Autodesk University 2007

The Best of Both Worlds: .NET and LISP Can Coexist

Wayne Brill - Autodesk

DE211-2 You know that .NET is a modern and powerful programming environment. But, no matter how much you might want to start using it, you can't seem to abandon all your old LISP code. We'll show you how to write .NET code that can interoperate with your existing LISP code, with a special emphasis on user interface components.

About the Speaker:

Wayne has been a member of Autodesk Developer Technical services since 2000. Currently, Wayne works for Autodesk supporting the APIs for Inventor, AutoCAD Mechanical, AutoCAD, AutoCAD OEM, and RealDWG.

The Best of Both Worlds: .NET and LISP Can Coexist

This document has two sections. The first section is a discussion about communication between AutoLISP and AutoCAD .NET functions. The second section is a tutorial that steps you through creating a .NET form and using this form with the AutoLISP tutorial that ships with AutoCAD.

Communication between AutoLISP functions and .NET functions

The AutoCAD .NET API was introduced as a preview in AutoCAD 2005. Prior to the AutoCAD 2007 release there was not much interaction possible from AutoLISP and AutoCAD .NET. In AutoCAD .NET 2007 the LispFunction attribute was introduced. Using this attribute it is now possible to run .NET functions from AutoLISP.

The LispFunction attribute takes a string that defines the function name. The function that follows the LispFunction attribute will be the .NET function that will be called when the LispFunction is run from AutoLISP. Here are the LispFunction Signatures for C# and VB.NET.

C#

```
[LispFunction("HelloWorld")]
static public ResultBuffer helloWorld(ResultBuffer theArgs)
```

VB.NET

```
<LispFunction("HelloWorld")> _
Public Function helloWorld(ByVal theArgs As ResultBuffer) As
ResultBuffer
```

In these examples the LISP function name is “HelloWorld”. When the (HelloWorld) function is run from AutoLISP the .NET helloWorld function will be called. Notice that a ResultBuffer is passed into the function and is returned. The ResultBuffer contains arguments that allow data to be passed between AutoLISP and the .NET function. In AutoCAD 2008 a ResultBuffer is the only type that can be returned. This should not be a major issue as the ResultBuffer can contain different types of data.

The ResultBuffer contains TypedValues. A TypedValue is a data pair. (Data type and the data value) When you create a new ResultBuffer you can pass in TypedValues to the constructor. The LispDataType object will allow you to easily set the type. Here are the Result Type codes that can be used for the data type. (These are 16 bit integer codes)

_atom	Nil	DottedPair	ListEnd
ListBegin	Point3d	Orientation	ObjectId
Angle	Point2d	None	SelectionSet
Void	Int32	Int16	Double
Text			

Take a look at ,NET function below. This example will get the arguments from the ResultBuffer that is passed into the function. It then creates a new ResultBuffer using a TypedValue and LispDataType. It returns the updated arguments to the calling AutoLISP function. To test this function use the NETLOAD command and load the .NET dll that has this LispFunction defined. Enter the following on the command line: “(changeArgs 10.1 20.1 30.1)” Here is an excerpt from the command line that shows the result. Notice the returned values were added together.

```
Command: (changeArgs 10.1 20.1 30.1)
Updated arguments in .NET function
(30.2 50.2 60.3)
```

```
<LispFunction("changeArgs")> _
Public Function dNetLispDemo02a(ByVal myLispArgs As ResultBuffer) As
ResultBuffer
        ' Shows how to get arguments passed into a lisp function
        ' change them and send arguments back
        Dim ed As Editor
        ed =
Autodesk.AutoCAD.ApplicationServices.Application.DocumentManager.MdiActiveDoc
ument.Editor

        'Ensure there are arguments passed in
        If myLispArgs Is Nothing Then
            ed.WriteMessage("No arguments" & vbLf)
            Return myLispArgs
        Else

            Dim myArgsArray As Array
            myArgsArray = myLispArgs.AsArray
            'Ensure there are at least 3 arguments
            If myArgsArray.Length > 2 Then

                Dim myArg1, myArg2, myArg3 As Double

                'Get the arguments that were passed in
                Dim myTypeVal As TypedValue
                myTypeVal = myArgsArray.GetValue(0)
                myArg1 = myTypeVal.Value
                ' does the same thing as the previous two lines gets
Arguments 2 and 3
                myArg2 = CType(myArgsArray.GetValue(1), TypedValue).Value
                myArg3 = CType(myArgsArray.GetValue(2), TypedValue).Value

                ' Add the arguments to make a change
                myArg1 = myArg1 + myArg2
                myArg2 = myArg2 + myArg3
                myArg3 = myArg3 + myArg1

                ' Package data to send back to calling lisp function
                Dim rbfResult As ResultBuffer
                rbfResult = New ResultBuffer( _
                                New TypedValue(CInt(LispDataType.Double),
myArg1), _
                                New TypedValue(CInt(LispDataType.Double),
myArg2), _
                                New TypedValue(CInt(LispDataType.Double),
myArg3))

                ed.WriteMessage("Updated arguments in .NET function" & vbLf)

                Return rbfResult
            Else
                ed.WriteMessage("Not enough arguments" & vbLf)
                Return myLispArgs
            End If
        End If

        Return myLispArgs

    End Function
```

The following AutoLISP function demonstrates using the .NET example above. Run the function testchangeArgs and pass in a double. “(testchangeArgs 20.0)” The value will be changed. The original and updated value will be displayed on the command line.

```
(defun testchangeArgs (dToChange / myDoubles d1 str str2)
 (setq myDoubles(changeArgs 20 dToChange 30)); send the arguments to .NET
 (Setq d1(nth 1 myDoubles));get the second argument returned
 (setq str(strcat ".NET changed " (rtos dToChange)))
 (setq str2(strcat " to " (rtos d1)))
 (princ (strcat str str2))
 (princ)
 )
```

Here is the result after running this function:

```
Command: (testchangeargs 20)
Updated arguments in .NET function
.NET changed 20.0000 to 50.0000
```

The rest of this document is a tutorial that will show how to use a .NET form to gather input from the user. This input is then used from AutoLISP.

.NET Tutorial - Using a .NET Form with the AutoLISP Garden Path Tutorial

In this tutorial, we will create a lisp callable function (LispFunction) that will display a form. This form will replace the DCL dialog in the AutoLISP GardenPath tutorial.

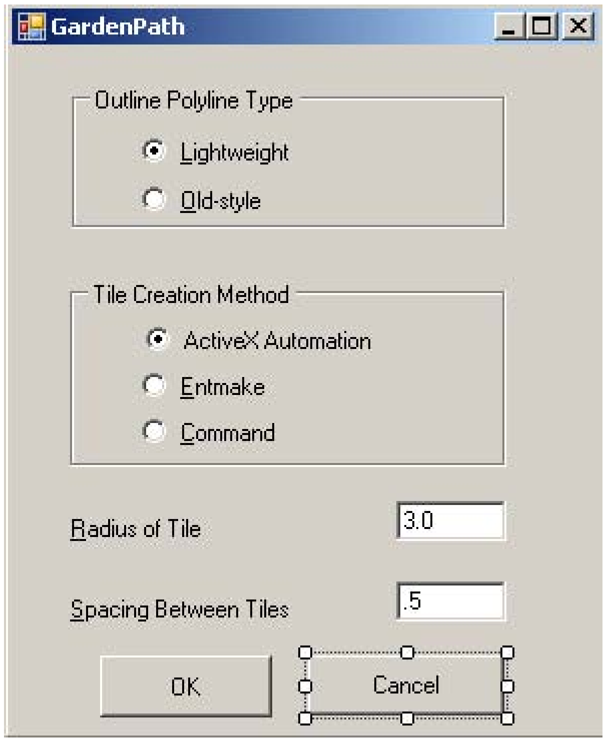
You can use the LispFunction attribute to create a lisp callable function. In this section we will add a Form to our project that will be displayed when the LispFunction is called from an AutoLISP function. This form will allow the user to make settings that will be returned to the AutoLISP function. In this lab we will replace the DCL dialog used in the GardenPath tutorial that ships with AutoCAD with our .NET form.

This tutorial assumes you have created a .NET project for AutoCAD. If you need some help doing this, then download the .NET training from the link below. Lab one in this training will show you how to create a .NET project for AutoCAD. Lab 6 shows how to add forms to a project.

http://www.autodesk.com/developautocad

1) Create a .NET form.

First we need to add a new form to the project. Select the Project Menu and use Add Windows Form. On the Add New Item dialog enter GardenPath.vb for the name. When you are finished adding the controls, the form the will look something like this:


GardenPath

Outline Polyline Type
● Lightweight
○ Old-style

Tile Creation Method
● ActiveX Automation
○ Entmake
○ Command

Radius of Tile: 3.0

Spacing Between Tiles: .5

Display the Visual Studio ToolBox (Cntrl+Alt+X) and add two Group boxes to the form. Using the properties Window, (Alt+Enter) change the Text property for the Group boxes:

```
<First, Group box>
```

```
Text = <Outline Polyline Type>
<Second, Group box>
Text = <Tile Creation Method>
```

Add two radio buttons inside of the “Outline Polyline Type” Group box. Change the properties of the radio buttons.

```
<First, Radio button>
Name = <rb_LightWeight>
Text = <&LightWeight>

<Second, Radio button>
Name = <rb_Oldstyle>
Text = <&Old-style>
```

Add three radio buttons inside of the “Tile Creation Method” Group box. Change the properties of the three radio buttons:

```
<First, Radio button>
Name = <rb_ActiveX>
Text = <&ActiveX Automation>

<Second, Radio button>
Name = <rb_Entmake>
Text = <&Entmake>

<Third, Radio button>
Name = <rb_Command>
Text = <&Command>
```

Add two labels below the “Tile Creation Method” Group box. Change the Text properties:

```
<First, Label>
Text = <Radius of tile>

<Second, Label>
Text = <Spacing Between Tiles>
```

Add two TextBoxes to the right of the labels. Change the properties of the TextBoxes:

```
<First Textbox>
Name = <tb_RadiusOfTile>
Text = <3.0>
```

```
<Second Textbox>
Name = <tb_SpacingBetweenTiles>
Text = <0.5>
```

Add two buttons for OK and Cancel.

```
<First button>
Name = <bt_OK>
Text = <OK>

<Second button>
Name = <bt_Cancel>
Text = <Cancel>
```

Double click on the OK button to bring up the code window for the form. In the

`bt_OK_Click event handler add Me.Close()`

```
Private Sub bt_OK_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles bt_OK.Click
        Me.Close()
End Sub
```

Add Me.Close() to the event handler for the button bt_Cancel as well.

```
Private Sub bt_Cancel(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles bt_OK.Click
        Me.Close()
End Sub
```

2) Create a function using the LispFunction attribute.

The LispFunction attribute was introduced in the AutoCAD 2007 .NET API. This attribute is designed to allow AutoLISP functions to call AutoCAD .NET functions. (it allows sending and returning values) The LispFunction attribute is similar to the CommandMethod attribute. It can be used in the same places and has the similar properties.

Add a LispFunction named VBNetFunction to Class1.vb. Use dNetGpath as the LispFunction name. This function will display the form created in step 1. This function will return the users selections on the form to the GardenPath Lisp routine.

```
<LispFunction("dNetGpath")> _
Public Function VBNetFunction(ByVal myLispArgs As ResultBuffer) As
ResultBuffer
End Function
```

First show the GardenPath dialog by adding code to the VBNETFunction that instantiates a new GardenPath form and shows it:

```
Dim gPathFrm As New GardenPath
Application.ShowModalDialog(gPathFrm)
```

We can return a list to the calling AutoLISP function using a ResultBuffer. A ResultBuffer is used to contain data. The data in the ResultBuffer is organized by pairs. Each pair of data will contain a data type description and a value.

Next declare a ResultBuffer by adding the following code:

```
Dim rbfResult As ResultBuffer
```

Our function needs to return different values depending on how the user interacts with our GardenPath form. First we need to know if the User clicked OK or Cancel. Add a global Boolean variable named bOK to the GardenPath class.

```
Public bOK As Boolean = False
```

In the bt_OK_Click event handler make this variable true.

```
bOK = True
```

We need to test the bOK variable in our vbNetFunction. If this variable is true the user clicked OK and we need to get the values from the GardenPath form. Once we have these vaules we can return them to the calling AutoLISP function. If the user clicked Cancel we need to return nil. Add an if statement to the vbNetFunction testing bOK:

```
If gPathFrm.bOk = True Then
```

Inside the bOK “if” statement add code that will get the values from the form. Other nested if statements can be used to get the values from the radio buttons by querying the Checked property.

```
' Get the radius of tile to use
Dim dRadOfTile As Double
dRadOfTile = gPathFrm.tb_RadiusOfTile.Text

' Get the spacing of tiles to use
Dim dSpaceOfTiles As Double
dSpaceOfTiles = gPathFrm.tb_SpacingBetweenTiles.Text

' Get the creation type
Dim strCreationType As String
If gPathFrm.rb_ActiveX.Checked Then
     strCreationType = "ActiveX"
ElseIf gPathFrm.rb_Entmake.Checked Then
     strCreationType = "Entmake"
Else
```

```
    strCreationType = "Command"
End If

Dim strPlineType As String
If gPathFrm.rb_Lightweight.Checked Then
    strPlineType = "Light"
Else
    strPlineType = "Heavy"
End If
```

Now that we have the input selected by the user we can populate the ResultBuffer. We use a TypedValue to package up the data we want to send back to the AutoLISP function. The TypedValue accepts two parameters. One is the type and the other is the data.

Add the following code to the end of the “If bOK = True” statement:

```
'Dim rbfResult As ResultBuffer
rbfResult = New ResultBuffer(New TypedValue(CInt(LispDataType.Double),
dRadOfTile), _
New TypedValue(CInt(LispDataType.Douple), dSpaceOfTiles), _
New TypedValue(CInt(LispDataType.Text), strCreationType), _
New TypedValue(CInt(LispDataType.Text), strPlineType))
Return rbfResult

End If
```

This will return a list to the AutoLISP function. Similar to this:

(3.0 0.5 “ActiveX” “Light”)

Now we need to handle the case where the user cancels the GardenPath form instead of hitting OK. After the End If statement add the following code:

```
    rbfResult = New ResultBuffer(New TypedValue(CInt(5019)))
    Return rbfResult
End Function
```

Here is the complete code for the dNetGpath function:

```
    <LispFunction("dNetGpath")> _
     Public Function VBNetFunction(ByVal myLispArgs As ResultBuffer) As
ResultBuffer
        Dim gPathFrm As New GardenPath
        gPathFrm.ShowDialog()
        Dim rbfResult As ResultBuffer
        If gPathFrm.bOk = True Then
            ' Get the radius of tile to use
            Dim dRadOfTile As Double
            dRadOfTile = gPathFrm.tb_RadiusOfTile.Text

            ' Bet the spacing of tiles to use
            Dim dSpaceOfTiles As Double
            dSpaceOfTiles = gPathFrm.tb_SpacingBetweenTiles.Text

            ' Get the creation type
            Dim strCreationType As String
            If gPathFrm.rb_ActiveX.Checked Then
                strCreationType = "ActiveX"
            ElseIf gPathFrm.rb_Entmake.Checked Then
                strCreationType = "Entmake"
            Else
                strCreationType = "Command"
            End If

            Dim strPlineType As String
            If gPathFrm.rb_Lightweight.Checked Then
                strPlineType = "Light"
            Else
                strPlineType = "Heavy"
            End If

            'Dim rbfResult As ResultBuffer
            rbfResult = New ResultBuffer( _
                            New TypedValue(CInt(5001), dRadOfTile), _
                            New TypedValue(CInt(5001), dSpaceOfTiles), _
                            New TypedValue(CInt(5005), strCreationType), _
                            New TypedValue(CInt(5005), strPlineType))
            Return rbfResult
        End If

        rbfResult = New ResultBuffer(New TypedValue(CInt(5019)))
        Return rbfResult
    End Function
```

3) Modify the LISP tutorial to call the LispFunction created in Step 2.

Run AutoCAD and open the Visual LISP editor. (VLIDE) On the Projects menu select “Open Project”, navigate to the VisualLISP tutorial directory Lesson 7 and open the gpath7.prj. The Visual Lisp Tutorial can be found in this directory if the option to install it was included during the AutoCAD install: C:\Program Files\AutoCAD 2008\Tutorial\VisualLISP

After the project files are opened SaveAs GPMAIN.lsp file. Name the new file GPMAIN_NET.lsp. We will edit this file to call our .NET LispFunction.

Change the function name from Gpath to GpathN. Update the variable list to include a variable that we will be using. (valuesFromdNet)

```
(defun C:GPathN (/gp_PathData gp_dialogResults PolylineName  p_PathData tileList PolylineList
valuesFromdNet )
```

Now we need to change the function so that it will call the dNetGpath function instead of gp:getDialogInput. Comment out these lines of code after setq gp_dialogResults.

```
;(gp:getDialogInput
;                        (cdr (assoc 40 gp_PathData))
;                  ) ;_ end
```

The logical flow of the function is updated with a couple of new progn statements. Add a progn statement after (if (setq gp_PathData (gp:getPointInput))

```
 (if (setq gp_PathData (gp:getPointInput))
   (progn ; Update for .NET LAB
```

Now we need to call the LispFunction. Add the call to dNetGpath and set the return value to the valuesFromdNet variable. Also add an if statement to check to see if the user hit Cancel. (valuesFromdNet will be nil) Add a progn statement before the existing "if setq gp_dialogResults".

```
; Call our .NET function
 (setq valuesFromdNet (dNetGpath))      ; Update for .NET LAB
 (if (/= nil (nth 0 valuesFromdNet)); Update for .NET LAB
          (progn                                   ; Update for .NET LAB
          (if (setq gp_dialogResults
```

We need to populate gp_dialogResults with the values from the valuesFromdNet variable. Use the list function and cons to construct the correct values for gp_dialogResults. (This replaces the code we commented out above)

```
 (if (setq gp_dialogResults
                     ; Get the data returned from our .NET function
                         (list
                              (cons 42 (nth 0 valuesFromdNet))
                              (cons 43 (nth 1 valuesFromdNet))
                              (cons 3 (nth 2 valuesFromdNet))
                              (cons 4 (nth 3 valuesFromdNet))
                              (cdr (assoc 40 gp_PathData))
                              )
```

Now add the closing parentheses for the two progn statements and the one if statement. Near the end of the function find the code for the vlr-editor-reactor. Add the last three closing parenthesis as seen in this code snippet. If you have trouble with the placement of the parenthesis use "Edit>Parenthesis Matching". (Ctrl+[and Ctrl+])

```
                  ) ;_ end of vlr-editor-reactor
                 )
           )

            )    ; progn after if (setq gp_dialogResults
          ); if (setq gp_dialogResults
```

```
        ) ;_ progn after if (/= nil valuesFromdNet - Update for .NET
LAB
```

Finally move the “Function cancelled” princ statement inside of the (if (/= nil (nth 0 valuesFromdNet)) statement. Notice the comment also needs to be changed. The closing parenthesis is no longer for the progn.

```
      (princ "\nFunction cancelled.")

       ) ;_ end of progn
       ;(princ "\nFunction cancelled.")
```

Here is the complete lisp function:

```
(defun C:GPathN ( / gp_PathData gp_dialogResults
                  PolylineName gp_PathData tileList PolylineList valuesFromdNet Result
                  )
 (setvar "OSMODE" 0)
 ;; Ask the user for input: first for path location and
 ;; direction, then for path parameters.  Continue only if you have
 ;; valid input.  Store the data in gp_PathData
 (if (setq gp_PathData (gp:getPointInput))
   (progn ; Update for .NET LAB
    ; Call our .NET function
    (setq valuesFromdNet (dNetGpath))   ; Update for .NET
    (if (/= nil (nth 0 valuesFromdNet)); Update for .NET
        (progn    ; Update for .NET
         (if      (setq gp_dialogResults
                  ; Get the data returned from our .NET function
                           (list
                                   (cons 42 (nth 0 valuesFromdNet))
                                   (cons 43 (nth 1 valuesFromdNet))
                                   (cons 3 (nth 2 valuesFromdNet))
                                   (cons 4 (nth 3 valuesFromdNet))
                                   (cdr (assoc 40 gp_PathData))
                             )

;;;          (gp:getDialogInput
;;;               (cdr (assoc 40 gp_PathData))
;;;          ) ;_ end of gp:getDialogInput
        ) ;_ end of setq
    (progn

;;;     (princ "\nReceived gp_dialogResults: ") (princ gp_dialogResults)

        ;; Now take the results of gp:getPointInput and append this to
        ;; the added information supplied by gp:getDialogInput
        (setq gp_PathData (append gp_PathData gp_DialogResults))

        ;; At this point, you have all the input from the user.
        ;; In lesson 7, the gp:drawOutline function was modified to
        ;; return a list of the pointer to the polyline as well as
        ;; the list of boundary points (the 12, 13, 14, 15 lists)
        ;; Draw the outline, storing the resulting polyline "pointer"
        ;; in the variable called PolylineName, and the boundary
        ;; points into gp_pathData
        ; (trace gp:drawOutline)
        (setq PolylineList    (gp:drawOutline gp_PathData)
              PolylineName    (car PolylineList)
              gp_pathData     (append gp_pathData (cadr PolylineList))
        ) ;_ end of setq


        ;; Next, it is time to draw the tiles within the boundary.
        ;; The tileList contains a list of the object pointers for
        ;; the tiles.  By counting up the number of points (using the
        ;; length function), we can print out the results of how many
        ;; tiles were drawn.
        (princ "\nThe path required ")
        (princ
         (length
           (setq tileList
                  (gp:Calculate-and-Draw-Tiles
                    ;; path data list
                    gp_PathData
                    ;; object creation style to use - should be nil
                    ;; when drawing initial path.  Subsequent calls
                    ;; from reactor will provide the function to use.
                    nil
                  ) ;_ end of gp:Calculate-and-Draw-Tiles
           ) ;_ end of setq
         ) ;_ end of length
        ) ;_ end of princ
        (princ " tiles.")

        ;; Add the list of pointers to the tiles (returned by
        ;; gp:Calculate-and-Draw-Tiles) to the gp_PathData variable.
        ;; This is stored in the reactor data for the reactor attached
        ;; to the boundary polyline.  With this data, the polyline
        ;; "knows" what tiles (circles) belong to it.
        (setq gp_PathData   (append (list (cons 100 tileList))
                                                    ; all the tiles
```

```
                                        gp_PathData
                            ) ;_ end of append
) ;_ end of setq

;; Before we attach reactor data to an object let's look at
;; the function vlr-object-reactor.
;; vlr-object-reactor has the following arguments:
;;          (vlr-object-reactor owners data callbacks)
;;      The callbacks Argument is a list comprising of
;;                      owner , Reactor_Object list
;;                      For further explanation see Help system
;; For this exercise we will use all arguments
;; associated with vlr-object-reactor

;; These reactor functions will excecute only if
;; the polyline in  PolylineName is modified or erased

(vlr-object-reactor

  ;; The first argument for vlr-object-reactor is
  ;; the "Owners List" argument.  This is where to
  ;; place the object to be associated with the
  ;; reactor.  In this case it is the vlaObject
  ;; stored in PolylineName

  (list PolylineName)

  ;; The second argument contains the data for the path

  gp_PathData

  ;; The third argument is the list of specific reactor
  ;; types that we are interested in dealing with

  '(
    ;; reactor that is called upon modification of the object
    (:vlr-modified . gp:outline-changed)
    ;; reactor that is called upon erasure of the object
    (:vlr-erased . gp:outline-erased)
   )
) ;_ end of vlr-object-reactor



;; Next, register a command reactor to adjust the polyline
;; when the changing command is finished.
```

```
          (if (not *commandReactor*)
           (setq   *commandReactor*
                   (VLR-Command-Reactor

                     nil                                 ; No data is associated with the editor reactor
                     ;; call backs
                     '
                      (
                       (:vlr-commandWillStart . gp:command-will-start)
                       (:vlr-commandEnded . gp:command-ended)
                       )
                     ) ;_ end of vlr-editor-reactor
                   )
           )

          (if (not *DrawingReactor*)
           (setq *DrawingReactor*
                   (VLR-DWG-Reactor

                     nil                                 ; No data is associated with the editor reactor
                     ;; call backs
                     '
                      (
                       ;; This is extremely important!!!!!!!!!
                       ;; Without this notification, AutoCAD will
                       ;; crash upon exiting.
                       (:vlr-beginClose . gp:clean-all-reactors)
                       )
                     ) ;_ end of vlr-editor-reactor
                   )
           )

     )      ; progn after if (setq gp_dialogResults
             ); if (setq gp_dialogResults
           ) ;_ progn after if (/= nil valuesFromdNet - Update for .NET LAB


          (princ "\nFunction cancelled.")

     ) ;_ end of progn
     ;(princ "\nFunction cancelled.")
   ) ;_ end of if
   (princ "\nIncomplete information to draw a boundary.")
 ) ;_ end of if
 (princ)                                ; exit quietly
) ;_ end of defun
```

```
;;; Display a message to let the user know the command name
(princ "\nType GPATHN to draw a garden path.")
(princ)
```

After following the steps above load the .NET project, GPMAIN_NET.lsp and all of the Garden Path tutorial files. Run the GPATHN command. If all goes well the .NET form will be displayed and used to get input for the garden path.

If you debug the .NET project you will not be able to run VLIDE. (Error occurs) You can use another means such as the APPLOAD command to load the LISP files. To have the LISP files automatically load you can use the Start Up suite. (Available from the APPLOAD dialog)

Additional Resources

Here are a few links with additional resources for AutoCAD .NET:

AutoCAD Developer Center - download training labs
http://www.autodesk.com/developautocad

AutoCAD .NET Training (classroom)
http://www.autodesk.com/apitraining

Through the Interface blog - Site focuses on .NET
http://blogs.autodesk.com/through-the-interface

Also see the ObjectARX Help file for topics related to AutoCAD .NET