The Mother of All TreeViews, Part 2

Doug Hennig

Last month, Doug presented a reusable class that encapsulates most of the desired behavior for a TreeView control. He discussed controlling the appearance of the TreeView, loading the nodes in the TreeView, and restoring the TreeView's state. This month, he finishes the discussion of this class and shows a form class that provides the features needed in any form that displays a TreeView and the properties of the selected node.

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 just does a couple of things: it calls SelectNode, which we'll look at in a moment, and it 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.

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 value, you can pass that instead and SelectNode will figure out which node you want. Next, it 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. The 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
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
      return .F.
  endcase

* 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 and cCurrentNodeID to the type and
* ID of the selected node.

  loObject          = .GetTypeAndIDFromNode(loNode)
  .cCurrentNodeType = loObject.Type
  .cCurrentNodeID   = loObject.ID
```

```
* Call the NodeClicked method for any custom behavior.

  .NodeClicked()
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. In WebEditor.SCX, NodeClicked locates the appropriate record in the appropriate table (based on cCurrentNodeType and cCurrentNodeID), sets lAllowDelete to indicate if the node can be deleted (I'll discuss this property later), and calls Thisform.Refresh so all the controls on the form can refresh themselves.

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 WebEditor.SCX, 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, ccPAGE_KEY, ccCOMPONENT_KEY, ;
  ccPAGE_COMPONENT_KEY)
  loObject.Type = lcType
  loObject.ID   = val(substr(lcKey, 2))
else
  loObject.Type = lcKey
endif inlist(lcType ...
return loObject
```

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. In WebEditor.SCX, CanStartDrag returns .F. if a "header" node (the "Pages" and "Components" nodes) is selected or .T. for any other node.

- TreeOLEDragStart, called from the OLEDragStart event of the TreeView when a drag operation is started, calls the StartDrag method, passing it an OLE drag and drop data object. I don't want to get into the mechanics of OLE drag and drop in this article (see the VFP Help topic for details), but StartDrag calls the SetData method of the data object to store some information about the node being dragged (the source object). It concatenates cCurrentNodeType, a colon, and cCurrentNodeID, so any method that wants to determines the type and ID of the source node simply has to parse that out. If you want different information, override StartDrag in a subclass.

- 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 that has information about both the source object and 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. WebEditorTreeView.CanDrop returns .T. if a component node is dragged to a page node or a container node within a page (this will add the component to the page or container), if a page content node is dragged to another page content node (this will rearrange the order of the content on the page), or if text is dragged to the "Components" node (this will create a new component with the text as the content).
- TreeOLEDragDrop, called from the OLEDragDrop event of the TreeView when something is dropped on it, calls the abstract HandleDragDrop method. In a subclass, you'll code this method to determine what happens. For example, WebEditor.SCX handles the cases mentioned in the previous point by updating the appropriate table as necessary and calling LoadTree to reload the TreeView with the updated information.
- 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 four properties: DragType and DropType, which contain the type of source and target objects, and DragKey and DropKey, which contain the ID values for the source and target objects. This method is called from TreeOLEDragOver, which passes the resulting object to CanDrop, and TreeOLEDragDrop, which passes the object to HandleDragDrop, so these methods 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, and to fill DragType and DragKey by parsing the data in the OLE data object. You can override this in a subclass if, for example, you need numeric rather than character key values. That's exactly what the code in this method in WebEditor.SCX does:

```
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. To prevent the user from editing the current node, set the parameter's value to .T. in a subclass; this is what TreeBeforeLabelEdit does in SFTreeViewContainer, so editing is disabled by default. The AfterLabelEdit event of the TreeView, fired when the user is done making changes, calls the abstract TreeAfterLabelEdit method. In a subclass, implement what ever 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 by calling the ShowMenu method of the container to display a shortcut menu if one is defined. I discussed using shortcut menus in my February 1999 column ("A Last Look at the FFC"). SFTreeViewContainer doesn't implement a shortcut menu; fill in the ShortcutMenu method (which is called from ShowMenu) 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 RemoveNode method. The TreeView's KeyDown method calls the TreeKeyDown method, which calls the RemoveNode method if the Delete key was pressed and lAllowDelete is .T. In SFTreeViewContainer, RemoveNode simply removes the selected node from the TreeView. However, in a subclass, you'll likely want to take other action, such as deleting a record from a table, in this method as well. You can even make this behavior dynamic by setting lAllowDelete to .T. only for nodes that can be deleted. For example, the NodeClicked method in WebEditor.SCX sets lAllowDelete to .F. for the "Pages" and "Components" header nodes because we don't want the user to delete those, and sets it to .T. for all other nodes. RemoveNode deletes the appropriate record or records from the proper tables.

- 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 that the Delete key is. InsertNode is abstract in SFTreeViewContainer, but in a subclass, you can implement any behavior you wish. In WebEditor.SCX, InsertNode adds a new record to the appropriate table, then calls LoadTree to reload the TreeView.

- 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 NodeChecked event is fired when the user checks or unchecks a node. This event calls TreeNodeChecked, which is abstract in SFTreeViewContainer. Fill in whatever code is necessary in this method in a subclass.

SFTreeViewForm

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 SFTreeViewForm is for. Figure 1 shows this class in the Class Designer. As you can see, it contains more than just an SFTreeViewContainer and a pageframe for properties; it also includes:

- a splitter control (the rectangular shape between the SFTreeViewContainer and pageframe; I discussed this control in my July 1999 column, "Splitting Up is Hard to Do") to adjust the relative sizes of the TreeView and pageframe;

- a status bar control (I used Rick Strahl's wwStatusBar control rather than the ActiveX version because it looks better in Windows XP; see http://www.west-wind.com/presentations/wwstatusbar/wwstatusbar.asp for an article and the source code) to display information about the state of the form, such as whether it's ready or currently loading the TreeView;

- a timer to ensure the TreeView is redrawn properly when the form is resized at startup by calling its DoVerb method (I have no idea why this is necessary, but it seems to do the trick); and
- an object to persist the form size and position (see my January 2000 article, “Persistence without Perspiration”, for details on this object).

It also supports a toolbar if the cToolBarClass and cToolBarLibrary properties are filled in, and resizing controls when the form is resized (see my June 2003 column, “Ahoy! Anchoring Made Easy”, for details).

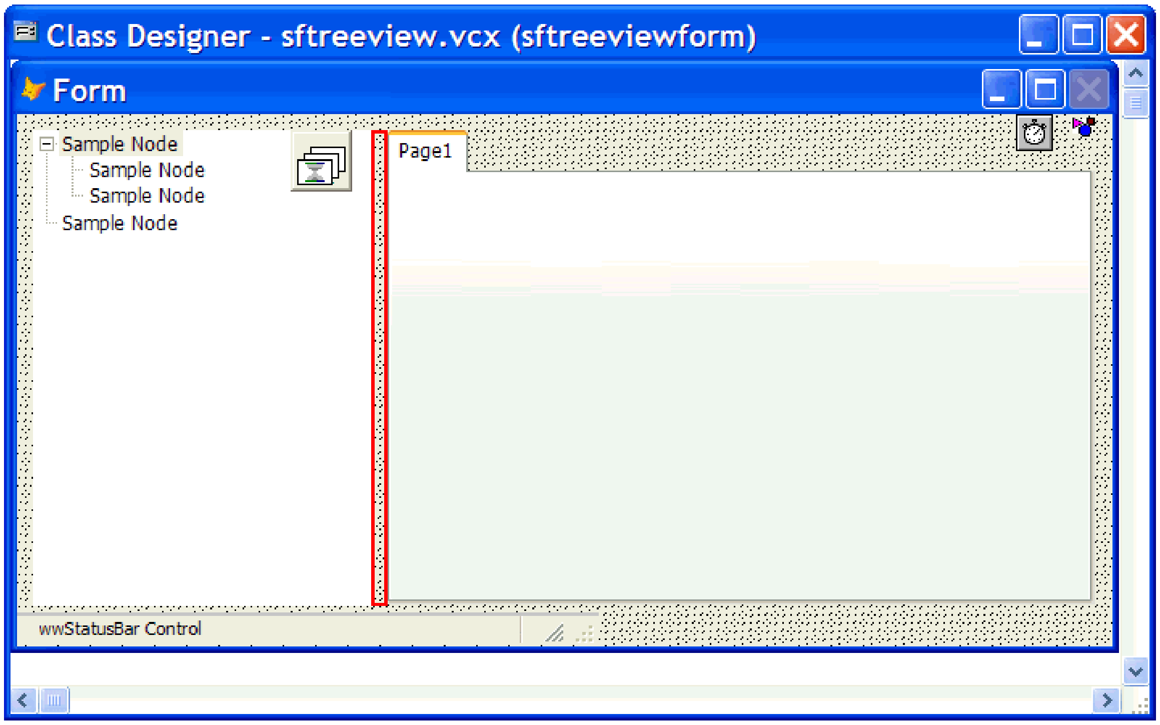
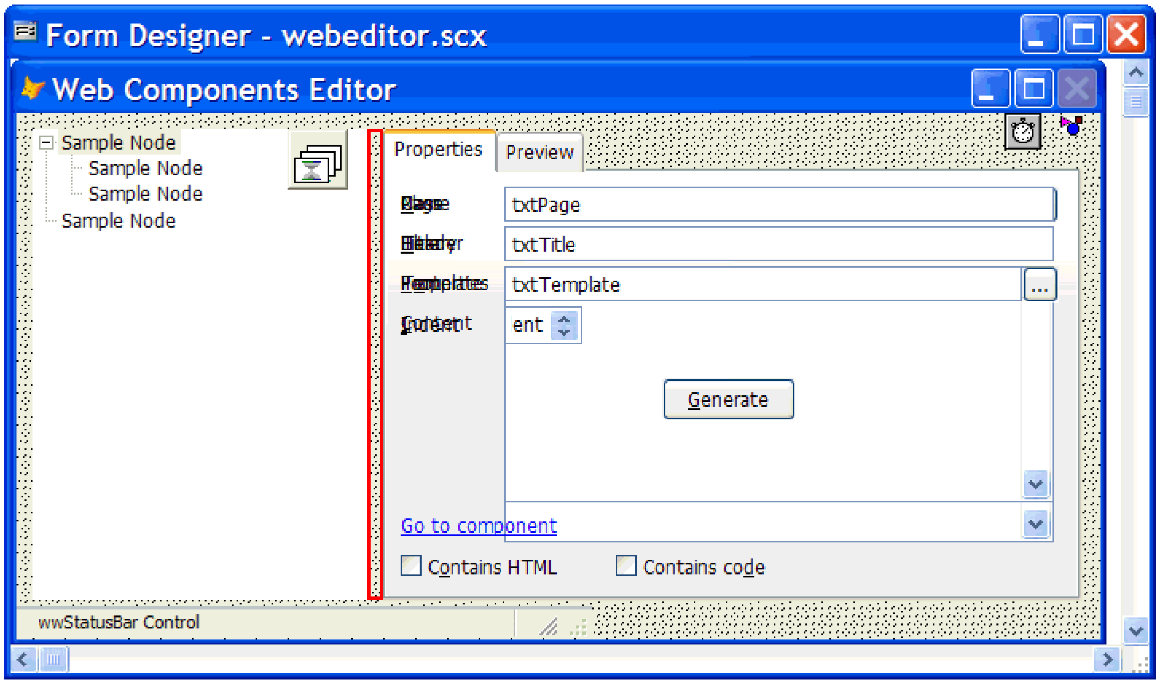


Figure 1. SFTreeViewForm provides all of the core features of a TreeView-based form.

There isn’t a lot of code in this form; most of it is associated with setting up the persistence object, splitter, status bar, and toolbar.

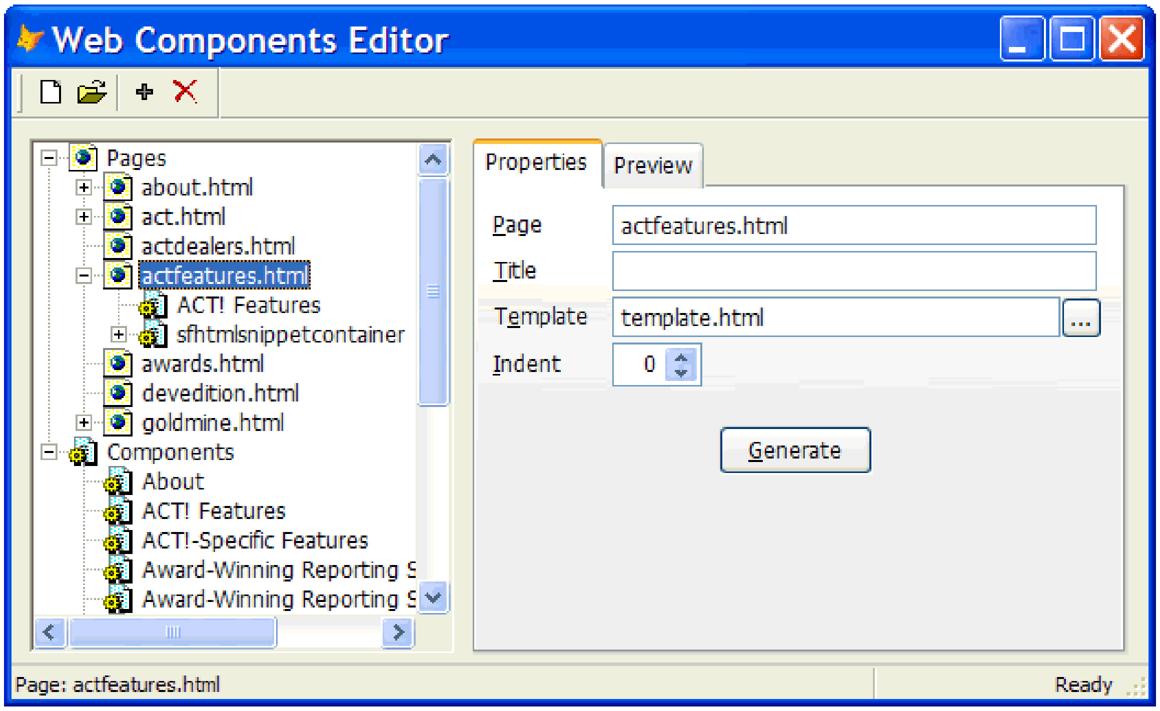
When a node in the TreeView is selected, how do you display the properties for that node? I created a subclass of SFContainer called SFPropertiesContainer. This container simply has code in Refresh that sets the Visible property to .T. only if the custom cNodeType property matches the value of Thisform.oTreeViewContainer.cCurrentNodeType, which, as we saw earlier, is set by the SelectNode method when a node is selected. When the user selects a node and the form is refreshed, any SFPropertiesContainer object on the form will become visible only if it supports the type of node selected.

To create controls that display properties for the selected node, create one subclass of SFPropertiesContainer for every type of node you’ll use, fill it with the appropriate controls, and set cNodeType to the node type the container is used for. Then drop each of these subclasses on the first page of the pageframe in an instance of SFTreeViewForm and ensure they overlap. When the user selects a node and the form is refreshed, only one of the properties containers will be visible, so the properties for the selected node will be displayed. In Figure 2, you can see that WebEditor.SCX, an instance of SFTreeViewForm, looks kind of messy at design time because of the overlapping controls, but, as Figure 3 shows, only the proper properties container is displayed at runtime.



txtPage
txtTitle
txtTemplate
ent

Figure 2. WebEditor.SCX looks messy in the Form Designer because of overlapping properties containers.



Page: actfeatures.html
Title:
Template: template.html
Indent: 0

Page: actfeatures.html

Figure 3. Only the proper properties container is visible at runtime.

WebEditor.SCX

We've been discussing the features and code in WebEditor.SCX as we went along, but let's finish by discussing some additional things this form supports.

First, it supports multiple Web sites; the toolbar (instantiated from WebEditorToolBar in WebEditor.VCX) has New and Open buttons to create or open the Web component tables (PAGES.DBF, CONTENT.DBF, and PAGECONTENT.DBF) in a particular directory. The Init method restores the last selected directory (saved when the form was closed the last time it was used) from Settings.INI, which is also used to persist the form's size and position. The SaveSelectedNode and RestoreSelectedNode methods of the SFTreeViewContainer object in this form save and restore the selected and expanded nodes to another INI file, also called Settings, that's located in the Web site directory (we discussed those methods last month). Thus, running WebEditor.SCX automatically restores the form's size and position, opens the last used Web site, and restores the expanded and selected status of each node. In other words, it comes up exactly as it was when it was closed.

Second, the pageframe has two pages: one for the properties for the selected node and one that contains an instance of PreviewControl (contained in WebEditor.VCX) to preview the HTML for the node using a Web browser control. PreviewControl simply instantiates the appropriate class from SFHTML.VCX (which we discussed two months ago in my article titled "Web Page Components"), sets its properties, call its Render method to generate the HTML, and displays the HTML in the Web browser control. When you're on a page node, you can also generate an HTML file for the page by clicking on the Generate button in page 1 of the pageframe.

Summary

When I started working on an editor for the Web components tables, I was planning on just whipping up a quick and dirty form to handle the tasks. However, I'm really glad I took to time to create the classes discussed in the past two articles because I now have some reusable classes I can use to create any type of similar form. I hope you also find uses for these classes.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, author of the CursorAdapter and DataEnvironment builders that come with VFP 8, and co-author of "What's New in Visual FoxPro 8.0", "What's New in Visual FoxPro 7.0", and "The Hacker's Guide to Visual FoxPro 7.0", from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com