ActiveX Tricks for Visual LISP and VBA
R. Robert Bell – MW Consulting Engineers
Peter Jamtgaard – Cordeek

CP23-3 You can do some amazing things in AutoCAD® using ActiveX. This course shows you several examples of the power available to you, both in Visual LISP® and VBA. Because you see the same approach taken in both languages, you will gain confidence in using the ActiveX interface. This class is designed for programmers who want to see the advantages to programming using the ActiveX interface.

Who Should Attend
Advanced-level programmers

Topics Covered
* Auto-rotating attributes upon insertion
* Setting a table cell—s background to mask object below
* Deleting layer filters
* Non-permanent context menu editing
* Placing new insertions on same layer as first instance

About the Speakers:
Robert is the network administrator for MW Consulting Engineers, an engineering consulting firm in Spokane, Washington, where he has worked for the last 16 years. He is responsible for the network and all AutoCAD® customization. Robert has been writing AutoLISP® code since AutoCAD v2.5, and Visual Basic programs for 6 years. He has customized applications for the electrical/lighting, plumbing/piping, and HVAC disciplines. Robert has also developed applications for AutoCAD as a consultant. He is on the Board of Directors for AUGI® and is active on Autodesk newsgroups.

RobertB@MWEngineers.com

Peter has 18 years of experience running AutoCAD® in a multidiscipline environment, including Civil Engineering, Structural Engineering, Mechanical, Architectural, and many others. He has served on the AUGI® Board of Directors. Peter has been programming AutoLISP® for 18 years and is experienced with VBA(A). He holds an Associates degree in Civil Engineering Technology and a B.S. degree in Civil and Environmental Engineering.

cordeck@acronet.net
ActiveX Tricks for Visual LISP and VBA

Have you programmed for years using AutoLISP yet regard the ActiveX interface with a measure of trepidation? Are you frustrated by the apparent lack of documentation for the `vla-` functions? Perhaps you thought about trying some programs in VBA but didn’t know where to start. Are you familiar with VBA, yet looking for some interesting ideas?

This course is designed for those of you with such concerns. The examples shown will be written in both Visual LISP and VBA. This will help you understand how the ActiveX interface provides power to both languages. Also, we will discuss how to interpret the ActiveX Reference into Visual LISP.

Please consider these examples as a framework upon which you can build when you return to the office. These examples work fine in the context of this course but may need additional programming in a working environment.

Auto-rotating attributes upon insertion

Many new programmers are confused on how to access attributes using either Visual LISP and VBA. The method of accessing attributes is different in traditional AutoLISP compared to VBA. So it would behoove more experienced programmers to consistently use the ActiveX method of accessing attributes in both Visual LISP and VBA.

The following VBA example demonstrates a simple event handler that watches for newly inserted blocks and rotates all the attributes to the current UCS 0 angle. Therefore, it shows how you access an inserted block’s attributes.

```vba
Option Explicit

Dim blkAdded As AcadBlockReference
Private Sub AcadDocument_EndCommand(ByVal CommandName As String)
    Select Case CommandName
        Case "INSERT"
            If Not (blkAdded Is Nothing) Then
                If blkAdded.HasAttributes Then FixAttributes blkAdded
            End If
    End Select
    Set blkAdded = Nothing
End Sub

Private Sub AcadDocument_ObjectAdded(ByVal Object As Object)
    If TypeOf Object Is AcadBlockReference Then
        Set blkAdded = Object
    End If
End Sub

Function FixAttributes(Block As AcadBlockReference) As Boolean
    With ThisDrawing
        Dim ucsAngle As Double
    End With
    Dim allAtts As Variant
    allAtts = Block.GetAttributes
    Dim didAllAttsFix As Boolean
    didAllAttsFix = True
    On Error GoTo Trouble
    Dim i As Integer
    For i = 0 To UBound(allAtts)
        allAtts(i).Rotation = ucsAngle
        didAllAttsFix = didAllAttsFix And True
    Next i
    FixAttributes = didAllAttsFix
    Exit Function
    Trouble:
        Resume
    End Function
```
Trouble:
Select Case Err.Number
Case -2145386413 ' locked layer
didAllAttsFix = False
Resume Next
Case Else
Debug.Print Err.Description, vbCritical, "AutoFixAtts"
End Select
End Function

Now that you have seen the VBA approach that must use the ActiveX interface, here is a similar approach in Visual LISP. The reactor, once loaded, can be unloaded and then reloaded. The code is very similar to the VBA approach so it will make an interesting comparison.

```lisp
(defun vlr:EndCommand (reactor callback)
  (cond (and (= (car callback) "INSERT") *BlockAdded"
    (FixAttributes *BlockAdded*)))
  (setq *BlockAdded* nil)))

(defun vlr:BlockAdded (reactor callback / objBlk)
  (cond (eq (vla-Get-ObjectName
    (setq objBlk (vlax-EName->vla-Object (cadr callback)))))
    "AcDbBlockReference"
  (setq *BlockAdded* nil)))

(defun FixAttributes (objBlk / dblAngle objAtts intCount lstAtts lstResult)
  (setq dblAngle (angle (getvar "UCSOrg") (getvar "UCSXDir"))
  objAtts (vlax-Variant-Value (vla-GetAttributes objBlk)))
  (cond (> (vlax-SafeArray-Get-U-Bound objAtts 1) -1)
    (repeat (setq intCount (length (setq lstAtts (vlax-SafeArray->List objatts))))
      (setq intCount (1- intCount)
      lstResult (cons (not (vl-catch-all-error-p
      (vl-catch-all-apply
        'vla-Put-Rotation
        (list (nth intCount lstAtts) dblAngle)))))
      lstResult))))

(defun C:AutoFixAttributes ()
  (vl-load-com)
  (princ "AutoFixAttribute reactors are ")
  (cond (not (and *vlrEndCommand* *vlrBlockAdded*))
    (setq *vlrEndCommand* (vlr-Editor-Reactor nil '((:vlr-CommandEnded . vlr:EndCommand)))
    *vlrBlockAdded* (vlr-AcDb-Reactor nil '((:vlr-ObjectAppended . vlr:BlockAdded)))
  (princ "loaded.\n")
  (and (vl-r-Add-P *vlrEndCommand*) (vl-r-Added-P *vlrBlockAdded*))
  (vl-r-Remove *vlrEndCommand*)
  (vl-r-Remove *vlrBlockAdded*)
  (princ "unloaded.\n")
  (vl-r-Add *vlrEndCommand*)
  (vl-r-Add *vlrBlockAdded*)
  (princ "reloaded.\n")
  (princ))

(defun C:AFA () (C:AutoFixAttributes))
```

When you try these samples back at the office, consider the following enhancements:

- Other ways that blocks get inserted
- Multiple insertions occurring during one command
- Providing for attributes that should align with the block insertion
- Auto-rotating only specific blocks

**Setting a table’s cells to use a background to mask objects**

You can modify a table to add a background mask using DXF codes. However, the process is easier and more understandable using ActiveX. This is a prime example of how learning to use the ActiveX interface will improve your code.
First, the VBA code. Notice how the code documents itself because of the use of descriptive variable names and the ActiveX object model. This code is easier to read than a straight AutoLISP function using DXF codes would be.

```vba
Option Explicit

Sub SetTableToBackground(Table As AcadTable)
    Dim newColor As AcadAcCmColor
    Set newColor = New AcadAcCmColor
    newColor.SetRGB 255, 255, 255
    With Table
        Dim row As Long, column As Long
        For row = 0 To .Rows - 1
            For column = 0 To .Columns - 1
                If .GetCellBackgroundColorNone(row, column) = True Then
                    .SetCellBackgroundColorNone row, column, False
                    .SetCellBackgroundColor row, column, newColor
                End If
            Next column
        Next row
    End With
    Set newColor = Nothing
    Table.Update
End Sub
```

Once again, you can see that the Visual LISP approach is not all that different from the VBA version. The advantages of programming using the ActiveX interface become evident as you start doing more cross-language programming.

```lisp
(defun set-table-to-background (table / new-color rows-count cols-count row column)
    (setq new-color (vla-get-true-color table)
          rows-count (vla-get-rows table)
          cols-count (vla-get-columns table)
          row rows-count)
    (repeat rows-count
        (setq row (1- row)
              column cols-count)
    (repeat cols-count
        (setq column (1- column))
        (if (= (vla-get-cell-background-color-none table row column) :vlax-true)
            (prog1 (vla-set-cell-background-color-none table row column :vlax-false)
                  (vla-set-cell-background-color table row column new-color)))
    (vla-update table)
    (princ))
```

Even in Visual LISP your code becomes more self-documenting thanks to the ActiveX interface. The function shown above does not require additional comments to clarify what each statement is doing. This would not be true of traditional AutoLISP code that requires the use of DXF codes.

**Deleting layer filters automatically**

Layer filters are a welcome enhancement to AutoCAD. However, prior to AutoCAD 2004, layer filters from other drawings would be added when those drawings were inserted or XRef’d. Some drawings would end up with thousands of undesired layer filters. Happily, that problem was fixed. However, there are still many drawings out there with all these undesired layer filters. AutoCAD 2005 gives you an easy way to delete these filters in the Layer Manager. But who wants to do this by hand? And what about layer filters that you want to keep, somewhere in the 1000’s of filters?

AutoCAD 2005 stores layer filters in two extension dictionaries attached to the Layers collection. The following procedures delete the filters from both dictionaries that do not match the “keep” filter.

```vba
Option Explicit

Private Function GetDictionary(DictionaryName As String) As AcadDictionary
    On Error Resume Next
    Set GetDictionary = ThisDrawing.Layers.GetExtensionDictionary(DictionaryName)
End Function
```
Private Function LayerFiltersDelete(FiltersToKeep As String) As Boolean
Dim xRec As AcadXRecord
Dim pre2005 As Boolean, post2005 As Boolean
Dim myDict As AcadDictionary
Set myDict = GetDictionary("ACAD_LAYERFILTERS")
If Not (myDict Is Nothing) Then
    For Each xRec In myDict
        If Not (xRec.Name Like FiltersToKeep) Then xRec.Delete
    Next xRec
    pre2005 = True
End If
Dim dxfCodes As Variant, dxfData As Variant
Dim i As Long
Set myDict = GetDictionary("AcLyDictionary")
If Not (myDict Is Nothing) Then
    For Each xRec In myDict
        xRec.GetXRecordData dxfCodes, dxfData
        For i = 0 To UBound(dxfCodes)
            If dxfCodes(i) = 300 Then Exit For
        Next i
        If Not (dxfData(i) Like FiltersToKeep) Then xRec.Delete
    Next xRec
    post2005 = True
End If
LayerFiltersDelete = (pre2005 And post2005)
End Function

Public Sub LFD()
Dim keepFilters As String
keepFilters = ThisDrawing.Utility.GetString(False, 
    "Wildcard mask for filters to keep, or <Enter> to delete all: ")
Call LayerFiltersDelete(keepFilters)
End Sub

The Visual LISP approach has an advantage over the VBA version, in that it will permit multiple “keep” filters. This is a prime example of not letting your language preference get in the way of using the best tool for the job.

(defun rrbI:LFDGetDictionary  (objXDict strDict / objLFDict)
(vl-Catch-All-Apply
 (function (lambda () (setq objLFDict (vla-Item objXDict strDict))))
objLFDict))

(defun rrbI:LayerFiltersDelete  (strKeepWC / objXDict objDict i)
;; This function insures that an Extension Dictionary exists,
;; and works on both locations for layer filters
(vl-load-com)
(vl-Catch-All-Apply
 (function (lambda ()
   (setq objXDict (vlax-vla-Object->EName objXRec)))))
(cond (objXDict
   (cond ((setq objDict (rrbI:LFDGetDictionary objXDict "AcLyDictionary"))
       (setq i 0)
       (vlax-for objXRec objDict
       (cond ((not (and strKeepWC
               (wcmatch (cdr (assoc 300 (entget (vlax-vla-Object->EName objXRec))))
               strKeepWC)))
           (setq i (1+ i))
           (vlax-Delete objXRec)))))
   (princ (strcat "\n" (itoa i) " 2005-based filters deleted.")))))
(cond (objXDict
   (cond ((setq objDict (rrbI:LFDGetDictionary objXDict "ACAD_LAYERFILTERS"))
       (setq i 0)
       (vlax-for objXRec objDict
       (cond ((not (and strKeepWC
               (wcmatch (vla-Get-Name objXRec) strKeepWC)))
           (setq i (1+ i))
           (vlax-Delete objXRec)))))
   (princ (strcat "\n" (itoa i) " legacy filters deleted."))))))
ActiveX Tricks for Visual LISP and VBA

(defun C:LFD (/ inpKeep)
  (setq inpKeep (getstring
    "\nWildcard mask for filters to keep, or <Enter> to delete all: ")
  (rrbi:LayerFiltersDelete (cond (\=/ inpKeep "") inpKeep)))
  (princ))

The code presented has the weakness of being incapable of deleting layer filters touched by vertical products such as ADT or LDD. Regardless, it does provide a mechanism to delete layer filters in many drawings.

**Transient context menu editing**

Context shortcut menus are a powerful tool. All those common tasks that you perform in a specific context are right there with a right-click of the mouse. And editing menus is easy to do. In fact, it is so easy to edit a menu in the normal manner, by using a text editor to edit the .mns file directly, that you may wonder why this course introduces an alternate approach; to use code to add or remove items to context-sensitive menus.

One reason why you may consider this approach is: Most context-sensitive menus still reside in the default Acad menu in many offices. Quite often the Acad menu is considered “off-limits” by the CAD Manager and you may not be permitted to modify it. Granted, a context-sensitive menu may be copied to a partial menu for further editing. However, this may be overkill if all you want to do is add one item a single context-sensitive menu. And if you want to remove an item from a context-sensitive menu, without permanently changing the menu, what can you do? You can use code to perform these tasks without editing the menu files.

Another reason might be that you want to provide a context-sensitive menu item only when particular conditions exist, such as the user working on a particular type of drawing. The changes made to a context-sensitive menu via the code presented here are not permanent; they are only active during the current AutoCAD session.

The following VBA code should be placed in a class module named “ContextMenu”.

```
Option Explicit

Private Function ItemIsThere(Name As String, Menu As AcadPopupMenu) As Long
    ItemIsThere = -1
    Dim i As Long
    Dim menuItem As AcadPopupMenuItem
    For Each menuItem In Menu
        If menuItem.Label = Name Then
            ItemIsThere = i
            Exit For
        End If
        i = i + 1
    Next menuItem
End Function

Public Sub List()
    Dim aMenu As AcadPopupMenu
    With ThisDrawing.Utility
        For Each aMenu In AcadApplication.MenuGroups("ACAD").Menus
            If aMenu.Name Like "Context" Then
                .Prompt vbCrLf & aMenu.Name
                .Prompt vbCrLf & .GetLong(ItemIsThere("", aMenu)) & vbCrLf
            End If
        Next aMenu
    End With
End Sub
```
Public Sub Add(Menu As String, Label As String, Macro As String)
Dim myMenu As AcadPopupMenu
Set myMenu = GetPopMenu(Menu)
If Not (myMenu Is Nothing) Then
    If ItemIsThere(Label, myMenu) = -1 Then
        Dim lastChar As String
        lastChar = Right$(Macro, 1)
        If Not (lastChar = " " Or lastChar = ";") Then
            Macro = Macro & " 
        End If
        Macro = Replace(Expression:=Macro, Find:="^C", Replace:=Chr(27), Compare:=vbTextCompare)
        myMenu.AddMenuItem myMenu.Count, Label, Macro
    End If
End If
End Sub

Public Sub Remove(Menu As String, Label As String)
Dim myMenu As AcadPopupMenu
Set myMenu = GetPopMenu(Menu)
Dim i As Long
If Not (myMenu Is Nothing) Then
    i = ItemIsThere(Label, myMenu)
    If i <> -1 Then
        myMenu.Item(i).Delete
    End If
End If
End Sub

Here is some sample code that can be located in either the ThisDrawing object or a standard module in the same VBA project as the class module.

Option Explicit

Sub ListMenus()
Dim myMenu As ContextMenu
Set myMenu = New ContextMenu
myMenu.List
End Sub

Sub AddZoomObject()
Dim myMenu As ContextMenu
Set myMenu = New ContextMenu
myMenu.Add "Context menu for edit mode", "Zoom Object", "Zoom_Object"
End Sub

Sub RemoveZoomObject()
Dim myMenu As ContextMenu
Set myMenu = New ContextMenu
myMenu.Remove "Context menu for edit mode", "Zoom Object"
End Sub

Because the ActiveX interface is usable by both VBA and Visual LISP we can take the same approach and write it in Visual LISP. It is interesting that, in this case, the Visual LISP code is longer than the VBA code.

Written by: R. Robert Bell
Copyright © 2004 by R. Robert Bell

You may use the (I:AddToContextMenu) function to add additional items to any context menu. Just follow the sample for the Zoom Object item.

You may use the (I:RemoveFromContextMenu) function to remove items from any context menu. Just follow the sample for the Options item.

You may get a list of the context menu names by using:
(I:ListMenus)
ActiveX Tricks for Visual LISP and VBA

;;; Subr to return object if menu item exists
(defun I:IsItemThere (myItem myPopMenu / myItemName result)
  (cond ((= (type myItem) 'STR) ; if using string to search on
      (setq myItemName (strcase myItem)); convert search string to uppercase
      (vlax-for aItem myPopMenu ; loop thru all items in menu
        (cond ((= (strcase (vla-get-Label aItem)) myItemName)
          (setq result aItem))))) ; if item found, save object
    ((vl-catch-all-apply ; searching on index number, trap errors
      (function (lambda ()
        (setq result (vla-Item myPopMenu myItem))))) ; attempt to save object
      result)); return object if found

;;; Subr to replace all occurrences of substring in string
(defun I:SubstAll (new old str)
  (cond ((vl-String-Search old str) ; if substring is found
      (I:SubstAll new old (vl-String-Subst new old str)))
    (str)))); if substring not found, return current string

;;; Subr to format string as an ActiveX-compatible menu macro string
(defun I:FormatMacro (macro)
  (cond (not
    (member (last (vl-string->list macro))
      (list (ascii "") (ascii ";")))); if last char of macro not an <Enter>
    (setq macro (strcat macro " "))); then add an <Enter>
    (setq macro (I:SubstAll " " ';' macro)) ; replace all semi-colons with spaces
    (I:SubstAll (chr 27) '^C' (I:SubstAll '^C' '^C' macro)))); replace all ^C's with ASCII cancels

;;; Primary function to list all context-sensitive menus and their items.
;;; Listing will include the current ActiveX item index number.
(defun I:ListMenus (/ acMenu menuName menuCount itemCount)
  (vl-load-com)
  (setq acMenu (vla-get-Menus ; get Acad menu object
    (vla-Item (vla-get-MenuGroups (vlax-Get-Acad-Object)) "ACAD")))
  (vlax-for aMenu acMenu ; loop thru all menus in object
    (setq menuCount (cond (menuCount (1+ menuCount)) ; increment count
      0))) ; initialize counter
  (cond ((wcmatch ; if a context-sensitive menu
      (strcase (setq menuName (vla-get-Name aMenu)))
        "CONTEXT", "GRIPS CURSOR", "OBJECT SNAP")
    (princ (strcat "\n" ; print...
      (itoa menuCount) ; current ActiveX index number...
      (cond (\(< menuCount 10) " . \t"); zero-pad index number
        (\begin{tabular} {c}
          \(\text{\footnotesize menu name}\)
        \end{tabular})); current menu's name
      (vlax-for aItem aMenu ; loop thru menu's items
        (setq itemCount (cond (itemCount (1+ itemCount)) ; increment counter
          0))) ; initialize counter
    (princ (strcat "\n\t" ; print...
      (itoa itemCount) ; current ActiveX index number...
      (cond (\(< itemCount 10) " . \t"); zero-pad index number
        (\begin{tabular} {c}
          \(\text{\footnotesize item label}\)
        \end{tabular})); current menu item's label
      (termpr)))); print a blank line

;;; Primary function to remove an item from a menu
(defun I:RemoveFromContextMenu (menuName acMenu myPopMenu myItem itemName)
  (vl-load-com)
  (setq acMenu (vla-get-Menus ; get Acad menu object
    (vla-Item (vla-get-MenuGroups (vlax-Get-Acad-Object)) "ACAD")))
  (cond ((and menu ; if menu name was provided
      (not ; if menu object is found
        (vl-catch-all-error-p
          (vl-catch-all-apply ; searching on index number, trap errors
            (function (lambda () (setq myPopMenu (vla-Item acMenu menu)))))
          (setq myItem (I:IsItemThere name myPopMenu))))) ; if menu item object is found
    (setq itemName (vla-get-Caption myItem)); save for success message
    (vDelete myItem)); remove menu item from menu
    (princ (strcat "\n" ; print success message
      (itemName " has been removed from the "
      (vla-Get-Name myPopMenu) "")))); clean exit

(princ) ; clean exit

AUTODESK
UNIVERSITY*
2004
The ability to affect context menus in a transient fashion is impossible without the ActiveX interface. Also, the presented code is not necessarily limited to context-sensitive menus but can be modified to work with all POP menus!

**New insertion properties same as first instance**

Quite often when inserting the same block over and over, you want to apply the same properties to each insertion. It is possible to do this without worrying about the state of the current properties. The following example uses a combination of VBA and Visual LISP to perform this task.
ActiveX Tricks for Visual LISP and VBA

Set objBlockProto = FindBlock(objBlockItem)
On Error Resume Next
  If blnFlag And Err.Number = 0 Then
    On Error Resume Next
    objBlockItem.Layer = objBlockProto.Layer
    If Err.Number = 0 Then
      objBlockItem.XScaleFactor = objBlockProto.XScaleFactor
      objBlockItem.YScaleFactor = objBlockProto.YScaleFactor
      objBlockItem.ZScaleFactor = objBlockProto.ZScaleFactor
    End If
  End If
End If
Next objBlockItem
strHandleAppend = ""
End If
End Sub

Private Sub ACADApplication_SysVarChanged(ByVal SysvarName As String, ByVal newVal As Variant)
  If UCase(SysvarName) = "*USERS5" Then
    If Left(newVal, 12) = "Blockprops: " Then
      strHandleAppend = Mid(newVal, 13)
    End If
  End If
End Sub

Function FindBlock(objBlockItem As Object) As Object
' MsgBox "FindBlock"
Dim intCount As Integer
Dim colBlocks As Object
Dim objBlock As Object
Dim objBlockProto As Object
intCount = 0
Set colBlocks = ThisDrawing.ActiveLayout.Block
While intCount < colBlocks.Count
  Set objBlock = colBlocks.Item(intCount)
  intCount = intCount + 1
  If objBlock.ObjectName Like "AcDbBlockReference" Or objBlock.ObjectName Like "AcDbMInsertBlock" Then
    If UCase(objBlock.Name) = UCase(objBlockItem.Name) Then
      Set objBlockProto = objBlock
      intCount = colBlocks.Count
      End If
  End If
Wend
Set FindBlock = objBlockProto
End Function

(defun ACDBInsertAppendVBA (evtReactor lstCallback)
  ;; Copyright © 2004 by Peter Jamtgaard
  ;; Reactor to change new instances of inserts to be like existing instances
  ;; Database Insert Append reactor
  ;; Create a list of newly appended inserts
  (defun ACDBInsertAppend (evtReactor lstCallback)
    (printf "\nACDBInsertAppend: \n")
    (cond ((and (wcmatch (vla-get-objectname (vlax-ename->vla-object (cadr lstCallback))) "AcDbBlockReference,AcDbMInsertBlock")
                (= (substr (getvar "USERS5") 1 15) "BlockpropsVBA: ")
            (setq strHandleAppend (vla-get-handle objBlock)))
      (printf strVBAFile (findfile "blockprops.dvb"))
    )
  )
)

The following code is a pure Visual LISP version.

(defun ACDBInsertAppend (evtReactor lstCallback)
  (printf "\nACDBInsertAppend: \n")
  (cond ((and (wcmatch (vla-get-objectname (vla-get-handle objBlock)) "AcDbBlockReference,AcDbMInsertBlock")
              (= (substr (getvar "USERS5") 1 15) "BlockpropsVBA: ")
          (setq strHandleAppend (vla-get-handle objBlock)))
      (printf strVBAFile "\n\n\n")
    )
  )
)

(defun ACDBInsertAppend (evtReactor lstCallback)
  (printf "\nACDBInsertAppend: \n")
  (cond ((and (wcmatch (vla-get-objectname (vla-get-handle objBlock)) "AcDbBlockReference,AcDbMInsertBlock")
              (= (substr (getvar "USERS5") 1 15) "BlockpropsVBA: ")
          (setq strHandleAppend (vla-get-handle objBlock)))
      (printf strVBAFile "\n\n\n")
    )
  )
)
The reactor/event handler combination watches for insert-type objects being added to the drawing, then waits for the command to end. Once the command has ended, the new block insertions are modified to match the properties of the first block of that name inserted after the reactors were loaded.

**Interpreting the ActiveX Reference for Visual LISP**

Thinking about objects, properties, and methods is foreign to the traditional AutoLISP programmer. The lack of straightforward documentation on the *vla-* functions compels many to avoid the ActiveX interface. However, Autodesk knew that the “translation” of the documented VBA documentation to Visual LISP was so easy that there was no real need to duplicate the documentation unnecessarily.

Take a moment to look at the Object Model map in the Reference. This map is a great way to start a session in the documentation. You can click on any of the objects or collections to go directly to that
ActiveX Tricks for Visual LISP and VBA

object/collection’s documentation. It will also give you an understanding of the hierarchy of the objects in the ActiveX interface. Notice how the Application object is the start for most of the object model.

Once you have the Object Model map displayed, select the Layer object. This screen lists all the methods and properties (also events; we won’t be discussing those) available to the object or collection, in this case the Layer object. You will see nothing about Visual LISP here. However, note these following three rules:

• All the methods you see are available in Visual LISP by prefixing the method name with vla-, e.g. vla-Delete.
• All the properties are retrievable in Visual LISP by prefixing the property name with vla-Get-, e.g. vla-Get-Description.
• All the properties that are not read-only are modifiable in Visual LISP by prefixing the property name with vla-Put-, e.g. vla-Put-Description.

As you can see, up to this point the effort to “translate” the ActiveX Reference is trivial.

Now, select the Description property and examine the documentation provided. The Visual LISP statement shown in the graphic shows you how the documentation is “translated”. Notice how you prefix the property name, provide the object as the first argument to the Visual LISP function, and then provide the value for the property.

Methods will usually require more arguments. All you need to do is provide the arguments in the order documented. Just remember that for the Visual LISP function the first argument is always the object to which the method applies and the documented arguments follow.

Also, some methods will output data. The documentation lists those as output-only. These output-only arguments need a quoted variable in which the data will be placed. After the Visual LISP statement is run the variable(s) quoted will hold data that your programs can examine and use.

Hopefully this short introduction into how to “translate” the ActiveX Reference will open the doors to your programming in Visual LISP using the ActiveX interface.

Conclusion

There is much power to be gained from the use of the ActiveX interface. As you grow more comfortable programming thru the interface your abilities, to use either language to best accomplish your tasks, will improve.

Please enjoy the rest of Autodesk University!