Advanced Uses for Dynamic Form

Doug Hennig

Dynamic Form is an under-used project in VFPX. Its ability to create forms quickly and dynamically isn't something every developer needs but if you need it, Dynamic Form is indispensable. In this article, Doug discusses some advanced uses for Dynamic Form, including data driving the markup code.

In the May 2013 issue of FoxRockX, Rick Schummer discussed Dynamic Form, a VFPX project (http://vfpx.codeplex.com) which creates forms on the fly using markup code similar in purpose (although not in syntax) to XAML or HTML. Like Rick, I was initially uncertain whether I would make use of Dynamic Form myself; after all, it's not that hard to create a form for a specific task. However, I came across the need to ask the user for input about things that can change from user to user or application to application, and Dynamic Form soon became a frequently used tool in my toolbox.

This article discusses some advanced uses of Dynamic Form, especially data driving the creating of the markup code for the forms. Although you may certainly use the code included in the downloads for this article as is, I expect it will more likely serve as a source of inspiration for you to create your own dynamic forms.

Note: this article uses a version of DynamicForm.prg that I've customized a bit. I've passed on the changes I made to Matt Slay, author of Dynamic Form, but also included my copy of DynamicForm.prg with the downloads.

Custom fields

Some applications were built to be customized. For example, GoldMine and ACT!, two popular customer relationship management (CRM) systems, both allow users to add custom fields to their tables. The reason is that often organizations want to record additional details about customers beyond the standard things like company name and address, such as type of customer and what month their year end is. Both programs allow the user to specify the field name, data type, size, caption, and other meta data for the custom fields, and provide a data entry form in which the user can enter the values of those custom fields for a customer.

Let's look at doing something similar in a VFP application. We'll start with FieldDef.dbf, a table which contains the definitions of the custom fields. Its structure is shown in **Table 1**. As you can see, FieldDef contains both the definition of the custom fields (the actual custom field values are stored in CustomFields.dbf) and the specifications for the controls used to edit the values of the fields. **Figure 1** shows the records in the copy of FieldDef.dbf included in the downloads for this article.

Presumably, the application includes a data entry form that allows the user to define new custom fields, which adds records to FieldDef and new fields to CustomFields. We won't look at that part, just how they can edit the values of the custom fields, since this article is about creating dynamic forms.

Table 1. The structure of FieldDef.dbf.

Name	Type	Purpose
Order	I	The order of the field in the data entry form
Name	C(30)	The field name
DataType	C(1)	The field data type
Length	I	The field length
Decimals	I	The number of decimals
Caption	C(60)	The caption
Class	C(30)	The class for the control used to edit the field; empty to use a class based on DataType, such as a textbox for character fields
Library	C(30)	The VCX containing the class
Top	I	The top position of the control; 0 to auto-position
Left	I	The left position of the control; 0 to auto-position
Height	I	The height of the control; 0 for auto
Width	I	The width of the control; 0 for auto
Anchor	I	The Anchor setting for the control
Properties	M	The values of properties for the control using Dynamic Form syntax
Events	M	A comma-delimited list of events to handle for the control
Code	M	The event code to execute

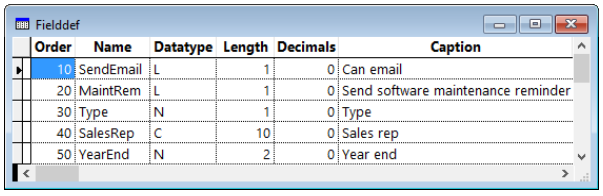

Figure 1. The FieldDef table contains the definitions of the custom fields and the controls used to edit them.

The CustomFields class in FieldControls.vcx is a simple form class used to edit the values of the custom fields for the specified customer. It has OK and Cancel buttons and an instance of the SFDynamicRender class, but no other controls. The Init method accepts as parameters the name of the current customer and the customer's ID. It finds the customer's record in CustomFields.dbf, creating one if necessary. It then uses SCATTER to create a data object for the record (the properties of this object are the ControlSource for the controls that Dynamic Form adds to the form), opens the FieldDef table, and asks the SFDynamicRender object on the form to render the controls for the custom fields. We'll look at SFDynamicRender later. Here's the code for Init:

```
lparameters tcCustomerName, ;
    tiCustNo
This.Caption = 'Custom Fields for ' + ;
    tcCustomerName

* Open the CUSTOMFIELDS table and find the
* record for the specified customer, creating
* it if necessary.

if used('CUSTOMFIELDS')
    select CUSTOMFIELDS
    set order to CUSTNO
else
    select 0
    use CUSTOMFIELDS order CUSTNO again shared
endif used('CUSTOMFIELDS')
seek tiCustNo
if not found()
    insert into CUSTOMFIELDS (CUSTNO) ;
        values (tiCustNo)
endif not found()

* Create a data object from the record for the
* rendering engine.

scatter memo name This.oDataObject
This.oRender.oDataObject = This.oDataObject

* Render controls for the custom fields.

select 0
use FIELDDEF order ORDER again shared
This.oRender.Render(This)
use
```

The Click methods of the OK and Cancel buttons close the form, but OK first uses GATHER to save the properties of the data object to the record in CustomFields.

SFDynamicRender, contained in SFDynamicForm.vcx, is a Custom subclass that uses Dynamic Form to render the controls defined in a table or cursor in the current workarea. It isn't specific for working with custom fields; we'll use this same class in the second section of this article for a different purpose. It can be used for any data-driven form as long as the structure of the table or cursor is similar to FieldDef.dbf (DataType, Length, and Decimals aren't used but the other fields are) and as long as the form it's used with has an oDataObject property containing an object with properties matching the contents of the Name field in the table or cursor. For example, if there's a record in the table with Name containing "SalesRep," the object referenced by oDataObject must have a SalesRep property. The Init method instantiates a

DynamicFormRenderEngine object and sets its properties the way we want. Note that it specifies my base classes defined in SFCtrls.vcx; feel free to change this to use your own classes instead.

```
* Create a Dynamic Form rendering engine
* object and use our classes as defaults.

This.oRenderEngine = ;
    newobject('DynamicFormRenderEngine', ;
    'DynamicForm.prg')
with This.oRenderEngine
    .cLabelClass            = 'SFLabel'
    .cLabelClassLib         = 'SFCtrls.vcx'
    .cTextboxClass          = 'SFTextBox'
    .cTextboxClassLib       = 'SFCtrls.vcx'
    .cEditboxClass          = 'SFEditBox'
    .cEditboxCLassLib       = 'SFCtrls.vcx'
    .cCommandButtonClass    = ;
        'SFCommandButton'
    .cCommandButtonClassLib = 'SFCtrls.vcx'
    .cOptionGroupClass      = 'SFOptionGroup'
    .cOptionGroupClassLib   = 'SFCtrls.vcx'
    .cCheckboxClass         = 'SFCheckBox'
    .cCheckboxClassLib      = 'SFCtrls.vcx'
    .cComboboxClass         = 'SFComboBox'
    .cComboboxClassLib      = 'SFCtrls.vcx'
    .cSpinnerClass          = 'SFSpinner'
    .cSpinnerClassLib       = 'SFCtrls.vcx'

* Specify that we don't want the container
* resized to fit the controls, we don't want
* Save and Cancel buttons (we'll use buttons
* on the form), we're using the form's data
* object, we don't want controls numbered, and
* we want objects 5 pixels apart vertically.

    .lResizeContainer       = .F.
    .cFooterMarkup          = ''
    .cDataObjectRef         = ;
        'Thisform.oDataObject'
    .lAddControlNumberToName = .F.
    .nVerticalSpacing       = 5
endwith

* Create a collection of behavior objects.

This.oBehaviors = newobject('SFCollection', ;
    'SFCtrls.vcx')
```

The Render method, called from the form's Init method, creates the markup code Dynamic Form uses to render the controls from the records in the current workarea and has the DynamicFormRenderEngine object render the controls to the specified container (in this example, the form):

```
#define ccCR chr(13)
lparameters toContainer
local lcRender, ;
    lcName, ;
    loBehavior, ;
    llSetupBehaviors

* Create Dynamic Form markup to create a
* control for each record in the current
* workarea.

This.oContainer = toContainer
lcRender        = ''
scan
  lcName   = trim(NAME)
  lcRender = lcRender + lcName + ;
    " .Caption = '" + trim(CAPTION) + ;
    "'" + ccCR
  if not empty(CLASS)
    lcRender = lcRender + ".Class = '" + ;
      trim(CLASS) + "'" + ccCR + ;
      iif(empty(LIBRARY), '', ;
      ".ClassLibrary = '" + ;
      trim(LIBRARY) + "'" + ccCR)
    endif not empty(CLASS)
  if TOP > 0
    lcRender = lcRender + " .Top = " + ;
      transform(TOP) + ccCR
  endif TOP > 0
  if LEFT > 0
    lcRender = lcRender + " .Left = " + ;
      transform(LEFT) + ccCR
  endif LEFT > 0
  if HEIGHT > 0
    lcRender = lcRender + " .Height = " + ;
      transform(HEIGHT) + ccCR
  endif HEIGHT > 0
  if WIDTH > 0
    lcRender = lcRender + " .Width = " + ;
      transform(WIDTH) + ccCR
  endif WIDTH > 0
  if ANCHOR > 0
    lcRender = lcRender + " .Anchor = " + ;
      transform(ANCHOR) + ccCR
  endif ANCHOR > 0
  if not empty(PROPERTIES)
    lcRender = lcRender + PROPERTIES + ccCR
  endif not empty(PROPERTIES)
  lcRender = lcRender + '|' + ccCR

* If we have behavior code, create a behavior
* object to handle it.

  if not empty(CODE)
    loBehavior = ;
      newobject('SFDynamicBehavior', ;
      'SFDynamicForm.vcx')
    loBehavior.cEvents = EVENTS
    loBehavior.cCode   = CODE
    This.oBehaviors.Add(loBehavior, lcName)
    llSetupBehaviors = .T.
  endif not empty(CODE)
endscan

* If we have any behavior code, call
* SetupBehaviorEvents for every control
* created.

if llSetupBehaviors
  bindevent(This.oRenderEngine, ;
    'AddControl', This, ;
    'SetupBehaviorEvents', 1)
endif llSetupBehaviors

* Render the markup into the container.

with This.oRenderEngine
  .oDataObject = This.oDataObject
  .cBodyMarkup = lcRender
  .Render(toContainer)
endwith

* Clean up before we exit.

if llSetupBehaviors
  unbindevent(This.oRenderEngine)
endif llSetupBehaviors
This.oContainer = .NULL.
```

The behaviors code requires some explanation. Since you can't add code to a class at runtime, if you want a control to do something special in some events, such as Refresh or InteractiveChange, you either need to use a

control that has code in those events or you need to use BINDEVENT to bind from the control to an object that handles the events. Assuming you don't want to create special subclasses just for this, I went with the latter. FieldDef.dbf (or whatever table you're using with this class) has an EVENTS field that lists the events we need to bind to and a CODE field that contains the code for the events. The Render method instantiates an SFDynamicBehavior object and fills in its cEvents and cCode properties with the values of those fields, then adds the object to a collection. Once all of the records have been processed, Render uses BINDEVENT to bind the AddControl method of the DynamicFormRenderEngine object, which creates a control, to the SetupBehaviorEvents method. This method finds the control the engine just created and for each of the events the behavior object is supposed to bind to, binds that event to the appropriate Handle method of the behavior object. Here's the code for SetupBehaviorEvents:

```
lparameters tcControlClass, ;
    tcClassLib, ;
    tcControlSource, ;
    tcDataType
local loControl, ;
    lcControl, ;
    loBehavior, ;
    laEvents[1], ;
    lnEvents, ;
    lnI, ;
    lcEvent
loControl = ;
  This.oContainer.Controls(;
  This.oContainer.ControlCount)
if lower(loControl.BaseClass) <> 'label'
  lcControl  = substr(loControl.Name, 4)
  loBehavior = ;
    This.oBehaviors.Item(lcControl)
  if not isnull(loBehavior)
    loBehavior.oObject = loControl
    lnEvents = alines(laEvents, ;
      loBehavior.cEvents, 5, ',')
    for lnI = 1 to lnEvents
      lcEvent = laEvents[lnI]
      if pemstatus(loControl, lcEvent, 5)
        bindevent(loControl, lcEvent, ;
          loBehavior, 'Handle' + lcEvent, 1)
      endif pemstatus(loControl, lcEvent, 5)
    next lnI
  endif not isnull(loBehavior)
endif lower(loControl.BaseClass) <> 'label'
```

Here's an example. One of the custom fields in the sample FieldDef table is the type of company: Prospect or Customer. If they're a Customer, we want to record the name of their salesperson and their year end. If they're a Prospect, we don't need that information. So, we want the control for Type to refresh the form when its value changes and the controls for salesperson and year end to disable themselves if Type is Prospect when they're refreshed. The Events field in FieldDef for the Type custom field contains "InteractiveChange" and the Code field contains:

```
lparameters toObject, ;
  tcEvent
toObject.Parent.Refresh()
```

Code in the Code field has to accept two parameters: a reference to the control the code applies to and the name of the event in upper case. If a control needs to handle more than one event, use a CASE statement like this:

```
lparameters toObject, ;
  tcEvent
do case
  case tcEvent = 'INTERACTIVECHANGE'
    toObject.Parent.Refresh()
  case tcEvent = 'REFRESH'
    * do something
endcase
```

So, through event binding, when the user changes the value of Type, the form is refreshed.

The Events field for both the SalesRep and YearEnd custom fields contains "Refresh" and the Code field contains:

```
lparameters toObject, ;
    tcEvent
toObject.Enabled = ;
  toObject.Parent.opgType.Value = 2
```

This causes the controls for those fields to only be enabled when Type is Customer (the second choice in the option group specified as the control for Type).

We won't look at the code in SFDynamicBehavior, as it's quite simple: the various Handle methods simply use EXECSCRIPT to execute the code specified in the cCode property, which comes from the Code field in the definition table.

Main.prg in the downloads for this article calls the CustomFields class like this:

```
loForm = newobject('CustomFields', ;
  'FieldControls.vcx', '', ;
  'ACME Supplies Ltd.', 1)
loForm.Show()
```

In a real-world application, the form would likely be displayed by clicking a button in the Customers form or choosing a "Edit Custom Fields" menu item and rather than hard-coding the customer name and ID, something like TRIM(CUSTOMERS.COMPANYNAME), CUSTOMERS.CUSTOMERID would be used.

The resulting form is shown in **Figure 2**. The values displayed in the form are stored in CustomFields.dbf. Adding a new custom field is as simple as adding a new record to FieldDef.dbf and a new field to CustomFields.dbf.

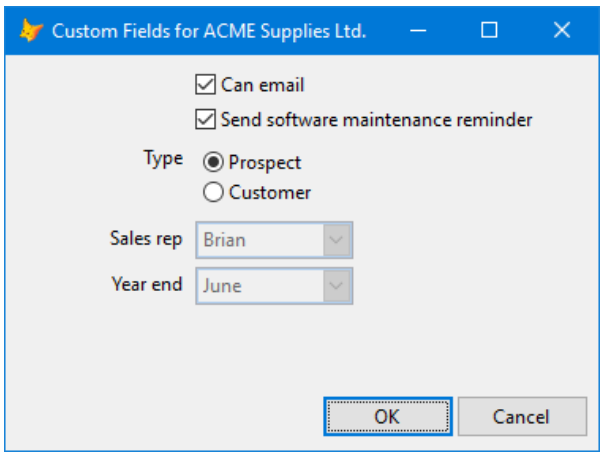

Figure 2. The custom fields form.

Dynamic application settings

Most applications need configuration settings of some type. Configurable settings allow a user to customize how the application works for them. Examples of settings are where data files are located (making that configurable allows the user to run the application on their local workstation but store the data on a server where it can be shared), what email settings to use (if the application sends emails), and what the company's year end is (if the application works with accounting data).

After creating what seemed like the hundredth dialog allowing a user to change configuration settings, I decided to create a generic one. I've always like the look of the Visual Studio Options dialog (**Figure 3**), so I modeled my form after that.

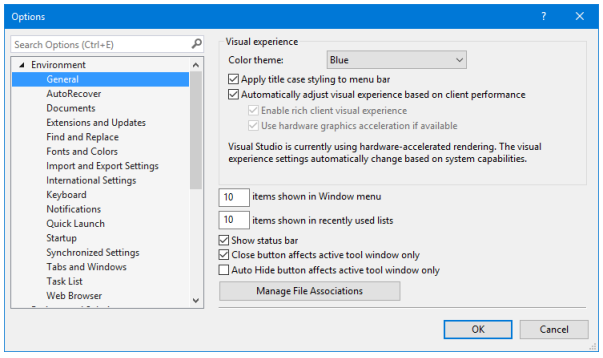

Figure 3. The Visual Studio Options dialog.

Let's start with where the settings are stored. They can be stored in lots of places, but a table, an INI file, and the Windows Registry are the most common. To be flexible, I use a set of persistence classes I wrote about a very long time ago (see "Persistence without Perspiration" at http://doughennig.com/papers/default.html) that have the same programmatic interface but different storage mechanisms: SFPersistentINIFile is for INI files, SFPersistentTable is for tables, and SFPersistentRegistry is for the Registry. Here's an example that uses Settings.ini to store settings:

```
loPersist = newobject('SFPersistentINIFile', ;
  'SFPersist.vcx', '', .T.)
loPersist.cFilePath = 'Settings.ini'
loPersist.cSection  = 'Options'
```

Here's one that stores settings in the Registry in the specified key under HKEY_CURRENT_USER:

```
loPersist = newobj('SFPersistentRegistry', ;
  'SFPersist.vcx', '', .T.)
loPersist.cKey = 'Software\Stonefield ' + ;
  'Software Inc.\MyApplication\Options'
```

How does the persistence object know what values to store? That's handled with an options class, SFOptions in SFOptions.vcx. SFOptions uses a data-driven approach; it uses a table whose name is specified in the cOptionsFile property (defaults to Options.dbf) to define which settings are used.

Options.dbf has almost the same structure as FieldDef.dbf that I discussed earlier because it's used by SFDynamicRender to display the settings to the user. It has a few additional fields:

- Rectype contains "P" for a Page record (the item in the TreeView defining a page of settings) or "O" for an Option record (a specific setting).
- Page contains the name of the page the option belongs to.
- Parent contains the name of the parent page the current page belongs under. This allows hierarchical pages. **Figure 4** shows an example of this: Email appears under the General item. To have Email appear at the top level instead, simply blank the Parent field in the sample Options.dbf accompanying this article.
- GetValue contains custom code to execute to convert the value from how it's stored to how it's displayed. For example, GetValue contains the following code for the Password record to decrypt the encrypted password:

```
lparameters toOptions, ;
  tcProperty
set library to VFPEncryption
lcValue = evaluate('toOptions.' + ;
  tcProperty)
if not empty(lcValue)
  lcValue = Decrypt(lcValue, ;
    'MyKey', 1024)
endif not empty(lcValue)
return lcValue
```

- SaveValue contains custom code to convert the value from how it's displayed to how it's stored. SaveValue for the Password record contains similar code to GetValue but calls Encrypt instead.

There are two main methods in SFOptions: LoadSettings and SaveSettings. We'll just look at the code for LoadSettings.

```
local lnSelect, ;
    lcAlias, ;
    loException, ;
    lcProperty, ;
    lcDataType, ;
    luValue

* Open the options table and create properties
* for each of the settings, plus add an item
* to the persistent object so it knows what to
* save and restore.

lnSelect = select()
lcAlias  = sys(2015)
with This
  select 0
  try
    use (.cOptionsFile) alias (lcAlias) ;
      again shared
    catch to loException
      .cErrorMessage = loException.Message
  endtry
  if used(lcAlias)
    scan for RECTYPE = 'O'
      lcProperty = trim(NAME)
      lcDataType = DATATYPE
      do case
        case lcDataType = 'C'
          luValue = ''
        case lcDataType = 'L'
          luValue = .F.
        case lcDataType = 'D'
          luValue = {/}
        case lcDataType = 'T'
          luValue = {/:}
        otherwise
          luValue = 0
      endcase
      try
        addproperty(This, lcProperty, luValue)
        .oPersist.AddItem(lcProperty, ;
          'This.oObject.' + lcProperty, ;
          lcDataType)
      catch to loException
        .cErrorMessage = loException.Message
      endtry
    endscan for RECTYPE = 'O'

* Now use the persistence object to load the
* values of the settings. Note that to avoid
* dangling references, we don't keep the
* object reference around any longer than we
* need to.

    .oPersist.oObject = This
    .oPersist.Restore()
    .oPersist.oObject = .NULL.

* Handle any settings that have code to
* execute to get the value.

    scan for RECTYPE = 'O' and ;
      not empty(GETVALUE)
      lcProperty = trim(NAME)
      store execscript(GETVALUE, This, ;
        lcProperty) to ('This.' + lcProperty)
    endscan for RECTYPE = 'O' ...
    use
  endif used(lcAlias)
endwith
select (lnSelect)
```

This code goes through the options table and adds properties to itself for each one. That allows you to access the settings using code like loSettings.YearStart and loSettings.DataDirectory. LoadSettings also configures the persistence object stored in oPersist so it knows which settings to read and write. Then it calls Restore to actually load the settings from the appropriate persistence location, and finally executes any custom code in the GetValue memo for each record. SaveSettings is similar but it calls Save rather than Restore and executes custom code in the SaveValue memo.

To load the settings for an application at startup, do something like this:

```
loPersist = newobject('SFPersistentINIFile', ;
  'SFPersist.vcx', '', .T.)
loPersist.cFilePath = 'Settings.ini'
loPersist.cSection  = 'Options'
loPersist.lSaveOnDestroy = .F.
loSettings = newobject('SFOptions', ;
  'SFOptions.vcx', '', loPersist)
loSettings.LoadSettings()
```

Here's an example that uses the YearStart setting, which contains which month a company's fiscal year starts in (for example, 11 for November), to determine which fiscal quarter a date is in:

```
lnQuarter = quarter(ldDate, ;
  loSettings.YearStart)
```

SFOptionsDialog (shown in Figure 4) is the class used to edit the settings. It consists of an SFTreeViewContainer object (see "The Mother of All TreeViews" part 1 and 2 at http://doughennig.com/papers/default.html for a discussion of that class) to display the list of pages of settings and a pageframe with Tabs set to .F. to contain the controls to edit the setting. Selecting an item in the TreeView sets the ActivePage property of the pageframe to the specified page.

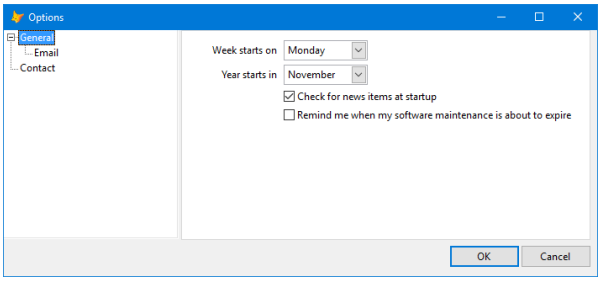

Figure 4. SFOptionsDialog allows the user to edit their settings.

We won't look at the code in this form other than the part related to Dynamic Form. The following is used to get a list of the options for the current page and render them to the appropriate page of the pageframe:

```
select * from OPTIONS ;
  where RECTYPE = 'O' and ;
  trim(PAGE) == lcPage ;
  into cursor RENDER order by ORDER
loPage = Thisform.pgfOptions.Pages[lnPage]
Thisform.oRender.Render(loPage)
```

As with the CustomFields class we saw earlier, SFOptionsDialog uses an SFDynamicRender object to do the hard work. To display the Options dialog to the user, use code like this:

```
loForm = newobject('SFOptionsDialog', ;
  'SFOptions.vcx', '', loSettings)
loForm.Show()
```

Summary

If you need the ability to create forms that appear differently for different users or different applications, Dynamic Form is a great tool that'll save you a lot of time. Being able to data-drive the markup code makes Dynamic Form even more flexible than I originally imagined. Feel free to use the code or the ideas in this article in your own applications.

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

Doug is co-author of "VFPX: Open Source Treasure for the VFP Developer," "Making Sense of Sedna and SP2," the "What's New in Visual FoxPro" series (the latest being "What's New in Nine"), "Visual FoxPro Best Practices For The Next Ten Years," and "The Hacker's Guide to Visual FoxPro 7.0." He was the technical editor of "The Hacker's Guide to Visual FoxPro 6.0" and "The Fundamentals." All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

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