Creating Explorer Interfaces in Visual FoxPro

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Web site: http://www.stonefield.com
Web site: http://www.stonefieldquery.com
Blog: http://doughennig.blogspot.com

Overview

Explorer-style user interfaces are among the most popular for today's applications. However, working with and integrating the controls necessary to implement an explorer interface can be challenging. This session presents a set of classes and techniques that makes short work of developing explorer interfaces for your applications.

Introduction

An explorer-style interface is one which displays a list of items at the left and properties about the selected item at the right. An explorer interface has been popular for years and has been used in many applications, such as Windows Explorer and Outlook. It's not necessarily the ideal interface for all types of applications, but it works well when you have a hierarchical data structure, such as files within folders, orders for customers, or contacts for organizations.

There are several common components in a typical explorer interface:

- A list of items. Although you could use a VFP Listbox or Grid, a TreeView control is a better choice because it natively provides a hierarchical view of your data.
- A set of controls showing properties about the selected item. One way I've handled this is the past is to create one container class with the desired controls for each type of item to display, overlay instances of each container on the right side of a form, and use code in the Refresh method of the containers that hides or shows the container based on the type of item currently selected. While it looks fine at run time, it's pretty messy at design time. I now prefer to use a pageframe with Tabs set to .F. so it appear to be just a container, and an instance of the container classes on each page. It's then a simple matter to set the pageframe's ActivePage property to the appropriate value to display the desired container when an item is selected in the TreeView.
- A splitter control between the item list and the properties pane. This allows the user to change the relative sizes of the list and properties by dragging the splitter left or right.
- A status bar control. This typically displays information about the selected item, such as its name, and form status information, such as "Processing..." when it performs some action.

Let's take a look at how to create an explorer interface.

The Microsoft TreeView control

A TreeView control is the starting point for most explorer interfaces. Several years ago, I wrote a white paper, available at http://www.stonefield.com/pub/axsamp.zip, on using ActiveX controls in VFP, and part of that paper covers the Microsoft TreeView and ImageList controls (the ImageList provides images the TreeView uses for its nodes) that come with VFP in detail. Feel free to read that white paper for background on these controls and information on their properties, events, and methods (PEMs).

The TreeView is a fairly ornery control. Here are just some of the gotchas about working with a TreeView:

- Loading nodes is fairly slow, so if there are a lot of them, it's better to load only the top-level or root nodes, and then load the child nodes on demand the first time a root node is expanded. This requires a bit of work: you have to add "dummy" child nodes or else the "+" won't appear in front of parent nodes, and when the node is expanded, you have to see if the first child is the "dummy" node, and if so, remove it and add the child nodes.

- Removing the "dummy" node and adding child nodes can cause a lot of screen update. For VFP forms, we set LockScreen to .T. to prevent the screen from being updated until we're ready, but the TreeView doesn't respect the setting of LockScreen and doesn't have its own equivalent property. Instead, you have to call a Windows API function to lock and unlock the TreeView.
- The coordinate system for TreeViews is twips ($1/1440^{th}$ of an inch) rather than pixels, so you have to convert pixels to twips before calling TreeView methods that expect coordinate parameters. Again, this involves using Windows API functions.
- Right-clicking or dragging from a node doesn't make it the selected node, so code that displays a menu for the current node or supports node dragging must manually figure out which node the user clicked on.
- You may want the user to be able to double-click a node to take some action, such as bringing up a dialog to edit the underlying record for the node. Unfortunately, double-clicking a node automatically causes it to expand or collapse if it has children.
- There is no RightClick event for the TreeView control, so if you want shortcut menu, you have to look for a right-click in MouseDown.
- Sometimes clicking a node in the TreeView doesn't fire its NodeClick event, so the node you think is selected isn't necessarily the node that is selected.
- If you support drag and drop, the TreeView doesn't automatically scroll up or down when you drag into the top or bottom edges, nor does it automatically highlight the node the mouse is over to give visual feedback to the user. You have to handle these things yourself.

Having created forms that had to deal with all of these quirks and coordinate TreeView node clicks with the rest of the form, I decided it was time once and for all to create a reusable component that would encapsulate all the behavior I wanted in one place. So, step one in creating a reusable explorer interface was to create a reusable TreeView class.

SFTreeViewContainer, defined in SFTreeView.VCX, is a subclass of SFContainer (my base class Container located in SFCtrls.VCX) that encapsulates all the behavior I'd ever want in a TreeView, including:

- Automatically linking an ImageList control with the TreeView.
- Providing control over whether all nodes are loaded, only parent nodes with "dummy" nodes under them are loaded (and automatically removing the "dummy" node and loading the child nodes when the parent is expanded), or to what hierarchy level nodes are initially loaded.
- Locking the TreeView during load operations to minimize flicker.
- Maintaining user-defined information about the selected node, including the "type" of node and the ID of the "thing" the node represents.
- Converting the pixel values passed to VFP methods into twips values the TreeView expects.

- Automatically selecting a node whenever it's clicked, including by right-clicking and dragging.
- Allowing double-clicking a node to take some action without expanding the node.
- Providing a shortcut menu when you right-click a node.
- Automatically scrolling the TreeView when you drag into its top or bottom edges and automatically highlighting the node the mouse is over.
- Support for dynamically allowing or disallowing the user to edit the text for a node.
- Support for adding or deleting nodes by pressing Insert or Delete.
- Providing "go back" functionality similar to the Back button in a browser.
- Support for saving and restoring the state of the TreeView: which node was selected and which nodes were expanded.

I won't describe how SFTreeViewContainer works here; see Appendix A for details. I also won't describe how to use it, because it's intended to be very flexible and thus requires a fair bit of coding in a subclass to completely implement. Instead, we'll look at a data-driven subclass of SFTreeViewContainer called SFTreeViewCursor, also contained in SFTreeView.SCX.

SFTreeViewCursor

SFTreeViewCursor allows you to load nodes in the TreeView by simply filling a cursor with information about those nodes. It has a custom cCursorAlias property that Init sets to SYS(2015) so it has a unique name. The LoadTree method, which loads the TreeView and is called from Init if the lLoadTreeviewAtStartup property is .T. (it is by default), closes the cursor if it's already open, then calls CreateTreeViewCursor to create the cursor and FillTreeViewCursor to fill the cursor with records. Finally, it uses DODEFAULT() to perform the actual job of loading the TreeView as coded in the parent class.

CreateTreeViewCursor creates the cursor named in cCursorAlias from the structure defined in cCursorStructure. You can add additional fields to the cursor by adding their names and data types to that property, but don't remove or rename the existing fields because many methods in SFTreeViewCursor depend on them (you can change the size of the columns if necessary). cCursorStructure defines a cursor with the fields shown in **Table 1**. It creates indexes on the ID, PARENTID, PARENTTYPE, NODEKEY, and TYPE fields, as well as TEXT if the lSortRootNodes property is .T.

Table 1. The structure of the cursor used to load nodes in SFTreeViewCursor.

Field	Type	Purpose
ID	C(60)	The ID of the record the node is for. For example, for customer records, this would be the customer ID. Since this is a character value, use TRANSFORM() if the ID for the source record is a different data type.
TYPE	C(60)	The type of record. This can be anything you wish, such as “Customer” or “Order.”
PARENTID	C(60)	The ID of the parent record for this record. For example, for order records, ID would be the order ID and PARENTID would be the customer ID for the customer who placed the order. Leave this blank if this should be a top-level node. As with ID, use TRANSFORM() if necessary.
PARENTTYPE	C(60)	The type of the parent record. PARENTID isn’t necessary enough to distinguish records because you may have several types of records with the same IDs (tables that have auto-incrementing fields, for example).
TEXT	C(60)	The text to display for the node in the TreeView.
IMAGE	C(20)	The key of the image in the ImageList to use for this node.
SELIMAGE	C(20)	The key of the image in the ImageList to use for this node when it’s selected. Leave this blank to use the same image as IMAGE.
EXPIMAGE	C(20)	The key of the image in the ImageList to use for this node when it’s expanded. Leave this blank to use the same image as IMAGE.
SORTED	L	.T. if the child nodes under this node should be sorted or .F. to display them in the order they appear in the cursor.
NODEKEY	C(60)	The Key for the node in the TreeView. You can leave this value blank since SFTreeViewCursor concatenates the values in TYPE and ID to create a key value if one isn’t specified. However, you can also assign your own values if you wish.

FillTreeViewCursor is an abstract method in this class. Typically, you use code in an instance or subclass that fills the cursor specified in cCursorAlias with parent and child records, such as the following which loads customers as parent nodes and orders for those customers as child nodes:

```
select Customers
scan
  insert into (This.cCursorAlias)
       (ID, ;
       TYPE, ;
       TEXT, ;
       IMAGE, ;
       SORTED) ;
    values ;
       (transform(Customers.CustomerID), ;
       'Customer', ;
       Customers.CompanyName, ;
       'Customer', ;
       .T.)
 endscan
 select Orders
 scan
  insert into (This.cCursorAlias) ;
       (ID, ;
       TYPE, ;
       PARENTID, ;
       PARENTTYPE, ;
       TEXT, ;
       IMAGE) ;
    values ;
       (transform(Orders.OrderID), ;
       'Order', ;
       transform(Orders.CustomerID), ;
       'Customer', ;
       'Order ' + transform(Orders.OrderID), ;
```

```
      'Order')
 endscan
```

As described in Appendix A, LoadTree loads the TreeView by calling GetRootNodes to fill a collection with information about the root nodes. Other methods that need to load the children for a particular node call GetChildNodes to fill a collection with information about the child nodes for the parent node (which could be a root node or a child that itself has children). These methods are abstract in SFTreeViewContainer, but SFTreeViewCursor has the following code in GetRootNodes to fill the collection with all parent records; that is, records with PARENTID blank.

```
lparameters toCollection
local lnSelect, ;
  llHasChildren, ;
  loNodeItem
with This
  lnSelect = select()
  select (.cCursorAlias)
  if .lSortRootNodes
    set order to TEXT
  else
    set order to
  endif .lSortRootNodes
  scan for empty(PARENTID)
    .AddNodeToCollection(toCollection)
  endscan for empty(PARENTID)
  select (lnSelect)
endwith
```

If you want root nodes to be sorted, set lSortRootNodes to .T. Otherwise, they appear in the order they were added to the cursor.

GetChildNodes uses similar code, but doesn't care about the sort order because the TreeView will take care of sorting nodes automatically if the Sorted property for a node, set by the SORTED column in the cursor, is .T. In this case, however, GetChildNodes looks for records that have PARENTID and PARENTTYPE set to the ID and type values for the parent node.

```
lparameters tcType, ;
  tcID, ;
  toCollection
local lnSelect
with This
  lnSelect = select()
  select (.cCursorAlias)
  set order to
  scan for PARENTID = tcID and PARENTTYPE = tcType
    .AddNodeToCollection(toCollection)
  endscan for PARENTID = tcID ...
  select (lnSelect)
endwith
```

Both GetRootNodes and GetChildNodes call AddNodeToCollection, which is a new method added to SFTreeViewCursor. It creates an object with information about a node in a TreeView, fills it with information from the current record in the cursor, and adds it to the collection. The method in SFTreeViewContainer that actually adds a node to the TreeView uses the objects in the collection to set properties of the node.

```
lparameters toCollection
local lnRecno, ;
  lcID, ;
  lcType, ;
```

```
  llHasChildren, ;
  lcKey, ;
  loNodeItem
with This

* See if the current record has any children.

  lnRecno = recno()
  lcID    = ID
  lcType  = TYPE
  locate for PARENTID = lcID and PARENTTYPE = lcType
  llHasChildren = found()
  go lnRecno

* If we don't have a key, create one and put it into NODEKEY.

  if empty(NODEKEY)
    lcKey = .GetNodeKey(TYPE, ID)
    replace NODEKEY with lcKey
  else
    lcKey = trim(NODEKEY)
  endif empty(NODEKEY)

* Create a node item object and fill its properties from fields in the cursor
* for the current record, then add it to the collection

  loNodeItem = .CreateNodeObject()
  with loNodeItem
    .Key           = lcKey
    .Text          = alltrim(TEXT)
    .Image         = alltrim(IMAGE)
    .SelectedImage = iif(empty(SELIMAGE), .Image, alltrim(SELIMAGE))
    .ExpandedImage = iif(empty(EXPIMAGE), .Image, alltrim(EXPIMAGE))
    .Sorted        = SORTED
    .HasChildren   = llHasChildren
  endwith
  toCollection.Add(loNodeItem)
endwith
```

Now that we've seen how the TreeView is loaded from records in the cursor, let's see what happens when the user clicks a node. As described in Appendix A, the NodeClick event of the TreeView, which fires when the user clicks a node, calls the SelectNode method of the container, although you can also call SelectNode to programmatically select a node. SelectNode does the work of selecting the specified node, as well as some other tasks discussed in Appendix A, and then calls NodeClicked. NodeClicked is abstract in SFTreeViewContainer but has the following code in SFTreeViewCursor to find the record for the selected node in the TreeView cursor:

```
local lnSelect
with This
  lnSelect = select()
  select (This.cCursorAlias)
  locate for ID = .cCurrentNodeID and TYPE = .cCurrentNodeType
  select (lnSelect)
  .DisplayRecord()
endwith
```

SelectNode set the values of three properties, cCurrentNodeID, cCurrentNodeType, and cCurrentNodeKey, to the appropriate values for the selected node, so NodeClicked uses those values to find the matching record. It then calls DisplayRecord, which is abstract in this class, but is used to subclasses to update the rest of the form when a node is clicked.

Using SFTreeViewCursor

Let's see how to use SFTreeViewCursor. In an instance or subclass, we really only have to worry about the following methods and properties for the simplest behavior of displaying a TreeView and reacting to node clicks:

- FillTreeViewCursor: add code to fill the cursor with the desired records for parent and child nodes.
- LoadImages: add code to load images into the ImageList control.
- DisplayRecord: add code to do something when a node is selected.
- lSortRootNodes: set this property to .F. if you don't want root nodes sorted but to appear in the order you added them to the cursor.
- cRegistryKey or cRegistryKeySuffix: as discussed in Appendix A, set cRegistryKey to the Registry key you want the TreeView selected and expanded nodes saved to, or if the form has a cRegistryKey property and you want to use a subkey of that, set cRegistryKeySuffix instead. Leave these blank to not save and restore the TreeView state.
- lTrackNodeClicks: set this to .F. if you don't want to support "go back" behavior.

Create a form. Put the following code into Load to open the sample Northwind database:

```
close tables all
set path to SourcePath

open database home() + 'Samples\Northwind\Northwind'
```

where *SourcePath* is the path to the Source folder of the code provided with this document.

Drop an instance of SFTreeViewCursor on it. Name the instance oTreeView and set cRegistryKey to "Software\Stonefield Software\SFTreeViewCursor Test." Add the following code to FillTreeViewCursor:

```
* Parent records are customers.

local lcAlias
lcAlias = This.cCursorAlias
select 0
use Customers
scan
  insert into (lcAlias) (ID, TYPE, TEXT, IMAGE, SORTED) ;
    values (Customers.CustomerID, 'Customer', Customers.CompanyName, ;
      'Customer', .T.)
endscan

* Child records are orders.

select 0
use Orders
scan
  insert into (lcAlias) (ID, TYPE, PARENTID, PARENTTYPE, TEXT, ;
      IMAGE) ;
    values (transform(Orders.OrderID), 'Order', Orders.CustomerID, ;
      'Customer', 'Order ' + transform(Orders.OrderID), 'Order')
endscan
```

Put the following code into LoadImages; the images specified here are included with the samples for this document:

```
with This.oImageList
  .ListImages.Add(1, 'Customer', loadpicture('Images\Customer.ico'))
  .ListImages.Add(2, 'Order',    loadpicture('Images\Order.ico'))
endwith
```

Put the following somewhat annoying code <g> into DisplayRecord. Note that we use PADR() on cCurrentNodeID when finding the customer for the selected node because the ID field in the TreeView cursor isn't the same size as CustomerID in Customers, so this ensures we use the same length. Also note that we use VAL() on cCurrentNodeID when finding the order because the ID field in the cursor is character while OrderID in Orders is numeric.

```
with This
  if .cCurrentNodeType = 'Customer'
    = seek(padr(.cCurrentNodeID, len(Customers.CustomerID)), 'Customers', ;
      'CustomerID')
    messagebox('Customer:' + chr(13) + chr(13) + ;
      'Company: ' + trim(Customers.CompanyName) + chr(13) + ;
      'Contact: ' + trim(Customers.ContactName) + chr(13) + ;
      'Country: ' + trim(Customers.Country))
  else
    = seek(val(.cCurrentNodeID), 'Orders', 'OrderID')
    messagebox('Order:' + chr(13) + chr(13) + ;
      'Order ID: ' + transform(Orders.OrderID) + chr(13) + ;
      'Order Date: ' + transform(Orders.OrderDate))
  endif .cCurrentNodeType = 'Customer'
endwith
```

Finally, drop a command button on the form with the following code in Click:

```
Thisform.oTreeView.GoBack()
```

The resulting form is shown in **Figure 1**. When you run it a second time, it restores the expanded nodes and selected node automatically.

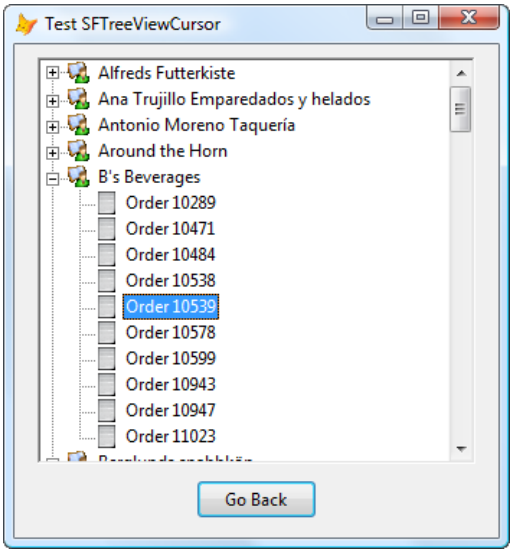


Figure 1. This sample form shows how easy it is to use SFTreeViewCursor.

SFExplorerForm

Now that we have a control that implements most of the behavior we'd ever need in a TreeView, what about a form that's hosts the TreeView and controls showing properties about the selected

node? That's what SFExplorerForm is for. SFExplorerForm, defined in SFExplorer.VCX, is a subclass of SFForm, my base class form class defined in SFCtrls.VCX. **Figure 2** shows SFExplorerForm in the Class Designer. As you can see, it doesn't actually contain a TreeView control, but it does have some other controls:

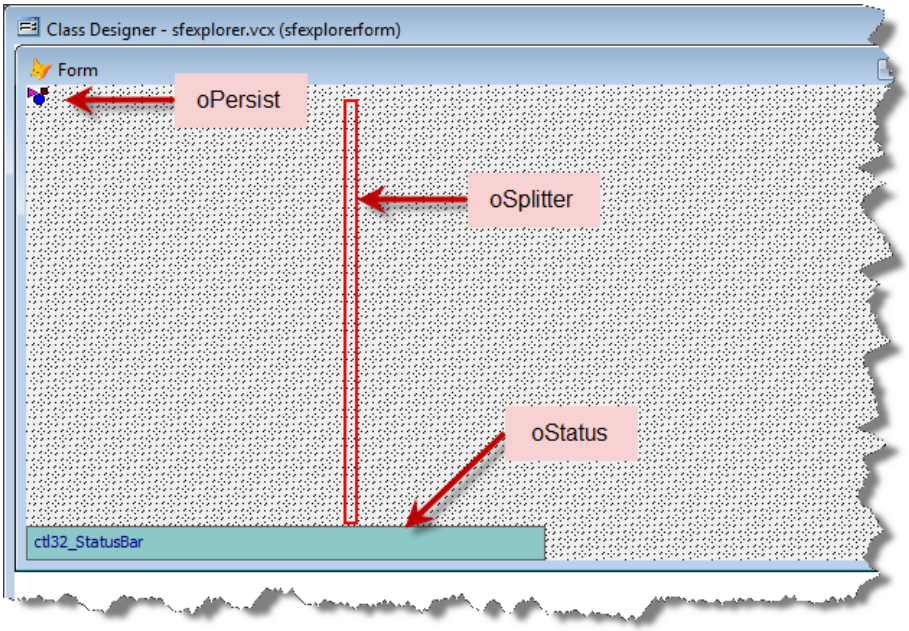


Figure 2. SFExplorerForm is the starting point for explorer forms.

- A splitter control named oSplitter to adjust the relative sizes of an object at the left and one at the right. The splitter only has a red border in the Class Designer so you can see it at design time; it has no border at runtime. Drag the splitter left or right to resize the controls to the left and right of the splitter at runtime
- A ctl32_StatusBar control named oStatus to display information about the state of the form, such as whether it's ready or currently loading the TreeView. ctl32_StatusBar was written by Carlos Alloatti and is one of the projects on the VFPX site (www.codeplex.com/vfpx).
- An instance of SFPersistentForm, named oPersist, to persist the form size and position. SFPersistentForm is defined in SFPersist.VCX and was discussed in my January 2000 FoxTalk article "Persistence without Perspiration."

SFExplorerForm has the following custom properties:

- cCurrentNodeID, cCurrentNodeKey, and cCurrentNodeType: these properties contain the values of the same named properties of an SFTreeViewContainer object so you can use Thisform.*property* rather than Thisform.oTreeView.*property* to access the values.
- cDefaultStateMessage: the message displayed in the right-most panel of the status bar (the "state" panel) when the form isn't doing anything; it defaults to "Ready."

- cGoBackIcon: the icon to use in the "go back" panel of the status bar; the default is "Back.ico," one of the images provided with the samples for this document.
- cRegistryKey: the Registry key to use to persist form and TreeView settings.
- cStateIconBusy: the icon to use in the state panel of the status bar when the form is busy, such as when the TreeView is being loaded; defaults to "Red2.ico."
- cStateIconReady: the icon to use in the state panel of the status bar when the form is not busy; defaults to "Green2.ico."
- cStatePanelName: the name of the state panel; defaults to "StatePanel."
- cToolbarClass, cToolbarLibrary, and oToolbar: put the class and library for a toolbar you want added to the form into cToolbarClass and cToolbarLibrary. The form instantiates the specified class into oToolbar. Leave them blank to not use a toolbar.
- lStatePanelAutoSize and nStatePanelWidth: set lStatePanelAutoSize to .T. if the state panel should automatically adjust its size based on its content; if .F. (the default), the panel has a fixed width of the value in nStatePanelWidth (defaults to 200).
- lUseFormFont: the form uses Tahoma in Windows XP and Segoe UI in Vista. If lUseFormFont is .T. (the default), the form uses SetAll to set the font of all controls to the same font as the form.
- nSplitterLeft: the current splitter position. This property is used for internal purposes.

SFExplorerForm has a few methods associated with the status bar. SetupStatusBar, called from Init, initializes the various panels in the status bar and uses BINDEVENT() to bind clicks in the status bar to the form's StatusBarClick method, which is empty in this class. UpdateMessagePanel accepts a message to display in the left-most panel of the status bar and returns the former message. You can use the return value in code that temporarily changes the message before changing it back. UpdateStatePanel is similar, but if you don't pass anything to that method, it uses the value of cDefaultStateMessage as the message. Also, if the message is anything other than the default message, it sets the icon for the state panel to the value of cStateIconBusy; otherwise, it uses the icon specified in cStateIconReady. UpdateProgressBar accepts a numeric value, typically between 0 and 100, and sets the value of the progress bar embedded in the status bar to that value. It also makes the progress bar visible if it isn't already. HideProgressBar does exactly what its name suggests.

There are only a few other methods with code in this form. RestoreFormSize, called from Init, uses the oPersist object to restore the form size and position and the value of nSplitterLeft, which Show uses to restore the position of the splitter control. SaveFormSize, called from ReleaseMembers, which is itself called from Destroy, saves the form size and position and the splitter position. Show does some final setup chores, including moving the splitter control to its restored position; this automatically adjusts the sizes of the objects to the left and right of it (we'll see how to register those objects with the splitter later). Show also instantiates the toolbar class specified in cToolbarClass and cToolbarLibrary into oToolbar if a class is specified, uses SetAll to set the font of all controls if lUseFormFont is .T., and calls UpdateStatePanel to display the default "ready" message. One final method, DisplayProperties, is empty in this class.

SFExplorerForm isn't intended to be used by itself; instead, you'll use one of the subclasses included in SFExplorer.VCX. You can also create your own subclass; see the notes in the About method for a list of things to do. Let's look at the subclasses of SFExplorerForm.

SFExplorerFormTreeView

SFExplorerFormTreeView (see **Figure 3**) is a subclass of SFExplorerForm with two controls added to it: an SFTreeViewExplorer (described in the next paragraph), named oTreeViewContainer, at the left and a SFPageFrame (my pageframe subclass defined in SFCtrls.VCX) named pgfProperties at the right. These controls are registered with the splitter by setting its cObject1Name property to "oTreeViewContainer" and cObject2Name to "pgfProperties." If desired, you can change the values of the nObject1MinSize and nObject2MinSize properties, which control how small the two objects can get when the splitter is moved, to suitable values in an instance or subclass.

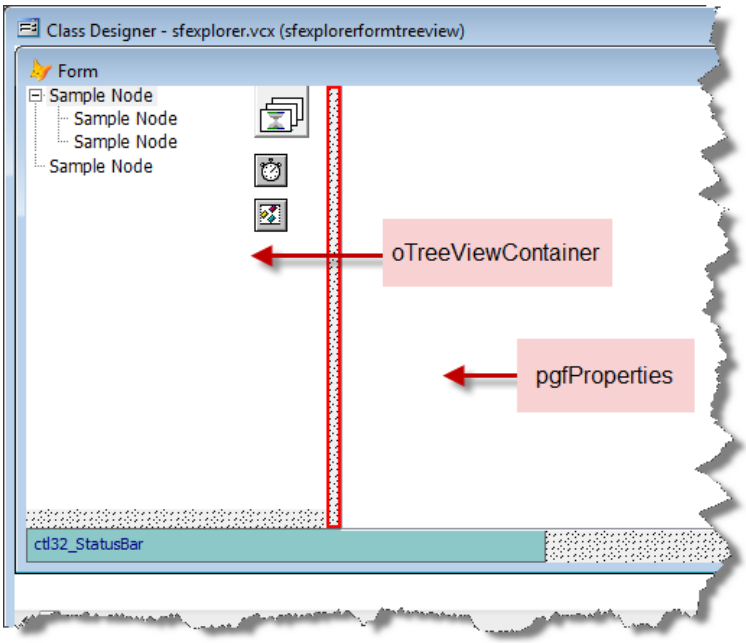


Figure 3. SFExplorerFormTreeView can be used to create explorer forms.

The purpose of this form is to display items in the TreeView control and properties about the selected item in the pageframe at the right.

SFTreeViewExplorer is a subclass of SFTreeViewCursor intended to be used in subclasses of SFExplorerForm. It has the following changes from SFTreeViewCursor:

- cCursorStructure has an additional PAGE column that holds the page in pgfProperties to display when a node is selected.
- LoadTree calls UpdateStatePanel before and after loading the TreeView so a "Loading" message displays while the process runs. LoadExpandedNode does something similar in case there are a lot of children under the node being expanded.

- DisplayRecord, called when the user clicks a node, gets the page number to display in pgfProperties from the PAGE column in the TreeView cursor's current record, sets the form's cCurrentNodeID, cCurrentNodeKey, and cCurrentNodeType properties to the same values as the control's, and passes the page number to the form's DisplayProperties method (described later) so the correct page is displayed and refreshed.

Other than its size and position, the only property of oTreeViewContainer changed in this form is lLoadTreeviewAtStartup, which is set to .F. because we may want to do some setup tasks before the TreeView is loaded. Also, note that the Left and Top values of oTreeViewContainer are -2; this prevents the TreeView from having a 3-D appearance, which is considered to be old-style now.

The Tabs property of pgfProperties is set to .F. so no tabs appear. Instead, in a subclass, we'll set PageCount to the desired number of pages and put controls on each page, then set ActivePage to the desired value for the properties we want to currently show.

There isn't much code in this form. DisplayProperties accepts a page number parameter, sets pgfProperties.ActivePage to that value, and refreshes that page. Show calls oTreeViewContainer.LoadTree to load the TreeView. StatusBarClick, called when the user clicks the status bar, checks whether they clicked the "go back" panel, and if so, calls oTreeViewContainer.GoBack.

Using SFExplorerFormTreeView

Let's see how this works. DataDict.SCX (**Figure 4**) is a form based on SFExplorerFormTreeView. It displays data dictionary information stored in a table called RepMeta.DBF. The top-level nodes are tables, under which are "Fields" and "Relations" nodes. Under those nodes are fields and relations for the selected table. When the user selects a table, field, or relation, the form displays properties for the selected object. The status bar shows the type and name of the selected object.

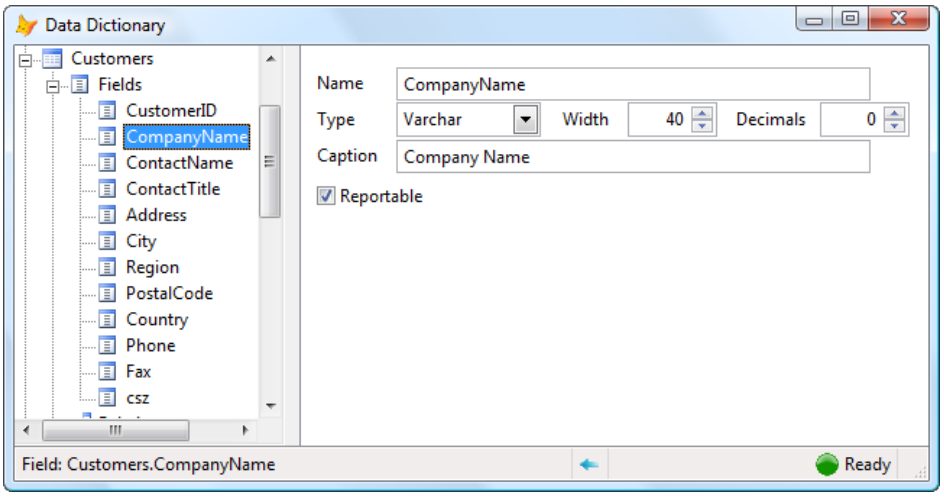


Figure 4. DataDict.SCX is a form based on SFExplorerFormTreeView.

I started by creating a form based on SFExplorerFormTreeView. I set cRegistryKey to "Software\Stonefield Software Inc.\Explorer Interfaces\DataDict Form" so the form's size and

position and the splitter location persist. I added the following code to the FillTreeViewCursor method of the oTreeViewContainer object:

```
local lnRecno, ;
  lcAlias, ;
  llRelations, ;
  lcParent, ;
  lcChild
select REPMETA
scan
  do case

* For table records, add the table to the cursor as well as "Fields" and
* "Relations" header records.

    case Rectype = 'C'
      insert into (This.cCursorAlias) (ID, TYPE, TEXT, IMAGE, PAGE) ;
        values (RepMeta.ObjectName, 'Table', RepMeta.ObjectName, ;
        'Table', 2)
      insert into (This.cCursorAlias) (ID, TYPE, PARENTID, PARENTTYPE, ;
          TEXT, IMAGE, PAGE) ;
        values (trim(RepMeta.ObjectName) + '~Fields', 'FieldHeader', ;
          RepMeta.ObjectName, 'Table', 'Fields', 'Field', 1)
      lnRecno = recno()
      lcAlias = padr(ObjectName, len(Alias1))
      locate for RecType = 'R' and (Alias1 = lcAlias or Alias2 = lcAlias)
      llRelations = found()
      go lnRecno
      if llRelations
        insert into (This.cCursorAlias) (ID, TYPE, PARENTID, ;
            PARENTTYPE, TEXT, IMAGE, PAGE) ;
          values (trim(RepMeta.ObjectName) + '~Relations', ;
            'RelationHeader', RepMeta.ObjectName, 'Table', ;
            'Relations', 'Relation', 3)
      endif llRelations

* Add the field record to the cursor.

    case Rectype = 'F'
      insert into (This.cCursorAlias) (ID, TYPE, PARENTID, PARENTTYPE, ;
          TEXT, IMAGE, PAGE) ;
        values (RepMeta.ObjectName, 'Field', ;
          trim(RepMeta.Alias) + '~Fields', 'FieldHeader', ;
          justext(RepMeta.ObjectName), 'Field', 1)

* For relation records, add records to the cursor for both the parent and child
* tables.

    case Rectype = 'R'
      lcParent = Alias2
      lcChild  = Alias1
      insert into (This.cCursorAlias) (ID, TYPE, PARENTID, PARENTTYPE, ;
          TEXT, IMAGE, PAGE) ;
        values (RepMeta.ObjectName, 'Relation', ;
          trim(lcChild) + '~Relations', 'RelationHeader', lcParent, ;
          'Relation', 3)
      insert into (This.cCursorAlias) (ID, TYPE, PARENTID, PARENTTYPE, ;
          TEXT, IMAGE, PAGE) ;
        values (RepMeta.ObjectName, 'Relation', ;
          trim(lcParent) + '~Relations', 'RelationHeader', lcChild, ;
          'Relation', 3)
  endcase
endscan
```

This code is straight-forward but I want to point out a few things:

- Tables, fields, and relations are stored in a single RepMeta table with the RECTYPE column distinguishing the record type. So, as the code goes through the records in the table, it uses a CASE statement to decide how to process each record type.

- The code specifies that page 1 of the properties pageframe is used for field nodes, page 2 for tables, and page 3 for relations.
- Although a table always has fields, it may not have any relations, so the "Relations" subnode is only added if there is at least one relation.
- The "Fields" and "Relations" subnodes of a table have the table name followed by "~Fields" and "~Relations" as their IDs.
- Because a relation belongs to both tables involved, a relation has two records in the TreeView cursor, one for each table.

Rather than hard-coding the images to load, oTreeViewContainer.LoadImages loads them from the GRAPHIC memo of an images table. KEY contains the key name to assign to the image.

```
local lnSelect, ;
  lnImage, ;
  lcKey, ;
  lcFile
with This.oImageList
  lnSelect = select()
  select 0
  use Datadict\Images
  lnImage = 1
  scan
    lcKey = trim(KEY)
    lcFile = lcKey + '.bmp'
    copy memo GRAPHIC to (lcFile)
    .ListImages.Add(lnImage, lcKey, loadpicture(lcFile))
    erase (lcFile)
    lnImage = lnImage + 1
  endscan
  use
  select (lnSelect)
endwith
```

The DeleteNode method, called when the user presses the Delete key, calls Thisform.DeleteObject. We won't look at the code for that method; it uses cCurrentNodeType and cCurrentNodeID to figure out which record to delete in the RepMeta table. The TreeAfterLabelEdit method, called after the user has finished editing the text of a node in the TreeView, calls Thisform.RenameObject. Like DeleteObject, it uses cCurrentNodeType and cCurrentNodeID to find the appropriate record in RepMeta and changes the OBJECTNAME column to the new name, updating related records as necessary (such as renaming the alias of fields when you rename a table).

Because not all nodes can be deleted or renamed (specifically, not the "Fields" and "Relations" nodes under a table), the form's DisplayProperties method prevents that from happening by setting lAllowDelete and lAllowRename to .F. We'll see later how those properties are set to .T. for some items.

```
lparameters tnPage
store .F. to This.oTreeViewContainer.lAllowDelete, ;
  This.oTreeViewContainer.lAllowRename
dodefault(tnPage)
```

I set the PageCount property of pgfProperties to 3 since we have three types of objects to show properties for. I created container classes for each type of object: TableProperties,

FieldProperties, and RelationProperties, all of which are defined in DataDict.VCX. These straight-forward classes simply have textboxes and other controls bound to fields in RepMeta. The Refresh method of each container ensures RepMeta is positioned to the desired record and places the object type and name into the status bar. Here's the code from FieldProperties.Refresh as an example:

```
local lcMessage
= seek('F' + upper(Thisform.cCurrentNodeID), 'RepMeta', 'Object')
store found() to Thisform.oTreeViewContainer.lAllowDelete, ;
  Thisform.oTreeViewContainer.lAllowRename
lcMessage = 'Field: ' + trim(RepMeta.ObjectName)
Thisform.UpdateMessagePanel(lcMessage)
```

So, as you can see, there isn't a lot of code needed to implement a TreeView-based explorer form. You just have to worry about how the TreeView is filled and what controls display the properties of the selected node. Of course, if you want additional behavior, such as deleting, adding, or renaming nodes and having those changes reflected back into the source data, you have a little more work to do, but all of the plumbing is provided in the classes in SFTreeView.VCX and SFExplorer.VCX.

Now let's get a little fancier.

SFExplorerFormOutlook

Figure 5 shows what the sample Northwind.SCX form looks like when it's running. Notice the Outlook-like bar at the left. This provides an additional level of hierarchy beyond what the TreeView provides. Some of the other features of this form are:

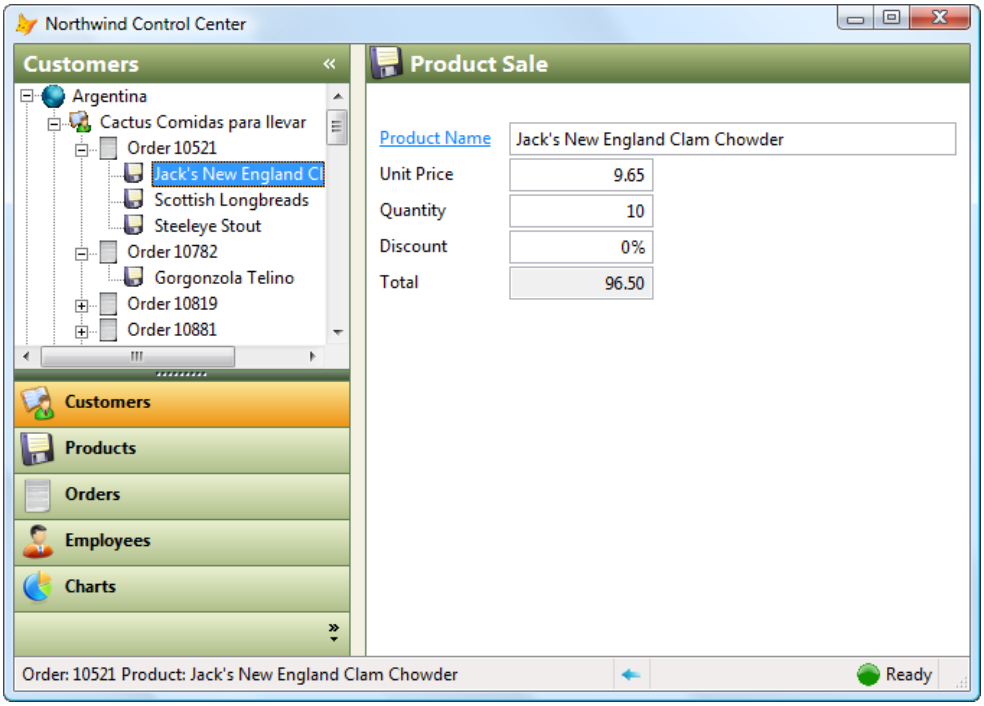


Figure 5. This form was created from the SFExplorerFormOutlook class.

- The user can select a different category or pane in the Outlook control by clicking the appropriate bar (for example, "Orders") or by clicking the small chevron (») near the bottom of the Outlook control to display a shortcut menu for the control and selecting the desired category. The selected category appears in the top pane and its bar has a different color to visually indicate it's selected.
- A category has both a title and an icon associated with it. The title appears above the top pane when the category is selected as well as in the bar for the category and in the shortcut menu. The icon appears in the bar and a smaller version in the menu.
- Clicking the reverse chevron («) near the top of the Outlook control collapses the control to the left so only icons are shown and the control inside the selected pane is hidden. Clicking it again expands the Outlook control to its original size.
- You can display fewer categories by dragging the splitter below the top pane down. This expands the top pane while eliminating other categories. You can also choose Show Less from the shortcut menu for the control. To display more categories, drag the splitter back up or choose Show More from the shortcut menu.
- The shortcut menu also allows you to select different themes, which affect the colors of the title bars of the top pane and properties page, the Outlook bars, and the space between the Outlook bar and properties page.
- The properties page for countries shows the total sales for the selected country (a SQL SELECT in the Refresh method of the container class calculates that value) and a Test Progress button so you can see how the progress bar works.
- Similarly, the properties page for products shows the total sales amount and total quantity sold for the selected product and the properties page for employees shows the total sales for each person, again calculated with SQL SELECT statements in the Refresh methods of the container classes.
- The properties page for orders has a link on the salesperson. Clicking the link makes the Employees category the active one, with the salesperson as the selected employee.
- Similarly, the properties page for a product in an order has a link to the Products category.
- The same properties page for orders is displayed whether you select an order from the Customers category or from the Orders category. Similarly, the same properties page is displayed for a product in an order regardless of which category you select it from.
- Two charts are available in the Charts category; each displays a chart in the properties page.
- You can drag a customer from one country to another to move the customer to that country.
- You can drag a vCard file from Windows Explorer or the Windows Desktop to the Customers TreeView to add a new customer.

This form is a subclass of SFExplorerFormOutlook. That class is similar to SFExplorerFormTreeView but rather than a TreeView container at the left, it has an

OutlookNavBar object named oNavBar. OutlookNavBar was written by Emerson Santon Reed and is one of the classes in the ThemedControls project of VFPX (www.codeplex.com/vfpx). It provides the Outlook bar for the form. Themes support is provided by another class from the ThemedControls project, ThemesManager.

Notice that SFExplorerFormOutlook doesn't contain any TreeView controls at all. Instead, you'll add instances of SFTreeViewOutlook, a subclass of SFTreeViewExplorer, to the various panes in a subclass or instance of SFExplorerFormOutlook. I won't go through all the code in this form class; instead, I'll just touch on the high points.

Although the OutlookNavBar object handles its own themes support via an instance of the ThemesManager class (which is instantiated as a member of _SCREEN so it's available to other forms in an application for a consistent theme), the form also needs themes support. To this end, I added an image object, which the Init of the class sizes to match the size of the form, and added a custom ChangeTheme method, which calls a method of the themes manager and sets the Picture property of the image to the result. The Init method of SFExplorerFormOutlook binds ChangeTheme to the ChangeTheme method of the themes manager so anything changing the theme automatically notifies the form so it can support the change. The various controls making up OutlookNavBar do the same thing.

In addition to saving and restoring the form size and position, the oPersist object must also handle the settings of the Outlook bar, specifically which category is selected and how many categories are displayed. The first value comes from the SelectedButton property of the Outlook bar and the second from a custom property of the form, nNavBarButtonsShown, which gets its value from the ShowedButtons property of the Outlook bar. The Init method of SFExplorerFormOutlook registers these two properties with the oPersist object so when it does its normal save and restore processes, they are also handled. Restoring SelectedButton automatically selects the specified category but restoring ShowedButtons is a little trickier: rather than setting its value, you have to call ShowLess to reduce the number of categories as necessary. Due to the timing of how events occur, you have to do this once the form is visible, so the first time Activate is called, it calls AdjustShownButtons. That method calls oNavBar.ShowLess in a DO WHILE loop until the previously saved number of categories is shown. When the form closes, the current value of oNavBar.ShowedButtons is copied to nNavBarButtonsShown so oPersist can save it to the Registry.

Although the Outlook bar takes care of itself when it collapses, the splitter and properties pageframe must adjust. The ViewModeChanged method of oNavBar, called when the Outlook bar expands or collapses, takes care of this. When the control collapses, the code saves the current value of the splitter's nObject1MinSize, which determines how narrow the left object (in this case, the Outlook bar) can be, sets that property to the current width of the Outlook bar, and disables the splitter. When the control expands, it restores nObject1MinSize to the saved value and enables the splitter. In either case, it moves the splitter so it's just to the right of the Outlook bar. Doing so automatically resizes the properties pageframe. Thus, the net effect is that when the Outlook bar collapses or expands, the properties pageframe width is adjusted to take up the rest of the form width.

What happens when you click the "go back" panel in the status bar of SFExplorerFormOutlook is a little more complicated than it is for SFExplorerFormTreeView because the node that was previously selected might be in a TreeView in a different category than the current one. To make this work, I added an instance of SFStack (discussed in Appendix A),

which manages the order in which items were selected, to the form. SFTreeViewOutlook, the class you'll use for TreeViews in SFExplorerFormOutlook, just has one change from its parent SFTreeViewExplorer: rather than saving the current node to its own stack, it saves a value comprised of its name, a tilde (~), and the selected node's key to the form's stack. The BeforeChangeSelectedButton method of oNavBar, called when you click a different category, pushes the previously selected category onto the form's stack. Thus, when you click the "go back" panel, there could be one of two types of things to pop off the stack: a key to the previously selected node in the specified TreeView or the number of the previously selected category. The StatusBarClick method of the form, called when you click the status bar, has the following code to decide what to do based on what it finds: either selecting the previous category (and of course not storing that on the stack, so lPushButtonOnStack is temporarily set to .F., which tells BeforeChangeSelectedButton to do nothing) or selecting the previous node in the TreeView of the current category.

```
local loPanel, ;
  luPop, ;
  lnPos, ;
  lcControl, ;
  lcKey, ;
  loPane
with This
  if .oStatus.nPanel > 0
    loPanel = .oStatus.ctlPanels(.oStatus.nPanel)
    if vartype(loPanel) = 'O' and loPanel.ctlName = 'Back'
      luPop = .oStack.Pop()
      do case
        case vartype(luPop) = 'N' and luPop <> .oNavBar.SelectedButton
          .lPushButtonOnStack     = .F.
          .oNavBar.SelectedButton = luPop
          .lPushButtonOnStack     = .T.
        case vartype(luPop) = 'C'
          lnPos     = at('~', luPop)
          lcControl = left(luPop, lnPos - 1)
          lcKey     = substr(luPop, lnPos + 1)
          loPane    = .oNavBar.Panes.Pages[.oNavBar.SelectedButton]
          loPane.&lcControl..SelectNode(lcKey, .T.)
      endcase
    endif vartype(loPanel) = 'O' ...
  endif .oStatus.nPanel > 0
endwith
```

DisplayProperties has the same code SFExplorerFormTreeView does to select the specified page in the properties pageframe and refresh that page. Since both classes have the same code, why not put it into their parent, SFExplorerForm? Because that class doesn't have a pageframe and it's possible you could use it in another subclass that doesn't require one. Also, while the code is currently the same, that may change in the future, so I decided against creating a subclass of SFExplorerForm that both SFExplorerFormTreeView and SFExplorerFormOutlook descend from and placing the code in the DisplayProperties method of that class.

When the user selects a different category, we have to make sure the properties for the selected node in the TreeView of that category are displayed (assuming there is a TreeView in that category, which isn't necessarily the case). Since the user didn't click the node, we can't rely on our existing mechanism to handle that. Instead, the ButtonClicked method of oNavBar, which is called when the user clicks a category, calls the DisplayRecord method of the TreeView container object in the new pane, if there is one, resulting in the same effect as if the user actually clicked a node. Thus, when you move from category to category, you see the previously selected node in the current category and the properties for that node at the right.

Speaking of properties, the properties pageframe isn't an instance of SFPageFrame like it is in SFExplorerFormTreeView. Instead, it's an instance of ThemedTitlePageFrame, one of the themed controls, so the properties page displays the proper icon and title for the selected category and the pageframe supports themes.

Now that we've gone through the plumbing, let's see how it works.

Using SFExplorerFormOutlook

Here's how I created Northwind.SCX. First, I created a form based on SFExplorerFormOutlook. So the form can stand alone, I added code to Load to SET PATH appropriately and open the sample Northwind database (I included a copy of this database with the sample code for this document because this form allows you to change the data and I didn't want to mess up the copy under your VFP home directory). The Unload method cleans up the path and closes the database.

Next, I set the PageCount property of the Panes pageframe in the Outlook bar to 5 to create five panes. For each page, I set Caption to the desired caption and Picture16 and Picture24 to the names of the image files to use for the icons. For example, for page 1, the Customers category, Caption is "Customers" and Picture16 and Picture24 are both "Images\Customer.PNG." I then added an SFTreeViewOutlook object to the each page, named them oTreeViewContainer, filled in code in the FillTreeViewCursor and LoadImages methods to load the nodes and images appropriate for the category, and set cRegistryKeySuffix to the category name so the settings for the TreeView are persisted to a subkey of the form's key in the Registry. For example, for the Customers category, it loads countries as the top-level nodes, then customers in each country, then the orders for each customer, and finally the products sold in each order. Each node type in the cursor specifies which page of the properties pageframe to activate so the appropriate properties are shown.

Just to show off, I added some more code to the oTreeViewContainer object in the Customers pane. I wanted to support drag and drop in two ways: dragging a customer from one country to another to move it to the new country and dragging a vCard from Windows Explorer to add a new customer. First, CanStartDrag, which determines whether a node can be dragged or not, only returns .T. if the selected node is a customer.

```
return This.cCurrentNodeType = 'Customer'
```

CanDrop, called when the mouse moves over a node during a drag operation, only returns .T. if the object being dragged is a vCard, in which case we don't care where it's dropped, or is a customer node, in which case it can only be dropped on a country node different from its current country. Setting tnEffect, which is passed by reference to this method, controls the appearance of the mouse pointer.

```
lparameters toData, ;
  toNode, ;
  toObject, ;
  tnEffect, ;
  tnButton, ;
  tnShift
local llReturn, ;
  llCopy, ;
  lcFile, ;
  lnHandle, ;
  lcLine, ;
  lcCountry, ;
  lcCustomerID, ;
```

```
  lnRecno
do case

* If a file is being dragged, ensure it's a VCard.

  case toData.GetFormat(CF_FILES)
    llReturn = .T.
    llCopy   = .T.
    for each lcFile in toData.Files
      llReturn = upper(justext(lcFile)) = 'VCF'
      if not llReturn
        exit
      endif not llReturn
    next lcFile

* If a node is being dragged, ensure it's a customer being dragged onto a
* different country.

  case toData.GetFormat(CF_MAX) and toObject.DragType = 'Customer' and ;
    toObject.DropType = 'Country'
    lcCountry    = toNode.Text
    lcCustomerID = toObject.DragKey
    lnRecno      = recno('Customers')
    = seek(padr(lcCustomerID, len(Customers.CustomerID)), 'Customers', ;
      'CustomerID')
    llReturn = not trim(Customers.Country) == lcCountry
    go lnRecno in Customers
endcase

* Set the drop effect.

do case
  case not llReturn
    tnEffect = DROPEFFECT_NONE
  case llCopy
    tnEffect = DROPEFFECT_COPY
  otherwise
    tnEffect = DROPEFFECT_MOVE
endcase
return llReturn
```

Finally, HandleDragDrop deals with the drop event. In the case of a vCard, it calls ReadVCard to parse the vCard file and return an object with properties matching the field names in the Customers table, adds a record to the Customers table, reloads the TreeView, and selects the newly added customer. For a customer being dragged to another country, it replaces the value of COUNTRY in the customer record and reloads the TreeView.

```
lparameters toData, ;
  toNode, ;
  toObject
local lcFile, ;
  lcVCard, ;
  loContact, ;
  lcKey, ;
  lcCountry, ;
  lcCustomerID
with This
  do case

* The user dropped a list of files, so import the VCard.

    case toObject.DragType = 'ImportFiles'
      for each lcFile in toObject.Files
        lcVCard   = filetostr(lcFile)
        loContact = Thisform.ReadVCard(lcVCard)
        insert into Customers from name loContact
        lcKey = .GetNodeKey('Customer', loContact.CustomerID)
      next lcFile
      .LoadTree()
      .SelectNode(lcKey)
```

```
* The user dragged a customer to a different country, so move them.

    case toData.GetFormat(CF_MAX) and toObject.DragType = 'Customer' and ;
      toObject.DropType = 'Country'
      lcCountry    = toNode.Text
      lcCustomerID = '' + toObject.DragKey
      = seek(padr(lcCustomerID, len(Customers.CustomerID)), ;
        'Customers', 'CustomerID')
      replace Country with lcCountry in Customers
      .LoadTree()
  endcase
endwith
```

The Customers TreeView supports two other behaviors. DeleteNode has code that deletes a customer from the Customers table and the TreeView when the user presses Delete. However, only customers who don't have any orders should be deleted, so DisplayRecord only sets lAllowDelete if a customer node is selected and that customer has no orders. Double-clicking a product in the TreeView should select that product in the Products category so you can see information about that product, so TreeDblClick does that. As you may guess, it was adding this ability that prompted me to add the "go back" functionality, since you may look at a product in an order for a customer, want to see more information about that product, then return to the order.

```
local lcKey, ;
  lnPage
if This.cCurrentNodeType = 'LineItem'
  lcKey  = This.GetNodeKey('Product', OrderDetails.ProductID)
  lnPage = 2
  Thisform.SelectNodeInPanel(lnPage, lcKey)
endif This.cCurrentNodeType = 'LineItem'
```

The fifth pane in the Outlook bar displays the names of charts you can view. The names of these charts come from the Charts table. We'll see how the charts are drawn later.

Now that we have the Outlook bar taken care of, let's add the properties containers. I created one container class for each type of item I want properties displayed for; all of these are subclasses of SFThemedPropertiesContainer, also defined in SFExplorer.VCX. SFThemedPropertiesContainer is a subclass of ThemedContainer, another class in the ThemedControls VFPX project. There are only two changes in SFThemedPropertiesContainer: BorderWidth is 0 so no border appears and Init positions the container to the proper location on the page. You'd think you could just drop a container on a page in a pageframe and when you run the form, it'd appear in the location you specified in the Form or Class Designer. Unfortunately, that's not the case here for reasons I haven't yet discovered; instead, the container appears in the upper left corner of the page. So, I added nPropertyContainerTop and nPropertyContainerLeft properties to SFExplorerFormOutlook, set them to default values to position containers in a reasonable location (although you can, of course, change these values as necessary), and added code to the Init method of SFThemedPropertiesContainer to position the container to the specified location and size the container to fill the page.

The container classes for the properties are stored in Northwind.VCX. All are relatively straightforward: controls bound to fields in the appropriate table and code in Refresh to navigate to the record in the table for the selected node and update the message panel of the status bar. For example, here's the code from CustomerProperties.Refresh:

```
local lcMessage
= seek(padr(Thisform.cCurrentNodeID, len(Customers.CustomerID)), 'Customers', ;
```

```
  'CustomerID')
lcMessage = 'Customer: ' + trim(Customers.CompanyName)
Thisform.UpdateMessagePanel(lcMessage)
```

Some of the container classes have additional code in Refresh to calculate values. For example, OrderProperties calculates the total of all line items as a subtotal and calculates the grand total as the subtotal plus freight. Also, LineItemProperties and OrderProperties have hyperlinked labels that navigate to the selected product and salesperson, respectively. They use code in the Click method of those labels similar to the TreeDblClick method of the Customers TreeView we saw earlier. CountryProperties also has a Test Progress button with the following code in Click to show how the progress bar in the status bar works:

```
Thisform.UpdateStatePanel('Testing...')
for lnI = 1 to 100
  Thisform.UpdateProgressBar(lnI)
  inkey(0.005, 'H')
next lnI
Thisform.HideProgressBar()
Thisform.UpdateStatePanel()
```

After creating the containers, I set the PageCount property of pgfProperties to the desired number of pages and set the Caption and Picture16 properties of each page to the appropriate values. Finally, I dropped an instance of the appropriate container class on each page.

Selecting a chart in the Chart pane of the Outlook control displays the page of the properties pageframe that has an instance of ChartProperties. This class has an instance of the FoxChart control, yet another VFPX project used in these samples. FoxChart uses GDI+ to create gorgeous 3-D charts. The Refresh method of the container finds the appropriate record in the Charts table, executes the code stored in the SQL memo of that table to create a cursor used as the data source for the chart, and then sets the properties of the FoxChart object as specified in other fields in the Charts table, such as what type of chart to draw, the title captions for the chart and X and Y axis, and so on. Finally, Refresh calls the DrawChart method of the FoxChart object to draw the chart. **Figure 6** shows an example of one of the charts.

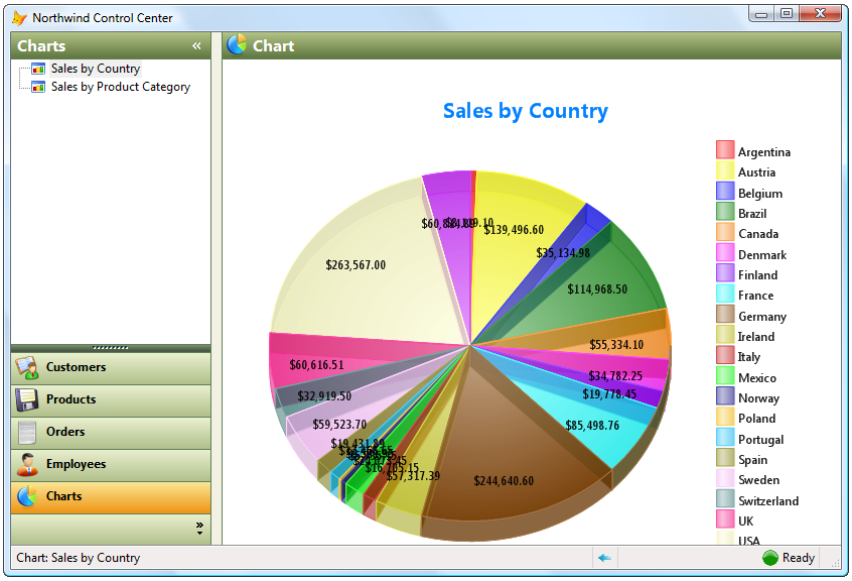


Figure 6. FoxCharts makes creating charts fast and simple.

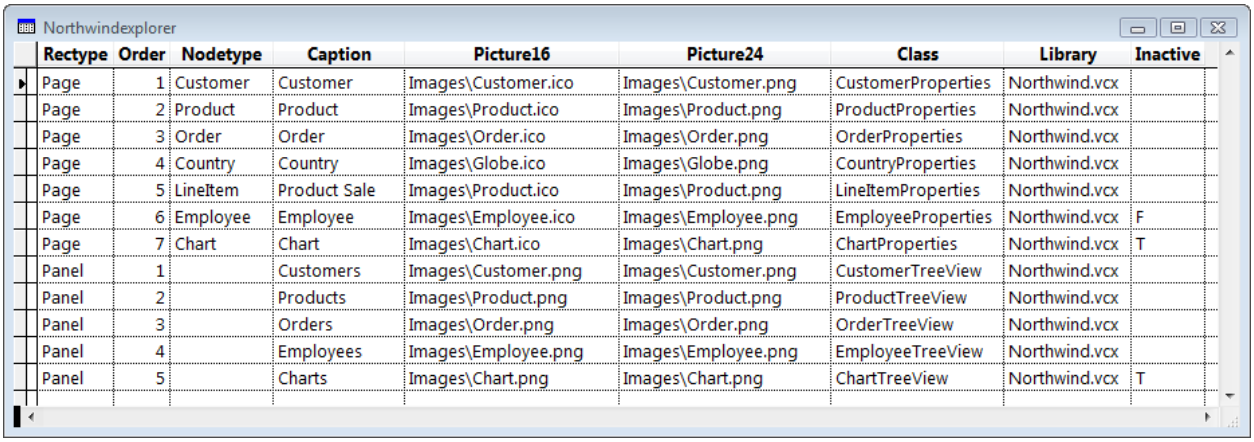
SFExplorerFormOutlookDataDriven

Even though most of the behavior of the Outlook control, status bar, TreeView, and form is taken care of for you, if it still seems like a lot of work to create an explorer form, you'll like SFExplorerFormOutlookDataDriven. It's a data-driven subclass of SFExplorerFormOutlook. Its Show method uses a table whose name is specified in the cOutlookTable property of the class to define how many categories the Outlook bar should have, what the names and images are for those categories, what controls to add to each pane, and which container classes display the properties for each type of node in the various TreeViews. The table has the structure shown in **Table 2**.

Table 2. The structure of the table used by SFExplorerFormOutlookDataDriven.

Field	Type	Purpose
RECTYPE	C(10)	Either "Page" for controls in the properties pages or "Panel" for controls in panels in the Outlook bar.
ORDER	N(2)	The order in which this page or panel should be added.
NODETYPE	C(30)	Specifies the TreeView node type that this control displays properties for (only used for Page records).
CAPTION	C(30)	The caption for the page or panel.
PICTURE16	C(60)	For Page records, the icon to use in the TreeView for nodes of the type specified in NODETYPE. For Panel records, the image to use for the Picture16 property.
PICTURE24	C(60)	The image to use for the Picture24 property of the page or panel.
CLASS	C(30)	The class to use for the TreeView in the panel or the container in the properties page.
LIBRARY	C(60)	The library containing the class specified in CLASS.
INACTIVE	L	.T. to ignore this record.

Figure 7 shows the content of NorthwindExplorer.DBF, used by NorthwindDataDriven.PRG to create a data-driven version of the Northwind form.


Northwindexplorer

Rectype	Order	Nodetype	Caption	Picture16	Picture24	Class	Library	Inactive
Page	1	Customer	Customer	Images\Customer.ico	Images\Customer.png	CustomerProperties	Northwind.vcx	
Page	2	Product	Product	Images\Product.ico	Images\Product.png	ProductProperties	Northwind.vcx	
Page	3	Order	Order	Images\Order.ico	Images\Order.png	OrderProperties	Northwind.vcx	
Page	4	Country	Country	Images\Globe.ico	Images\Globe.png	CountryProperties	Northwind.vcx	
Page	5	LineItem	Product Sale	Images\Product.ico	Images\Product.png	LineItemProperties	Northwind.vcx	
Page	6	Employee	Employee	Images\Employee.ico	Images\Employee.png	EmployeeProperties	Northwind.vcx	F
Page	7	Chart	Chart	Images\Chart.ico	Images\Chart.png	ChartProperties	Northwind.vcx	T
Panel	1		Customers	Images\Customer.png	Images\Customer.png	CustomerTreeView	Northwind.vcx	
Panel	2		Products	Images\Product.png	Images\Product.png	ProductTreeView	Northwind.vcx	
Panel	3		Orders	Images\Order.png	Images\Order.png	OrderTreeView	Northwind.vcx	
Panel	4		Employees	Images\Employee.png	Images\Employee.png	EmployeeTreeView	Northwind.vcx	
Panel	5		Charts	Images\Chart.png	Images\Chart.png	ChartTreeView	Northwind.vcx	T

Figure 7. The structure and content of a table used by SFExplorerFormOutlookDataDriven.

As the code in SFExplorerFormOutlookDataDriven.Show goes through the records in this table, it creates enough pages in the Outlook bar to hold the number of panels specified and sets the Caption, Picture16, and Picture24 properties of each page to those specified in the table. It also adds an instance of the class and library specified to the page. This means, of course, that you'll have to create subclasses of SFTreeViewOutlook (actually, SFTreeViewOutlookDataDriven that overrides LoadImages to load the images specified in the table and DisplayRecord to find the page number to display from the table) that have the desired code in FillTreeViewCursor, plus any other methods you need to support the desired behavior. Show also creates enough pages in the properties pageframe for the types of nodes you want properties displayed for, sets the Caption and Picture16 properties of those pages, and adds an instance of the container specified in the CLASS and LIBRARY fields to each page. The

NODETYPE field is used by the form to associate a node type, coded in the FillTreeViewCursor method of the TreeViews, with a particular page in the properties pageframe.

While you still need to create the TreeView and properties container classes and add records to the table, you don't have to create an instance of SFExplorerFormOutlookDataDriven to create an explorer form. NorthwindDataDriven.PRG is a simple program that displays a data-driven version of the Northwind form.

```
* Set a path to the Source folder.

set path to ..\Source, ..\Source\Bmps, ..\Source\ThemedControls

* Open the Northwind sample database.

close tables all
open database 'Northwind\Northwind'

* Create a generic SFExplorerFormOutlookDataDriven, set its properties, and
* call Show.

loForm = newobject('SFExplorerFormOutlookDataDriven', 'SFExplorer.vcx')
with loForm
  .Caption       = 'Northwind Control Center - Data-Driven'
  .cOutlookTable = 'NorthwindExplorer.dbf'
  .cRegistryKey  = 'Software\Stonefield Software Inc.' + ;
    '\Explorer Interfaces\Northwind Form'
endwith
loForm.Show(1)
close tables all
```

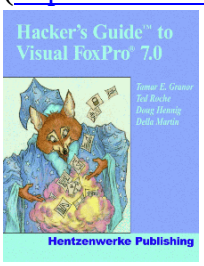
Summary

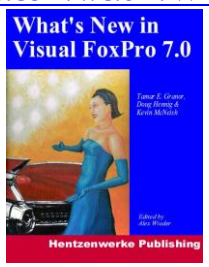
Creating an explorer interface for an application means wiring up the interactions between several types of components, including a TreeView to display items and a set of controls showing properties about the selected item. It can be tricky to get everything working just right, but the set of classes provided in the source code accompanying this document makes it quite easy. An explorer interface isn't necessarily the right choice for all applications, but if your data is somewhat hierarchical in nature, such an interface might make your application more visually appealing, easier to use, and more modern in appearance.

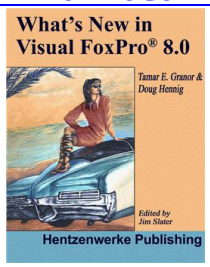
Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. and 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 the "What's New in Visual FoxPro" series, "The Hacker's Guide to Visual FoxPro 7.0," and the soon-to-be-released "Making Sense of Sedna and VFP 9 SP2." 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). Doug wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor and Advisor Guide. He currently writes for FoxRockX (http://www.foxrockx.com). He spoke at every Microsoft FoxPro Developers Conference (DevCon) since 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://www.codeplex.com/VFPX). He has been a Microsoft Most Valuable Professional (MVP) since 1996. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://fox.wikis.com/wc.dll?Wiki~FoxProCommunityLifetimeAchievementAward).







Appendix A. SFTreeViewContainer

SFTreeViewContainer, defined in SFTreeView.VCX, is a subclass of SFContainer (my base class Container located in SFCtrls.VCX). It contains both TreeView and ImageList controls and code in the Init method of the container associates the two by setting the TreeView's ImageList property to a reference to the ImageList control. **Figure 8** shows SFTreeViewContainer in the Class Designer.

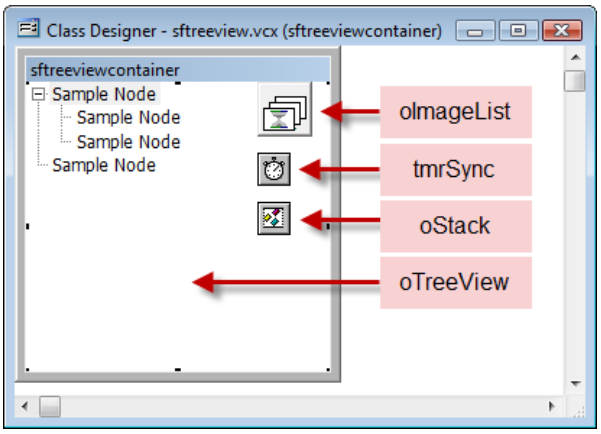


Figure 8. SFTreeViewContainer provides all the behavior you'd ever need in a TreeView control.

SFTreeViewContainer also contains a timer named tmrSync. Sometimes clicking a node in the TreeView doesn't fire NodeClick. So, the timer periodically checks whether the node we think is selected matches the actual selection, and if not, fires NodeClick.

Finally, SFTreeViewContainer also contains a subclass of Collection called SFStack that has Push and Pop methods so it acts like a stack (I blogged about SFStack at http://doughennig.blogspot.com/2008/05/stacking-stuff.html). This instance pushes node clicks onto a stack so it provides "go back" functionality similar to a browser. You can disable this functionality by setting the lTrackNodeClicks property to .F.

Most TreeView events call methods of the container for several reasons, including that good design dictates that events should call methods and it's easier to get at the PEMs of the container than to drill down into the TreeView in the Class Designer. I considered using BINDEVENTS() so I didn't have to put any code at all into the TreeView events, but some TreeView events, such as BeforeLabelEdit, receive their parameters by reference, which BINDEVENTS() doesn't support.

TreeView appearance

I set the properties of the TreeView control visually using the TreeCtrl Properties window as shown in **Figure 9**. If you want to use different settings in a subclass, such as using 0 for Appearance so the TreeView appears flat rather than 3-D, you'll need to change them in code; for some reason, changing the properties of the TreeView in a subclass doesn't seem to work. So, in the Init method of a subclass of SFTreeViewContainer, use DODEFAULT() followed by code that changes the properties of the TreeView as desired.

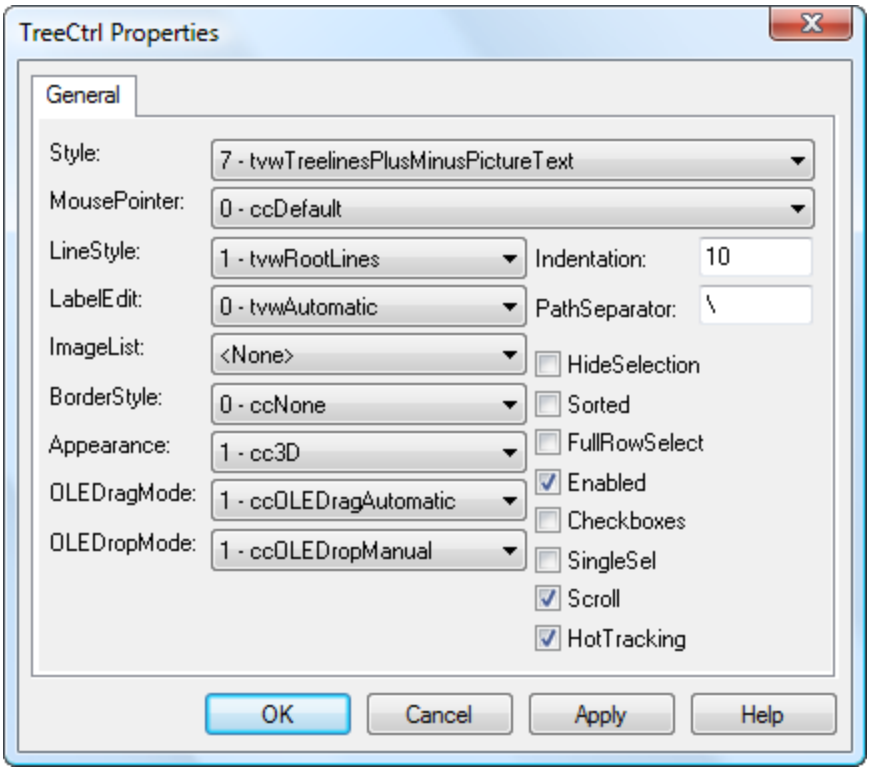


Figure 9. The properties of the TreeView control are set visually.

The images used for the TreeView come from the ImageList control in the container. As with the TreeView, loading images into the ImageList visually in a subclass doesn't seem to work, so the Init method of SFTreeViewContainer calls the abstract LoadImages method. In a subclass, call the Add method of the ListImages collection of the ImageList object to add images. Note that you have to use the VFP LOADPICTURE() function to pass a reference to a loaded image object to the Add method. By default, the ImageHeight and ImageWidth properties of the ImageList control are set to 16, so change these values in LoadImages if your images use a different size. Here's an example of code for LoadImages that loads PAGE.GIF as image 1 with a key of "Page" and COMPONENT.GIF as image 2 with a key of "Component":

```
with This.oImageList
  .ListImages.Add(1, 'Page',      loadpicture('PAGE.GIF'))
  .ListImages.Add(2, 'Component', loadpicture('COMPONENT.GIF'))
endwith
```

Loading the TreeView

There are four custom properties of SFTreeViewContainer associated with loading the TreeView with the proper nodes:

- lLoadTreeViewAtStartup: if this property is .T. (the default), the container's Init method calls the LoadTree method to load the TreeView. If it's .F., you'll have to manually call LoadTree when you want the TreeView loaded. This might be necessary if the Init of the form the container resides in must do some work before the TreeView is loaded.

- lAutoLoadChildren: if this property is .T., all nodes in the entire TreeView are loaded. This is fine if there aren't very many, but can cause a serious performance issue otherwise; it might take so long to load the TreeView that the user thinks the application has crashed. In that case, set lAutoLoadChildren to .F. (the default setting). The container will only load the top-level nodes (or more, depending on other settings described later). Any loaded node that has child nodes will have a "Loading…" dummy node added under it rather than the real child nodes so the + will appear for the node, indicating it can be expanded. When a node is expanded for the first time, the TreeExpand method of the container (called from the Expand method of the TreeView) removes the dummy node and adds the real child nodes.
- nAutoExpand: this property indicates to what level the nodes of the TreeView are automatically expanded when the TreeView is loaded. The default is 0, meaning none of the nodes are expanded (although, as you'll see later, restoring the TreeView to its last viewed state will override this for some nodes). Setting it to 1 means that the top-level nodes should be expanded; this means that even if lAutoLoadChildren is .F., the container will load the immediate children (but not the children's children) of the top-level nodes. Setting it to 2 means the immediate children of the top-level nodes should be expanded (causing their immediate children to be loaded), and so forth.
- lUsePathAsKey: if this property is .F., you'll assign unique key values to nodes in the TreeView. If it's .T. (the default), you don't have to worry about keys; the container will use the path of the node (similar to the path of a file) as its key.

The LoadTree method does the main work of loading the TreeView. It starts by calling the LockTreeView method to lock the TreeView control (using a Windows API function) so updates aren't displayed as they're performed. Next, any existing nodes that are currently expanded are saved into the aExpandedNodes array property and the key for the currently selected node is saved in the cLastNode property; this allows you to call LoadTree a second time to refresh the contents (for example, if the TreeView displays records from a table that other users on a network may also be editing) and have the expanded state of each node and the currently selected node restored. LoadTree then loads the top-level nodes by calling the GetRootNodes method, passing it a Collection object. GetRootNodes is an abstract method that you must implement in a subclass to fill the collection with objects containing information about the top-level nodes; I'll discuss this further later. LoadTree then calls LoadNode for each object in the collection to load it into the TreeView. Any formerly expanded nodes are re-expanded and the former selected node is reselected (if these nodes still exist). Finally, LoadTree calls LockTreeView again to unlock the TreeView.

```
lparameters tlNoSelect
local lnExpandedNodes , ;
  lnI, ;
  laExpandedNodes[1], ;
  loNodesCollection, ;
  loNodeItem, ;
  lcKey, ;
  loNode
with This

* Lock the TreeView so we don't see updates until they're done.

  .LockTreeView(.T.)
```

```
* If we have any existing nodes, let's save the keys of the expanded ones and
* the selected one so we can restore them later, then nuke the nodes. That way,
* LoadTree can be called to refresh or reload the TreeView.

  with .oTree
    if .Nodes.Count > 0
      lnExpandedNodes = 0
      for lnI = 1 to .Nodes.Count
        if .Nodes[lnI].Expanded
          lnExpandedNodes = lnExpandedNodes + 1
          dimension This.aExpandedNodes[lnExpandedNodes]
          This.aExpandedNodes[lnExpandedNodes] = .Nodes[lnI].Key
        endif .Nodes[lnI].Expanded
      next lnI
      if vartype(.SelectedItem) = 'O'
        This.cLastNode = .SelectedItem.Key
      else
        This.cLastNode = ''
      endif vartype(.SelectedItem) = 'O'
      .Nodes.Clear()
    endif .Nodes.Count > 0
  endwith

* Get the root nodes and add them to the TreeView.

  loNodesCollection = createobject('Collection')
  .GetRootNodes(loNodesCollection)
  for each loNodeItem in loNodesCollection foxobject
    .LoadNode(loNodeItem)
  next loNodeItem

* Re-expand any previously expanded nodes.

  with .oTree
    for lnI = 1 to alen(This.aExpandedNodes)
      lcKey = This.aExpandedNodes[lnI]
      if type('.Nodes[lcKey]') = 'O'
        This.TreeExpand(.Nodes[lcKey], .T.)
      endif type('.Nodes[lcKey]') = 'O'
    next lnI
  endwith

* If we're supposed to re-select the node selected last time and that node
* still exists, select it. Otherwise, select the first node.

  loNode = .NULL.
  do case
    case tlNoSelect
    case not empty(.cLastNode) and type('.oTree.Nodes[.cLastNode]') = 'O'
      loNode = .oTree.Nodes[.cLastNode]
    case .oTree.Nodes.Count > 0
      loNode = .oTree.Nodes[1]
  endcase
  if not isnull(loNode)
    .SelectNode(loNode)
  endif not isnull(loNode)

* Unlock the TreeView.

  .LockTreeView()
endwith
```

You must fill in code in the GetRootNodes method of a subclass or instance of SFTreeViewContainer to fill the collection with objects containing information about the top-level nodes. To create such an object, call the CreateNodeObject method; it simply creates an Empty object and adds the following properties:

- Key: the key for the node in the TreeView. This is automatically set to SYS(2015) but you should change it to a unique value if the lUsePathAsKey property is .F.

- Text: the text to display for the node.
- Image: the name or number of an image in the ImageList control for this node.
- SelectedImage: the name or number of an image in the ImageList control to use when this node is selected. You can leave this blank to use the same value as Image.
- ExpandedImage: the name or number of an image in the ImageList control to use when this node is expanded. You can leave this blank to use the same value as Image.
- Sorted: .T. if the node should be sorted.
- HasChildren: .T. if this node has any children.

You can override CreateNodeObject if you need additional properties. In that case, you should also override GetNodeItemFromNode, which creates and fills the properties of a node item object from the current TreeView node.

GetRootNodes should create one object for each top-level node, fill in the properties, and add it to the passed-in collection. Here's some sample code that creates two top-level nodes: "Pages" and "Components."

```
lparameters toCollection
local loNodeItem
loNodeItem = This.CreateNodeObject()
with loNodeItem
  .Key         = 'HP'
  .Text        = 'Pages'
  .Image       = 'Page'
  .Sorted      = .T.
  .HasChildren = .T.
endwith
toCollection.Add(loNodeItem)

loNodeItem = This.CreateNodeObject()
with loNodeItem
  .Key         = 'HC'
  .Text        = 'Components'
  .Image       = 'Component'
  .Sorted      = .T.
  .HasChildren = .T.
endwith
toCollection.Add(loNodeItem)
```

You must also implement the GetChildNodes method. Like GetRootNodes, this method fills a collection with node item objects. In this case, these objects represent the children of a node. GetChildNodes is passed three parameters: the type and ID for the node whose children are needed and the collection to fill. Here's some sample code. In the case of the "Pages" root node, the records in the PAGES table are loaded into the collection, with "P" and the ID as the value for Key. For the "Components" root node, the records in the CONTENT table are loaded, using "C" and the ID for Key.

```
lparameters tcType, ;
  tcID, ;
  toCollection
local loNodeItem, ;
  lcWhere
do case

* If this is the "Pages" node, fill the collection with the pages.

  case tcType = 'HP'
```

```
    select PAGES
    scan
      loNodeItem = This.CreateNodeObject()
      with loNodeItem
        .Key         = 'P' + transform(ID)
        .Text        = trim(PAGE)
        .Image       = 'Page'
        .HasChildren = seek(PAGES.ID, 'PAGECONTENT', 'PAGEID')
      endwith
      toCollection.Add(loNodeItem)
    endscan

* If this is the "Components" node, fill the collection with the components.

  case tcType = 'HC'
    select CONTENT
    scan
      loNodeItem = This.CreateNodeObject()
      with loNodeItem
        .Key   = 'C' + transform(ID)
        .Text  = trim(NAME)
        .Image = 'Component'
      endwith
      toCollection.Add(loNodeItem)
    endscan

* We have a bad node type, so do nothing (we don't actually need the OTHERWISE statement, but
* this way, I won't get heck from Andy Kramek <gd&rfAK>).

  otherwise
endcase
```

Restoring the TreeView state

When you run a form that uses SFTreeViewContainer a second time, you may expect it to appear the same as it did last time: the node that was selected is still selected and all the nodes that were expanded are still expanded. This is handled by the SaveSelectedNode and RestoreSelectedNode methods, which are called from Destroy and Init, respectively.

SaveSelectedNode saves the keys of the selected node and all expanded nodes to the Windows Registry. The GetRegistryKey method determines which Registry key under HKEY_CURRENT_USER to use by the values of the cRegistryKey and cRegistryKeySuffix properties. If cRegistryKey is filled in (for example, “Software\My Company\My Application\My Form”), that key is used. If the form the SFTreeViewContainer is on has a cRegistryKey property (for example, “Software\My Company\My Application\My Form”) and the SFTreeViewContainer’s cRegistryKeySuffix property is filled in (for example, “My TreeView”), the concatenation of those two is used (“Software\My Company\My Application\My Form\My TreeView” to follow the example). SaveSelectedNode then instantiates an SFRegistry object and uses it to save the key of the selected node to the SelectedNode entry in the Registry. SaveSelectedNode goes through all nodes in the TreeView and saves the keys of those that are expanded to the Expanded*N* entry, where *N* is simply a node number. Finally, it removes any additional Expanded entries in the Registry in case there were more expanded nodes the last time than there are now.

```
local lcKey, ;
  loRegistry, ;
  lnNode, ;
  lnI, ;
  loNode, ;
  laValues[1], ;
  lnValues, ;
  lcValue
with This.oTree
```

```
  lcKey = This.GetRegistryKey()
  if not empty(lcKey)
    loRegistry = newobject('SFRegistry', 'SFRegistry.vcx')
    if vartype(.SelectedItem) = 'O'
      loRegistry.SetKey(lcKey, 'SelectedNode', .SelectedItem.Key)
    endif vartype(.SelectedItem) = 'O'
    lnNode = 0
    for lnI = 1 to .Nodes.Count
      loNode = .Nodes(lnI)
      if loNode.Expanded
        lnNode = lnNode + 1
        loRegistry.SetKey(lcKey, 'Expanded' + transform(lnNode), ;
          loNode.Key)
      endif loNode.Expanded
    next lnI

* Remove extra lines from the Registry in case there were more expanded nodes
* the last time it was written to.

    lnValues = loRegistry.EnumerateKeyValues(lcKey, @laValues)
    for lnI = 1 to lnValues
      lcValue = laValues[lnI, 1]
      if lcValue = 'Expanded' and val(substr(lcValue, 9)) > lnNode
        loRegistry.DeleteKeyValue(lcKey, lcValue)
      endif lcValue = 'Expanded' ...
    next lnI
  endif not empty(lcKey)
endwith
```

RestoreSelectedNode uses similar code to restore the previously selected node to the cLastNode property and the previously expanded nodes to the aExpandedNodes array.

```
local lcKey, ;
  loRegistry, ;
  lnNode, ;
  laValues[1], ;
  lnValues, ;
  lnI, ;
  lcValue
with This
  lcKey = .GetRegistryKey()
  if not empty(lcKey)
    loRegistry = newobject('SFRegistry', 'SFRegistry.vcx')
    .cLastNode = loRegistry.GetKey(lcKey, 'SelectedNode')
    lnNode     = 0
    lnValues   = loRegistry.EnumerateKeyValues(lcKey, @laValues)
    for lnI = 1 to lnValues
      lcValue = laValues[lnI, 1]
      if lcValue = 'Expanded'
        lnNode = lnNode + 1
        dimension .aExpandedNodes[lnNode]
        .aExpandedNodes[lnNode] = laValues[lnI, 2]
      endif lcValue = 'Expanded'
    next lnI
  endif not empty(lcKey)
endwith
```

Handling node selection

The NodeClick event of the TreeView fires when a node is selected. Since events should call methods, this event calls the TreeNodeClick method of the container. TreeNodeClick starts by checking whether the selected node is the "dummy" node; as noted in the comment, this can happen under some conditions, so TreeNodeClick calls TreeExpand to expand the parent node and load the children. Next, if the user has clicked a new node, TreeNodeClick calls SelectNode, which we'll look at in a moment, and saves the current Expanded state of the node and the time the node was clicked into custom properties. That's because TreeViews exhibit an annoying behavior: if you double-click on a node to take some action (for example, if the TreeView shows

table names and double-clicking should open the selected table in a BROWSE window), the TreeView automatically expands the node if it has any children. I don't like that behavior, so TreeDblClick, which is called from the TreeView's DblClick event, simply restores the Expanded state of the node saved by TreeNodeClick.

```
lparameters toNode
local loParent, ;
  loNode, ;
  lnSeconds
with This

* If you click lightly on the + for a parent node, sometimes Expand doesn't
* fire but the node still expands, so you see "Loading..." as the node.
* Since that isn't a valid node, if the user clicks it, we'll expand the
* parent node.

  if toNode.Text = ccLOADING
    loParent = toNode.Parent
    .TreeExpand(loParent, .T.)
    loNode = loParent.Child
  else
    loNode = toNode
  endif toNode.Text = ccLOADING

* If we're on the same node as before (likely because we were called from
* TreeMouseDown on the down-press and again from NodeClick when the mouse is
* released), do nothing. Otherwise, select the node.

  if isnull(.oTree.SelectedItem) or not loNode.Key == .cCurrentNodeKey
    .SelectNode(loNode)

* Save its Expanded property if this isn't the second click of a double-click.

    lnSeconds = seconds()
    if lnSeconds - .nNodeClick > _dblclick
      .lExpanded = loNode.Expanded
    endif lnSeconds - .nNodeClick > _dblclick
    .nNodeClick = seconds()
  endif isnull(.oTree.SelectedItem) ...
endwith
```

SelectNode has some interesting behavior. First, it can be called manually if you want to select a node programmatically for some reason. Normally, SelectNode expects to be passed a reference to the node being selected, but if you know the key or index value, you can pass that instead and SelectNode will figure out which node you want. Next, if SelectNode isn't passed .T. for the second parameter and lTrackNodeClicks is .T., it calls PushKey to push the key of the currently selected node onto a stack that can be used for "go back" behavior described later. SelectNode then ensures the node is visible (any parents are expanded and the TreeView is scrolled as necessary) and that only this node is selected (setting the Selected property of a node to .T. doesn't turn off the Selected property of any other node automatically). It then calls the GetTypeAndIDFromNode method to obtain an object whose properties specify the type of node selected and the ID of any underlying object (such as a record in a table), and sets the cCurrentNodeType and cCurrentNodeID properties to these values and sets cCurrentNodeKey to the Key property of the selected node. These properties can be used by any other methods so they can adapt their behavior based on the type of node selected. Finally, SelectNode calls the abstract NodeClicked method. Here's the code for SelectNode:

```
lparameters toNode, ;
  tlNoPush
local loNode, ;
  loObject
```

```
with This

* If we were passed a key or index rather than a node, try to find the proper
* node.

  do case
    case vartype(toNode) = 'O'
      loNode = toNode
    case type('.oTree.Nodes[toNode]') = 'O'
      loNode = .oTree.Nodes[toNode]
    otherwise
      loNode = .NULL.
  endcase
  if vartype(loNode) = 'O'

* Push the previous node onto the stack if we're supposed to.

    if not tlNoPush and .lTrackNodeClicks and ;
      vartype(.oTree.SelectedItem) = 'O'
      .PushKey(.oTree.SelectedItem.Key)
    endif not tlNoPush ...

* Ensure the node is visible and selected. Prevent two items from being
* selected by nulling the currently selected item before selecting this one.

    loNode.EnsureVisible()
    .oTree.SelectedItem = .NULL.
    loNode.Selected     = .T.

* Set cCurrentNodeType, cCurrentNodeID, and cCurrentNodeKey to the type, ID,
* and key of the selected node.

    loObject          = .GetTypeAndIDFromNode(loNode)
    .cCurrentNodeType = loObject.Type
    .cCurrentNodeID   = loObject.ID
    .cCurrentNodeKey  = loNode.Key

* Call the NodeClicked method for any custom behavior.

    .NodeClicked()
  endif vartype(loNode) = 'O'
endwith
```

Because you'll want additional behavior when a node is selected, such as updating some controls that display information about the selected node, put any necessary code in the NodeClicked method of a subclass or instance.

You must implement the GetTypeAndIDFromNode method in a subclass or instance of SFTreeViewContainer. This method must set the Type and ID properties of an Empty object returned by calling DODEFAULT() to the appropriate values for the selected node. For example, in the sample code I showed earlier, with the exception of the "Pages" and "Components" header nodes, the first letter of a node's Key property is the node type and the rest is the ID of the record the node represents. So, GetTypeAndIDFromNode just parses the Key property of the selected node to get the proper values.

```
lparameters toNode
local loObject, ;
  lcKey, ;
  lcType
loObject = dodefault(toNode)
lcKey    = toNode.Key
lcType   = left(lcKey, 1)
if inlist(lcType, 'P', 'C')
  loObject.Type = lcType
  loObject.ID   = val(substr(lcKey, 2))
else
  loObject.Type = lcKey
endif inlist(lcType ...
return loObject
```

Supporting "go back" behavior

A popular feature of browsers is the ability to go back to the previous page you viewed. I wish more applications had that behavior, so I decided to add support for it into SFTreeViewContainer. As we saw earlier, SelectNode calls PushKey, which adds the key for the selected node to the stack managed by the SFStack object. The GoBack method pops the last added key off the stack and calls SelectNode to select that node. However, it passes .T. for the second parameter so SelectNode doesn't re-push the key.

Handling TreeView coordinates

Several methods in SFTreeViewContainer call the TreeView's HitTest method to determine which node the mouse is over. Unfortunately for VFP developers, HitTest expects to be passed X and Y coordinates in twips, which are 1/1440th of an inch, rather than pixels. The conversion from twips to pixels isn't straightforward because it depends on the resolution of the user's monitor.

To handle this, the Init method calls CalcTwipsPerPixel, which calls several Windows API functions to determine the conversion factors (which could be different for X and Y values) and stores them in the nTreeFactorX and nTreeFactorY properties. All methods needing a reference to the node under the mouse call GetNodeUnderMouse, which has the following code:

```
lparameters tnXCoord, ;
  tnYCoord
local loNode
with This
  loNode = .oTree.HitTest(tnXCoord * .nTreeFactorX, ;
    tnYCoord * .nTreeFactorY)
endwith
return loNode
```

Supporting drag and drop

SFTreeViewContainer can be both a source and a target for OLE drag and drop operations. You may wish to drag one node to another, drag from somewhere else to the TreeView, or drag a node from the TreeView to somewhere else. The various OLE drag and drop events of the TreeView, such as OLEDragOver and OLEDragDrop, call methods of the container to do the actual work. These methods do whatever is necessary and call hook methods where you can customize the behavior. Because SFTreeViewContainer does all the work, you don't have to know much about how OLE drag and drop works; you simply code tasks like whether a drag operation can start and what happens when something is dropped on a node.

Here are the various places you can control the behavior of drag and drop operations:

- TreeMouseDown, called from the MouseDown event of the TreeView, calls the CanStartDrag method to determine if a drag operation can start. In SFTreeViewContainer, CanStartDrag always returns .F. so no drag operation occurs by default. You can put some code into the CanStartDrag method of a subclass that returns .T. if the selected node can be dragged.
- TreeOLEDragStart is called from the OLEDragStart event of the TreeView when a drag operation is started. I don't want to get into the mechanics of OLE drag and drop in this document (see the VFP Help topic for details), but TreeOLEDragStart calls the SetData

method of the OLE drag and drop data object passed into the method to store information about the node being dragged (the source object). It concatenates cCurrentNodeType, a colon, and cCurrentNodeID into a custom data type stored in the object, so any method that wants to determines the type and ID of the source node simply has to parse that out. It also stores the text of the selected node into the text data type of the object. This allows dragging a node to, for example, Notepad, to have the text of the node inserted into a file.

- TreeOLEDragOver, called from the OLEDragOver event of the TreeView when something is dragged over it, highlights the node under the mouse (you'd think the TreeView would do this automatically, but unfortunately not), scrolls the TreeView up or down if the mouse pointer is close to the top or bottom edges of the TreeView (again, behavior you'd think would be automatic). It then calls GetDragDropDataObject to get an object with information about both the source object and the node under the mouse (we'll look at this method in a moment), then calls the abstract CanDrop method to determine if the current node can accept a drop from the source object. In SFTreeViewContainer, CanDrop always returns .F. so nothing can be dropped on the TreeView by default, but in a subclass, you'll likely examine the properties of the object returned by GetDragDropDataObject to see if the source object can be dropped on the node or not.
- TreeOLEDragDrop, called from the OLEDragDrop event of the TreeView when something is dropped on it, sets the DropHighlight property of the TreeView to .NULL. to remove highlighting, calls GetDragDropDataObject to get an object with information about both the source object and the node under the mouse, and passes that object among other things to the abstract HandleDragDrop method. In a subclass, you'll code HandleDragDrop to determine what happens.
- TreeOLECompleteDrag, called from the OLECompleteDrag event of the TreeView once the drag and drop operation is complete (whether the target was successfully dropped on a node or not), doesn't call a hook method but does turn off node highlighting that was turned on from TreeOLEDragOver (again, you'd think this would be automatic).

GetDragDropDataObject is used to fill an Empty object with properties about the source object and the node under the mouse. This object has seven properties (and possibly an eighth as we'll see in a moment): DragType and DropType, which contain the type of source and target objects; DragKey and DropKey, which contain the ID values for the source and target objects; Button and Shift, which indicate which mouse button was pressed and the status of the Shift, Alt, and Ctrl keys; and Data, which contains the data from the OLE drag and drop data object, depending on what type of data is stored. For example, if you're dragging one node in the TreeView to another, Data contains the concatenated cCurrentNodeType, a colon, and cCurrentNodeID from the custom data type stored in the object, and DragType and DragKey contain the node type and ID from the source node, parsed from Data. If the object contains only text, such as when you drag text from Notepad to the TreeView, Data contains that text and DragType contains "Text." If you drag one or more files from Windows Explorer to the TreeView, Data is blank, DragType contains "ImportFiles," and a new eighth property, a collection of the file names, is added to the Empty object.

```
lparameters toNode, ;
  toData, ;
```

```
  tnButton, ;
  tnShift
local loObject, ;
  loDataObject, ;
  lcData, ;
  lnPos
loObject = createobject('Empty')
addproperty(loObject, 'Data',     '')
addproperty(loObject, 'DropType', '')
addproperty(loObject, 'DropKey',  '')
addproperty(loObject, 'DragType', '')
addproperty(loObject, 'DragKey',  '')
addproperty(loObject, 'Button',   tnButton)
addproperty(loObject, 'Shift',    tnShift)
with loObject
  if vartype(toNode) = 'O'
    loDataObject = This.GetTypeAndIDFromNode(toNode)
    .DropType    = loDataObject.Type
    .DropKey     = loDataObject.ID
  endif vartype(toNode) = 'O'
  try
    do case
      case toData.GetFormat(CF_MAX)
        lcData    = toData.GetData(CF_MAX)
        lnPos     = at(':', lcData)
        .Data     = lcData
        .DragType = '' + left(lcData, lnPos - 1)
        .DragKey  = '' + substr(lcData, lnPos + 1)
      case toData.GetFormat(CF_TEXT)
        lcData    = toData.GetData(CF_TEXT)
        .Data     = lcData
        .DragType = 'Text'
      case toData.GetFormat(CF_FILES)
        addproperty(loObject, 'Files', createobject('Collection'))
        for each lcFile in toData.Files
          loObject.Files.Add(lcFile)
        next lcFile
        .DragType = 'ImportFiles'
    endcase
  catch
  endtry
endwith
return loObject
```

Several methods use the Empty object returned by GetDragDropDataObject, including CanDrop and HandleDragDrop, so they can determine what to do based on the source and target objects. The default behavior of GetDragDropDataObject is to fill DropType and DropKey by calling the GetTypeAndIDFromNode method discussed earlier to get the type and ID for the node under the mouse. You can override this in a subclass if, for example, you need numeric rather than character key values. For example:

```
lparameters toNode, ;
  toData
local loObject
loObject         = dodefault(toNode, toData)
loObject.DragKey = val(loObject.DragKey)
return loObject
```

Whew! That's a lot of behavior and code. Fortunately, to support drag and drop behavior, you just need to implement the following methods: CanStartDrag to determine if the current node can be dragged, CanDrop to determine if the node under the mouse will accept a drop from the source object, and HandleDragDrop to perform the necessary tasks when the source object is dropped on a node. You may also need to override GetDragDropDataObject and TreeOLECompleteDrag, depending on your needs.

Supporting other behavior

TreeView controls have other behavior that SFTreeViewContainer supports and allows you to customize.

- If the LabelEdit property of the TreeView is set to 0, which it is in SFTreeViewContainer, the user can change the Text property of a node by clicking on it and typing the new text. However, you may not always want that to happen and you'll certainly want to be notified once the user has finished typing so you can save the change in the source data. The BeforeLabelEdit event of the TreeView, which fires just before the user can begin typing, calls the TreeBeforeLabelEdit method of the container, passing it a "cancel" parameter by reference. That method sets the parameter to .T. if the lAllowRename property of SFTreeViewContainer is .F., which it is by default. So, to allow the user to edit the text of a node, set lAllowRename to .T. You may wish to do this conditionally so some nodes can be edited and not others. The AfterLabelEdit event of the TreeView, fired when the user is done making changes, calls the abstract TreeAfterLabelEdit method. In a subclass, implement whatever behavior you wish in this method.
- As I mentioned earlier, the DblClick event of the TreeView calls TreeDblClick. If you want something to happen when the user double-clicks on a node, put code into this method in a subclass. Don't forget to use DODEFAULT() so the node's Expanded state isn't affected (or omit that if you want double-clicking to expand a node).
- In addition to handling dragging operations, TreeMouseDown also handles a right-click on a node because the TreeView doesn't have a RightClick event. When the user right-clicks a node, TreeMouseDown calls ShowMenu (which is actually defined in the parent SFContainer) to display a shortcut menu if one is defined. ShowMenu uses the SFMenu class to create a shortcut menu and calls ShortcutMenu to populate that menu. ShortcutMenu is an abstract method so there is no shortcut menu by default; fill in that method of a subclass if you wish.
- If you want to the user to be able to press the Delete key to remove the selected node, set the lAllowDelete property to .T. and fill in code in the DeleteNode method. The TreeView's KeyDown method calls the TreeKeyDown method, which calls DeleteNode method if the Delete key was pressed and lAllowDelete is .T. In SFTreeViewContainer, DeleteNode is abstract so nothing happens, but in a subclass you can call RemoveNode to remove the selected node from the TreeView. You'll likely want to take other action as well, such as deleting a record from a table. You can even make this behavior dynamic by setting lAllowDelete to .T. only for nodes that can be deleted.
- If you want to the user to be able to press the Insert key to add a node, set the lAllowInsert property to .T. and fill in code in the InsertNode method; this is handled the same way the Delete key is. InsertNode is abstract in SFTreeViewContainer, but in a subclass, you can implement any behavior you wish.
- If you set the Checkboxes property of the TreeView control in a subclass to .T., each node will have a checkbox in front of it. The NodeCheck event is fired when the user checks or

unchecks a node. This event calls TreeNodeCheck, which is abstract in SFTreeViewContainer. Fill in whatever code is necessary in this method in a subclass.

- SFTreeViewContainer has a SelectedNode property that contains a reference to the TreeView's SelectedItem property so you don't need to reference the TreeView in code external to the container.