Industrial Strength Add-Ins: Creating Commands in Autodesk® Inventor™

Brian Ekins – Autodesk, Inc.

DE211-4

This session focuses on techniques that will help you produce an “industrial strength” add-in application. You’ll learn how to create commands in Autodesk Inventor, handle user selections and mouse input, provide preview graphics, and support Undo. This class will benefit Inventor users and CAD and IT managers who want to automate or customize their systems and processes. Attendees should have solid Inventor background, and already be familiar with Inventor add-in creation and API basics.

About the Speaker:

Brian is an Autodesk Inventor API evangelist working with professionals and companies to smooth the process of learning the Inventor programming interface. He began working in the CAD industry over 25 years ago in various positions, including CAD administrator, applications engineer, CAD API designer, and consultant. Brian was the original designer of Inventor's API and has presented at conferences and taught classes throughout the world to thousands of users and programmers.

brian.ekins@autodesk.com

Autodesk®

This paper discusses the creation of Inventor commands. The basic concepts used to create an Add-In with a simple button to execute a function have been covered in the paper titled **Taking the Step from VBA to Inventor Add-Ins**. This paper picks up where that one left off and illustrates an example of creating a more complex command using Visual Basic 2008 Express Edition. There are several questions we'll address:

- What is the expected behavior of a command?
- How can I create an icon for my button?
- How can I define where my button goes?
- How do I create dialogs for my command?
- How can I control user selections?
- How can I show a preview?
- How can I support undo/redo of my command?

This paper isn't intended to be a thorough discussion of everything related to command development but instead focuses on typical use of the API to answer each of the questions above with emphasis on problems commonly encountered.

What is the expected behavior of a command?

Inventor is consistent in how its commands behave. For an Add-In command to feel like a standard Inventor command it should have the same behavior. Here are some of the behaviors of an Inventor command.

- When a command is invoked it terminates the currently running command.
- Complex commands provide a dialog to guide the user through the required input.
- Selection behavior is smart and limits selection to only what's appropriate.
- Provides a preview of the expected result.
- Supports undo in a logical way.

This paper will illustrate how the API provides support to help you meet these objectives and write Inventor commands.

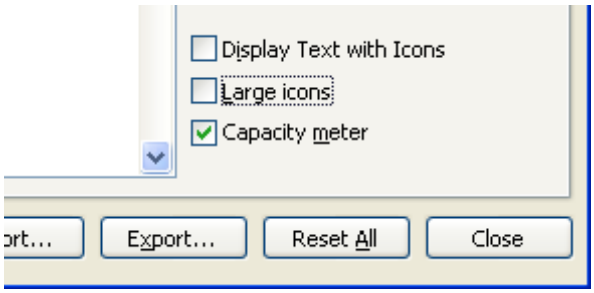
How can I create an icon for my button?

The paper **Taking the Step from VBA to Inventor Add-Ins** described how to create a ButtonDefinition object and use it to create a control on a custom toolbar. For most commands you'll want to do more than that. For example, you'll probably want an icon for the button and want more control over where the button is positioned within the user-interface.

Visual Basic 2005 Express does not provide a built-in tool to create icons but you can use any graphics editor you want to create a .bmp or .ico file. There is a document delivered as part of the SDK that discusses the guidelines you should follow when creating icons. The file is:

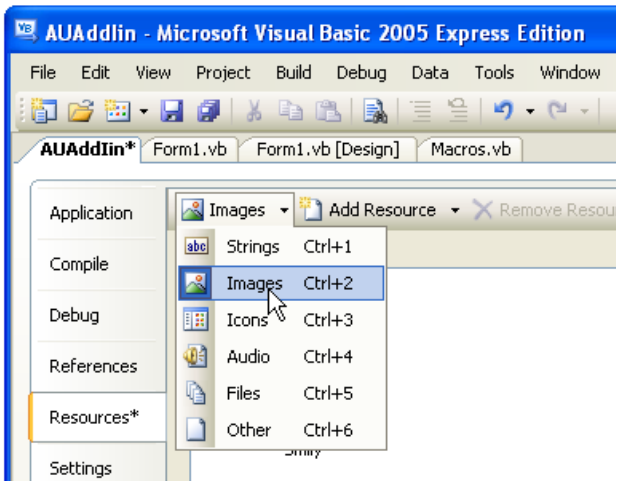
SDK\Docs\Guidelines\ Design Guidelines (Icons).doc

There are two standard sizes for icons, 16x16 and 24x24. The end-user can choose from the Toolbars tab of the Customize dialog whether they want large icons or not, as shown below. You can choose to only supply a small icon and Inventor will scale it to create a large one, but the result is not as good as when you specifically create the large icon.

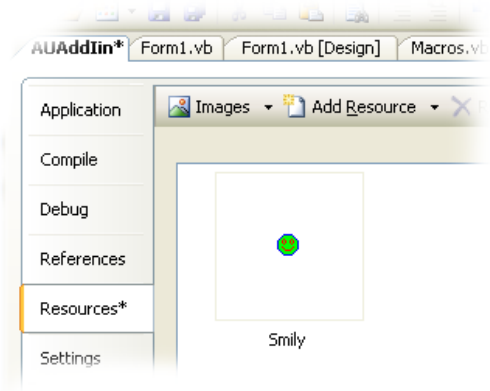


Icon design is somewhat of an art and can consume a lot of time to create something that looks good and represents the associated command. I've had my best luck designing icons by finding a command in Inventor whose icon has elements similar to the command I'm writing. I copy the icon by doing a screen copy and then edit it to get the desired result.

For this example I used the Paint program delivered with Windows to create a 16x16 icon and saved it as a .bmp file. A .bmp file can be imported into a VB project as a resource. To do this open the Properties page for your project by running the **Properties** command, (the last command in the Project menu), as shown below. On the Properties page select the **Resources** tab on the left and then choose the type of resource from the pull-down. The picture below illustrates selecting the "Images" type.



Next, you can add the resource using the “Add Resource” pull-down and selecting the “Add Existing File…” option. Choose your existing .bmp file to add it to the project. Finally, assign a logical name to the resource. You can see my image below that I’ve named “Smily”.



Finally, you can associate the image with the button definition you created in the Activate method of your Add-In. .Net uses a different type of object to represent bitmap data than VBA or VB6 did. Inventor uses the type that VBA and VB6 supported which is an IPictureDisp object. .Net uses the Image type, which is not compatible. When you access the picture as a resource it will be returned as an Image. You’ll need to convert it to an IPictureDisp before using it as input to the AddButtonDefinition method. To do this you’ll need to add references to two additional libraries to your VB project. These are Microsoft.VisualBasic.Compatibility and stdole. These are both available on the .Net tab of the Add Reference dialog.

With these references available you have access to the types and functions you need to create a button definition with an icon. The code below demonstrates this. Notice how the bitmap in the resources is accessed using its name (My.Resources.Smiley).

```
' Convert the Image to a Picture.
Dim picture As stdole.IPictureDisp
picture = Microsoft.VisualBasic.Compatibility.VB6.ImageToIPictureDisp( _
            My.Resources.Smily)

m_featureCountButtonDef = controlDefs.AddButtonDefinition( _
            "Count Features", _
            "AUAddInCountFeatures", _
            CommandTypesEnum.kQueryOnlyCmdType, _
            "{f0a47f25-537a-4082-a5b5-5d3d737d95ec}", _
            "Count the features in the active part.", _
            "Count Features",
            picture )
```

Now my command shows up with an icon like that shown below. Text is displayed with the icon depending on the end-user setting they set using the context menu within the panel bar.

How can I define where my button goes?

The paper **Taking the Step from VBA to Inventor Add-Ins** discusses creating a new toolbar and adding a button to it. In many cases it's better to integrate your button in with Inventor's buttons. For example, if you write a command that is useful when working with assemblies it will be good to have it appear on the assembly panel bar with the rest of the assembly commands.

It's possible to position your commands anywhere within Inventor's user-interface. The key to this is the CommandBar object. To insert your button into Inventor's user-interface you just need to find the existing command bar where you want your button. In inventor the panel bar, toolbars, menus, and context menu are all represented by CommandBar objects and all of them are accessible through the API.

One of the most common places to insert a button is the panel bar. The panel bar can host any command bar so you need to find the correct command bar to insert your button into. There is a default command bar used for each environment. Here are the names of the more commonly used command bars, along with the internal name. The internal name is how you identify it in your program.

Assembly Panel, AMxAssemblyPanelCmdBar
Assembly Sketch, AMxAssemblySketchCmdBar
Drawing Views Panel, DLxDrawingViewsPanelCmdBar
Drawing Sketch Panel, DLxDrawingSketchCmdBar
Drawing Annotation Panel, DLxDrawingAnnotationPanelCmdBar
Sheet Metal Features, MBxSheetMetalFeatureCmdBar
Part Features, PMxPartFeatureCmdBar
3D Sketch, SCxSketch3dCmdBar
2D Sketch Panel, PMxPartSketchCmdBar

Here's a portion of code from the Activate method that demonstrates inserting a button into the part feature panel bar.

```
If firstTime Then
    ' Get the part features command bar.
    Dim partCommandBar As Inventor.CommandBar
    partCommandBar = m_inventorApplication.UserInterfaceManager.CommandBars.Item( _
                     "PMxPartFeatureCmdBar")

    ' Add a button to the command bar, defaulting to the end position.
    partCommandBar.Controls.AddButton(m_featureCountButtonDef)
End If
```

This results in the part panel bar shown here.

The other interesting place to insert a button is the context menu. The context menu is also a CommandBar object, but it doesn't exist until just before it's displayed. To add a button to the command bar you listen to the OnContextMenu event and determine what the context is by checking to see what object is in the select set. If it's an object your command applies to you add your button to the command bar the OnContextMenu event provides through an argument.

How can I create dialogs for my command?

You create a dialog as a Visual Basic form and display it in response to the button definition's OnExecute event. You can use all of the available features in Visual Basic to create the dialog. The code below illustrates responding to the OnExecute event by displaying a dialog. This example uses the Show method to display the form in a modeless state. You can also use the ShowDialog method which will display it as a modal dialog.

```
Private Sub m_featureCountButtonDef_OnExecute( ... )
    ' Display the dialog.
    Dim myForm As New InsertBoltForm
    myForm.Show(New WindowWrapper(m_inventorApplication.MainFrameHWND))
End Sub
```

One issue with displaying a dialog is that by default it is independent of the Inventor main window. This can cause a couple of problems: the Inventor window can cover the dialog, and when the end-user minimizes the Inventor window your dialog is still be displayed. To get around these problems you can make the dialog a child of the Inventor window. The previous sample does this by using the WindowWrapper utility class. The code for this class is below.

```
#Region "hWnd Wrapper Class"
' This class is used to wrap a Win32 hWnd as a .Net IWind32Window class.
' This is primarily used for parenting a dialog to the Inventor window.
'
' For example:
' myForm.Show(New WindowWrapper(m_inventorApplication.MainFrameHWND))
'
Public Class WindowWrapper
    Implements System.Windows.Forms.IWin32Window
    Public Sub New(ByVal handle As IntPtr)
        _hwnd = handle
    End Sub

    Public ReadOnly Property Handle() As IntPtr _
      Implements System.Windows.Forms.IWin32Window.Handle
        Get
            Return _hwnd
        End Get
    End Property

    Private _hwnd As IntPtr
End Class
#End Region
```

How can I control user selections?

Selections are handled through the API using two different methods. First, and simplest, is the select set. Use of the select set is primarily limited to commands that have an object-action behavior. For example, when using the Delete command you select the objects (object) you want to delete and then execute (action) the command. The end-user has to know that the Delete command expects him to pre-select the entities and he has to know which entities are valid for deletion. Most Inventor commands don't behave like this but have an action-object type of behavior; you invoke the command and then do the selections.

An action-object style of selection is usually better because the selection can be smarter to help guide the user to make valid selections. For example, the Fillet command only lets you select edges and automatically chains along edges that are tangentially connected. In the Fillet dialog you can even change the selection behavior to select edges, loops, or features. You can also do these types of selection through the API using the InteractionEvents object.

The InteractionEvents object is very powerful and provides more functionality than just selection. One critical function it provides is Inventor command behavior; the termination of the active command. It also supports getting mouse and keyboard input and the display of preview graphics.

The basic steps of using the InteractionEvents object are:

1. When your command is invoked, create an InteractionEvents object.
2. Connect up to the events you're interested in: SelectEvents, MouseEvents, KeyboardEvents, and TriadEvents.
3. Define the behavior you want: the types of selectable entities, the prompt string, etc.
4. Start the InteractionEvents. This starts your command and terminates the running command.
5. Listen and respond to the events.
6. When the command is finished, stop the InteractionEvents or correctly exit if it's terminated by the end-user starting another command.

The code below illustrates this. For this example, all of the code for this command is contained within the SmartInsert class. Here's what the code does. After an instance of the class is created, the StartInsert method is called. This sets up and starts the InteractionEvents object. It listens to the OnSelect event of the SelectEvents object and allows the end-user to select circular edges. Once two edges are selected it creates an insert constraint between them and then allows the selection of two more edges.

```
Class SmartInsert
    ' Declare member variables.
    Private WithEvents m_InteractionEvents As Inventor.InteractionEvents
    Private WithEvents m_SelectEvents As Inventor.SelectEvents
    Private m_Edge1 As Inventor.EdgeProxy
    Private m_Edge2 As Inventor.EdgeProxy
    Private m_SelectingFirst As Boolean = True

    ' Starts the Smart Insert command.
    Public Sub StartInsert()
        ' Create an InteractionEvents object.
        m_InteractionEvents = _
                        g_inventorApplication.CommandManager.CreateInteractionEvents

        ' Connect to the select events.
        m_SelectEvents = m_InteractionEvents.SelectEvents

        ' Define the selection behavior.
        m_SelectEvents.AddSelectionFilter( _
                                Inventor.SelectionFilterEnum.kPartEdgeCircularFilter)
        m_SelectEvents.SingleSelectEnabled = True

        ' Set the text to show in the status bar.
        m_InteractionEvents.StatusBarText = "Select edge of first part."

        ' Start the command.
        m_InteractionEvents.Start()
    End Sub


    ' Handles selection within the command.
    Private Sub m_SelectEvents_OnSelect( _
                    ByVal JustSelectedEntities As Inventor.ObjectsEnumerator, _
                    ByVal SelectionDevice As Inventor.SelectionDeviceEnum, _
                    ByVal ModelPosition As Inventor.Point, _
                    ByVal ViewPosition As Inventor.Point2d, _
                    ByVal View As Inventor.View)

        ' Special case depending on if the first or second edge is selected.
        If m_SelectingFirst Then
            ' Save the selected edge.
            m_Edge1 = JustSelectedEntities.Item(1)

            ' Set the flag to select the second edge.
            m_SelectingFirst = False
        Else
            ' Save the selected edge.
            m_Edge2 = JustSelectedEntities.Item(1)

            ' Create the insert constraint.
            CreateInsert(m_Edge1, m_Edge2, True)
            m_Edge1 = Nothing
            m_Edge2 = Nothing
            m_SelectingFirst = True
        End If
    End Sub
End Class
```

Here's the code that creates and starts the smart insert command in response to the end-user clicking a button.

```
Private WithEvents m_featureCountButtonDef As ButtonDefinition
Private m_SmartInsert As SmartInsert


Private Sub m_featureCountButtonDef_OnExecute(ByVal Context As Inventor.NameValueMap)
    ' Create an instance of the SmartInsert class.
    m_SmartInsert = New SmartInsert

    ' Call the Start method to start the command.
    m_SmartInsert.StartInsert()
End Sub
```

When working with the InteractionEvents object you need to clean up when the command is terminated. This code does that by listening for the OnTerminate event and releasing the member variables.

```
' Handles termination of the command.
Private Sub m_InteractionEvents_OnTerminate()
    ' Release objects.
    GC.Collect()
    GC.WaitForPendingFinalizers()

    Marshal.FinalReleaseComObject(m_Edge1)
    Marshal.FinalReleaseComObject(m_Edge2)
    Marshal.FinalReleaseComObject(m_SelectEvents)
    Marshal.FinalReleaseComObject(m_InteractionEvents)
End Sub
```

The code above provides intelligent selection by limiting the selection to circular edges. This is much better than using the select set but still has limitations. For example, the user could pick two edges that are on the same part, which is not valid for a constraint. A useful feature would be to limit the selection of the second edge to one that is larger than the radius of the first edge, (for inserting a bolt into a hole). All of this is possible using the SelectEvents object and its OnPreSelect event.

The OnPreSelect event is fired as the end-user moves the mouse over the model. When they move the mouse over an edge that meets the filter criteria the OnPreSelect event is fired. In this example, whenever the end-user moves the mouse over any circular edge the OnPreselect event will fire. At this point, nothing is highlighted in the user interface. The entity the user moved the mouse over is provided to you through the OnPreSelect event. You can perform additional queries to determine if it should be selectable. This is demonstrated in the OnPreSelect implementation below where it checks to make sure the edges are on different parts and that the sizes are compatible. By controlling whether an entity is selectable or not through the OnPreSelect event you have the ability to filter the selection based on any criteria you choose.

```
Private Sub m_SelectEvents_OnPreSelect(ByRef PreSelectEntity As Object, _
                           ByRef DoHighlight As Boolean, _
                           ByRef MorePreSelectEntities As Inventor.ObjectCollection, _
                           ByVal SelectionDevice As Inventor.SelectionDeviceEnum, _
                           ByVal ModelPosition As Inventor.Point, _
                           ByVal ViewPosition As Inventor.Point2d, _
                           ByVal View As Inventor.View)
    If Not m_SelectingFirst Then
        Dim selectedEdge As EdgeProxy = PreSelectEntity

        ' Check if this edge is from the same occurrence as the first edge selected.
        If m_Edge1.ContainingOccurrence Is selectedEdge.ContainingOccurrence Then
            ' Set the return value to make this not selectable.
            DoHighlight = False
            Return
        End If

        ' Check if the radius of the second edge is smaller than the first edge.
        If getRadius(selectedEdge) <= GetRadius(m_Edge1) Then
            DoHighlight = False
        Else
            DoHighlight = True
        End If
    End If
End Sub
```

How can I show a preview?

Many of Inventor's commands provide a preview of the expected result. This is very useful for the end-user. This ability to show preview graphics is supported through the InteractionEvents object. However, depending on what the command is doing this can be difficult to implement. You basically need to construct client graphics to represent the preview. InteractionEvents supports its own set of client graphics through the InteractionGraphics object you get from an InteractionEvents object. Creating client graphics through the InteractionGraphics object is much faster than regular client graphics because it's not transacted. The lifetime of these graphics is only for the lifetime of your command too.

Client graphics expose graphics at a very low level which can be difficult to understand and use. Inventor 2008 added CurveGraphics support which greatly simplifies the creation of client graphics for wireframe graphics. A discussion of client graphics is outside the scope of this paper but now that you know the approach used for preview graphics you can start investigating and experimenting on your own. The online help has an overview discussion about this.

How can I support undo/redo of my command?

The undo/redo functionality of Inventor is controlled through *transactions*. A transaction wraps a set of actions within Inventor. When you undo, you're undoing the transaction and all of the actions within it. The API supports the ability to wrap a set of defined actions within a transaction. For example you can write a command that draws a sketch and creates a feature. This operation consists of many steps: drawing each sketch entity, constraining them, and

creating the feature. Using the transaction functionality you can group all of these actions within a single transaction. This allows the end-user to undo all of these actions in a single undo operation.

The use of transactions is very simple.

1. Create a transaction, giving it a logical name.
2. Perform the various operations needed for the command.
3. End the transaction.

If something should go wrong while you're performing the operations you can abort the transaction to return Inventor to the state it was in before your command was executed. All of this is demonstrated in the implementation of the CreateInsert function below, which is called in the OnSelect event. This is a simple example of using a transaction because there's only a single action being wrapped by the transaction but there can be any number of API calls between the start and end calls of the transaction.

```
' Create an insert constraint between two circular edges.
Public Function CreateInsert(ByVal EdgeProxy1 As Inventor.EdgeProxy, _
                             ByVal EdgeProxy2 As Inventor.EdgeProxy, _
                             ByVal IsOpposed As Boolean) As Inventor.InsertConstraint
    Dim transaction As Inventor.Transaction = Nothing
    Try
        ' Get the parent assembly component definition from one of the edges.
        Dim assemblyDef As Inventor.AssemblyComponentDefinition
        assemblyDef = EdgeProxy1.ContainingOccurrence.ContextDefinition

        ' Start a transaction.
        transaction = g_inventorApplication.TransactionManager.StartTransaction( _
                                    assemblyDef.Document, "Smart Insert")

        ' Create the constraint.
        Dim insert As Inventor.InsertConstraint
        insert = assemblyDef.Constraints.AddInsertConstraint(EdgeProxy1, _
                                    EdgeProxy2, IsOpposed, 0)

        ' End the transaction.
        transaction.End()

        Return insert
    Catch ex As Exception
        If Not transaction Is Nothing Then
            transaction.Abort()
        End If
        Return Nothing
    End Try
End Function
```