Chapter 16 Extending the Framework

Once you have become comfortable with the basics, it is time again for new challenges. Extending the Web Connection framework classes and creating some of your own classes can significantly improve both your productivity and the quality of your applications.

Most of the important classes in the Web Connection framework can be easily subclassed when needed. In this chapter I will present the proper method for creating and installing such subclasses. I will also show several examples of useful extensions to the Web Connection framework. But first, I want to make sure that all readers are comfortable with the mechanics of subclassing in general in Visual FoxPro. The next section presents a primer on this subject. Developers who are already comfortable with these concepts can skip over this material.

A subclassing primer

To be successful as a developer of Web Connection applications, it is essential to become comfortable with object-oriented programming (OOP) in Visual FoxPro. As a developer, if you are new to both the Web and OOP, you will face a steep learning curve. After mastering the basics with Web Connection, you will encounter needs to create subclasses of your own. In this section I will show the basics of defining a subclass in code. With Visual FoxPro, you have the choice of creating your classes and subclasses with either the Class Designer, which is a visual tool, or with code alone. I am only going to discuss creating classes in code, because the Web Connection classes that you will be interested in subclassing are defined in code.

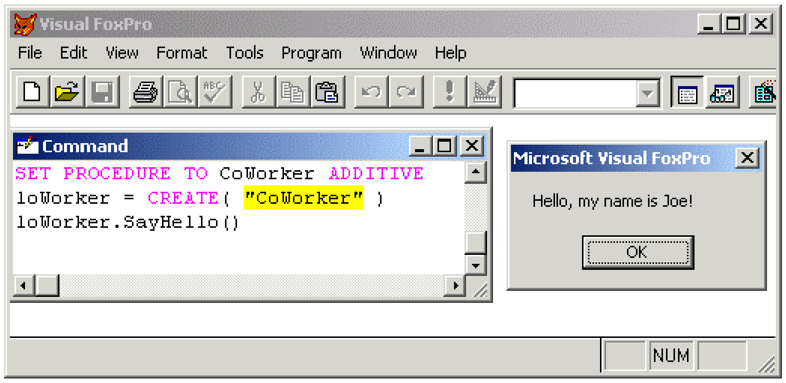
When you create a class, you start with a suitable Visual FoxPro base class, and then define all of its additional properties and methods from scratch. Properties are like variables that allow an object to have attributes assigned to it (such as height, width, color, and so forth). Methods are functions or procedures that are contained (or encapsulated) within an object. When you create a subclass, you are instead taking some existing class as a starting point and extending its properties and methods. The class from which you start is known as the parent class, or super class, if you are familiar with that terminology from another language. You extend and alter the parent class definition in the subclass. Any property or method in the parent that you do not specifically alter in the subclass is “inherited” by the subclass. You can make any combination of five types of alterations in a subclass:

- Add new properties.
- Override the initial values of parent properties.
- Add new methods.
- Completely override parent methods.
- Augment parent methods, while also calling the parent method code.

To see these five types of changes in action, you must start with a parent class—the class from which your subclass will be derived. Here is an example of a parent class named CoWorker that defines two properties and two methods:

```
DEFINE CLASS CoWorker AS CUSTOM
* Properties:
WorkerName = "Joe"
Efficiency = 1.0
* Methods:
FUNCTION SayHello
  MESSAGEBOX( "Hello, my name is " + THIS.WorkerName + "!" )
ENDFUNC
*
FUNCTION ListProperties
  ? "Name: " + THIS.WorkerName
  ? "Efficiency: " + TRANS( THIS.Efficiency)
ENDFUNC
ENDDEFINE && CLASS CoWorker
```

Class definition code is placed in PRG files just like other Visual FoxPro code. You can include one or more classes in a single PRG file. In this example class with two custom properties and two custom methods, you might create a file with the same name as the class (that is, CoWorker.PRG). You can easily deploy and test this class from the Visual FoxPro Command Window (see **Figure 1**).



```
SET PROCEDURE TO CoWorker ADDITIVE
loWorker = CREATE( "CoWorker" )
loWorker.SayHello()
```

Hello, my name is Joe!

Figure 1*. Deploying the parent class.*

Now that you have created a working parent class, it is time to look at an example of creating a subclass that includes all five of the types of changes listed previously. Here is such an example, which you could package in the same PRG file with the parent, or in its own separate file:

```
DEFINE CLASS LazyCoWorker AS CoWorker
* (1) Add a new property:
LazinessQuotient = 10
* (2) Override a parent property:
Efficiency = 0.5
```

```
* (3) Add new method:
FUNCTION TakeCoffeeBreak( lnSeconds )
  DECLARE Sleep IN WIN32API INTEGER
  Sleep( 1000 * m.lnSeconds )
  MESSAGEBOX( "I'm awake now and raring to go." )
ENDFUNC

* (4) Completely override a parent method:
FUNCTION SayHello
  MESSAGEBOX( "Yawn... My name is " + THIS.WorkerName + ;
    ", and I sure am lazy." )
ENDFUNC

* (5) Augment a parent method:
FUNCTION ListProperties
  DODEFAULT() && call parent method
  ? "Laziness Quotient: " + TRANS( THIS.LazinessQuotient )
ENDFUNC
ENDDEFINE && CLASS LazyCoWorker
```

This is a complete working subclass that includes five modifications to the parent class. Notice how little source code a class can require. There are important factors to understand about each of the five types of changes in the subclass:

1. The first change was to add a new property called *LazinessQuotient*. When doing so, only the subclass and its descendents can reference this property. Thus, if you instantiate an object based on the parent class CoWorker, you could not refer to its *LazinessQuotient* property. When you discover that a parent class really requires a property that only exists in a subclass (for example, you discover that all co-workers can be lazy on occasions), you can simply add the property back to the parent class definition.
2. The second change was to override the value of a property that was defined in the parent—to change *Efficiency* from 1.0 to 0.5. This is a very frequent type of change to make in a subclass.
3. The third change was to add a brand-new method called TakeCoffeeBreak(). This is very similar to adding a new property—only objects derived from the subclass can utilize this method.
4. The fourth change was to completely override the SayHello() method from the parent. In this case, the interface to any object derived from the subclass remains the same as for those derived from the parent class; it is the implementation that changes. You still can call to loWorker.SayHello(), for example, and if you had passed parameters to the parent SayHello() method, you would pass the same parameters to the SayHello() method of the subclass, provided you included the same PARAMETERS or LPARAMETERS statement at the top of both the parent and subclass.
5. The fifth and final change is to augment a parent method, in this case the ListProperties() method. You do this when you want to retain some of the parent class behavior, while adding some additional code. The technique for implementing this is to call the DODEFAULT() function in the subclass. You have the flexibility of

calling this anywhere in the method code, although most commonly this will be either the first or last line of code.

Subclassing Web Connection framework classes

Web Connection provides significant functionality in its framework classes. When starting out, you will find this functionality almost overwhelming, such that subclassing is the last action you would consider. Nevertheless, after working with the framework for some time, you may encounter situations, such as tasks that are repetitive in nature, where you think framework extensions are in order. You may also discover behavior that you would like to create and fine-tune in one place and have it apply to your entire application, or to all Web Connection applications in your domain. In this section I will discuss several of the Web Connection framework classes and which of those are particularly amenable to subclassing.

In fact, if you have explored the sample code or created a project of your own, you can see that you are already working with subclasses of two of the framework classes. First, the main program for your project consists primarily of a subclass of the server class (wwServer). Second, each process program consists of a subclass of the process class (wwProcess). These two examples are cases of "implementation" subclassing, meaning that they represent a specific usage in a single application. Every Web Connection developer performs this type of subclassing. A more interesting form of subclassing involves actually extending the Web Connection framework for reuse. When you do this, you create functionality from which each of your implementation subclasses can benefit. This chapter focuses more on this latter concept of extending the framework itself.

An example extension to the response class

In each case where you identify a need to extend the Web Connection framework, there are at least four steps for you to perform:

- Define the need.
- Identify the correct framework class to be extended, if any.
- Implement the design.
- Integrate the design into your application.

Because I want to focus on the mechanics of successfully implementing a framework extension, I will start with a very simple example. In Chapter 18, "Advanced Troubleshooting and Maintenance," I discuss the advantage of using comments in your code, but with an unusual twist—by inserting the comments into your HTML output stream, you get the "double impact" of seeing the same comment in both your Visual FoxPro source code and in the HTML source for the page. This can be very convenient when debugging HTML layout and formatting problems.

To embed a comment into your HTML response, you could issue a command similar to the first line here:

```
Response.Write( [<!-- Beginning of TABLE for query results. -->] + CR )
Response.Write( [<table border=1 bgcolor="gray" cellpadding=2>] + CR )
```

Now suppose that you start embedding comments like this liberally in your code. If you are like me, you will find it repetitive to type that syntax for HTML comments each time. Further, you may not even remember the syntax and have to look it up in your trusty, dog-eared HTML reference. You realize that efficiency would be gained if you had a simple method for inserting comments into the HTML output. You have thus completed the first step of defining the need.

The next step is to identify the correct framework class to be extended. Sometimes, this will be obvious. Other times, you will not be certain, and will have to analyze the situation more closely. Sometimes there will not even be a good answer, which might suggest the need for designing a new class from scratch. The more you study and understand the Web Connection messaging model, the easier it is to identify the proper class to be extended.

Fortunately, this example fits into the first category. You see from the preceding code that it is the Response object that you wish to have the extended behavior of providing a simple interface for adding comments. The requirement is to have a method that is more specific than the Write method to the needs of inserting comments into the HTML output. WriteComment() might be a good choice of name for this method.

Now that you have identified the object whose behavior you want to enhance, you must discover the correct class name from which to derive your subclass. By examining the Web Connection documentation, you will discover that the Response object is defined in the wwResponse class. Thus you implement your design change by extending that class in a subclass called MyResponse, which you create in a new file named MyResponse.PRG as follows:

```
#INCLUDE WCONNECT.H
DEFINE CLASS MyResponse AS wwResponse
FUNCTION WriteComment( lcText )
  THIS.Write( [<!-- ] + m.lcText + [ -->] + CR )
ENDFUNC  && WriteComment
ENDDEFINE  && CLASS MyResponse
```

That's all it takes! You now have a class that allows easy insertion of HTML comments without your having to remember and type that syntax. And due to inheritance, your new class still has the Write method and all of the other methods in the base wwResponse class. Using the new WriteComment() method in your application code is simple:

```
Response.WriteComment( "Beginning of table for query results.")
```

You are probably thinking that it cannot be quite that easy, and you are correct. You still have to get this change installed into your application so that your class is recognized and used. If you did nothing more, your application would continue to implement the framework wwResponse class, rather than your new MyResponse class, which would mean the Response object would not have a WriteComment() method at all. There are two steps required to get your class recognized and used.

First, you must add the class to the active set of defined classes at run time. Because this class is defined in program code, the required statement is SET PROCEDURE..ADDITIVE. Thus you insert the following line near the top of your application's main program file, right after the line DO WCONNECT:

```
SET PROCEDURE TO MyResponse ADDITIVE
```

This step serves only to make your class available for use. How do you tell Web Connection to use it in place of its own wwResponse class? The answer lies in the use of compiler constants in the sometimes mysterious WCONNECT.H file. This file contains numerous constants for a variety of uses. Some are available for developer modification and some should never be touched! One set of constants that is definitely meant for developer modification is the one that is specifically included to facilitate subclassing. These constants all start with the prefix WWC_ , and in this case the one you are interested in is WWC_RESPONSE. If you examine the WCONNECT.H file, you will find this setting:

```
#DEFINE WWC_RESPONSE                    wwResponse
```

This setting is used by the Web Connection framework to know the correct parent class to use when implementing framework-specific objects. Thus all you need to do in order to have your class used is to change the definition to:

```
#DEFINE WWC_RESPONSE                    MyResponse
```

For the purpose of testing this simple example, you can make this change directly to the WCONNECT.H file. In the section that follows, I will present the preferred way of changing these constants, which minimizes maintenance problems when new revisions of Web Connection are issued.

You'll also notice that I included a #INCLUDE WCONNECT.H line at the top of MyResponse.PRG. This is because I'm using the constant CR in my code instead of having to type out CHR(13)+CHR(10). Any #INCLUDE compiler directive applies only to the current PRG file, so it is necessary to repeat this line in each PRG where you intend to use these constants.

One last detail to point out is that our WriteComment() method does not alter the output stream directly, but rather uses the existing Write() method and simply supplies the specifics that format a comment for HTML. Because the Write() method already encapsulates the proper mechanism for sending strings to the output stream, you leverage this code rather than trying to duplicate it. This is more important than it might sound. The Web Connection framework actually deploys further subclasses, wwResponseFile or wwResponseString, based on whether file-based messaging or COM messaging is in use. Each of these subclasses has different code to implement the Write method. By simply calling this method, you allow these differences to be handled for you automatically, rather than writing messaging-specific code yourself. In fact, you do not even need to be aware that two different mechanisms exist.

Managing changes to #DEFINE constants in WCONNECT.H

In the previous example, you had to change a #DEFINE constant in WCONNECT.H in order to implement a subclass of a Web Connection framework class. In this section I will explain why this is necessary. I will then present a more maintainable approach for altering these constants.

The important concept is why you need these constants at all. In other words, if you are subclassing the wwResponse class, why isn't:

```
DEFINE CLASS MyResponse AS wwResponse
```

sufficient in and of itself? Why do we need to change WCONNECT.H at all? The answer lies in the internal workings of the Web Connection framework. In many cases, including that of the Response object, another framework object is responsible for instantiating and manipulating that object. In this case, it is the Process object that instantiates the Response object. The problem is that other Web Connection framework objects have no knowledge of the details or even the existence of your subclass. Therefore, how could they know to instantiate your subclass instead of the framework class?

The answer, as you can probably guess, comes from the compiler constants. Each relevant framework class has an associated constant that defines the class name to be used. In the case of the wwResponse class, the constant is WWC_RESPONSE and its default definition is:

```
#DEFINE WWC_RESPONSE    wwResponse
```

Now for the trick! Rather than ever instantiating the wwResponse class directly, the constant is always used instead. Thus the code might look like:

```
THIS.oResponse = CREATEOBJECT( [WWC_RESPONSE] )
```

This technique of providing an indirect way of referring to the class name provides all of the flexibility you need to implement your subclassing. All you need to do is revise the value of the constant, and the framework will now use your subclass in all places where wwResponse was previously used. This is very elegant and powerful.

The second important concept is maintainability. Obviously WCONNECT.H is a file provided with the Web Connection framework. As such, it is subject to change from one revision to the next. In fact, it always changes with each revision, because one of the constants in WCONNECT.H defines the Web Connection revision number. If you also make several changes to this file, how will you ensure that these changes are not lost when you install a new revision of Web Connection? Further, suppose you have two or more separate Web Connection applications, and these applications do not use the same subclasses as one another. How would you manage the constants in that scenario?

The answer to both situations is *not* to make the changes in the WCONNECT.H file itself, but instead in a separate header file of your own for each application. This file will contain all of the values to override from the framework defaults. The secret is to ensure that your revised constants be used. After all, the framework classes will not know about your new header file. The trick to making this work is to insert these lines at the *very bottom* of WCONNECT.H that reference your own header file:

```
#IF FILE( "WCONNECT_OVERRIDE.H")
  #INCLUDE WCONNECT_OVERRIDE.H
#ENDIF
```

Starting with version 4 of Web Connection, the preceding lines are included by default. If you are working with prior versions, simply add the lines manually. Now, you can create your header file WCONNECT_OVERRIDE.H, which will need two lines for each class that you want to subclass. For the example I presented in the previous section, your header file would look like:

```
* WCONNECT_OVERRIDE.H
*   Application-specific overrides to WC framework constants.
*
* Subclass the Response class:
#UNDEF WWC_RESPONSE
#DEFINE WWC_RESPONSE    MyResponse
*
* more subclassing as required..
```

Note the use of the #UNDEF compiler directive. This causes the previous definition of this constant to be dropped, prior to substituting your own definition. Failure to include this line will produce a compile-time warning "Constant is already created with #DEFINE."

If you are managing multiple applications, this approach will still work. If you need different subclass definitions in each application, simply move the individual WCONNECT_OVERRIDE.H files into application-specific paths.

There is one potential problem you must be aware of when working with multiple Web Connection applications, if you also subclass framework classes. You must be very careful when testing these applications from the Command Window by simply running your main program file. If you do this after switching from one application to another, the framework classes may have been last compiled using the overrides from a different application, possibly leading to problems that could be very difficult to diagnose and debug. This is not an issue when running your compiled EXE files. This problem can be avoided, allowing you to test successfully from Visual FoxPro, by first recompiling all of your classes when switching to a different application.

Framework classes suitable for subclassing

Now that you have seen an example of how to implement a subclass, and you know how to manage the header files that control the compilation process properly, it is time to examine when you would want to create these subclasses.

Web Connection provides many framework classes. Some of these are well suited to subclassing, while others should probably never be considered. In this section, I will examine each of the major Web Connection classes plus a few of the supporting classes and discuss their suitability for subclassing.

Subclassing the wwServer class

The wwServer class is the basis for the primary Server object that runs your application. You always create a new subclass of this class in the main program of each Web Connection application. As such, it is trivial to add functionality to this object for a specific application, simply by adding the code to the class implementation in the main program. Beyond this basic extension, you can also create a subclass of wwServer from which each of your applications derives its server class. This would be appropriate if you identified additional Server object functionality that you wanted to apply to more than one of your applications.

From a completeness standpoint there is little reason to extend the wwServer class. This class already performs all of the functions needed to respond to requests from WC.DLL and to call the process classes that you design. Little could be added to this mechanism. Nevertheless, there is one tempting reason to add functionality here, and that involves persistence. Whereas the Request, Response, Process, and Session objects are instantiated and destroyed on every hit, the Server object persists until your EXE terminates. Because of this, the server becomes a tempting target for performing various tasks, even though they may be unrelated to Web Connection messaging. Tasks such as opening files and setting up the environment can be performed here.

Beyond that level, you are probably better off designing separate worker classes and instantiating them from the Server object, thus providing the persistence, but not adding to the complexity of the server class itself. The best place for adding such code is the SetServerProperties() method, which is always called from the Web Connection framework before the first hit is processed.

Subclassing the wwRequest class

The wwRequest class is used to encapsulate the incoming request for each Web hit. Although it is suitable for subclassing, for most situations there is no reason to do so, because it already contains the basic functionality required to read information about the incoming request. Some examples of reasons to subclass this class include:

- If you install a third-party ISAPI filter that alters the information received by WC.DLL, you might need one or more methods to facilitate parsing out this additional information.
- If your developers are all trained in Active Server Page (ASP) technology, you might want to revise the Form method in Web Connection so that it can handle multiple-selection popups in the same manner as ASP (via repeated calls to the Form method, rather than using the Web Connection GetFormMultiple() method).
- If your application includes any forms that include file upload controls, the wwRequest class is missing two methods that are needed to allow more flexibility in those forms. If you have multiple-selection popups on the same form, Web Connection does not include a method for reading the popup values. There is also not a method for verifying a form variable’s existence. Neither the GetFormMultiple() nor the IsFormVar() method works with “multi-part” forms, which are the type used when uploading files. Following is a subclass that includes the needed methods. The methods GetMultipartFormMultiple() and IsMultipartFormVar() can be used for multi-part forms in exactly the same way that their counterparts work for basic forms.

```
DEFINE CLASS WebRadRequest AS wwRequest
* Sub-class of Web Connection wwRequest.
* ------------------------------------------------------- *
FUNCTION GetMultipartFormMultiple(taVars,tcVarName)
* Adapted by Randy Pearson from other methods in wwRequest class.
*
* This method retrieves multipart, multiselect HTML form variables
* from the request buffer into an array.
*
* Multipart form variables are submitted on the client side by
* specifiying an encoding type of "multipart/form-data":
```

```
* <form METHOD="POST" ENCTYPE="multipart/form-data">

* Parameters:
* @taVars
*  An array that will receive the form variables. Pass by reference!!
* tcVarname
*  The name of the form variable to retrieve.

* Returns:
*  Numeric - count variables retrieved into the array.

* Example:
* DIMENSION laVars[1]
* lnVars=Request.GetMultipartFormMultiple(@laVars,"LastName")
* --------------------------------------------------------- *
LOCAL xx, lcValue, lnAt, lcFind, lcPointer
xx=0
lcPointer = THIS.cFormVars
lcFind = [NAME="] + m.tcVarName + ["]
lnAt = ATC(m.lcFind, m.lcPointer)
IF m.lnAt = 0
   RETURN 0
ENDIF

* Following is required as of WC 3.20, which adds new handling of
* multi-part borders:
IF EMPTY(THIS.cMultiPartBorder)
   THIS.GetMultiPartBorder()
ENDIF

DO WHILE m.lnAt > 0
  lcValue = Extract( @lcPointer, ;
    tcVarName + ["] + CHR(13) + CHR(10) + CHR(13) + CHR(10), ;
    CHR(13) + CHR(10) + "--" + THIS.cMultipartBorder)
  * Before WC 3.20, was: **  CHR(13)+CHR(10)+"---------"
  xx = m.xx + 1
  DIMENSION taVars[ m.xx]
  taVars[ m.xx] = m.lcValue
  lcPointer = SUBSTR( m.lcPointer, m.lnAt + LEN( m.lcFind))
  lnAt = ATC( m.lcFind, m.lcPointer)
ENDDO

RETURN m.xx
ENDFUNC  && GetMultipartFormMultiple

* --------------------------------------------------------- *
FUNCTION IsMultipartFormvar()
* Created by Harold Chattaway.
* Determines whether a form variable name was part of
* the current request submittal.
LPARAMETER lckey
LOCAL lcMultiPart, lnLoc
lcMultiPart = THIS.cFormVars
lnLoc = ATC([NAME='] + m.lckey + ['], m.lcMultiPart)
IF m.lnLoc=0
  RETURN .F.
ELSE
  RETURN .T.
```

```
ENDIF
ENDFUNC  && IsMultipartFormvar
* -------------------------------------------------------- *
ENDDEFINE  && WebRadRequest
```

Subclassing the wwResponse class

In the example already presented in this chapter, you have seen the mechanism for subclassing the wwResponse class to enhance the methods available for simplifying the creation of HTML output. This class is very suitable for subclassing, and many Web Connection developers do so. You should weigh these situations carefully. This is a lightweight class with good performance. If you were to burden the class with properties and methods to automate every aspect of HTML, this performance would suffer. On the other hand, additional methods can improve your development productivity.

You need to find a good balance here. I suggest using the default Web Connection class for your first few projects, and moving to subclassing only after becoming convinced of the need to improve your productivity. Another alternative is to create helper classes that are instantiated only when the need arises. The wwShowCursor and wwDbfPopup classes, which are provided with Web Connection, are examples of such classes.

Subclassing the wwProcess class

The wwProcess class is at the heart of your application. If you develop more than one application, you are bound to identify ways that you want to extend this class to suit your own style and requirements. I have found almost limitless possibilities in this area. Just a few of these ideas are presented in the section “Extending the wwProcess class” later in this chapter.

Subclassing the wwSession class

The wwSession class provides a convenient tool for managing state in Web Connection applications. Chapter 13, “Identifying Users and Managing Session Data,” presents this class in detail. Although the class can be subclassed, doing so is not simple, particularly if you want to support moving the session data to SQL Server using the wwSessionSQL class. Furthermore, the potential reasons for doing so are relatively minor.

The most frequent reason I have identified is when you have a session variable that you need to reference in most if not all of your process methods. Session variable names and values are stored in XML format in the Vars memo field of wwSession.DBF, which must be parsed to read the value. You could improve the performance of the Session object in this case by adding a field to the data structure that is specific to this session variable, thus allowing its value to be read directly, rather than by parsing through the Vars field. However, in order to use this technique, you must still create a method for storing data to and reading data from that field. It does not happen automatically. Nevertheless, one look through the source code for these classes will probably convince you not to implement such a change. There are several monolithic methods that embed the entire data structure in such a way that you would be forced to copy and paste the entire methods into your subclass for the purpose of addressing the one or two additional fields that you want to add. This would put you in a precarious position each time a new version of Web Connection is released.

Subclassing the wwHtmlHeader class

The wwHtmlHeader class in an optional class that provides an easy way to encapsulate the otherwise manual process of creating the detailed <head> section of an HTML response. This class already contains convenient methods for adding JavaScript, cascading style sheets (CSS) links, and some other frequent needs. It does not, however, encapsulate everything you might want to include in this section. Although you can address any further needs via the built-in AddMetaTag() method, or by addressing the *cHeadSection* property directly, these require that you know the often arcane syntax of the items that can be placed in this section of an HTML document. In addition, the AddMetaTag() method covers only the use of the *name* attribute in <meta> tags, while there are also needs to use the *http-equiv* attribute in some cases (such as adding content rating information). As you might be guessing, I find this a ripe area for subclassing.

Here is a simple subclass that adds some further functionality to this class:

```
#INCLUDE WCONNECT.H
DEFINE CLASS MyHtmlHeader AS wwHtmlHeader

* --------------------------------------------------------- *
FUNCTION AddMetaEquivTag
* Adds a META tag to the header.
LPARAMETERS lcEquiv, lcValue
THIS.cHeadSection = THIS.cHeadSection + ;
  [<meta http-equiv="] + m.lcEquiv + [" content="] + ;
  m.lcValue + [">] + CRLF
ENDFUNC  && AddMetaEquivTag

* --------------------------------------------------------- *
FUNCTION AddKeywords
* Adds keywords to the header for indexing information.
LPARAMETERS lcKey
THIS.AddMetaTag( "keywords", m.lcKey )
ENDFUNC  && AddKeywords

* --------------------------------------------------------- *
ENDDEFINE  && Class MyHtmlHeader
```

Your new class can easily be tested from the Command Window. The following shows some of the new functionality combined with some of the built-in framework functionality:

```
SET PROCEDURE TO MyHtmlHeader ADDITIVE
SET PROCEDURE TO wwHttpHeader ADDITIVE  && includes wwHtmlHeader
loHead = CREATE( "MyHtmlHeader" )
loHead.AddTitle( "My Special Web Page" )
loHead.AddStyleSheet( "styles/myAppStyle.css" )
loHead.AddKeywords( "widgets, cheap, best" )
? loHead.GetOutput()
```

The output produced is as follows:

```
<html>
<head>
<title>My Special Web Page</title>
<link rel="stylesheet" type="text/css" href="styles/myAppStyle.css">
```

```
<meta name="keywords" content="widgets, cheap, best">
</head>
<body>
```

Finally, I should note that you do not need to invoke the familiar technique of altering a define constant in order to implement this subclass. This is because the wwHtmlHeader class is an optional class for your own use and is never invoked directly by the Web Connection framework. Thus all you need to do is make certain that you instantiate your subclass rather than the framework class.

Subclassing the wwShowCursor class

The wwShowCursor class is a RAD tool that can be used to quickly convert a Visual FoxPro cursor to an HTML table with one row for each record in your cursor. This is a very handy tool for producing content with almost no code during the early stages of development in your applications. However, in many cases developers end up wanting more control over final appearance than the wwShowCursor class allows. There have been several questions posted on the West Wind support forums about the possibility of creating a subclass in order to gain this control.

Unfortunately, attempting to create such a subclass is not practical in most cases. An examination of the source code for wwShowCursor reveals a few monolithic methods that produce the bulk of the work for this class with very little use of customizable properties or hooks for added functionality. The only way to add control over the formatting would be to completely override the huge ShowCursor() method, and probably the BuildFieldListHeader() and ShowRecord() methods as well. Before undertaking this task, you should carefully evaluate either creating your own class from scratch, or hand-coding SCAN loops to generate your tables. This latter approach is more labor-intensive, but provides you with total control over the appearance of your application, which can be essential when those change requests are made.

The zero-maintenance subclass

You have now seen both the approach for creating subclasses and some of the candidate classes that can be altered to suit your needs. One point to remember is that there is some maintenance involved in using your own framework extensions. Always consider that there may be one other option: If your need is sufficiently generic that other Web Connection developers would also benefit from its inclusion, consider making a suggestion to West Wind Technologies that the framework itself be modified. Over the years I have recommended dozens of improvements to the Web Connection framework, many of which have been adopted. The best subclass is no subclass at all.

Extending the wwProcess class

As I discussed earlier, the wwProcess class is a great candidate for subclassing. Why is this? The Process object is at the heart of your application. It is a point of orchestration (or mediation) between the incoming request and the outgoing response. It is where you perform between 70 and 90 percent of your work (depending, for example, on whether you have separate business classes). There should be little surprise then that many opportunities to extend the framework are created at this point.

In this section I will present two extensions that could be used in any Web Connection application. The first is a Web-based assertion mechanism that can be very convenient for streamlining your development process. The second is a Web-based substitute for the ubiquitous MESSAGEBOX() function in Visual FoxPro. In each of these examples, the overriding design requirement is to simplify the coding for the Web Connection developer who uses these methods. In other words, these are ideal framework methods.

A Web-based assertion mechanism

The wwProcess class has a very convenient method to handle aborting out of one of your methods if a problem is identified. This method is called ErrorMsg(). It provides two services:

1. It aborts the current page, discards any content that has been generated up to that point, and ensures that no further output is sent to the user.
2. In place of the expected response, it displays a Web page back to the user with a description of the error or other condition that occurred. When deployed by the developer, this message can contain customized content explaining the situation.

A typical hypothetical example of how you might deploy this method would be:

```
IF NOT User.IsAdmin()
  THIS.ErrorMsg( "System Message", ;
    "Only administrators can perform this function." )
  RETURN .T.
ENDIF
.. remainder of processing code
```

As this example suggests, you might use this function frequently for security checks, to handle error conditions, and so forth. In fact, the Error() method of wwProcess calls the ErrorMsg() method to display a page to the user if an unhandled error occurs in your application.

What more could you ask than this? Well, first of all, you have already written four lines of code, and all you have accomplished is to advise the user of the error condition. It is very likely that, in this type of example, triggering this message is indicative of either a logic failure (this user should never have had a link to this page in the first place) or perhaps a hack attempt on your Web site. In either case, you definitely want to take other action. At minimum, you would want to log the event, but more likely you would want to alert a system administrator immediately, probably via email. Fortunately, there is the SendErrorEmail() method available to handle this situation. Thus you can alter your code as follows:

```
IF NOT User.IsAdmin()
  THIS.SendErrorEmail( "Invalid Access", ;
    "A non-administrator attempted to gain access to this page!" )[1]
  THIS.ErrorMsg( "System Message", ;
    "Only administrators can perform this function." )
```

[1] This code assumes you are already connected to your mail server. If your development machine requires a dial-up connection and you are not already connected, you'll need a command like wwipstuff:rasdial to establish a connection before this command.

```
  RETURN .T.
ENDIF
.. remainder of processing code
```

This is fine, but now you have a reasonable chunk of code to handle just one, possibly unlikely, scenario that you need to check. Web development presents countless needs to perform checks like this, given the stateless nature of the Web, sophisticated hacker tools, and so forth. The more code you have to write to check for conditions that may seem unlikely or impossible, the less likely you are to implement the checks. If you get lazy about performing such checks, you will be sorry in the long run. What is needed is a more convenient way for the developer to perform checks like the one shown previously.

When I tackled this need, I decided to create the programmatic interface that would do the best job and then develop the implementation to make that interface work. The solution is patterned after the Visual FoxPro assertion technique, wherein you can simply insert a single line in your code, such as:

```
ASSERT <condition>, <message>
```

What you do here is assert that a condition is true, and if not, display the message. If the condition is true, processing continues with the next line of code. If not, the program aborts with the message shown. This is not precisely what you want in your Web applications, but it is very close. It is not acceptable to be providing failure messages on the server or to quit the application, but you do want to stop further processing of the current page, and you do want to display a message back to the user. Thus, I decided that in the simplest form the developer should be able to enter a line of code like this:

```
THIS.Assert( <condition>, <message> )
```

In this simple form, if the condition is not true, a simple Web page should be returned to the user displaying the message, *and further processing of the page should be aborted*. The second factor is very important, because once you have identified an invalid condition of this nature, you want to ensure that this user does not perform the indicated task.

First, I will show a method that can be added to the wwProcess class to support this basic interface. After introducing that class, I will add some optional parameters to handle some interesting variations. Here is the first revision of the new method:

```
DEFINE CLASS MyProcess AS wwProcess
FUNCTION Assert( llCondition, lcMessage )
IF VARTYPE( m.llCondition) <> "L" OR NOT m.llCondition
  * assertion failed!
  THIS.ErrorMsg( "Assertion Failure", m.lcMessage )
  RETURN TO Process
ENDIF
ENDFUNC && Assert
ENDDEFINE  && CLASS MyProcess
```

This is all it takes to add the basic mechanism. Note the RETURN TO line, which aborts any possible further processing of your own method and returns to the point from which your method was called.

To test this, you need to install this subclass as described at the beginning of the chapter. This includes adding the following line to your main program right after the line "DO Wconnect":

```
SET PROCEDURE TO MyProcess ADDITIVE
```

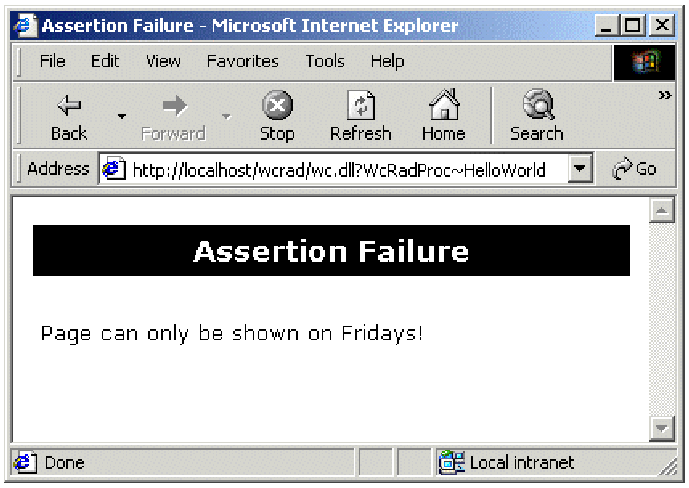
Next, you need to make sure your class is recognized and used. Either choose a method in one of your current Web Connection applications (such as the "Hello World" method in the example application). At the top of the program file that includes this method, add these lines:

```
#UNDEF  WWC_PROCESS
#DEFINE WWC_PROCESS    MyProcess
```

Now you are able to test this basic mechanism. Open one of your Process methods and insert a line of code such as the second line of code here:

```
FUNCTION HelloWorld
THIS.Assert( CDOW(DATE()) = "Friday", "Page can only be shown on Fridays!")
*...remainder of method
```

Now fire up your browser and navigate to the page in question. For this example, I used the basic "Hello World" example that is provided with Web Connection. The result is shown in **Figure 2**.


Assertion Failure - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address http://localhost/wcrad/wc.dll?WcRadProc~HelloWorld
Go
Done
Local intranet

Assertion Failure

Page can only be shown on Fridays!

Figure 2*. Users get a blunt message when they don't belong here.*

You can probably see the advantage in using this function: Because it aborts from your method when the assertion fails, you need only enter a single line of code to check a critical condition and keep the user out if that condition is not met. I will often have five or more calls to the Assert() method near the beginning of most of my methods. Would I be so meticulous if each of these checks required several lines of code? Probably not!

I will now finish up the presentation of the assertion mechanism by adding three more features to enhance its power and flexibility:

- Notice that the result page in Figure 2 does not provide the user with any place to go. The method needs an optional URL parameter that, if provided, causes a hyperlink to display after the message.
- There needs to be a flag to control whether the assertion failure is sufficiently serious to require the generation of an email notice to the system administrator.
- There needs to be the ability for additional information to be provided to the system administrator that would not be appropriate to display to the user (typically for security reasons).

Here is the final Assert method, shown within the class definition:

```
DEFINE CLASS MyProcess AS wwProcess
FUNCTION Assert
LPARAMETERS llCondition, lcMessage, lcUrl, llEmail, lcExtraInfo
IF VARTYPE( m.llCondition) <> "L" OR ;
  NOT m.llCondition  && assertion failed!
  *
  IF m.llEmail  && send alert message to admin
    THIS.SendErrorEmail( "Assertion Failure", ;
      "Message Presented: " + m.lcMessage + ;
      IIF( EMPTY( m.lcExtraInfo), "", CHR(13) + CHR(10) + ;
        "Additional Info: " + m.lcExtraInfo ))
  ENDIF
  THIS.ErrorMsg( "Assertion Failure", m.lcMessage + ;
    IIF( EMPTY( m.lcUrl), "", [<BR><H3 ALIGN=CENTER><A HREF="] + ;
      m.lcUrl + [">OK</A></H3>] ))
  RETURN TO Process
ENDIF
ENDFUNC && Assert
ENDDEFINE  && CLASS MyProcess
```

There are plenty of ways to enhance this function and improve the displayed appearance. Then again, the messages should be displayed only when something goes wrong, so it does not need to be the flashiest part of your Web site! I am certain you can make extensive use of this method in your applications.

A Web-based MESSAGEBOX function

One area that every desktop application developer finds to be missing when they first start developing Web applications is the easy ability to interact with the user in a granular fashion by using functions such as the ubiquitous MESSAGEBOX() in Visual FoxPro. Although you can use the JavaScript alert and confirm functions to produce modal dialogs on the client system, these are seldom adequate substitutes for server-side needs. In addition, some browsers do not support JavaScript, while others allow the user to disable script languages for security reasons.

Consider a simple example where the user can click a “Delete” hyperlink in order to delete a customer record from the database. Suppose that you are not willing to delete a record this important without receiving confirmation. As a seasoned desktop developer, you might be tempted to include this code in your Process method:

```
lnConfirm = MESSAGEBOX("Delete this customer?", MB_YESNO, "Confirm")
```

The result of this would be a modal dialog appearing *on the server* waiting for a non-existent user to answer the question. This is equivalent to crashing your application, probably not your intention. What you need is a technique for posing this question to the client and processing the answer that the client provides back on the server. First, I will show how to ask a question in a specific case, and then I will generalize the approach for use in the framework.

For this example, suppose you are creating a DeleteCustomer() method wherein the ID of the customer to be deleted is passed as a URL parameter. In order to determine whether the user has confirmed his or her intention to delete the record, an additional URL parameter will be used. The technique will be to examine the URL and see whether there is evidence that the question has been answered. If so, the record will be deleted or not, depending on the answer. If there is not an answer, it means the question has not yet been asked, so you stop and ask the question, and defer any decision until the next hit from the current user.

The code presented here uses the named constants that are typically used with the MESSAGEBOX() function. Before implementing the actual code, you need to ensure that these constants are defined. The following definitions are best placed in your WCONNECT_OVERRIDE.H file, or equivalent; however, for this example it will suffice to place them at the top of the current process program file.

```
*-- MessageBox parameters
#DEFINE MB_OK                  0       && OK button only
#DEFINE MB_OKCANCEL            1       && OK and Cancel buttons
#DEFINE MB_ABORTRETRYIGNORE    2       && Abort, Retry, and Ignore buttons
#DEFINE MB_YESNOCANCEL         3       && Yes, No, and Cancel buttons
#DEFINE MB_YESNO               4       && Yes and No buttons
#DEFINE MB_RETRYCANCEL         5       && Retry and Cancel buttons

#DEFINE MB_ICONSTOP            16      && Critical message
#DEFINE MB_ICONQUESTION        32      && Warning query
#DEFINE MB_ICONEXCLAMATION     48      && Warning message
#DEFINE MB_ICONINFORMATION     64      && Information message

*-- MsgBox return values
#DEFINE IDOK           1       && OK button pressed
#DEFINE IDCANCEL       2       && Cancel button pressed
#DEFINE IDABORT        3       && Abort button pressed
#DEFINE IDRETRY        4       && Retry button pressed
#DEFINE IDIGNORE       5       && Ignore button pressed
#DEFINE IDYES          6       && Yes button pressed
#DEFINE IDNO           7       && No button pressed
```

Now insert the following DeleteCustomer() method in your process program (for example, MyDemoProcess.PRG):

```
FUNCTION DeleteCustomer
LOCAL lnCust, lnAnswer, lcUrl
lnCust = INT( VAL( Request.QueryString( 'CustId')))
IF m.lnCust <= 0
```

```
  THIS.ErrorMsg( "No customer specified!" )
  RETURN
ENDIF
* [Code to confirm valid customer ID would go here.]
lnAnswer = INT( VAL( Request.QueryString( 'MsgBox')))
lcUrl = Request.GetCurrentUrl()
DO CASE
CASE m.lnAnswer = IDYES  && user said yes
  * [Actual code to delete the customer would go here.]
  THIS.StandardPage( "You have deleted customer #" + ;
    TRANSFORM( m.lnCust))
CASE m.lnAnswer = IDNO   && user said no
  THIS.StandardPage( "You elected <b>not</b> to delete customer #" + ;
    TRANSFORM( m.lnCust))
OTHERWISE
  THIS.StandardPage( "Confirm Customer Deletion", ;
    "Do you really want to delete customer #" + ;
    TRANSFORM( m.lnCust) + "?<P><P ALIGN=CENTER>" + ;
    [<A HREF="] + m.lcUrl + [&MsgBox=] + TRANSFORM( IDYES) + [">] + ;
    '<B><LARGE>[Yes]</LARGE></B></A>' + [&nbsp;&nbsp;] + ;
    [<A HREF="] + m.lcUrl + [&MsgBox=] + TRANSFORM( IDNO) + [">] + ;
    '<B><LARGE>[No]</LARGE></B></A>' + [</P>] )
ENDCASE
ENDFUNC  && DeleteCustomer
```

To see this code in action, navigate your browser to access the DeleteCustomer() method. You can do this by first navigating to the HelloWorld method and then manually changing the URL. In my case, the virtual directory is /wcrad and the class name is WcRadProc, so the complete local URL is:

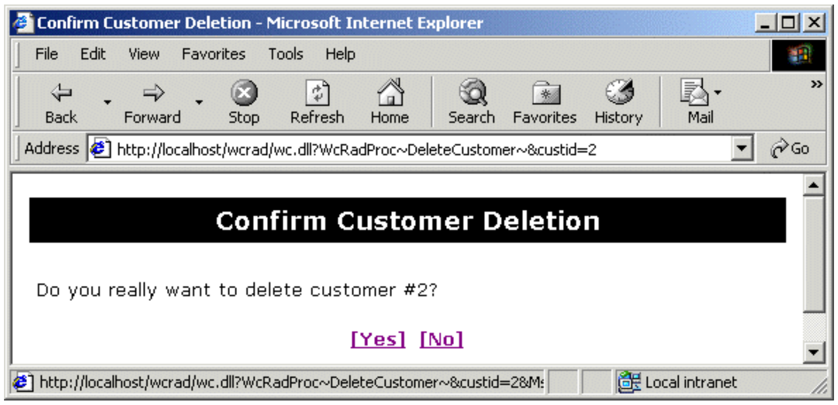
```
http://localhost/wcrad/wc.dll?WcRadProc~DeleteCustomer
```

If you enter that URL by itself, you will trigger the “no customer specified” error message. Bypass that validation check by appending a hypothetical customer ID to the URL:

```
http://localhost/wcrad/wc.dll?WcRadProc~DeleteCustomer~&custid=2
```

Your code now verifies that a customer has been specified, but that no answer has yet been provided to a confirmation question. Therefore, as depicted in **Figure 3**, the user is asked to confirm the deletion.

If the user clicks the Yes hyperlink, pay careful attention to what transpires. The URL that corresponds to this hyperlink is identical to the previous one, except for the addition of &MsgBox=6 at the end. In other words, the hit will again be processed by the same DeleteCustomer() method, but in this case one more variable—the answer to the confirmation question—will be included in the query string. If you follow the code for this case, you should not be surprised that the result of the client clicking Yes is the confirming result page shown in **Figure 4**. In a real application, of course, you would also include the code that performs the customer deletion.


Confirm Customer Deletion - Microsoft Internet Explorer

Address http://localhost/wcrad/wc.dll?WcRadProc~DeleteCustomer~&custid=2

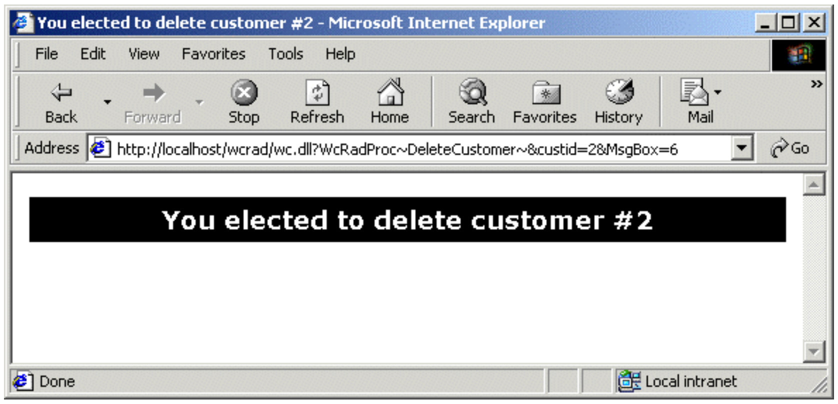
Confirm Customer Deletion

Do you really want to delete customer #2?

[Yes] [No]

http://localhost/wcrad/wc.dll?WcRadProc~DeleteCustomer~&custid=2&M:

Figure 3. *A confirming question is presented before performing a critical action.*


You elected to delete customer #2 - Microsoft Internet Explorer

Address http://localhost/wcrad/wc.dll?WcRadProc~DeleteCustomer~&custid=2&MsgBox=6

You elected to delete customer #2

Done

Figure 4. *A confirming message provides feedback that the task was completed.*

One other important aspect of the DeleteCustomer() method is the manner in which the URLs for the Yes and No hyperlinks were constructed. Rather than building the URLs manually, the GetCurrentUrl() method of the Request object was used, and we simply appended a parameter to the query string. Because of this, our code does not explicitly reference the current class or method name, and is therefore much more suitable for reuse.

Although the approach shown previously is fairly simple, there are some shortcomings. First, although the additional code to produce the confirmation question is brief, it is still not nearly as simple as a call to MESSAGEBOX. Second, the style and layout of the HTML confirmation page is embedded in the code. If your application included a large number of these confirmations, you would need to manually ensure the consistency of each one, which can be an onerous task if a global change to the appearance is requested. A more generic solution is needed.

As with the assertion mechanism presented earlier, I will start with the interface. The goal is to produce a method that is as similar as possible to the desktop usage of MESSAGEBOX. A typical use of the method might look like:

```
IF THIS.MessageBox( <question>, MB_YESNO) = IDYES
 * do Yes stuff
ELSE
  * do No stuff
ENDIF
```

Notice that the only difference would be to use THIS.MessageBox() in place of MESSAGEBOX(). Achieving this result would greatly simplify application coding, and would be a particular benefit to developers used to coding for desktop applications. The trick is to make it happen. After studying the code in the DeleteCustomer() method shown earlier, you may have concluded that this goal is not possible in a Web application. After all, the preceding snippet does not appear to cover the case where the question has not been asked yet. In other words, it appears to presume that either a Yes or a No answer is forthcoming. As you will see, you can indeed achieve the desired result.

The secret is in using the same mechanism that the framework ErrorMsg method (and our new Assert method) uses, which is the ability to abort page processing, if needed. In this case, if no answer is available in the URL, the page processing will be aborted and the question will be presented in place of the result page. When the user then clicks on one of the allowed answers, your same method will be triggered again, except that this time there will be an answer ready to use in your decision tree. Following is the first working version of a subclass of wwProcess that can support this approach:

```
DEFINE CLASS MyProcess AS wwProcess
* ---------------------------------------------------------- *
FUNCTION MessageBox( lcMessage, lnType, lcTitle)
*
* Web approximation to MESSAGEBOX() function.
LOCAL lcAnswer, lnAnswer, lnDialog, lnIcon
lcAnswer = Request.QueryString( "MBAnswer")
IF NOT EMPTY( m.lcAnswer)
  * Pop and restore any previous form vars.
  Request.cFormVars = Session.GetSessionVar( "PreviousFormVars")
ENDIF
lnAnswer = INT( VAL( m.lcAnswer))
lnDialog = BITAND( m.lnType, 15)
lnIcon = m.lnType - m.lnDialog
DO CASE
CASE m.lnAnswer <= 0
  * No answer yet.
CASE m.lnDialog = MB_YESNO AND INLIST( m.lnAnswer, IDYES, IDNO)
  * Valid answer to YES-NO question.
  RETURN m.lnAnswer
CASE m.lnDialog = MB_OK AND INLIST( m.lnAnswer, IDOK)
  * Valid answer to OK dialog.
  RETURN m.lnAnswer
* [Additional CASE's removed for brevity.]
ENDCASE

* Save any form vars while we ask the question:
Session.SetSessionVar("PreviousFormVars", Request.cFormVars)

LOCAL lcMsg, lcBtn, lcUrl
lcUrl = Request.GetCurrentUrl() + "&MBAnswer="
IF VARTYPE( m.lcMessage) = "C"
```

```
  lcMsg = m.lcMessage
ELSE
  lcMsg = ""
ENDIF
lcBtn = ""
IF INLIST( m.lnDialog, MB_OK, MB_OKCANCEL)
  lcBtn = m.lcBtn + [&nbsp;] + [<A HREF="] + m.lcUrl + ;
    TRANSFORM( IDOK) + [">OK</A>]
ENDIF
IF INLIST( m.lnDialog, MB_YESNOCANCEL, MB_YESNO )
  lcBtn = m.lcBtn + [&nbsp;] + [<A HREF="] + m.lcUrl + ;
    TRANSFORM( IDYES) + [">Yes</A>]
ENDIF
IF INLIST( m.lnDialog, MB_YESNOCANCEL, MB_YESNO )
  lcBtn = m.lcBtn + [&nbsp;] + [<A HREF="] + m.lcUrl + ;
    TRANSFORM( IDNO) + [">No</A>]
ENDIF
IF EMPTY( m.lcTitle)
  lcTitle = "System Message"
ENDIF
THIS.StandardPage( m.lcTitle, ;
  m.lcMsg + [<P ALIGN=CENTER><B><FONT SIZE=3>] + m.lcBtn + ;
  [</FONT></B>] )
RETURN TO Process
ENDFUNC  && MessageBox
* ------------------------------------------------------------ *
ENDDEFINE  && CLASS MyProcess
```

This class achieves our basic result, but there is still some work to do. First, I have included only a few of the standard dialog types. The complete version should include support for all types, including Abort-Retry-Ignore, and so forth. These are trivial extensions of the previous code and are omitted here to save space. The source code for this chapter, available at **www.hentzenwerke.com**, contains the full version.

This method also addresses one serious problem that I have not yet discussed. It considers the possibility of an HTML form submission. If we abort processing the page to ask a question, any form variables that were posted with the original request would be lost. This is clearly not acceptable. One solution would be to limit the use of our new MessageBox() method to only GET requests, and exclude POST requests. This would be too severe a restriction, because confirmation dialogs are often needed in response to form submissions. Instead, the solution is to employ the Session object to save the previous form variables while the question is being answered, and to restore the form variables afterward. This requires that you make sure you've added This.InitSession() to your Process method, or the Session variable won't be initialized.

Partitioning large applications

The first thing you will discover when you develop a Web Connection application is that the process program file can become very large. Even with some of the editor enhancements incorporated into version 7 of Visual FoxPro, this can become unwieldy.

What I always recommend is to create an application-level subclass of wwProcess (or your own framework extension) and then create subclasses of that in separate implementation PRG files. You can place common methods in the application-level class. These common methods might include Process(), Login(), standard look-and-feel items, and so forth. Each of the other PRGs will have its class definition based on this common subclass rather than on WWC_PROCESS.

Be forewarned that if you use script mapping in your Web Connection applications, and you divide your application into two or more process PRG files, you will need to define a separate script map extension for each, because that is the discriminator used in the main program to decide which program to call for each method.

You may also consider adding *properties* to the application-level classes. As an example, if some methods require the user to be logged in and others do not, you could add an lLoginRequired property and then split your methods between PRGs based on this division. I don't use that particular division technique, instead preferring to divide my methods into PRGs based on common functionality or modules. In either case, this is a highly recommended approach. While it is particularly important for projects with multiple developers, it is even useful for large applications when only a single developer is involved.

As an example, consider an application where you can divide all of the Process methods into one of two "modules." When you adopt this approach, your code might be structured as follows:

```
 * MainProc.PRG - This is the application-level subclass.
DEFINE CLASS MainProc AS WWC_PROCESS
cNewProp1 = ''
cNewProp2 = ''
FUNCTION Process
...your code here...
= DODEFAULT()
ENDFUNC
FUNCTION Login
...
ENDFUNC
ENDDEFINE && MainProc

* ModuleA.PRG
* Generic WC code that instantiates class:
LPARAMETER loServer
LOCAL loProcess
loProcess = CREATEOBJECT( "ModuleA")
loProcess.Process()
RETURN
*********************************
* Actual class definition:
DEFINE CLASS ModuleA AS MainProc  && inheritance
FUNCTION PageA1
ENDFUNC
FUNCTION PageA2
ENDFUNC
ENDEFINE

* ModuleB.PRG
etc...
```

To implement this approach, you first need to ensure that your MainProc class is available by including an appropriate SET PROCEDURE TO..ADDITIVE statement in the main program. Next you add one CASE for each module in the Process method of your main program. For example:

```
CASE lcParameter == "MODULEA"
  DO ModuleA WITH THIS

CASE lcParameter == "MODULEB"
  DO ModuleB WITH THIS
* etc.
```

This is but one possible strategy for splitting up larger applications. Others have been discussed on the West Wind message board. You should consider your situation and your development environment carefully when choosing any such strategy.

Creating your own framework classes and tools

This chapter has endeavored to show you where and how the Web Connection framework can be extended. Sometimes you will need something additional, but none of the framework classes appear to be the right home for the required functionality. In this case, you may decide to purchase a third-party product, or to develop your own classes. In most instances it is easy to integrate new components with Web Connection.

If you design your own components, make sure you can test them as easily as possible. It is very inefficient to debug components that can run only within a complete, running Visual FoxPro application, and this is even more so for Web applications. My initial goal when designing any new class is *to be able to test it from the Command Window*. This may sound ambitious or even impossible, but it doesn't have to be that way. If you think outside the box a little bit and use a little creativity, you can usually meet this goal.

Web Connection also gives you some handy tools that allow you to test your designs more easily, such as from the Command Window. Browsing through the wwUtils.PRG file will often reveal something you had not used before. My favorite time saver is the ShowHtml() function, which is great for testing any component that generates either HTML or XML. All you do is pass any HTML as a string and this function calls up your default user agent (browser) and displays the result. If Microsoft Internet Explorer is your browser of choice, this works particularly well, because this browser is very flexible in what it will render. For example, because Microsoft Internet Explorer does not even require a <body> tag, or even an <html> tag, you can just pass little snippets of HTML and see how a browser would render it. You can see this function in action from the Visual FoxPro Command Window:

```
SET PROCEDURE TO wwUtils ADDITIVE  && pre-requisite
ShowHtml( [Two different browsers are often like <ul>]+;
  [<li>apples and<li>oranges.</ul>] )
```

I don't think you could ask for anything much easier than that!

The second important issue is how to install your classes so they are accessible when needed in your application. Typically, all that is required is one of these commands, depending on whether you use the code editor or the visual tools:

```
SET PROCEDURE TO <PRG_File_Name> ADDITIVE
SET CLASSLIB TO <VCX_File_Name> ADDITIVE
```

In almost all cases the best place to insert this code is in the main program, immediately after the line "DO Wconnect".

Leveraging existing code

When developing Web applications with Web Connection, you may encounter situations where there is an existing desktop application with its own large code base. Even though Web applications work quite differently, there may be strong incentives to reuse as much of the existing code base as possible.

Although there are countless possible variations to this situation, there are a few general areas worth covering:

- Making use of your application object class
- Integrating business objects
- Using FPD/FPW legacy code and data

Where does my application object go?

Many Visual FoxPro developers use either third-party frameworks or their own code that makes use of a master application object. This object is assigned to a public variable (or private variable created in the main program) with a name such as goApp. This object offers various services and performs actions that hold the entire application together. Further, many other modules in the application refer to the application object using the global variable name. When developing a Web application, developers often ask where their application object fits in.

First, you should be aware that you are not required to have an application object at all. Web applications work very differently from desktop applications. Each hit to your Web application is likely to come from a different user from the previous hit, so your application cannot simply log one user in and assume nothing changes until a subsequent logout. You are also not creating an application menu, toolbar, forms manager, or any of several other items that are unique to the desktop and not applicable to a Web application.

Second, your application must avoid all user interface (UI) code, such as message boxes and other dialogs that might pop up in the Visual FoxPro application. Your code must run unattended on the Web server. Thus, you must hunt down and eliminate any such code, possibly replacing it in a subclass with a Web-friendly approach. The less monolithic the application, the easier this task should be. Classic n-tier application components often can fit in quite easily. One technique for trapping any UI code is to insert a SYS(2335,0) statement in the SetServerEnvironment() method in your main Web Connection program. This causes any statements that would otherwise generate UI to instead trigger an error. You can use this technique only if you do not want to display the Web Connection server status form.

Those considerations aside, you may still want to use an application object. If you have an application class that is sufficiently flexible to exist in the stateless world of a Web application, and it provides services that would be helpful to your application, by all means use it. Most likely you will need to re-architect it for Web use, or design a Web-specific subclass. The question becomes where to instantiate it within the Web Connection framework.

In order to avoid having the application object be instantiated and destroyed on every hit, which could be very bad for performance, you need to create it from the Server object. Here I will consider a hypothetical class that includes a special method called BeforeProcess() that you have designed for Web use that adjusts the object to the changing user identity. What you could do is add a property oApplication to the server class in your main program. Then, in the SetServerProperties() method, add this line:

```
THIS.oApplication = CREATEOBJECT( "MyApplicationClass")
```

You now have an object that can be referenced by Server.oApplication, but when you are processing each hit you might prefer to refer to this object by the familiar goApp variable name. This may even be a requirement if you are calling other functions or classes that already refer to this variable. Further, you need to ensure that your clever new method has been called before the hit is processed. To accomplish this, you could add something like the following code to the Process method of your process class:

```
PRIVATE goApp
goApp = Server.oApplication
goApp.BeforeProcess()
DODEFAULT()
```

This is deliberately vague and is based on a hypothetical application class. You must analyze your own situation to determine whether you have this need, and whether your application class is sufficiently flexible for Web use. The purpose of the preceding discussion is to point you in the right direction when you are trying to determine how to plug such an object into your Web Connection application.

Integrating business objects into Web Connection applications

Business objects are very suitable for Web Connection applications. The Process object in the Web Connection framework is where incoming requests are analyzed and responses are generated. This is also the point at which you would deploy business objects.

A design goal should be to avoid any direct data manipulation in your Process object. Instead, you should deploy business objects to accomplish the same results. The design and use of business objects is beyond the scope of this book, but an example of how they might be deployed in a Web Connection Process method could be useful. Consider first this fragment of code, which uses direct data manipulation:

```
FUNCTION SaveOrder()
*
IF !USED('Orders')
  USE Orders IN 0
ENDIF
SELECT Orders
SET ORDER TO cOrderID
LOCAL lcOrderID
lcOrderID = Request.Form("OrderID")
IF NOT SEEK(lcOrderID)
  THIS.StandardPage ( 'Order not found!')
  RETURN
```

```
ENDIF
IF Request.Form('Amount') > Customer.MaxOrder
  THIS.StandardPage( 'Exceeded limit.')
  RETURN
ENDIF
IF RLOCK()
  REPLACE …
* etc.
```

Notice the problems inherent in this approach. First, it performs all of its data access using Xbase code (USE, SET ORDER, SEEK, and so forth), which is fine if you use local Visual FoxPro tables, but leaves you with a nightmare if you decide to migrate the application data to Oracle or SQL Server. Also, the data manipulation code is mixed in with user interface code, thus preventing future scaling to an n-tier architecture. Finally, the business rules are intertwined with specifics of the Web application. If you also needed a desktop version of this application, you would need to repeat all of the business rules there, creating a maintenance nightmare.

Consider now the following code fragment, in which a hypothetical business object is included:

```
FUNCTION SaveOrder()
*
LOCAL loOrder, lcOrderID
loOrder = CREATEOBJECT( "BizOrder")
lcOrderID = Request.Form("OrderID")
IF NOT loOrder.FindRecord(m.lcOrderID)
  THIS.StandardPage ( 'Order not found!')
  RETURN
ENDIF
loOrder.Amount = Request.Form('Amount')
IF NOT loOrder.Validate()
  THIS.StandardPage( loOrder.cErrorMessage )
  RETURN
ENDIF
loOrder.SaveRecord()
* etc.
```

In this code, a lot of the work has been delegated to the BizOrder business object. There is no assumption that the data resides in any particular format. Business rules are assumed to be implemented in the business object. This process code is simply orchestrating the business object with the Request and Response objects to create the Web version of this functionality. There are two big advantages to this. First, if the data was migrated to Oracle or SQL Server, no changes would be required in your Web Connection application code. Any changes would be made in either the business object classes or in other classes that the business objects utilize (such as separate data access classes). Second, you could create a desktop application that uses the same business objects and be assured that all of the business rules would be consistent between the two applications.

Alternatives to designing your own business classes

Rather than designing your own business classes, there are two interesting alternatives. The first alternative is to use a commercial third-party Visual FoxPro framework in conjunction with Web Connection. If you already are familiar with one of these frameworks, you might want to investigate the suitability of its business class approach, if any, for use with Web Connection.

A second alternative, available starting with version 4 of Web Connection, is to use the wwBusiness class, which is provided with Web Connection as an optional utility class. wwBusiness is a light-weight class, which is more of a data access class than a true business class. In other words, its built-in methods are more oriented to the mechanics of reading and writing data from their sources, rather than providing a robust mechanism for enforcing business rules. Nevertheless, if you are looking for an easy object-oriented approach to data access that also allows easy migration from a Visual FoxPro to a SQL Server back end, this class could be worth investigating. Refer to the Web Connection Help documentation for detailed information on the use of this class.

West Wind also sells an add-on application that implements a Web-based store using Web Connection. This add-on uses the wwBusiness class for all of its data access. If you purchase this add-on, the source code can be very useful for learning one approach to integrating business objects.

When things go wrong

Although it is difficult to anticipate the specific problems that might occur, there are a few common difficulties that can arise.

Symptom: You get a Visual FoxPro “class definition not found” error.

Diagnosis: This can be one of two problems. The first is that you created a subclass but did not take the proper action to ensure that the class is available at run time. The simplest way to correct this is to insert a SET PROCEDURE TO <your class> ADDITIVE statement in your main program. This command can appear just after the DO WConnect line, which sets up the Web Connection framework classes. The second possibility is that you issued a SET PROCEDURE statement somewhere and failed to include the ADDITIVE clause. This causes all previously declared procedure files to be dropped.

Symptom: Your subclasses no longer work after upgrading to a new version of Web Connection.

Diagnosis: You have overwritten the previous copy of WCONNECT.H with the one from the new version, but failed to edit that file to contain the same #INCLUDE statement as before. See the previous section in this chapter, “Managing changes to #DEFINE constants in WCONNECT.H,” for specific information on this topic.

Symptom: You created a subclass, but its methods are not being invoked.

Diagnosis: If the object is instantiated by your own code, check to be certain that you are instantiating your subclass rather than the default framework class. If the object is instantiated automatically by the framework, either you have not provided the proper override to the constant in WCONNECT.H, or you have not recompiled the framework classes since doing so. This problem is more common when testing from the Command Window (DO MainProg). Correct the problem by compiling all framework classes:

```
COMPILE CLASS \wconnect\classes\*.vcx
COMPILE \wconnect\classes\*.prg
```

There may also be other problems if, for example, the Web Connection console application is running (Console.EXE). This causes compile copies of classes to be loaded into memory, which may prevent new copies from loading. The following commands help clear the Visual FoxPro environment:

```
CLEAR ALL
CLOSE ALL
CLEAR PROG
SET CLASSLIB TO
```

If this still doesn’t fix the problem, sometimes the only solution is to quit Visual FoxPro and start back up. Also, remember to test correctly from your browser. Often you may have to back up a few pages and refresh the content, or perhaps even start a fresh browser session, in order to have a clean scenario for testing your modifications.

Once you take the plunge and follow the path of extending the Web Connection framework, you also become more responsible for diagnosing problems that occur. Good troubleshooting skills are a must. See Chapter 18 for a general discussion of troubleshooting Web Connection applications. Also, remember that the West Wind support forums provide an opportunity to get help from other developers.