

# Using the IExtenderProvider in Visual Basic .NET

## By Simon Broadhead

---

### Introduction

The VB.NET has added a great, *great* deal of functionality over its predecessor VB6; true object-oriented programming being the biggest addition. In this tutorial, I'm going to explain one of the coolest (in my opinions) uses of OOP included with .NET: The IExtenderProvider interface.

The IExtenderProvider that is included in the .NET framework is an interface that allows for the expansion of other controls. The interface allows you to add your own custom properties to other controls without the need to create subclasses of these controls, or any code on the form at all. By creating a component that implements the ExtenderProvider, you can expand other controls greatly and expand other controls with an ease that VB6 couldn't even dream of having. In this tutorial I assume that, since this is a pretty advanced concept, you already have a firm grasp on the language of .NET, and the concepts of OOP (inheritance, interfaces, etc).

In this particular tutorial, we will be developing a rather simple and somewhat pointless component; it will extend all TextBoxes with a 'HoverColor' property, and when you move your cursor into that TextBox, it will change the background color to the HoverColor. Now, if you're ready, let's get started.

## Putting the Interface Into Your Component

Implementing the interface into your component is actually quite easy. The first thing you need to do is, obviously, implement the `IExtenderProvider` into your component. The second thing you need to do is add the **CanExtend** function, implemented from the `IExtenderProvider` interface again. What this function does is determine whether or not a specific control should be extended or not. Generally, this is done by a **TypeOf** check, to look at the type of control; for example, you might want to only extend TextBoxes, so you would use the **TypeOf** function to see if the control being passed to this function is a `TextBox` or not. If it is, you return `True`, otherwise return `False`. Whenever you return `True`, the control being checked will be extended, otherwise it won't be.

Before I show you any code, I should point out that I have Imported the following namespaces:

```
System.ComponentModel
System.Windows.Forms
System.Drawing
```

Also, with my copy of VB.NET at least, the `System.Windows.Forms.dll` and `System.Drawing.dll` references are not added by default with new Class projects, so you may need to right click on your References option in the Solution Explorer and add them from the list. To check, just expand the item in the Solution Explorer tree that says References and look to see if they are in the list of referenced DLLs.

Anyway, Here is a basic class which implements the interface:

---

```
Public Class TestExtender
    Inherits Component
    Implements IExtenderProvider

    Private Function CanExtend(ByVal ctl As Object) _
        As Boolean Implements IExtenderProvider.CanExtend

        If TypeOf ctl Is TextBox Then
            Return True
        End If
    End Function
End Class
```

---

So now you have a basic class which extends the `TextBox` control. With what? Nothing yet, but the class's basic structure is there. On to the next section.

## Adding Properties

Okay, now you have a class built that extends TextBox controls with... nothing. There are no properties yet. That is the next part. If you have worked with classes for any amount of time, you've probably come across the use of attributes; that is, special commands that go between < and >, and are placed before the beginning of your class/property/method, which modify the way it works. In this case, we will need to use the **ProvideProperty** attribute. So, change your opening lines to this:

---

```
<ProvideProperty("HoverColor", GetType(TextBox))> _  
Public Class TestExtender
```

---

That will provide a property, using the IExtenderProvider, for all controls of type TextBox. That is just the definition of the property, however; the prototype, if you will. Now we need to add the Get and Set code.

When using an IExtenderProvider, properties are not actually defined as properties; they are actually methods which do the same things that Property Get and Set blocks would do. The only part of this component that really resembles a new property is in the Windows Forms Designer's property window.

Anyway, to create the property, you need to have two different methods; **SetHoverColor**, and **GetHoverColor**. The former being defined as a Sub, and the latter being defined as a Function, with the return type Color. I recommend using a HashTable for storing the values for each control, because they allow for an Object->Value pair. So, in your class, add a \_backgroundColors member variable (HashTable).

---

```
Private _backgroundColors As New Hashtable()
```

---

The Get and Set routines are really no different from how you might code a Property.

---

```
Public Sub SetHoverColor(ByVal ctl As TextBox, _  
                        ByVal val As Color)  
    If Not _backgroundColors.Contains(ctl) Then  
        _backgroundColors.Add(ctl, val)  
    Else  
        _backgroundColors(ctl) = val  
    End If  
End Sub  
  
Public Function GetHoverColor(ByVal ctl As TextBox) As Color  
    If Not _backgroundColors.Contains(ctl) Then  
        Return Color.Black  
    Else  
        Return _backgroundColors(ctl)  
    End If  
End Function
```

---

## Adding Real Functionality

And I use the term loosely; the “functionality” we’re going to be adding is nothing more than causing a textbox to “light up” when you move your mouse over it by changing the background. To do this, we need to hook up a few event handlers via the **AddHandler** function for the `MouseEnter` and `MouseLeave` functions of the textboxes. The only problem is that there is no-where to put the code so that it will modify **all** of the `TextBox`s on the form; remember that if we leave a `TextBox`’s `HoverColor` property default, the **SetHoverColor** method will never be called, so if we put code there, it won’t be called for the `TextBox`’s that are left default. So, we need to make a bit of a hack. We need to create a second extended property for the textbox, but hide it from the property list... we’ll call it “`DoBackground`”. Now, in the **GetDoBackground** routine that goes along with this property, we will need to return `True`. Always. I’ll explain why.

When a control is placed on a form, and its properties are changed, the changes are serialized in the code under the form (expand the ‘Windows Forms Designer Generated Code’ region to see it). However, the properties on the control that are *not* changed do not get serialized; they will be left as their default values. In the case of this control, the ‘`DoBackground`’ property is ‘`False`’ by default, but by returning ‘`True`’ in the **GetDoBackground** sub, we are fooling the designer into thinking that the property *has* changed, and so the **SetDoBackground** call is serialized into the form for every `TextBox` on the form.

So, let’s add the property into the class attribute. Change it to this:

---

```
<ProvideProperty("DoBackground", GetType(TextBox)), _  
ProvideProperty("HoverColor", GetType(TextBox))> _  
Public Class TestExtender
```

---

Now we add in the Set and Get routines.

---

```
Public Sub SetDoBackground(ByVal ctl As TextBox, _  
                           ByVal val As Boolean)  
    AddHandler ctl.MouseEnter, AddressOf tbMouseEnter  
    AddHandler ctl.MouseLeave, AddressOf tbMouseOut  
End Sub  
  
Public Function GetDoBackground(ByVal ctl As TextBox) As Boolean  
    Return True  
End Function
```

---

When **SetDoBackground** is called now (for every `TextBox` on the form), we need to hook up the event handlers, and as you can see in the code above, it is quite simple. You just give it the name of the event, and the **AddressOf** of the sub routine you wish to handle all events in; and that’s really all that’s left to do. On the next page is the code that will change the background colors.

---

```

Public Sub tbMouseOut(ByVal sender As Object, _
                    ByVal e As EventArgs)
    Dim tb As TextBox = DirectCast(sender, TextBox)

    'Restore the textbox background color
    tb.BackColor = SystemColors.Window
End Sub

Public Sub tbMouseEnter(ByVal sender As Object, _
                    ByVal e As EventArgs)
    Dim tb As TextBox = DirectCast(sender, TextBox)
    Dim clr As Color

    'We're using Color.Red as the default color, so if the
    'color hasn't been set, then just use red, otherwise use
    'the color that's been stored

    If _backgroundColors(sender) Is Nothing Then
        clr = Color.Red
    Else
        clr = DirectCast(_backgroundColors(sender), Color)
    End If

    'Set the background color
    tb.BackColor = clr
End Sub

```

---

So there you have it; it's got some problems, like if you had the background color set to something non-standard before, hovering over and then out of it will reset it to the default Windows color.

Anyway, compile the DLL and start a new Windows Forms project. Add the DLL to your toolbox (right-click on the toolbox, click Customize, .NET Framework Components, Browse... then find the DLL you compiled in the \bin\ directory of your TestExtender (or whatever the name of your solution was) directory. Add some TextBoxes to your form, and then add a TestExtender. Now, in the property window there should be a new property that isn't normally there:

**HoverColor on TestExtender1.** Change the color to something different on each of the textboxes and run the program. When you move your mouse over each of the textboxes, they should each change to the color that you specified in the HoverColor property. Pretty cool, eh?

There are many possibilities for developing components that use this. I myself have developed one which extends menus so that they have the ability to use the little icons in them. I'm sure you'll find something cool to do with it, too.

# The Full, Working Code

Just in case you can't get something working just right (could be a mistake on my part on writing the tutorial, too), here is the full text of the class file, that I **know** works.

```
<ProvideProperty("DoBackground", GetType(TextBox)), ProvideProperty("HoverColor",
GetType(TextBox))> _
Public Class TestExtender
    Inherits Component
    Implements IExtenderProvider

    Private _backgroundColors As New Hashtable()

    Private Function CanExtend(ByVal ctl As Object) As Boolean Implements
IExtenderProvider.CanExtend
        If TypeOf ctl Is TextBox Then
            Return True
        End If
    End Function

    Public Sub SetHoverColor(ByVal ctl As TextBox, ByVal val As Color)
        If Not _backgroundColors.Contains(ctl) Then
            _backgroundColors.Add(ctl, val)
        Else
            _backgroundColors(ctl) = val
        End If
    End Sub

    Public Function GetHoverColor(ByVal ctl As TextBox) As Color
        If Not _backgroundColors.Contains(ctl) Then
            Return Color.Red
        Else
            Return DirectCast(_backgroundColors(ctl), Color)
        End If
    End Function

    Public Sub SetDoBackground(ByVal ctl As TextBox, ByVal val As Boolean)
        AddHandler ctl.MouseEnter, AddressOf tbMouseEnter
        AddHandler ctl.MouseLeave, AddressOf tbMouseOut
    End Sub

    Public Function GetDoBackground(ByVal ctl As TextBox) As Boolean
        Return True
    End Function

    Public Sub tbMouseOut(ByVal sender As Object, ByVal e As EventArgs)
        Dim tb As TextBox = DirectCast(sender, TextBox)

        'Restore the textbox background color
        tb.BackColor = SystemColors.Window
    End Sub

    Public Sub tbMouseEnter(ByVal sender As Object, ByVal e As EventArgs)
        Dim tb As TextBox = DirectCast(sender, TextBox)
        Dim clr As Color

        'We're using Color.Red as the default color, so if the color hasn't been set,
        'then just use red, otherwise use the color that's been stored
        If _backgroundColors(sender) Is Nothing Then
            clr = Color.Red
        Else
            clr = DirectCast(_backgroundColors(sender), Color)
        End If

        'Set the background color
        tb.BackColor = clr
    End Sub
End Class
```