Presenting Dynamic Customised Views of Objects

John Peppiatt
Manukau Institute of Technology,
Auckland, New Zealand

This article illustrates a technique for making use of the .NET PropertyGrid component for use in a .NET WinForm application. The basis of the technology is described through snippets of VB.NET code that show the use of the PropertyGrid and the way reflection is used to provided a customised view of an object.

1. INTRODUCTION

This article demonstrates a technique for presenting selected properties of an object to controls (such as the PropertyGrid) that use Typedescriptor technology to discover an object's properties. The basic idea is demonstrated using the .NET PropertyGrid control and it shows how to control the properties that can be seen on the grid and the order of presentation. The author has extended the concept in several related areas including creating a technology for filtering columns in DataViews bound to Datagrids, and also to produce new controls that are derived versions of the PropertyGrid for data presentation and entry.

2. BACKGROUND

Like many fellow programmers, I find writing code to edit the properties of objects – like the details of a customer or order – boring but unfortunately necessary stuff. The driving force behind many a programming innovation is, of course, the boredom from repetition and a feeling that there simply must be a better way! This feeling led me to explore, and perhaps like many other programmers my attention turned to the PropertyGrid control. This control supports the familiar properties window used in the Visual Studio IDE to edit the properties of objects you use in design time editors, and, although it is not on the ToolBox by default, it is simply added using the Add/Remove feature of the ToolBox. I found that using it is easy and there is a lot of potential for using it – especially if you delve into creating custom type editors – but it lacks some very basic features that limit its use for a normal user data entry function. The key limitations are the inability to select the properties you wish the user to see and change, and the inability to specify the order the properties appear. There is some control of order – alphabetic, or by even by category, but categories must be defined using an attribute of the property at compile time (not a particularly flexible arrangement). Of course, I do not want to appear overly critical of this control as it was designed to support development IDE's and it really does employ some excellent concepts – but it is just not that 'bendable' for my purpose. Still the PropertyGrid was so close in concept that I went down that familiar path and asked *'could I fool it into doing what I want'*? I discovered I could, and as a bonus I found the technique allowed me to do similar things when working with ADO Dataviews and DataGrids – the idea however is most easily demonstrated using the PropertyGrid.

3. THE SOLUTION

I was already familiar with the concepts of reflection and the function of the TypeDescriptor class, its GetProperties methods and in particular how these work when a class implements an interface known as ICustomTypeDescriptor. My strategy was to devise a class that could wrap up an object of any type and expose only those properties defined in some list and in the order determined by the list. Then rather than presenting the raw object to the control (e.g. PropertyGrid), the wrapper class (I call this CustomView) object would be used

```
Dim p As New Person

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
                       System.EventArgs) Handles MyBase.Load
    Dim props() As String = {"Name", "Married"}
    p.Name = "John Peppiatt"
    p.DOB = #1/1/1970#
    p.Married = True
    pgrDemo.SelectedObject = New CustomView(p, props)
End Sub
```

Figure 1

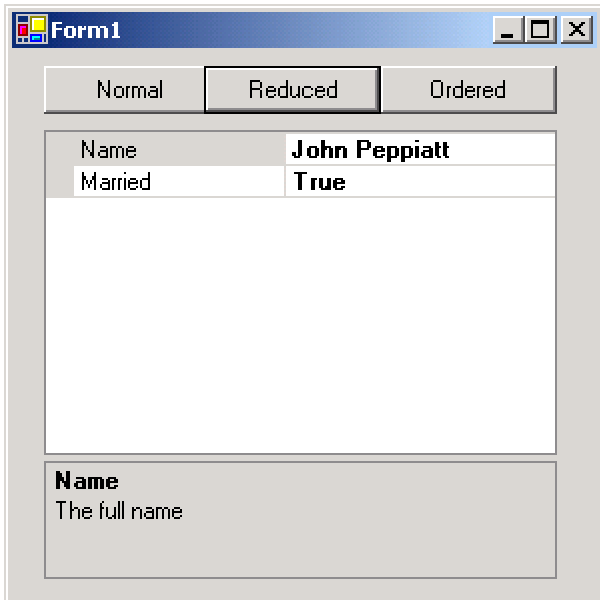

Figure 2

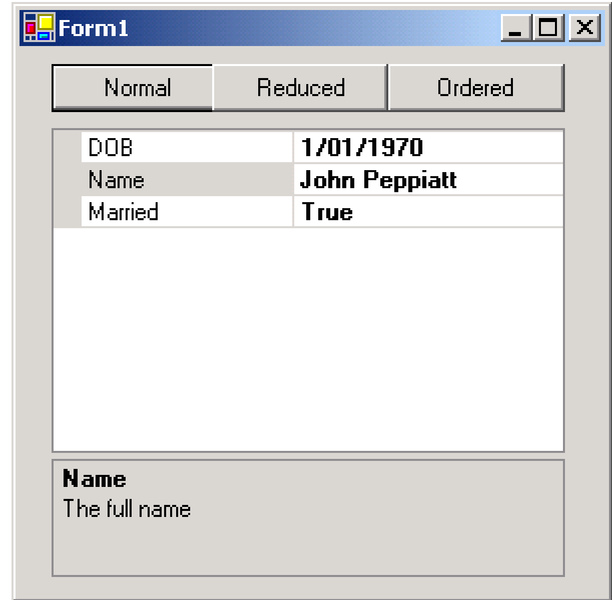

Figure 3

in its place. The following code snippet illustrates this process (Figure 1).

Where pgrDemo is a PropertyGrid, and Person is a simple class with three properties – Name, DOB and Married. The CustomView class is the wrapper class that works the magic. In my demo application the grid comes up like Figure 2.

Note that pgrDemo (the PropertyGrid) has the ToolBarVisible property set to False and the PropertySort set to NoSort.

The standard use of the PropertyGrid is to assign the raw object to its SelectedObject property i.e.

```
pgrDemo.SelectedObject = p
```

rather than

```
pgrDemo.SelectedObject = New CustomView(p, props)
```

where p is an object of the person class used in Figure 1. This more usual use would look like Figure 3.

The CustomView class illustrated here is a pared down version of the one I use in production code. This example simply uses a constructor requiring the object to be wrapped plus a string array containing the names of the properties in the order required for presention. My production code allows variations like making a property read-only on the grid when it is read-write in the class, together with the ability to rename properties and specify domain type values.

This basic CustomView class implements the ICustomTypeDescriptor interface and for the most part simply delegates to code supporting the class of the objected being wrapped through the PropertyDecriptors exposed by that type.

The code for CustomView is provided in Figure 4 (over page). As can be seen, CustomView is full of simple delegation code. The points of note are the constructor, the two forms of GetProperties and the supporting GetPropertyList function that really is the heart of the class. The GetPropertyList function simply substitutes the full list of properties returned from the wrapped object with a filtered ordered list based upon the array supplied at construction time.

```
Imports System.ComponentModel
Public Class Person
    Private _name As String
    Private _dob As Date
    Private _married As Boolean
    <Description("The full name")> Public Property Name() As String
        Get
            Return _name
        End Get
        Set(ByVal Value As String)
            _name = Value
        End Set
    End Property
    <Description("Date of birth")> Public Property DOB() As Date
        Get
            Return _dob
        End Get
        Set(ByVal Value As Date)
            _dob = Value
        End Set
    End Property
    <Description("Married or not")> Public Property Married() As
     Boolean
        Get
            Return _married
        End Get
        Set(ByVal Value As Boolean)
            _married = Value
        End Set
    End Property

End Class
```

Figure 5

4. OTHER NOTES

The demonstration grid does show help tips below the properties. The text for these comes from the Description attribute specified in the class of the wrapped object. Figure 6 illustrates this with the Person Class.

It is not necessary to have descriptions on properties, and the PropertyGrid will remove the pane if the HelpVisible property is set to false. It does however indicate the potential of dynamically creating property descriptors with dynamically created descriptions at run-time!

```
Imports System.ComponentModel
Public Class CustomView
      Implements ICustomTypeDescriptor
     Private _Me As Object
      Private _PropertyList() As String
      <Description("Obj is the object to manage, props is the list of properties to return")> _
      Public Sub New(ByVal Obj As Object, ByVal props() As String)
         _Me = Obj
           _PropertyList = props.Clone
    End Sub
      <Description("Returns the object being managed (set in the constructor).")> _
      Public ReadOnly Property BaseObject() As Object
         Get
             Return _Me
         End Get
     End Property
      <Description("Simply delegates to the TypeDescriptor method.")> _
      Public Function GetAttributes() As System.ComponentModel.AttributeCollection Implements
System.ComponentModel.ICustomTypeDescriptor.GetAttributes
```

Figure 4 (start)

5. CONCLUSION AND EXTENSIONS

This technique is remarkably simple and works with most objects. I have already extended the idea to allow specialized validation techniques and domain value selection options - all worked through properties of a derived PropertyGrid control. I have also adapted the concept for managing collections of objects to bind to a DataGrid but there is more complexity implementing interfaces to support a wrapped editable collection but it feels good when you've done it. Modern programming feels like an endless exercise of wrapping objects with other objects but every now and then a neat idea pops up and leverages well in many contexts.

```
            Return TypeDescriptor.GetAttributes(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetClassName() As String Implements
System.ComponentModel.ICustomTypeDescriptor.GetClassNaMe
            Return TypeDescriptor.GetClassName(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetComponentNaMe() As String Implements
System.ComponentModel.ICustomTypeDescriptor.GetComponentNaMe
            Return TypeDescriptor.GetComponentName(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetConverter() As System.ComponentModel.TypeConverter Implements
System.ComponentModel.ICustomTypeDescriptor.GetConverter
            Return TypeDescriptor.GetConverter(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetDefaultEvent() As System.ComponentModel.EventDescriptor Implements
System.ComponentModel.ICustomTypeDescriptor.GetDefaultEvent
            Return TypeDescriptor.GetDefaultEvent(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetDefaultProperty() As System.ComponentModel.PropertyDescriptor Implements
System.ComponentModel.ICustomTypeDescriptor.GetDefaultProperty
            Return TypeDescriptor.GetDefaultProperty(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Function GetEditor(ByVal editorBaseType As System.Type) As Object Implements
System.ComponentModel.ICustomTypeDescriptor.GetEditor
            Return TypeDescriptor.GetEditor(_Me.GetType, editorBaseType)

      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Overloads Function GetEvents() As System.ComponentModel.EventDescriptorCollection Imple-
ments System.ComponentModel.ICustomTypeDescriptor.GetEvents
            Return TypeDescriptor.GetEvents(_Me.GetType)
      End Function
       <Description("Simply delegates to the TypeDescriptor method.")> _
       Public Overloads Function GetEvents(ByVal attributes() As System.Attribute) As
System.ComponentModel.EventDescriptorCollection Implements
System.ComponentModel.ICustomTypeDescriptor.GetEvents
            Return TypeDescriptor.GetEvents(_Me.GetType, attributes)
      End Function
       <Description("Returns the list of properties as defined in the constructor")> _
       Public Overloads Function GetProperties() As System.ComponentModel.PropertyDescriptorCollection
Implements System.ComponentModel.ICustomTypeDescriptor.GetProperties
          Return GetPropertyList(TypeDescriptor.GetProperties(_Me.GetType))
      End Function
       <Description("Returns the list of properties as defined in the constructor")> _
       Public Overloads Function GetProperties(ByVal attributes() As System.Attribute) As
System.ComponentModel.PropertyDescriptorCollection Implements
System.ComponentModel.ICustomTypeDescriptor.GetProperties
          Return GetPropertyList(TypeDescriptor.GetProperties(_Me.GetType, attributes))
      End Function
          ' The most interesting part of the class
       Private Function GetPropertyList(ByVal props As PropertyDescriptorCollection) As
PropertyDescriptorCollection
          Dim a As New ArrayList
          For Each s As String In _PropertyList
               ' ignore any nonexistent properties
             Try
                  a.Add(props(s))
              Catch ex As Exception
             End Try
         Next
           Dim p(a.Count - 1) As PropertyDescriptor
          a.CopyTo(p)
            Return New PropertyDescriptorCollection(p)
      End Function
       <Description("Returns the wrapped object.")> _
       Public Function GetPropertyOwner(ByVal pd As System.ComponentModel.PropertyDescriptor) As
Object Implements System.ComponentModel.ICustomTypeDescriptor.GetPropertyOwner
          Return _Me
      End Function
End Class
```

Figure 4 (cont)