AUTODESK UNIVERSITY

IT21765

Perfect Plots – Every Time, Every User!

Frank Mayfield
Benham, a Haskell Company

Learning Objectives

- Understand the need for setting and adhering to standard naming conventions
- Learn how to maintain a single repository for company or project-specific page setups, ensuring quality and standardization
- Learn the workflow and application of Autolisp coding to take back and apply at your office
- Learn how to demonstrate tangible results and time savings to management by applying these techniques

Description

Managing printing, plotting, and publishing in a multi-project and multidiscipline environment can be a daunting challenge to a CAD manager. You may go to a lot of trouble at the beginning of a project to make sure your page setups are perfect, but your users change things (yes, even if they are not supposed to). Wouldn't it be great to ensure your setups are always correct every time someone opens a project file? In this class, you will learn an innovative method for doing just that. All you need is a little Autolisp knowledge (available in the class), some good naming standards, and a little work up front. Newly armed with these techniques, you will no longer have to watch your users click incessantly on plot dialog controls as you make your way through your department.

Your AU Expert

Frank is currently the CAD Manager for the Oil, Gas and Chemical division at Benham in Tulsa, Oklahoma. In this role, he oversees all aspects of the AutoCAD, CADWorx and Plant 3D environments.

He has been using AutoCAD since 1986 (v 2.61). Frank is an Autodesk AutoCAD Certified Professional, a volunteer with AUGI as a forum manager, and is a member of the AutoCAD Customer Council, Autodesk User Panel, and the Autodesk Advisors Community.

He was part of a select group invited by Autodesk to attend the inaugural CAD Manager Summit in 2015. As a top-rated member of the AutoCAD All-Star Mentor program, he has provided live guidance and answered questions for over 2,100 in-trial users in more than 50 countries worldwide.

Table of Contents

1. Introduction

This course is intended for CAD Managers, or those responsible for their duties, who are tasked with supporting projects for multiple clients, and/or multiple disciplines where the page setups may differ for each.

While everyone's situation is different, I'll present the basic concepts of my solution, and explore some common scenarios that may be closer to your company's setup.

Section 4 of this handout will present four different case studies, ranging from the simplest to quite complex, along with sample code for each. All code will be posted to the AU class site.

Page Setups: The Good, the Bad, and the Ugly

Named Page Setups were first introduced in AutoCAD 2000, and were updated to their present form in AutoCAD 2005 as a companion to Sheet Set Manager. Collecting the myriad of controls needed to produce a standardized print or plot was a vast improvement over the old system. In fact, most of you reading this probably don't remember the old system, so it's best just left in the past.

There is an inherit problem with the current implementation of page setups however: They are saved in the drawing file, and there is no control to keep a user from changing the settings. In a perfect world, there would be a concept of project settings in AutoCAD, but that's a topic for another time.

Meanwhile, at the beginning of a project, we CAD Managers make sure the necessary Page Setups are correct in either our templates, our Sheet Set Manager override file, or both.

Let's look at a simple example of what can happen. In this case, let's assume you have a single Page Setup named *HPLaserJetPro*. It lives in your project template, so each sheet created will contain that setup. At some point, a user changes something and resaves the setup. Now it doesn't match the project's master setup. To make things worse, that sheet is now copied and renamed to create many new, similar sheets. Now they're all infected with the wrong setup.

So you can see, the problem lies in that the same named setup *HPLaserJetPro* in Drawing A can be different than that in Drawing B, and each of those can be different than the project standard setup *HPLaserJetPro* in your template.

```
_$ (eq (getPS "DWG_A" "HPLaserJetPro") (getPS "DWG_B" "HPLaserJetPro"))
   nil
```

THIS IS A FAKE CODE SNIPPET – JUST TO MAKE A POINT !

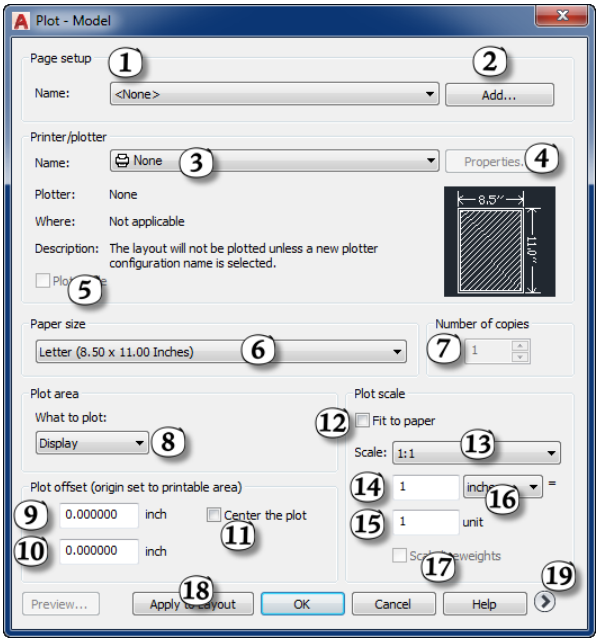
What's a CAD Manager to do?

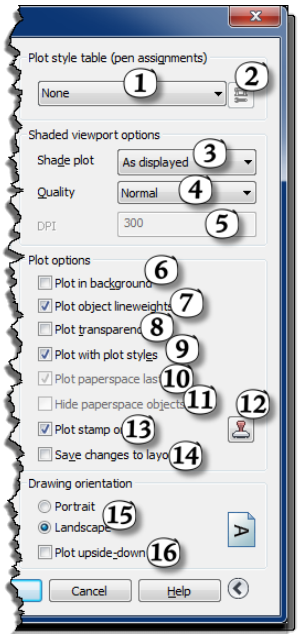
When I took over the CAD Manager's role at my company, there was a similar chaos in place. I found that we were a Sheet Set Manager shop, mainly for the reason to control plotting. Users had been instructed to only print or plot via the override file specified for each project in SSM, as the setups in the files couldn't be trusted.

Work was busy at the time, new users were cycling in and out, and naturally they expected to be able to use the Plot command. After making my rounds through the department, and watching them seemingly click every control possible in the Plot dialog to get decent output, it didn't take long for me to realize I needed to take action.

I realized there were a number of ways to accomplish this, but most require some kind of user input – macros to import the proper page setups, a tool somewhere to click on, a custom Plot routine etc. I'm in the camp that believes the user shouldn't have to do anything, nor should they even be aware it's happening. It just works. As noted CAD Management expert Robert Green states: "…the coolest standard is the one they don't even realize they're using"1. Having been an Autolisp programmer for years, and already a proponent of utilizing the ACADDOC.lsp file to set my user's environments upon entering a drawing, I gravitated toward that method.

> *"…the coolest standard is the one they don't even realize they're using"*
>
> – Robert Green





THERE ARE 35 CONTROLS IN THE PLOT DIALOG !

2. Work Flow Concept

Armed with the knowledge that we want this to work automatically, and every time a user opens a project file, we can construct a work flow to accomplish this.

First, to our code has to fire up whenever a file is opened. There's no better tool for that than the ACADDOC.lsp file. If you're already using it to set your user's environment, good. If not, more info will follow, and some links regarding it will be in the Addendum to this handout.

Next, we need to know something about the file being opened. This is where good folder and file naming standards are imperative. Using Autolisp, we'll search the file path looking for keys that will tell you where it is, and what it is.

Once we find out what we're dealing with, we'll have to know where the proper page setup file is for that project. You may have to build the path yourself, or based on what you now know about the file, you might already know where it is.

Now that you know what kind of file is being opened, and where to find its master page setup file, the real work begins. Every scenario I'll present ends this way. Simply delete all the existing page setups, and then import all from the master page setup file.

The whole process takes milliseconds of lisp processing time, and the user never knows it happened (that is, unless you prompt them).

Let 's take a look at this work flow in more detail:

ACADDOC.lsp

AutoCAD will always look for two files on start-up; ACAD.lsp and ACADDOC.lsp. If found, by default ACAD.lsp runs once when the program is started, and ACADDOC.lsp runs every time a drawing file is opened. Neither file exists after installation, so it is up to you to create them.

You'll need to make sure the file is saved to a location where AutoCAD can find it. It can be in any of the standard support folders AutoCAD uses, or, as I prefer, a custom location on your network. This method allows you, the CAD Manager to be the owner of the file (you can lock it down with permissions), and provides a one-stop shop for initiating any changes. If you're using a network license of AutoCAD, this custom path can be added to your deployment image. For more information on customizing deployments, I suggest the following:

Robert Bell's "Bootstrap AutoCAD Deployments for Customizations" Series
http://blogs.autodesk.com/autocad/bootstrap-autocad-deployments-for-customizations-series/

R.K. McSwain's "Deployments and AutoLISP Strategies for Easy Installations and Maintenance" http://au.autodesk.com/au-online/classes-on-demand/class-catalog/2015/autocad/it9952#chapter=0

You can use a simple text editor like NOTEPAD or the more advanced VLIDE within AutoCAD to create the files. You can utilize either or both, it's up to you.

For our problem, we want the page setup routines to run every time a user opens a file, so we'll be using the ACADDOC.lsp file.

Analyze the file being opened

Of course, the user could be opening any file from any location. It's our job to figure out what it is, where it is, and if it is a valid project file. Your local file structure may look similar to one of the following examples.

P:\FDB\1650-TUL\IND\ClientName\316363 AU_CLASS\20_DESGN\40_CAD\PIPING
L:\Industrial\Projects\33841 Large Mart\201 Design\20 AutoCAD\Arch
B:\2016\Retail\Buy More\Burbank\91501\Eng\CAD\MEP

And, you may have file names that are similar to these:

A501.dwg
A-410 - WALL SECTIONS.dwg
0057P-118W.dwg
HL102E12.dwg

As you see, the file names can range from the simplest AIA version, to a more descriptive method, or all the way to a rather cryptic name that only you understand. Which is really the point… only you know what means something and what doesn't.

Every situation is different, but I suggest analyzing the path instead of the file name. This method will ensure that the file is in a network location (not in My Documents or on a USB drive), and will also tell you the what the project is. Unless that info is built into your file naming standards (and you can enforce where the file is located), analyzing the file name may not give you enough information.

The first thing I do is to define some global functions to get the file path, the drawing name, and the drive letter:

```
(defun *dwgPath* () (strcase (getvar "dwgprefix")))
(defun *dwgName* () (strcase (getvar "dwgname")))
(defun *dwgDriveLtr* () (substr (*dwgPath*) 1 1))
```

Note: I use asterisks to designate global functions or variables. More on this later.

You could go to the trouble of concatenating the path and the name together, but you're probably going to analyze one or the other, so I skip this step and handle them separately, if necessary.

You may only need to find the drive letter of the current drawing. That may tell you everything you need to know. For example, all your "***B**uy More*" store projects are on the **B**:\ drive, and all your "***L**arge Mart*' projects are on the **L**:\ drive. If this is you, your analysis is done.

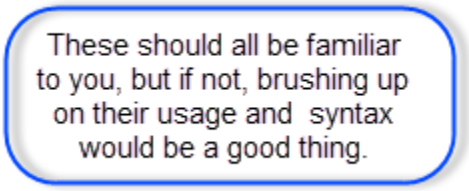
Typically, though, you'll probably need to look for a keyword or a number that will indicate to you that the file is a) within a proper project folder, and b) what the project is so that you can easily find the master page setup file for that project.

***Sidebar**:* LISP is an acronym for LISt Processing. It loves lists! From car to cdr to assoc to nth… nothing makes it happier. This is **not** the case for strings. Using LISP to parse and analyze strings is clunky at best. This is most likely why it's not called STRIP.

During this phase, you'll be using standard LISP string and equality functions, along with some visual lisp string functions.

For my examples found in section 4 of this handout, you'll use functions such as:

```
=
wcmatch
substr
strcat
strcase
findfile
vl-string-search
vl-string-position
```



You may already have some standard parsing functions written. If so, good for you! Feel free to bypass my functions and use your own (or your preferred method). There's no sense re-writing code.

At this point, you should know a few things about the file that is being opened.

- Where it is
- What it is
- Is it valid?
- Where is its master page setup file?

Armed with this info, you're ready to move on to the next step.

Build a path to the Page Setup File

I've mentioned "is the drawing valid" a couple of times. The main thing I mean by this, is it in a proper project location on the network? While we could further concern ourselves with making sure it is a sheet file as opposed to a 3D model, or a detail or a block etc. in most real world scenarios it doesn't matter.

By this I mean that Page Setups are inherently used for sheet files. You're using them to produce a consistent construction document. Typically, you're not printing from a 3D model or a 2D layout. If your company standard does have page setups in place for files of that type, you can add analysis to the previous step to further decide on what page setup file needs to be use, but for the most part, importing the project page setups into design files won't hurt anything, so I just let it happen.

So, as long as the file being opened is valid, and you know where it is and what it is, you can easily build a path to your master page setup file for that project.

Page Setups can be imported from .dwg or .dwt files, but not .dws

If you've not already done so in the analysis stage, do a *substr* on the drawing path to get to the folder within the project where the page setup file resides, then concatenate the path onto the file name. Using the last example path on page 7, Your code may look similar to this:

```
;; Set the path of the file being opened to the var 'dpath'.
(setq dpath (*dwgPath*))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\Eng\\CAD\\MEP\\"

;; Find the start position of the folder where the Page Setup file resides:
(setq startPos (vl-string-search "CAD\\" dpath));; Note CAD\\ is 4 digits.
> 42

;; From the front of the search string, go 4 digits past where CAD is found.
(setq PageSetupPath (substr dpath 1 (+ 4 startPos)))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\ENG\\CAD\\"

;; Concatenate the path and file together, while also making sure it exists.
(setq PageSetupFile (findfile (strcat PageSetupPath "MasterPageSetup.dwt")))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\ENG\\CAD\\MasterPageSetup.dwt"
```

Remember, when working with paths, Autolisp sees a backslash as an escape character, so either use two backslashes "\\" or a single forward slash "/".

Flush the existing Page Setups

The hard part is done now. There are only two simple steps left. Unless you need to keep some page setups in the file (a possible example to follow in section 4d), you need to delete all the existing setups in preparation to import the master set.

You're probably wondering why I'm deleting the existing setups instead of overwriting them. We could do that, but I want my users to expect the same list of setups every time. If I have a user decide to create their own page setup, not only is it against our standards, it allows for the possibility of it 'infecting' other files in the project. As the CAD Manager, it's my job to set up the project with any and all appropriate setups. My users are instructed to get with me if one is defined wrong, or if we need anything additional. The delete and import method gives me more control.

Below is the sample code to delete all Page Setups. Note we use the visual lisp commands here, so if you haven't already done so, make sure you make the call to *vl-load-com.*

```
(vl-load-com)

(vlax-for
  ps
  (vla-get-plotconfigurations
    (vla-get-activedocument (vlax-get-acad-object)))
  (vla-delete ps))
```

Import the project Page Setups

Finally, we're ready to import the page setups from the master file. We set the variable PageSetupFile to the full path where the master file resides, so all we have to do is call the command line function *–PSETUPIN*. We'll suppress *CMDECHO* while it's working, and monitor the system variable *CMDACTIVE* to pass the "No" answer to the redefinition prompt.

```
(setvar "CMDECHO" 0)

(command "-PSETUPIN" PageSetupFile "*")

;; say no to redefining if prompted.
(while (not (zerop (getvar "CMDACTIVE")))
  (command "N"))

(setvar "CMDECHO" 1)
```

3. Organizing your Autolisp code

Just as standard naming conventions and file / folder structure organization are keys to successfully running these programs, I think good code organization is as well. I organize my startup code into two files. AutoCAD automatically loads ACADDOC.lsp when a drawing file is opened, and in it, the first thing I do is to load Globals.lsp, as it contains variables and functions that are needed to successfully run ACADDOC.lsp. I'll explain each in brief here, with some examples of what I do (in addition to the scope of this class). A full version of each is also posted to the class website.

Could you just add the functions in Globals to your ACADDOC file? Yes, of course you could. I found that at some point though, that made ACADDOC.lsp way too big, so it was a logical step to pull out all my global variables and functions into a separate file. I have an old boss and mentor to thank for drilling that concept into my head. It turns out he was much smarter than I was, so ignoring his advice would be foolish.

Globals.lsp

I have a standard header I put in all my lisp files, which includes a description. Here is my description for my Globals file:

> This file is loaded via the ACADDOC.lsp file and sets global variables and functions based on the current drawing and environment for use in other routines.

That's as good of a concise description as I can come up with, so I'll plagiarize myself here.

As I've already noted, I use this file to define predicate functions to get the file's path, name and the drive letter:

```
(defun *dwgPath* () (strcase (getvar "dwgprefix")))
(defun *dwgName* () (strcase (getvar "dwgname")))
(defun *dwgDriveLtr* () (substr (*dwgPath*) 1 1))
```

Note the asterisks… there's nothing magic about them, they're just my method for designating a global function or variable – that is, something I'll have available in every drawing session. You may have a different method already, or might decide to make up your own.

Some of the other things I define in Globals.lsp include some functions to determine what office's server the file resides on (we'll see more on that in later examples):

```
(defun *is-Houston?* () (wcmatch (*dwgPath*) "*2203-HOU*"))
(defun *is-StLouis?* () (wcmatch (*dwgPath*) "*1648-STL*"))
```

We sometimes have various versions in production at the same time. I may need to know what version is opening the file, so I use the following predicates:

```
(setq *acadVer* () (getvar "acadver"))
(defun *is-2016?* () (= (substr *acadVer* 1 4) "20.1"))
(defun *is-2017?* () (= (substr *acadVer* 1 4) "21.0"))
```

I also may need to know what discipline the file belongs to. Here are my discipline predicates – yours will of course vary depending on your industry.

```
(defun *is-Pipe?* () (wcmatch (*dwgPath*) "*PIPING*"))
(defun *is-Struct?* () (wcmatch (*dwgPath*) "*STRUCTURAL*"))
(defun *is-Elec?* () (wcmatch (*dwgPath*) "*ELECTRICAL*"))
(defun *is-Iso?* () (wcmatch (*dwgPath*) "*ISOMETRICS*"))
```

Note: Predicates are functions that test their arguments for some specific conditions and returns nil if the condition is false, or some non-nil value if the condition is true. You'll find that my predicates don't use arguments, but the theory is the same.

You should already be familiar with some built in predicates in Autolisp. Most (but not all) end in "p". Examples include **listp**, **numberp**, **zerop** and **atom**. To differentiate my custom functions from Autolisp, I use the "is-" prefix.

I also go ahead and define the "Radians To Degrees" and "Degrees To Radians" conversion functions here, as I use them in a number of routines.

```
(defun rtd (r)(* (/ r pi) 180.0))
(defun dtr (d)(* (/ d 180.0) pi))
```

I work at work and an oil and gas firm, and we use two different software products as our 3D design tools; AutoCAD Plant 3D and a 3rd party application, CADWorx. I definitely need to know which one is running. There's always something that you can find to differentiate flavors of programs, in this case you'll only find a certain variable when Plant 3D is running. As for CADWorx, it loads version specific ARX routines. My predicates key of off these facts.

```
(defun *isPlant3D?* ()
  (and (getvar "plantcontentfolder")
       (*is-2015?*)))

(defun *isCADWorx?* ()
  (or (member "cadworxplant2013.arx" (arx))
      (member "cadworxplant2015.arx" (arx))))
```

As we'll see later in my real-world examples, I may also need to determine what project the file belongs to. Depending on the application the predicates may look like this:

```
(defun *isLarge_Mart_ENG?* ()
  (wcmatch (*dwgPath*) "326781"))

(defun *isBuyMoreIDIQ?* ()
  (vl-string-search "325864" dpath))
```

Finally, I define the functions I'll need to analyze the drawing path, and to parse out and return the portion I need. We'll see these in more detail later. My full Globals.lsp file is posted as additional materials on the class website.

ACADDOC.lsp

Hopefully you already have a well-organized ACADDOC.lsp file. Of course it needs to be located in a support path where AutoCAD can find it (don't forget to add any custom path to the Trusted Paths list). My ACADDOC will be posted to the class website, so you can see (or borrow from) the entire thing, but I'll mention some of the things I do in it, in addition to the code necessary for this class.

I do three very important things first. I go ahead and run *vl-load-com* so that the vl functionality will be available to me in any lisp code I load (you'll see it being called explicitly in later examples. Call it here and you can remove it there).

Next, I call *acad-push-dbmod*. I love this. If you use ACADDOC now, and you don't call *acad-push-dbmod* before doing anything else (such as setting sysvars), AutoCAD will think you've made changes to the database. This can be terribly irritating. Calling it first and then calling *acad-pop-dbmod* when you're finished will keep this from happening.

Here is more info from the help file on *acad-push-dbmod* and *acad-pop-dbmod.*

Finally, I make sure I load my Globals.lsp file. It's in the same support folder as ACADDOC.lsp, so I know it will be found, thus the lack of a qualified path.

```
(vl-load-com)
(acad-push-dbmod)
(load "globals.lsp")
```

Of course, the most important thing I load and run (for this class anyway), is my code to delete and import the correct page setups for the current file. (Your file name and function may vary)

```
(load "managePageSetups.lsp")
(deleteAndImportPageSetups)
```

Moving on, I also set variables so all my users will be on the same page. Here are 3 examples:

```
(setvar "XLOADCTL" 2)
(setvar "SAVETIME" 10)
(setvar "REMEMBERFOLDERS" 1)
```

I think I counted 36 variables that I set in this section.

As I mentioned before, I may have various versions running. I control the .DWG version being saved with this simple expression:

```
;; Set SAVETYPE to 2013 format
(if (setq opensaveObj
      (vla-get-opensave
        (vla-get-preferences (vlax-get-acad-object))))
  (vla-put-saveastype opensaveObj 60))
(if opensaveObj (vlax-release-object opensaveObj))
```

Users can still change it during SAVEAS, or even via OPTIONS, but I'll always make sure it's what I want it to be when a file is opened.

I have a number of handy functions that aren't in our custom menu anywhere – they're just called via the command line. Instead of pre-loading them, I use the *autoload* function to load them "on demand". If you're not familiar with autoload, I encourage you to check it out.

```
(autoload "northingeasting" '("NE"))
```

My full ACADDOC.lsp file is posted as additional materials on the class website.

4. Real world examples with code

Of all the difficulties involved in writing this handout, the most difficult was trying to imagine all the scenarios you might face, and trying to account for them in my code examples. The fact of the matter is; it can't be done. Everyone's situation is different.

Given that, following are four examples of what I've seen, ranging from the simplest to the most complex. That is, the most complex of these examples – the real world can always be worse!

About as easy as it gets

As I stated at the beginning of this handout, this class is intended for CAD Managers who are tasked with supporting projects for multiple clients, and/or multiple disciplines where the page setups may differ for each.

You may be one of the lucky ones who will be able to employ your code in the easiest fashion I can think of. This might be your scenario:

Every drawing and every user will always use a single page setup (or group of setups) for every drawing in your company. So there's not need to analyze the drawing upon opening it.

In this case, all you have to do is flush the existing page setups and import the master set.

My example solution begins on the next page.

To begin, in my Globals.lsp file, I set a global variable for the path to the master page setups, then a macro that allows me to add any file name to that path.

Note: Why do I do this, you may ask? I prefer to set up a single location to define any paths I may need in any of my lisp code. In addition to the below example, I have over a dozen paths defined. I do this for the inevitable changes that may come. Here are two examples that have happened to me.

- Either the drive mapping changes, or the share itself. In the five years I've been at my company, we've changed ownership or names three times, each resulting in a tweak to our network.
- You make a change to the structure yourself. Let's face it – sometimes you have a better idea!

This method gives you more flexibility when change happens. As an example, if you had hard coded paths to a symbol library sprinkled throughout your code, you'd have to find them, and change it all locations. This way, in conjunction with the concatenation macro, you only have to worry about changing the path in one place.

Next, I define the function *updatePageSetups*, that retrieves and deletes all of the current page setups, then imports the master set from the appropriate drawing file.

Finally, is ACADDOC.lsp, I make sure both Globals.lsp and *updatePageSetups.lsp* are loaded, then I run the *updatePageSetups* function.

The three examples that follow will follow the same flow:

- Variables and functions in **Globals.lsp**
- Preparation of the **updatePageSetups.lsp** file
- Loading and running the code in **ACADDOC.lsp**

```
;; In Globals.lsp
(setq *pageSetupPath* "B:\\2016\\RETAIL\\BUY MORE\\SUPPORT\\PAGE SETUPS\\")
(defun with-PageSetupPath (str /)(strcat *PageSetupFile* str))

;; In updatePageSetups.lsp file
(defun updatePageSetups ()
  (vl-load-com)

  ;; Here, I'm making sure my PageSetupFile exists before doing anything.
  (if (setq PageSetupFile
            (findfile (with-PageSetupPath "Buy More Page Setups.dwg")))
    (progn

      ;; Gather all the page setups in the drawing.
      (setq plotConfigs
         (vla-get-plotconfigurations
           (vla-get-activedocument (vlax-get-acad-object))))

      ;; Delete all the page setups in the drawing.
      (vlax-for ps plotConfigs
        (vla-delete ps))

      (setvar "CMDECHO" 0)

      ;; Use the macro as defined in Globals.lsp
      (command "-PSETUPIN" PageSetupFile "*")

      ;; say no to redefining if prompted.
      (while (not (zerop (getvar "CMDACTIVE")))
        (command "N"))

      (setvar "CMDECHO" 1)) ;; end the progn here
    (princ))

;; In ACADDOC.lsp
(load "globals.lsp")
(load "updatePageSetups.lsp")
(updatePageSetups)
```

The next best thing to as easy as it gets

The next easiest scenario might be the two or more programs your company works for. The Buy More and Large Mart example I used earlier (although I use two in my example, it could be any amount).

In this scenario, every Buy More project uses the same set of Page Setups, which are in a file within a support structure for that client. Something like this:

The path of the drawing being opened may be this:
B:\2016\Retail\Buy More\Burbank\91501\Eng\CAD\MEP

… and the master page setup file for Buy More projects is located here:
B:\2016\Retail\Buy More\Support\BuyMorePageSetups.dwg

You'll have to do some quick analysis of the path to decide what project you are in. You may only have to look at the drive letter (B for Buy More; L for Large Mart; O for Orange Orange etc.)
Follow the Example 1 code segments in the code below.

If your projects aren't grouped by different drive letters, it's still a simple string search for the proper keyword to know what project you are in. (e.g. Do a wcmatch for *Buy More*).
Follow the Example 2 code segments in the code below.

Of course, the Large Mart client would work the same way. Depending on how many client groups you have, you may end up using the *cond* function to finalize your decision tree. I illustrate this in the Example 2 segment below.

```
;; In Globals.lsp
(defun *dwgPath* () (strcase (getvar "dwgprefix")))

;; EXAMPLE 1
(defun *dwgDriveLtr* () (substr (*dwgpath*) 1 1))

;; EXAMPLE 2
(defun *isBuyMore?* () (wcmatch (*dwgpath*) "*BUY MORE*"))
(defun *isLargeMart?* () (wcmatch (*dwgpath*) "*LARGE MART*"))
(defun *OrangeOrange?* () (wcmatch (*dwgpath*) "*ORANGEORANGE*"))


;; In updatePageSetups.lsp file
(defun updatePageSetups ()
  (vl-load-com)

    ;; EXAMPLE 1
  ;; I'm using two clients here, making the test a simple If/Then/Else.
  ;;
  (if (= (*dwgDriveLtr*) "B")
    ;; Then, we know it's a Buy More project
    (setq
      PageSetupFile
       "B:\\2016\\Retail\\Buy More\\Support\\BuyMorePageSetups.dwg")

    ;; Else, it has to be Large Mart (our only two clients)
    (setq
      PageSetupFile
       "L:\\2016\\Retail\\Large Mart\\Support\\LargeMartPageSetups.dwg"))
```

```
;; EXAMPLE 2
;; This example uses 3 or more clients, so I'm using a COND statement,
;; grouping each condition based on the frequency of the test statement
;; returning T (e.g. you have hundreds of Buy Mores, but only 5 Orange
;; Orange projects. No need for the cond to look at more than necessary.
;;
(cond
  ((*isBuyMore?*)
   (setq
     PageSetupFile
      "B:\\2016\\Retail\\Buy More\\Support\\BuyMorePageSetups.dwg"))
  ((*isLargeMart?*)
   (setq
     PageSetupFile
      "L:\\2016\\Retail\\Large Mart\\Support\\LargeMartPageSetups.dwg"))
  ((*OrangeOrange?*)
   (setq
     PageSetupFile
      "O:\\2016\\Retail\\Orange2x\\Support\\Orange2xPageSetups.dwg")))

;; Here, I'm making sure my PageSetupFile exists before doing anything.
(if (and PageSetupFile (findfile PageSetupFile))
  (progn

    ;; Gather all the page setups in the drawing.
    (setq plotConfigs
       (vla-get-plotconfigurations
         (vla-get-activedocument (vlax-get-acad-object))))

    ;; Delete all the page setups in the drawing.
    (vlax-for ps plotConfigs
      (vla-delete ps))

    (setvar "CMDECHO" 0)

    ;; Import the master page setups
    (command "-PSETUPIN" PageSetupFile "*")

    ;; say no to redefining if prompted.
    (while (not (zerop (getvar "CMDACTIVE")))
      (command "N"))
    (setvar "CMDECHO" 1))) ;; end the progn here

  (princ))


;; In ACADDOC.lsp
(load "globals.lsp")
(load "updatePageSetups.lsp")
(updatePageSetups)
```

Getting a little more complicated

If the first two examples are what you deal with, consider yourself lucky. Things can get more complicated.

Say you have multiple projects going on, spanning a number of different clients. Some of the projects are based on your company standards, but some are using client standards – therefore client specific page setups.

You should know what folder contains the master page setup file for each project, and what its name is… as you set it up. All you have to do now is to figure out what project is being opened, and build a path to the page setup file. As outlined in Section 2 – Workflow Outline, you'll need to search the drawing path for a key-string that indicates to you that it is indeed a valid project, parse the string down to the necessary folder, and build the full path to the page setup file for that project. We'll continue building on the previous example, and consider these projects:

B:\2016\Retail\Buy More\Burbank\91501\Eng\CAD\MEP
B:\2015\Retail\Large Mart\Tulsa\74103\Eng\CAD\Arch
B:\2015\Retail\JR Nickels\Buffalo\14212\Eng\CAD\Struct
F:\2016\Food\Breezy's\Austin\78710\Eng\CAD\Civil

In this case, there's no easy string to key off of. The drives can be different, as are the client names. However, as the CAD Manager you know that your template files are always in the CAD folder within a project. You may have a single template file, or you may be like me and have a template for each discipline. In either case, you should know which file contains your master page setups.

> **Note**: While it might be tempting to search for the string "CAD" in the example paths, if possible, I would suggest adding the leading or trailing folder to the search (assuming it is a standard folder). I've seen too many cases of users creating a CAD folder outside of a project. In this case, searching for "\ENG\CAD\" would greatly lessen the likelihood of getting a false positive on your search.

Up to now, we've used *wcmatch* with wildcards to query the drawing path string. In this case though, with so many possibilities, knowing that a valid path should always contain "\ENG\CAD\", and that our project page setup file is within "\CAD\"; we might want to use *vl-string-search* instead.

What's the difference between them? The *wcmatch* function simply returns T or nil as a result, but *vl-string-search* returns either an integer indicating the start position of the string or nil. This becomes handy here, as we don't really care what the project is, as we're going to use the path up to "\CAD\" to build the full path to our page setup file.

You may want to use both to ensure a valid path though, as it doesn't take much more overhead to do so. Here's an example using the Burbank Buy More path above (set to the *dpath* variable):

```
(if (and (wcmatch dpath "*\\ENG\\CAD\\*") ;; returns T
         (setq startPos (vl-string-search "CAD\\" dpath))) ;; returns 42
```

Of course, if either fails and return nil, the *and* condition fails and you move on with your logic.

Let's continue to look at what this code might look like:

```
;; Set the path of the file being opened to the var 'dpath'.
(setq dpath (*dwgPath*))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\ENG\\CAD\\MEP\\"

(if (and (wcmatch dpath "*\\ENG\\CAD\\*")
         (setq startPos (vl-string-search "CAD\\" dpath)))
> T

  (progn
    (setq PageSetupPath (substr dpath 1 (+ 4 startPos)))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\ENG\\CAD\\"

    ;; Concatenate the path and file together,
    ;; while also making sure it exists.
    (setq PageSetupFile
           (findfile (strcat PageSetupPath "MasterPageSetup.dwt")))))
> "B:\\2016\\RETAIL\\BUY MORE\\BURBANK\\91501\\ENG\\CAD\\MasterPageSetup.dwt"
```

On the next page, we use the prior example to ensure the path's valid, and if so, parse it down using the *substr* function, then concatenate the page setup file onto it. Note that I also use *findfile* after doing the *strcat* to make sure it exists before continuing (i.e. if PageSetupFile is nil, bail out of the routine, or perhaps look in an alternate place).

Since this scenario requires more code than a simple test (as we saw in the first two, easier examples), I prefer to put it into its own function, then call it as necessary. Since I have other functions that rely on the same project path string, I like to keep this function within my Globals.lsp file.

As with each of these examples, once the evaluation and path building phases are done, all that's left is to delete the existing setups, and import the master set. Let's look at the finished code:

```lisp
;; In Globals.lsp
(defun *dwgPath* () (strcase (getvar "dwgprefix")))

(defun *get-projPath* (/ dpath startPos)
  (setq dpath (*dwgPath*))

  ;; As shown in the code snippet above, I use both methods
  ;; here to check the path validity.
  (if (and (wcmatch dpath "*\\ENG\\CAD\\*")
           (setq startPos (vl-string-search "CAD\\" dpath)))

    ;; Go 4 characters past where CAD\\ is found...
    (setq *projPath* (substr dpath 1 (+ 4 startPos))))

  ;; Make sure the path is returned.
  *projPath*)


;; In updatePageSetups.lsp file
(defun updatePageSetups ()
  (vl-load-com)

  ;; Make sure *get-projPath* returns non-nil.
  ;; I use this method so that my strcat won't error.
  (if (and (setq projPath (*get-projPath*))

           ;; Concatenate the path and file together,
           ;; while also making sure it exists also.
           (setq PageSetupFile
                  (findfile
                    (strcat projPath "MasterPageSetup.dwt"))))

    (progn

      ;; Gather all the page setups in the drawing.
      (setq plotConfigs
         (vla-get-plotconfigurations
           (vla-get-activedocument (vlax-get-acad-object))))

      ;; Delete all the page setups in the drawing.
      (vlax-for ps plotConfigs
        (vla-delete ps))

      (setvar "CMDECHO" 0)

      ;; Use the macro as defined in Globals.lsp
      (command "-PSETUPIN" PageSetupFile "*")

      ;; say no to redefining if prompted.
      (while (not (zerop (getvar "CMDACTIVE")))
        (command "N"))

      (setvar "CMDECHO" 1)) ;; end the progn here
    (princ))
```

I hope this isn't you (because it's me)

Finally, welcome to my world, which is basically just like the last example, but with these additional complications:

- There may be discipline and/or drawing type specific page setups.
  - We have a piece of 3rd party software that generates isometric drawings. It does it the way it wants to, and I can't alter that. In this case, importing our normal page setups doesn't make sense, so I bypass them in favor of a custom set.
- Other offices may have their page setups in the file as well. We don't want to delete those.
  - We have offices in a number of other cities, and we often share work on projects. I think it would be rude to delete the page setups that our team members in another location use, so I need to filter them out.
- Although you strive for consistency, one or more project structures may be outliers.
  - There's always one, isn't there? In our case, we've had a few projects that for hatever reason didn't fit our standard "mold". I know which ones they are, and how they differ, so I can adjust for it in my code.

Let's look at how I handle these situations.

In the first example, the isometrics are all to reside in a specific folder within the PIPING folder called ISOMETRICS. The first thing I do is to create a predicate function which determines whether or not the drawing I'm in is under an ISOMETRICS folder:

```
(defun *is-Iso?* () (wcmatch (*dwgPath*) "*ISOMETRICS*"))
```

Is-Iso? is one of a handful of discipline predicates I use. I define one for all of our disciplines.

Using this function, I can easily filter out any drawing that returns T for this test. Continuing to build on the previous example, I can easily filter out any ISO's by adding another test condition to the *and* function. Note that it is prior to the *setq*, allowing the *and* to fail before it even gets there.

```
(if (and (wcmatch dpath "*\\ENG\\CAD\\*")
         (not (*is-Iso?*))
         (setq startPos (vl-string-search "CAD\\" dpath)))
> T
```

The above code assumes you want to ignore any ISO's. What if you wanted to load a special set? In this case, I have a template file that contains the special page setups for ISO's (again, in my standard location of "/CAD/"). Your code may look similar to this, where I set the variable PageSetupFile to either my ISO template or the master file, depending on how *Is-Iso?* evaluates.

```
(if (*is-Iso?*)
  (setq PageSetupFile (findfile (strcat PageSetupPath "iso-template.dwt")))
  (setq PageSetupFile (findfile (strcat PageSetupPath "MasterPageSetup.dwt"))))
```

This example scenario uses ISO's as an example, but in theory it could be any type of drawing that you want to filter. It could be template files, client files, a particular discipline or file type. As long as your naming convention is set and adhered to, it's simple to write a predicate function to quickly analyze what that special file might be.

In the second example I want to find any page setup that our other offices have saved, and leave them alone when deleting and importing my own set.

Before that even happens though, I first want to make sure that the user hasn't opened a file that resides on their server. If that's the case, I don't want to do anything.

Again, I'll begin with a predicate function that looks for a known string in the drawing path.

In these examples, I just follow our corporate naming structure to decide if the file is on a different server than our local one.

```
(defun *is-Houston?* () (wcmatch (*dwgPath*) "*2203-HOU*"))
(defun *is-StLouis?* () (wcmatch (*dwgPath*) "*1648-STL*"))

(if (not (or (*is-Houston?*)
             (*is-StLouis?*)))
  (progn
    ;; ... continue with your code here.
    ))
```

Of course that's pretty simple, but it gets a little more complicated when you're needing to analyze each page setup. There are a couple of ways to go about it, one would be to compare each existing page setup name against a list of your master project setups, or if you have a good naming convention, you should be able to do a key word string search on the existing name. Perhaps you'll need to dig into the page setup itself to know whether to delete it or not. Let's look at all three examples.

First, building on the page setup delete code found on page 10, we can create a global list of all our page setup names. You list may look like this, resulting in the following code.

```
(setq *MasterPageSetupList* '("HPLaserJetPro" "OCEPlotWave360"
                              "EpsonMX80DotMatrix" "Gutenberg1440"))

(vlax-for
  ps
  (vla-get-plotconfigurations
    (vla-get-activedocument (vlax-get-acad-object)))

  ;; get the current page setup name, and check it against the list
  (if (member (vla-get-name ps) *MasterPageSetupList*)
    (vla-delete ps)))
```

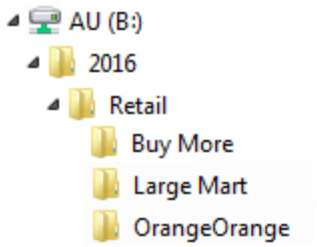
Next, knowing what key strings to search for, we can easily filter out any page setup with either “HOU” or “STL” within its name:

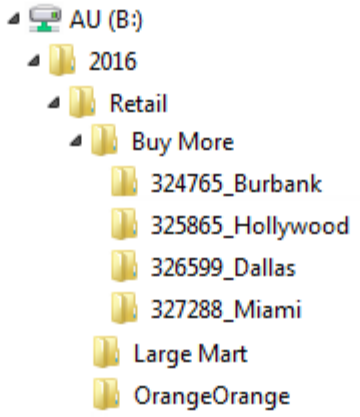
```
(vl-load-com)

(vlax-for
  ps
  (vla-get-plotconfigurations
    (vla-get-activedocument (vlax-get-acad-object)))
  (if (not (wcmatch (vla-get-name ps) "*HOU*,*STL*"))
    (vla-delete ps)))
```

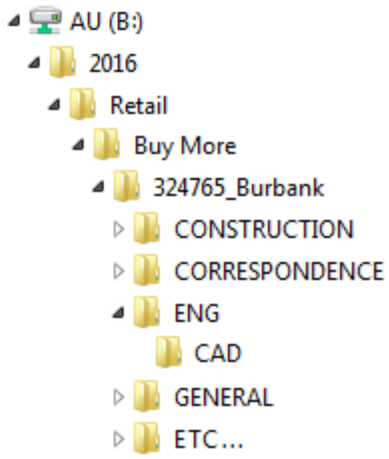
Finally, if the page setups are not named in a standard fashion; We can use the *vla-get-configName* function to find out what print device it is set to, and analyze that.

```
(vlax-for
  ps
  (vla-get-plotconfigurations
    (vla-get-activedocument (vlax-get-acad-object)))
  (setq deviceName (vla-get-configName ps))
  (if (wcmatch deviceName "*PTUL*")
    (vla-delete ps)))
```

Finally, let’s examine the outlier project scenario. Our typical setup is to create a client group on our projects drive, then create individual projects beneath that. Below is an expanded mock-up of what our setup might look like. Yours may be similar.

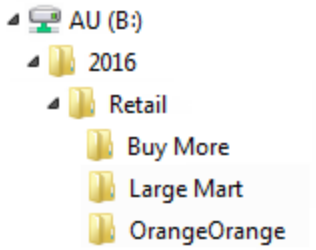


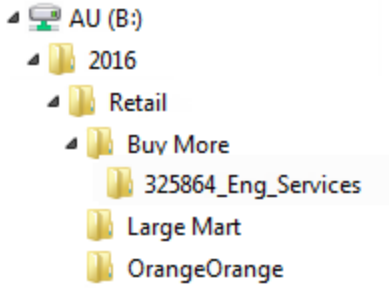


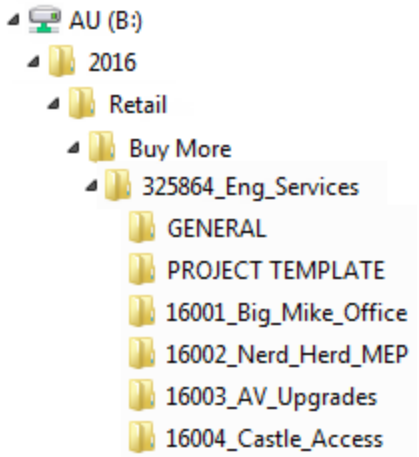


Once again we'll use Buy More as an example. In the last few years, we began working on a series of projects that were all grouped under a single client work order. Similar to an IDIQ, which we typically don't do. As you can see, we still have the same client structure under Projects, but then there is a single project that encompasses a number of others. You can see those in the last image. Obviously this doesn't match our standard (above), so I need to account for it in my evaluation programming.

IDIQ is a U.S. federal government contracting acronym meaning indefinite delivery/indefinite quantity. This is a type of contract that provides for an indefinite quantity of supplies or services during a fixed period of time







Again, as long as you know what your project setup is supposed to look like, even if it is different from your standard, you can account for it. In this case, knowing that all my templates, title blocks, etc. would be supplying each sub project, I changed their default location to the GENERAL folder under 325864_Eng_Services.

Armed with that information, I can conditionalize my code to account for the differences. Again, I'll begin with a global predicate function. We use accounting codes in our naming structure, so it's pretty easy to identify the project, as it should be unique.

```
(defun *isBuyMoreIDIQ?* () (wcmatch (*dwgPath*) "*325864*"))
```

Up till now, my example predicate functions have been a simple call to *wcmatch*. You can accomplish the same thing by using *vl-string-search*. Instead of returning T or nil like *wcmatch*, it will return an integer or nil. In this case, let's use *vl-string-search* so we can use its return value in our code.

```
(defun *isBuyMoreIDIQ?* () (vl-string-search "325864" dpath))
```

Of course these complications can show up in any scenario, and the application of the logic may be implemented differently, but for this example, it makes the most sense to alter the *get-projPath* global function. We saw the main logic for this code back in the second scenario; Getting a little more complicated. We'll expand on that here.

```
;; Here we use the value returned by *isBuyMoreIDIQ?*
(if (not (setq startPos (*isBuyMoreIDIQ?*)))

  ;; Then, if not a Buy More IDIQ, and it is a standard path...
  (if (and (wcmatch dpath "*\\ENG\\CAD\\*")
           (setq startPos (vl-string-search "CAD\\" dpath)))

    ;; Do your normal parsing here...
    (progn
      ;; find a common string in std. paths, and get its starting position.
      (setq startPos (vl-string-search "CAD\\" dpath))
      ;; from the front of the full path, go 4 digits past from "CAD".
      (setq *projPath* (substr dpath 1 (+ 4 startPos)))))

  ;; Else, it's the outlier IDIQ Buy More Services project.
  ;; Since the predicate returns the position, just add the needed folder.
  (setq *projPath* (strcat (substr dpath 1 (+ 20 startPos)) "GENERAL\\")))
```

As has been our norm, *projPath* will get passed to the updatePageSetups function, and we'll build the path to our master page setup file there.

Now that we've gone through these examples, let's look at what our code might look like, having incorporated all three complications

```
;; In Globals.lsp
(defun *isBuyMoreIDIQ?* () (vl-string-search "325864" dpath))
(defun *is-Houston?* () (wcmatch (*dwgPath*) "*2203-HOU*"))
(defun *is-StLouis?* () (wcmatch (*dwgPath*) "*1648-STL*"))

(defun *get-projPath* (/ dpath startPos startPos)
  (setq dpath (*dwgPath*))

  ;; Here we use the value returned by *isBuyMoreIDIQ?*
  (if (not (setq startPos (*isBuyMoreIDIQ?*)))

    ;; Then, if not a Buy More IDIQ, and it is a standard path...
    (if (and (wcmatch dpath "*\\ENG\\CAD\\*")
             (not (*is-Iso?*)) ;; 1st Complication. Filter out ISOs here.
             (setq startPos (vl-string-search "CAD\\" dpath)))

      ;; Do your normal parsing here...
      (progn
        ;; find a common string in std. paths, and get its starting position.
        (setq startPos (vl-string-search "CAD\\" dpath))

        ;; from the front of the full path, go 4 digits past from "CAD".
        (setq *projPath* (substr dpath 1 (+ 4 startPos)))))

    ;; Else, it's the outlier IDIQ Buy More Services project.
    ;; Since the predicate returns the position, just add the needed folder.
    (setq *projPath* (strcat (substr dpath 1 (+ 20 startPos)) "GENERAL\\")))

  *projPath*)
```

```
;; In updatePageSetups.lsp file
(defun updatePageSetups ()
  (vl-load-com)

  ;; Get all the Page Setup objects
  (setq plotConfigs
        (vla-get-plotconfigurations
          (vla-get-activedocument (vlax-get-acad-object))))

  ;; Delete each Page Setup that's not from Houston or St. Louis.
  (vlax-for
    ps
    (vla-get-plotconfigurations
      (vla-get-activedocument (vlax-get-acad-object)))
    (if (not (wcmatch (vla-get-name ps) "*HOU*,*STL*"))
      (vla-delete ps)))

  ;; Build the string to the project page setup file.
  (if (*is-Iso?*)
    (setq PageSetupFile (findfile (strcat *projPath* "iso-template.dwt")))
    (setq PageSetupFile (findfile (strcat *projPath* "BuyMoreIDIQ.dwt"))))
  ;; As long as the file exists, continue
  (if PageSetupFile
    (progn
      (setvar "CMDECHO" 0)

      ;; Import all page setups.
      (command "-PSETUPIN" PageSetupFile "*")

      ;; Answer N to prompt
      (while (not (zerop (getvar "CMDACTIVE")))
        (command "N"))
      (setvar "CMDECHO" 1)))

  (princ))

;; In ACADDOC.lsp
(load "globals.lsp")
(load "updatePageSetups.lsp")
(updatePageSetups)
```

5. Conclusion

As stated previously, named page setups are a vast improvement over the way we had to manage plotting in the “old days”. As a CAD Manager, it should be your responsibility to make sure the process is as consistent and easy as possible. I find that users are inherently lazy (not in a bad way). By that I mean they’ll gravitate toward the easiest method possible to accomplish a task.

My philosophy is to make the right way to do things so much easier to do than the wrong way, that they'll actually have to work harder to do it the wrong way. After all, isn't that what most instructional AU classes are all about – reducing picks and clicks to make our work more accurate and efficient?

I've said a number of times that everyone's situation will be different. No company has the same file or folder naming conventions. It's up to you to find the keys in your naming structure to apply the programming concepts I've presented. Once you do, and once you incorporate your version of my solution, hopefully you too can check off "plotting problems" from your to-do list.