if any) to the object's shortcut menu, in essence combining the two into one menu.

- `ShortcutMenu`: an abstract method in most classes; in subclasses or instances, it contains calls to `AddBar` to populate the shortcut menu for the object.
- `ShowMenu`: a new method with code shown in **Listing 6**.

Listing 6. The `ShowMenu` method of my base classes.

```
local lcLibrary
private loObject, ;
    loHook, ;
    loForm
with This

* Define reference to objects we might have menu items from in case the action
* for a bar is to call a method of an object, which can't be done using "This.
* Method" since "This" isn't applicable in a menu.

    loObject = This
    loHook   = .oHook
    loForm   = Thisform

* Define the menu if it hasn't already been defined.

    lcLibrary = 'SFMenu.vcx'
    if vartype(.oMenu) <> '0' and file(lcLibrary)
        .oMenu = newobject('SFMenuShortcutMenu', lcLibrary, '', ;
            'loObject.oMenu'
    endif vartype(.oMenu) <> '0' ...
    if vartype(.oMenu) = '0'

* If there aren't any bars in the menu, have the ShortcutMenu method populate
* it.

        if .oMenu.Count = 0
            .ShortcutMenu(.oMenu, 'loObject')

* Use the hook object (if there is one) to do any further population of the
* menu.

            if vartype(loHook) = '0' and pemstatus(loHook, 'ShortcutMenu', 5)
                loHook.ShortcutMenu(.oMenu, 'loHook')
            endif vartype(loHook) = '0' ...

* If desired, use the form's shortcut menu as well.

            if .lUseFormShortcutMenu and type('Thisform.Name') = 'C' and ;
                pemstatus(loForm, 'ShortcutMenu', 5)
                loForm.ShortcutMenu(.oMenu, 'loForm')
            endif .lUseFormShortcutMenu ...
        endif .oMenu.Count = 0

* Activate the menu if necessary.

        if .oMenu.Count > 0
            .oMenu.ShowMenu()
```



```

        endif .oMenu.Count > 0
    endif vartype(.oMenu) = 'O' ...
endwith

```

The ShortcutMenu method (**Listing 7**) of some base classes, such as SFTextBox, provides a shortcut menu with edit functions (Cut, Copy, Paste, etc.).

Listing 7. The ShortcutMenu method of some base classes.

```

lparameters toMenu, ;
    tcObject
local lcCut, ;
    lcCopy, ;
    lcPaste, ;
    lcClear, ;
    lcSelect
if type('oLocalizer.Name') = 'C'
    lcCut    = oLocalizer.GetLocalizedString('MENU_CUT')
    lcCopy   = oLocalizer.GetLocalizedString('MENU_COPY')
    lcPaste  = oLocalizer.GetLocalizedString('MENU_PASTE')
    lcClear  = oLocalizer.GetLocalizedString('MENU_CLEAR')
    lcSelect = oLocalizer.GetLocalizedString('MENU_SELECT_ALL')
else
    lcCut    = 'Cu<\t'
    lcCopy   = '\<Copy'
    lcPaste  = '\<Paste'
    lcClear  = 'Cle<\ar'
    lcSelect = 'Se<\lect All'
endif type('oLocalizer.Name') = 'C'
with toMenu
    .AddBar(lcCut, ;
        "sys(1500, '_MED_CUT', '_MEDIT')", ;
        'not ' + tcObject + '.Enabled or ' + tcObject + '.ReadOnly', ;
        'CutXPSSmall.bmp', ;
        '_med_cut')
    .AddBar(lcCopy, ;
        "sys(1500, '_MED_COPY', '_MEDIT')", ;
        , ;
        'CopyXPSSmall.bmp', ;
        '_med_copy')
    .AddBar(lcPaste, ;
        "sys(1500, '_MED_PASTE', '_MEDIT')", ;
        'not ' + tcObject + '.Enabled or ' + tcObject + '.ReadOnly', ;
        'PasteXPSSmall.bmp', ;
        '_med_paste')
    .AddBar(lcClear, ;
        "sys(1500, '_MED_CLEAR', '_MEDIT')", ;
        'not ' + tcObject + '.Enabled or ' + tcObject + '.ReadOnly', ;
        '_med_clear', ;
        '_med_clear')
    .AddSeparatorBar()
    .AddBar(lcSelect, ;
        "sys(1500, '_MED_SLCTA', '_MEDIT')", ;
        , ;

```

```
        '_med_slcta', ;  
        '_med_slcta')  
endwith
```

A few notes about this code:

- It supports localizing the bar captions using a global localizer object. If it doesn't exist, the default captions, such as "Cu\<t," are used.
- ShortcutMenu is passed a reference to a menu object to fill. It's also passed a variable name that at runtime contains a reference to the object. That way, you can reference it in bars, such as what method of the object to call when a bar is chosen or what property of the object determines if the bar is enabled. For example, the Cut bar is disabled if the current object's Enabled property is .F. or ReadOnly is .T. Note how the object has to be referenced: "tcObject" has to be used outside of quotes, which makes the syntax a little hard to read.
- I put each parameter on a separate line to make it easier to see which parameter is which, especially when some aren't used.

Typically, all you need to do to add a shortcut menu to any class or object instance is to add code to its ShortcutMenu method that adds bars to the menu. Testform.scx demonstrates this (**Listing 8**).

Listing 8. The ShortcutMenu method of Testform.scx.

```
lparameters toMenu, ;  
            tcObject  
  
* Add the New and Open buttons from the menu bar to the shortcut menu.  
  
toMenu.AddBar(This.oMenuBar.FilePad.FileNew)  
toMenu.AddBar(This.oMenuBar.FilePad.FileOpen)  
  
* Add additional bars.  
  
toMenu.AddBar('\<Find Customer', ;  
            tcObject + '.FindCustomer()', ;  
            , ;  
            'GridExtras\Search16.bmp')  
toMenu.AddBar('F\<ilter...', ;  
            tcObject + '.Filter()', ;  
            , ;  
            'GridExtras\ShowFilters16.bmp')  
toMenu.AddSeparatorBar()  
toMenu.AddBar('\<Print Invoice', ;  
            tcObject + '.PrintInvoice()', ;  
            'not ' + tcObject + '.lInvoiceSelected', ;  
            'GridExtras\Printer16.bmp')
```

When you run the form and right-click somewhere other than the textbox, you see the menu shown in **Figure 2**. Turn on the Can Invoice checkbox to enable the Print Invoice function in the menu. Right-click the textbox to see the shortcut menu SFTextBox creates.

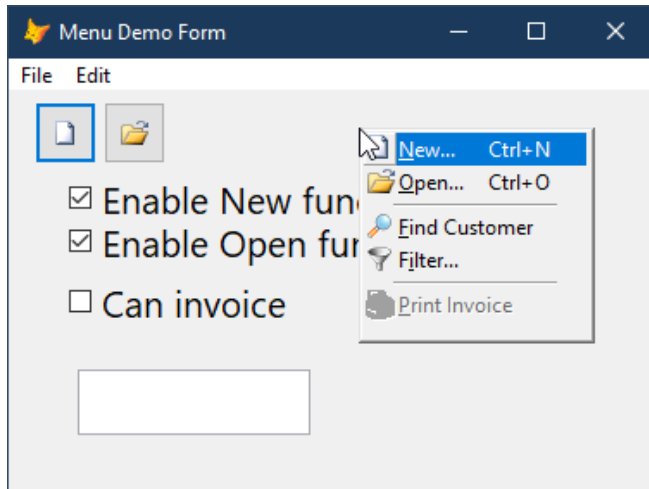


Figure 2. Testform.scx has a simple shortcut menu.

The ability to pass an existing SFCBar object to AddBar means you can reuse bars in your application menu in shortcut menus. Note the code shown above uses the Open and New bars of the main menu in the shortcut menu. Since the bars use implementation objects, the both menus are automatically synchronized with respect to bars being enabled or visible and what's executed when selected.

Other ideas

A great add-on for these classes would be a visual tool to create a menu. Rumor has it that Rick Schummer is working on such a tool.

Conclusion

Object-oriented menus have long been a requested enhancement to VFP. However, as you can see from the simple classes presented here, they're easy to implement. These menu classes make it easier to create dynamic menus for your applications.

Object-oriented reports

My company's main product, Stonefield Query, generates reports dynamically. That is, based on a list of fields and their properties (width, formatting, grouping, etc.), it generates an FRX file on the fly and then runs it. Since FoxPro 2.0, the FRX file has been a table, so it seems like this should be an easy task. However, the fact of the matter is that the FRX structure is ugly as sin. Although it's documented (you can run the 90FRX report in the Tools\FileSpec subdirectory of the VFP home directory to print the FRX file structure), different report object types have subtle uses of the fields in this table structure and that's not well documented. Also, there are a lot of fields, making for some pretty long INSERT INTO statements. Also, the size and position values are in 10,000th of an inch (known as FoxPro Report Units, or FRUs), and there are a weird bunch of "fudge" factors you have to

apply to these numbers such as the height of a band bar in the Report Designer. After struggling for hours with some really complex code that created an FRX, I finally screamed “I want report objects” (this isn’t metaphorical; I really did scream this <g>). Unfortunately, there was no one around to hear me, so I created them myself.

The report object classes I created are all in SFRepObj.vcx, and all are based on SFReportBase, a subclass of Custom. Each uses SFRepObj.h as its include file; constants like cnBAND_HEIGHT are much easier to understand than hard-coded values like 2083.333.

I don’t have room to show much of the code in these classes (you can look at the code in the VCX yourself), but that’s not really important anyway. You can consider these classes to be “black boxes;” you don’t have to know how they work, just how to use them.

The object-oriented report classes we’ll look at in this document are a VFPX project, available at <https://github.com/VFPX/OOPReports>.

SFReportFile

The only class you actually instantiate directly is SFReportFile. This class represents the report file, and has properties and methods that expose the Report Designer interface programmatically. It’ll instantiate objects from other classes when necessary (such as when you add a field to the report).

Table 1 shows the public properties of SFReportFile. Other than cReportFile and cUnits, the public properties simply represent the same options for the report you see in the Report Designer. Each has an Assign method that prevents values of the wrong data type or range from being stored. cUnits needs a little explanation. Sometimes, it’s easier to express report units (such as the horizontal and vertical position of an object) in characters and lines. For example, if you have a 30-character field, it’s easier to specify the width for it on a report as 30 characters rather than figuring out how many inches it should be. All position and size properties in all objects we’ll look at should be expressed in the units defined in cUnits.

Table 1. Public properties of SFReportFile.

Property	Purpose
cDevice	The name of the printer to use
cFontName	The default font for the report (the default value is “Courier New”)
cMemberData	The member data for the report header record
cReportFile	The name of the report file to create
cUnits	The unit of measurement: "C" (characters; the default), "I" (inches), or "M" (centimeters) (constants are defined for these values in SFRepObj.h)
IAdjustObjectWidths	.T. (the default) to ensure no objects are wider than the paper width
INoDataEnvironment	.T. (the default) to prevent the DataEnvironment from being edited
INoPreview	.T. to prevent the report from being previewed or printed in the Report Designer
INoQuickReport	.T. (the default) to prevent access to the Quick Report function
IPrintColumns	.T. to print records in columns across the page, .F. to print top to bottom then in columns
IPrivateDataSession	.T. to use a private datasession with this report

Property	Purpose
lSummaryBand	.T. if the report has a summary band
lTitleBand	.T. if the report has a title band
lWholePage	.T. to use the whole page, .F. to use the printable page
nColumns	The number of columns in the report (the default value is 1)
nColumnSpacing	The spacing between columns
nDefaultSource	The default paper source for the report (the default value is -1)
nDetailBands	The number of detail bands in the report
nFontSize	The default font size for the report (the default value is 10)
nFontStyle	The default font style for the report
nGroups	The number of groups in the report
nLeftMargin	The left margin for the report
nMinPaperWidth	The minimum paper width
nOrientation	The orientation for the report: 0 = auto-set to landscape if the report is too wide for portrait, 1 = use portrait, 2 = use landscape (constants are defined for these values in SFRepObj.h)
nPaperLength	The paper length
nPaperSize	The paper size (the default value is PRTINFO(2); see help for that function for a list of values)
nPaperWidth	The paper width
nRepWidth	The calculated report width
nRulerScale	The scale used for the ruler: 1 = inches, 2 = metric, 3 = pixels (constants are defined for these values in SFRepObj.h)
nWidth	The report width

Here's some sample code that instantiates an SFReportFile object, specifies the name of report to create, indicates that the report has a summary band, and sets the default font for the report.

```
loReport = newobject('SFReportFile', 'SFRepObj.vcx')
loReport.cReportFile = 'CustomerReport.frx'
loReport.lSummaryBand = .T.
loReport.cFontName = 'Arial'
```

SFReportFile's public methods are shown in **Table 2**. We'll look at some of these methods as we explore other objects.

Table 2. Public methods of SFReportFile.

Method	Description
CreateDetailBand	Creates a detail band
CreateGroupBand	Creates a new group band
CreateVariable	Creates a report variable
GetHFactor	Calculates the horizontal factor for the specified font
GetPaperWidth	Gets the max. paper width
GetRelativeVPosition	Get the vertical position of the specified object relative to the start of its band
GetReportBand	Returns an object reference to the specified band
GetVariable	Returns a variable object
GetVFactor	Calculates the vertical factor for the specified font
Load	Loads the specified FRX into report objects
LoadFromCursor	Loads the FRX in the current work area into report objects

Method	Description
Save	Creates the report file

Report bands

SFReportFile creates an object for each band in a report by instantiating an SFReportBand object into a protected property. For example, the Init method uses the following code to create page header, detail, and page footer band objects automatically, because every report has at least these three bands:

```
with This
    .oPageHeaderBand = newobject('SFReportBand', .ClassLibrary, '', ;
        'Page Header', This)
    .oDetailBand      = newobject('SFReportBand', .ClassLibrary, '', ;
        'Detail', This)
    .oPageFooterBand = newobject('SFReportBand', .ClassLibrary, '', ;
        'Page Footer', This)
endwith
```

The Init method of SFReportBand accepts two parameters: the band type (which it puts into the cBandType property) and an object reference to the SFReportFile object (which it puts into oReport) so it can call back to some SFReportFile methods if necessary.

In addition to the three default bands, you can create additional bands in one of three ways. To specify that the report has a title or summary band, set the ITitleBand or ISummaryBand property to .T. To create group header and footer bands, call the CreateGroupBand method. Groups are automatically numbered in the order they're defined; a possible enhancement would allow groups to be reordered. To create additional detail bands, call the CreateDetailBand method, passing .T. if you want detail header and footer bands. Like groups, detail bands are automatically numbered in the order they're defined.

The GetReportBand method returns an object reference to the specified band. In the case of a group header, group footer, detail, detail header, or detail footer band, you also specify which number you want the band for, such as 1 for the first group footer. Here's some sample code that sets the height of the page header and detail bands.

```
loPageHeader = loReport.GetReportBand('Page Header')
loPageHeader.nHeight = 8
loDetail     = loReport.GetReportBand('Detail')
loDetail.nHeight = 1
```

By the way, you don't have to hard-code band names as I've done in this example and the previous example. All band names are defined as constants in SFRepObj.h, the include file for all classes in SFRepObj.vcx. For example, ccBAND_PAGE_HEADER is defined as "PAGE HEADER", so you could use this code:

```
loPageHeader = loReport.GetReportBand(ccBAND_PAGE_HEADER)
```

Table 3 shows the public properties of SFReportBand; as with SFReportFile, they simply expose band options available in the Report Designer as properties. nHeight is expressed in

the units defined in the `cUnits` property of the `SFReportFile` object; for example, if `cUnits` is "C," setting `nHeight` to 8 defines an 8-line high band. One nice feature: if the objects in a band extend below the defined height of the band, saving the report automatically adjusts the band height if the `lAdjustBandHeight` property is `.T.` (the `nHeight` property isn't changed, but the band's record in the `FRX` is).

Table 3. Public properties of `SFReportBand`.

Property	Purpose
<code>cOnEntry</code>	The "on entry" expression for the band
<code>cOnExit</code>	The "on exit" expression for the band
<code>cTargetAlias</code>	The target alias for detail bands
<code>lAdjustBandHeight</code>	<code>.T.</code> (the default) if the height of the band should be adjusted to fit the objects in it
<code>lConstantHeight</code>	<code>.T.</code> if the band should be a constant height
<code>lDeleteObjectsOutsideBand</code>	<code>.T.</code> (the default) to delete objects outside the band height; this is only used if <code>lAdjustBandHeight</code> is <code>.F.</code>
<code>lPageFooter</code>	<code>.T.</code> if this is a summary band and a page footer should be printed
<code>lPageHeader</code>	<code>.T.</code> if this is a summary band and a page header should be printed
<code>lStartOnNewPage</code>	<code>.T.</code> if this band should start on a new page
<code>nCount</code>	The number of items in the band
<code>nHeight</code>	The height of the band
<code>nNewPageWhenLessThan</code>	Starts a group on a new page when there is less than this much space left on the current page

`SFReportBand` just has three public methods that you'll use: `Add`, `GetReportObjects`, and `Remove` (it has a few other public methods but they're called from methods in `SFReportFile`). `Add` is probably the method you'll use the most in any of the classes; it adds an object to a band, so you'll call it once for every object in the report. Pass `Add` the object type you want to add: "field," "text," "image," "line," or "box" (these values are defined in `SFRepObj.h`, so you can use constants rather than hard-coded values), and it returns a reference to the newly created object. `GetReportObjects` populates the specified array with object references to the objects added to the band. You can optionally pass it a filter condition to only get certain items; typically, the filter would check for a property of the objects being a certain value. You may not use this method yourself, but `SFReportFile` uses it to output a band and all of its objects to the `FRX` file. `Remove` removes the specified object from the band; it isn't used often since you likely wouldn't add an object only to remove it later.

`SFReportGroup` is a subclass of `SFReportBand` that's specific for group header and footer bands. Its public properties are shown in **Table 4**. The most important one is obviously `cExpression`, since this determines what constitutes a group.

Table 4. Public properties of `SFReportGroup`.

Property	Purpose
<code>cExpression</code>	The group expression
<code>lPrintOnEachPage</code>	<code>.T.</code> to print the group header on each page
<code>lResetPage</code>	<code>.T.</code> to reset the page number for each page to 1

Property	Purpose
lStartInNewColumn	.T. if each group should start in a new column
nNewPageWhenLessThan	Starts a group on a new page when there is less than this much space left on the current page

Here's some code that creates a group, gets a reference to the group header band, and sets the group expression, height, and printing properties.

```
loReport.CreateGroupBand()  
loGroup = loReport.GetReportBand('Group Header', 1)  
loGroup.cExpression = 'CUSTOMER.COUNTRY'  
loGroup.nHeight = 3  
loGroup.lPrintOnEachPage = .T.  
loGroup.nNewPageWhenLessThan = 4
```

Fields and text

The most common thing you'll add to a report band is a field (actually, an expression which could be a field name but can be any valid FoxPro expression), since the whole purpose of a report is to output data. SFReportField is the class used for fields. However, before we talk about SFReportField, let's look at its parent classes.

SFReportRecord is an ancestor class for every class in SFRepObj.vcx except SFReportFile and SFReportBase. It has a few properties that all objects share:

- Recno: the record number of the object in the report.
- cComment: the comment for the report record.
- cMemberData: the member data for the report record.
- cUniqueID: the unique ID for the report record (normally left blank and auto-assigned).
- cUser: the user-defined property.
- nObjectType: contains the OBJTYPE value for the FRX record of the object (for example, fields have an OBJTYPE of 8).

It also has two methods: CreateRecord and ReadFromFRX. CreateRecord is called from the SFReportFile object when it creates a report. SFReportFile creates an object from a record in the FRX using SCATTER NAME loRecord BLANK MEMO to create an object with one property for each field in the record and then passes that object to the CreateRecord method of the SFReportRecord object, which fills in properties of the report record object from values in its own properties. SFReportRecord is an abstract class; it isn't used directly, but is the parent class for other classes. Its CreateRecord method simply ensures a valid report record object was passed and sets the ObjType property of this object (which is written to the OBJTYPE field in the FRX) to its nObjectType property. ReadFromFRX sort of does the opposite: it assign its properties the values from the current record in an open FRX table.

SFReportObject is a subclass of SFReportRecord that’s used for report objects (here, I mean “object” in the “thingy” sense, such as a field, rather than “what you get when you instantiate a class” sense). It has the public properties shown in **Table 5**, which represent the minimum set of options for a report object. As with other classes, these properties simply expose options available in the Report Designer as properties.

Table 5. Public properties of SFReportObject.

Property	Purpose
cAlignment	The alignment for the object: "left", "center", or "right" (constants are defined for these values in SFRepObj.h)
cName	A name for the object (used by SFReportBand.Item to locate an item by name)
cPrintWhen	The Print When expression
lAutoCenter	.T. (the default) to automatically center this object vertically in a row when using character units for the report
lPrintInFirstWholeBand	.T. (the default) to print in the first whole band of a new page
lPrintOnNewPage	.T. to print when the detail band overflows to a new page
lPrintRepeats	.T. (the default) to print repeated values
lRemoveLineIfBlank	.T. to remove a line if there are no objects on it
lStretch	.T. if the object can stretch
lTransparent	.T. (the default) if the object is transparent, .F. for opaque
nBackColor	The object’s background color; use an RGB() value (-1 = default)
nFloat	0 if the object should float in its band, 1 (the default) if it should be positioned relative to the top of the band, or 2 if it should be relative to the bottom of the band (constants are defined for these values in SFRepObj.h)
nForeColor	The object’s foreground color; use an RGB() value (-1 = default)
nGroup	Non-zero if this object is grouped with other objects
nHeight	The height of the object
nHPosition	The horizontal position for the object
nPrintOnGroupChange	The group number if this object should print on a group change
nVPosition	The vertical position for the object relative to the top of the band
nWidth	The width of the object

The CreateRecord method first uses DODEFAULT() to execute the behavior of SFReportRecord, then it has some data conversion to do. For example, object colors are stored in the PENRED, PENGREEN, and PENBLUE fields in the FRX record, but we want to have a single nForeColor property that we set (for example, to red using RGB(255, 0, 0)) like we do with VFP controls. Other properties are similar; for example, the value in nFloat updates the FLOAT, TOP, and BOTTOM fields in the FRX.

Finally, we’re back to SFReportField, the subclass of SFReportObject that holds information about fields in a report. This class adds the properties shown in **Table 6** to those of SFReportObject. As you can see, you have the same control over the properties of an object in a report as you do in the Report Designer.

Table 6. Public properties of SFReportField.

Property	Purpose
cCaption	The design-time caption for the field
cDataType	The data type of the expression: “N” for numeric, “D” for date, and “C” for everything

Property	Purpose
	else (only required if you'll edit the report in the Report Designer later)
cExpression	The expression to display
cFontName	The font to use (if blank, which it is by default, SFReportFile.cFontName is used)
cPicture	The picture (format and inputmask) for the field
cTotalType	The total type: "N" for none, "C" for count, "S" for sum, "A" for average, "L" for lowest, "H" for highest, "D" for standard deviation, and "V" for variance (constants are defined for these values in SFRepObj.h)
IFontBold	.T. if the object should be bolded
IFontItalic	.T. if the object should be in italics
IFontUnderline	.T. if the object should be underlined
IResetOnPage	.T. to reset the variable at the end of each page; .F. to reset at the end of the report
nDataTrimming	Specifies how the Trim Mode for Character Expressions is set
nFontCharSet	The font charset to use
nFontSize	The font size to use (if 0, which it is by default, SFReportFile.nFontSize is used)
nResetOnDetail	The detail band number to reset the value on
nResetOnGroup	The group number to reset the value on

As with SFReportObject, SFReportField's CreateRecord method uses DODEFAULT() to get the behavior of SFReportRecord and SFReportObject, then it does some data conversion similar to what SFReportObject does (for example, IFontBold, IFontItalic, and IFontUnderline are combined into a single FONTSTYLE value).

SFReportText is a subclass of SFReportField, since it has the same properties but only slightly different behavior. It automatically adds quotes around the expression since text objects always contain literal strings rather than expressions. It also sets the PICTURE field in the FRX to match the alignment of the data (because that's how alignment is handled for text objects), and sizes the object appropriately for the size of the text (in other words, it acts like setting the AutoSize property of a Label control to .T.).

Here's some code that adds text and field objects to the detail band and sets their properties. This code uses characters as the units, so values are in characters or lines.

```
loObject = loDetail.Add('Text')
loObject.cExpression = 'Country:'
loObject.nVPosition = 1
loObject.lFontBold = .T.
loObject = loDetail.Add('Field')
loObject.cExpression = 'CUSTOMER.COUNTRY'
loObject.nWidth = fsize('COUNTRY', 'CUSTOMER')
loObject.nVPosition = 1
loObject.nHPosition = 10
loObject.lFontBold = .T.
```

Lines, boxes, and images

SFReportShape is a subclass of SFReportObject that defines the properties for lines and boxes (it isn't used directly but is subclassed). nPenPattern is the pen pattern for the object: 0 = none, 1 = dotted, 2 = dashed, 3 = dash-dot, 4 = dash-dot-dot, and 8 = normal

(constants are defined for these values in SFRepObj.h). nPenSize is the pen size for the line: 0, 1, 2, 4, or 6.

SFReportLine is a subclass of SFReportShape that’s used for line objects. It adds one property, lVertical, that you should set to .T. to create a vertical line or .F. (the default) for a horizontal one. Its CreateRecord method sets the height for a horizontal line or the width for a vertical one to the appropriate value based on the pen size.

The following code adds a heavy blue line on line 4 of the page header band:

```
loObject = loPageHeader.Add('Line')
loObject.nWidth      = lnWidth
loObject.nVPosition = 4
loObject.nHPosition = 0
loObject.nPenSize   = 6
loObject.nForeColor = rgb(0, 0, 255)
```

Boxes use SFReportBox, which is also a subclass of SFReportShape. It adds nCurvature (the curvature of the box corners; the default is 0, meaning no curvature), lStretchToTallest (.T. to stretch the object relative to the tallest object in the band), and nFillPattern (the fill pattern for the object: 0 = none, 1 = solid, 2 = horizontal lines, 3 = vertical lines, 4 = diagonal lines, leaning left, 5 = diagonal lines, leaning right, 6 = grid, 7 = hatch; constants are defined for these values in SFRepObj.h) properties.

SFReportImage, a subclass of SFReportObject, is used for images. Set the cImageSource property to the name of the image file or General field that’s the source of the image and nImageSource to 0 if the image comes from a file, 1 if it comes from a General field, or 2 if it’s an expression (constants are defined for these values in SFRepObj.h). nStretch defines how to scale the image: 0 = clip, 1 = isometric, and 2 = stretch (the same values used by the Stretch property of an Image control; constants are defined for these values in SFRepObj.h). Its CreateRecord method automatically puts quotes around the image source if a file is used, and sets the appropriate field in the FRX record if the cAlignment property is set to “Center” (only applicable for General fields).

Report variables

Report variables are defined using the CreateVariable method of the SFReportFile object. This method returns an object reference to the SFReportVariable object it created so you can set properties of the variable. **Table 7** lists the public properties for variables.

Table 7. Public properties of SFReportVariable.

Property	Purpose
cInitialValue	The initial value
cName	The variable name
cTotalType	The total type: “N” for none, “C” for count, “S” for sum, “A” for average, “L” for lowest, “H” for highest, “D” for standard deviation, and “V” for variance (constants are defined for these values in SFRepObj.h)
cValue	The value to store

Property	Purpose
IReleaseAtEnd	.T. to release the variable at the end of the report
IResetOnPage	.T. to reset the variable at the end of each page; .F. to reset at the end of the report
nResetOnGroup	The group number to reset the variable on

The following code creates a report variable called lnCount and specifies that it should start at 0 and increment by 1 for each record printed in the report. This variable is then printed in the summary band of the report, showing the number of records printed.

```

loVariable = loReport.CreateVariable()
loVariable.cName      = 'lnCount'
loVariable.cValue     = 1
loVariable.cInitialValue = 0
loVariable.cTotalType  = ccTOTAL_SUM
loSummary = loReport.GetReportBand('Summary')
loObject  = loSummary.Add('Field')
loObject.cExpression = 'transform(lnCount) + ' + ;
    '" record" + iif(lnCount = 1, "", "s") + " printed"'
loObject.nWidth      = 21
loObject.nVPosition  = 2
loObject.nHPosition  = 0
loObject.lFontBold   = .T.
    
```

Examples

Customers.prg and Employees.prg are sample programs that create reports for the Customer and Employee tables in the VFP Testdata database. Customers.prg creates (and previews) CustomerReport.frx, which shows customers grouped by country, with the maximum order amount subtotaled by country and totaled at the end of the report. Employees.prg creates EmployeeReport.frx, which shows the name and photo of each employee. These reports aren't intended to be realistic; they just show off various features of the report classes described in this article, including printing images and lines, setting font sizes and object colors, positioning objects in different bands, use of report variables and group bands, etc.

Craig Boyd's GridExtras (<https://tinyurl.com/ycmqo8fg>; see **Figure 3** for a sample) provides a grid on steroids: sorting, incremental searching, and filtering on each column, column selection, and output to Excel and Print Preview. The Print Preview feature uses SFReportFile to dynamically create a report based on the current grid layout. **Figure 4** shows that the generated report has the same layout as the grid.

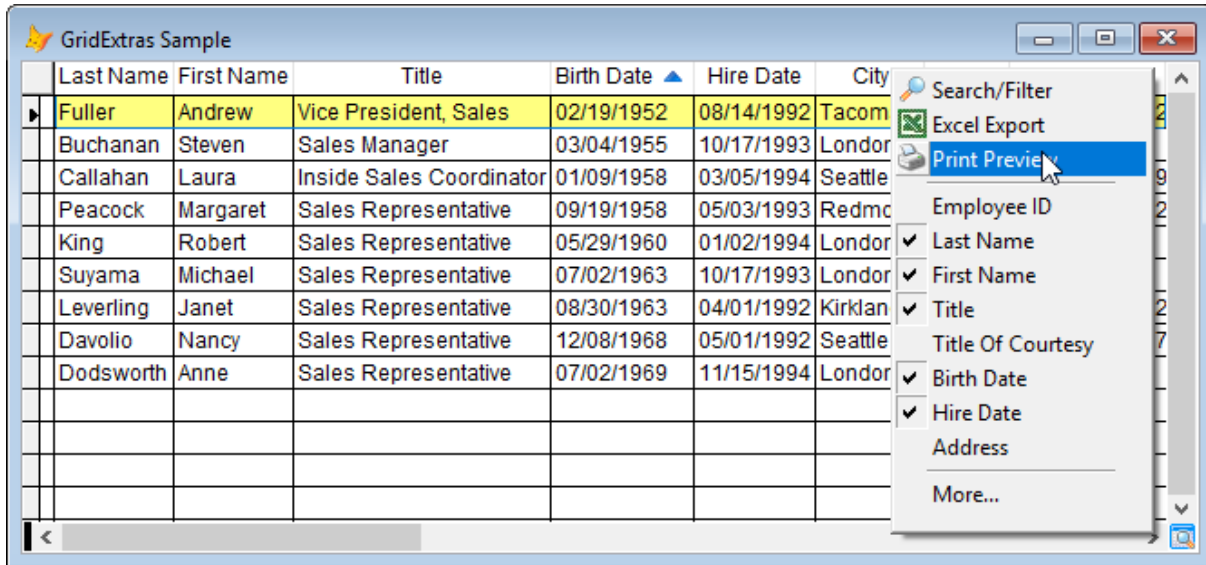


Figure 3. GridExtras provides a grid control on steroids.

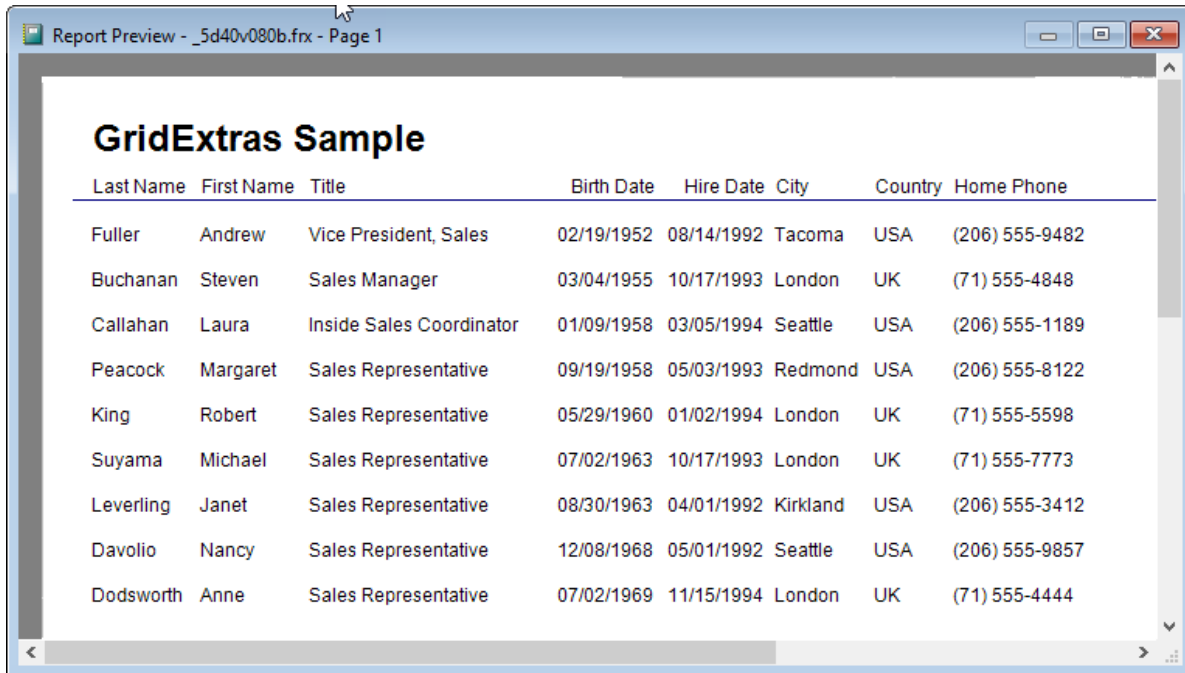


Figure 4. GridExtras uses SFReportFile to generate a report on the fly.

Altering existing reports is also much easier to do using these classes. Suppose you have a report that contains some sensitive information. Some staff shouldn't see that information, so you use the Print When expression for those fields to not output them unless the staff have the correct permissions. For example, **Figure 5** shows a full employee listing, but Birth Date and Home Phone should only be visible to Human Resources (HR) staff.

Employee Listing								
12/12/2018								Page 1
Last Name	First Name	Birth Date	Hire Date	Home Phone	City	Country	Postal Code	Ext
Buchanan	Steven	03/04/1955	09/13/1992	(71) 555-4848	London	UK	SW1 8JR	3453
Suyama	Michael	07/02/1963	09/13/1992	(71) 555-7773	London	UK	EC2 7JR	428
King	Robert	05/29/1960	11/29/1992	(71) 555-5598	London	UK	RG1 9SP	465
Callahan	Laura	01/09/1958	01/30/1993	(206) 555-1189	Seattle	USA	98105	2344
Dodsworth	Anne	01/27/1966	10/12/1993	(71) 555-4444	London	UK	WG2 7LT	452
Hellstern	Albert	03/13/1960	03/01/1993	(206) 555-4869	Bellevue	USA	98006	7559
Smith	Tim	06/06/1973	01/15/1993	(206) 555-3857	Kent	USA	98042	6261
Patterson	Caroline	09/11/1972	05/15/1993	(206) 555-3487	Auburn	USA	98002	1411
Brid	Justin	10/08/1962	01/01/1994	88 83 83 16	Haguenau	France	67500	377
Martin	Xavier	11/30/1960	01/15/1994	88 62 43 53	Schiltigheim	France	67300	380
Pereira	Laurent	12/09/1965	02/01/1994	88 01 01 68	Strasbourg	France	67000	376
Davolio	Nancy	12/08/1948	03/29/1991	(206) 555-9857	Seattle	USA	98122	5467
Fuller	Andrew	02/19/1942	07/12/1991	(206) 555-9482	Tacoma	USA	98401	3457
Leverling	Janet	08/30/1963	02/27/1991	(206) 555-3412	Kirkland	USA	98033	3355
Peacock	Margaret	09/19/1937	03/30/1992	(206) 555-8122	Redmond	USA	98052	5176

Figure 5. The full employee listing.

The Print When expression for those two fields and their column headers is “pLHR.” The variable pLHR is .T. for HR staff and .F. otherwise. **Figure 6** shows the report run by non-HR staff. As you can see by the red boxes (not part of the report, just added to the figure for illustration), BirthDate and HomePhone don’t appear but leave obvious holes in the report layout. What would be nice is if the other columns could move left to take up the empty space.

Employee Listing								
12/12/2018								Page 1
Last Name	First Name	Birth Date	Hire Date	Home Phone	City	Country	Postal Code	Ext
Buchanan	Steven		09/13/1992		London	UK	SW1 8JR	3453
Suyama	Michael		09/13/1992		London	UK	EC2 7JR	428
King	Robert		11/29/1992		London	UK	RG1 9SP	465
Callahan	Laura		01/30/1993		Seattle	USA	98105	2344
Dodsworth	Anne		10/12/1993		London	UK	WG2 7LT	452
Hellstern	Albert		03/01/1993		Bellevue	USA	98006	7559
Smith	Tim		01/15/1993		Kent	USA	98042	6261
Patterson	Caroline		05/15/1993		Auburn	USA	98002	1411
Brid	Justin		01/01/1994		Haguenau	France	67500	377
Martin	Xavier		01/15/1994		Schiltigheim	France	67300	380
Pereira	Laurent		02/01/1994		Strasbourg	France	67000	376
Davolio	Nancy		03/29/1991		Seattle	USA	98122	5467
Fuller	Andrew		07/12/1991		Tacoma	USA	98401	3457
Leverling	Janet		02/27/1991		Kirkland	USA	98033	3355
Peacock	Margaret		03/30/1992		Redmond	USA	98052	5176

Figure 6. The employee listing for non-HR staff.

HackReport.prg (**Listing 9**) shows how to do this. The code isn’t complicated: instantiate SFReportFile, load the FRX, remove objects in the Page Header and Detail bands that don’t

appear because of their Print When expressions, and move the rest of the fields the appropriate distance to the left.

Listing 9. HackReport.prg adjusts the report so there are no holes.

```
loReport = newobject('SFReportFile', 'SFRepObj.vcx')
loReport.Load('Employees.frx')

* Process objects in the page header and detail bands.

loBand = loReport.GetReportBand('Page Header')
ProcessObjects(loBand)
loBand = loReport.GetReportBand('Detail')
ProcessObjects(loBand)

* Save the updated report and run it.

loReport.cReportFile = addbs(sys(2023)) + 'EmployeeReport.frx'
loReport.Save()
report form (loReport.cReportFile) preview

* Specifically release the report object so proper object cleanup occurs.

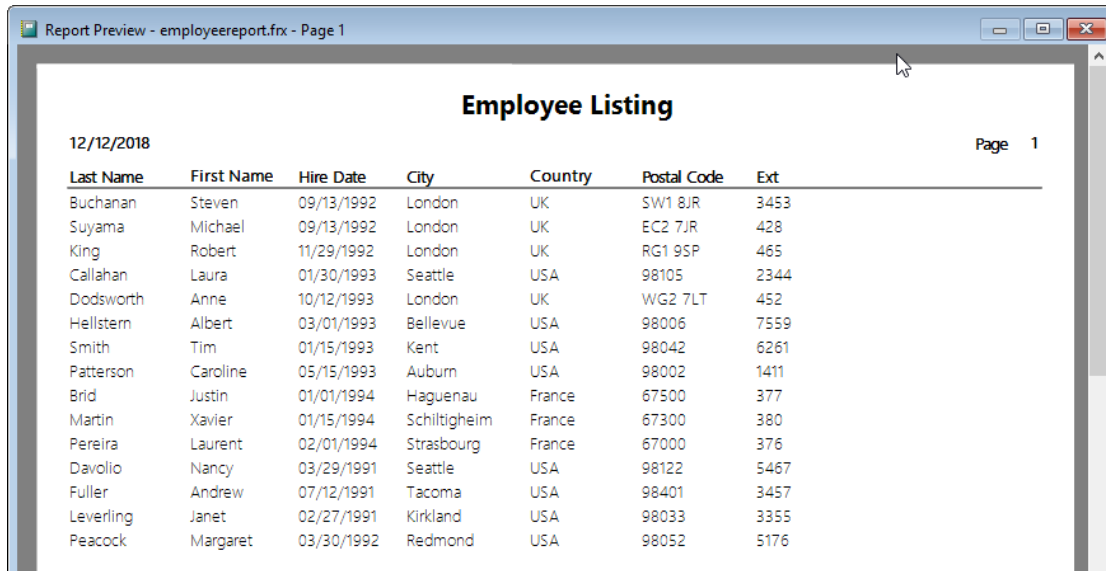
loReport.Release()

* Remove any objects that fail their PrintWhen expression and move objects to
* the right of those objects on the same line to the left.

function ProcessObjects(toBand)
local laObjects[1], ;
    lnObjects, ;
    lnAdjust, ;
    lnI, ;
    loObject, ;
    lcExpr, ;
    lnVPos, ;
    llVisible
lnObjects = toBand.GetReportObjects(@laObjects)
lnAdjust = 0
for lnI = 1 to lnObjects
    loObject = laObjects[lnI]
    lcExpr = loObject.cPrintWhen
    if lnAdjust <> 0 and loObject.nVPosition = lnVPos
        loObject.nHPosition = loObject.nHPosition - lnAdjust
    endif lnAdjust <> 0 ...
    llVisible = .T.
    if not empty(lcExpr)
        try
            llVisible = evaluate(lcExpr)
        catch
            endtry
    endif not empty(lcExpr)
    if not llVisible
        if lnI < lnObjects
            lnAdjust = laObjects[lnI + 1].nHPosition - ;
```

```
        loObject.nHPosition
    endif lnI < lnObjects
    toBand.Remove(loObject.cUniqueID)
    lnVPos = loObject.nVPosition
endif not llVisible
next lnI
```

Figure 7 shows what the report looks like when run with plHR .F.



Last Name	First Name	Hire Date	City	Country	Postal Code	Ext
Buchanan	Steven	09/13/1992	London	UK	SW1 8JR	3453
Suyama	Michael	09/13/1992	London	UK	EC2 7JR	428
King	Robert	11/29/1992	London	UK	RG1 9SP	465
Callahan	Laura	01/30/1993	Seattle	USA	98105	2344
Dodsworth	Anne	10/12/1993	London	UK	WG2 7LT	452
Hellstern	Albert	03/01/1993	Bellevue	USA	98006	7559
Smith	Tim	01/15/1993	Kent	USA	98042	6261
Patterson	Caroline	05/15/1993	Auburn	USA	98002	1411
Brid	Justin	01/01/1994	Haguenau	France	67500	377
Martin	Xavier	01/15/1994	Schiltigheim	France	67300	380
Pereira	Laurent	02/01/1994	Strasbourg	France	67000	376
Davolio	Nancy	03/29/1991	Seattle	USA	98122	5467
Fuller	Andrew	07/12/1991	Tacoma	USA	98401	3457
Leverling	Janet	02/27/1991	Kirkland	USA	98033	3355
Peacock	Margaret	03/30/1992	Redmond	USA	98052	5176

Figure 7. The hacked report doesn't include the holes of the missing fields.

Conclusion

Although you might write a fair bit of code to create an FRX using the report object classes I presented in this document, the code is simple: create some objects and set their properties. It sure beats writing 50 INSERT INTO statement with 75 fields to fill for each. Please report (pun intended) to me any suggestions you have for improvements.

Summary

Object-oriented wrappers for menus and reports allow you to do things like dynamically altering menus at runtime and generating or updating reports as necessary. Feel free to use the classes presented in this document as you see fit and let me know of any suggestions for enhancements.

Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *The Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *The Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. He wrote over 100 articles in 10 years for FoxRockX and FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe.

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox conference (<http://www.swfox.net>). He is one of the administrators for the VFPX VFP community extensions Web site (<http://vfp.org>). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (<http://tinyurl.com/ygnk73h>).

