

Using the PropertyGrid in .Net

Introduction

The PropertyGrid offers you the possibility to let users edit defined properties of any given class as easy as you can use the Object Inspector in the .Net IDE. This tool will accept one, or more, classes as input and will display the public properties sorted by category, name or whatever kind of sorting you want to add.

Some of the most used applications of the PropertyGrid is to offer an easy way to change to settings of your application. For instance you create a class with all the settings as public properties and give the PropertyGrid an instance of this class.

Simple Example.

Since the PropertyGrid is not in the Toolbox of Visual Studio you will have to add it to the code manually. When this is done and you have added it to one of the Control containers in your GUI you can modify it using the IDE. In this example we'll use Visual Basic, but it is easy portable to C# or C++. We will be creating a small setting class that contains our settings, the second class will be a GUI to display our settings. This GUI will contain nothing other then the PropertyGrid.

```
Namespace MyApp1
Public Class MySettings
    Public Event AdvancedSettingChanged(ByVal oldValue as Boolean, ByVal
newValue as boolean)
    Public event RegNameSettingChanged(ByVal oldValue as String, ByVal newValue
as String)

    Private _Advanced as boolean
    Private _regName as string

    Public Property ShowAdvanced() as Boolean
        Get
            Return Me._Advanced
        End Get
        Set (Value as Boolean)
            RaiseEvent AdvancedSettingChanged(Me._Advanced, value)
            Me._Advanced = Value
        End Set
    End Property

    Public Property RegisteredName as String
        Get
            Return Me._regName
        End Get
        Set (Value as String)
            RaiseEvent AdvancedSettingChanged(Me._regName, value)
            Me._regName
        End Set
    End Property
End Class
End Namespace
```

The above class is a very simplified settings class that raises an event as soon as the value of a property is changed. It is your task to handle this event and process the information being sent. It may be a bit simplistically, but it does show how it works. No for our GUI that will be easier.

```
Namespace MyApp1
  Public class MySettingWindow
    Inherits System.Windows.Forms.Form

    Public Sub New()
      MyBase.New()
    End Sub

    .....
    Some declarations done by the IDE Designer
    .....

    Friend WithEvents myProperty As System.Windows.Forms.PropertyGrid
    Private Sub InitializeComponents()
      Me.myProperty = new System.Windows.Forms.PropertyGrid
      Me.SuspendLayout()

      Me.myProperty.Location = new System.Drawing.Point(0,0)
      Me.myProperty.Dock = System.Windows.Forms.DockStyle.Fill

      Me.Controls.Add(myProperty)
      Me.ResumeLayout()
    End Sub

    Public Sub setSettingClass(ByRef [Class] As MySettings)
      ' Make the propertygrid display the class
      Me.myProperty.SelectedObject = [Class]
    End Sub
  End Class
End NameSpace
```

Now the above form is by no means a good example of how you should make your form, but again it does show you how to initialize and use the PropertyGrid.

Advanced use

If you are interested in using the PropertyGrid and having a bit more options then you usually have then look into the TypeConverter and the UITypeEditor. These can be set above any property to indicate to the PropertyGrid how to deal with the property when the user wants to change or edit it.

Of course your properties will not look neat with the code in the above example. To get it a bit neater you could add some of the code that is below to it.

```
<System.ComponentModel.Description("This is my properties description"),  
System.ComponentModel.Category("Normal Settings") > _
```

With the above code you can easily modify the description, category or the type of editors or converters that are to be used. To learn more on these settings read the tutorial on classes in VB.Net or C#, please note that a property in VB is equivalent to a `__declspec(property(get = ..., put = ...))` in C++.

Customizing the editor

In order to be able to use the editor for your property it must inherit from `System.Drawing.Design.UITypeEditor`. In designing your own editor you must keep in mind that the following routines must be overridden.

- Overloads Public Overridable Function EditValue(ITypeDescriptorContext, IServiceProvider, Object) As Object
- Overloads Public Overridable Function GetEditStyle(ITypeDescriptorContext) As UITypeEditorEditStyle

The first one of the two tells the PropertyGrid what to do when the GridItem is clicked on, you probably want to create a new form in this function and return the value that is filled out in this form. The second one indicates the type of symbol that is to be displayed next to the property. This can be a drop-down arrow or a '...' button.

In the example of the typeditor below we will be creating a general OpenFileDialog to retrieve a file name.

What if I need something more dynamically?

On the previous page I described how you can easily modify the already existing PropertyGrid item that comes with the .Net Framework. However this still has some limitations as it does not allow you to dynamically decide what to include and what not, but more importantly when to include certain properties and when not to. So on this page I will be discussing a method that allows complete dynamic building of the property grid.

Note: If you are a first timer this part of the tutorial might be a bit complicated to follow, I'll try to keep it as simple as I can though.

The quick answer.

Bad luck, the PropertyGrid as it is delivered with the .Net Framework does not support any dynamic building of the properties. However there is a way to re-program some parts of the .Net Framework, which we will be doing in this part of the tutorial. So lets start out with making a list of objects we will be looking at.

- PropertyGrid, this is the actual Object inheriting from the Control base class and can be used to display properties.
- PropertyTag, inheriting from this class enables us to create personalized tab pages for the property grid.
- PropertyDescriptor, class that (as the title says) describes how the property grid should deal with the property.

The last two classes in the list work on the background and will play a big role in our implementation of making a completely dynamic property grid.

The inner workings.

For the actual displaying of properties the property grid uses one or more PropertyTab classes. How many and which one's are up to you as the programmer. If you just want to make some new tabs that give you access to different properties, or like we will be doing dynamic properties, and add them keeping the default tab you don't need to alter the PropertyGrid. However we want the default tab to be the dynamic one, so we will have to tinker with the PropertyGrid. Once you place an object in either SelectedObjects or SelectedObject property of the PropertyGrid it will start querying the active tab it has to draw all the properties.

During these queries it will call the [GetProperties](#) function of the PropertyTab. So when we are writing our dynamic version we will put some focus on this function, disabling the default fetching of properties. Normally this function will call the GetProperties function on the object that's put selected in the PropertyGrid. This will result in the returning of a list containing all the public properties that the selected object has.

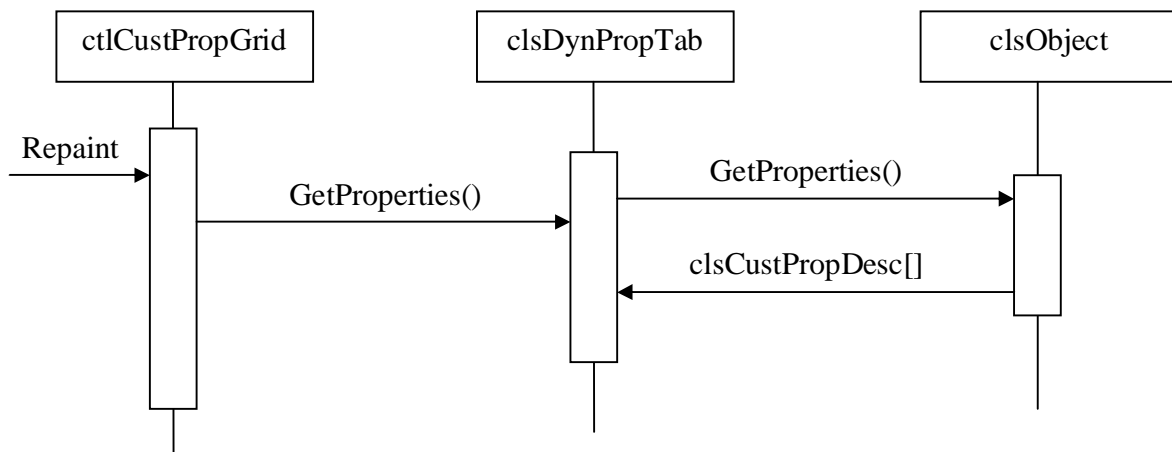


Figure 1: Dynamic generation on repaint

The new situation that we will be creating is the following”

1. Object gets selected by either SelectedObject or SelectedObjects
2. PropertyGrid receives a repaint from the Form
3. PropertyGrid calls GetProperties of the active PropertyTab
4. PropertyTag calls the GetProperties function implemented in the object selected.

As you can see most of the actions in our plan are similar to the ones in the original implementation, however we will let our own written class decide what properties to return.

As you can see in figure 1 there’s one more detail I haven’t yet mentioned and that’s the custom property descriptor. First let me try and explain why it is needed to create a custom property descriptor class. By default the property descriptor is a very good class that keeps all the basic information for one property of an object. So we won’t be changing these features, but remember the little annoyance of not being able to do it dynamically. Well I opted to create a new property descriptor which allows us to use normal functions (or VB subs) to access private data members of classes. Why functions and not properties (for definition see part one of this tutorial) is easy to explain. I have my roots in C++ and in that language there is no such thing as a property, well not the same as in VB anyway. You declare one as two functions, one for getting the information and one for setting. So why stray away from something you know and love.

Implementing our solution.

Since I started this tutorial out in VB.Net I will be using the same language here, but as before its easy to port to different languages.

Basics

Lets start with the a basic interface which forces us to implement the functions and properties needed for the dynamic building of the propertygrid. This interface is rather simplistic, but keep in mind you might want to add your own stuff soon enough.

```

Public Interface iBaseObject

    Function GetProperties() As
System.ComponentModel.PropertyDescriptorCollection

End Interface

```

The property descriptor

The next class to look at is our custom property descriptor, since this will be the most important component of the solution. This class will inherit from System.ComponentModel.PropertyDescriptor and will override most of its basefunctions. For the beginners among you this is the more complex part I talked about earlier.

Lets begin with writing a new constructor that will provide us with all the information we will need for accessing the information of the property. The head information of this function is rather complex with a lot of optional parameters, please bear with me I will explain the why and what after showing you the constructor.

```

Public Sub New(ByVal Name As String, _
                ByVal ValueType As Type, _
                ByVal OnComponent As Interfaces.iBaseObject, _
                Optional ByVal UseNameForGetSet As Boolean = True, _
                Optional ByVal Setter As String = Nothing, _
                Optional ByVal Getter As String = Nothing, _
                Optional ByVal Params As Parameter() = Nothing, _
                Optional ByVal Attrs As System.Attribute() = Nothing)

    MyBase.New(Name, Attrs)

    Me.tpValueType = ValueType

    ' Conditionally set the getter and setter
    If (UseNameForGetSet) Then
        strSetter = "Set" & Name
        strGetter = "Get" & Name
    Else
        strSetter = Setter
        strGetter = Getter
    End If

    ' Set read / write flags
    Me.blnCanWrite = Not CType(OnComponent, _
        Object).GetType().GetMethod(strSetter, intFlags) Is Nothing
    Me.blnCanRead = Not CType(OnComponent, _
        Object).GetType().GetMethod(strGetter, intFlags) Is Nothing

    ' Set the param array
    If Not (Params Is Nothing) Then
        ReDim Preserve Me.prmArray(Params.GetUpperBound(0))
        Array.Copy(Params, Me.prmArray, Params.GetUpperBound(0))
    Else
        Me.prmArray = New Parameter() {}
    End If

    If Me.blnCanRead Then

```

```
' Retrieve the default value
Me.objDefaultValue = Me.GetValue(OnComponent)
End If
End Sub
```

So what are we doing, well tell me. Its actually quite simple, to enable the descriptor to retrieve and set the property we need the name of the two functions (I now we can also do it with IntPtr but lets keep it simple.) The exact reason will become more clear in a little bit. To support all of the functionalities of the property grid I also retrieve the value of the object as it is when it was passed to the property grid. We also need to set the flags indicating wether the property is read and or write only, if we don't do it now its impossible to figure out later on. The deal with the *Params* will be explained a bit later on.

One more thing I need to give you to use the above code is the content of the *intFlags* variable. In order to be able to retrieve methods I've chosen to be case insensitve and only include public or instance members. This leads to the following value for intFlags:

```
Reflection.BindingFlags.Public Or _
Reflection.BindingFlags.Instance Or _
Reflection.BindingFlags.IgnoreCase
```

Now keep in mind that eventhough it is an enumerate these are nothing more then integers that use binary methods for setting and getting the actual value.

Note: Please keep in mind that the extracted value may not be the default value you wish to have and may lead to somewhat unexpected defaultvalues, but it's a short cut.

Now that we have a constructor I will highlight some of the other functions that will need to be implemented for successfully completing the class:

- *CanResetValue*, returns a boolean that tells the property grid if this property has a default value.
- *ComponentType*, used to retrieve the *System.Type* of the component the property belongs to. (So not the type of the property itself)
- *PropertyType*, retrieves the property type that should be expected
- *IsReadOnly*, tells the property grid if it should allow modification of the value or not
- *ResetValue*, will be called when the user chooses to reset the value to the default
- *SetValue*, you'll proberly be able to guess this one aint you
- *GetValue*, and yet another one that's speaks for itself

The last three will be the most important to implement correctly, since these will actually interact with our objects.

Ok so far for the boring stuff, lets go on with programming the class. I'll start the implementation in the same order as they are stated above. To speed along the tutorial a bit I will state the first four functions in one part, I choose to do this because of there simplicity.

```
Public Overrides Function CanResetValue(ByVal component As Object) As _
Boolean
Return Not Me.objDefaultValue Is Nothing
End Function
```

```

Public Overrides ReadOnly Property ComponentType() As System.Type
    Get
        Return GetType(Object)
    End Get
End Property

Public Overrides ReadOnly Property PropertyType() As System.Type
    Get
        Return Me.tpValueType
    End Get
End Property

Public Overrides ReadOnly Property IsReadOnly() As Boolean
    Get
        Return Not Me.blnCanWrite
    End Get
End Property

```

Well I kept my end of the deal, these functions are as simple as I can keep them and mostly speak for themselves. However I will highlight that the CanResetValue is not usable in any professional application. You will probably want to devise your own method of determining whether a variable has an initial value or not.

So I guess the next function to implement is CanResetValue. This one is actually pretty easy and goes something as follows:

```

Public Overrides Sub ResetValue(ByVal component As Object)
    If Not (Me.objDefaultValue Is Nothing) Then _
        Me.SetValue(component, Me.objDefaultValue)
End Sub

```

Like I stated before this one was easy. All we need to do is check to see if there is actually a default value to begin with. If this is the case then we call our own set function to set the default value.

The next one is a bit more tricky it's the function to set the properties value. It goes something like this. We use the type information of the object to retrieve the function with the same name as the setter that we have on record. Once found we will generate a parameter array and call the function. How this parameter array is generated is besides the point for now, I'll get to that later on.

```

Public Overrides Sub SetValue(ByVal component As Object, ByVal value As Object)
    If Me.blnCanWrite Then
        Try
            ' Get the function pointer
            Dim mtbInfo As System.Reflection.MethodBase = _
                component.GetType().GetMethod(strSetter, intFlags)

            If mtbInfo Is Nothing Then _
                Throw New MethodAccessException(strSetter & " not found.")
        End Try
    End If
End Sub

```

```

    Dim params As Object = Me.GenereteParams(mtbInfo)
    params(Array.IndexOf(params, Nothing)) = value

    mtbInfo.Invoke(component, params)
Catch ex As Exception
    MessageBox.Show(ex.Message, "Warning", MessageBoxButtons.OK, _
        MessageBoxIcon.Warning)
End Try
End If
End Sub

```

This might look like magic, and it is to an extent. However there are some small things to keep in mind. Never have two functions with identical names as the GetMethod call will then return an error that there are ambiguous methods.

The last function of this class is similar to the previous one, so I won't go in to great detail on the workings of it. The only huge difference is that we don't need to check where to fill in the value we're trying to pass to the object.

```

Public Overrides Function GetValue(ByVal component As Object) As Object
    If Me.blnCanRead Then
        Try
            Dim mtbInfo As System.Reflection.MethodBase = _
                component.GetType.GetMethod(strGetter, intFlags)

            If mtbInfo Is Nothing Then _
                Throw New MethodAccessException(strSetter & " not found.")

            Dim params As Object = Me.GenereteParams(mtbInfo)
            Return mtbInfo.Invoke(component, params)
        Catch ex As Exception
            MessageBox.Show(ex.Message, "Warning", MessageBoxButtons.OK, _
                MessageBoxIcon.Warning)
        End Try
    End If
End Function

```

Ok now the bit more complex part, namely the building of the parameter list that the function expects. Do you still remember the parameter 'params' in the constructor, well we're going to use this here for building the parameter array. Let me first start of by explaining why I opted for this method.

Lets start daydreaming, in a wonderful world you have three children and want to give the world access to this children. There is only one problem you don't have the money to pay for three doors, so you tell the people that they have to call the name (or indice) of the child they want to see when coming to that one door. Now back to the real world, our problem is similar with arrays. If you have a dynamic array you can't build one function for every indice so you want the property descriptor to tell your one function which of the items in the array it wishes to modify or retrieve.

To make this possible I've made a small class that inherits from the default PropertyDescriptor class in the .Net foundation. Why, because I am too lazy to write much code and this class is

already available, pethedic I know. So below is the implementation of how each parameter is defined.

```
Public Class Parameter
    Inherits Reflection.ParameterInfo

    Private objValue As Object

    Public Sub New(ByVal Name As String, ByVal Value As Object)
        MyBase.New()

        Me.objValue = Value
        MyBase.NameImpl = Name
    End Sub

    Public Shadows Function Equals(ByVal objB As Object) As Boolean
        If Not (objB.GetType Is GetType(Reflection.ParameterInfo)) Then _
            Return False

        Return Me.Name = CType(objB, Reflection.ParameterInfo).Name
    End Function

    Public Function GetValue() As Object
        Return Me.objValue
    End Function

End Class
```

I'll let you all in on a little secret as to why I implemented the Equals function. When you have an array of System.Array then you have a neat little function called *IndexOf* returning the position in the array of an element. When looking in the MSDN you'll soon learn it will call the Equals function of each object in the array if it is implemented. (Note that using a ArrayList will not do, this class is stupid and only uses the System.Object.Equals function)

So now on the generation of our parameter list, which is why we did all of this in the first place. It's a rather small but very efficient function that loops through all the parameters that the function has matching them up with the list given during construction. If no match is found and the parameter is optional its forgiven and Reflection.Missing.Value is entered as value. In all other cases the function can no longer be used as not all parameters can be filled out.

```
Private Function GenereteParams(ByVal MethodInfo As Reflection.MethodBase) _
    As Object()
    Dim param As Reflection.ParameterInfo
    Dim a_Retval As New ArrayList

    For Each param In MethodInfo.GetParameters()
        ' Find a match in the param array
        Dim intPositionInArray As Int32 = Array.IndexOf(Me.prmArray, param)

        If (intPositionInArray < -1) Then
            a_Retval.Add(Me.prmArray(intPositionInArray).GetValue())
        Else
            If param.IsOptional Then
```

```

        a_Retval.Add(Reflection.Missing.Value)
    Else
        If (param.Name.ToLower <> "value") Then _
            Throw New Reflection.TargetException("Parameter unspecified for"& _
                " target function")

        a_Retval.Add(Nothing)
    End If
End If
Next

Return a_Retval.ToArray(GetType(Object()))
End Function

```

So that's basically it for the property descriptor that we will be using instead of the default PropertyDescriptor class. Let's continue on to the easier parts of the tutorial and start implementing our custom PropertyTab for displaying our properties.

The Property tab

This class is simple and straight forward, so I won't spend too much time explaining it in detail not per function. The most important thing it does is whenever it receives a GetProperties call to query the component for a list of all the properties. Of course this can only be done if the specified function in iBaseObject is present, if not it will ask the base class to complete the call.

```

Public Class clsPropertyTab
    Inherits Windows.Forms.Design.PropertyTab

    Public Overloads Overrides Function GetProperties( _
        ByVal component As Object, _
        ByVal attributes() As System.Attribute) As _
        System.ComponentModel.PropertyDescriptorCollection
        Dim mtbProperties As Reflection.MethodBase = _
            component.GetType.GetMethod("GetProperties")

        If (mtbProperties Is Nothing) Then
            Return MyBase.GetProperties(component)
        Else
            Return mtbProperties.Invoke(component, Nothing)
        End If
    End Function

    Public Overrides ReadOnly Property TabName() As String
        Get
            Return "Properties"
        End Get
    End Property

    Public Overrides ReadOnly Property Bitmap() As Drawing.Bitmap
        Get
            Return New Bitmap(16, 16)
        End Get
    End Property
End Class

```

If you have difficulty with the tab showing up in the property grid please verify that the *bitmap* property is behaving normally as this is the most likely cause. It's a pain in the butt, but for some reason the tab might not work if this property is omitted.

The PropertyGrid

The implementation of the propertygrid is, if that is even possible, simpler than the propertytab. Though this propertygrid is dynamic you may want to expand its functionality with searching for specific items to expand. However since this is not in the scope of this tutorial I will not take any documentation nor code needed for this in this article.

I was not lying when I said this class would be easy, we only have to override the function CreateTabPage, this function will be expected to return a tabpage that will be displayed in the propertygrid.

```
Protected Overrides Function CreatePropertyTab( _  
    ByVal tabType As Type) _  
    As System.Windows.Forms.Design.PropertyTab  
    Return New clsPropertyTab  
End Function
```

Well that's just about it for this tutorial, was a bit longer then I had first anticipated. Hopefully this will help some of you that found the same shortcomings in the .Net Framework. If you found any mistakes or problems with the tutorial don't hesitate to contact me about it.