Autodesk University 2008

Good Habits for Coding in Visual LISP®

R. Robert Bell – Sparling

CP319-1

The power of AutoCAD® lies in its customization capabilities. Visual LISP is a powerful tool for expanding your options. Unhappily, it is easy to have a “scatter-shot” approach to the way you write code. This course will broaden your horizons regarding good coding practice. You will learn the importance of writing modular code. You will see how to identify portions of code that belong to subroutines and avoid the trap of monolithic applications. Toolbox routines for your own code are important to reducing the time spent writing the overall application. Some extremely useful toolbox routines will be discussed. Bring your own ideas for toolbox routines and we will discuss them.

About the Speaker:

Robert is the Design Technology Manager for Sparling, the largest specialty electrical engineering and technology consulting firm in the United States, located in Seattle Washington. He provides strategic direction, technical oversight, and high-level support for Sparling’s enterprise design and production technology systems. He is instrumental in positioning Sparling as an industry and client leader in leveraging technology in virtual building and design. Robert has been writing AutoLISP® code since the release of AutoCAD v2.5, and VBA since introduced in R14. He has customized applications for the electrical/lighting, plumbing/piping, and HVAC disciplines. Robert has also developed applications for AutoCAD as a consultant. A former member of the Board of Directors for AUGI®, he is active on AUGI forums and Autodesk discussion groups.
rbell@sparling.com

Autodesk®

Introduction

Let's face it. Many people pick up programming in Visual LISP by coping posted code and simply tweaking it to fit their needs. This class is going to explore some topics that will help take Visual LISP code to the next level.

Self-Documenting Code, or The Holy Grail of Programming

Your main code should be concise and brief enough so as to be self-documenting. Indeed, most of the code you write should reach this goal. Comments should only be required for sections of code that perform complex operations. This brings up comment formats. There are 4 different styles. Most common are the single- and triple- semi-colon styles. The other two formats are double- semi-colon and in-line.

- Single semi-colon, used to comment a line of code on the same line, to the right of the code
- Double semi-colon, comment on a separate line, at the same indent as the code above the comment
- Triple semi-colon, comment on a separate line, at the left margin
- In-line (;|<comment> |;), comment embedded in the middle of code

The in-line comment is particularly useful for creating code headers. The delimiters can be on separate lines. Many programmers make verbose comments at the beginning of the file. However, you will often see the triple- semi-colon style used. Use the in-line style and avoid the line wrap issues that occur with the triple- semi-colon style.

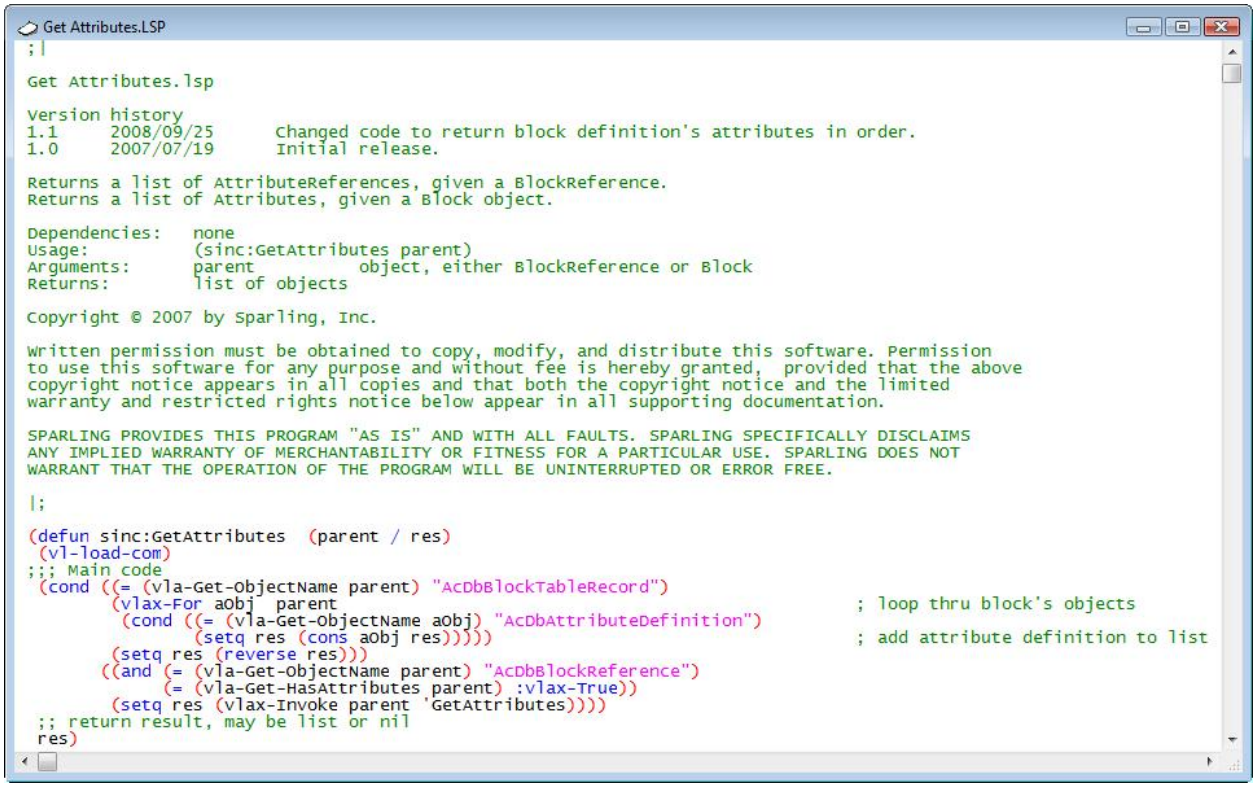

Figure 1

Notice the function name in Figure 1. Note that it has a prefix, similar to “c:”. You may use any prefix you desire. This makes it easy to determine functions that have been developed by you in

support of the main code. For example, if you prefix all your “toolbox” functions with “i:” you can easily search a large application for any statements that may be using external toolbox functions. Also, these prefixes may help you when posting code on forums or discussion groups for support. As you get ready to post your code you may notice those toolbox functions due to their prefix and include those external functions. This makes it easier to get support.

Look for sections of code that perform a specific task. Code that is a prime candidate to make into a function is code that can be boiled down to a single task. For example, you may have a section of code that is related to returning a list of the attributes for an inserted block. This could be converted into a function called GetAttributes that takes an inserted block as an argument.

Most functions should be designed so that they return something upon success and nil otherwise. This permits them to be used as test expressions. For example, when the GetAttributes function is passed a block reference that doesn't contain any attributes it will return nil. You can then use the function to verify your data before continuing your code.

```
(defun sinc:GetAttributes  (parent / res)
 (vl-load-com)
(cond ((= (vla-Get-ObjectName parent) “AcDbBlockTableRecord”)
        (vlax-For aObj  parent
         (cond ((= (vla-Get-ObjectName aObj) “AcDbAttributeDefinition”)
                (setq res (cons aObj res)))))
        (setq res (reverse res)))
       ((and (= (vla-Get-ObjectName parent) “AcDbBlockReference”)
             (= (vla-Get-HasAttributes parent) :vlax-True))
        (setq res (vlax-Invoke parent 'GetAttributes))))
res)
```

You can take the above example even further. It is far better to validate all your data before executing your full code. This avoids the kludge of using *quit* or *exit*. The use of the *and* function makes this very elegant. The *and* function will evaluate its expressions in order, and drop out at the first expression that returns *nil*. It will not continue to process the rest of its expressions. Therefore, you may verify each piece of data within a single *and* statement in the sequential order of gaining the data.

In the example below, the code is going to check four different data before running the main expressions. If any of the data returns *nil* the test will fail and the *cond* will go to another test. The first test expression is using the result of user-defined function and is a perfect example of designing a function to return *nil* when there is no valid data. The last test expression is noteworthy because it is executing code within the *and* statement that would usually been seen outside the *cond* expression.

```
 (cond ((and (setq chkDWT (sinc:GetTemplate))
             (setq chkPlot (wcmatch chkDWT “Sparling*x*.dwt”))
             (setq chkBlank (<= (vla-Get-Count (vla-Get-ModelSpace (sinc:ThisDoc))) 1))
             (setq fileName (getfiled “Select Architect's Titleblock”
                                      (getvar “DwgPrefix”)
                                      “dwg”
                                      16)))
        <main expressions>
```

So why not just perform the *setq* statements outside the *cond* expression? Because there is no reason to display a dialog box asking the user to select a file if any of the first three conditions fail. In fact, there is no reason to check the template name when there is no template since *wcmatch* would cause an error if *sinc:GetTemplate* returned nil. There is no reason to check for 1 or less objects in ModelSpace if the template is not correct.

Making a Toolbox, or What Did I Do with That Hammer?

A few very useful “toolbox” routines are described below. One formats a prompt for the user to provide input and return the user's selection, including a default value. The next one provides a

prompt only when a specific flag is set. The final one is a robust error handler that handles normal exits and also dumps detailed information when a specific flag is set.

Toolbox: Prompting a User for Specific Input

The process of initializing a list of keywords, formatting the prompt string, and dealing with a default entry is the same no matter how many different times you use the code. Since the process involves the same statements and tests, with only a few variables, it makes perfect sense to create a function to wrap the process.

```
(defun i:GetInp  (myPrompt myOptions / inpDefault inpOptions inpPrompt)
 (initget 0 myOptions)
 (setq inpDefault (substr myOptions 1 (vl-string-position (ascii " ") myOptions)))
 (setq inpOptions (vl-string-left-trim inpDefault myOptions))
 (setq inpPrompt (strcat myPrompt
                         " ["
                         inpDefault
                         (vl-string-translate " " "/" inpOptions)
                         "] <"
                         inpDefault
                         ">: "))
 (cond ((getkword inpPrompt))
       (inpDefault)))
```

Toolbox: Debug.Print

VBA provides a useful method called Debug.Print to print messages only to the IDE for the programmer to view. Visual LISP does not have such a mechanism yet the concept of printing messages for debugging is useful. Imagine being able to tell a user over the telephone to "flip a switch" and then run the code that is having a problem. Messages then appear at the command prompt to verify data or code execution. The user may then "turn the switch off" to return the code to normal prompts.

Setting a blackboard variable
(vl-bb-set '*Debug* T)

The code checks if a Visual LISP blackboard namespace variable named **Debug** has been set. It then takes the argument and checks if it is supposed to print on a new line. If so, it prefixes "Debug> " to the message to help users differentiate normal messages from debug messages. When the blackboard namespace variable is not set the function returns T so that it won't interfere with test expressions if it is located in a test expression.

```
(defun i:DebugPrint  (msg)
 (cond ((vl-bb-ref '*Debug*)
        (princ
         (cond ((= (substr msg 1 1) "\n") (strcat "\nDebug> " (substr msg 2)))
               (msg))))
       (T)))
```

Toolbox: Error Handling in the New Millennium

Error handling in Visual LISP is more robust than in early versions of AutoLISP. Prior to AutoCAD 2000, you had to save the current error handler to a variable, define the new error handler, and finally restore the original error handler from the variable at the end of your code. If you have code similar to the code below, you are not using Visual LISP to its full advantage.

```
;;; R14 error handling
(defun newErr  (msg)
 (princ (strcat "newErr> " msg))
 (cond (oldHi (setvar "Highlight" oldHi)))
 (setq *Error* oldErr))
```

```
(defun C:TestR14  (/ oldErr oldHi inp1 inp2)
(setq oldErr  *Error*
      *Error* newErr
      oldHi   (getvar "Highlight"))
(setvar "Highlight" 0)
(princ (strcat "\nHighlight is set to " (itoa (getvar "Highlight"))))
(setq inp1 (getdist "\nSpecify numerator: "))
(setq inp2 (getdist "\nSpecify denominator: "))
(princ (/ inp1 inp2))
(setvar "Highlight" oldHi)
(setq *Error* oldErr)
(princ))
```

That sort of code was necessary in early versions of AutoCAD because, even if you attempted to declare the error handler to be local, it would become global as soon as it ran. So it was necessary to save the original global error handler, define the new error handler (which would become global as soon as it ran, regardless of it being declared local), and reset the original error handler at the end of the new one. All of that is unneeded in most AutoCAD installations at this time.

Visual LISP error handlers remain local when they are defined as local and execute. Therefore, creating a reliable local error handler for a specific application is simple. Not only that, but they can also be used for a normal exit from the application. Think about it for a moment. The same tasks you perform when normally exiting an application are the same tasks you will want to perform when handling an error. Tasks such as returning the environment to the original state need to run in both cases. All that is different is how you handle errors themselves. In most cases, you are going to report the error and that's all. So the difference between a function for normal exit and abnormal exit boils down to just a print statement.

With that in mind, note the following code. There is a separate routine called *i:Exit* for exiting the application. The function checks the *msg* variable and, if it is nil, reports no error. If *msg* is bound, there was an error, so it is reported unless it was simply the user hitting the Escape key. After that, regardless of the state of the *msg* variable, the remainder of the code runs, performing the restoration code. The *i:Exit* function then ends with a *princ* statement which presents a clean exit from the routine, either in an error condition or normal exit condition. Note that the main code calls the *i:Exit* function with nil as the argument as the last statement of its code. There is no need for a *princ* statement since the *i:Exit* function will provide it.

In order for the i:Exit routine to run as an error handler, the variable **Error** is declared as local in the main application and then bound to the *i:Exit* function. This causes *i:Exit* to act as an error handler, yet it remains local to the application. The global error handler is unaffected.

```
;;; 2000 error handling
(defun i:Exit  (msg)
 (cond ((not msg)) ; normal exit
       ((member msg '("Function cancelled" "quit / exit abort"))) ; <esc> or (quit)
       ((princ (strcat "\nError: " msg)) ; fatal error, display it
        (cond ((vl-bb-ref '*Debug*) (vl-bt)))) ; if in debug mode, dump backtrace
 (cond (oldHi (setvar 'Highlight oldHi))) ; restore original setting
 (princ)) ; clean exit
```

```
(defun C:TestR17  (/ *Error* oldHi inp1 inp2)
(setq *Error* i:Exit
      oldHi   (getvar 'Highlight))
(setvar 'Highlight 0)
(princ (strcat "\nHighlight is set to " (itoa (getvar 'Highlight))))
(setq inp1 (getdist "\nSpecify numerator: "))
(setq inp2 (getdist "\nSpecify denominator: "))
(princ (/ inp1 inp2))
(i:Exit nil))
```

Look again at the error handler. There is a statement that checks for the state of the blackboard namespace variable **Debug**. If it is not nil then the *vl-bt* function is run. This is Visual LISP's backtrace function and will display the statement that caused the error. Reading the backtrace is not trivial, yet you can usually isolate the cause. Look at the following backtrace from running

the above code and providing zeros as both arguments. Find the line that begins with ":ERROR-BREAK". The line that follows is the line in the code that caused the error.

```
Backtrace:
[0.50] (VL-BT)
[1.46] (I:EXIT "divide by zero") LAP+119
[2.40] (_call-err-hook #<USUBR @13b28cf8 I:EXIT> "divide by zero")
[3.34] (sys-error "divide by zero")
:ERROR-BREAK.29 "divide by zero"
[4.26] (/ 0.0 0.0)
[5.20] (C:TESTR17) LAP+135
[6.15] (#<SUBR @13b28d5c -rts_top->)
[7.12] (#<SUBR @1340a35c veval-str-body> "(C:TESTR17)" T #<FILE internal>)
:CALLBACK-ENTRY.6 (:CALLBACK-ENTRY)
:ARQ-SUBR-CALLBACK.3 (nil 0)
```

Autoloading, or "I'm Sick and Tired of Loading My Tools"

Another advantage to modularizing your code is that you can save these functions that perform distinct tasks as a library of functions and use them in new code. The key to doing this is to have a good autoloading function. Autodesk provides an *autoload* function in Acad2009Doc.lsp but this function has a few limitations. (Note that it is still very useful for loading ARX applications!)

- It only creates command-line functions
- It cannot create functions that take arguments
- It cannot load .vlx or .fas files before finding a .lsp file of the same name
- It cannot accept subfolders in the filename specification
- It does not warn you if the target file does not exist when making the stub code
- It will fall into an endless loop when a function doesn't exist in the target file

Figure 2 on the next page is used to provide the narrative for what the code does. This is a case of code that is not self-documenting without additional comments. The code in the figure is replicated as text in the appendix for copy and paste, without some of the comments seen in Figure 2.

The i:FindApp function searches for the 3 supported lisp file formats, in the order of compiled to source code. The first condition is a provision for the person writing code. If they set the environment variable *AcadCode* to the folder holding their source code, all auto-loading code will load from that source folder or its subfolders. This makes it easy to keep the source code in a location different than the production code. Yet the programmer's code will always be loading from the source code folder so they can thoroughly test the code before releasing it to production.

The *i:AutoLoad* function takes care of all of the deficiencies in Autodesk's *autoload* function.

- It can create normal functions in addition to command-line functions
- It can create functions that take arguments
- It can load .vlx or .fas files first
- It can accept subfolders in the filename specification, e.g. a Toolbox folder
- It warns you if the target file does not exist when making the stub code
- It will not fall into an endless loop when a function doesn't exist in the target file, rather, it displays an alert to warn the programmer

```
(defun i:FindApp  (fileName / res)
 (cond ((and (getenv "AcadCode")                                        ; if override path specified
             (setq res (findfile (strcat (getenv "AcadCode") "\\" fileName ".lsp"))))
        res)                                                            ; use it
       ((findfile (strcat fileName ".vlx")))
       ((findfile (strcat fileName ".fas")))
       ((findfile (strcat fileName ".lsp")))
       ((findfile fileName))))

(defun i:AutoLoad  (data / fileName funcs warnUser fqnFilename)
 (mapcar 'set '(fileName funcs) data)
 (setq warnUser (strcat "Unable to load " fileName "."))
 (cond ((setq fqnFileName (i:FindApp fileName))
        (mapcar (function                                               ; loop thru provided function names
                  (lambda (aFunc)                                       ; create wrappers for each function
                   (eval                                                ; execute the created wrappers to activate them
                    (cond ((= (type aFunc) 'LIST)                       ; function w/arguments
                           (list 'defun-q                               ; start function
                                 (nth 0 aFunc)                          ; add function name
                                 (nth 1 aFunc)                          ; add arguments
                                 ;; The below statement is needed to avoid
                                 ;; endless loops where there is a bad
                                 ;; function name but the filename exists.
                                 (list 'setq (nth 0 aFunc) nil)         ; clear function name to avoid endless loop
                                 (list 'load fqnFileName warnUser)      ; add load statement
                                 ;; If the load is successful when the wrapper
                                 ;; runs, the function name is once again bound.
                                 ;; The first condtion will execute it.
                                 ;; Otherwise, the variable is still nil and
                                 ;; the alert is thrown.
                                 (list 'cond
                                       (list (nth 0 aFunc) (cons (nth 0 aFunc) (nth 1 aFunc)))
                                       (list (list 'alert
                                                   (strcat "Cannot find function name "
                                                           (vl-symbol-name (nth 0 aFunc))
                                                           " in file "
                                                           fileName
                                                           "."))))))
                          (T                                            ; function w/o arguments
                           (list 'defun-q                               ; start function
                                 aFunc                                  ; add function name
                                 nil                                    ; no arguments, this is effectively ()
                                 ;; The code below has the same logic
                                 ;; as the above code
                                 (list 'setq aFunc nil)
                                 (list 'load fqnFileName warnUser)
                                 (list 'cond
                                       (list aFunc (list aFunc))
                                       (list (list 'alert
                                                   (strcat "Cannot find function name "
                                                           (vl-symbol-name aFunc)
                                                           " in file "
                                                           fileName
                                                           "."))))))))))
                funcs))                                                 ; return list
       (((= (logand (getvar "CmdActive") 4) 4) (alert warnUser))        ; if no script running, use alert, return nil
       ((princ (strcat "\n" warnUser)) nil)))                           ; script is running, use command line, return nil

;;; Samples
(i:AutoLoad '("BogusFile" (i:GetInp)))
(i:AutoLoad
 '("Toolbox\\Get Input" ((i:GetInp (myPrompt myOptions)) i:AmBogus)))
```

Figure 2 Autoloading Functions

Storing Data the User Cannot See, or "How Does He Do That?!"

At times you need to store data in a drawing that isn't easily accessible to the user. The best mechanism is to use Dictionaries and XRecords. Data stored in this fashion are also available to other languages used to customize AutoCAD, such as VBA and C#. This is far superior than the (vlax-ldata-*) functions. LData is not visible to VBA applications and therefore is a poor choice.

Dictionaries are primarily containers for XRecords. XRecords store the data in a DXF-style structure of group codes. The DXF Reference lists what types of data may be stored in specific group codes. You may use codes 1 thru 369, except 5, 105, and 210-239.

> Certain versions of AutoCAD may have additional undocumented limitations in usable DXF codes.

When attempting to read the data in an XRecord you would first need to check if the host dictionary exists. If it exists, attempt to retrieve the XRecord. If you get the XRecord you could trim the header data that AutoCAD attaches to the XRecord.

Setting an XRecord is similar to reading an XRecord. You first need to get the host dictionary. The difference at this point is that if the dictionary does not exist you need to create it. Once you have a valid dictionary you would search for an existing XRecord of that name and delete it if found. (You cannot overwrite an existing XRecord, you must delete the XRecord and recreate it.) At this point you may add the new XRecord to the dictionary.

It is a good idea to add a version number to your XRecord for migration purposes. This would give you the ability to migrate legacy data into the current form.

```
(defun i:ReadXRec  (dictName xrName / dictObj xrObj)
 (cond ((and (setq dictObj (dictsearch (namedobjdict) dictName))
             (setq xrObj (dictsearch (cdr (assoc -1 dictObj)) xrName)))
        (cdr (vl-member-if (function (lambda (a) (= (car a) 280))) xrObj)))))
```

```
 (defun i:SetXRec  (dictName xrName data / dictObj)
 (cond ((setq dictObj (cdr (assoc -1 (dictsearch (namedobjdict) dictName)))))
       ((setq dictObj (dictadd (namedobjdict)
                               dictName
                               (entmakex '((0 . "DICTIONARY") (100 . "AcDbDictionary")))))))
 (cond ((dictsearch dictObj xrName) (entdel (dictremove dictObj xrName))))
 (dictadd dictObj
          xrName
          (entmakex (append '((0 . "XRECORD") (100 . "AcDbXrecord")) data))))
```

```
;;; Samples
(i:ReadXRec "AU2008" "Test")
(i:SetXRec "AU2008" "Test" '((1 . "Version") (40 . 1.0)))
(i:ReadXRec "AU2008" "Test")
(i:SetXRec "AU2008" "Test" '((1 . "Version") (40 . 1.1) (60 . 1)))
```

Efficiently Opening Files, or "Warp Speed, Captain!"

ObjectDBX allows you to access drawings without loading them in the drawing editor. This results in a huge improvement in processing speed at the expense of not having the user interface. The lack of a user interface means that you cannot use selection sets. There may also be issues when modifying objects such as attributes. It is also useful for decomposing complex objects without the risk of modifying the original object or its host drawing.

The performance gains make ObjectDBX an excellent tool for querying multiple drawings. For example, many firms have developed applications that allow them to create and maintain sheet indexes across hundreds of drawings in seconds rather than hours.

ObjectDBX cannot directly work on drawings that are read-only or templates. Therefore, a simple wrapper function that detects those conditions and provides a temporary drawing file to use in such a case is useful.

```
(vl-load-com)
(load "C:\\Datasets\\CP319-1\\Toolbox\\Get Attributes.lsp")
```

```
(defun i:IsReadOnly  (fileName / fileH)
 (cond ((setq fileH (open filename "a"))
        (close fileH))
       ((not fileH))))
```

```
 (defun i:OpenDBXDoc  (fileName / newFile dbxDoc chkOpen)
 (cond ((or (i:IsReadOnly fileName)
            (= (strcase (vl-filename-extension filename)) ".DWT"))
        (setq newFile (vl-filename-mktemp "Temp .dwg"))
        (vl-file-copy fileName newFile)))
 (setq dbxDoc (vla-GetInterfaceObject
                (vlax-Get-Acad-Object)
                (strcat "ObjectDBX.AxDbDocument." (substr (getvar "AcadVer") 1 2))))
 (setq chkOpen (vl-catch-all-apply
                 'vla-Open
                 (list dbxDoc
                       (cond (newFile)
                             (fileName)))))
 (cond ((vl-catch-all-error-p chkOpen) (vlax-Release-Object dbxDoc) nil)
       (dbxDoc)))
```

```
 (defun i:CloseDBXDoc  (dbxDoc)
 (vl-catch-all-apply 'vlax-Release-Object (list dbxDoc))
 (setq dbdDoc nil))
```

```
;;; Samples
(defun C:Test1  (/ chk res)
 (setq chk (i:openDBXDoc "C:\\Datasets\\CP319-1\\TestDBX.dwg"))
 (setq res (vla-Get-TextString
            (car
             (sinc:GetAttributes
              (vla-Item (vla-Get-Block (vla-Item (vla-Get-Layouts chk) "Layout1"))
                        1)))))
 (setq chk (i:CloseDBXDoc chk))
 (princ (strcat "\nFound: " res))
 (princ))
```

```
;; Slow motion
(defun C:Test2  (/ myDocs i fileName myDoc)
 (setq myDocs (vla-Get-Documents (vlax-Get-Acad-Object)))
 (setq i 1)
 (repeat 20
  (setq fileName (strcat "C:\\Datasets\\CP319-1\\DBX Sample Drawings\\E2."
                         (itoa i)
                         ".dwg"))
  (setq myDoc (vla-Open mydocs fileName))
  (princ
   (strcat "\n"
           (vla-Get-TextString
            (car
             (sinc:GetAttributes
              (vla-Item (vla-Get-Block (vla-Item (vla-Get-Layouts myDoc) "Layout1"))
                        1))))))
  (vla-Close myDoc)
  (setq i (1+ i)))
 (princ))
```

```
;; Warp speed
(defun C:Test3  (/ myDoc i fileName)
 (setq myDoc (vla-GetInterfaceObject
              (vlax-Get-Acad-Object)
              (strcat "ObjectDBX.AxDbDocument." (substr (getvar "AcadVer") 1 2))))
 (setq i 1)
 (repeat 100
  (setq fileName (strcat "C:\\Datasets\\CP319-1\\DBX Sample Drawings\\E2."
                         (itoa i)
                         ".dwg"))
  (vla-open myDoc fileName)
  (princ
   (strcat "\n"
           (vla-Get-TextString
            (car
             (sinc:GetAttributes
              (vla-Item (vla-Get-Block (vla-Item (vla-Get-Layouts myDoc) "Layout1"))
                        1))))))
  (setq i (1+ i)))
 (setq myDoc (i:CloseDBXDoc myDoc))
 (princ))
```

DbMod, or, Never Start AutoCAD with a Modified "New" Drawing

This condition is all too common. The moment a firm begins to customize the AutoCAD environment at startup is usually the moment the users start complaining about Document1. Changing the environment at startup is perfectly acceptable. However, if you make changes that cause a drawing to be immediately "dirty" when the user hasn't done anything, they will not tolerate the situation for long.

The sad part is this: in most cases you can avoid the issue with a few lines of code. A drawing is "dirty", flagged as modified, when the system variable DbMod is greater than 0. This system variable is read-only. However, for years there has been a couple of AutoLISP functions that can save the current state of the system variable and then restore that saved state back into the otherwise read-only system variable.

Place this statement at the top of the code that starts your customization: *(acad-push-dbmod)*.

This saves the current state. Note that if the drawing or environment is already “dirty” this will save the dirty state. In some cases, such as opening legacy AEC-based drawings the drawing database may be altered before your code begins to execute.

Place this statement near the end of your code: *(acad-pop-dbmod)*.

This restores the state of the system variable at the time you saved it. In most cases this means that blank drawings are no longer marked as modified before the user has done anything.

Conclusion

The topics covered in this handout only scratch the surface of what is possible with Visual LISP. The techniques of self-documenting code, modular functions, init-and-forget autoloading, storing data, opening AutoCAD drawings at high speed, and reducing false modifications all make code more efficient. Use this handout as a launch pad to developing (groan, sorry for the pun) good coding habits.

These techniques are not simply one person’s view on how to write code. Practices and techniques such as these are applied by professional programmers no matter what language is used. The book “Code Complete, 2nd Edition” by Steve McConnell (ISBN 0735619670) is recommended reading for anyone wishing to further develop these presented techniques.

Appendix

```
(defun i:FindApp  (fileName / res)
 (cond ((and (getenv "AcadCode")
             (setq res (findfile (strcat (getenv "AcadCode") "\\" fileName ".lsp"))))
        res)
       ((findfile (strcat fileName ".vlx")))
       ((findfile (strcat fileName ".fas")))
       ((findfile (strcat fileName ".lsp")))
       ((findfile fileName))))
```

```
 (defun i:AutoLoad  (data / fileName funcs warnUser fqnFilename)
 (mapcar 'set '(fileName funcs) data)
 (setq warnUser (strcat "Unable to load " fileName "."))
 (cond ((setq fqnFileName (i:FindApp fileName))
        (mapcar (function
                 (lambda (aFunc)
                  (eval
                   (cond ((= (type aFunc) 'LIST)
                          (list 'defun-q
                                (nth 0 aFunc)
                                (nth 1 aFunc)
                                ;; The below statement is needed to avoid
                                ;; endless loops where there is a bad
                                ;; function name but the filename exists.
                                (list 'setq (nth 0 aFunc) nil)
                                (list 'load fqnFileName warnUser)
                                ;; If the load is successful when the wrapper
                                ;; runs, the function name is once again bound.
                                ;; The first condtion will execute it.
                                ;; Otherwise, the variable is still nil and
                                ;; the alert is thrown.
                                (list 'cond
                                      (list (nth 0 aFunc) (cons (nth 0 aFunc) (nth 1 aFunc)))
                                      (list (list 'alert
                                                  (strcat "Cannot find function name "
                                                          (vl-symbol-name (nth 0 aFunc))
                                                          " in file "
                                                          fileName
                                                          "."))))))
                         (T
                          (list 'defun-q
                                aFunc
                                nil
                                ;; The code below has the same logic
                                ;; as the above code
                                (list 'setq aFunc nil)
                                (list 'load fqnFileName warnUser)
                                (list 'cond
                                      (list aFunc (list aFunc))
                                      (list (list 'alert
                                                  (strcat "Cannot find function name "
                                                          (vl-symbol-name aFunc)
                                                          " in file "
                                                          fileName
                                                          "."))))))))))
                funcs))
       ((/= (logand (getvar "CmdActive") 4) 4) (alert warnUser))
       ((princ (strcat "\n" warnUser)) nil)))
```

```
;;; Samples
(i:AutoLoad '("BogusFile" (i:GetInp)))
(i:AutoLoad
 '("Toolbox\\Get Input" ((i:GetInp (myPrompt myOptions)) i:AmBogus)))
```