Displaying and Editing Child Records

Doug Hennig

This month's article presents a grid class to display child records for a parent and a form class to edit the selected child record.

I'm not exactly sure why, but in general, editing in a grid has never appealed to me. Grids are great for displaying lists of records or for updating information in existing records (like a spreadsheet), but I've never liked adding records or editing records that require controls like comboboxes in a grid. For displaying child records of a parent, like outstanding invoices for a customer, a grid can't be beat. However, if, like me, you prefer to avoid using a grid to enter or edit those child records, what do you do?

The approach I've taken has been with us since before VFP: having Add, Edit, and Delete buttons that add a new child record or edit or delete the selected child record. Add and edit are similar: they bring up a dialog with either a blank child record or the selected child record, allow the user to edit the values as necessary, and then choose OK or Cancel to close the dialog and save or cancel the changes.

Because I've done a lot of applications lately that use this style of interface, I've created some classes that provide the features needed. The first is a grid class to display the child records and the second is a form class to maintain a single child record.

SFParameterizedViewGrid

This class, which is based on SFGrid (our grid base class in SFCTRLS.VCX) and located in SFCCTRLS.VCX, is used to simplify the display of child records for the current parent record. The key behind this is that you create a parameterized view for the child table, with the WHERE clause looking something like this: CHILD.PARENTID = ?vpParentID. vpParentID will contain the ID of the parent record that you want child records for. Then, you'll bind the view to an SFParameterizedViewGrid and tell it what the name of the view parameter is and how to get its value. It'll take care of requerying the view whenever the form is refreshed (such as when the user moves to a different parent record).

Let's look at how this class was created. First, the following properties of SFParameterizedViewGrid were set: RecordMark to .F. and ScrollBars to 2-Vertical (so the grid is more visually appealing), and DeleteMark to .F. and ReadOnly to .T. so the user can't make any changes directly in the grid. Next, three custom properties were added: aParameters, which will contain an array of view parameter information; cParameterName, a comma-delimited list of the parameters for the view; and cParameterSource, a comma-delimited list of what to use as the source of values for the view parameters. Although usually you'll only have a single view parameter, multiple view parameters are supported by separating them with commas.

The Refresh method requeries the view when we refresh the grid; if .T. is passed to this method (which you could do if you call it manually, rather than when it's called automatically when the form is refreshed), it won't requery the view. Here's the code:

```
lparameters tlNoRequery
if not tlNoRequery
  This.Requery()
endif not tlNoRequery
```

The custom Requery method does the actual work of requerying the view. It calls the custom GetViewParameters method to populate the aParameters array with information about the view parameters, then it processes this array. Each array row contains the name of the view parameter, the source of the value for that parameter, and the former value for the parameter. If .T. is passed (meaning we should requery the view whether we need to or not) or if the value of any view parameter has changed from its former value, variables are created for each view parameter, they're assigned the appropriate value, and the view is requeried.

```
lparameters tlForce
local llRequery, ;
  lnParms, ;
  lnI, ;
  lcParmSource, ;
  luValue, ;
  luFormer, ;
  lcParmName, ;
  lcAlias
with This

* Ensure the aParameters array is populated. Set a flag
* that we may not have to requery the view.

  .GetViewParameters()
  llRequery = .F.

* For each view parameter, get the value of the parameter
* source and see if it's changed.

  lnParms = alen(.aParameters, 1)
  for lnI = 1 to lnParms
    lcParmSource = .aParameters[lnI, 1]
    if not empty(lcParmSource)
      lcParmName = .aParameters[lnI, 2]
      luValue    = evaluate(lcParmSource)
      luFormer   = .aParameters[lnI, 3]
      if tlForce or isnull(luFormer) or ;
        not luValue == luFormer
        .aParameters[lnI, 3] = luValue
        llRequery = .T.
      endif tlForce ...
    endif not empty(lcParmSource)
  next lnI

* If we need to requery the view, create variables for
* each view parameter and assign it to the view parameter
* value, then requery the view.

  if llRequery
    for lnI = 1 to lnParms
      lcParmName = .aParameters[lnI, 2]
      luValue    = .aParameters[lnI, 3]
      local &lcParmName
      store luValue to (lcParmName)
    next lnI
    lcAlias = .RecordSource
    if cursorgetprop('SourceType', ;
      lcAlias) <> DB_SRCTABLE
      requery(lcAlias)
    endif cursorgetprop('SourceType', ...
  endif llRequery
endwith
```

GetViewParameters, which is called from Requery, populates the aParameters array by parsing the cParameterName and cParameterSource properties (these properties may contain information for a single view parameter or comma-delimited lists of information). Note that it only does this the first time it’s called, since the view parameters won’t change after that.

```
local lnParms, ;
  lnSStart, ;
  lnNStart, ;
  lnI, ;
  lnSPos, ;
  lnNPos, ;
  lcParmSource, ;
  lcParmName
with This

* We only need to do this if we haven't already.

  if empty(.aParameters[1])
```

```
* Ensure cParameterSource and cParameterName are valid
* and specify the same number of view parameters.

    assert not empty(.cParameterSource) and ;
      vartype(.cParameterSource) = 'C' and ;
      not empty(.cParameterName) and ;
      vartype(.cParameterName) = 'C' ;
      message 'SFParameterizedViewGrid.' + ;
      'GetViewParameters: cParameterSource and/or ' + ;
      'cParameterName not specified'
    assert occurs(',', .cParameterSource) = occurs(',', ;
      .cParameterName) ;
      message 'SFParameterizedViewGrid.' + ;
      'GetViewParameters: cParameterSource does not ' + ;
      'match cParameterName'

* Populate aParameters with the names of the view
* parameter sources and the view parameter names by
* parsing the comma-delimited cParameterSource and
* cParameterName properties.

    lnParms  = occurs(',', .cParameterSource) + 1
    lnSStart = 1
    lnNStart = 1
    for lnI = 1 to lnParms
      if lnI = lnParms
        lnSPos = len(.cParameterSource) + 1
        lnNPos = len(.cParameterName)   + 1
      else
        lnSPos = at(',', .cParameterSource, lnI)
        lnNPos = at(',', .cParameterName,   lnI)
      endif lnI = lnParms
      dimension .aParameters[lnI, 3]
      .aParameters[lnI, 1] = ;
        alltrim(substr(.cParameterSource, lnSStart, ;
        lnSPos - lnSStart))
      .aParameters[lnI, 2] = ;
        alltrim(substr(.cParameterName, lnNStart, ;
        lnNPos - lnNStart))
      .aParameters[lnI, 3] = .NULL.
      lnSStart = lnSPos + 1
      lnNStart = lnNPos + 1
    next lnI
  endif empty(.aParameters[1])
endwith
```

The AfterRowColChange method ensures that the form is refreshed when the record pointer is moved because, as we’ll see, the status of some controls may depend on what record is selected in the grid.

```
lparameters tnColIndex
dodefault(tnColIndex)
Thisform.RefreshForm()
```

Using an SFParameterizedViewGrid is easy: just drop it on a form, set RecordSource to the name of the child view, setup the columns to display the appropriate fields from the view, and set cParameterName and cParameterSource to the names of the view parameters and the source of their values. For example, if the primary key for the parent table is PARENT.ID and the view parameter is vpParentID, you’d enter “vpParentID” into cParameterName and “PARENT.ID” into cParameterSource. When the user moves to a certain parent record, your form will likely use Thisform.Refresh to refresh the form’s controls. This will cause SFParameterizedViewGrid.Refresh to fire, which will call the Requery method, which will requery the view with the current value of the parent key.

Let’s look at a concrete example. The sample files available on the Subscriber Downloads site include a database called TEST that has two tables: CLIENTS and its child CONTACTS (individuals who work for the client). The database also has a parameterized view, LVCONTACTS_FOR_CLIENT, that selects the ID and NAME from CONTACTS for a given client ID (the view parameter is called vpClientID). The

CLIENTS form, a data entry form based on SFMaintForm (in SFFORMS.VCX), has an instance of SFParameterizedViewGrid called grdContacts to display the name of each contact for the current client. I simply dropped an SFParameterizedViewGrid on the form, set RecordSource to LVCONTACTS_FOR_CLIENT (which I added to the DataEnvironment of the form and set NoDataOnLoad to .T. so the user isn't asked for the value of vpClientID when the view is first opened), defined a column to display the NAME field from this view, and set cParameterName to vpClientID and cParameterSource to CLIENTS.ID. When you run this form, you'll see that when you click on the Next and Previous buttons in the toolbar, the grid shows the appropriate contacts for the current client, without having to write any code in this form.

SFAddEditForm

OK, now we can display child records. How do we add and edit them? In their own form, of course, which will be very similar to other data entry forms. However, this form will only work with one child record at a time, the one we're adding or editing, so it'll have OK and Cancel buttons and we'll disable the record navigation features.

The form class for maintaining a single child record is SFAddEditForm, defined in SFFORMS.VCX and based on SFMaintForm, our data entry form class. This class has two buttons, cmdOK (based on SFOKButton in SFBUTTON.VCX) and cmdCancel (based on SFCancelButton, also in SFBUTTON.VCX). It has the following properties overridden: WindowType is 1-Modal so the user has to either save or cancel the record they're on before they can do anything else; MinButton is .F. since it doesn't make sense to minimize a modal form; lCanAdd and lCanDelete are .F. because we want the New and Delete buttons in the toolbar disabled; and cToolbarClass and cToolbarLibrary are empty because we don't want this form creating its own toolbar. This class has two custom properties: cPrimaryKey, which will contain the name of the primary key tag for the child table, and lSaved, which is set to .T. if the user clicks the OK button.

The Init method expects to be passed three parameters: tuKey, which is the key for child record we want to edit or 0 if we're adding a new record; tuInsertValues, which contains values to be inserted into a new record (often just the primary key for the parent, which is inserted into the foreign key field in the child, but we'll discuss this more later); and tcParent, a string that identifies the parent record to the user (such as the client name). After executing the default behavior, Init gets the name of primary key tag for the child table from the DBC (if it wasn't filled in already). It then either calls AddRecord to create a new record if tuKey is empty or finds that record in the table if not, and updates the Caption to show the "mode" (add or edit) and tcParent (so it'll look something like "New Contact for The Big Company"). Finally, it calls the CheckSave method, which determines when cmdOK can be enabled. CheckSave is an abstract method in the class; you'll put code into a specific subclass or form instance that would, for example, only enable the OK button when the minimum information has been entered.

```
lparameters tuKey, ;
  tuInsertValues, ;
  tcParent
local lcDatabase, ;
  lcPrimary
with This

* Do the default behavior.

  dodefault()

* Ensure things are set up correctly.

  assert not empty(tcParent) and ;
    vartype(tcParent) = 'C' ;
    message 'SFAddEditForm: parent name not passed'

* Determine the primary key tag for the table if necessary.

  if empty(.cPrimaryKey)
    .cPrimaryKey = dbgetprop(.cMainTable, 'Table', ;
      'PrimaryKey')
    assert not empty(.cPrimaryKey) ;
```

```
      message 'SFAddEditForm: no primary key for ' + ;
      .cMainTable
  endif empty(.cPrimaryKey)

* If the key wasn't specified, we're adding a new record.

  if empty(tuKey)
    .AddRecord(tuInsertValues)
    .Caption = 'New ' + .Caption + ' for ' + tcParent
  else

* If the key was specified, we're editing an existing
* record.

    = seek(tuKey, .cAlias, lcPrimary)
    .Caption = 'Edit ' + .Caption + ' for ' + tcParent
  endif empty(tuKey)

* Call the CheckSave method to enable or disable the OK
* button.

  .CheckSave()
endwith
```

The AddRecord method of SFMaintForm is overridden in this class with simple code that accepts a parameter and adds a blank record to the child table:

```
lparameters tuInsertValues
append blank in (This.cMainTable)
```

You'll likely override this code in a subclass or form instance because, at a minimum, you'll want to insert the primary key for the parent into the foreign key field in the child.

The Save method uses the default method of SFMaintForm, which does a TABLEUPDATE() in all open tables and handles any errors that occur, but passes .T. to ensure that the form isn't refreshed afterward. Then, if the save was successful, it sets the custom lSaved property to .T. and releases the form.

```
lparameters tlNoRefresh
local llReturn, ;
  lcKey
with This
  llReturn = dodefault(.T.)
  if llReturn
    .lSaved = .T.
    .Release()
  endif llReturn
endwith
```

The Cancel method, which is called if the user clicks on the form's Close box or Cancel button, simply uses the default behavior of SFMaintForm (which does a TABLEREVERT() in each open table) but passes .T. so this method doesn't refresh the form when it's done.

```
lparameters tlNoRefresh
dodefault(.T.)
```

There are a few other methods with code in this form. cmdOK.Click saves the record by calling Thisform.Save and cmdCancel.Click cancels any changes by calling Thisform.Cancel (it also uses DODEFAULT() to execute the default behavior of SFCancelButton, which is to release the form). FirstRecord, LastRecord, PreviousRecord, NextRecord, IsThisFirst, and IsThisLast, all record navigation methods, are overridden with a comment so they don't execute if called. Unload returns the value of This.lSaved so the calling form will know that, for example, it has to update itself because a new child record was added.

So, as you can see, we've simply customized the behavior of our data entry form class to specifically handle adding or editing a single child record. Creating a form for maintaining a child table is easy. Simply create the form from SFAddEditForm, add the child table (and any other tables desired) to the DataEnvironment and place controls bound to the appropriate fields onto the form. Set the form Caption to indicate what's being maintained, put code into AddRecord to insert the desired values into the child table, and put any code necessary into CheckSave to enable the OK button when appropriate (you'll want to call CheckSave from the AnyChange or Validation methods of the controls whose values determine when OK is enabled).

The CONTACTS form is an example of this. The Caption is "Contact" and Name is frmContacts. AddRecord inserts the passed value, which it assumes is the primary key for the CLIENT record that this is a child for, into the CLIENT field in the CONTACTS table.

```
lparameters tuInsertValues
insert into (This.cMainTable) ;
    (CLIENT) ;
  values ;
    (tuInsertValues)
```

CheckSave ensures the user enters the name of the contact:

```
with This
  .cmdOK.Enabled = not empty(.txtName.Value)
endwith
```

That's it! Simple, huh? However, what if you need to insert more than just a single value into the child table? In that case, I recommend creating a "parameter" object that contains properties for each of the values to be inserted, pass it to the SFAddEditForm, and then pull those values out of the object and insert them. For example, one SFAddEditForm-based form I created in a client's application needed to insert both a customer primary key and the ID for the current user into an activity child record for the customer. Here's a trick I used to handle this: I instantiated a object based on Custom, used the AddProperty method of this object to create iContactID and iUserID properties, filled them with the appropriate values, then passed this object to the activity form. Here's the code in the AddRecord method of this form:

```
lparameters tuInsertValues
local liContactID, ;
  liUserID
liContactID = tuInsertValues.iContactID
liUserID    = tuInsertValues.iUserID
insert into (This.cAlias) ;
    (CONTACTID, ;
    USERID) ;
  values ;
    (liContactID, ;
    liUserID)
return
```

Calling an SFAddEditForm

Now that we've created our child maintenance form, how do we call it from the parent? First, add buttons to the parent form to add, edit, and delete child records (in the CLIENTS sample form, these buttons are named cmdAdd, cmdEdit, and cmdDelete). The Refresh method of cmdAdd has the following code to ensure the user can't add a contact record if they don't have a valid client record:

```
local lcTable
lcTable = Thisform.cMainTable
This.Enabled = not eof(lcTable) and not bof(lcTable) ;
  and not deleted(lcTable)
```

The Refresh methods of cmdEdit and cmdDelete are similar to that of cmdAdd but ensure the user can't edit or delete a contact record unless they're sitting on one:

```
local lcTable
lcTable = Thisform.grdContacts.RecordSource
This.Enabled = not eof(lcTable) and not bof(lcTable) ;
```

```
  and not deleted(lcTable)
```

The Click methods of cmdAdd and cmdEdit are similar—they both call Thisform.EditContact—but cmdEdit passes .T. to indicate we're editing an existing record rather than adding a new one. Here's the code for EditContact:

```
lparameters tlEdit
local llSaved
do form CONTACTS with ;
  iif(tlEdit, lvContacts_For_Client.ID, 0), CLIENTS.ID, ;
  trim(CLIENTS.NAME) to llSaved
if llSaved
  This.grdContacts.Requery(.T.)
  This.grdContacts.Refresh(.T.)
endif llSaved
```

This code runs the CONTACTS form, passing it either the ID of the current child record (if we're editing) or 0 (if we're adding), the ID for the parent record, and the parent name. It expects a return value from the form, indicating whether the user saved or not. If so, the contacts grid is requeried (with a parameter passed to force it to do the requery; it normally wouldn't because the parent record hasn't changed) and refreshed (with a parameter passed to tell it not to requery, since that was just done).

Finally, the Click method of cmdDelete calls Thisform.DeleteContact, which has the following code to delete the current child record (after confirming the deletion with the user):

```
with This
  if messagebox('Delete this contact?', 36, .Caption) = 6
    delete in lvContacts_For_Client
    .Save()
  endif messagebox('Delete this contact?' ...
endwith
```

Thus, a little work is required in the parent form: adding three buttons and two custom methods for each child grid. I've created a form in a client's application that has 10 different child grids (in a pageframe, obviously), and it was only a couple of hour's work to get it all working.

Conclusion

If, like me, you prefer editing child records in their own form rather than in a grid, you might find the classes and techniques presented in this article useful.

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 and Stonefield Query. 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 the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP). He can be reached at dhennig@stonefield.com.