Develop Rich-UI Apps

Windows Forms do more than create form-based apps—use them to create MDI and document-view–based apps too.

by Karl E. Peterson and Juval Löwy

Technology Toolbox

- ❑ VB.NET
- ☑ C#
- ❑ SQL Server 2000
- ❑ ASP.NET
- ❑ XML
- ☑ VB6
- ☑ Note:
  Karl Peterson's solutions also work with VB5

Q: Create MDI Apps With Windows Forms

Microsoft calls its rich-UI framework Windows Forms, and when you start a new application, you get a dialog-based (form-based) application. What if I want to develop multiple document interface (MDI) applications, à la Microsoft Foundation Classes (MFC), with Windows Forms? What about using document-view–based architecture with Windows Forms?

A:

Although Microsoft named its rich-UI framework Windows Forms, you can use Windows Forms to develop any type of rich-UI application you're familiar with, from dialog-based to MDI applications. In fact, you can create Windows Forms apps with the same ease you used to create applications with MFC or Visual Basic 6.0 (if not more easily).

Presently, there's no wizard support for nonform-based applications, but it's easy to create a nonform-based application, starting with a form-based application. However, unlike MDI applications, there's no out-of-the-box support for the document-view architecture (using wizards or base classes), mostly because nowadays, you're more likely to use Windows Forms to develop a multitier application's presentation tier, and document-view architecture is rooted in the days of the standalone application. However, nothing prevents you from rolling your own document-view architecture if you wish to.

You need to follow a few steps when developing MDI-based applications. Create a new Windows Forms application and call it MDIApp. You need two types of windows: a main frame window and a child window (although you could have multiple types of child windows). The main frame window serves as the parent window of all the child windows. The Visual Studio .NET Windows Forms application wizard provides a form class by default, which you can use as the parent window. Display that form Properties window, and set the IsMdiContainer property to True. This sets the form's background color immediately to the standard container dark gray, instead of

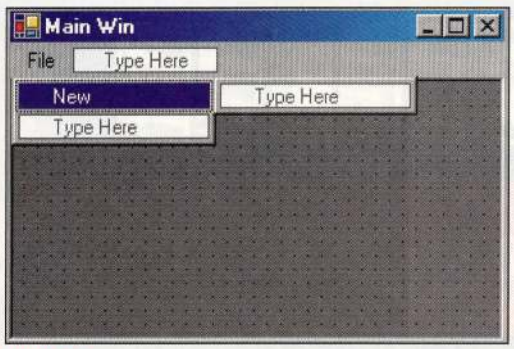

Figure 1 Build the Menu of the Main Frame's Parent Window. The Windows Forms visual designer lets you build menu items, then map them to handling methods. Windows Forms merge a child window's menu automatically in its MDI parent's menu.

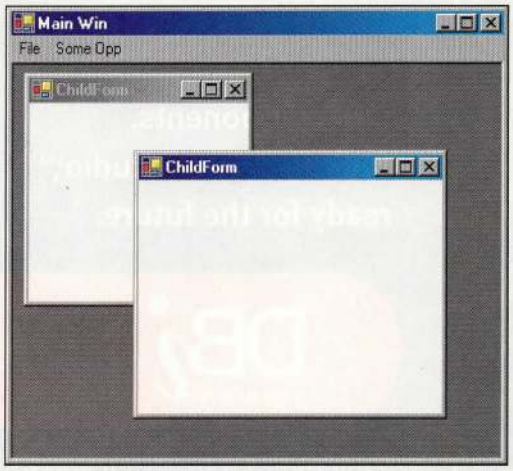

Figure 2 Create the MDI Application. You can create as many child windows as you like, hosted in the main parent frame window. Note that the child window menu is merged with that of its parent.

Resources

Word 95 WordBasic Help file: www.microsoft.com/downloads/release.asp?releaseid=26572

the light gray used in forms. Set the Window caption (the Text property) to Main Win while you have the Properties window open.

Next, add a child window: From the Project menu, select Add Windows Form, type ChildForm in the dialog box, and click on OK. Set the new child form's Background property to Window. For an MDI application, the convention is to launch new child windows (hosted in their parent main window) using the File | New menu item. In addition, the convention is to merge the active child window's menu with its parent window's menu. To do that, you need to add a menu to the parent and a menu to the child window. Starting with the child window, drag a menu item control from the Toolbox and drop it on the child window. Type "Some Opp" for the child single menu item. Open the parent form, and drop a menu on it as well. Type File for the top-level menu item, and New as a submenu item (see Figure 1). Display the menu item events property page, and type OnFileNew for the name of the method handling the Click event. This generates the handling method. Add this code in the handling method:

```
private void OnFileNew(object
   sender,EventArgs e)
{
   ChildForm childForm = new
      ChildForm();
   childForm.MdiParent = this;
   childForm.Show();
}
```

The code is straightforward: You create a new child window, set its MdiParent property to the parent frame, and display the child window (see Figure 2). —*J.L.*

Q: Reflect Your Assembly Version

I'm trying to reflect my assembly version programmatically to address some configuration management needs. My reflection code can reflect any assembly attribute except AssemblyVersion. What's going on?

A:

You use the AssemblyVersion attribute to specify your assembly version (usually in the Assembly.cs or .vb file):

```
[assembly: AssemblyVersion ("1.2.3.4")]
```

However, AssemblyVersion is a special attribute whose value isn't recorded in the metadata but rather in the manifest. Because of that, you can't reflect it. If you want to reflect the assembly version programmatically, you can use the Assembly type's GetName() method. GetName() returns an instance of the AssemblyName class. The AssemblyName class has a public property called Version, of a class called Version. You can then either access individual version numbers (see Listing 1) or simply convert the property to a string:

```
void TraceVersion(Assembly assembly)
{
   Version version =
      assembly.GetName().Version;
   Trace.WriteLine("Version is " +
      version.ToString());
}
```

C# • Make the Most of the Version Class

```
public sealed class Version :
   ICloneable,IComparable
{
   //Constructors
   public Version();
   public Version(int major, int minor);
   public Version(int major, int minor, int build);
   public Version(int major, int minor, int
      build, int revision);
   public Version(string version);

   //Properties
   public int Build { get; }
   public int Major { get; }
   public int Minor { get; }
   public int Revision { get; }

   //Methods
   public virtual object Clone();
   int CompareTo(object obj);
   public string ToString(int fieldCount);
}
```

Listing 1 The Version class provides a type-safe way of building and retrieving an assembly version. You can access individual version numbers, get the version as a string, and compare version numbers.

The Version class is a type-safe way of representing an assembly version. You can build an assembly version using individual version numbers, and you can compare two version numbers:

```
Version v1 = new Version("1.2.3.4");
Version v2 = new Version(5,6,7,8);
Debug.Assert(v1 != v2);
```

—*J.L.*

Q: Determine If Word is Installed

My application needs to know if Microsoft Word is installed, and if so, what version. I use this information to make the best possible use of automation, when available. Can I dig this out of the Registry somewhere?

A:

You could, but that sounds like an awful lot of work. You want to optimize automation, so let's use that technique to answer this question. The Application object exposed by Word offers a Version property, which you can query like this:

```
Public Function WordVersion() As String
   Dim obj As Object
   ' Quick test to determine if Word is
   ' installed, and return version.
   On Error Resume Next
   Set obj = _
```

```
      CreateObject("Word.Application")
   WordVersion = obj.Version
   Set obj = Nothing
End Function
```

This function relies on ignoring an error that's triggered if Word isn't installed. Test the return value first for length, as an empty string indicates no response from (or no creation of) the queried object. Now, you'll run into a slight problem if your specification calls for automating Word 95 as well, because that version used WordBasic rather than VBA. The good news is that newer versions of Word continue to expose the Word.Basic object. So, to be as universal as possible, you'd want to modify the preceding routine like this:

```
Public Function WordVersion() As String
   Dim obj As Object
   ' Quick test to determine if Word is
   ' installed, and return version.
   On Error Resume Next
   Set obj = CreateObject("Word.Basic")
   WordVersion = obj.AppInfo$(2)
   obj.AppClose
   Set obj = Nothing
End Function
```

Use either of these functions by examining the string returned. An empty string indicates no automatable version of Word is installed; otherwise, branch according to the string's contents (see Table 1). The Office group has gone out of its way to maintain backwards compatibility in this case. In fact, you can develop automation code for Word 95, even if it's not installed on your machine, by coding against the WordBasic model using whatever version of Word you do have installed.

I recommend using WordBasic only in cases where you find Word 95 installed, however. Using the newer Word.Application object is preferable, both in terms of speed and robustness of code. A further disincentive might be the fact that Microsoft no longer supports the WordBasic model actively. *(My thanks to Jonathan West, Word MVP, for his valuable input to this response.) —K.E.P.*

Q: Force Painting of Menu Checks

I like to set the Checked and Enabled properties of some dropdown menu items when the top-level menu is selected. For example, I use a checkmark to indicate that the current selection is Bold, and this attribute is far easier to test on demand than to track continuously. I've noticed that the checkmarks aren't always painted when the menu first drops. Toggling the Enabled property doesn't display this paint problem. As it is now, the user must wave the cursor over the item to force a repaint as the highlight bar moves over the item. What can I do programmatically to force VB to paint menu checkmarks properly?

A:

I've seen this same behavior, but never spent the time to track down exactly when it was occurring. In other words, thanks for asking, as this is something that's irritated me too! I wrote a little test applet that toggles the Checked property value of two first-level menu items each time the top-level menu is dropped:

```
Private Sub mMain_Click(Index As Integer)
   mTest(0).Checked = Not mTest(0).Checked
```

Version	Return String
Word 95	"7.0" "7.0a" "7.0c"
Word 97	"8.0" "8.0 SR-1" "8.0 SR-2"
Word 2000	"9.0" "9.0 SR-1" "9.0 SR-2" "9.0 SR-2a"
Word 2002	"10.0" "10.0 SP-1"

Table 1 Deal With Potential Win32 Versions of Word. If you're going to automate Microsoft Word, be prepared to deal with many possible versions. You can use the strings in this table to identify the main variants to date. Using the Val() function against these strings, a return value of 7 (or less) means you must use the Word.Basic object, while a return of 8 or more allows full use of VBA.