Chapter 3 What's Not So Obvious

So far we have discussed all major object-oriented concepts and how they are implemented in Visual FoxPro. However, Visual FoxPro supports a couple of features you won't find in other object-oriented languages. This chapter gives us the chance to discuss these and other issues that aren't as obvious as they might seem.

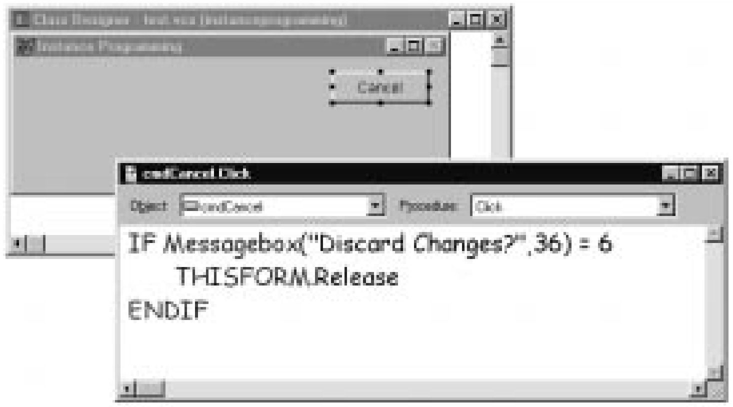
Instance programming and pseudo-subclassing

Visual FoxPro has features called *instance programming* and *pseudo-subclassing*. You may not have heard about them since they aren't widely discussed or generally known; however, almost everybody uses them without even knowing it.

This feature must be easy to use, but every now and then, the lack of specific knowledge brings up some serious problems. For this reason, I'd like to discuss this issue in detail, which should help you to use all the power of pseudo-subclassing and instance programming without running into their potential problems.

Instance programming

What exactly is instance programming? Well, let me give you a little example. Let's assume we create a form class that contains one command button. We add some code to the command button to release the form when the button is clicked. **Figure 1** illustrates this scenario.

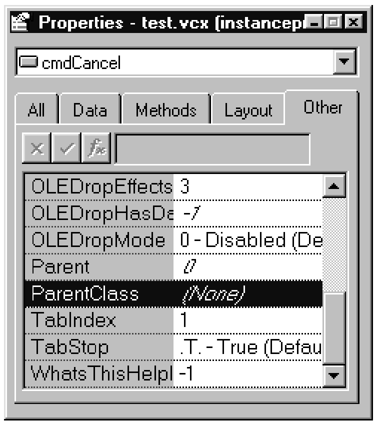


Object: cmdCancel Procedure: Click

```
IF Messagebox("Discard Changes?",36) = 6
    THISFORM.Release
ENDIF
```

Figure 1. *Our form class and command button.*

To review: We created a new form class (let's call it *IP* for instance programming), which is a subclass of the FoxPro base class *Form*. The *IP* class has a member object called *cmdCancel*, which is a command button. This button is a subclass of … well … let's have a look at the Properties window in **Figure 2**.

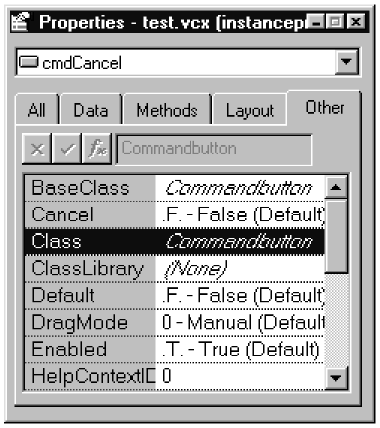

Properties - test.vcx (instancep

cmdCancel

All | Data | Methods | Layout | Other

OLEDropEffects	3
OLEDropHasDa	-1
OLEDropMode	0 - Disabled (De
Parent	0
ParentClass	(None)
TabIndex	1
TabStop	.T. - True (Defau
WhatsThisHelpI	-1

Figure 2. *The Properties window.*


Properties - test.vcx (instancep

cmdCancel

All | Data | Methods | Layout | Other

Commandbutton

BaseClass	Commandbutton
Cancel	.F. - False (Default)
Class	Commandbutton
ClassLibrary	(None)
Default	.F. - False (Default)
DragMode	0 - Manual (Default
Enabled	.T. - True (Default)
HelpContextID	0

Figure 3. *The current class of the Properties window.*

Hmmm … the Properties window says it doesn't have a parent class. That's strange. After all, I told you we couldn't create a class that didn't have a parent class. So what is the actual class? See **Figure 3**.

FoxPro says the current class is *CommandButton*. That's strange, too! We just created a class that doesn't have a parent class, and the current class name is a FoxPro base class. Did we create such a class?

In fact, we didn't! All we created was a subclass of the FoxPro form *class* that had a predefined instance of a command button. There was no subclassing going on with the button. It is just an *instance* of the *CommandButton* FoxPro base class. So theoretically, we shouldn't be able to add code to one of the button's methods. Rather, we should create a button class, add the code there, and drop the button on the form. Everything else would break all kinds of object rules, because we would add code to an object rather than to a class.

However, adding code directly to object instances allows us to develop prototypes and actual applications at tremendous speed; it's one of FoxPro's most important Rapid Application Development (RAD) features.

By the way, creating FoxPro forms is 100 percent instance programming because not even the form itself is subclassed.

Pseudo-subclassing

Now let's take this a step further and create a subclass of the whole *IP* form class. This class (let's call it *PS* for pseudo-subclassing) inherits everything from the *IP* class, including properties, methods, and all predefined member objects plus their assigned properties and methods.

Take note that only the form has been subclassed, just as in the *IP* class. The button is still of class *CommandButton* and doesn't have a parent class (well, it has, but only an internal one). However, FoxPro is very generous and will still allow you to add code to the button. In fact, you can even overwrite and inherit code just as if it were a real subclass. This is called *pseudo-subclassing*.

Most people use the terms *pseudo-subclassing* and *instance programming* interchangeably. However, they are not the same thing. Internally, pseudo-subclassing and instance programming are handled very differently. As you'll see later (when I explain how visual classes are stored internally), visual classes are stored in FoxPro tables. Each class gets one record in this table, as do all the newly defined member objects. FoxPro stores the code that is added to an instance (instance programming) in each member's record. When applying pseudo subclasses, the new class inherits all the information about the members but does not create a record for each of them, so FoxPro doesn't have a good place to store the new code and the overwritten properties. So it uses a little trick and stores this data directly with the class record. To assign the code and properties to the member objects, FoxPro adds the name of each object. So the button's Click method is now called "Function cmdCancel.Click" rather than "Function Click" as it would be in normal scenarios.

Of course, the user never sees these things, but he might experience some resulting problems. Let's try to fool FoxPro a little. To do so, we go back to the *IP* class and rename the Cancel button *cmdCancelButton*. Now let's look at the button in the *PS* class. The button is still there and it inherited the new name from its parent class. However, for some strange reason, all the code we assigned to this button has been removed. On first sight, this might appear to be a bug, but if you think about it, it's rather simple. As we've just discussed, FoxPro stores the name of the member object in order to assign the code to it. The code that has been removed belonged to an object called *cmdClick*. Of course, this object can't be found anymore because we renamed it. The subclass cannot know that the button in the form is still the same object. It thinks we removed the original object and added a new one. If we want to recover the original code, we need to cancel the current operation, go back to the parent class, and rename the button to its original name. Make sure you don't save the subclass. Otherwise FoxPro simply removes all your code and you'll have to start over from scratch.

This behavior might seem bad, but it used to be worse. In earlier versions, FoxPro thought the class library was corrupt whenever it couldn't find referenced objects, and wouldn't allow us to modify it at all. Imagine if you renamed an object in a class at the beginning of a class hierarchy. You might have to start your whole project from scratch again. Considering these facts, the current situation appears quite acceptable. After all, the advantages outweigh the disadvantages by far, especially when you're aware of the possible problems.

But wait, there's more! If you use DoDefault() in the member object's method, you can move on to the next paragraph, but if you use the scope resolution operator (::), you are in deep trouble. The scope resolution operator requires the name of the parent class, the method name, and possibly some parameters. But as we already know, the command button doesn't have a parent class, so we can't provide the necessary information for the scope resolution operator. This is one of the reasons why DoDefault() was introduced in Visual FoxPro 5.0. However, there are some tricks to make this work, even if you use Visual FoxPro 3.0, which didn't have DoDefault(). Here's an example that uses the previous example to demonstrate how this would work:

```
IP.cmdCancel::Click()
```

This code would go in the Click() event of the button. Remember I told you FoxPro simply stores the methods with the class and adds the name to assign the code to each object? We can now use this fact to our advantage and add the object name in front of the message name. This way, FoxPro can identify the code we try to run and execute it, even if the syntax doesn't seem to match the normal scope resolution requirements.

However, I recommend using DoDefault() instead, even though you might take a bit of a performance hit. I think the ease of maintenance outweighs that by far. Keep in mind that neither instance programming nor pseudo-subclassing is truly object-oriented. They are shortcuts to make the developer's life easier and more productive. If you want to avoid all the troubles this might introduce, you could go the truly object-oriented route. You could create a form class, create a button class, and finally create a subclass of the form class and change nothing but drop the button on it. Whenever you wanted to change the behavior of the button, you would create a subclass of the button and another subclass of the original form, and drop the button on this form as well. This becomes a nightmare if you have many member objects. Imagine a form that has only five members. Depending on how you want to change the behavior, you could end up with as many as 25 different subclasses of the form. Now imagine you have 25 different objects rather than five. In this case, you might end up with 625 different subclasses. This, of course, would be the case only when you wanted to change the behavior of each object independently from all the other objects, which is improbable.

I think you can see the issue, and I believe it's well worth it to accept the disadvantages of instance programming and pseudo-subclassing instead.

Visual vs. non-visual classes

One of the great features of Visual FoxPro is its Visual Class Designer. It allows you to create classes in a visual way rather than deal with a huge amount of source code. However, you don't have to use the designer. You can always go the source code route; by doing so, you gain a couple of advantages.

Advantages of non-visual classes

Let's examine a couple of reasons for using source-code-only classes. First, they might be a bit faster. According to my own measurements, classes that are stored in PRGs instantiate about twice as fast as classes that are stored in VCX libraries. This changes from scenario to scenario, but it seems that source code classes are a lot faster, especially for first-time instantiation. Once an instance of a class has been created, FoxPro caches the class definition, and the speed seems to be about equal.

The reason for the speed disadvantages of visual classes is due to the way they are stored. VCX files are simply DBFs with a different extension. When a class gets instantiated, FoxPro scans the table, looks for compiled code that is stored in the table, and checks for information about possible parent classes. If parent classes are found, they have to be searched as well. Classes that are stored in PRG files are one huge chunk of compiled code, and VFP doesn't have to add all the overhead of SEEKing classes and their inheritance information.

Another factor that needs to be considered is class size. The larger the class, the less the difference in instantiation speed. The reason is simple. The bigger the class, the longer it takes to instantiate it. If it takes 50 milliseconds to instantiate a class, it doesn't really matter that it took three milliseconds to search the VCX and only one to find the definition in the PRG file. However, if it only takes a millisecond or two to instantiate the class, an additional overhead of three milliseconds matters a lot.

A couple of handling issues seem to be resolved a lot better in source code classes. Include files (.H extension files), for instance, are hard to handle in visual classes, and you're also limited to a single include file at a time. Further, you cannot use precompiler commands wrapped around methods or property definitions.

Another issue is size. Visual class libraries have a tendency to grow huge, because every time they are modified, FoxPro deletes the old version of your class and adds it again at the bottom of the file. Also, a table with memo fields is always a little bigger than a plain text file. Source code classes, on the other hand, are very compact. Furthermore, because source code classes are stored in regular text files, file corruption isn't a big issue. VCX files, on the other hand, are more fragile and might get corrupted every now and then.

The final advantage I want to point out is the ease of renaming classes, properties and methods, and the ease of redefining class structures. However, this advantage might also turn into a disadvantage because renaming and redefining can lead to other problems further down the road.

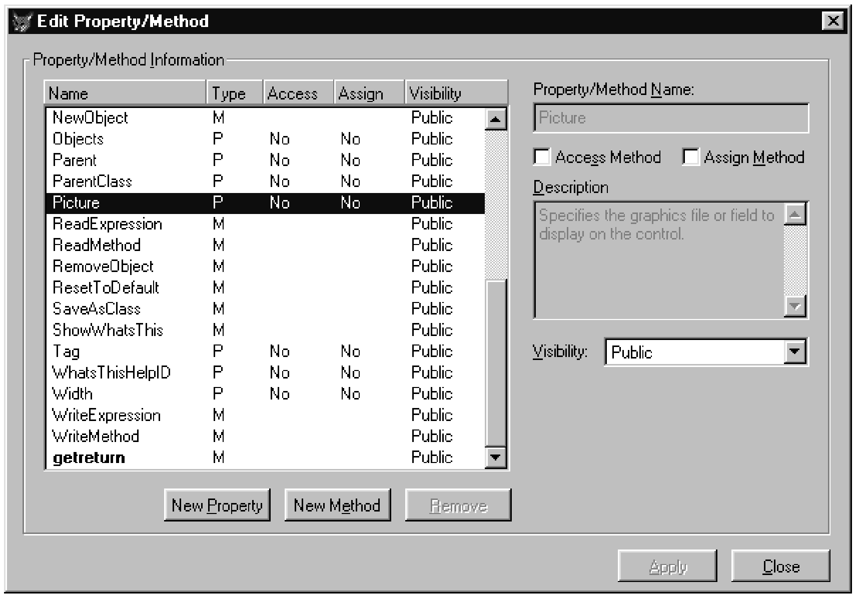
Why not go the visual route?

Going the visual route has many advantages. Almost all of them are a result of the proper internal organization of classes in each library. It's easy to browse source code on a per-class basis. Viewing class hierarchies and inheritance trees is supported by many tools like the Class Browser. Visual design tools consolidate properties and methods from classes and superclasses and display all the available ones in a properly ordered list called *Property-Sheet*.

Using non-visual classes, you're on your own with all these issues. Non-visual class libraries can be a big mess of code that has no particular order or organization whatsoever. Properties and methods are spread over the whole file, possibly even over multiple files, and there is no way to see them all at once. This makes it easy to forget about them, or even to redefine them accidentally, since there is no integrated mechanism to warn you about a possible problem. This gets even trickier if you want to use predefined events. In the property sheet, you can simply pick one of the available events and add code to it. In source code, this isn't so easy. You have to remember the event names for each class and where each class was derived from. Otherwise, you have to look it up, which is a very time-consuming process.

The Visual Class Designer takes care of all the stupid little standard tasks like adding objects to containers, setting properties, and adding code to methods. Following the concept of "information at your fingertips," the whole class definition is broken into pieces to show only the part of the class you're currently working on. This enhances productivity a great deal so you can concentrate on the essentials of programming, which is resolving a business problem—*not* taking care of technical issues.

The majority of the visual design tools have been enhanced in Visual FoxPro 6.0. The Visual Class Designer got a whole new dialog to manage methods, properties and member objects, as shown in **Figure 4**.



Name	Type	Access	Assign	Visibility
NewObject	M			Public
Objects	P	No	No	Public
Parent	P	No	No	Public
ParentClass	P	No	No	Public
Picture	P	No	No	Public
ReadExpression	M			Public
ReadMethod	M			Public
RemoveObject	M			Public
ResetToDefault	M			Public
SaveAsClass	M			Public
ShowWhatsThis	M			Public
Tag	P	No	No	Public
WhatsThisHelpID	P	No	No	Public
Width	P	No	No	Public
WriteExpression	M			Public
WriteMethod	M			Public
getreturn	M			Public

Property/Method Name: Picture

☐ Access Method ☐ Assign Method

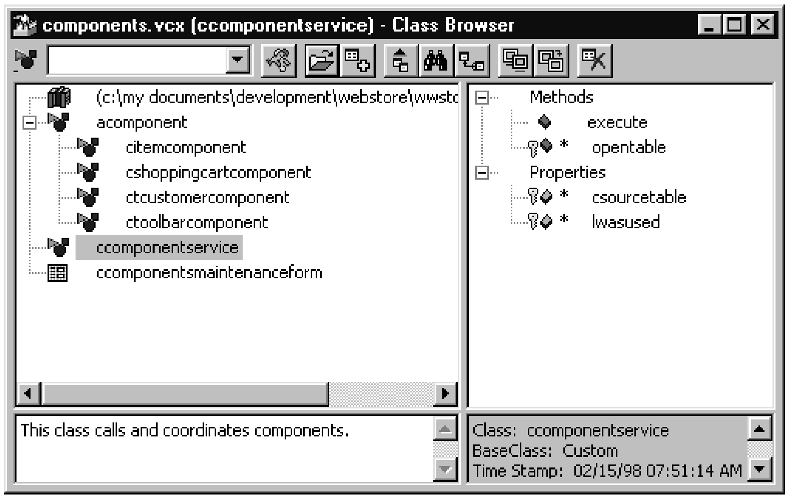
Description: Specifies the graphics file or field to display on the control.

Visibility: Public

Figure 4*. The Visual Class Designer's new Edit Property/Method dialog for managing methods, properties and member objects.*

Using this dialog, you can create and delete new properties and methods, specify member visibility, and create access and assign methods. After using this dialog for a couple months, it was hard for me to imagine going back to Visual FoxPro 5.0 and living without it.

Another great tool that has been around since the first version of Visual FoxPro is the Class Browser. It has changed a lot since then; it's become easier to use and more powerful at the same time. The new browser also has a slightly different look and feel, as illustrated in **Figure 5**.



(c:\my documents\development\webstore\wwstc
acomponent
citemcomponent
cshoppingcartcomponent
ctcustomercomponent
ctoolbarcomponent
ccomponentservice
ccomponentsmaintenanceform

Methods
execute
* opentable
Properties
* csourcetable
* lwasused

This class calls and coordinates components.

Class: ccomponentservice
BaseClass: Custom
Time Stamp: 02/15/98 07:51:14 AM

Figure 5. *The new Class Browser.*

The browser allows you to view classes in hierarchical or alphabetical order, even across class libraries. It also displays class details such as properties, methods and class documentation. We'll examine the Class Browser in more detail in Chapter 5.

Of course, all these tools are only in addition to the centerpiece, which is the Visual Class Designer itself. It consists of four main components: the Class window, the Code Snippet Editor, the Properties window and the Classes/Controls toolbar. See **Figure 6**.

I do not intend to explain the details of the Class Designer because many other people have spent a lot of time doing that already. I think the advantages of this visual design tool are rather obvious.

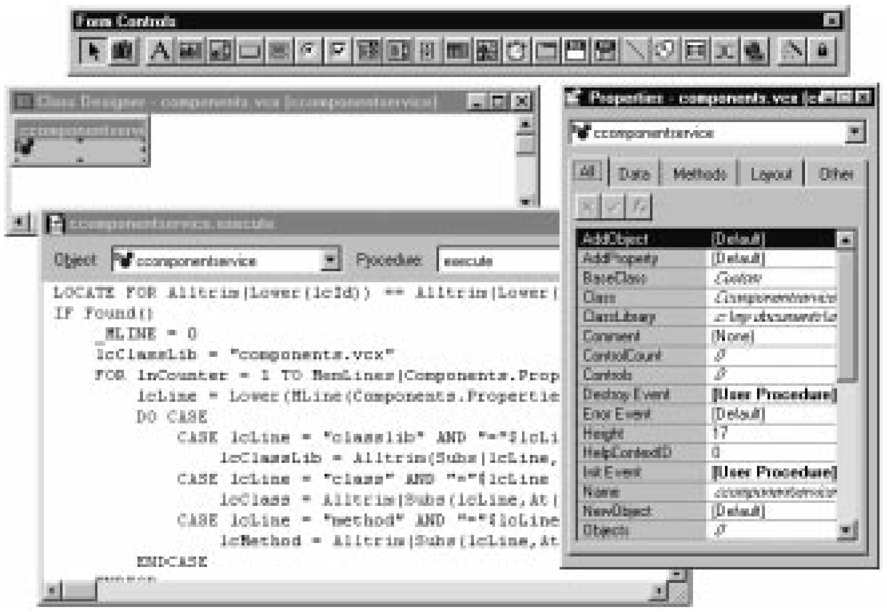
A further advantage of visual classes is the ability to create builders. Builders can automate tasks while designing a class. The concept of builders is unique to Visual FoxPro. It is based on the fact that FoxPro always uses live objects in the Visual Class Designer. This means that you can talk to classes programmatically, as if they were objects. This way you can assign properties, add code to methods, and instantiate new member objects. Despite the fact that this is an extremely interesting topic, I will leave this one to another book in this set.

Also, keep in mind that most third-party tools are optimized for visual class libraries. Even some tools Microsoft provides rely on VCX storage. A typical example would be the Modeling Wizards that I'll discuss in Section 3 of this book.

Visual classes: Nintendo for adults?

Here are some of the arguments I keep hearing: "Real programmers don't use visual design tools" and "The Visual Class Designer is like a video game for adults." To make a long story short: *I couldn't agree less!* (This is sort of like the old adage "Real programmers don't use code generators.")

Does a person become a better programmer because he's able to specify a default value for a property in source code rather than in a Properties window? I don't think so! Does one become a better programmer because he is able to identify a method in a huge PRG file rather than getting to it with a double-click in the Visual Class Designer? I don't think so! Does one become a better programmer because she can add member objects programmatically rather than dropping them in a container by a simple mouse operation? I doubt it! Does one create more efficient code when creating PRGs than when using VCXes? Well, maybe! But even if non-visual classes have a slight performance advantage, other issues outweigh that by far. Creating user interfaces in a non-visual way is quite a nightmare and the results are usually rather ugly. And even for non-interface classes, productivity and handling benefits are overwhelming.


Form Controls
Class Designer - components.vcx (ccomponentservice)
ccomponentservice
ccomponentservice.execute
Object: ccomponentservice
Procedure: execute
Properties - components.vcx (cc
ccomponentservice
All
Data
Methods
Layout
Other

```
LOCATE FOR Alltrim(Lower(lcId)) == Alltrim(Lower(
IF Found()
    _MLINE = 0
    lcClassLib = "components.vcx"
    FOR lnCounter = 1 TO MemLines(Components.Prop
        lcLine = Lower(MLine(Components.Propertie
        DO CASE
            CASE lcLine = "classlib" AND "="$lcLi
                lcClassLib = Alltrim(Subs(lcLine,
            CASE lcLine = "class" AND "="$lcLine
                lcClass = Alltrim(Subs(lcLine,At(
            CASE lcLine = "method" AND "="$lcLine
                lcMethod = Alltrim(Subs(lcLine,At
        ENDCASE
```

AddObject	[Default]
AddProperty	[Default]
BaseClass	Custom
Class	Ccomponentservice
ClassLibrary	[illegible]
Comment	(None)
ControlCount	0
Controls	0
Destroy Event	[User Procedure]
Error Event	[Default]
Height	17
HelpContextID	0
Init Event	[User Procedure]
Name	ccomponentservice
NewObject	[Default]
Objects	0

***Figure 6**. The four main components of the Visual Class Designer.*

In the end, the only difference between good and bad programmers is the resulting application they produce, and in order to create a good application, highly productive programmers are needed. At the same time, code quality has to remain high.

Visual design tools, property sheets, code snippet editors and the Class Browser are outstanding tools that raise a programmer's productivity and help to maintain code quality at the same time. Let's be more productive!

Some classes are non-visual only

Unfortunately, not all classes are available in the Visual Class Designer. Among the ones that can only be edited in source code are *Pages* (not *PageFrames*), *Grid Columns* and *Headers*. However, you can subclass all these classes in source code.

Many of the classes that can't be modified in a visual way are specialized containers that can only live in certain other containers. Pages, for instance, can only live in PageFrames, and Columns can only live in Grids. Nevertheless, almost any kind of object can be contained in these classes. Usually you'd modify the container classes in the Visual Class Designer and set some kind of property to instantiate these specialized member objects. In Grids, for instance, you can simply set the *ColumnCount* property and FoxPro will add new columns on the fly. However, all the added columns are of the FoxPro base class *Column* and can't be of a special user-defined class. If you want to add your own column class, you can define that in source code, set the *ColumnCount* property to 0 and add the columns on the fly (at runtime) using the AddObject() method. Because columns have to have some member objects, these must also be instantiated on the fly or defined in the source code. No matter how this is done, you always lose the advantages and power of the visual design tools.

Creating your own set of base classes

Using Visual FoxPro base classes directly without subclassing is a big no-no. You should subclass each base class before instantiating it or using it in the Form Designer. This adds a lot of flexibility to the design. You can always go back later, change a couple of properties, modify some behavior or add new methods. This is especially important when starting with Visual FoxPro, because it makes a project more forgiving—you can always go back and correct mistakes you made earlier in the cycle.

Creating your own set of classes makes it possible to make system-wide changes within a matter of minutes. Let's assume you discovered a bug that influences your whole system. I just had such a bug. I used the InteractiveChange and the ProgrammaticChange events to discover record pointer movement and other changes in text fields. However, sometimes this event wouldn't fire, so I added an assign method for the *Value* property in my textbox base class that fired an OnChange() method. I basically created my own system-wide event that would fire whenever the *Value* property changed. This helped to resolve a problem that had us hooked for months. After making this change (which took me only a couple of minutes), I was able to remove about 50 items from our bug-tracking system. I could do this only because I had created my own "top-level" entry point.

Using your own base classes does more than help to resolve your own mistakes and problems. You can also use them to change standard behavior or appearance. Maybe you don't like FoxPro's default font. No problem! Go to your base class and change it. It only takes a couple of minutes…

This first level of subclasses is usually referred to as your own set of *base* or *foundation* classes. Once you have this class library in place, you can basically forget about FoxPro's original base classes. Unfortunately, there is no way to tell FoxPro to display these classes instead of the internal base classes. This leaves us with the risk of using the wrong set of classes, which might result in hard-to-find bugs.

Fortunately, there are a couple of tools that check class libraries and force the use of certain base classes. The PowerBrowser is one of these tools. It comes with some wizards that deal with all kinds of base class issues. This tool is freeware and can be downloaded from the Developer's Download Files at www.hentzenwerke.com. Future updates to this tool will be available at www.eps-software.com.

Some suggestions

When creating your set of base classes, you can take care of some issues you might run into in a later stage of your development cycle.

I'm always concerned with creating applications that provide an interface the user is familiar with. Usually I stick to the Microsoft Office standards. One of the first steps to meet these standards is to change the standard font for all controls from Arial to MS Sans Serif or to the newer Tahoma. Font size should be 8 points. However, this could lead to problems if the user runs large fonts. In this case I switch back to the Arial font, the logic for which is built into my base classes.

I also try to make sure all objects have a consistent programming interface. Unfortunately, many of the FoxPro base classes do things a little differently. Some have a Release method, some don't. Some containers support an *Objects* collection while others have specialized ones like *Forms* or *Pages*. Some objects have Show and Hide methods while others only have a *Visible* property. I can take care of all these issues right in my base classes and save myself a lot of headache later down the road.

I think you get the idea about what kind of things belong in your set of base classes. Keep in mind that all the changes you made are subclassed into every single class you use in your project. Therefore, you should be concerned about performance. Adding 10 milliseconds to the instantiation time of a textbox might end up adding another second or two when instantiating a complex form.

Having a powerful set of base classes can add a lot to your application and make it flexible for changes later on. Nevertheless, you should be very careful with the changes you make. A little change in a base class not only can fix a system-wide bug, but it also can introduce one. Keep in mind that one small change can affect your whole application.