COM Corner: Shell Namespace Extensions

by Steve Teixeira

We're all developers here, so I'm going to be straight with you. Sometimes COM is just too hard. Take namespace extensions: they provide a nifty way for developers to make their applications look and feel as if they are an integrated part of Windows Explorer. However, the price of entry for this novelty is that you must implement a few dozen methods using a handful of vaguely documented interfaces across multiple COM objects. As a consultant, I work with clients on the design and architecture of their systems. Many times, I recommend COM as a framework for a software system, only to be told 'we thought about using COM, but it's too complex.' Maybe I can help with this by illuminating the trail when the going gets rough. Which brings us back to namespace extensions...

What? Why?

So, what is a namespace and why would I want to extend it? Generically, a namespace is a collection of symbols within some entity. For example, a Pascal unit implies a namespace comprising all the types, constants and variables declared in it. Namespaces can be nested: for example a procedure can exist globally in a unit, whereas a local variable of that procedure could be said to exist within the namespace of that procedure. The Explorer shell uses the concept of namespaces to organize all the objects that live in the shell. For example, the *Desktop* contains *My Computer*, which in turn contains things like your hard disk and dial-up networking. Microsoft refers to this hierarchical collection of files, devices, and other objects as the *shell namespace*.

All those drives and devices are useful, but what happens when you want to write code that displays your own view of some information as if it were a part of the shell? The answer is *namespace extensions*. They make it possible for you to extend the shell by writing a special COM server that manages the details of integrating your data within the shell's hierarchy. It is your namespace extension's job to provide Explorer with the captions, icons and details of the items in your namespace. Namespace extensions can also provide extra functionality, like drag-and-drop support or context menus.

Terminology

Before we can jump into the details we must clear up a few points of terminology. To accomplish this hierarchical arrangement, the Explorer uses folders and file objects: analogous to directories and files in a file system. An example of a folder is *My Documents*, which contains file objects representing the physical files in that folder. It's important to note, however, that file objects don't have to be files. For example, the *Dial-Up Networking* folder contained within *My Computer* contains dial-up connection objects that do not necessarily map to disk files. Within the shell, each file object is represented by an identifier, unique within its parent folder's namespace, called an *item identifier*. An *item identifier list* uniquely identifies a file object within the entire shell namespace because it contains the entire path name of the object. Most often, pointers to item identifier lists are passed around within the shell to identify file objects, and they are known as pidls (pronounced 'piddles').

Roots

You have two options for integrating your namespace extension with the shell: *rooted* or *nonrooted*. Nonrooted extensions are most common, as they enable users to browse into your namespace using Explorer. In this case, the *Desktop* serves as the root for your namespace extension, which is nested within the desktop's namespace. Rooted namespace extensions imply a completely separate namespace, and you must run a new instance of Explorer to browse the namespace extension. From a code perspective, there's little difference between rooted and nonrooted. The motivating factor for choosing one or the other is how you want your extension to interact with the end-user and what you feel makes sense for your purposes.

Conjunction Junction

Namespace extensions must have some sort of jump-off point that defines when the user leaves the warmth and security of the default shell and enters into the scary world of the custom view provided by the namespace extension. This jump-off point is referred to as a *junction point*. For example, the junction point for the *Recycle Bin* is on the *Desktop* and the junction point for *Dial-Up Networking* is in *My Computer*.

Structure

Namespace extensions require the implementation of a number of COM interfaces. First and foremost is `IShellFolder`. This interface represents the folder object for your namespace extension: the shell will use it to obtain information on items in your namespace and to obtain pointers to other interfaces in your extension. Table 1 lists the methods of `IShellFolder`.

You will notice that the method `CreateViewObject` is intended to return an `IShellView` interface pointer for this `IShellFolder`. `IShellView` handles presentation of a view of your namespace within Explorer. `IShellView` contains the methods described in Table 2.

In addition to these interfaces, the namespace extension must also implement `IPersistFolder` and `IEnumIDList`. To provide more interactive functionality for your

extension, you can also provide implementations for: IExtractIcon (to provide items in your view with icons), IContextMenu (to support context menus on your view's items), IDataObject (for data transfer) plus IDropSource and IDropTarget (for drag-and-drop).

Implementation

I spend a lot of time messing around with various COM servers on my machine, registering and unregistering them, so a real handy tool for me would be a namespace extension that enabled me to browse all these COM servers. A good junction point for this extension is *My Computer*.

The first thing you need to do is of course create a new COM server application. You can do this by first creating a new ActiveX server and then adding a new COM object to it using the COM object wizard. I called my server ComNameExt. With that out of the way, get ready to implement lots of interfaces. Actually, this is probably less than you might think because there are only a few key methods that need to be implemented to create a namespace extension. So, we'll be making extensive use of the E_NOTIMPL return value. Keep in mind that, since there are several hundred lines of code associated with the complete implementation of this COM server, I can't practically discuss every part of it here, but I'll certainly hit the highlights. The disk contains the complete source code for the COM server.

While all this interface stuff is well and good, let's not forget that the brains of this extension is the bit of code that runs through the registry to find all the COM servers registered. Fortunately, this is a straightforward task since all the COM servers live in the HKEY_CLASSES_ROOT\CLSID area of the registry and Delphi's TRegistry class makes working with the registry easy. The code in Listing 1 grabs the COM servers and adds them to a list object called ServList.

I mentioned pidls a bit earlier. The nice thing about item IDs is that they are user-defined. The first word of the item ID record must be the size of the record, and the rest of the record holds whatever data the implementer sees fit. In this case, I store the CLSID for each COM server. The record representing my item ID is called TServInfo (see the code on the disk).

One of the first methods the shell calls after loading your namespace extension is IShellFolder.CreateViewWindow. You'll recall this is the method responsible for creating and returning a new IShellView. One important gotcha here is that CreateViewObject can be called many times on one IShellFolder to create multiple distinct views, so the implementation for IShellView must exist in a separate object from that which implements IShellFolder. See CreateViewObject on the disk.

While IShellFolder is the heart of a namespace extension, IShellView is the brains. The methods of this interface will serve as the consumer for several of the methods you implement in IShellFolder. The first IShellView method to be called is CreateViewWindow. In this method, I create an instance of a Delphi form that contains a TListView and parent that form into the owner window passed in the hwndOwner parameter of TViewObject.Create. Delphi has a clear advantage over other tools here. With a C++ tool, we'd have to deal with message processing and other housekeeping for windows and controls created in the view.

The other key method is Refresh, where the view items are enumerated, listview items are created, captions are obtained and icons generated with the help of an IExtractIcon from IShellFolder.GetUIObjectOf. See Listing 2.

One of the first things Refresh does is obtain an IEnumIDList enumerator using IShellFolder.EnumObjects. The IEnumIDList methods for the namespace extension are implemented by the ServList object and aggregated into IComNameExt using implements:

```
property ServList: TComServerList
  read FServList write FServList
  implements IEnumIDList;
```

Method	Description
ParseDisplayName	Translate the display name of an item into an item ID list.
EnumObjects	Retrieves the IEnumIDList used to enumerate items in folder.
BindToObject	Retrieves the IShellFolder for the specified folder.
BindToStorage	Returns the storage instance of a subfolder.
CompareIDs	Compare two file or folder objects based on their item ID lists.
CreateViewObject	Creates and returns a new IShellView for this folder.
GetAttributesOf	Retrieves the attributes of the specified file or folder.
GetUIObjectOf	Obtains an IExtractIcon, IContextMenu, IDataObject, IDropSource, or IDropTarget for the specified items.
GetDisplayNameOf	Retrieves the display name of the specified item.
SetNameOf	Sets the display name of the specified file or subfolder.

➤ *Table 1*

Method	Description
TranslateAccelerator	Enables the view to process keystrokes before the shell acts on them.
EnableModeless	Enables or disables modeless dialog boxes.
UIActivate	Notifies view of focus and activation changes.
Refresh	Refreshes display in response to user input, eg when F5 is pressed.
CreateViewWindow	Creates window containing the view, usually one containing a listview.
DestroyViewWindow	Destroys the window created in the above method.
GetCurrentInfo	Retrieves the current settings for the folder view.
AddPropertySheetPages	Allows the view to add pages to the Options property sheet.
SaveViewState	Provides view with the opportunity to persistently store state settings.
SelectItem	Changes the selection state of items within the view.
GetItemObject	Retrieves an interface for data presented in the view.

➤ *Table 2*

```
procedure TComNameExt.RefreshServerList;
var
  Reg: TRegistry;
  I, KeyIdx: Integer;
  ClsidKeys, SubKeys: TStringList;
  CurrentKey: string;
begin
  Reg := TRegistry.Create;
  ClsidKeys := TStringList.Create;
  try
    ServList.Clear;
    Reg.RootKey := HKEY_CLASSES_ROOT;
    if not Reg.OpenKeyReadOnly('CLSID') then
      raise Exception.Create('Failed to open registry');
    Reg.GetKeyNames(ClsidKeys);
    Reg.CloseKey;
    SubKeys := TStringList.Create;
    try
      for I := 0 to ClsidKeys.Count - 1 do begin
        CurrentKey := ClsidKeys[I];
        if Reg.OpenKeyReadOnly('CLSID\'+CurrentKey) then begin
          Reg.GetKeyNames(SubKeys);
          Reg.CloseKey;
          KeyIdx := SubKeys.IndexOf('InprocServer32');
          if KeyIdx < 0 then KeyIdx := SubKeys.IndexOf('LocalServer32');
          if KeyIdx < 0 then Continue;
          ServList.AddGuid(StringToGUID(CurrentKey));
        end;
      end;
    finally
      SubKeys.Free;
    end;
  finally
    Reg.Free;
    ClsidKeys.Free;
  end;
end;
```

➤ *Listing 1*

`ServList` is of type `TComServerList`, and descends from `TList`. The method `IEnumList.Next` fills an array of `PItemIDLists` with the requested number of items.

`Refresh` also calls `IShellFolder.GetDisplayNameOf` to get a display string for the COM server: it simply looks to the registry for the name of the COM server (Listing 3).

Registration

I like COM servers to be completely self-contained, so registration of my namespace extension is handled in the `DllRegisterServer` and `DllUnregisterServer` exports for the COM server DLL.

This is easily accomplished by creating a special `TComServerFactory` descendent of the `TComNameExt` class. Most notably, the registration for this server must establish the junction point of the namespace extension, add it to the 'approved' extensions under Windows NT, plus establish attributes and a default icon for the extension. The code for the `UpdateRegistry` method of my custom factory object is shown in Listing 4.

Test Drive

Once you're ready to try out the namespace extension, it needs to be registered, either from the `Run` menu in the Delphi 4 IDE, or from the command line using regsvr32.exe or tregsvr.exe. Then just open *My Computer* in the shell to see the results.

Debugging Shell Extensions

Now that you know all about writing a namespace extension, you might be thinking, 'well, that's all jim-dandy, but my code usually doesn't run perfectly the first time, how do I debug this thing?' Great question, I'm glad you asked. Because shell extensions execute from within the shell's own process, how is it possible to 'hook into' the shell in order to debug your shell extension?

The solution to the problem is based on the fact that the shell is an executable (not very different than any other application) called explorer.exe. This has a property, however, that is kind of unique: the first instance of explorer.exe will invoke the shell. Subsequent instances will simply invoke additional Explorer windows in the shell.

Using a little known trick in the shell, it's possible to close the shell without closing Windows. Follow these steps to debug your shell extensions in Delphi.

➤ *Listing 2*

```
function TViewObject.Refresh: HResult;
var
  EnumObj: IEnumIDList;
  Folder: IShellFolder;
  IconOMatic: IExtractIcon;
  Fetched: ULONG;
  ItemID: array[1..100] of PItemIDList;
  Str: TStrRet;
  ListItem: TListItem;
  I, IconIndex: Integer;
  IconIdx: Word;
  IconFile: array[0..MAX_PATH] of char;
  Icon: HICON;
begin
  Result := S_OK;
  try
    FControlExt.RefreshServerList;
    with FViewForm.ListView do begin
      for I := 0 to Items.Count - 1 do
        if Items[I].Data <> nil then
          FControlExt.ShellMalloc.Free(Items[I].Data);
      Items.Clear;
    end;
    FViewForm.ImageList.Clear;
    Folder := FControlExt as IShellFolder;
    Folder.EnumObjects(FOwner, SHCONTF_FOLDERS or
      SHCONTF_NONFOLDERS or SHCONTF_INCLUDEHIDDEN, EnumObj);
    while EnumObj.Next(100, ItemID[1], Fetched) = S_OK do
      for I := 1 to Fetched do begin
        ListItem := FViewForm.ListView.Items.Add;
        ListItem.Data := ItemID[I];
        OleCheck(Folder.GetDisplayNameOf(ItemID[I],0,Str));
        case Str.uType of
          STRRET_WSTR :
            begin
              ListItem.Caption :=
                WideCharToString(Str.pOleStr);
              FControlExt.ShellMalloc.Free(Str.pOleStr);
            end;
          STRRET_CSTR : ListItem.Caption := Str.cStr;
        end;
        if Folder.GetUIObjectOf(FOwner, 1, ItemID[I],
          IExtractIcon, nil, Pointer(IconOMatic))
          = S_OK then begin
          IconOMatic.GetIconLocation(GIL_FORSHELL, IconFile,
            SizeOf(IconFile), IconIndex, Fetched);
          if Fetched and GIL_NOTFILENAME = 0 then begin
            IconIdx := IconIndex;
            Icon := ExtractAssociatedIcon(MainInstance,
              IconFile, IconIdx);
            if Icon <> 0 then
              ListItem.ImageIndex := ImageList_AddIcon(
                FViewForm.ImageList.Handle, Icon);
          end;
        end;
      end;
  except
    on E: TObject do
      Result := Controller.SafeCallException(E, ExceptAddr);
  end;
end;
```

First make explorer.exe the host application for your shell extension in the Run | Parameters dialog. Be sure to include the full path (eg c:\windows\explorer.exe).

Then, from the shell's Start menu, select Shut Down. This will invoke the Shut Down Windows dialog.

Thirdly, in the Shut Down Windows dialog, hold down Ctrl+Alt+Shift and click the No button. This will close the shell without closing Windows.

Next, using Alt+Tab, switch back to Delphi and run the shell extension. This will invoke a new copy of the shell running under the Delphi debugger. You can now set breakpoints in your code and debug as usual.

Lastly, when you are ready to close Windows, you can still do so properly without using the shell. Under Win95 or NT, use Ctrl+Esc to invoke the Tasks window and then select Windows | Shutdown Windows to close Windows. Under Win98, use Ctrl-Alt-Del to invoke the Close Program dialog, and click the Shut Down button.

More And More Features

There's lots of functionality under the umbrella of namespace extensions, so there's certainly room to add more features to this demo. For example, drag-and-drop could be supported: you'd need to implement the IDragTarget, IDropTarget, and IDataObject interfaces, and return them via the IShellFolder.GetUIObjectOf method. You would also need to add to the SFGAO_* flags found in the registration code and in the IShellFolder.GetAttributesOf method. One other feature of namespace extensions that this example didn't demonstrate was adding main menu items and buttons to the Explorer window. This can be done by manipulating the shell's IShellBrowser interface, which is passed to the IShellView.CreateViewWindow method. There's a fine line between a demo you can learn from and a functional piece of software, and I wanted to avoid bogging the implementation down with all the little details. Maybe I'll add some features to this namespace extension in a future article.

Summary

Like I said, sometimes COM is just too hard. While I'm the first one to state that namespace extensions can be a royal pain to implement, I also firmly believe that the payoff can be well worth the pain. There's nothing like integrating your application so that it looks and feels like a natural part of the Windows Explorer. This discussion of namespace extensions took you through the terms, concepts, and structure of namespace extensions, and then into the specifics of a Delphi implementation.

For further reading on this topic, I recommend the MSDN library CD.

Hopefully namespace extensions are now one area of COM that just moved from hard to doable on your difficulty scale, and you're ready to tackle an implementation on your own.

Steve Teixeira is the Director of Software Development at DeVries Data Systems, specializing in Inprise tools, and co-author of *Delphi 4 Developer's Guide*. Let Steve know how you like COM Corner and what you'd like to see in future editions by emailing him at steve@dvdata.com

```
function TComNameExt.GetDisplayNameOf(pidl: PItemIDList; uFlags: DWORD;
  var lpName: TStrRet): HResult;
var
  Guid: TGUID;
  NameStr, GuidStr: string;
  Reg: TRegistry;
begin
  Result := S_OK;
  try
    FillChar(lpName, SizeOf(lpName), 0);
    Guid := PServInfo(pidl)^.CLSID;
    GuidStr := GuidToString(Guid);
    lpName.uType := STRRET_CSTR;
    if HiWord(uFlags) and (SHGDN_FORPARSING) = 0 then begin
      Reg := TRegistry.Create;
      Reg.RootKey := HKEY_CLASSES_ROOT;
      Reg.OpenKeyReadOnly('CLSID\' + GuidStr);
      NameStr := Reg.ReadString('');
    end;
    if NameStr = '' then
      NameStr := GuidStr;
    StrLCopy(lpName.cStr, PChar(NameStr), SizeOf(lpName.cStr));
  except
    on E: TObject do
      Result := SafeCallException(E, ExceptAddr);
  end;
end;
```

➤ *Listing 3*

```
procedure TNamespaceExtensionFactory.UpdateRegistry(Register: Boolean);
const
  NamespaceKey = 'SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\' +
    'MyComputer\Namespace\';
  ApproveKey =
    'SOFTWARE\Microsoft\Windows\CurrentVersion\Shell Extensions\Approved\';
var
  ClsID: string;
  Value: DWORD;
begin
  ClsID := GUIDToString(Class_ComNameExt);
  inherited UpdateRegistry(Register);
  if Register then begin
    CreateRegKeyEx(NameSpaceKey + ClsId, '', PChar(Description), REG_SZ,
      Length(Description) + 1, HKEY_LOCAL_MACHINE);
    if Win32Platform = VER_PLATFORM_WIN32_NT then
      CreateRegKeyEx(ApproveKey, ClsId, PChar(Description), REG_SZ,
        Length(Description) + 1, HKEY_LOCAL_MACHINE);
    Value := SFGAO_FOLDER;
    CreateRegKeyEx('CLSID\' + ClsId + '\ShellFolder\', 'Attributes',
      @Value, REG_BINARY, SizeOf(DWORD), HKEY_CLASSES_ROOT);
    CreateRegKey('CLSID\'+ClsId+'\DefaultIcon','',ComServer.ServerFileName+',0');
  end else begin
    RegDeleteKey(HKEY_LOCAL_MACHINE, PChar(NameSpaceKey + ClsId));
    if Win32Platform = VER_PLATFORM_WIN32_NT then
      DeleteRegValue(ApproveKey, ClsId, HKEY_LOCAL_MACHINE);
  end;
end;
```

➤ *Listing 4*