Custom Design Time UI in .NET

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

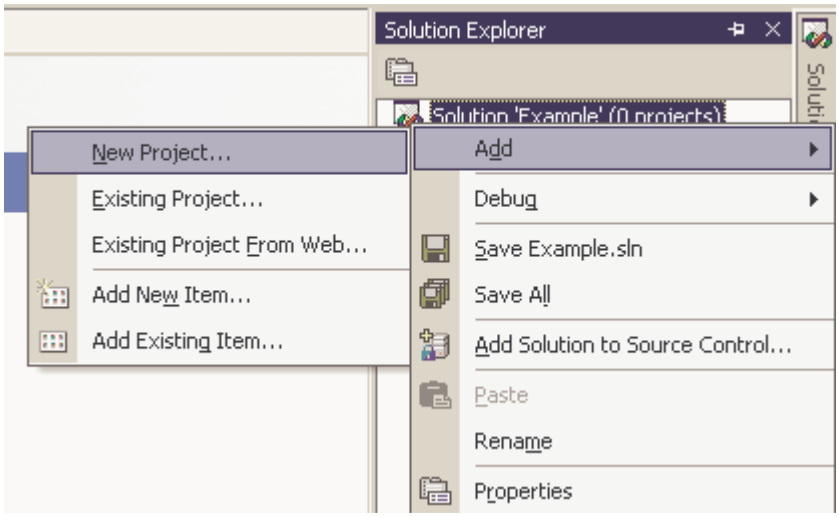
Abstract

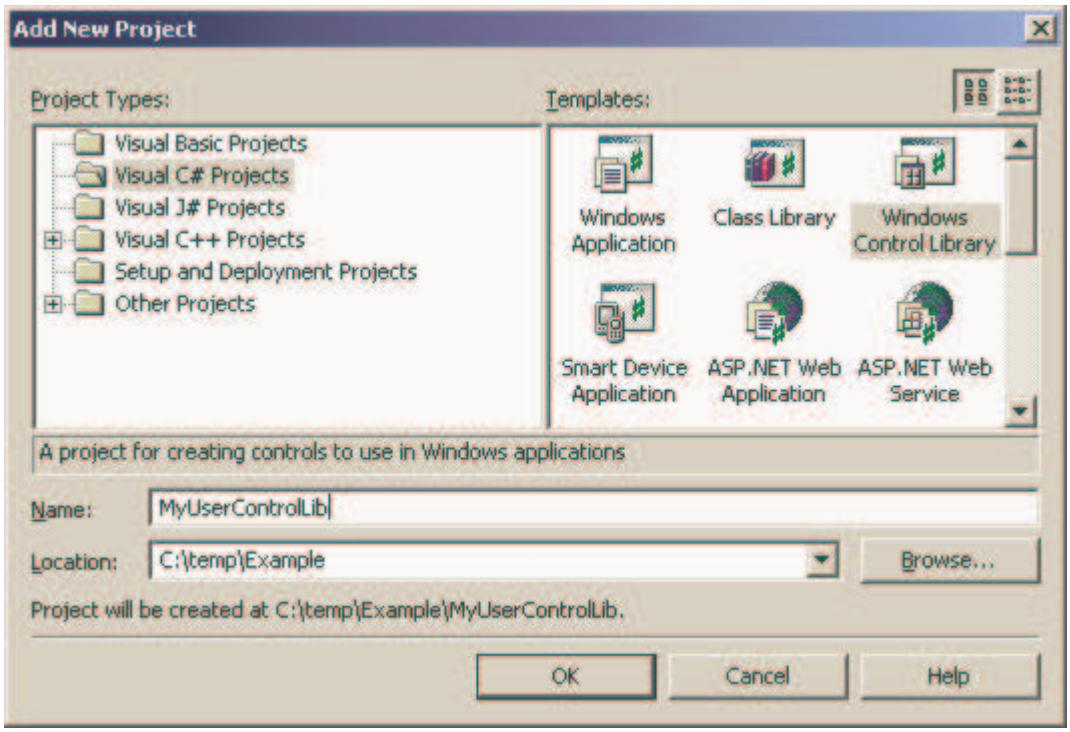
Visual Studio allows us to change various properties of controls at design time. Well defined attributes and classes in the System.ComponentModel namespace and System.Drawing.Design namespace allow us to control how Visual Studio works with controls and properties. Using a simple example we will illustrate these capabilities in this article.

Visual Studio and Properties of Controls

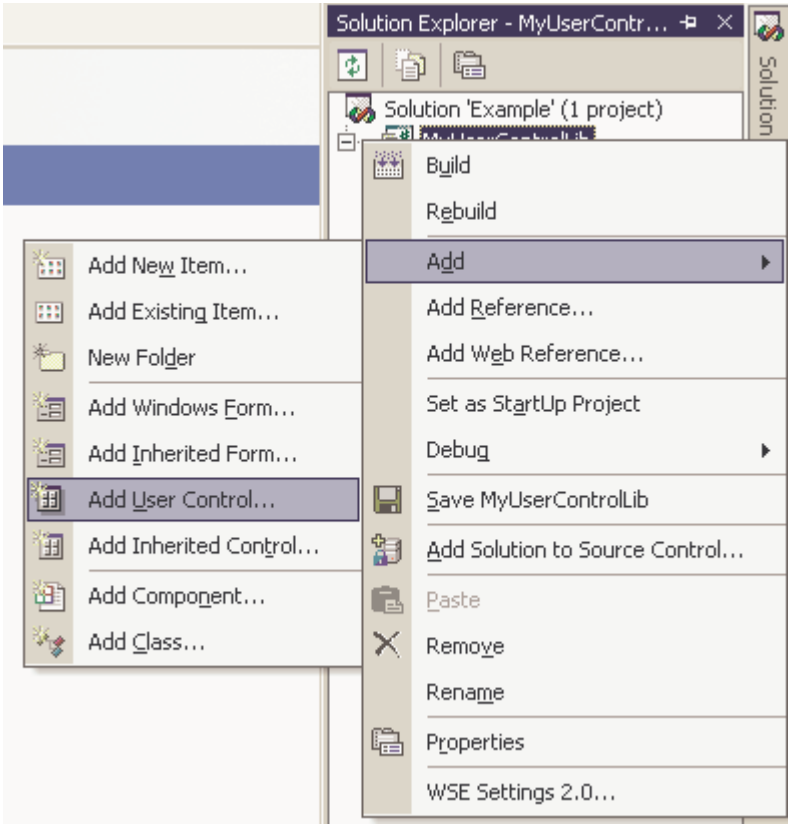
Visual Studio allows us to set various properties of controls at design time. This applies to predefined controls as well as user controls. The motivation to discuss about custom design time UI Type Editor comes from a question that was asked by a subscriber to Agility eLetter.

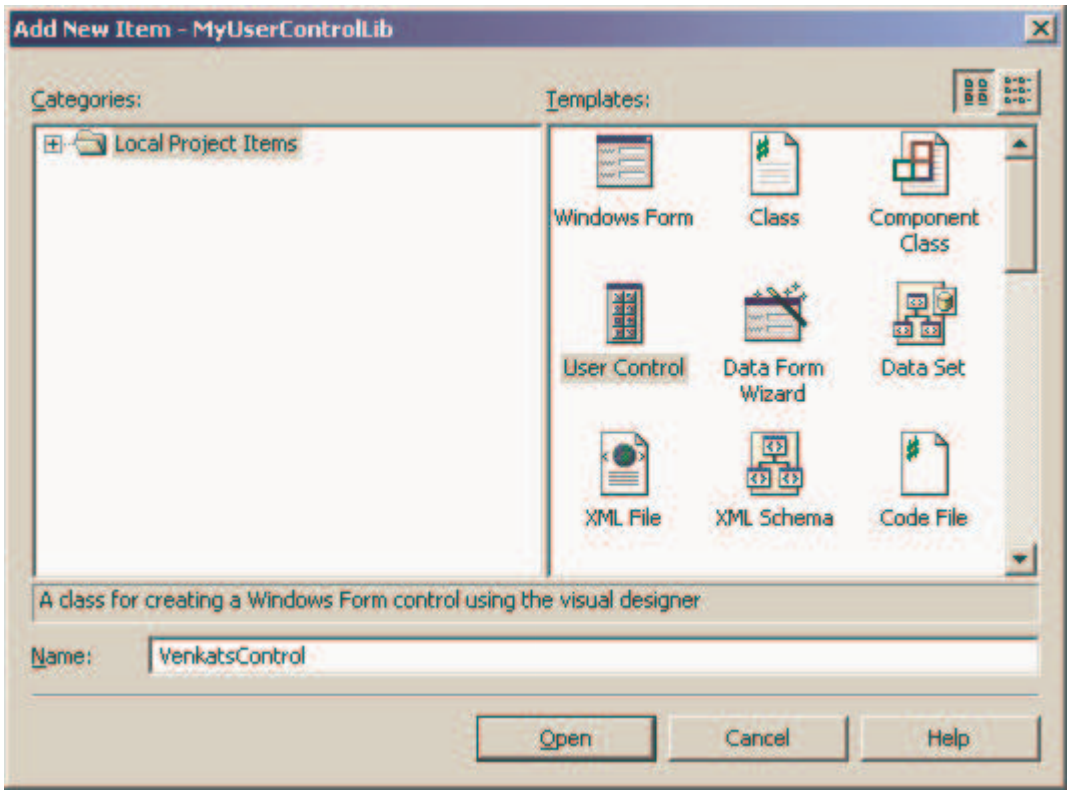
How does one control how Visual Studio interacts with properties of a control at design time? We will study this by using simple examples. Let's start by creating a Blank Solution named Example in Visual Studio 2003. Under this solution we will first create a C# Windows Control Library project named MyUserControlLib as shown below:



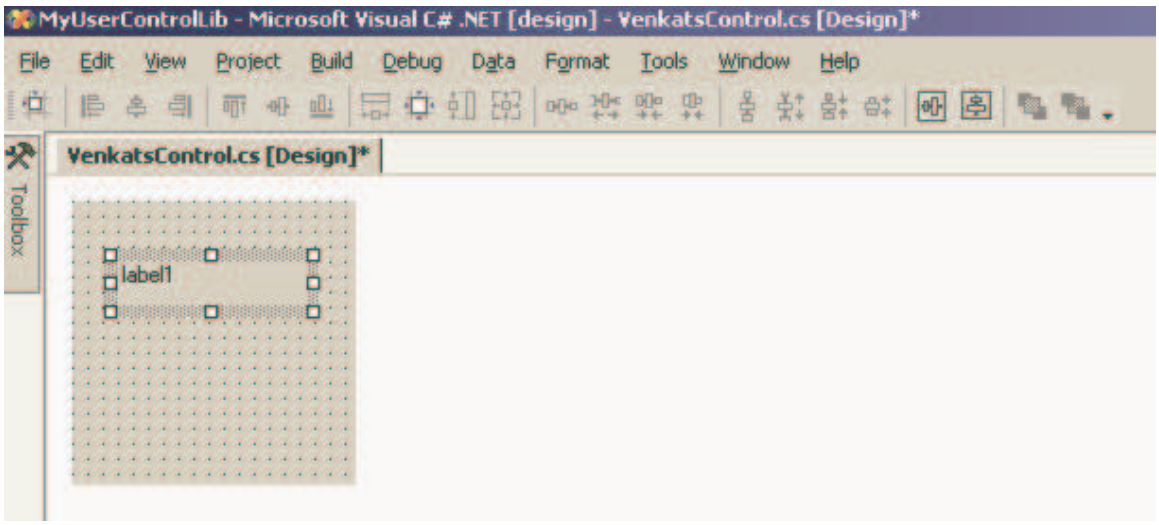


I first delete the default wizard created user control named UserControl1.cs. Then, I create a user control named VenkatsControl as shown below:





In solution explorer, double click on VenkatsControl.cs. This should bring up the control in design view. Click on View | Toolbox and select Label control from the Windows Forms list. Drag and drop Label onto the control. The resulting placement of the label is shown below:



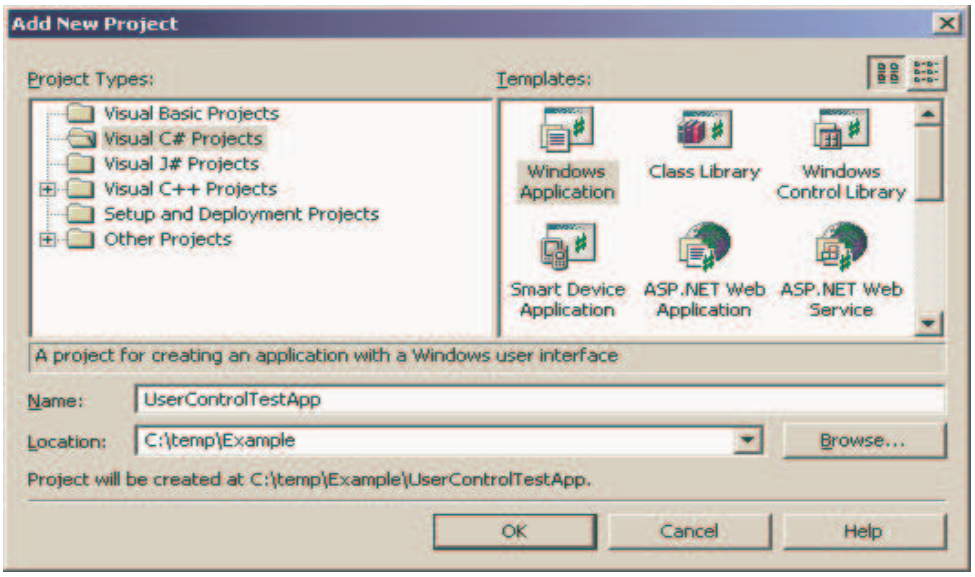
Now, right click on the control in design view and click on View Code. Edit the code to add a field and a property as shown below:

```
public class VenkatsControl : System.Windows.Forms.UserControl
{
	private System.Windows.Forms.Label label1;

	private string theMessage;

	public string Message
	{
		get
		{
			return theMessage;
		}
		set
		{
			theMessage = value;
		}
	}

…
```

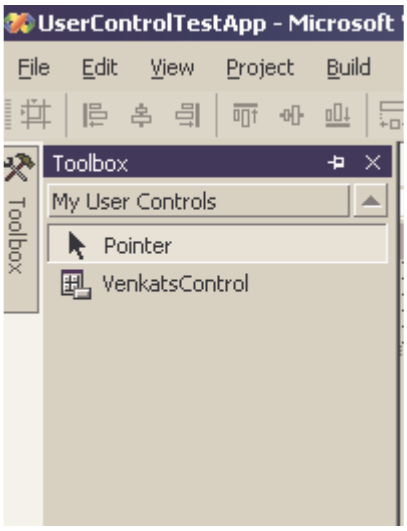
Now double click on the control in design view and you will notice that the Load event handler is added. Edit the event handler as shown below:

```
…

private void VenkatsControl_Load(object sender, System.EventArgs e)
{
	label1.Text = theMessage;
}
…
```

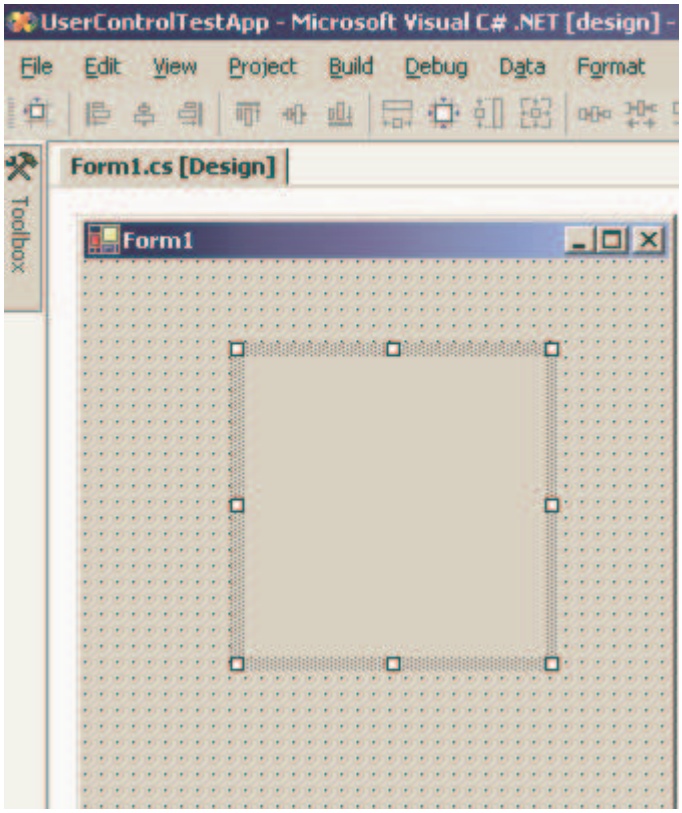
Now again under the Example solution, create another C# Windows Application project named UserControlTestApp as shown below:



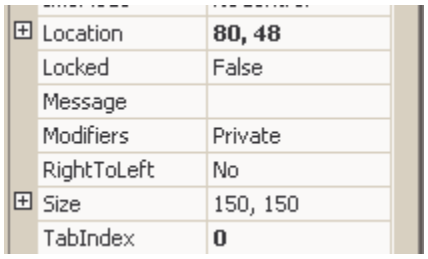
Form1.cs is created for you by default. With the Form in the design view, view the "My User Controls" list in the Textbox. The VenkatsControl should appear in this list as shown below:



Drag and drop the VenkatsControl on the Form as shown below:



Right click on the control and click on Properties. The Message property appears in the Properties list as shown below:



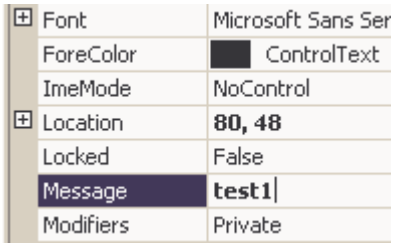
Now type in a value of "test" for the Message property and notice that the value appears in boldface.

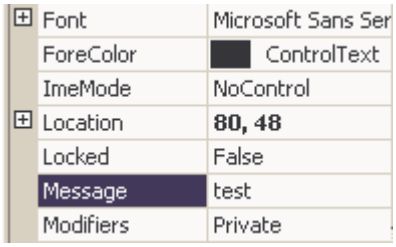
Properties and Design Time Default Values

The value of the Message property appears in boldface because the value of "test" is different from the default value. What is the default value though? In this case, since we did not assign it any thing, it is empty. Let us go ahead and provide it a default value. Modify the code for VenkatsControl as shown below:

```
[DefaultValue("test")]
public string Message
{
	get
	{
		return theMessage;
	}
	set
	{
		theMessage = value;
	}
}
```

Now, in the Form, right click on the VenkatsControl1's property and look for the Message property. Its value "test" is not in boldface. Now modify the value of the Message property to test1 and note that it appears in boldface. Changing it back to "test" removes the boldface again as shown below:





Visual Studio queries the attributes of the property, looking for the DefaultValueAttribute. If the value given for the property at design time is different from the default value, then it displays the value in boldface. DefaultValueAttribute class is part of the System.ComponentModel namespace.

Enumeration Properties and Visual Studio

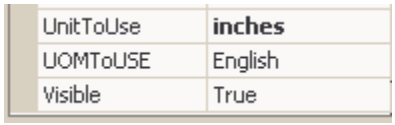
In the above example, the type of Message property was a string. The type of the property may be of different types. If the type of the property is an enumeration, however, Visual Studio displays a combo box instead of a simple textbox. Further, the values in the combo box are the different values of the enumeration. This is illustrated in the following example. Let's continue with the above code. We will create three enumerations UOM, EnglishLengthUnits and MetricLengthUnits as shown below:

```
public enum UOM
{
      English,
      Metric
}

public enum EnglishLengthUnits
{
      inches,
      feet,
      miles
}

public enum MetricLengthUnits
{
      mm,
      cm,
      m,
      km
}
```

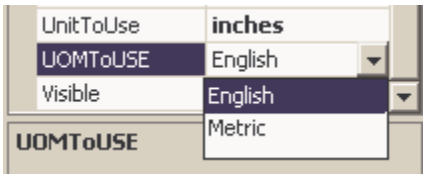
Now, we will add two properties and two fields to VenkatsControl class as shown below:

```
      private UOM theUOM;
      private EnglishLengthUnits theUnit;

      [DefaultValue(UOM.English)]
      public UOM UOMToUSE
      {
            get { return theUOM; }
            set { theUOM = value; }
      }

      public EnglishLengthUnits UnitToUse
      {
            get { return theUnit; }
            set { theUnit = value; }
      }
```

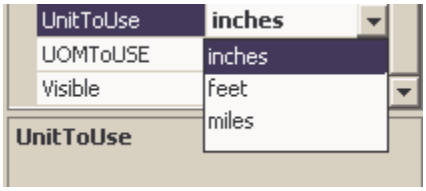
Now, in the Form, view the VenkatsControl1's property. You should see the following:



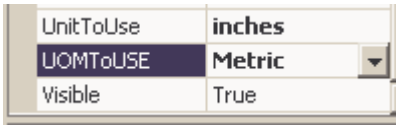
If you click on the UOMToUse property's value, a Combobox appears at that location. Clicking on the dropdown displays the following:



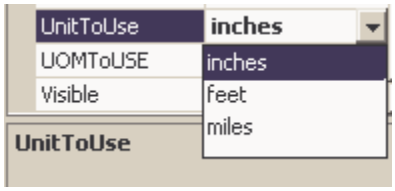
Simlarly, selecting the UnitToUse displays:



Now, let's modify the UOMToUse from English to Metric.



Now, let's click on the UnitToUse and see what appears in the combo box. We see:



Of course, it shouldn't be a surprise that we still see the Metric Units and not English units. This is not desirable, however. When the UOMToUse is English, we want UnitToUse to be inches, feet, etc. When the UOMTOUse is changed to Metric, we would like to see the options of mm, cm, m, etc. for the UnitToUse. How do we do that?

Value Type Dependency?

While we want the values for the UnitToUse to correspond to the value chosen for UOMToUse, this does not happen automatically. The first problem is the type of UnitTouse is hardwired to EnglishLengthUnits. This has to change. If the value of UOMToUse is English, the type of UnitToUse should be EnglishLengthUnits. However, if the value of UOMToUse is Metric, we want the type of UnitToUse to be MetricLengthUnits. How can we change the type of a property based on the value of another property? (Next time you see me, if I have lesser hair on my head, you know the reason☺)

UI Type Editor to Rescue

Fortunately, Visual Studio goes beyond what is discussed above in terms of its flexibility. It allows one to wire a different editor for each property, if so desired. When you click on a property's value in the Property list, Visual Studio checks with the metadata for the property to see if the author (of the property) has asked a specific editor to be used. If so, it uses that editor. If not, it uses a default editor depending on the type of the property.

The editor to edit the property derives from the UITypeEditor class of the System.Drawing.Design namespace. This class acts as a base class for custom type editors for the design time environment. In order to write your own custom editor for editing the properties at design time, you may write a class that derives from this UITypeEditor class. You may then override the GetEditStyle method to return the type of the editor you are implementing. The possible values are DropDown, Modal and None from the UITypeEditorEditStyle enumeration. The value of DropDown tells that a dropdown arrow button must be placed in the property's value field/area and the UI will be a dropdown dialog. The value of Modal indicates that an ellipsis button (…) should be placed instead. Finally, the value of None indicates that no UI is provided. In addition to overriding the GetEditStyle method, one may also override the EditValue method. This method handles the user input processing and value assignment. Further one may override other methods if more UI presentation is desired.

Implementing UI Type Editor

Let's get back to our example and see how we can implement the UITypeEditor so we can achieve the desired behavior for UnitToUse property based on the UOMToUse property's value.

We first create a class named UnitToUseTypeEditor in the MyUserControlLib project as shown below:

```
using System;
using System.Windows.Forms; // For ListBox
using System.Drawing.Design; // For UITypeEditorEditStyle
using System.Windows.Forms.Design; //  For IWindowsFormsEditorService
using System.ComponentModel; // For ITypeDescriptorContext

namespace MyUserControlLib
{
      public class UnitToUseTypeEditor :
                                    System.Drawing.Design.UITypeEditor
      {
            private IWindowsFormsEditorService editorService;

            public override UITypeEditorEditStyle GetEditStyle(
                                       ITypeDescriptorContext context)
            {
                  return UITypeEditorEditStyle.DropDown;
            }

            private void ItemSelected(object sender, EventArgs e)
            {
                  if (editorService != null)
                  {
```

```
            editorService.CloseDropDown();
        }
    }

    public static Type GetTypeOfEnumerationToUse(UOM UOMValue)
    {
        Type typeOfEnumToUse = null;
        switch(UOMValue)
        {
            case UOM.English:
                typeOfEnumToUse =
                            typeof(EnglishLengthUnits);
                break;

            case UOM.Metric:
                typeOfEnumToUse =
                            typeof(MetricLengthUnits);
                break;
        }

        return typeOfEnumToUse;
    }

    public override object EditValue(
            ITypeDescriptorContext context,
            IServiceProvider provider, object value)
    {
        editorService =
                provider.GetService(
                typeof(IWindowsFormsEditorService)) as
                IWindowsFormsEditorService;

        if(editorService == null)
            return value;

        VenkatsControl theControl =
                        context.Instance as VenkatsControl;

        Type typeOfEnumToUse =
                        GetTypeOfEnumerationToUse(
                        theControl.UOMToUSE);

        ListBox aListBox = new ListBox();
        foreach(string anItem in
                            Enum.GetNames(typeOfEnumToUse))
        {
            aListBox.Items.Add(anItem);
        }

        try
        {
            aListBox.SelectedItem =
                        Enum.Parse(typeOfEnumToUse,
                        value.ToString()).ToString();;
        }
        catch
        {// Ignore if value does not match Enumeration
```

```
                }

                aListBox.SelectedIndexChanged +=
                            new EventHandler(ItemSelected);

                editorService.DropDownControl(aListBox);
                return Enum.Parse(
                                typeOfEnumToUse,
                                aListBox.SelectedItem.ToString());
            }
        }
}
```

Let’s understand what the above code does.

The IWindowsFormsEditorService is an interface in the System.Windows.Forms.Design namespace. This interface allows the UITypeEditor to display a UI to edit the property in design mode. This will be used from within the EditValue method of the UITypeEditor (as seen later). The EditValue method is provided a reference to this interface which is saved in the editorService field within our UITypeEditor.

The GetEditStyle method simply indicates that the UI is a dropdown style.

The ItemSelected method is an event handler that is assigned to the ListBox in the EditValue method. When an item is selected from the listbox, this handler instructs the IWindowsFormEditorService to close the editor.

The GetTypeOfEnumToUse is a helper method. Based on the value of the UOMToUse property, it determines which Enumeration, EnglishLengthUnits or MetricLengthUnits to use.

Finally, the important and interesting method EditValue is implemented. In this method we first cache or save away a reference to the IWindowsFormsEditorService. We then obtain a reference to the control being edited in the design mode (in this case it is the VenkatsControl). We determine the type of the Enumeration to use based on the UOMToUse property of VenkatsControl’s. Then we create a ListBox and populate it with the values of the appropriate enumeration (EnglishLengthUnits or MetricLengthUnits). We then set the selected value of the listbox to the current value of the UnitToUse property (given to us though the value variable). We tie the ItemSelected handler to the SelectedIndexChanged event of the ListBox. Finally we ask the IWindowsFormsEditorService to popup the listbox. Once the listbox item is selected, the selected value is obtained from the listbox and returned.

There is still some tweaking that is needed on the VenkatsControl. We will see that next.

Connecting the Control’s Property to the UI Type Editor

First, we can't let the UnitToUse to be of type EnglishLengthUnit. Further, we need to indicate to Visual Studio that the editor to be used to modify the values of this property is our UnitToUseTypeEditor. How do we do that? That is the easy part as shown below:

```
private UOM theUOM;
private Enum theUnit = EnglishLengthUnits.inches;

[DefaultValue(UOM.English),
            RefreshProperties(RefreshProperties.All)]
public UOM UOMToUSE
{
      get { return theUOM; }
      set { theUOM = value; }
}

[Editor(typeof(UnitToUseTypeEditor),
      typeof(System.Drawing.Design.UITypeEditor)),
      DefaultValue(EnglishLengthUnits.inches)]
public Enum UnitToUse
{
      get
      {
            Type typeOfEnumToUse =
             UnitToUseTypeEditor.GetTypeOfEnumerationToUse(
                        theUOM);

            try
            {
                  if (Enum.IsDefined(typeOfEnumToUse,
                              theUnit))
                  {
                        return (Enum) Enum.Parse(
                              typeOfEnumToUse,
                              Enum.GetName(
                               typeOfEnumToUse, theUnit));
                  }
            }
            catch
            {// Simply return the first value in this case.
            }

            return (Enum) Enum.GetValues(
                              typeOfEnumToUse).GetValue(0);
      }
      set
      {
            theUnit = value;
      }
}
```

The RefreshProperty attribute on UOMToUse tells Studio that whenever the value of UOMToUse is changed, it needs to redisplay the values of other properties for this control in the design environment.

The Editor attribute on the UnitToUse tells Studio which UITypeEditor to use to edit this property. Note that the type of the UnitToUse property is Enum (not an int or EnglishLengthUnits). The getter for this property checks the type of the UOMToUse and returns the appropriate enumeration type of the field theUnit.

Since significant change was made to the control, it is better to remove the VenkatsControl1 from the Form1 and place a new instance of the control on the Form.

The following shows the value of the UOMToUse being modified (note that UnitToUse accordingly changes automatically):

The following shows the effect of UnitToUse being changed to km and then the UOMTOUse being modified to English:

In the above example, not that EnglishLengthUnits has only three values while MetricLenghUnits has four values (km being the last). When we changed UOMToUse from Metric to English, our code set the UnitToUse to the first value (inches).

Refactoring the code

There are a couple of things that may be improved. One, a factory may be used in the GetTypeOfEnumerationToUse method to determine the type of Enumeration. Also, the GetTypeOfEnumerationToUse may be moved (to a utility class) to eliminate cyclic dependency between VenkatsControl and UnitToUseTypeEditor. Further, since the UnitToUseTypeEditor is so dependent on VenkatsControl, it may be written as a nested class of VenkatsControl class.

Conclusion

Visual Studio provides a very powerful capability to interact with controls at design time. By effectively using the metadata and set of classes and interfaces, one can fine tune the design time behavior of controls and properties. This article has shown, using simple examples, how one may benefit from the custom UI Type Editors for the design time environment.

References

1. http://msdn.microsoft.com