Error Handling Revisited

Doug Hennig

While objects have Error methods to provide local error handling, how do you provide common, global error handling services to your application? How do you recover when an error occurs? This month's Reusable Tools looks at a proven strategy for implementing error handling in Visual FoxPro applications, starting from individual controls and working up to a global error object.

In the June and July 1996 issues of FoxTalk, I discussed an error handling scheme for Visual FoxPro. While many of the ideas presented in those articles are still valid, some of them were just concepts and not actually usable code. Since then, I have refined the error handling scheme we use in production applications at Stonefield. This month's article presents an general error handling system as a reusable tool. Next month, we'll look at handling specific types of errors, especially those arising in data entry forms.

Designing an Error Handling Scheme

VFP supports both global (using ON ERROR) and local (an object's Error method) error handlers. These handlers represent opposite ends of a spectrum:

- The global handler is far removed from the source of the error while the Error method is part of the object that caused the error. Thus, the Error method should know a lot more about the environment it's in, the potential errors that could occur, and how to resolve them. For example, the CommonDialogs ActiveX control (which displays file, print, color, and printer dialogs) can cause an error if the user chooses Cancel. It'd be dumb to let the global error handler try to handle that error, since it would only see it as an OLE error of some kind and wouldn't know what to do about it. It makes more sense to put code into the Error method of the CommonDialogs control that knows how to handle a Cancel situation.
- The global error handler is called outside VFP's event handler using FoxPro's older "ON" event scheme (this scheme is still used by menus and on key labels). This means you can't use object syntax like THISFORM, and issues like private datasessions can complicate error resolution.
- The global error handler can efficiently consolidate error handling services (such as error logging and display) into one place. It's inefficient to try to handle most types of unanticipated errors (like a network connection going down) in the Error method of every object in your application.

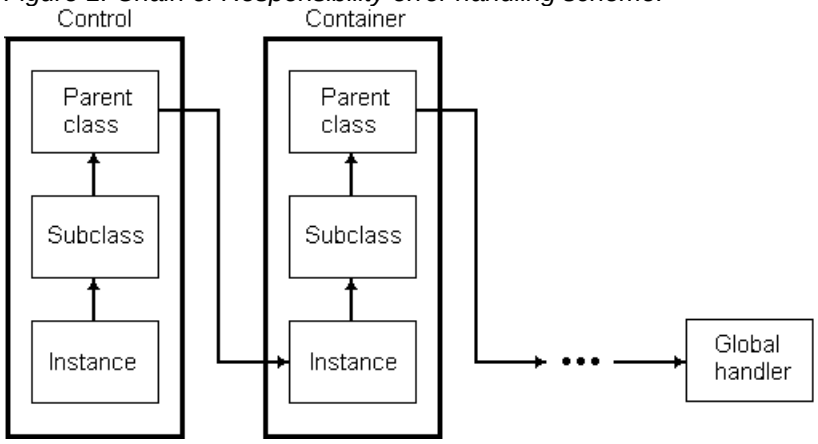
Let's look at a design of an error handling scheme that incorporates the best of both worlds. We want to handle errors in the most efficient manner possible, yet still provide the ability of individual objects to handle their own specific errors. Here's the strategy we'll use:

- Like most kids, an object knows more about what's really going on than its parents do, so the Error method of an object will handle any errors it can. It will pass those it can't handle up the class hierarchy using DODEFAULT(). If a subclass or instance of a class doesn't need to handle any specific errors, no code is placed in the Error method, causing the parent class code to automatically be used. Each subclass in the class hierarchy will do the same. Thus, SFDeepSubClassTextBox.Error calls SFSubClassTextBox.Error which calls SFTextBox.Error.
- The Error method of the topmost parent class for the object will handle any errors it can. It will pass those it can't handle to its container using This.Parent.Error.
- Because the container classes work the same as the control classes (they pass unhandled errors to their parent classes, and the topmost parent class passes errors to *their* container classes), the net effect is that we move up the class hierarchy then up the containership hierarchy.
- The Error method of the topmost parent class of the outermost container will handle any errors it can. It will pass those it can't handle to the global error handler.

This is a Chain of Responsibility design pattern: each object in the chain either handles the error or passes it on to the next object in the chain. In this multi-layered scheme, error handling gets less specific

and more generic as you move from the object to the global error handler, allowing errors to be handled at the appropriate level. Figure 1 illustrates this strategy.

Figure 1. Chain of Responsibility error handling scheme.



I decided to make the global error handler an object that's instantiated into the global variable oError from the SFErrorMgr class at application startup. One of its methods (ErrorHandler) is called both directly by objects as described above and indirectly since it's also the ON ERROR handler. The error handling object should have a simple interface (meaning the programmatic, not user, interface), so SFErrorMgr accepts only the same parameters as the Error method of objects (the error number, method, and line number) and returns a string indicating what choice the user (or object) made for resolving the error. The error object is at the end of the chain of responsibility, so it doesn't know much about the environment it was called from (it might be several objects removed, in a different data session, etc.). As a result, it can't really "handle" (that is, resolve) much. Its purpose is to display a message to the user, log the error for post-mortem purposes, and either decide what action to take (under certain foreseeable conditions) or more likely ask the user what action to take. Thus, the error object should really only be used to handle foreseeable errors you haven't yet foreseen (once they occur, you'll change the object, class, or routine that caused the error to handle that case) and unforeseeable errors (true bugs or unforeseeable environmental conditions).

The global error handler may take a global resolution itself (bring up the VFP Debugger or shutting down the application) or may allow the object originating the error to have the final resolution. To allow the latter, each step in the error handling chain returns a resolution code to the previous level. For simplicity, I decided to return a string indicating what resolution is chosen: "retry" to retry the command that caused the error, "continue" to return to the line of code following the one that caused the error, or "closeform" to close the form the control is sitting on. Each object then takes the appropriate action based on the return message. Because the Error method of a container object may have been called from a member object or by an error that occurred in one of its own methods, the container must decide whether to pass the return message on or process it itself. We'll see the code for this later.

This scheme has one problem: controls sitting on VFP base class Page, Column, or other containers with no Error method code essentially have no error trapping because they call an empty method! The solution is to travel up the containership hierarchy until we find a parent that has code in its Error method. If we can't find such a parent, then display a generic error message (this isn't likely, since I base all forms on the SFForm class, which does have Error method code).

On thing to keep in mind is that the complete error handling chain must be the most bug-free part of your application, since the only fallback if an error occurs in any code called while in the error state is the VFP Cancel/Ignore dialog. Fortunately, since you can put most of the error handling code into your framework, once you've got it working, it won't be much of a concern (although flaky environmental conditions can still cause the error handler itself to fail). Don't bother trying to create an Error method in the error handling object: it doesn't get called when an error occurs in the error handler itself.

The Error Method

Let's look at the strategy in more detail. The starting point when an error occurs is the Error method of the object the error occurred in, so let's start there.

The code for the Error method of most classes in the application base classes we've been building throughout this series, which are contained in SFCTRLS.VCX, is listed below (constants such as ccMSG_RETRY are defined in LIBRARY.H, which each class has as its include file). I say "most classes", because intermediate containers like PageFrames and OptionGroups and top-level containers like forms and toolbars have to work a little differently. This is one of the few times I wish VFP supported multiple inheritance; as it is, you need to use the VB method of subclassing to put the same code into the Error method of all classes (select the code in the method to subclass, press Ctrl-C, put the cursor in new method, and press Ctrl-V to tell it to use the desired parent class code <g>).

```
lparameters nError, cMethod, nLine
local oParent, ;
  lcReturn

* Travel up the containership hierarchy until we find a
* parent that has code in its Error method.

if type('Thisform') = 'O'
  oParent = iif(pemstatus(Thisform, 'FindErrorHandler', ;
    5), Thisform.FindErrorHandler(This), .NULL.)
else
  oParent = .NULL.
endif type('Thisform') = 'O'
do case

* We have a parent that can handle the error.

  case not isnull(oParent)
    lcReturn = oParent.Error(nError, This.Name + '.' + ;
      cMethod, nLine)

* We have an error handling object, so call its
* ErrorHandler() method.

  case type('oError') = 'O' and not isnull(oError)
    lcReturn = oError.ErrorHandler(nError, ;
      This.Name + '.' + cMethod, nLine)

* Display a generic dialog box.

  otherwise
    messagebox('Error #' + ltrim(str(nError)) + ;
      ' occurred in line ' + ltrim(str(nLine)) + ;
      ' of ' + cMethod + ' in object ' + This.Name, ;
      0, _VFP.Caption)
endcase
lcReturn = iif(type('lcReturn') <> 'C' or ;
  empty(lcReturn), ccMSG_CONTINUE, lcReturn)

* Handle the return value.

do case
  case lcReturn = ccMSG_RETRY
    retry
  case lcReturn = ccMSG_CANCEL
    cancel
  otherwise
    return
endcase
```

This code checks to see if the form the control is sitting on has a FindErrorHandler method (our base class for forms, SFForm, has this method), and if so, calls it to locate the first parent of the control with code in its Error method (we won't bother looking at this code; you can check it out yourself in the source code provided in the Subscriber Downloads). This prevents the problem of error handling stopping on base class Page, Column, or other containers because they have no code in the Error method. If a parent prepared

to handle the error is found, its Error method is called with the same parameters this Error method received, except the name of the object is added to cMethod so our error handling services can know which object the error originated in. If a parent isn't found but a global error handler exists (we'll look at the global handler later), its ErrorHandler method is called. If we have nothing to pass the error on to, we'll use MESSAGEBOX() to display an error message. The return value from the error handler is then used to decide how to resolve the error: retry, cancel, or return.

"Intermediate" container classes (SFContainer, SFControl, SFGrid, SFOptionGroup, and SFPageFrame) have nearly the same code as listed above, but must return the resolution message rather than handling it if the error is not their own. We'll check for this by looking for a period in the name of the method where the error occurred; since VFP passes just the method name if the error occurred in a method of the class but we coded the Error method of member objects to pass the name of the object and the method, this provides a quick way to distinguish errors caused by the object itself or a member. Here's an excerpt from the Error method of these classes showing the RETURN statement:

```
* Handle the return value.

do case
  case '.' $ cMethod
    return lcReturn
  case lcReturn = ccMSG_RETRY
    retry
```

Because they are the "top-level" containers (I don't use Formsets), the Error method for the SFForm and SFToolbar classes are different than other objects. This method uses the custom SetError method to populate some custom properties with information about the error, and the HandleError method to handle the error. It then processes the return value, either taking an action itself (such as closing the form) or returning it to the object that called this method. Notice it doesn't return a value if the object is the DataEnvironment but instead handles those errors itself. You may wish to change this behavior.

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
local lcReturn

* Use SetError() and HandleError() to gather error
* information and handle it.

with This
  .SetError(tnError, tcMethod, tnLine)
  lcReturn = .HandleError()
endwith

* Handle the return value, depending on whether the error
* was "ours" or came from a member.

do case
  case lcReturn = ccMSG_CLOSEFORM
    This.Release()
    return to master
  case '.' $ tcMethod and ;
    not 'DATAENVIRONMENT' $ upper(tcMethod)
    return lcReturn
  case lcReturn = ccMSG_RETRY
    retry
  case lcReturn = ccMSG_CANCEL
    cancel
  otherwise
    return
endcase
```

We won't look at the SetError method (you can examine it yourself in the source code) but here's the code for the HandleError method:

```
local lnError, ;
  lcMethod, ;
  lnLine, ;
  loError, ;
  lcMessage, ;
  lcReturn
with This
  lnError  = .aErrorInfo[.nLastError, cnAERR_NUMBER]
  lcMethod = .Name + '.' + ;
    .aErrorInfo[.nLastError, cnAERR_METHOD]
  lnLine   = .aErrorInfo[.nLastError, cnAERR_LINE]

* Get a reference to our error handling object if there
* is one. It could either be a member of the form or a
* global object.

  do case
    case type('.oError') = 'O' and not isnull(.oError)
      loError = .oError
    case type('oError') = 'O' and not isnull(oError)
      loError = oError
    otherwise
      loError = .NULL.
  endcase

* We don't have an error handling object, so display a
* dialog box.

  if isnull(loError)
    lcMessage = ccMSG_ERROR_NUM + ltrim(str(lnError)) + ;
      ccCR + ccMSG_MESSAGE + ;
      .aErrorInfo[.nLastError, cnAERR_MESSAGE] + ccCR + ;
      iif(empty(.aErrorInfo[.nLastError, cnAERR_SOURCE]), ;
      '', ccMSG_CODE + ;
      .aErrorInfo[.nLastError, cnAERR_SOURCE] + ccCR) + ;
      iif(lnLine = 0, '', ccMSG_LINE_NUM + ;
      ltrim(str(lnLine)) + ccCR) + ccMSG_METHOD + lcMethod
    lcReturn = iif(messagebox(lcMessage, ;
      MB_ICONEXCLAMATION + MB_OKCANCEL, ;
      _screen.Caption) = IDCANCEL, ccMSG_CANCEL, ;
      ccMSG_CONTINUE)

* We have an error handling object, so call its
* ErrorHandler() method.

  else
    lcReturn = loError.ErrorHandler(lnError, ;
      lcMethod, lnLine)
  endif isnull(loError)
endwith
lcReturn = iif(type('lcReturn') <>'C' or ;
  empty(lcReturn), ccMSG_CONTINUE, lcReturn)
return lcReturn
```

HandleError tries to pass the error to a global error handling object, referenced either through a global oError variable or through an oError property of the form. This scheme allows you to have a customized version of the global error handler associated with a specific form if desired. If no global error handler can be found, MESSAGEBOX() is used to display an error message. The return value from the error handler is then passed back to the Error method.

Global Error Handler

SFErrorMgr is a non-visual class based on SFCustom. It's contained in SFMGRS.VCX and uses the ERRORMGR.H include file for the definitions of several constants. It's instantiated into the global variable oError at application startup. We won't look at all the code for this class, only those methods which help illustrate the overall scheme of error handling services. Feel free to examine any other methods yourself.

The Init method accepts three parameters: the title to use for the dialog displayed when an error occurs (stored in the cTitle property), a flag indicating whether Init should save the current ON ERROR handler and change it to its ErrorHandler method, and the name of the object the class is being instantiated into (this is needed for the ON ERROR command, because we can't use THIS).

As is usually the case, the Destroy method cleans up things the class has changed; in this case, it resets VFP's error handler to the one that was in effect before the object was instantiated.

The ErrorHandler method is called both directly by objects as the last object in the chain of responsibility and indirectly since it's also the ON ERROR handler. Here's the code for this method:

```
lparameters tnError, ;
  tcMethod, ;
  tnLine
local lcCurrTalk, ;
  lcChoice, ;
  lcProgram
with This

* Ensure TALK is off.

  if set('TALK') = 'ON'
    set talk off
    lcCurrTalk = 'ON'
  else
    lcCurrTalk = 'OFF'
  endif set('TALK') = 'ON'

* Put the error into the aErrorInfo array and set the
* lErrorOccurred flag.

  .GetErrorInfo(tcMethod, tnLine)

* If errors aren't being suppressed, display the error
* and get the user's choice of action.

  lcChoice = ccMSG_CONTINUE
  if not .lSuppressErrors

* Log the error if necessary.

    if .lLogErrors
      .LogError()
    endif .lLogErrors

* Display the error and get the user's choice if desired.

    if .lDisplayErrors
      lcChoice = .DisplayError()
      do case

* Cancel or Quit in development environment: remove any
* WAIT window, revert all open cursors and issue a CLEAR
* EVENTS (in the case of Quit), and then return to the
* top-level program.

        case lcChoice = ccMSG_CANCEL or ;
          (lcChoice = ccMSG_QUIT and version(2) <> 0)
          wait clear
          if lcChoice = ccMSG_QUIT
            .lQuit = .T.
            .RevertAllTables()
            clear events
          endif lcChoice = ccMSG_QUIT
          lcProgram = .cReturnToOnCancel
          return to &lcProgram

* Display the debugger (development environment): activate
* the Trace and Debug windows.
```

```
        case lcChoice = ccMSG_DEBUG and version(2) <> 0
          activate window debug
          set step on

* Retry programmatic code: we must do the retry here,
* since nothing will receive the RETRY message (as is the
* case with an object).

        case lcChoice = ccMSG_RETRY
          lcMethod = upper(tcMethod)
          if at('.', lcMethod) = 0 or ;
            inlist(right(lcMethod, 4), '.FXP', '.PRG', ;
            '.MPR', '.MPX')
            if lcCurrTalk = 'ON'
              set talk on
            endif lcCurrTalk = 'ON'
            retry
          endif at('.', lcMethod) = 0 ...

* Quit: revert all open cursors, then quit.

        case lcChoice = ccMSG_QUIT
          .lQuit = .T.
          .RevertAllTables()
          on shutdown
          quit
      endcase
    endif .lDisplayErrors
  endif not .lSuppressErrors

* Restore TALK.

  if lcCurrTalk = 'ON'
    set talk on
  endif lcCurrTalk = 'ON'
endwith
return lcChoice
```

When an error occurs, three parameters are passed to ErrorHandler: the error number, the name of the routine in which the error occurred, and the line number where the error occurred. ErrorHandler uses the GetErrorInfo method to set the lErrorOccurred property to .T. and put information about the error into the aErrorInfo property. If the lSuppressErrors property is .T., the error isn't logged and no error message is displayed (this is used when you want an error to be trapped but not logged or displayed to the user). Otherwise, the LogError method is called to log the error and the DisplayError method is used to display a message about the error and get the user's choice about what action to take. The choices are:

- **Debug**: this option, which is only available if the lShowDebug property is .T. and we're running a development version of VFP, brings up the Trace and Debug windows. lShowDebug should be set to .T. only for developers (this can be looked up in a user table or the Windows Registry).
- **Continue**: returns to the command following the one that caused the error.
- **Retry**: retries the command. This gets a little tricky: if ErrorHandler was explicitly called (that is, from the Error method of an object), it can't just issue the RETRY command, since that would simply return control to the method which called it, rather than the method which caused the error. In that case, we'll just return the message "retry". However, if the error occurred in programmatic code (a PRG or MPR), ErrorHandler got called as the ON ERROR routine, so just returning this message won't work. In this case, ErrorHandler must directly issue the RETRY command itself.
- **Cancel**: a frequent question on CompuServe is: how do you prevent the rest of the code in the method or program that caused the error from executing? You can't use CANCEL, since that cancels the entire application. RETURN returns to the same method so that doesn't help either. The solution is to return to the main program, which puts you back on the READ EVENTS statement (which is normally where you're sitting when method code isn't executing). Since the "main" program may not be the first program in the application, we'll return to the program specified in the cReturnToOnCancel property rather than RETURN TO MASTER. When you instantiate SFErrorMgr, you can set this property to the

name of your "main" program (you can set it to "MASTER" if the first program in the application contains the READ EVENTS). Note: this technique only works if your READ EVENTS statement is in a PRG; if you put it in a method of an object, this won't work since you can't RETURN TO an object method. Thanks to fellow FoxTalk author Steve Sawyer for pointing this out.

- **Quit**: quits the application. We have different needs here depending on whether we're running a development copy of VFP or not. It'd be a pain if you had to restart VFP every time you got an error in development mode, so in that case, this option should CLEAR EVENTS and return to the main program so the application can shut down in an orderly manner and then return to the Command window. If this is a runtime version of VFP, we'll just clean up and quit. In both cases, we'll use a custom RevertTables method to perform a TABLEREVERT(.T.) on all cursors in all datasessions so we don't get additional errors (such as the infamous "uncommitted changes" error) on the way out.

Summary

This month's article presented the design and code for a general error handling scheme. This scheme uses a Chain of Responsibility design pattern so error handling can be escalated from the Error method of individual objects all the way to a global error handler providing common services. However, notice we didn't address what to do about specific errors, such as field or table rule violations or trigger failures. Letting these types of foreseeable errors be handled by the global error handler makes no sense, since it can't possibly be aware of the environment these errors occurred in or know what resolution to take.

Next month, we'll take a look at the typical errors a data entry form must be prepared to handle. We'll also examine a workaround for a fairly disastrous problem with the code generated by VFP's RI Builder that under some conditions allows restricted updates or deletions to partially succeed, leaving orphaned child records.

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit for Visual FoxPro and Stonefield Data Dictionary for FoxPro 2.x. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at user groups and regional conferences all over North America, and spoke at the 1997 Microsoft FoxPro Developers Conference. He is a Microsoft Most Valuable Professional (MVP). 75156.2326@compuserve.com or dhennig@stonefield.com.