Autodesk University 2007

Creating a Docking Palette for AutoCAD® with VB.NET

Mike Tuersley – Ohio Gratings, Inc.

CP205-2 This is a VB.NET translation of the C# example that ships with the ObjectARX SDK. This class demonstrates how to create a simple palette inside a dockable window within AutoCAD that contains the contents of a .NET user control created in VB.NET. Other functionality demonstrated includes docking states and associated reactor events, drag-and-drop from the Tool Palette window, and activating different Tool Palettes within the Tool Palette set. This is a must-have user-interface option in every developer's arsenal.

About the Speaker:

Mike works for Ohio Gratings Inc. as a Senior Programmer/Analyst and focuses on enterprise level automation using the latest Microsoft technologies. Before this, he was the founding member and senior lead application developer for RAND IMAGINiT's National Services Team, which focuses on providing customization services to meet customer visions. Mike has been customizing AutoCAD since release 2.5 and is a past columnist for "CADalyst" Magazine. For 6 years, he wrote the AutoCAD "Help Clinic" and the "CAD Clinic"; the latter focusing on teaching AutoCAD programming topics.

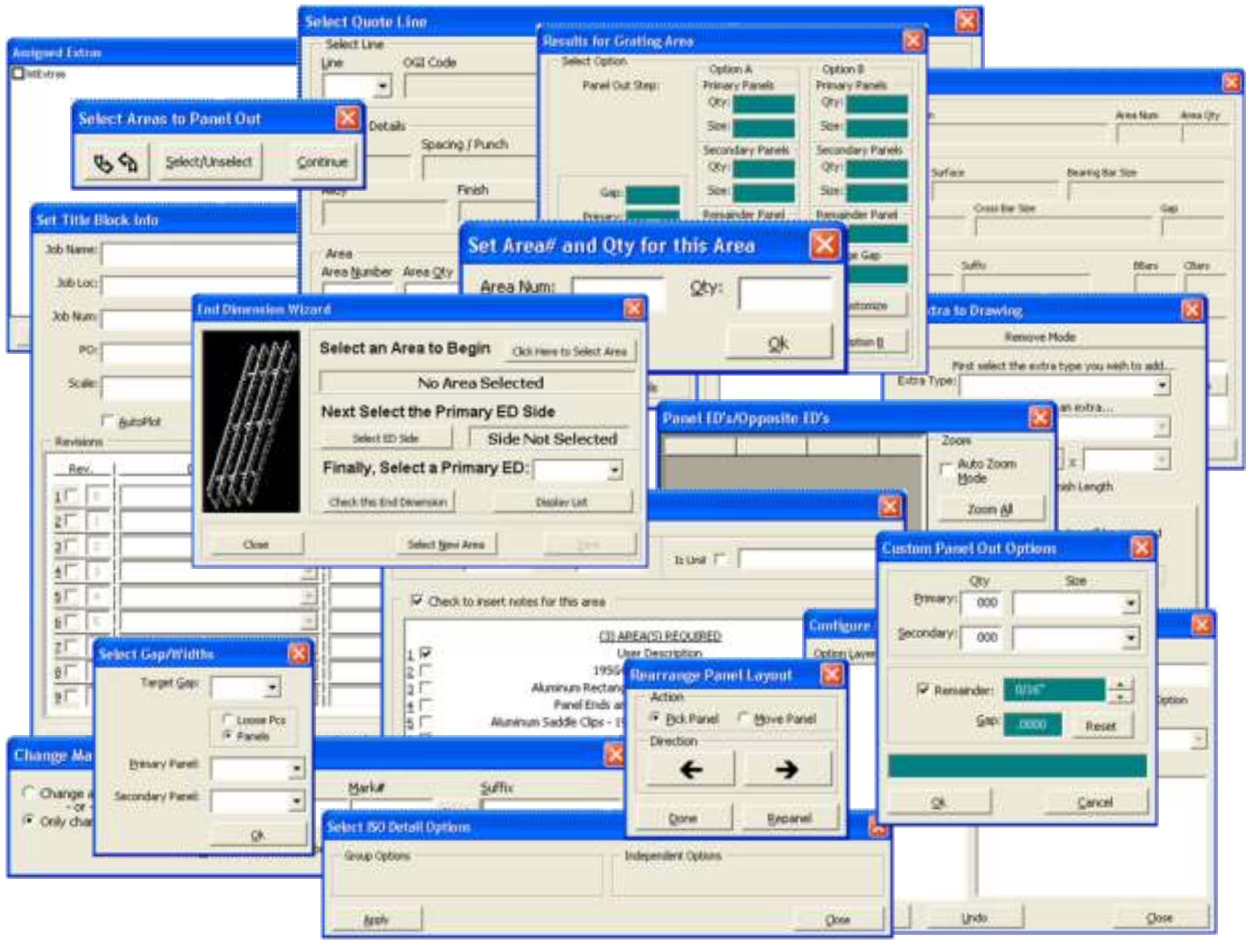
Email: mike.tuersley@hotmail.com

Table of Contents

Introduction

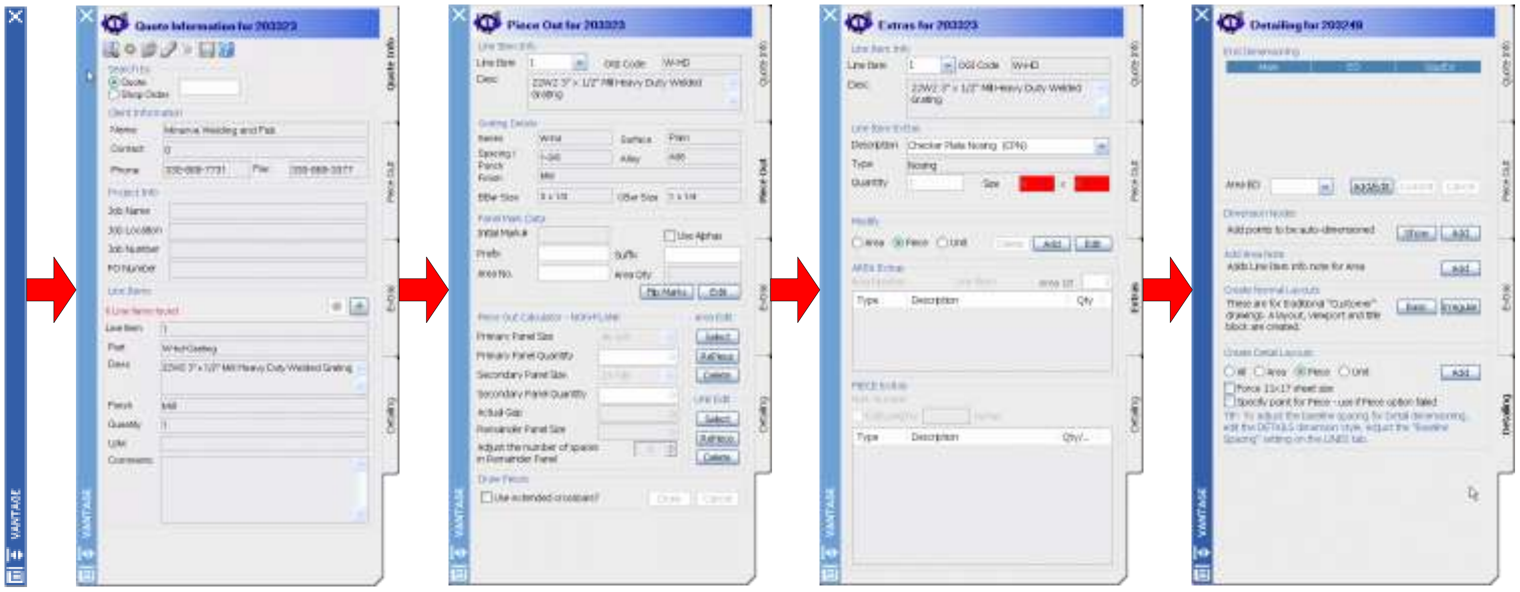
One of the key design goals of a developer, or hobbyist programmer, is to have their application add-in appear as if it is part of the original program; to seamlessly integrate it into the host environment. To accomplish this lofty goal, one must use the same user interface (UI) constructs wherever possible. Tool Palettes in AutoCAD-based add-ins are a perfect example and a must-have user-interface option in every developer's arsenal!

To expand upon why to use Tool Palettes, consider the following real world example. As a developer, I was tasked with the project of updating and consolidating an existing vba application for Ohio Gratings Inc. that had grown out of control after years of multiple people adding multiple things. The existing application consisted of dozens of forms along with the dozens of toolbar buttons to launch the forms:



Additionally, there were even more complementary AutoLISP and VBA commands all accessed by their own toolbar buttons. There were literally dozens of toolbar strips docked along the top, left and right sides of the AutoCAD drawing area that shrunk the drawing space by a good third.

As every developer whose actually used AutoCAD themselves knows, AutoCAD users value their drawing real estate. So to provide the users with more real estate as well as more functionality, the optimum UI choice was a *Tool Palette Set*. A tool palette set can auto hide to maximize the drawing space as well as contain multiple palettes to allow for organization of commands/concepts. It also made the processes easier to orchestrate by providing a work flow where the user interacts with the various tool palettes instead of bouncing around between different dialog boxes. Here is the final delivery that organized all the original clutter:



This new interface:

- Received data from web services tied to a *Progress* data base as well as XML data documents.
- Provided an interactive user experience by having the user select a button, then allowing them to interact with the drawing entities before posting the interactive data back into the tool palette.
- During drawing interaction or when not in use, collapsed (auto hid) to return the user's drawing real estate.

For this class, the tools required are simple:

- Any version of Microsoft Visual Studio including the Express[1] versions. For this class, Microsoft Visual Studio 2005 Professional will be used.
- Any version of AutoCAD or AutoCAD-based product 2005 or newer. For this class, AutoCAD 2008 will be used.

[1] If you are not familiar with how to setup the Express versions for debugging, please refer to an excellent blog article by Autodesk's Kean Walmsley: http://through-the-interface.typepad.com/through_the_interface/2006/07/debugging_using.html

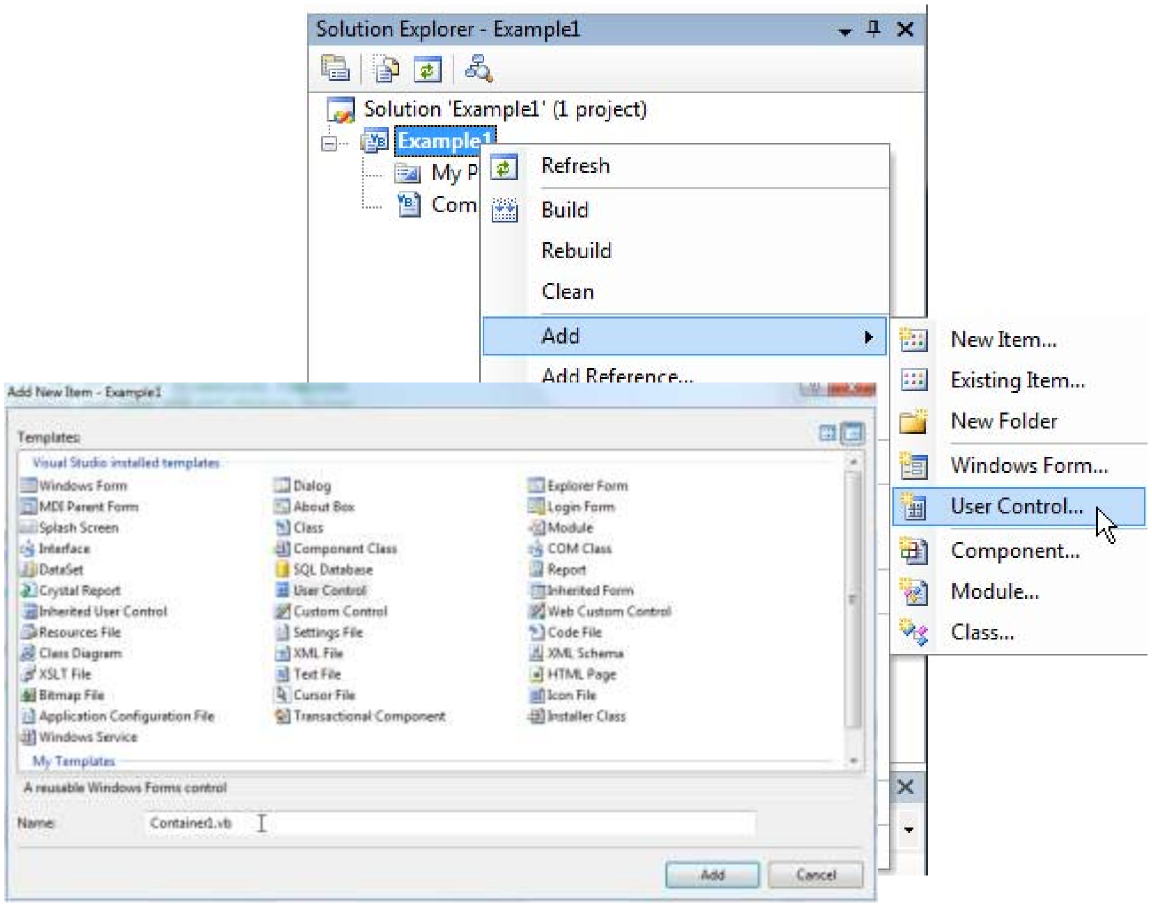
First Tool Palette

Every class must start somewhere so we'll begin with a variation of the classic, but corny, *Hello World Example* entitled *My First Tool Palette*.

1. To begin, start up Visual Studio 2005 and select to create a new Windows Class project (You can optionally use the ObjectARX Wizard if it is installed) and name it Example1
2. Rename Class1.vb to Commands.vb
3. Next make sure that you add references to the two AutoCAD managed libraries: *acmgd.dll* and *acdbmgd.dll*. Then add the following Imports statements:

```
Imports System.Runtime
Imports Autodesk.AutoCAD.Runtime
```

4. Now add a user control to the project and name it Container1



Name: Container1.vb

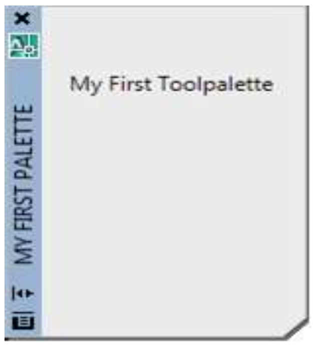
5. In the design view of Container1, add a Label and set its Text property to **"My First Tool Palette"**

6. Switch back to the initial class module (should be Commands) and add the following code:

```
 1' Define command
 2 <CommandMethod("TestPalette")> _
 3 Public Sub DoIt()
 4
 5    Dim ps As Autodesk.AutoCAD.Windows.PaletteSet = Nothing
 6    ps = New Autodesk.AutoCAD.Windows.PaletteSet("My First Palette")
 7    Dim myPalette As Container1 = New Container1()
 8    ps.Add("My First Palette", myPalette)
 9    ps.Visible = True
10
11 End Sub
```

To explain the code, line 2 is the basic command attribute assigned to the function to define it within AutoCAD. Next we need to instantiate a PaletteSet object which will hold the user control object. In Line 6, the PaletteSet is assigned a name and in Line 8 the user control is added to the PaletteSet along with a string assignment for its tab.

7. Compile and Debug the program. As with any DLL, use **NETLOAD** to load the library and then type **TestPalette** into the AutoCAD command prompt to launch your tool palette.



You should now see your first tool palette visible inside of AutoCAD. Play around with it; you should be able to drag and dock it around all four edges of the drawing window. If you accidentally close it, just rerun the *Example1* command. If you try to run the *TestPalette* command while the tool palette is visible, you should get another duplicate tool palette. This is not a desirable feature so return to your project and revise your code as follows:

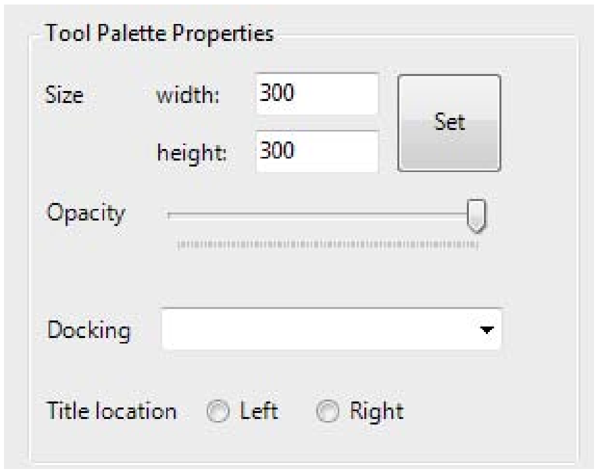
```
'ensure single instance of this app...
Friend Shared m_ps As Autodesk.AutoCAD.Windows.PaletteSet = Nothing

' Define command
<CommandMethod("TestPalette")> _
Public Sub DoIt()
  'check to see if paletteset is already created
  If m_ps Is Nothing Then
      'no so create it
      m_ps = New Autodesk.AutoCAD.Windows.PaletteSet("My First Palette")
      'create new instance of user control
      Dim myPalette As Container1 = New Container1()
      'add it to the paletteset
      m_ps.Add("My First Palette", myPalette)
  End If
  'turn it on
  m_ps.Visible = True
End Sub
```

Now if you rerun your code, you should be able to type in *TestPalette* as many times as you wish and only have one instance of your tool palette running at a time. That's it! Now let's continue looking at expanding on our knowledge of tool palettes.

Controlling Tool Palette Set Properties

To effectively control a tool palette, let's look at setting some of its basic properties:

- Opacity
- Title Location
- Docking
- Size
- Restoring User Settings
- Turning on Standard Buttons
- Adding a Custom Icon



Back inside the code project, remove the label and add the following controls:

- 2 textboxes with labels
- 1 trackbar with a label
- 2 radio buttons with a label
- 1 command button
- 1 combobox with a label

OPACITY

Set the following TrackBar settings:

Minimum = 10
Maximum = 100
Value = 100

Then add this code to the TrackBar's ValueChanged event:

```
Private Sub TrackBar1_ValueChanged(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles TrackBar1.ValueChanged
    'adjust opacity
    Example1.Commands.m_ps.Opacity = TrackBar1.Value
End Sub
```

TITLE LOCATION

Add the following code to the RadioButton's CheckChanged event:

```
Private Sub RadioButton1_CheckedChanged(ByVal sender As System.Object,
    ByVal e As System.EventArgs) _
    Handles RadioButton1.CheckedChanged, RadioButton2.CheckedChanged
      'toggle title location
      Example1.Commands.m_ps.TitleBarLocation = _
        IIf(sender.text.Equals("Left"), _
            PaletteSetTitleBarLocation.Left, _
            PaletteSetTitleBarLocation.Right)
End Sub
```

SIZE

Add the following code to the CommandButton's Click event assuming the textboxes are named txtHeight and txtWidth:

```
Private Sub Button1_Click(ByVal sender As System.Object, _
  ByVal e As System.EventArgs) Handles Button1.Click
    'resize paletteset
    Dim hgt As Int16 = Example1.Commands.m_ps.Size.Height
    Dim wid As Int16 = Example1.Commands.m_ps.Size.Width
    If Me.txtHeight.Text.Length > 0 And _
      Not Me.txtHeight.Text.Equals(0) Then
        hgt = Convert.ToInt16(Me.txtHeight.Text)
    End If
    If Me.txtWidth.Text.Length > 0 And _
      Not Me.txtWidth.Text.Equals(0) Then
        wid = Convert.ToInt16(Me.txtWidth.Text)
    End If
    Example1.Commands.m_ps.Size = New Drawing.Size(wid, hgt)
End Sub
```

Notice how the call is made back to the initial class (Commands in this example) to set the size of the palette set? There are easier ways to accomplish this as discussed later in this document. Also, we are assuming that the user will type in numeric values for the height and width. If this was for an actual project, validation code would need added to verify this assumption and avoid the potential unhandled exception error.

DOCKING

To the ComboBox, add the following values to the Items property*: Bottom, Left, Right, Top*, and *Floating* or *None.* Then add this code to the ComboBox's SelectedIndexChanged event:

```
Private Sub ComboBox1_SelectedIndexChanged( _
   ByVal sender As System.Object, ByVal e As System.EventArgs) _
   Handles ComboBox1.SelectedIndexChanged
     'toggle docking
     With Example1.Commands.m_ps
        Select Case Me.ComboBox1.SelectedIndex
            Case Is = 0 'bottom
                .Dock = DockSides.Bottom
            Case Is = 1 'left
                .Dock = DockSides.Left
            Case Is = 2 'right
                .Dock = DockSides.Right
            Case Is = 3 ''top
                .Dock = DockSides.Top
            Case Is = 4 'float
                .Dock = DockSides.None
        End Select
     End With
End Sub
```

Notice, again, the call back to the initial class' global palette set variable as well as the fact that we are checking the SelectedIndex rather than the text property of the combo box. If you added the items in a different order, you will need to account for this.

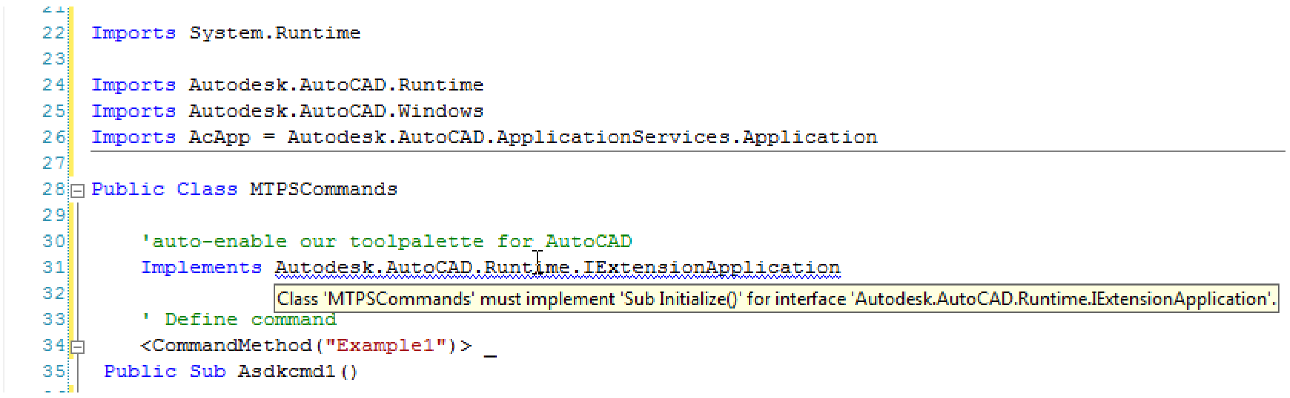
Now, build the project and debug it. Everything should work as expected except for the docking. If your tool palette is floating, you can select a docking option. After this, or if your tool palette is already docked, notice how you cannot select an item from the drop down list with your mouse cursor. This is one of the odd, little quirks with using tool palettes. The next section will explain how to get around this quirk.

RESTORING USER SETTINGS

Since we want our tool palette set to function like a part of AutoCAD, it needs to be able to remember the users last settings – if it was docked, where it was docked, its size, etc. To implement this is very simple. First, add the following statement to the Commands object:

```
    'auto-enable our toolpalette for AutoCAD
    Implements Autodesk.AutoCAD.Runtime.IExtensionApplication
```

Once you have added this line [two lines with the comment], Visual Studio should underline it as an error telling you that you need to implement Initialize and Terminate for the IExtensionApplication object:



```
Imports System.Runtime

Imports Autodesk.AutoCAD.Runtime
Imports Autodesk.AutoCAD.Windows
Imports AcApp = Autodesk.AutoCAD.ApplicationServices.Application

Public Class MTPSCommands

    'auto-enable our toolpalette for AutoCAD
    Implements Autodesk.AutoCAD.Runtime.IExtensionApplication
               Class 'MTPSCommands' must implement 'Sub Initialize()' for interface 'Autodesk.AutoCAD.Runtime.IExtensionApplication'.
    ' Define command
    <CommandMethod("Example1")> _
 Public Sub Asdkcmd1()
```

To do this, we will add them but will not add any code at this time:

```
  Public Sub Initialize() Implements IExtensionApplication.Initialize
    'add anything that needs to be instantiated on startup
  End Sub

  Public Sub Terminate() Implements IExtensionApplication.Terminate
    'handle closing down a link to a database/etc.
  End Sub
```

So after all of this, you will be able to tie into the palette set's events so you can add these two functions:

```
  Private Shared Sub ps_Load(ByVal sender As Object, _
   ByVal e As Autodesk.AutoCAD.Windows.PalettePersistEventArgs)
    'demo loading user data
    Dim a As Double = _
    CType(e.ConfigurationSection.ReadProperty("Example1", 22.3), Double)
  End Sub
  Private Shared Sub ps_Save(ByVal sender As Object, _
   ByVal e As Autodesk.AutoCAD.Windows.PalettePersistEventArgs)
     'demo saving user data
     e.ConfigurationSection.WriteProperty("Example1", 32.3)
  End Sub
```

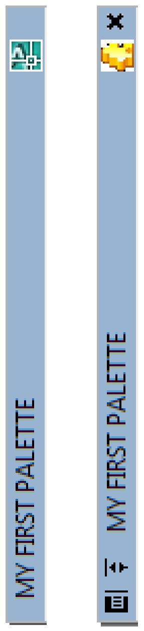
Now we need to change the initialization of the palette set by assigning it a GUID:

```
m_ps = New Autodesk.AutoCAD.Windows.PaletteSet("My First Palette", _
       New Guid("{ECBFEC73-9FE4-4aa2-8E4B-3068E94A2BFA}"))
```

The events for Load and Save will now automatically read and write the tool palette set's settings.

TURNING ON STANDARD BUTTONS

The tool palette set's title bar can have the standard Close, AutoHide and Options buttons turned on by setting the Style property. This code will turn on all of the buttons.



```
m_ps.Style = PaletteSetStyles.ShowPropertiesMenu Or _
             PaletteSetStyles.ShowAutoHideButton Or _
             PaletteSetStyles.ShowCloseButton
```

ADDING A CUSTOM ICON

The tool palette set's title bar can have a custom icon applied to it for that personalized look by setting the Icon property:

```
m_ps.Icon = GetEmbeddedIcon("Example1.gold_1_32.ico")

Private Shared Function GetEmbeddedIcon(_
  ByVal sName As String) As Icon
   'pulls embedded resource
   Return New Icon(System.Reflection.Assembly.
       GetExecutingAssembly.GetManifestResourceStream(sName))
 End Function
```

This code accomplishes assigning the Icon property by using an embedded icon so there is no need for a manifest or icon file that needs to be read by the program.

Quirks

As with any API, there are always little quirks or nuances that a programmer needs to identify and workaround – it wouldn't be programming if this never happened! So, here are the top three quirks when programming tool palettes:

Combo Boxes

As identified in the last section, combo boxes behave differently depending on whether the tool palette is docked or floating. Luckily this is an easy fix. To fix, simply add the following code to the combo box's DropDown event:

```
Private Sub ComboBox1_DropDown(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles ComboBox1.DropDown
    'check to see if docked
    If Not Example1.Commands.m_ps.Dock.Equals(0) Then
        'docked so keep focus
        Example1.Commands.m_ps.KeepFocus = True
    End If
End Sub
```

This code will check to see if the palette set is docked. If not, it will pass through and perform the normal event. If it is docked, though, the code will force the palette set to keep focus so the cursor will work with the drop down. Of course, the palette set is now set to keep focus so we need to reset it like this:

```
Private Sub ComboBox1_DropDownClosed(ByVal sender As Object, _
  ByVal e As System.EventArgs) Handles ComboBox1.DropDownClosed
    'check to see if docked
    If Not Example1.Commands.m_ps.Dock.Equals(0) Then
        'docked so keep focus
        Example1.Commands.m_ps.KeepFocus = False
    End If
End Sub
```

NOTE: If you are using .Net Framework 1.1 or Visual Studio 2003, the ComboBox does not have a DropDown event. To accommodate, you need to create a custom control and add the event using Windows API code. An example is provided in the source download for this class.

AutoRollUp

AutoHide or AutoRollUp is when the tool palette contracts back to just the title bar. While there are no problems with this from the user's standpoint, accomplishing this via code offers some unique challenges. You will definitely understand once you have received the "Operation is not valid due to the current state of the object." exception. Before looking at the code to help with this, consider the challenges:

1. If the tool palette is docked, it cannot auto rollup.
2. While it's easy enough to set the docking property to "*None*", we need to wait for the tool palette to reinitialize before the auto rollup can occur.
3. If the cursor is located in the palette window, the palette will not be automatically refreshed; the cursor has to be moved off the palette window to have the change take effect.

So let's try this. Add a Button to the tool palette and add this code to its Click event:

```
Private Sub btnRollUp_Click(ByVal sender As System.Object, _
 ByVal e As System.EventArgs) Handles btnRollUp.Click
    'check to see if docked...
    If Example1.Commands.m_ps.Dock.Equals(DockSides.None) Then
        'yep so check for Snappable and turn off
        If Example1.Commands.m_ps.Style.Equals(32) Then
             Example1.Commands.m_ps.Style = 0
        End If
        'roll it up and toggle visibility so palette resets
        With Example1.Commands.m_ps
            .AutoRollUp = True
            .Visible = False
            .Visible = True
        End With
    Else
        'it's docked so undock
        With Example1.Commands.m_ps
            .Dock = Autodesk.AutoCAD.Windows.DockSides.None
            'roll it up and toggle visibility so palette resets
            .AutoRollUp = True
            .Visible = False
            .Visible = True
        End With
        'create timer to handle paletteset's change in docking
        CreateTimer()
    End If
End Sub
```

The comments within the code should be self-explanatory with the exception of the check for the palette set's Style property. The palette set has a value for Style called Snappable. Snappable is similar to docking and is used between two palette sets - the same as docking to an edge of the drawing area. Turning the whole palette set *OFF* and *ON* to refresh the window is a neat trick by DevTech's Bill Zhang who supplied this code [minus the Style check] in DevNote TS88082. Using this method does not require moving the cursor off of the palette window.

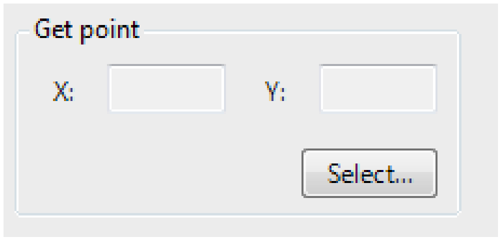
Next is the CreateTimer function which is used to implement the AutoRollUp setting after the palette set has undocked itself:

```
Private Shared Clock As System.Windows.Forms.Timer

 Friend Shared Sub CreateTimer()
     Clock = New System.Windows.Forms.Timer
     Clock.Interval = 500
     Clock.Start()
     AddHandler Clock.Tick, AddressOf Timer_Tick
 End Sub
```

```
    Friend Shared Sub Timer_Tick(ByVal sender As Object, _
      ByVal eArgs As EventArgs)
        If sender Is Clock Then
            Try
                With Example1.Commands.m_ps
                    .AutoRollUp = True
                    .Visible = False
                    .Visible = True
                End With
                'stop the clock and destroy it
                Clock.Stop()
                Clock.Dispose()
            Catch ex As Exception
                If Example1.Commands.m_ps.AutoRollUp.Equals(True) Then
                    'stop the clock and destroy it
                    Clock.Stop()
                    Clock.Dispose()
                End If
            End Try
        End If
    End Sub
```

Buttons

If you want the user to interact with the drawing [selecting entities, picking points, etc.], you will notice another quirk based on whether the palette set is docked or floating. Assuming you have a button for the user to select to start the drawing related task, if the palette set is docked everything will work as expected. If the palette set is floating, the user will have to pick the button, then pick inside the drawing area before the task will start. This two picks to get to the action can be annoying. To remedy this, use the AutoRollUp code from above and force the focus to the drawing area. To demonstrate this, add two textboxes with labels and a button to the container (as seen to the right). Then add code to the button that has the user select a point. The selected point will then populate the two textboxes.



```
        Dim bRollUp As Boolean = False
        <setting up Editor and prompt options omitted; see sample app>
        If m_Host.Dock.Equals(DockSides.None) Then  'check for docked...
            If m_Host.Style.Equals(32) Then m_Host.Style = 0
            With m_Host
                .AutoRollUp = True
                .Visible = False
                .Visible = True
            End With
            bRollUp = True  'set a flag so we can unroll if needed
        End If
```

Adding More Palettes

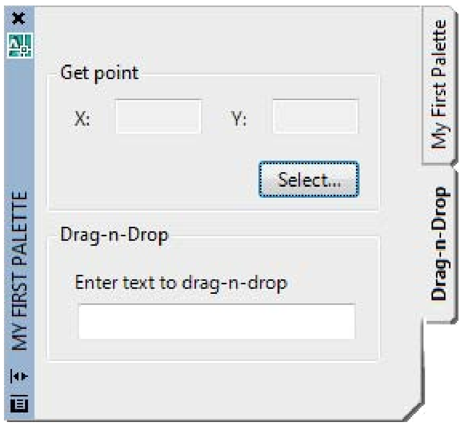
Now that we have covered the majority of the tool palette interaction, let's add another tool palette to our palette set. Simply add another user control to the project and name it Container2. For the moment, we will not add any controls or code to it.

Switch to the initial class which should be called Commands and modify the DoIt sub as follows:

```
    Private m_Container1 As Container1 = Nothing
    Private m_Container2 As Container2 = Nothing

    <CommandMethod("TestPalette")> _
 Public Sub DoIt()
    'check to see if paletteset is already created
    If m_ps Is Nothing Then
        'no so create it
        m_ps = New Autodesk.AutoCAD.Windows.PaletteSet("My First Palette")
        'create new instance of user control
        m_Container1 = New Container1()
        'add it to the paletteset
        m_ps.Add("My First Palette", m_Container1)
        'create new instance of 2nd user control
        m_Container2 = New Container2(m_ps)
        'add it to the paletteset
        m_ps.Add("Drag-n-Drop", m_Container2)
    End If
        'turn it on
        m_ps.Visible = True
    End Sub
```

As you can see, we just duplicated the Container1 lines and renamed the Container to Container2. I also gave Container2's tab "Drag-n-Drop" for a name which will lead us into the next topic…implementing drag-n-drop from our tool palette into the AutoCAD drawing area.

Implementing Drag-n-Drop



The first thing required for Drag and Drop is an object to drag so add a textbox to Container2 and name it txtDD. From this textbox, we will be able to handle drag and drop into the AutoCAD drawing editor.

To detect when a drag event is taking place, we need to know when certain mouse operations take place as well as consider what type of control we are using since each control behaves differently. For the textbox, we will use the MouseMove event. So select txtDD from the *Class Name* pulldown and then MouseMove in the *Method* pulldown. This will import the skeleton for handling the event:

```
    Private Sub txtDD_MouseMove(ByVal sender As Object, _
        ByVal e As System.Windows.Forms.MouseEventArgs) _
        Handles txtDD.MouseMove
            'code to follow later
    End Sub
```

Now that we can detect the mouse move operation, we need a way to know when, or if, the object is dropped into the AutoCAD drawing editor. To detect the drop, we need to add another class that will inherit the .NET base class DropTarget. Then we can implement any of the methods. We will need OnDrop in this exercise so add a new class to the project called DropTargetNotifier which inherits from Autodesk.AutoCAD.Windows.DropTarget and add the OnDrop event just like we did with the MouseMove event:

```
Public Class DropTargetNotifier

    Inherits DropTarget

    Public Declare Auto Function acedPostCommand Lib "acad.exe" _
      Alias "?acedPostCommand@@YAHPB_W@Z" (ByVal Expr As String) _
      As  Integer

    Private Shared m_DroppedData As String

    Public Overrides Sub OnDrop( _
      ByVal e As System.Windows.Forms.DragEventArgs)
        'catch the drop
        Dim dropTxt As String = e.Data.GetData(GetType(String))
        'read the data and store it
        m_DroppedData = dropTxt
        'start a command to handle the interaction with the user.
        'Don't do it directly from the OnDrop method
        AcApp.DocumentManager.MdiActiveDocument.SendStringToExecute _
          ("netdrop" & vbLf, False, False, False)
    End Sub

    'command handler for the netdrop command which is executed when the
    'drop occurs in the acad window.
    <CommandMethod("netdrop")> _
    Public Shared Sub netdropCmd()
        If Not data Is Nothing Then
            acedPostCommand(m_DroppedData & vbLf)
            m_DroppedData = Nothing
        Else
            acedPostCommand("nothing to do.")
        End If
    End Sub

End Class
```

To begin with, this class inherits from the DropTarget object which allows the ability to access the OnDrop event. There is also a local variable m_DroppedData to store the text being dropped and picked up by the OnDrop event through the DragEventArgs object. Then the OnDrop event calls the netdropCmd which sends the text string to the AutoCAD command line. Sending it to the command line is just for demonstration purposes. Once you have the text, you could query it or do something else depending upon what you are anticipating. Now we can go back to the textbox's MouseMove event and add the final bit of code:

```
Private Sub txtDD_MouseMove(ByVal sender As Object, _
 ByVal e As System.Windows.Forms.MouseEventArgs) _
 Handles txtDD.MouseMove
   If Control.MouseButtons.Equals(MouseButtons.Left) Then
     AcApp.DoDragDrop(Me, txtDD.Text, DragDropEffects.All, _
       New DropTargetNotifier())
   End If
End Sub
```

So we are going to check when the mouse moves within/over the textbox if the left mouse button is held down. If so, we activate the AutoCAD Application's DoDragDrop method. We pass the container object, the text from the textbox, define the DragDropEffects [which will generally be ALL], and a new instance of the DropTargetNotifier object. The DropTargetNotifier object is supplied so it can start watching for the drop to occur within the AutoCAD editor window – similar to a watched folder application. If the drop does not occur, the DropTargetNotifier object will be dismissed; if a drop occurs, the DropTargetNotifier object will handle the event accordingly.

For the most part, implementing the drag and drop functionality is complete – you can compile and run the code to test it. While it will work in this scenario, there is one more piece of the puzzle to add to this solution. This scenario works because we are sending the dropped data to the command line. If the data were to go to the drawing itself, the application would pop a fatal error because we are accessing the document from outside the command structure. By design, AutoCAD stores its data in documents where the data can only be edited by commands that have the required rights to make modifications. We need to lock the document while we access it. This is done via the Document.LockDocument method:

```
Private Sub txtDD_MouseMove(ByVal sender As Object, _
 ByVal e As System.Windows.Forms.MouseEventArgs) _
 Handles txtDD.MouseMove, txtADwg.MouseMove
    If Control.MouseButtons.Equals(MouseButtons.Left) Then
        Dim docLock As DocumentLock = _
            AcApp.DocumentManager.MdiActiveDocument.LockDocument()
        AcApp.DoDragDrop(Me, txtDD.Text, DragDropEffects.All, _
            New DropTargetNotifier())
        docLock.Dispose()
    End If
End Sub
```

In the MouseMove event, we instantiate a DocumentLock variable before sending the information to the DoDragDrop method. Then we dispose of the lock when the program returns from the method. That's all there is to implementing the DocumentLock!

Active Drawing Tracking

Since AutoCAD supports a multi-document interface, it may be important for your tool palette set to know which drawing is the active drawing. Suppose your tool palette set interacts with the drawing and remembers specific things about that particular drawing. If the cad user opens another drawing, your programming could go awry if it is unaware of the change in active drawing. To track the active document, we need to:

1. Add a class to handle storing the tracking information [See Appendix 1 for class code]
2. Switch to the initial class and add:
    a. A global variable: `Private Shared m_DocData As MyDocData = Nothing`

    b. The following code to the **DoIt** sub routine:

    ```
    If m_DocData Is Nothing Then m_DocData = New MyDocData
    AddHandler AcApp.DocumentManager.DocumentActivated, _
            AddressOf Me.DocumentManager_DocumentActivated
    AddHandler AcApp.DocumentManager.DocumentToBeDeactivated, _
            AddressOf Me.DocumentManager_DocumentToBeDeactivated
    ```

    c. Add the following events:

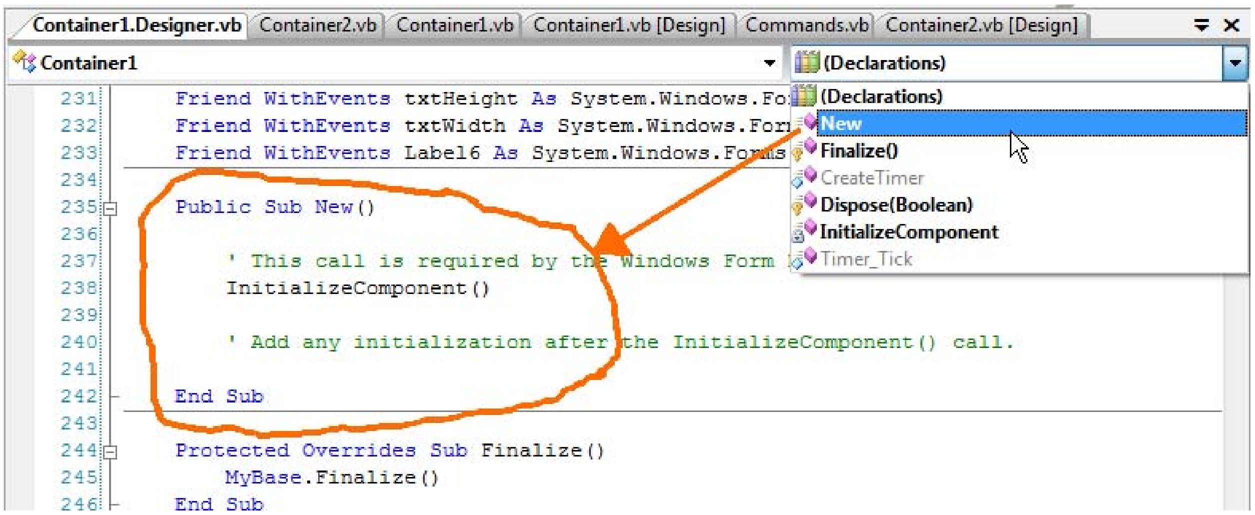
    ```
    Private Sub DocumentManager_DocumentActivated( _
      ByVal sender As Object, _
      ByVal e As DocumentCollectionEventArgs)
        'display the current active document
        If Not m_DocData Is Nothing Then
          m_Container2.txtADwg.Text = m_DocData.Current.Stuff
        End If
    End Sub

    Private Sub DocumentManager_DocumentToBeDeactivated( _
      ByVal sender As Object, _
      ByVal e As DocumentCollectionEventArgs)
        'store the current contents
        If Not m_DocData Is Nothing Then
          m_DocData.Current.Stuff = m_Container2.txtADwg.Text
        End If
    End Sub
    ```

    d. To Container2, add a textbox named txtADwg

Now as drawings are opened and closed, the DocumentActivated and DocumentToBeDeactivated events will fire and effectively track which drawing is currently active. The name of the active drawing will be displayed in the textbox so you can verify that the programming is working.

Cross Communication

Cross communication is how to talk between tool palettes as well as a better method for talking back to the host tool palette set. The method is surprisingly simple if you are familiar with customizing your constructors. To each Container object, we are going to change the **NEW** constructor. To get to it, the easiest way is to select Container1 in the *Class Name* pulldown and then New in the *Method* pulldown:



```
    Friend WithEvents txtHeight As System.Windows.Fo
    Friend WithEvents txtWidth As System.Windows.For
    Friend WithEvents Label6 As System.Windows.Forms

    Public Sub New()

        ' This call is required by the Windows Form

        InitializeComponent()

        ' Add any initialization after the InitializeComponent() call.

    End Sub

    Protected Overrides Sub Finalize()
        MyBase.Finalize()
    End Sub
```

This will take you into the Container1.Designer.vb file which is the old Windows Generated Code section from Visual Studio 2003 – that area that you weren't supposed to touch that Visual Studio 2005 now hides from you [depending on your 2005 environment settings of course]. To this sub, we are going to add the following code:

```
Private m_Host As PaletteSet

Public Sub New(ByRef Host As PaletteSet)
    ' This call is required by the Windows Form Designer.
    InitializeComponent()
    ' Add any initialization after the InitializeComponent() call.
    m_Host = Host
End Sub
```

We begin by adding a local variable to store the PaletteSet. Then add a ByRef parameter to the New sub. Making the addendum to the constructor is a better method than floating a global variable in the startup object which, in this case, would be to declare m_ps as *Public Shared*.

Now ordinarily, I would check to see whether m_Host was already instantiated or not before I instantiate it. In this scenario, I am not concerned because that check already exists on the Container object back in the *DoIt* sub routine. From here on, we can use m_Host to talk to the palette set object instead of typing in the fully qualified reference to it.

PLEASE pay attention! You must declare the parameter as *By Reference* (ByRef), not *By Value* (ByVal). ByRef passes a reference to the object in question, not a local copy of it.

While this is great way to talk to the host palette, suppose we want to talk to one of the other palettes in our palette set? It is possible to access other palettes via m_Host.PaletteSet.Items(int_for_order_loaded) but that is a lot of code. The best method is to change the constructor to accept a reference to our initial class object, Commands in my example. Then if I change the declaration of the local variables for each container to Friend Shared, I can access any container from any other container.

To the container class:

```
Private m_Host As Commands

Public Sub New(ByRef Host As Commands)
    ' This call is required by the Windows Form Designer.
    InitializeComponent()
    ' Add any initialization after the InitializeComponent() call.
    m_Host = Host
End Sub
```

To the initial class, change the local variables:

```
Friend Shared m_Container1 As Container1 = Nothing
Friend Shared m_Container2 As Container2 = Nothing
```

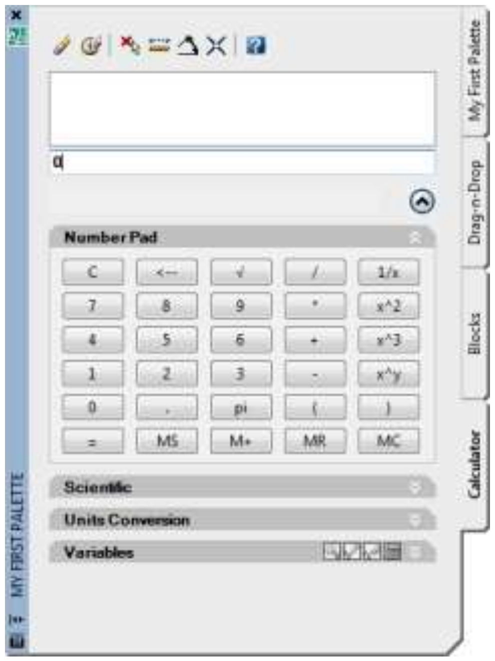
And to the constructor, change the value passed from the m_PS object to **Me**. Now we can access the other containers through m_Host.m_ContainerX – where X is the number corresponding to the container with which we want to interact.

AutoLoading

Now that you are all set to start implementing tool palettes in your application, how do you set your automation to automatically load whenever AutoCAD starts? It's as simple as a couple of registry entries:

[HKEY_LOCAL_MACHINE\SOFTWARE\Autodesk\AutoCAD\R17.1\ACAD-6001:409\Applications\Example1] **add key using the name of your application**
"DESCRIPTION"=**"Example1"** **name of your application**
"LOADER"=**"C:\\Program Files\\AutoCAD 2008\\Example1.dll"** **location of your application**
"LOADCTRLS"=**dword:00000000** **0 to automatically load your application**

Calculator Extra



I always like to end a class, or a column, with something I think is cool and most programmers are unaware of. There is another managed library for which I have not seen any documentation – acmgdinternal.dll. You can explore it on your own but it has some publishing, sheet set and calculator interaction. The coolest one is the calculator because it enables us to add the AutoCAD Calculator to a tool palette – here's how to do it:

1. Add a reference to acmgdinternal.dll
2. Add a reference to AcCalcUI.dll
3. Add the following import statements to the base class object [again. Commands in this example]:

```
Imports Autodesk.AutoCAD.AcCalc
Imports Autodesk.AutoCAD.CalculatorUI
Imports CalcDialogCreator
```

4. To the *DoIt* subroutine, add this code:

```
Dim ucCalc As New UserControl
ucCalc = New
CalculatorControl.AcCalcCalcCtrl(ucCalc)
m_PS.Add("Calculator", ucCalc)
```

5. Compile, debug, load and run Example1 in AutoCAD and you should see a new tab with the AutoCAD Calculator on it!

Appendix 1: DocData & MyDocData Classes

The following class is a VB.NET conversion of the C# class that originally appeared as part of DevNote TS88082

```
//Copyright (C) 2004-2006 by Autodesk, Inc.

Imports Autodesk.AutoCAD.ApplicationServices
Imports AcApp = Autodesk.AutoCAD.ApplicationServices.Application

MustInherit Class DocData

    Private Shared m_docDataMap As System.Collections.Hashtable

    Private Shared Sub DocumentManager_DocumentToBeDestroyed(_
      ByVal sender As Object, ByVal e As DocumentCollectionEventArgs)
        m_docDataMap.Remove(e.Document)
    End Sub

    Protected Delegate Function CreateFunctionType() As DocData

    Protected Shared CreateFunction As CreateFunctionType

    Public Shared ReadOnly Property Current() As DocData
        Get
            If m_docDataMap Is Nothing Then
                m_docDataMap = New System.Collections.Hashtable()
                AddHandler AcApp.DocumentManager.DocumentToBeDestroyed, _
                       AddressOf DocumentManager_DocumentToBeDestroyed
            End If
            Dim active As Document = AcApp.DocumentManager.MdiActiveDocument
            If Not m_docDataMap.ContainsKey(active) Then
                m_docDataMap.Add(active, CreateFunction())
            End If
            Return DirectCast(m_docDataMap(active), DocData)
        End Get
    End Property

End Class
```

```
Class MyDocData
    Inherits DocData

    Private m_stuff As String

    Shared Sub New()
        CreateFunction = New CreateFunctionType(AddressOf Create)
    End Sub

    Public Sub New()
        m_stuff = AcApp.DocumentManager.MdiActiveDocument.Window.Text
    End Sub

    Protected Shared Function Create() As DocData
        Return New MyDocData()
    End Function

    Public Property Stuff() As String
        Get
            Return m_stuff
        End Get
        Set(ByVal value As String)
            m_stuff = value
        End Set
    End Property

    Public Shared Shadows ReadOnly Property Current() As MyDocData
        Get
            Return DirectCast(DocData.Current, MyDocData)
        End Get
    End Property

End Class
```