ARTICLES

June 2004

Article 8 of 10

Index: Please, select

More articles from this author: Please, select

The PropertyGrid control

Eric Moreau, Moer Inc.
Eric Moreau is an independent contractor. He holds the Microsoft Certified Solution Developer (MCSD) certification and is also a Visual Developer - Visual Basic MVP. He is mainly programming client/server applications using VB/VB.Net and MS SQL Server (and all the stuff surrounding it) particularly in the financial industry. He works with VB since version 4 and teaches it since version 5. He is a member of the Montreal Visual Studio User Group (www.guvsm.net) where he gives some sessions. He is also a speaker at the DevTeach conferences since its beginning in 2003.

Have you ever needed a control in your own application having the same features as the properties dialog you use everyday to set the properties of forms and/or controls into the VS.Net IDE? This control is now available to you right from the box. It is called the PropertyGrid control.

As you'll soon discover, the control Microsoft gave us is pretty basic (not surprisingly!!!) but is also flexible and very extensible.

The PropertyGrid control can be used to browse, view, and edit properties of one or more objects of your own application. These objects are not required to be controls. They can be class instances too. The PropertyGrid control uses Reflection to inspect your object and to display the properties using a decent editor.

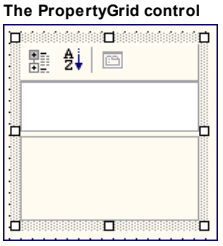
This control is not very used probably because programmers didn't find it yet. I am sure you will find a place to put it in your own applications before you have completed the reading of this column!

I don't see it in my toolbox?

This control is not intrinsic. You need to add it to your toolbox. So open the "Customize Toolbox" dialog (by right-clicking the Toolbox and clicking the "Add/Remove items..." menu item or by using the "Add/Remove Toolbox items..." menu item from the Tools menu). From the ".Net Framework Components", find the PropertyGrid control (if you have more than one, be sure to use the one that is in the System.Windows.Forms namespace).

Adding the control to the form

Figure 1:
The PropertyGrid control



Add an instance of the PropertyGrid control to your form. You will surely recognize a dialog that is familiar. You already see the 3 bands of the PropertyGrid control (the dialog toolbar, the properties band, and the description band).

Filling the control

Now we need something to interact with the control. I have said in the introduction that the control can be used with objects of your applications. Everything you used in your application is an object so it means that everything can be bound to the PropertyGrid!

So to prove it, add this very simple line to your form's Load event and execute it:

```
PropertyGrid1.SelectedObject = Me
```

What is Me? It is an object containing the current form's instance. The SelectedObject property receives the object that the

PropertyGrid control needs to display.

So what you see in the PropertyGrid control are the properties of the form. The Categorized and Alphabetic buttons are already working as expected. The description band is also already working as expected. Exactly as in the VS.Net IDE. And to prove that it is not just a browser, change some properties like the Text property. As soon as you hit Enter, the caption of the form will be changed to whatever you typed! Try to change some other values. For example, try to enter letters into a numeric property like Size.Width (an error will be raised and the value will not be accepted). I finally want to bring to your attention that some properties are not visible. Tell me if you can find the (Name) property!

I agree that you do not want to bind forms to the property grid to let the user modify their properties at runtime but you can hide something for your own use. Because properties are affecting the form (or any other control), you can see immediately the effect of your modifications without having to stop and restart your applications (much what you can do from the Autos, Locals, and Watch windows but within a familiar look).

Filling the control with your own objects

Displaying your own objects in the control is not different. Pretty much any object can be passed to the SelectedObject property. It means that your class instances can be shown.

So here is a useless class which only serves as a demonstration of the features of the PropertyGrid control:

```
Option Strict On

Public Class DemoClass

#Region " --- Members --- "
    Private mBooleanProperty As Boolean
    Private mDateProperty As Date
    Private mDecimalProperty As Decimal
    Private mStringProperty As String
#End Region ' --- Members
```

```
#Region " --- Properties --- "
    Public Property BooleanProperty() As Boolean
        Get
            Return mBooleanProperty
        End Get
        Set(ByVal Value As Boolean)
            mBooleanProperty = Value
        End Set
    End Property

    Public Property DateProperty() As Date
        Get
            Return mDateProperty
        End Get
        Set(ByVal Value As Date)
            mDateProperty = Value
        End Set
    End Property

    Public Property DecimalProperty() As Decimal
        Get
            Return mDecimalProperty
        End Get
        Set(ByVal Value As Decimal)
            mDecimalProperty = Value
        End Set
    End Property

    Public Property StringProperty() As String
        Get
            Return mStringProperty
        End Get
        Set(ByVal Value As String)

            mStringProperty = Value
        End Set
    End Property

#End Region ' --- Properties

End Class
```

For now, this class is much like any one you wrote. It contains properties that are having data types. Now change your form's Load event to browse an instance of this class in the Property Grid control:

```
Dim x As New DemoClass
PropertyGrid1.SelectedObject = x
```

Run your demo application. It's working! The PropertyGrid control is using Reflection to find the data type of each properties and displays an editor that suits the data type. For example, the BooleanProperty is giving you a combo containing True and False. The DateProperty displays a DateTimePicker control. Try entering valid and invalid values.

You can now extend the class definition to add description, category, default value and much more to have a more friendly display. This is done through the use of Attributes. To use attributes in your class, you have to use the required namespace. You need to add this line at the top of your class:

```
Imports System.ComponentModel
```

For example, you can classify your properties and add description to them by changing the BooleanProperty property declaration for this:

```
<CategoryAttribute("Category 1"), _
DescriptionAttribute("Please select a value.")> _
Public Property BooleanProperty() As Boolean
```

The more useful attributes are listed here:

Description	
Category("CategoryName") or CategoryAttribute("CategoryName")	Specifies the category in which the property or event will be displayed in a visual designer. The default value is Misc.
Description("Description") or DescriptionAttribute("Description")	Specifies a description for a property.
Browsable(True/False) or BrowsableAttribute(True/False)	Specifies whether a property should be displayed in a Properties window. The default is True.
DefaultValue("Value") or DefaultValueAttribute("Value")	Specifies the default value for a property.

Most attributes are working as-is. But some other require more code. For example, you add a DefaultValueAttribute to your property; the PropertyGrid control will not automatically display it. You need to manually initialize your variables. Don't worry, I will give you the required code:

```
Dim objAttColl As AttributeCollection
Dim objDefault As DefaultValueAttribute

objAttColl = _
TypeDescriptor.GetProperties(Me)("StringProperty").Attributes
objDefault = CType(objAttColl(GetType(DefaultValueAttribute)), _
                   DefaultValueAttribute)
mStringProperty = objDefault.Value.ToString
```

Take if for granted, adding this code to your class constructor, will search for the default value attribute of a property (the StringProperty in this case), retrieve its value and initialize its member (mStringProperty in this case).

There are much more attributes you can use. You can see the complete list here.

The PropertyGrid control displays an editor that is suitable for the data type of the property.

This means that if you add Font, Color, Icon, Size, Point, and... everything other System data type you are used to, you will see the property grid editor displaying a suitable editor (font dialog, color picker, ...) without having to do anything else. Isn't it wonderful?

property grid editor displaying a suitable editor (font dialog, color picker, ...) without having to do anything else. Isn't it wonderful?

Your own enumerations

Now, let say you want to add a property to your class that has an enumeration (countries, states...). For the demo, I will add a Pasta property (remember that it has to be useless!). For the PropertyGrid being able to display this property correctly, we have to provide a TypeConverter class. This class must inherit from the System.ComponentModel.TypeConverter (or any class derived from it). Because the pasta names are text, we will inherit from the StringConverter.

This class needs to gives enumeration's values and also needs to override some method like this (this is the content of a class named PastaList):

```
Inherits System.ComponentModel.StringConverter

Private PastaNames As String() = New String() { _
   "Fettuccini", "Fusilli", "Lasagna", "Linguini", "Spaghetti", "Macaroni", "Manicotti", _
   "Penne", "Rigatoni", "Rotini", "Vermicelli", "Wagon Wheels", "Chef Boyardee"}

'Returns a collection of standard values for the data type this type converter is designed for.
Public Overloads Overrides Function GetStandardValues _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext) _
    As System.ComponentModel.TypeConverter.StandardValuesCollection

    Return New StandardValuesCollection(PastaNames)
End Function

'Returns whether the collection of standard values returned from GetStandardValues is
'an exclusive list. Returning False would change the drop down list to an editable combo box
Public Overloads Overrides Function GetStandardValuesExclusive _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext) As Boolean

    Return True
End Function

'Returns whether this object supports a standard set of values that can be picked from a list.
Public Overloads Overrides Function GetStandardValuesSupported _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext) As Boolean

    Return True
End Function
```

Now that your TypeConverter class is defined, you have to specify to use it using an attribute:

```
<TypeConverter(GetType(PastaList)), _
Category("Custom enum"), _
Description("Select your prefered favor of pasta from the list.")> _
Public Property PastaProperty() As String
    Get
        Return mPastaProperty
    End Get
    Set(ByVal Value As String)
        mPastaProperty = Value
    End Set
End Property
```

That's it! You can now select my preferred ready-made pasta cook: Chef Boyardee.

Your own expandable properties

Just like Size, Point, Font, and many other properties that are expandable, you can create your own expandable property. Once again, it is simply a matter of providing a class inherited from the ExpandableObjectConverter class (itself inherited from the TypeConverter class). The derived class also needs to convert to a string (much like a Location property contains X and Y properties and displays as X, Y when collapsed).

In fact this time, you need to provide a class. The first one defines your custom type. The second class is your TypeConverter.

Here is the code for the first class:

```
<TypeConverter(GetType(SizeOfCanConverter))> _
Public Class SizeOfCan

    Private mHeight As Short
    Private mWidth As Short

    <Description("Set the Height of the can")> _
    Public Property Height() As Short
        Get
            Return mHeight
        End Get
        Set(ByVal Value As Short)
            mHeight = Value
        End Set
    End Property

    <Description("Set the Width of the can")> _
    Public Property Width() As Short
        Get
            Return mWidth
        End Get
        Set(ByVal Value As Short)
            mWidth = Value
        End Set
    End Property
End Class
```

From this code, we see that we define a data type that has two properties. This class is just like any other except for the TypeConverter attribute used at the top that points to a-yet-undefined converter!

Here is now the code of the converter itself:

Here is how the code of the converter itself:

```
Inherits ExpandableObjectConverter

'Returns whether this converter can convert an object
'of one type to the type of this converter.
Public Overloads Overrides Function CanConvertFrom _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext, _
    ByVal sourceType As System.Type) As Boolean

    If sourceType Is GetType(String) Then
        Return True
    End If
    Return MyBase.CanConvertFrom(context, sourceType)
End Function

'Returns whether this converter can convert the object to the specified type.
Public Overloads Overrides Function CanConvertTo _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext, _
    ByVal destinationType As System.Type) As Boolean

    If destinationType Is GetType(SizeOfCan) Then
        Return True
    End If
    Return MyBase.CanConvertFrom(context, destinationType)
End Function

'Converts the given value to the type of this converter.
Public Overloads Overrides Function ConvertFrom _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext, _
    ByVal culture As System.Globalization.CultureInfo, ByVal value As Object) As Object

    If TypeOf value Is String Then
        Try
            Dim strValue As String = value.ToString

            Dim arrValues() As String
            'Dim strResult As String
            Dim objSizeOfCan As SizeOfCan = New SizeOfCan

            arrValues = strValue.Split(",")c)
            If Not IsNothing(arrValues) Then
                If Not IsNothing(arrValues(0)) Then
                    objSizeOfCan.Height = Convert.ToInt16(arrValues(0))
                End If
                If Not IsNothing(arrValues(1)) Then
                    objSizeOfCan.Width = Convert.ToInt16(arrValues(1))
                End If
            End If
            Return objSizeOfCan
        Catch ex As Exception
            Throw New ArgumentException("This value cannot be converted to a SizeOfCan.")
        End Try
    End If

    Return MyBase.ConvertFrom(context, culture, value)
End Function

'Converts the given value object to the specified type.
Public Overloads Overrides Function ConvertTo _
    (ByVal context As System.ComponentModel.ITypeDescriptorContext, _
    ByVal culture As System.Globalization.CultureInfo, _
    ByVal value As Object, _
    ByVal destinationType As System.Type) As Object

    If destinationType Is GetType(String) AndAlso TypeOf value Is SizeOfCan Then
        Dim objSizeOfCan As SizeOfCan = CType(value, SizeOfCan)
        Return objSizeOfCan.Height & ", " & _
               objSizeOfCan.Width
    End If
    Return MyBase.ConvertTo(context, culture, value, destinationType)
End Function
```

The main thing about this class is that it has to convert from a string to an object (or vice-versa). That's why the ConvertFrom and ConvertTo methods have more lines of code.

Now that the converter is declared, using it is really simple:

```
<Category("Custom enum"), _
Description("Specify the size of the can.")> _
Public Property SizeOfCanProperty() As SizeOfCan
    Get
        Return mSizeOfCanProperty
    End Get
    Set(ByVal Value As SizeOfCan)
        mSizeOfCanProperty = Value
    End Set
End Property
```

Formatting output

You may also want to format the value of some properties. For example if the class has a Price property, you may want to format it using the currency symbol.

This task requires another converter class that is then assigned to a property using the TypeConverter attribute. Download the demo code and have a look at the CurrencyConverter class and to the CanPrice property.

Want to see a File/Folder browser dialog?

You may want to display a file (or folder) browser in one of the properties. This is pretty simple when you know that pre-built external editors exist. Microsoft does not give us many of these pre-built editors. I am only aware of FileNameEditor, the FolderNameEditor, the AnchorEditor, and the DockEditor. This time we need to use the Editor attribute to specify which editor to use:

```
<Category("Browser"), _
 Editor(GetType(FileNameEditor), GetType(UITypeEditor))> _
Public Property FileProperty() As String
    Get
        Return mFileProperty
    End Get
    Set(ByVal Value As String)
        mFileProperty = Value
    End Set
End Property

<Category("Browser"), _
Editor(GetType(FolderNameEditor), GetType(UITypeEditor))> _
Public Property FolderProperty() As String
    Get
        Return mFolderProperty
    End Get
    Set(ByVal Value As String)
        mFolderProperty = Value
    End Set
End Property
```

That's it! Since the editor is simply returning a string, your private member that is holding the value also need to be declared as a string.

Implementing a Custom editor form

It's not easy! But it's possible. The trick is to create a new form, a class to declare the data type, a class to declare the converter, and finally to use the Editor attribute to associate the new editor to the property just like we did in the previous example (file/folder browser).

I will not go further on this topic. Download the demo of this column and have a look at the Description property of the DemoClass and at the fEditorLongString form and all the classes it contains. You should be able to understand what's going on. It's the same thing we did in the last few samples.

Want to sort the properties in a different order?

Yes it can be done and it is easy! What you need this time is to derive a class from the ExpandableObjectConverter, override the GetProperties function to give a array of string enumerating your properties in the order you want to see them appearing. Simple no? Most of the mechanics I just described is done in a generic class. Download the demo and have a look at the BaseSorter class. This class can be reused as-is into your own projects.

Figure 2: The final output

The important class for you is named DemoClassSorter. This class inherits the BaseSorter class (so that you don't have to re-implement the mechanics every time you want to sort the properties of a class). It also implement a constructor that initializes the SortOrder property to an array of string containing the list of properties in the order you want them to appear.

Here is the code of the DemoClassSorter:

```
Inherits BaseSorter

Public Sub New()
   SortOrder = New String() { _
      "PastaProperty", _
      "SizeOfCanProperty", _
      "CanPrice", _

      "FolderProperty", _
      "FileProperty", _
      "Description" }
End Sub
```

As you can see, all properties are not listed. Other properties will be sorted alphabetically after the one you specified.

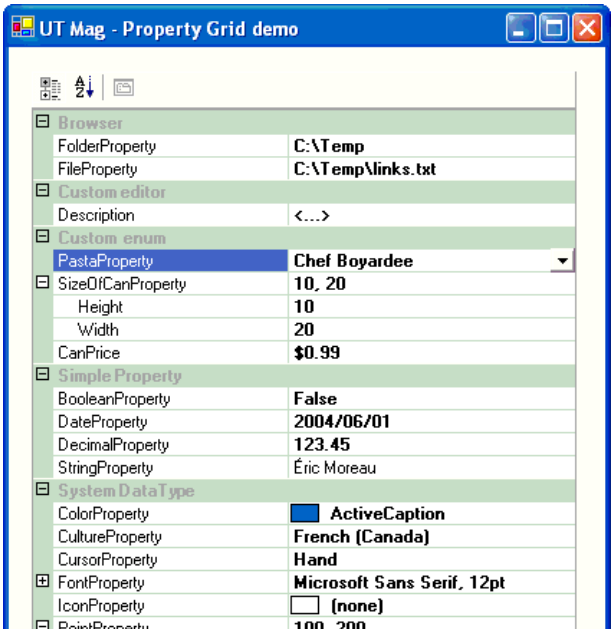
Now that your sorter class is defined, you need to tell the PropertyGrid to use the class to sort the properties. Guess what: this is done using another attribute. But this time, it is an attribute of the class:

```
<TypeConverter( _
   GetType(DemoClassSorter) _
   )> _
Public Class DemoClass
```

Conclusion

You can go even further customizing the property grid. Take a look at an article on MSDN: Getting the Most Out of the .NET Framework PropertyGrid Control that explains other customization of the PropertyGrid.

As you have seen, this control can be very

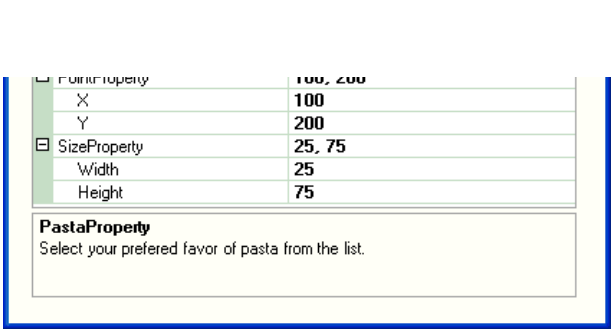


powerful if you dare to add some classes around it. But you will soon discover that many classes (like the CurrencyConverter class and the BaseSorter class) can be reused as-is form project to project.

Have you found where you will add it to your own projects?

I hope you appreciated the topic and see you next month.

Source code



More articles from this author

No	Title	Date
1.	A custom MessageBox	January 2006
2.	A Folder Browser dialog (in Windows Forms)	April 2003
3.	Adding plug-ins to your applications	August 2005
4.	An Action List component	August 2007
5.	Applications Settings in VB.Net	May 2003
6.	AppSettings revisited	March 2007
7.	Code diagnostic (an article on tracing and debugging)	November 2003
8.	Colors of disabled controls	July 2005
9.	ComboBoxes in Windows Forms DataGrids	January 2003
10.	Common dialogs	June 2003
11.	Compression in the .Net Framework 2.0	February 2007
12.	Creating help files and linking them to a VB application	December 2004
13.	Creating your own Windows Custom Control	March 2003
14.	Crystal Reports – Part II	October 2006
15.	Drag and Drop in VB.Net	February 2004
16.	Embedding a font into an application	February 2008
17.	Extender providers	October 2004
18.	Extending the My namespace	July 2007
19.	Feeding Crystal Reports from your application	September 2006
20.	Folders synchronization using the System.IO namespace	March 2006
21.	Fun with MDI forms	September 2003
22.	Handling an offline feature	October 2005
23.	Interfacing the Windows Task Scheduler	August 2004
24.	Localization	July 2004
25.	MARS and Asynchronous ADO.Net	November 2006
26.	Microsoft Visual Basic Power Packs 3.0	April 2008
27.	Monitoring your applications using custom performance counters	February 2006
28.	Multi columns ComboBox	February 2005
29.	My thoughts on LINQ	May 2007
30.	Object Serialization in VB.Net	July 2003
31.	Running assemblies from a shared folder	April 2004
32.	Scrolling text	December 2005
33.	Setting a master/detail relationship between two ComboBoxes	January 2007
34.	Setting Windows default printer	March 2005
35.	Strings, Strings, Strings	August 2006
36.	The (incomplete) TabControl	May 2004
37.	The BackgroundWorker component	December 2006
38.	The Clipboard class	August 2003
39.	The DateTimePicker control	January 2005
40.	The ErrorProvider Control (in Windows Forms)	February 2003
41.	The FileSystemWatcher component	April 2005
42.	The My Namespace	June 2007
43.	The NotifyIcon class	January 2004
44.	The Process component	December 2003
45.	The StatusBar control	September 2004
46.	The Strippers' club	May 2006
47.	The TextBox and the MaskedTextbox controls	June 2006
48.	The Toolbar control	November 2004
49.	The Treeview control	April 2006
50.	The Treeview control - Part II	July 2006
51.	The WebBrowser control	April 2007
52.	This month's profile: Eric Moreau	June 2001
53.	Using MS-Agent in a VB.Net application	June 2005
54.	Using Reflection to spy forms' content	September 2005
55.	Using System.Net.Mail	September 2007
56.	Using the ConnectionStrings section of the configuration file	April 2009
57.	Using the registry from a VB.Net application	October 2003
58.	Using The WebRequest object to retrieve Yahoo stock quotes	March 2004
59.	VB.Net and the Google API	May 2005
60.	Visual Basic and VB.Net FAQs	July 2001
61.	Visual Basic and VB.Net FAQs	June 2001
62.	What's new in VS2005 for Windows Forms Developers?	November 2005
63.	Wiki-based help system	March 2008