AS226771

Programming the Work Out of CAD Management

Chris Lindner
Shremshock: Architects & Engineers

Learning Objectives

- Use AutoLISP to help with AutoCAD configurations.
- Explore how standards can enable automation via AutoLISP.
- Use AutoLISP can to check, apply, report, and maintain company standards.
- Learn how AutoLISP can also allow for exceptions while preserving company standards

Description

Some CAD Managers are challenged with balancing their “real work” (as a billable employee) with their CAD management responsibilities (often considered non-billable time). This can be a daunting and thankless assignment. Billable work and deadlines consume your workday, so the needs of CAD management get squeezed in between projects, during lunches, or after hours.

But it’s possible to leverage some simple (and a few not-so-simple) AutoLISP tricks to have AutoCAD do some of the heavy lifting or tedious tasks of CAD management for you so you can get back to your work (and back to your life). Like many tools, AutoLISP can be intimidating, frustrating, dangerous, and powerful all at the same time. But also, like a tool, understanding how it works and how to work it makes a huge difference. In this session, we’ll look at some real-world scenarios, a number of examples, a variety of resources available to take advantage of back at your company.

Speaker(s)

Chris is admittedly an “old school” AutoCAD user (having used it pretty much daily since graduating from Oklahoma State University in 1985) and a fledgling Revit user (some habits die hard!). He’s worked primarily in architecture, structural steel, and post-frame buildings, and has been a drafter, IT manager, trainer, programmer, and Autodesk consultant for notable companies such as Kroger, L Brands, and Huntington Bank and was a member of Autodesk’s AutoCAD Mentor team. He is currently serving his third term on the AUGI Board of Directors in the role of Treasurer. He is the CAD Manager for Shremshock: Architects & Engineers in central Ohio where he lives with Sonia, his wife of 30+ years. He’s the parent of two adult children and spends his free time reading, camping, gardening, doing art, and tinkering in the garage on the vintage 1966 International Scout he’s had since high school.

CAD Manager: Utility Player

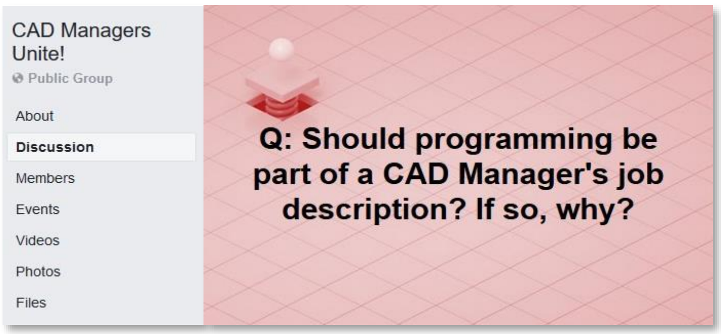
EVERYONE'S REFUSING TO WORK BECAUSE THE COFFEE MACHINE'S DEFECTIVE
CAD MANAGER
PENWILL

The premise of this class is this: most individuals who are responsible for their company's (or office's) CAD management duties wear multiple (sometimes unrelated) hats and may not have the luxury of being a full-time CAD Manager. They are expected to remain as billable as possible, while doing their part in insuring that everyone else is as billable as possible too! So, anything we can do to leverage technology to help with our CAD management responsibilities is a plus.

There is an advantage, though, to being in this situation. Who better to understand what needs improved, standardized, automated, and customized than someone who is living with the issues firsthand? It's easy for a full-time CAD Manager to be become distanced from the day-to-day needs. So, while it may feel (rightfully so) that you are pulled in multiple directions, this in-the-middle position does have its benefits.

Programming and CAD Management

A few months ago, I posed a question on the CAD Managers Unite! Facebook group page: "Should programming be part of a CAD Manager's job description? If so, why?"



Q: Should programming be part of a CAD Manager's job description? If so, why?

Interestingly, the responses aligned closely with the objectives of this class which had been submitted three months earlier. The responses, mostly in the affirmative, fell into the following broad categories:

- Configuration
- Automation
- Customization
- Standards

We'll discuss and look at examples of each category. Hopefully you'll be able to bridge the gap between the generic examples presented and your specific needs.

Programming the Work out of: Configuration

AutoCAD's open architecture is both a blessing and a curse. The same features that enable you to tailor AutoCAD to your company's unique needs are the same features that can be used by your users to "fix" something or personalize their environment. These well-intentioned "fixes" are sometimes to the exasperation of the CAD Manager. Attempting to support users in this "wild wild west" culture can be exhausting.

"WE MUST GET THE MOST FROM THE CAD TECHNOLOGY WE ALREADY HAVE"
PENWILL

Profile Flaws

The common approach for wrangling AutoCAD systems is by using profiles. Profiles are a great start to managing your CAD systems setup and maintenance. Over time, though, you are likely to run into some of the same limitations that I've found.

- Fragile – If a path in the Support File Search Path (SFSP) is not found, AutoCAD removes it assuming it will never be needed again. Autodesk's explanation is that by removing the missing paths, AutoCAD "*doesn't end up spending time searching for files in places that do not exist.*". Unfortunately, this doesn't account for the occasional network hiccup or remote users in which case a path may be temporarily unavailable but shouldn't be removed.
- Local – Once a profile is loaded, all its specific settings are stored locally in the user's registry. These settings will remain static until the profile is reloaded or the user makes changes. Autodesk even states "*Profile information is neither automatically saved nor updated.*" (link) Trying to manually manage settings that are stored locally on multiple systems is no fun.
- Inflexible – Sometimes it's necessary to "reset" AutoCAD by reloading a profile and restoring it to an earlier known, working configuration. When a profile is reloaded, though, any personalized tweaks that a user may have made (background color, cursor size, etc.) are overwritten.
- Vulnerable – There's no way to secure certain areas of the profile that you don't want users from changing. A variable changed by an uneducated user could significantly impact production.

Through programming, it is possible to overcome these shortcomings. AutoLISP can help us achieve that sweet spot between control and freedom.

Let's dive in!

Key Files

Before we start coding, we need to establish where to put our code. The best mechanism for doing this is by utilizing two files: ACAD.LSP or ACADDOC.LSP. These user-defined files will load automatically making them the perfect way to provide on-the-fly automated CAD Management.

- The ACAD.LSP is typically loaded only when AutoCAD launches and is primarily used for application-specific routines.
- The ACADDOC.LSP loads as each drawing is opened and is suited for drawing-specific routines.

(*Alternatively, you can bypass the ACADDOC.LSP and just use the ACAD.LSP if the ACADLSPASDOC variable is set to 1. For this class, we'll be using both to demonstrate how they will work together.*)

Additionally, each CUI file will automatically load an accompanying MNL (menu lisp) file if found. This file will be loaded after the ACADDOC.LSP file.

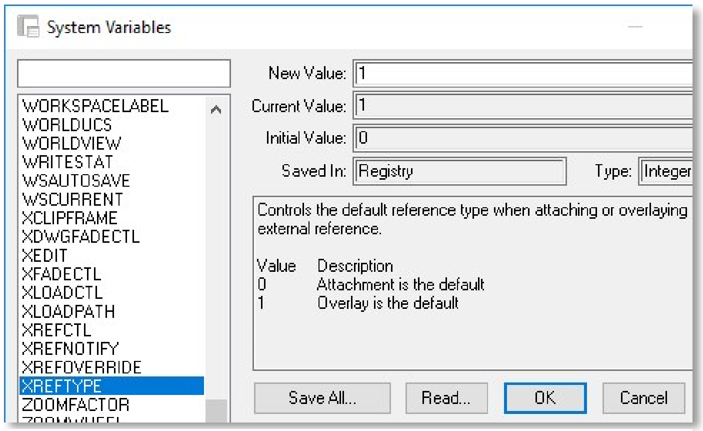
Starting with Startup

The code that's placed in your ACAD.LSP or ACADDOC.LSP is loaded into memory before the drawing has finished opening. Some of your code, however, you may not want to actually "run" until after the drawing has finished opening (i.e. is "initialized"), especially functions that contain (Command) calls. AutoCAD uses a special function, (S::Startup), to do this. The technical term for it is "post-initialization execution".

Note: Since the (S::Startup) function can be defined in each of the key files mentioned above, the (defun-q) function should be used to define them independently. They can be appended together to execute in the desired order.

System Variables

Most of the configuration settings within an AutoCAD profile can be managed via AutoLISP thus addressing some of the profile limitations listed above. Many of these settings are stored in what are called "system variables". These variables store the settings assigned in the Options dialog box or other configuration dialog boxes. And there's no shortage of system variables in AutoCAD. In fact, if you want to explore the 900+ (*literally!*) system variables, the Express Tools SYSVDLG command is a great "variable browser" of sorts.



New Value: 1
Current Value: 1
Initial Value: 0
Saved In: Registry
Type: Integer

Controls the default reference type when attaching or overlaying external reference.

```
Value   Description
0       Attachment is the default
1       Overlay is the default
```

Figure 1 The SYSVDLG dialog box

Working with system variables in your ACAD.LSP and ACADDOC.LSP is done with two functions:

Function: (getvar) used for accessing current values
Syntax: `(getvar <variable name>)`

Function: (setvar) for assigning new values.
Syntax: `(setvar <variable name> <value>)`

Below is an example of some system variables that you might place in your ACAD.LSP. These values are set (or reset) each time AutoCAD launches and will apply to that entire AutoCAD session.

```
(setvar "FILEDIA" 0)                   ; display dialog boxes
(setvar "CONSTRAINTINFER" 0)           ; turn off inferred constraints
(setvar "OBJECTISOLATIONMODE" 0)       ; turn isolated objects back on
(setvar "XREFTYPE" 1)                  ; default to "overlay" xrefs
```

FIGURE 2 SYSTEM VARIABLES IN A SAMPLE ACAD.LSP

Below is an example of some system variables you might place in your ACADDOC.LSP. These values are drawing-specific and will be set (or reset) for each drawing as it's opened.

```
(setvar "CECOLOR" "Bylayer")           ; insure current color is ByLayer
(setvar "CELTYPE" "Bylayer")           ; insure current linetype is ByLayer
(setvar "HPLAYER" ".")                 ; force hatch to use current layer
(setvar "LUNITS" 4)                    ; default to Arch units
```

FIGURE 3 SYSTEM VARIABLES IN A SAMPLE ACADDOC.LSP

Note: Some system variables are read-only. You can use the (getvar) function to access the current value, but (setvar) will return an error.

Environment Variables

You'll soon run into some settings that can't be assigned by the (setvar) function. These are most likely environment variables, not system variables. AutoCAD uses environment variables to store operating system-related environment settings and can be done with two functions:

Function: (getenv) for accessing current values,
Syntax: `(getenv <variable name>)`

Function: (setenv) for assigning new values.
Syntax: `(setenv <variable name> <value>)`

Here are a few environment variables settings our company pushes via the ACAD.LSP:

- hide our system printers and use only PC3 files
- save drawings in the 2013 file format because some of the consultants we work with have not upgraded yet
- always display the Model and Layout tabs
- always show drawing's full path in the title bar

```
(setenv "HideSystemPrinters" "1")      ; hide system printers
(setenv "DefaultFormatForSave" "60"); force saving to 2013 file format
(setenv "ShowTabs" "1")                ; show layout tabs
(setenv "ShowFullPathInTitle" "1")    ; show full path in title bar
```

FIGURE 4 ENVIRONMENT VARIABLES SAMPLE

A few notable differences between system and environment variables:

- Environment variables are case sensitive, so "HideSystemPrinters" is not the same as "hidesystemprinters".
  System variables are not as particular.

- Environment variables accept strings values only. If you need to save a numeric value, it will have to be converted to a string via either (itoa) or (itof). Conversely, string values returned by (getenv) that should be numeric will need to be converted via (atoi) or (atof).
  System variables can accept integers, reals, strings, and 2D or 3D points.

- Environment variables can't be typed in on the command line.
  System variables can be typed like a normal command

- Environment variables are generally placed in the ACAD.LSP.
  System variables can be placed in either. A good rule of thumb is this: if it's a setting that saved in the drawing, place it in the ACADDOC.LSP

- Environment variables are stored in the registry.
  System variables are saved in one of four places:

Saved in	Description	Best file
Registry	These variable values are stored in the registry and are restored when AutoCAD is opened.	ACAD.LSP
Drawing	These variables are saved in each separate drawing and restored when that drawing is opened.	ACADDOC.LSP
User-settings	Legacy setting; handled the same as variables saved in the Registry.	ACAD.LSP
Not-saved	These values are is lost each time the drawing is closed and are restored to their default setting when the drawing is reopened. They can still be added to the ACADDOC.LSP to preset the desired default value.	ACADDOC.LSP

You may also find that some setting can be assigned by both (setvar) and (setenv), although the variable names may vary. Here's an example:
`(setvar "XLoadPath" "C:\\Temp")` or `(setenv "XrefLoadPath" "C:\\Temp")`

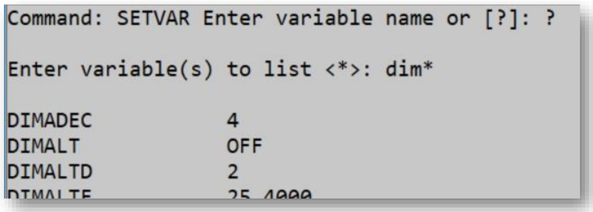
Additionally, some system variables that are read-only are not as environment variables. Here's an example:

`(setvar "TempPrefix" "C:\\Temp")` will return an error, but
`(setenv "TempDirectory" "C:\\Temp")` works just fine.

Which Variable to Use

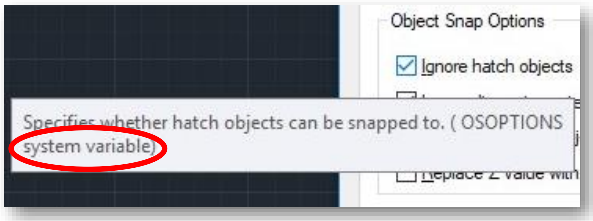
So, is the setting you're in search of a system variable or an environment variable? Here's a few tips on how to find out:

- The SYSVDLG box mentioned above it worth browsing through. Also, browse through the SETVAR command. Use wildcards to narrow the search. See if any variables look promising. (Unfortunately, there's no corresponding SETENV command)



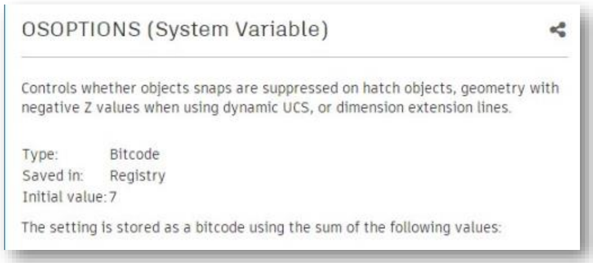
```
Command: SETVAR Enter variable name or [?]: ?

Enter variable(s) to list <*>: dim*

DIMADEC             4
DIMALT              OFF
DIMALTD             2
```

- Some settings in the Options dialog will provide you a hint via their tool tip. The "Ignore hatch objects" tool tip references the "OSOPTIONS *system variable*". Alternately, pressing F1 will display a tab-specific help page that may contain the details you're looking for.



Specifies whether hatch objects can be snapped to. (OSOPTIONS system variable)

- Searching AutoCAD's help for the variable will provide more insight.



OSOPTIONS (System Variable)

Controls whether objects snaps are suppressed on hatch objects, geometry with negative Z values when using dynamic UCS, or dimension extension lines.

Type: Bitcode
Saved in: Registry
Initial value: 7

The setting is stored as a bitcode using the sum of the following values:

- For settings without an obvious system variable, like the "AutoSnap Marker Size", a web search will usually provide the needed information. Including the word "AutoLISP" in your search will generally give you better search results.

Programming Detour: Beyond Vanilla AutoLISP

Before we go on, we're going to take a programming detour and introduce some extended AutoLISP functions that will open up more possibilities. I'll be honest, this stuff is all black magic to me. I know it works, and I'm amazed at what people do with it, but I can't explain it in all the correct technical terms. So, I will defer to more experienced authors to do this

According to afralisp.net, "*Visual LISP is an extension of (not a replacement for) the AutoLISP programming language. It is more powerful than AutoLISP because it can access the AutoCAD object model... It forms the perfect 'next step' for AutoLISP users who want a little more power over AutoCAD in their routines.*"

"VISUAL LISP FORMS THE PERFECT 'NEXT STEP' FOR AUTOLISP USERS WHO WANT A LITTLE MORE POWER OVER AUTOCAD"

Visual LISP functions allow you to get more "under the hood" than is possible with vanilla AutoLISP alone. Visual LISP functions enable VBA-type interactions with the program, the drawing, and the entities in them. In this class, we'll only scratch the surface of what Visual LISP is capable of. In my code examples, you will see a blend of both AutoLISP and Visual LISP. Whether that's good or bad is debatable, but my research says that each has its limitations.

Note: Some Visual LISP extensions are limited to Windows only.

To unlock the world of Visual LISP, your ACAD.LSP needs to have this statement, preferably near the beginning:

```
(vl-load-com)
```

This simply loads the supporting Visual LISP code, making the extended functions available.

Next, we need to connect to the AutoCAD application object. We'll store it in a global variable because we'll refer to it multiple times. I am assigning this to a variable named *acad*. (*I typically use leading and trailing asterisks for my global variable names; the variable name you use is your choice.*)

```
(setq *acad* (vlax-get-acad-object))
```

Next, we'll store a reference to a couple other objects, the Preferences and Files objects. These two are stored as global variables for future code to use.

```
(setq *prefs* (vla-get-preferences *acad*)
      *files* (vla-get-files *prefs*))
```

You might think of the Preferences object as the Visual LISP equivalent of the Options dialog box, and the Files object the equivalent of the Files tab.

Automating Search Paths

Automating AutoCAD's search paths is a great way to bring consistency to your AutoCAD environment. As mentioned earlier, if one of these paths is not found when AutoCAD launches, it gets removed. By programming, you can make your search paths "self-healing"!

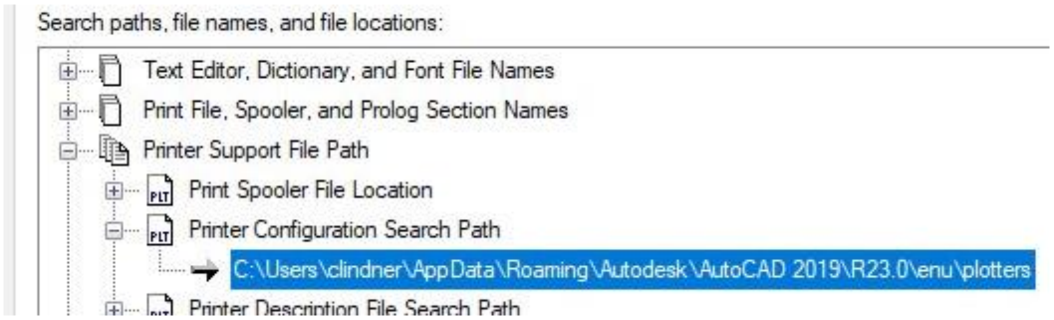
Using the *files* variable above and the Visual LISP function (vla-get-PrinterConfigPath), you can access the "Printer Configuration Search Path" that contains your PC3 files:

```
(vla-get-PrinterConfigPath *files*)
```

This returns…

```
"C:\\Users\\clindner\\AppData\\Roaming\\Autodesk\\AutoCAD 2019\\R23.0\\enu\\plotters"
```

… the same path shown in my Options dialog box:



Search paths, file names, and file locations:

- Text Editor, Dictionary, and Font File Names
- Print File, Spooler, and Prolog Section Names
- Printer Support File Path
  - Print Spooler File Location
  - Printer Configuration Search Path
    - C:\Users\clindner\AppData\Roaming\Autodesk\AutoCAD 2019\R23.0\enu\plotters
  - Printer Description File Search Path

Some paths simply contain a single path string, while others contain multiple path strings. The "Automatic File Save Location" setting, for example, can only point to a single folder. To assign this path in your ACAD.LSP, you simply provide a new path string, as shown below:

```
(vla-put-AutoSavePath *files* "C:\\Temp")
```

Figure 5 Single Path Assignment

Other path variables can contain more than one folder. The SFSP, for example, contains multiple paths. Maintaining this list of paths is important to a stable environment. To make self-healing SFSP, the following function (SFSPFix) could be placed in your ACAD.LSP:

```
(defun SFSPFix ()
  ;; Set standard Support File Search Paths
  (vla-put-SupportPath
    *files*
    (strcat
      "E:\\_AU2018 Support;"
      "C:\\ACAD\\My ACAD 2019\\Lisp;"
      "C:\\Users\\clindner\\AppData\\Roaming\\Autodesk\\AutoCAD 2019\\R23.0\\enu\\Support;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\Support;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\Support\\en-US;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\Fonts;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\help;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\express;"
      "C:\\Program Files\\Autodesk\\AutoCAD 2019\\Support\\Color;"
     ) ;_ end of strcat
  ) ;_ end of vla-put-SupportPath
  ;; Set trusted paths
  (setvar "trustedpaths"
          (strcat
            "E:\\_AU2018 Support;"
            "C:\\ACAD\\My ACAD 2019\\Lisp;"
          ) ;_ end of strcat
  ) ;_ end of setvar
) ;_ end of defun
```

Figure 6 Self-healing Support File Search Path

For Further Exploration

Here are a few other possible options for you to try your hand at or to simply spark some ideas of your own:

- Create a dynamic SFSP that can add, remove, swap, and/or position paths to better manage load order.
- Explore the registry read & write functions, (vl-registry-read) and (vl-registry-write), as an alternate for accessing and controlling configuration settings.
- Explore what the “AutoCAD Object model” to see what else Visual LISP gives you access too.

Programming the Work out of: Standards

Ah, standards; the things we love to hate. But the only thing worse than having standards, is not having them! And even with the inconveniences that they impose, we certainly appreciate drawings that have them.

Once standards are established, next comes the task of implementing, enforcing, and monitoring them. Ideally, they would just be seen for their inherent value and eagerly adopted. Realistically, though, they can encounter the roadblocks of old habits or just plain resistance. My philosophy has always been this if you can make it easier for users to do something right (i.e. per the standard) than it is to do it wrong, then most people will typically do it right and take the path of least resistance! It's a true win-win.

IF YOU CAN MAKE IT EASIER FOR THEM TO DO IT RIGHT THAN IT IS TO DO IT WRONG, THEY'LL USUALLY DO IT RIGHT.

The Document Object

Since many of the standards we deal are in the drawings themselves, there's another Visual LISP object we need to reference, the "Document" object. I will assign this object to the global *doc* variable by placing the code below in my ACADDOC.LSP.

```
(setq *doc* (vla-get-ActiveDocument *acad*))
```

I generally assign this variable at the same time that I assign the three other global variables defined earlier, *acad*, *prefs*, and *files*.

Pushing Standards

The strong-arm approach to implementing standards may not make help with your CAD Manager likability ratings but it (usually) ensures compliance and adoption. By pushing those non-negotiable and universal standards, it helps create a stable, dependable, and predictable environment which maximizes production, reduces surprises, and minimizes support. Also, by providing a consistent configured environment, users are less likely to feel they need to go "fixing" things on their own…which is never a good thing.

Here's a "push" example used in our office. In our architectural drawings, we don't want users accidently erasing or moving a structural column or column grid line. So, as our drawings are opened, we force these layers to be locked via the following function, (LockColLayers), in our ACADDOC.LSP:

```
(defun LockColLayers (/ layer)
  (foreach layer '("S-COLM" "S-COLM-IDEN" "S-COLM-GRID" "S-COLM-DIMS")
    (if (tblobjname "layer" layer)
      (vla-put-lock
        (vla-item (vla-get-layers *doc*) layer)
        :vlax-true
      ) ;_ end of vla-put-lock
    ) ;_ end of if
  ) ;_ end of foreach
) ;_ end of defun
```

Figure 7 Auto-locked Layers

We also want all our paper space viewports on a designated non-plotting layer named "0-VPRT" and want them all "display locked". This function, (LockVPorts), is also defined in our ACADDOC.LSP:

```
(defun LockVPorts (/ lay ent)
  ;; insure that 0-VPRT layer exists and is non-plot
  (setq lay (vla-Add (vla-get-Layers *doc*) "0-VPRT"))
  (vla-put-plottable lay :vlax-false)
  ;; loop thru all layouts
  (vlax-for lay (vla-get-Layouts *doc*)
    (if (eq :vlax-false (vla-get-ModelType lay))         ; skip model space layout
      (vlax-for ent (vla-get-Block lay)                  ; for each ent in layout
       (if (eq (vla-get-ObjectName ent) "AcDbViewport") ; if entity is a viewport
         (progn
           (vla-put-DisplayLocked ent :vlax-true) ; lock the viewport
           (vla-put-Layer ent "0-VPRT")           ; assign viewport to "0-VPRT"
         ) ;_ end of progn
       ) ;_ end of if
      ) ;_ end of vlax-for
    ) ;_ end of if
  ) ;_ end of vlax-for
) ;_ end of defun
```

Figure 8 Auto-locked Viewports

Both of the above functions are executed by the (S::Startup) after the drawing initializes.

```
(if (= 1 (getvar "DWGTITLED"))    ; skip this on unnamed drawings
  (progn
    (princ "\nLocking viewports...")
    (LockVPorts)
  )
) ;_ end of if
```

Figure 9 Locking viewports via (S::Startup)

Pulling Standards

It's possible to convert some of the "push" standards that are in our (S::Startup) into a "pull" standard so the user can apply them when needed. For example, rather than force all the viewports to be locked when each drawing is opened, the (LockColLayers) function could be turned into a new LockColLayers command with the following code. Similarly with (LockVPorts):

```
(defun C:LockColumns ()
  (LockColLayers)
  (princ)
  )
```

Figure 10 "Pull" method for locked layers

```
(defun C:LockVPorts ()
  (LockVPorts)
  (princ)
  )
```

Figure 11 "Pull" method for locked viewports

Standards By Default

Another less pushy approach is to offer a relevant (standards-compliant) default. Using the "make it easier to do right" philosophy mentioned earlier, this can make your users' lives easier and improve adherence to company standards, but still allows for user overrides.

Let's say that your standard dimension style is named "Arch_Tick-Anno". As your drawings are opened, you could set it as the default dimstyle via the ACADDOC.LSP as shown below. The user can choose a different dimstyle at any time, but it's initially set to the standard style.

```
(vla-put-activedimstyle
  *doc*
  (vla-item (vla-get-dimstyles
              *doc*
            ) ;_ end of vla-get-dimstyles
            "Arch_Tick-Anno"
  ) ;_ end of vla-item
) ;_ end of vla-put-activedimstyle
```

Figure 12 Suggested Defaults

For Further Exploration

Here are a few other possible automated standards for you to try your hand at or to simply spark some ideas of your own:

- Assign all xrefs to a designated layer, to have a saved path, and/or be a certain type (attached or overlaid).

- Turn annotation visibility (ANNOALLVISIBLE) off so that the only annotation you see is the annotation that's assigned the current annotation scale.
- Make sure that all FRAMES (xclip, image, pdf, etc.) are visible but won't plot.
- Adapt the (LockColLayers) function to work with wildcards, instead of a list of specific layer names.

Adaptive Standards

One thing about standards, there always seems to be variations or exceptions that mess things up. You've heard it before, "It always works like this…well, except when _____..."! Legitimate exceptions could be needed for certain projects, clients, software, etc. Programs that are too rigid can hinder productivity and quickly become a source of frustration for your users. Creating code that can adapt to different scenarios will empower you to use minimize the amount of redundant code.

The code in Fig 11 above is an example of non-adaptive code. What happens when you open a drawing which doesn't have the "Arch_Tick_Anno" dimstyle? The modified code below is adaptable. By adding a simple line of code, it will now "exempt" drawings that don't have that dimstyle defined:

```
(if (tblsearch "DIMSTYLE" "Arch_Tick_Anno")
  (vla-put-activedimstyle
    *doc*
    (vla-item (vla-get-dimstyles
                *doc*
              ) ;_ end of vla-get-dimstyles
              " Arch_Tick_Anno "
    ) ;_ end of vla-item
  ) ;_ end of vla-put-activedimstyle
) ;_ end of if
```

Figure 13 Adaptive Code

Ironically, it's consistent, dependable standards that actually make exceptions possible! It allows you to incorporate conditionals in your code that are based on predictable conditions.

THE FOUNDATION FOR AUTOMATION IS STANDARDIZATION.

Folder and file naming standards, for example, can play an important role in making it possible to identify things like discipline, project, client, etc.

Let's look at how we could leverage this simple file name standard to determine the drawing's discipline:

- Standard format: <project no>_<disc><sht #>.dwg
- File name example: 201805_A0201.dwg

Based on this file naming standard, you could determine what the sheet's discipline is by the following code:

```
(cond ((eq "A" (substr (getvar "DWGNAME") 8 1))
       (setq *dwgdisc* "Arch")
      )
      ((eq "E" (substr (getvar "DWGNAME") 8 1))
       (setq *dwgdisc* "Elec")
      )
      ((eq "M" (substr (getvar "DWGNAME") 8 1))
       (setq *dwgdisc* "Mech")
      )
) ;_ end of cond
```

Figure 14 Filename-based Disciplines

This *dwgdisc* global variable could then be referenced by other code so that it can to adapt to discipline-specific conditions, such as the tool palette scenario mentioned earlier:

```
(setvar "*_TOOLPALETTEPATH"
        (cond ((eq "Mech" *dwgdisc*) "N:\\Palettes\\Mech\\")
              ((eq "Elec" *dwgdisc*) "N:\\Palettes\\Elec\\")
              (t
               "N:\\Palettes\\Arch\\"
              )
        ) ;_ end of cond
) ;_ end of setvar
```

Figure 15 Leveraging Adaptive Code

Adapting To User Preferences

Nothing can jeopardize your efforts to implement standards more than having things so locked down that users can't do any of their own customization and personalization. Some CAD Managers yield to the backlash from users (and possibly upper management) and just surrender. Just as we've shown some possibilities of creating code that adapts to client/project/drawing exceptions, it's also possible to adapt to user exceptions while maintaining a degree of control.

It's not unrealistic to have users who can write AutoLISP code of their own. The following code at the end of the (S::Startup) could be used to load a user's local AutoLISP file.

```
(if (setq usr (findfile "C:\\ACAD\\My ACAD 2019\\Personal.lsp"))
  (progn
    (princ (strcat "\nLoading personal code for \""
                   (getvar "LOGINNAME")
                   "\"..."
           ) ;_ end of strcat
    ) ;_ end of princ
    (load usr)
  ) ;_ end of progn
) ;_ end of if
```

Figure 16 Loading User Code

The following code ensures that the Autosave time is turned on and is no greater than 30 mins. If a user sets their save time to a shorter duration (for more frequent saves), I don't change it. But if they turn it off or set it to a longer time (less frequent saves), it gets set back to a 30 minute maximum when they re-open AutoCAD (via the ACAD.LSP).

```
;; force maximum allowable savetime
(if (or (= (getvar "SaveTime") 0) ; if autosave is off, or
       (> (getvar "SaveTime") 30) ; if savetime > 30 mins
       )
  (progn
    (setvar "SaveTime" 30) ; force savetime to 30 mins max
    (princ "\nAutoSave adjusted to 30 minutes maximum. ")
    )
  )
```

Figure 17 Flexibility within Constraints

For Further Exploration

Here are some other adaptive programming possibilities for you to try your hand at or to simply spark some ideas of your own:

- One of your consultant's is using AutoCAD 2012 so when you work on their drawings, they need saved in 2010 format, but all other drawings should be saved in 2018 format.
- You want your 2D drawing production users to "Ignore Z object snaps" but need to allow it for your 3D design users.

Programming the work out of: Customization

The opportunities for customizing AutoCAD are limited only by your imagination, and the more you delve into the programming world, the more opportunities you'll see. Also, the more you listen to your users, the more opportunities you'll see as well. Further, the more custom solutions you develop for your users, the more opportunities *they* will see too.

"SURELY THERE'S MORE TO CUSTOMISING THE SOFTWARE THAN JUST DECORATING THE BOX"

Creating New Commands

AutoLISP's (defun C:) function can be used to create commands that emulate an AutoCAD command. Many AutoLISP newbies take their first crack at creating their own commands by modifying existing AutoCAD's commands. The 2-letter commands below, shortcuts for the XLINE command, are simple examples of a simple AutoLISP command that couldn't be defined in the PGP file.

```
(defun C:XH () (command ".XLINE" "H") (princ))
(defun C:XV () (command ".XLINE" "V") (princ))
```

Figure 18 Simple Custom Commands

By incorporating some additional AutoLISP magic, you can further enhance on existing

```
;; Shortcut to send objects to the back
(defun C:BF (/ SS)
  (if (not (setq SS (ssget "_i")))
    (progn
      (princ "\nSelect object(s) to bring to front: ")
      (setq SS (ssget))
      (if SS
        (command "DRAWORDER" SS "" "F")
        (command)
      ) ;_ end of if
    ) ;_ end of progn
    (command "DRAWORDER" "F")
  ) ;_ end of if
  (princ)
) ;_ end of defun
```

Figure 19 Enhancing an AutoCAD Command

AutoCAD commands. For example, the following code defines the BF command, a 2-letter shortcut that is a specialized version of the DRAWORDER command:

Hijacking Existing Commands

AutoCAD's UNDEFINE command can be used to take an existing command out of commission completely, making room for you to create your own replacement command. Savvy users, however, can find out how to bypass this, executing the actual command. There's another option that's easier, more powerful, and more bulletproof.

Rather than completely replacing an AutoCAD command, it's possible to "hijack" AutoCAD so that when certain conditions occur, your code is triggered. The feature I'm referring to is called "reactors". They are super cool, very powerful, and fairly easy to create. They could easily be an entire AU class of their own.

The (MyCommandWillStart) function below is designed to be triggered when any AutoCAD command is started. When called, it looks at the name of the command that has just started. If it's a dimension command (i.e. it starts with "DIM"), it sets the "Dims" layer current (creating it if missing) then returns control to the called dimension command.

```
(defun MyCommandWillStart (calling-reactor cmdinfo-list / lay)
  (cond
    ((eq (substr (nth 0 cmdinfo-list) 1 3) "DIM")
     (setq ol (getvar "CLAYER")) ;save original current layer
     (setq lay (vla-add (vla-get-layers *doc*) "Dims"))
     (vla-put-color lay 3)
     (vla-put-activelayer *doc* lay)
    )
  ) ;_ end of cond
) ;_ end of defun
```

Figure 20 Command Started Reactor

This function could be expanded to react to other AutoCAD commands as well.

An accompanying (MyCommandEnded) function is triggered after each AutoCAD command is

```
(defun MyCommandEnded (calling-reactor cmdinfo-list /)
  (cond
    ((eq (substr (nth 0 cmdinfo-list) 1 3) "DIM")
     (vla-put-activelayer
       *doc*
       (vla-item (vla-get-layers *doc*) ol)
     ) ;_ end of vla-put-activelayer
    )
  ) ;_ end of cond
) ;_ end of defun
```

Figure 21 Command Ended Reactor

completed. Like (MyCommandWillStart), this one specifically looks to see if the command that just ended was a dimension command. If so, ut returns the drawing to the previous current layer (that was saved in the "ol" variable).

These two functions define what will happen when the reactor fires. Now, we need to associate them to a specific AutoCAD event. In this case, the two events we care about are the built-in (:vlr-commandwillstart) and (:vlr-commandends) events. These events, when triggered, will call my functions (MyCommandWillStart) and (MyCommandEnded) respectively.

```
(vlr-editor-reactor nil '((:vlr-commandwillstart . MyCommandWillStart)
                          (:vlr-commandended     . MyCommandEnded))
```

Figure 22 Assigning Reactor Events

Important note: Reactors functions should not contain calls to other AutoCAD commands.

For Further Exploration for Customization

Here are a few possible reactors to try your hand at or to simply spark some ideas of your own:

- After pasting an anonymous block via PasteAsBlock, prompt the user for a block name.
- Prevent certain blocks (list of blk names) from being exploded
- Lock VPs when switching layouts (not during startup)
- Explore the other types of reactors: Database, Document, and Object

Conclusion

The potential for programming the work out of CAD Management is tremendous. Hopefully you've seen some of the possibilities.

It will be a process. Embrace it. Fail fast. Fail forward.

Additional links and resources available at: tinyurl.com/ProgrammingCM

Feel free to reach out to me with questions, success, learnings, feedback, etc.!

Email: chris.lindner@gmail.com

Please take a moment to complete the class survey!

Thanks for Attending!