Magically Fix your Broken AutoCAD® Drawings with Visual LISP®

Matt Stachoni – CADapult Ltd

CP3821  Tired of manually cleaning up inaccuracies due to human error in your architectural AutoCAD drawings? Would you like to build an easy-to-use Visual LISP application that can eliminate this day-to-day drudgery? Then this class is for you! We will look at a series of problem architectural drawings and build a Visual LISP application that will clean up all of the geometric inaccuracies automatically and easily to provide a sound foundation for the documentation process. This class will focus on problem solving, using a detailed algorithm to build the programming logic. As we build the application from the ground up, we will concentrate on fundamental programming skills such as developing good style, readability, error control, design elegance, and optimizing performance.

Learning Objectives
At the end of this class, you will be able to:

- Understand and apply solid, time-tested Visual LISP programming techniques to solve production and design-related problems
- Understand how Visual LISP's VLA-functions improve processing speed across large data sets
- Break large problems down into smaller parts, and build highly modular and reusable program code
- Develop good programming style, write elegant-but-fast support functions, and optimize your program’s performance
- Discover some time-tested techniques to keep yourself drawing accurately.

About the Speaker:
Matt is the senior AEC applications and building information modeling (BIM) implementation technical specialist for CADapult Ltd., an Autodesk Silver reseller in the Mid-Atlantic region. Matt specializes in Revit® training and BIM implementation for design firms and the construction industry. He has over 20 years of experience as a CAD and IT Manager for several A/E firms in Delaware, Pennsylvania, and Boston, Massachusetts, and has been using Autodesk® software since 1987.

matt.stachoni@cadapult.net
matt@stachoni.com
http://www.cadapult.net/aecblog/
Introduction and Class Notes

In this class we will learn important programming techniques used to fix the broken geometry in your AutoCAD drawings easily, quickly, and predictably. We first learn to identify common drawing problems, create a strategy for fixing them, work out a solution algorithm, and create the necessary programming code to apply the solution.

In the process we will examine additional programming and customization that can aid your day to day drafting tasks, speed up your efficiency, enhance your accuracy, and keep errors from reoccurring.

This handout will cover several things that we probably will not be able to review in class, in order to focus on the problem solving process as well as writing the actual code. It includes a whole host of time-tested drawing techniques, helpful CUI menu customizations, and support programming code that you can utilize to keep your rate of errors quite low.

The AU web site for this class has a ZIP file of all of the VLisp code (and more) that we review in class, which is highly commented for your learning purposes. Study these routines and understand how they work at a low level.

Dealing with Bad CAD

In the thousands of different AutoCAD drawings you have worked with over time, you have all come across instances of Bad CAD™, where things just aren’t right. Either by accident, inattentiveness, or other nefarious circumstance, we have all dealt with drawings that are just a horrible mess.

You know the symptoms:

- Endpoints that should intersect cleanly, but do not;
- Lines that are a “little off” vertical or horizontal;
- Dimensions that do not report or add up properly, even when you explicitly specify distances such as offset values;
- Hatch pattern generation problems;
- Repeated errors in Audits;
- Random crashes and drawing instabilities

The result is an irritating drafting and documentation experience. However, fixing these kinds of drawing issues is actually a fairly easy process once you understand the reasons why your files are difficult to work with in the first place.
The Two Main Types of CAD Problems

In general, drawing file issues fall into two camps: First, there are the problems related to the drawing’s “Named Objects” – that is, errors related to layering, blocks, linetypes, text and dimension styles, and other anomalies associated with the drawing metadata.

For example, let’s say you have Window blocks which have the sub-elements drawn on layer A-WIN-GLAZ (instead of layer 0), but the block references themselves are placed on A-WALL, A-FURN, A-CASE-EXST, etc. When you freeze one layer, half of your drawing disappears.

Or maybe there is no real CAD management in place, so drawings are populated with layers called “1”, “2”, “My_Extra_Stuff”, or worse. Text styles have the wrong fonts or heights. Dimensions look and act weird. We’ve all been there.

These kinds of multi-faceted problems are typically endemic of insufficiently enforced CAD standards, general laziness, or other structural problems in the CAD and design team. Such tales of woe are generally self-inflicted, generated when the originator of the CAD drawing data either did not understand proper standards techniques or just did not care.

For these problems Autodesk has made several tools available – first as Express Tools, now as part of the core program – that can directly help with cleaning up the file, namely, the Layer Translator and Check Drawing Standards, which will help fix these kinds of ills. This class will not cover these kinds of issues.

Instead, we are going to address the second class of problem, that of the harder-to-fix issue of inaccurate drawings. Creating accurate geometry is a skill that develops over time, in conjunction with a certain mindset that simply won’t accept poorly drawn elements.

Dimensional inaccuracy may not be due to sloppiness on the part of human error at all. If you are getting files in formats other than DWG, the source files may be inaccurate, or the conversion process induces inaccuracy. Maybe someone scanned a paper plot and used Raster Design to create linework out of it. Maybe someone converted a PDF or DWF back into a DWG, and the scaling process did not get everything right. There are a million reasons why drawings aren’t accurate; this class will explain how to make sure they are no longer a factor.

The good news is that drawing accurately is no harder than drawing inaccurately; you simply have to use proper technique to enter in proper values for distances and offsets, instead of kind of winging it with the mouse. You also have to develop a certain mistrust about things, constantly double-checking distances, lengths, and coordinates. Diligence is the key.

AutoCAD is, at its core, a very expensive but highly accurate calculator. It has 64-bit floating-point accuracy which is used for all its own internal calculations and AutoLISP operations, providing about 15 decimal places of accuracy. AutoCAD needs this very high level of precision to be able to draw items very far away from the World Coordinate System (WCS) origin, and to complete arithmetic functions to a very high level of accuracy on extremely large/small objects.

To showcase this high level of accuracy, back in AutoCAD version 2.18, Autodesk created a sample drawing file called solar.dwg which was a drawing of the complete solar system –
including the then-planet Pluto – at a 1:1 scale in kilometer units. In this file you could zoom in on the moon and read the plaque placed by the Apollo lunar module.

Regardless, it is put upon the user to draw very accurately at all times. Given the ease at which it is to **not draw accurately at all**, eternal vigilance – and the code you will learn in this class - will indeed go a long way.

**Identifying the Root of the Problem**

The first step in the process is to completely understand the causes of drawing inaccuracies. From there we can strategize on a programmed solution to automate the fixing of CAD geometry.

**Drawing Precision Settings**

Of all of the hundreds of settings used in AutoCAD, the LUPREC and AUPREC system variables are arguably two of the most important in regards to overall accuracy. Both are typically accessed using the UNITS command.

The above screen shot shows the values in the standard out of the box acad.dwt template. As you can see, the linear precision (LUPREC) is set to 1/16”, and angular precision (AUPREC) is set to 0 degrees. What this means is that AutoCAD will, whenever it reports a coordinate, length, or angle, round off those numbers to the nearest 1/16” (0.0625”) or 0 degrees in the display.
That’s incredibly imprecise and simply unacceptable for our needs. In effect, AutoCAD is lying to you when you query an object’s properties, because you don’t see the actual, true coordinate, length, or angle values that are used internally. You see some whitewashed fiction.

Worse, what happens is that users invariably use these imprecise values when inputting additional geometry. As an example, with your Units set as above and your SNAP turned off, draw a vertical line anywhere on the screen. Copy it off to one side, using a random mouse pick to place the copy. Use Polar Tracking or the SHIFT button to constrain the copy horizontally.

Do a Distance command from NEARest to PERPendicular and you will get something like this (where I used a dimension to display the length).

That doesn’t look bad. However, offset the left line a true 4-1/2”, and see what happens:

What the…? Why is there a double line on the right?

Basically, AutoCAD lied to you when it says the original two lines were 4-1/2” apart. They aren’t. You placed them in space using the mouse, and AutoCAD registers those points very accurately. With such a crude precision value, the resultant distance query and dimension give you a false reading of 4-1/2”. When you try to bank on it and perform some operation where you enter in the wrong dimensional value, you get problems like that double line.

Let’s crank the precision up to its highest allowable value for Architectural units, 1/256”:

There’s the problem; the original dimension was not 4-1/2” exactly, but a little off. We didn’t change the geometry; we just adjusted the precision to be enough to display the error. But
1/256” is still only precise to 0.00390625. Let’s move to decimal units and crank that up to 8 decimal places of accuracy:

Here we get even more accurate, as 4-121/256" is actually 4.4725625, a difference of 0.00099097 from the more precise decimal value. And even then, it’s still off, because internally AutoCAD is calculating to 15 or so significant digits, whereas LUPREC is limited to 8 digits of precision. The actual distance could be 4.473647218733456223929.

This kind of problem metastasizes in hundreds of different forms, but the results are always the same: inaccuracy at every turn. What’s more, these dimensional issues always compound themselves over time. For architectural drawings – the focus of this class – you constantly copy existing geometry to make new things. Start by offsetting one slightly-off-axis wall line to make a row of offices, and you have all manner of accuracy problems in the file.

Thus, one of the main causes for inaccurate drawings is Autodesk itself, because it continually ships AutoCAD with templates loaded with such crappy precision values.

The first step in drawing accurately, and understanding how to programmatically fix things, is to set your UNITS precision to their highest values for everything. This translates to an LUPREC value of 8 (1/256” linear precision) and an AUPREC of 8 (0.00000000° of angular precision). I highly recommend setting this automatically in your ACADDOC.LSP like so:

(setvar “luprec” 8)
(setvar “auprec” 8)

Using SNAP and GRID to Explain Accuracy

Luckily there is a built in toolset which can help illuminate and kill these kinds of errors. These are the SNAP and GRID tools.

Both the SNAP and GRID features get a bad rap because people do not understand how to leverage their power for good. They usually see them as hindrances to drawing which just get in the way. Nothing could be further from the truth. With a little bit of customization (explained later in this handout), you can turn these once-disparaged utilities into powerful helpers.

**Snap**

SNAP will automatically snap your cursor to a defined precision point based from the origin of the current User Coordinate System (UCS). When set to a moderately comfortable setting, such as 1/8”, every pick point you pick in space when you draw is going to be at some factor of 1/8” from the origin. Every line will be X/8” long and every coordinate will be at some X/8” distance in...
X/Y/Z. For example, here’s a line drawn in space with SNAP off and then with it on and set to 1/8”:

As you can see, the line drawn on the left has no hope of being drawn accurately without explicitly entering coordinates for the start point and a designated length. The line on the right may not be precisely the right length or in the right position, but for architectural plans – which typically do not show details below 1/8” for most things - it’s at least in the ballpark.

Note that SNAP got a little bit of a change in 2012. In previous versions, your cursor snapped to the SNAP unit setting whenever you moved it around. This was highly annoying. In 2012, the new behavior is to attach the cursor to the snap grid only when a command is active. This means it does not affect your ability to make object selection, which is where it traditionally caused the most pain.

This behavior is driven by the new SNAPGRIDLEGACY system variable; when set to 0 (the default), it will use the 2012 behavior. When set to 1, SNAP will behave as it had in previous releases.

Grid
Let’s now turn on the Grid, which in AutoCAD 2011 got a makeover to look like an actual grid of lines instead of dots. Another new feature is the “adaptive” display, which will optionally show the grid with subdivisions below the grid spacing. I recommend we uncheck this for the purposes of this discussion.

When turned on and set to 1/8” – the same as SNAP - the effect of using SNAP in general drawing tasks is readily apparent. The rectangle on the left was drawn with SNAP off; the rectangle on the right was drawn with SNAP enabled and set to 1/8”.

As you can see, the line drawn on the left has no hope of being drawn accurately without explicitly entering coordinates for the start point and a designated length. The line on the right may not be precisely the right length or in the right position, but for architectural plans – which typically do not show details below 1/8” for most things - it’s at least in the ballpark.

Note that SNAP got a little bit of a change in 2012. In previous versions, your cursor snapped to the SNAP unit setting whenever you moved it around. This was highly annoying. In 2012, the new behavior is to attach the cursor to the snap grid only when a command is active. This means it does not affect your ability to make object selection, which is where it traditionally caused the most pain.

This behavior is driven by the new SNAPGRIDLEGACY system variable; when set to 0 (the default), it will use the 2012 behavior. When set to 1, SNAP will behave as it had in previous releases.

Grid
Let’s now turn on the Grid, which in AutoCAD 2011 got a makeover to look like an actual grid of lines instead of dots. Another new feature is the “adaptive” display, which will optionally show the grid with subdivisions below the grid spacing. I recommend we uncheck this for the purposes of this discussion.

When turned on and set to 1/8” – the same as SNAP - the effect of using SNAP in general drawing tasks is readily apparent. The rectangle on the left was drawn with SNAP off; the rectangle on the right was drawn with SNAP enabled and set to 1/8”.
Once more, any dimensional inaccuracy of the right hand figure is *predictable*. That's key: if the length of the rectangle is supposed to be 2-1/8" but it is drawn at 2", all you have to do is grip edit the right hand edge and drag it over 1/8" to the next snap node.

The rectangle on the left at first looks so dimensionally worthless that you may think it would be faster to simply redraw it from scratch. But that’s not really true; you can select the rectangle, grip edit the midpoint of a segment (with the new SNAP behavior in place, you won’t have any issues picking it), then simply snap to the next grid point. In 2011 and previous versions the same operation may have required you to turn SNAP off to grab the midpoint, then turn it back on to place the point accurately. Often, it is the new “micro” features that mean the most.

But still, doing this sort of thing by hand is for the birds. As you will see, we can automatically fix the badly drawn stuff on the left to look like the clean stuff on the right in mere milliseconds.

**Architectural Units, Drawing Conventions, and Accuracy vs. Precision**

When you are tasked with drawing architectural plans, elevations, sections, and details, the level of accuracy you are working with is generally to the nearest 1/8". Note that I did not say *precision*; the two are not the same thing. Accuracy in this context refers to how close to desired real-world conditions you draw. Drawing both precisely and accurately is a key skill to master.

One of the nice things about architectural design in general is that the materials you are using are in standard widths that are some multiple of 1/8". Metal studs are typically 3-5/8"; gypsum wallboard is 1/2"or 5/8" (and typical interior partitions are 4-7/8"); masonry units are 3-5/8", 5-5/8", or 7-5/8" thick x 15-5/8" long, and so on. (At least in the US, those in the UK and elsewhere are working with millimeters and base-10 numbers, and this 1/8" nonsense is unnecessary).

If you typically lay out partitions using their centerlines, you may wish to go down to 1/16" to snap to the wall lines on either side of the 4-7/8" partition centered on a grid or mullion. For wall sections and details you may need to work down to 1/16" or 1/32" quite often. But for most plan work, a snap setting of 1/8" means you are going to be closer to reality when sketching linework to lay out offices and create partitions.
Of course, you may opt to draw partitions at 5” instead of the real-world 4-7/8”. That represents your predictable level of accuracy and you accept that difference (for whatever reason). But this practice has nothing to do with the problem at hand. The problem most people encounter is when things are just a tiny fraction of an inch off. Meaning, it’s not that the wall is drawn 5” wide when it should be drawn 4-7/8”. Rather, the wall was meant to be drawn at 4-7/8” but ended up being 4-117/128” wide. And that’s the Bad CAD™ we are trying to eliminate.

You may say that it doesn’t matter if the wall thickness is a little off, because you would never dimension it to 4-117/128”; you would round off the precision of the dimension to 4-7/8.

But why do that in the first place? If the wall is drawn correctly, there is no need to round off the precision of dimensions. If you keep them at 1/256”, you can use that high level of precision as an indicator that your linework is inaccurate. Because you should almost always see dimensions to the X/8”, anything that is X/256” indicates an improperly drawn entity OR something that geometrically should be that distance. If you always round off dimensions, inaccuracies will often go undetected.

For the vast majority of architectural design and detailing work, that 1/256” dimension precision value will be fine. When you have dimensions that will, technically, be at some weird X/256” true value due to the geometric lengths involved (e.g., dimensioning angled walls), then knock the precision down for those individual dimensions only. That’s an easy right-click option away.

**Architectural Relativism**

Because of the standard thickness of materials used in modern construction, the other factor that we can exploit is the predictable relative relationship one object has to another.

Let’s say you have a series of new offices in an existing building, and the new 4-7/8” gypsum wallboard on metal stud partitions (in blue) are drawn to align with window openings and centers of mullions as so:

![Diagram of office layout](image)

The top left corner of the exterior wall is located at WCS (263'-8", 386'-4"). With everything in some fractional #/8” relationship to everything else, including the exterior windows, the interior corner of the right hand office ends up at WCS (325'-3 3/4", 368'-11 1/8"). That’s just how the math works out with the horizontal partition aligning to the existing office on the left, and the other partitions aligning to the window openings and centerlines of the mullions.
This reveals that, for orthogonal building plans, if the building itself is based at some known, nice (x/8”) dimension from the origin, and every line you draw is some nice x/8” length, then every point you draw in that coordinate system will be at a nice X/Y distance from the origin.

With this fact in mind we can come up with a rather simple, elegant theory to solve the dimensional accuracy problem. Take an object in the drawing, and adjust its start point, endpoint, and any other important definition point to be located to some precisional value, say to the nearest 1/8”, from the user coordinate system origin. Do this for every object in the drawing and you instantly clean up any dimensional inaccuracies. A partition that is drawn 10'-5 17/256” away from another partition, and has with a width of 4-117/128” now becomes a proper 4-7/8” wide wall. Column grids that were 26'-0 13/256” apart are now 26'-0” apart. And so on.

In action, it looks like this. Before:

And after adjusting each endpoint of the walls, doors, and other elements to the nearest 1/8”:
Functionally Thinking Through the Solution

We first looked at the problem and now understand the reasoning behind the nature of accuracy and precision. This led to the concept that, due to how most orthogonal drawings are laid out based on the distance of one object from the origin and common material thicknesses, if you clean up the coordinate points for the drawing objects to a particular precision value, e.g., 1/8”, you logically clean up any geometric inaccuracies in the process.

Now we need to think through this idea to see how our solution would need to work in the real world for most situations.

We require a way to modify the coordinates of our entities in a predictable, consistent manner. First, we only want to move any points the least amount required to clean up the accuracy. Let’s establish a cleanup precision of 1/8” – that is, we want all orthogonally drawn entities to be a multiple of 1/8” in terms of coordinates, radii, and lengths. That satisfies the needs of typical construction in the U.S.

To clean up a horizontal line whose left endpoint is at X=6.1025, Y=8.5012, we want to move that endpoint to our desired cleanup precision, the nearest 1/8” so X=6.125, Y=8.5. In other words, we round up or down the X and Y values of the 3D coordinate point to our desired precision of 1/8”.

If the line was drawn perfectly horizontal with an exact length of 4-6”, the right hand point would originally be at (5'-0.1025, 8.5012). If we round that coordinate to the nearest 1/8” as well, it would move to (5'-0.125, 8.5), and the length would still be 4'-6”. Thus we have another corollary to our line of thinking: If the original length of a line is an even multiple of the 1/8” precision, and we move each point to the nearest precision point, the distance and direction we moved the first point will be the same distance and direction we move the second point, and the length will not change.

However, if the length was not exactly 4'-6” but some tiny fraction off, you may get a line exactly at 4'-6”, or one of a larger or smaller length, depending on whether either start or end coordinates rounded up or down. However, you would not be more than 1/8” off of the desired length, making any manual dimensional fixes predictable and somewhat rudimentary.

Leveraging Tools that Autodesk Provides

The other problem you are going to face is linework that is not perfectly horizontal or vertical, when it should be. Lines can be tiny fractions of a degree off and throw accuracy into question.

Let’s look this similar example, with a twist. Let’s say we start with the following condition:
Notice the Y-values are unequal and the line is angled, when we want it to be horizontal. If we simply round off the endpoints to the nearest 1/8”, we get this:

The two endpoints rounded to the nearest 1/8”, but the left endpoint rounded down and the right endpoint rounded up, giving an even more divergent line. The errors in each original endpoint were too severe for a simple rounding function to make the line properly horizontal.

This is where **Parametric Constraints** come to the rescue. Parametric constraints were added to AutoCAD 2010, and are used for wiring in desired geometric relationships between objects that can flex and retain those relationships. A rectangular polyline, when constrained, will retain its rectangular shape even when a single grip is edited, because the segments are constrained to be perpendicular to each other and the opposite sides parallel.

The **AUTOCONSTRAIN** command can be used to analyze selected geometry and automatically apply the proper constraints. In the context of our problem, it provides a very powerful cleanup tool to handle slightly-off-axis lines. We can first AutoConstrain the geometry, typically most efficiently using these settings:
With this result:

In this case, the AUTOCONSTRAIN command analyzes the line geometry and applies a Horizontal constraint, as the tolerance angle is set to work on any line that is less than 1.0 degree off of horizontal. This effectively flattens the line removing the discrepancy in the Y direction. If the line was more than 1° from horizontal the constraint would not have been applied. If the line were close to vertical, AUTOCONSTRAIN would apply a vertical constraint.

Of course, the endpoints are still off of SNAP, so applying a rounding function to the line start and endpoints provides this cleaned up element. Note the length stayed the same:

Leveraging AutoConstrain Properly

AutoConstrain would, at first, seem like a perfect tool to use in our utility, but you need to be aware of some things. AutoConstrain could be implemented either on the whole selection set at once, or on individual lines. There are pros and cons to each approach.

In the “whole selection set” approach, you would first apply an AutoConstrain across the entire selection set of elements you wish to clean up. However, if you select more than 100 items, AutoCAD will shoot back a rather ominous message as seen here.

The other problem is that it is slow and very expensive, processor-wise. With only 300 entities selected, it sent the fans in my PC running on high and repeatedly crashed AutoCAD. If you leave the AutoConstrain settings to their defaults, with all constraint types enabled, it will perform worse on smaller datasets.

The other way to handle this is to AutoConstrain only line objects as you process them. This has the advantage of being much more efficient, because with only a single line to work on, the AutoConstrain command can only apply a Horizontal or Vertical constraint, and it is very quick.
However, the problem is that if you AutoConstrain the line before you fix its endpoints, you may end up with connecting linework that doesn’t meet it anymore, because the endpoints all now clean up in different directions.

Our program should take this into account and offer three options: (a) AutoConstrain the entire selection set before the cleanup process begins, and only if the number of items is below 100; (b) AutoConstrain only each individual line as it is independently processed; or (c) not apply any AutoConstraints at all. There are other options to explore as well, such as only processing 50 items at a time, or prefiltering a selection set to only work on line objects, etc.

The other thing to do is to immediately delete the constraints after you apply them, so that AutoCAD doesn’t have to process them repeatedly as geometry is worked on.

Determining the Proper Values to Fix

Lines are, by definition, defined by their start and end points, so in the line example above there were two coordinate points to fix. Other entity types, such as arcs, circles, blocks, linear dimensions, and polylines are defined by different coordinate points.

An Arc is defined by a center point, a radius, a start point and an end point. Blocks are typically defined about an Insertion Point, have X/Y/Z scales, and have an angle. Polylines are defined by their vertices. Linear dimensions are defined by their definitions points which, in conjunction with its internal round off value, determine the value displayed.

In addition, there may be entity types we do not need to fix. Text, Splines, Regions, and 3D solids, for example, typically do not need to be cleaned up in 2D architectural drawings.

We also need to examine what coordinate points are important and which are not. For Arcs, we definitely want to fix the center point and the radius, because these two values often determine
how they are laid out in plan. However, the start and end points are by definition cannot be fixed to a new coordinate point without throwing off the radius or center point in the process.

For Blocks, we typically want to clean up the insertion point, because it usually is some intelligently placed point on the object, such as the center of a door. We probably don’t need to worry about the scale, and for polylines we probably don’t care about the bulge factor.

For lines, we have to consider the context. We want to clean up close-to-horizontal and vertical lines, but you don’t want to affect two lines that represent a 45-degree angled wall, because by doing so you will knock out the resulting angle of the lines. They will also not be parallel to each other anymore.

So, any solution is going to need the following:

1. A list of all of the entity types we wish to work on;
2. Their important coordinate points;
3. The precision that we want to use to clean up the geometry

We then selectively pick the entities to clean up. At first glance it may be troublesome to stay away from selecting angled linework or other entities which would become seriously wacked if moved to precise coordinates, but our programmed solution should do some of this analysis for us.

However, as we will see later, we can leverage User Coordinate Systems to clean up floor plans that look like this:
Coding Your Solution

The Importance of Algorithms

To follow through with creating a VLISP application to fix our geometry based on these somewhat simple rules, we will first break the solution down into smaller component parts, and come up with a programmatic algorithm which will determine the shape of our code.

An algorithm is, according to Wikipedia, “an effective method expressed as a finite list of well-defined instructions for calculating a function…In simple words an algorithm is a step-by-step procedure for calculations.”

Creating this step by step procedure is one of the most important skills you can develop. As I was taught back in programmer school, if you cannot clearly think through a problem first, you cannot hope to code it later. Flailing around in VisualLISP hoping you get working code isn’t very productive; you need to be able to predict success.

For example, take the lyrics of Frank Zappa’s song “Tiny Sick Tears”, which (among other things) maps out an algorithm for our sad teenage protagonist to get a snack from the kitchen:

1. Out of your bedroom
2. Out into the hall
3. Down to the living room
4. To the living room
5. To the kitchen
6. To the cookie jar where you wanna get your cookies
7. And you take the top off the cookie jar
8. And you stick your tiny sick hand in the cookie jar
9. And you reach around in the cookie jar
10. To find a raisin cookie (a spongy one with the little plump raisins)
11. Squeeze the raisin on the cookie
12. Pull the cookie out of the jar
13. Stuff the raisin into your eating hole
14. Push it all the way in your eating hole
15. Now make your eating hole wrap itself around the tiny sick cookie
16. Scarf the cookie
17. Put the lid back on the jar
18. Go over to the ice box
19. Open the ice box
20. Pull out the box of milk
21. Open the box of milk into a triangular beak
22. Pull the little triangular beak up to your drinking hole
23. Pour the white fluid from the drinking box into your hole
24. Close the beak
25. Reinsert the box into the ice box
26. Close the box door
27. Walk out of the kitchen
28. Through the living room
29. Back up the stairs
30. Past your sister’s room
31. Past your brother’s room...
Once we have logically analyzed the problem, we come up with this programmatic algorithm:

1. Have the user decide on the precision value;
2. Select the entities to clean up;
3. (Optional): Autoconstrain the selection set;
4. For each selected entity:
   a. Determine its entity type (Line, Arc, Circle, etc.);
   b. Get the important values to work on (start/endpoints, centers, radii);
   c. Determine any exceptions. For example, if it is a line:
      i. (Optional): AutoConstrain the individual line (if Step 3 is not selected) to the horizontal or vertical in the current UCS, if it within 1 degree tolderance.
      ii. Only fix lines that are orthogonal to the current UCS
         a) Get the angle of the line and see if it is orthogonal to the current UCS;
         b) If so, go to step d.
         c) If not, skip the entity entirely
   d. For the important values, round them off based on the desired precision;
      i. For coordinate points, round off the X/Y values.
      ii. For circles and arcs, also round off the radius

Beginning to Code

So now we can finally start writing some code. If you’ve been writing VisualLISP routines for a while you have probably developed a style, or way of coding that makes sense to you. I will present some of my style conventions which helps me keep my thoughts clear and focused on the code flow, rather than flailing around and ultimately getting lost.

First, start by creating a new, blank text file called FIXUP.LSP in your favorite text editor or the VLIDE environment. Put in a search path folder, or add a folder to your AutoCAD search path for customization.

My convention for structuring my source files is generally as follows:

- Title and header information in a large comment block;
- Support functions and global symbols;
- Main command function

When writing code you will necessarily develop conventions and habits. Two very important aspects of your standard coding practices are comments and style.

Programming with Style

Using Comments

Commenting your code thoroughly is one of the best things you can do to help yourself later in the debugging stage. Some think commenting wastes time or slows down your code, but neither is true. Comments are stripped out when the file is loaded and do not consume memory. Comments come in two forms:
• Single lines of comments are preceded by a single semicolon (;). Anything on the line after the semicolon is ignored.
• Multiline and inline comments are delimited by a semicolon-bar combination, like so:
  ;|--insert comment here ----|

Some people use variations in formatting comments, such as

;;; This is a short block-style multiline
;;; comment used before a block of code

Or you can use comments to create logical section breaks in the file, for example:

;;; ----------------------------
;;; Title of Section here

Headers
In all of my source files and individual functions, I provide a header that provides a record of its version number, description, history, and other information, which looks something like this:

;| ENTLIB.LSP
AppVersion: 2008.10.09
Author:
  Matt Stachoni
Description:
  Entity data and general object handling library
Version History:
  2008.10.09: Misc. update
;|

For functions, I create a header directly under the (defun) statement, like so:

(defun ent:GetXent (e / e3 ed etype)
  ;| Version: 2008.06.11
  Description:
  Get ename inside xref (one level deep) e = nentsel object. Sends up an alert if an attribute is selected (returns nil).
  Parameters:
  e - entity data list as returned by (nentsel) command
  Notes:
  2008.06.11: If a block inside of an Xref is selected, return the outermost block.
  Keywords: xref nentsel ename
  |
  (code here)
)

For each header I provide information such as the version number, description, and history of changes. For the version I just use a simple date format of YYYY.MM.DD. For functions, I include a Parameters section in the header that describes the parameters and their data types.

In my particular case, the formatting of my headers is fairly rigid because I have coded an external utility that reads a source LSP file, parses each (defun () statement and the associated following header block, and dumps the data into an Excel file that can be further processed into
a Windows CHM help file. This allows me to create a complete set of documentation for all of my various libraries and command utilities. I use the Keywords section to provide a list of search terms for the help file. Of course, that’s just me. There’s no rule that states you need to establish such a convention, but it sure can help over time.

**Indentation Conventions**

Indenting code is another formatting convention which is extremely important in LISP. All VisualLISP statements are nested inside of parentheses, so logically indenting these layers of code is an easy way to track parenthetical matching vertically, and make sure you avoid the dreaded “error: malformed list on input” message as much as possible.

In my code I indent each block by 2 spaces. I have set up my preferred text editor, TextPad ([www.textpad.com](http://www.textpad.com)), to use 2 spaces for tabs, and to not change the tabs to spaces. That makes it easy to add and subtract tabs when correcting code. You can also highlight blocks of text and use CTRL+Tab to indent the entire block to the right, and SHIFT+TAB to move it to the left.

I also ensure I end a block of code with the final parentheses on its own line, so that I can visually line up the start of the block with the end. Any mismatched parentheses are much easier to spot.

The following code illustrates the usefulness of a logical indentation convention:

```lisp
(defun ent:GetDxfList (c ed)
  ;; Version: 2008.02.11
  ;; Description:
  ;; Returns multiple occurrences of DXF code in elist (LWPOLYLINES)
  ;; Parameters:
  ;;  C - DXF code as Integer
  ;;  Ed - Entity data list
  ;; Notes:
  ;;  2008.02.11: Added as part of general updates to entlib.lsp
  ;; Keywords: DXF elist
  ;;
  ;; (if (assoc c ed)
  ;;  (apply 'append
  ;;    (mapcar
  ;;      '(lambda (x)
  ;;        (if (eq c (car x))
  ;;          (list (cdr x))
  ;;          ed)
  ;;    ed)
  ;;
  ;; Indentation is particularly helpful when writing long conditional (cond) statements, because you need to see each block of ((test condition)(response statements)) clearly. In the function on the next page, the conditional tests and the resultant code to be executed if the test evaluates to T are easily identified and visually packaged in groups:
```
(defun ent:GetPVerts (e / ed vl)
;;; Version: 2007.03.21
Description:
  Return list of heavy/lightweight polyline vertex edata
Parameters:
  e - ename or VLA-Object
Notes:
  2007.03.21: Rewritten using ActiveX VLA functions, recursive to handle
  enames, edata lists or VLA objects
Keywords: polyline pline vertex
)
(cond ( (= 'ENAME (setq typ (type e)))
  (ent:GetPverts (vlax-ename->vla-object e))
)
( (= 'LIST typ)
  (ent:GetPVerts (vlax-ename->vla-object (cdr (assoc -1 e))))
)
( (and (= 'VLA-OBJECT typ) ;; pline entity type test
  (setq n (vla-get-ObjectName e))
  (or (= "AcDbPolyline" n)
      (= "AcDb2dPolyline" n)
  )
  (ent:vla-variant->coordlist (vla-get-coordinates e) 2)
)
(T (prompt "\n(ent:GetPVerts) error: Invalid object as argument."
  (print e)
))
)

Function Naming Standards
You may have noticed that the above utility functions all start with ent: in their name. This is a
convention I standardized on as I developed individual libraries of similar functions. In this case,
I have an ENTLIB.LSP function library file that contains functions for processing entity data and
general object handling, and each function in the library is named ent:Something.

There are advantages to writing separate library files. When I am writing a program or function
that requires entity data access, I first write a (load "entlib") statement at the top which loads the
entity handling library so the rest of the code has access to its functions.

This does two things: it makes the codebase much more modular (I’m not writing the same code
in multiple program source files), and allows me to demand load things. If I’m using AutoCAD for
viewing or plotting, and do not need any program or utility that needs specific data access, the
ENTLIB.LSP library is never loaded. This keeps the amount of loaded code in AutoCAD’s
memory footprint to a minimum.

Other libraries are for file handling (where all functions start with (file:<somename>), xref
handling, string handling, and so on. Currently I have about 20 separate library files that handle
utility chores for well over 50 additional LSP files consisting of thousands of lines of code.
In this class we are writing one file called FIXUP.LSP which will contain all of the code necessary to run the program, so there are no dependencies. All functions will start with `f:`.

**Programming Efficiently**

Developing a programming style is a side effect of your programming experience. As you continue to code you naturally continue to develop better habits. As part of this you should strive for leaner, more efficient coding practices.

For example, I see stuff like this a lot in the online forums:

```lisp
(if (= nil a) (do_something_to a)...)  
(if (not (= a nil)) (do_something_to a)...) 
```

This is a long-winded way of saying

```lisp
(if a (do_something_to a)...) 
```

When coding smaller functions, evaluate whether the overhead of a function is worth the effort. 

Take this classic function:

```lisp
(defun dxf (code elist) 
  (cdr (assoc code elist)) 
)
```

Which returns the value associated with a DXF code in an entity data list. Is it worth it to write this as a function? In your application, you would be saving the effort to write `(cdr (assoc c l))` with `(dxf c l)`, which is hardly any simpler and causes VisualLISP more work in the process.

Always think through the code to eliminate redundant steps and minimize the amount of work the VisualLISP interpreter has to do, particularly when working with AutoCAD’s object model using the vla-ActiveX functions. For example, if you want to get the Files object from the AutoCAD Preferences object, you can do this:

```lisp
(setq files (vla-get-Files (vla-get-preferences (vlax-get-acad-object)))) 
```

Which retrieves the objects implicitly inside the single statement. That looks clean, but if you need to get something else from the Preferences object, you need to get the AutoCAD object again; this is more expensive from a processing and memory standpoint. The better option is to get these things once and store them as symbols and be able to access them later:

```lisp
(setq AcadObj (vlax-get-AutoCAD-object) 
  Prefs (vla-get-Preferences AcadObj) 
  Files (vla-get-Files Prefs) 
)
```

Hopefully, other examples of efficient coding practices will be seen later in this handout and in the source files available in the downloadable dataset for this class.
The Program Header and Initial Coding

When I start a new file, I generally do several things to block it out. I first write out the header, using a standardized format. I then write "bookends" which is comprised of a pair of (prompt) statements saying it's loading/loaded, and a final (princ) statement to load quietly.

`;|  FIXUP.LSP
   AppVersion: 2011.10.01

   Author:
   Matt Stachoni

   Description:
   Fixup fixes line endpoints, circle centers and diameters, dimensions and block insertion points to the nearest user-defined precision coordinate point.

   History:
   2011.10.01: AU 2011 version

   Disclaimer:
   Permission to use, alter, copy and freely distribute this routine is permitted ONLY if the Author's name and copyright header is included. Author makes no guarantees on the long-term reliability of this program, and assumes no liability if it totally makes a mess of your drawing.
|;

(prompt "\nLoading FIXUP (c)2011 by Matt Stachoni")
;; (code goes here)
(prompt "\nApplication loaded. Type 'FIXUP' to run.")
(princ)

The next thing to do is carve out a small section header for support functions, initialize ActiveX functions with (vl-load-com), and write a small utility "load me" type command function between the bookends that helps with the debugging stage. All it does is reloads the file quickly, which is useful when testing code. With short, obvious functions I generally do not provide a header.

(prompt "\nLoading FIXUP (c)2011 by Matt Stachoni")
;;Start bookend
;;;  -----------------------------------------------
;;;  Support Functions
;
;; Initialize ActiveX functions
(vl-load-com)

(defun c:lfu ()
   (load "fixup.lsp")
   (princ)
)
(prompt "\nApplication loaded. Type 'FIXUP' to run.")
;;Ending bookend
(princ)
The Main Command Function

At this point we can block out the code for our algorithm as the main part of the program. When mapping out functions not yet written, you can write in some pseudo-code which can be used as a stand-in. I’ll often fill those “proto” functions with (prompt) statements to print out values. The following code will go after the Support Functions area:

```
;; Initialize global default values for precision
(if (not *prec*) (setq *prec* 0.125))

;;; Main program
;;; ---------------------------------------------------------------------

(defun c:fixup (/ tmp ss i e)
  ;; Set CMDECHO to 0 and begin of Undo
  (setvar "cmdecho" 0)
  (command "_.undo" "begin")

  ;; Get the precision value from the user, providing defaults
  ;; Save the value as a global variable for the next time its run
  (if
    (setq tmp
      (getreal (strcat \nPrecision of vertices [" (rtos *prec*) ] : ")
    )
    (setq *prec* tmp)
  )

  ;; Select objects to process and initialize counter for the while loop
  (setq ss (ssget) i 0)
  (while (setq e (ssname ss i))
    ;; Process the entity to the specified precision
    (f:RndObject e *prec*) ;; Process the entity to the specified precision
    (setq i (1+ i))
  )

  (prompt \nFIXUP operation completed.")
  (command "_.undo" "end")
  (princ)
)
```

Nothing too out of the ordinary here. We initialize some global variables to provide the user with some default values. Whatever the user enters for precision will be used and become the new default when the program is run again. The (if (setq temp (getreal)…) statement allows the user to hit [enter] to accept the defaults; if nothing is entered, temp is nil, and *prec* doesn’t change.

We then created the basic logic for the process using a WHILE loop: create a selection set, set a counter to 0, and while there are elements in the selection set, grab each one and process it, using the F:RndObject () function (which hasn’t been written yet) and increment the counter.

Style Note: I use asterisks in the symbol name to designate global symbols. I also ensure I localize all symbols used in functions. However, when first writing complex code, you may opt to leave these out so they become global variables to see them while testing. After you run the function you can check their status at the command line using ‘!', e.g.

```
(defun c:foo () ;/ ss i e num)
  ;;; code..)
)
Command: !num
5.105
```
Creating the Rounding Functions

Our next step is to understand some basics of how the functionality of how we process the entity is going to work. The first thing we need to do is write a general rounding function which simply rounds a real number to the specified precision. Everything after will build on that function.

It will take two arguments (also called parameters): the number to round, and the rounding precision. The algorithm of the rounding function logic is this:

1. Take an original number and divide it by the desired precision (let’s call that result X).
2. If the original number was positive, add 0.5 to X. If negative, subtract 0.5 from X.
3. Fix this result down to the nearest whole number, so e.g. 41.823 → 41.0
4. Multiply this whole number by the precision to get the final result

In code, it’s really quite simple:

```lisp
(defun f:rnd (num prec / pos flag fixdn rem1)
  ;; Version: 2011.11.12
  ;; Description: Round off a real number to a specified precision
  ;; Ex: (f:rnd 1.127 0.125) ==> 1.125
  ;; Parameters:
  ;;   num - The number to round off as Real
  ;;   prec - The precision as Real
  ;; Notes
  ;;   2011.11.12: AU 2011
  ;; Keywords: Rounding Round
  |;
  (* prec
   (if (minusp num)
     (fix (- (/ num prec) 0.5))
     (fix (+ (/ num prec) 0.5)))
  )
)
```

In other words, (f:rnd 5.105 0.125) → 5.125.

Rounding a Point List or a Value: UCS Considerations

Now we have to apply this rounding function to a coordinate point list. However, we need to take into account how to deal with the current UCS. Coordinate points gathered from entity data are always given in the World Coordinate System, and we want to round object coordinates relative to the current User Coordinate System (UCS). We do this so that we can leverage UCSs later when working with complex floor plans with angled wings.

For each coordinate point, we have to translate the coordinate to the UCS, apply the rounding function to each element in the list, and then translate that back to the WCS. Then it can be applied back to the entity object which updates the geometry.

When we start dealing with points from VLA-objects, we will see that some of them are provided as 2D coordinates and some as 3D coordinates. Because translating a 2D point to another UCS
and back makes it a 3D point, our code must take this into account as well, and return the same length of coordinate point to the calling function.

(defun f:rndpt (pt prec / np)
  ;| Version: 2011.10.01
  Description:
  Round each of the values in a coordinate point list to the nearest precision
  in the current UCS.
  Parameters:
  pt - 2D or 3D coordinate point list to be rounded
  prec - Precision as Real
  Returns:
  2D or 3D coordinate in WCS rounded off to precision, based on original
  coordinate list
  Notes:
  2011.10.01: AU 2011
  Keywords: Rounding Round
  |
  (cond
   (and pt (listp pt)) ;if the argument is a list
    (setq np
      (trans
        (mapcar
          '(lambda (x)
            (f:rnd x prec) ; round off the UCS value
          )
          (trans pt 0 1) ; translate pt to current UCS
        )
        1 0            ; translate back result to WCS
      )
    )
    (if (= 2 (length pt)) ; account for 2D points
      (setq np (list (car np)(cadr np)))
    ) ; end 1st condition statement
  ) ; arg is a single value
  (setq np (f:rnd pt prec))
  )
  ; end cond
  np ; return rounded value
)

We first test to see if the point argument is a list; if so, process the list. The next block of code features a (mapcar `'(lambda(x)...) statement, which is a typical programming cliché, that is, this form is very common in general LISP coding and can be applied in any one of a million situations. You will see it many times in this class.

The (lambda (x)...) function is an anonymous function; that is, it is a function written right in the code without the overhead of a formal (defun) statement. It’s most often used inside of (mapcar) expressions, as a function is always the first argument to (mapcar).

The (mapcar) function applies a function with the list or lists supplied as arguments to the function. The formal format is (mapcar `function list1....listn). The number of lists must match the number of arguments to the function. Typically, it is used to apply a single function to the members of a single list and return the modified list.
In this case, it essentially does this: For each number in the coordinate list, round it off to the desired precision, and return the modified coordinate list. The list provided to the ‘(lambda (x) ) function is a translated point from the WCS to the current UCS, so the roundoff function works on UCS coordinates. The modified coordinate list result is in turn translated back to the WCS and is the list returned by the function.

Note that if the pt argument is a single value, such as a radius, then the function simply rounds off the number and returns that.

**Identifying the Object Properties to Fix**

As we work on different entity types in the drawing, we are going to work with a combination of the ActiveX API to access the AutoCAD Object Model as well as old-school DXF entity data access methods. The purpose of using ActiveX for modifying objects is primarily pure speed. The ActiveX API is much faster in accessing and modifying object data than working with DXF codes and updating entity list data, because you access the objects in a much more direct manner. In other words, (entget) is slow, as is (entmod). To work with the ActiveX API in VisualLISP we use the VL-, VLA- and VLAX- functions.

However, sometimes you have to step back to DXF-style methods using (entget) and (entmod), because the ActiveX API does not provide access to the particular properties we need. For this program we are going to work on the following entity types and their properties:

<table>
<thead>
<tr>
<th>Entity Type</th>
<th>VLA Object Name</th>
<th>DXF Codes</th>
<th>ActiveX Property Names</th>
<th>Value Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>LINE</td>
<td>AcDbLine</td>
<td>10</td>
<td>StartPoint</td>
<td>3D Points. Apply a horizontal and vertical AutoConstraint</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>EndPoint</td>
<td></td>
</tr>
<tr>
<td>LWPOLYLINE</td>
<td>AcDbPolyline</td>
<td>10</td>
<td>Coordinates</td>
<td>Array of 2D Points</td>
</tr>
<tr>
<td>CIRCLE</td>
<td>AcDbCircle</td>
<td>10</td>
<td>Center</td>
<td>3D Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>Radius</td>
<td>Real Value</td>
</tr>
<tr>
<td>ELLIPSE</td>
<td>AcDbEllipse</td>
<td>10</td>
<td>Center</td>
<td>3D Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11</td>
<td>MajorRadius</td>
<td>Real Value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>MinorRadius</td>
<td>Real Value</td>
</tr>
<tr>
<td>ARC</td>
<td>AcDbArc</td>
<td>10</td>
<td>Center</td>
<td>3D Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>40</td>
<td>Radius</td>
<td>Real Value</td>
</tr>
<tr>
<td>XLINE</td>
<td>AcDbXline</td>
<td>10</td>
<td>BasePoint</td>
<td>Real Value</td>
</tr>
<tr>
<td>SOLID</td>
<td>AcDbSolid</td>
<td>10 - 13</td>
<td>Coordinates</td>
<td>Array of 3D Points</td>
</tr>
<tr>
<td>INSERT</td>
<td>AcDbBlockReference</td>
<td>10</td>
<td>InsertionPoint</td>
<td>3D Point</td>
</tr>
<tr>
<td></td>
<td></td>
<td>50</td>
<td>Rotation</td>
<td>Real Value</td>
</tr>
<tr>
<td>MTEXT</td>
<td>AcDbMText</td>
<td>10</td>
<td>InsertionPoint</td>
<td>3D Point</td>
</tr>
<tr>
<td>TEXT</td>
<td>AcDbText</td>
<td>10</td>
<td>InsertionPoint</td>
<td>3D Point</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>AcDbRotatedDimension</td>
<td>10</td>
<td>DefinitionPoint</td>
<td>Dimensions do not expose definition points in ActiveX; we must use DXF codes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>DefinitionPoint</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>14</td>
<td>DefinitionPoint</td>
<td></td>
</tr>
</tbody>
</table>
Notice that coordinates array for polylines are 2D, while a 2D solid is an array of 3D points, and the others are single 3D points. That will become important as we look into variants and safearrays of points.

**Looping Through the Selection Set - Revisited**

If you recall, we had this code in the main command function to loop through the selection set:

```lisp
;; Select objects to fix and initialize counter for the while loop
(setq ss (ssget) i 0)
(while (setq e (ssname ss i))
  (f:RndObject e *prec*)
  (setq i (1+ i))
)
```

Whenever we set up a while loop to process a selection set, we always follow the same process: We initialize a counter at 0, get the selection set element for that counter, do something to the element, increment the counter, and repeat. Because we loop through selection sets so often in VLISP, it helps to create a function to do this for us instead of coding the same thing over and over again.

Similar to what we have seen with the `(mapcar `(lambda (x) ...))`, where we process a function across a list, we can do the same with entire selection sets using the `(apply)` function:

```lisp
(defun f:ssmap (fun ss / n)
  ;; Version: 2011.10.01
  Description:
  Process a function on each ename of a selection set.
  Also allows ss to be nil
  Parameters:
  fun - Function, usually of the form `(lambda (e)...)`
  ss - Selection set
  Notes:
  2011.10.01: AU 2011
  Keywords: ename map
  
  ;;
  (setq n -1)
  (if (= 'PICKSET (type ss))
    (repeat (fix (sslength ss))
      (apply fun (list (ssname ss (setq n (1+ n))))))
    )
  )
)
```

This essentially wraps up the `(while (setq e (ssname ss i)) ..)` loop into a function that applies a function to each entity name in the selection set. Wrapping the selection set processing loop itself into a function like this allows us to replace all that loop code in the main command function with an altogether more elegant weapon:

```lisp
(ent:ssmap '(lambda (e)
    (f:RndObject e *Prec*)
  )
  (ssget)
)
```
Starting to Build the (F:RndObject) Function

In building the (f:RndObject) function we will again code the basic algorithmic logic to create the structure, then flesh it out as we incrementally test and refine things. At this basic level, let's just concentrate on fixing up Lines.

(defun f:RndObject (e prec / Obj ObjName p1 p2)
  ;; Version: 2011.10.01
  Description:
  Round off entity points to a specified precision
  Temporarily autoconstrains line objects to horizontal and vertical
  Will not affect angled lines over 1° from horizontal/vertical
  Parameters:
  e - Entity name (ename) to process
  Prec - Precision as Real
  Notes:
  2011.10.01: AU 2011
  Keywords: Rounding Round
  ;; Get the VLA-Object and its name
  (setq Obj (vlax-get-object e)
        ObjName (vla-get-ObjectName Obj)
       )
  ;; Process the AcDbLine object, with any special behaviors and exceptions.
  (cond ( (= "AcDbLine" ObjName)
    (prompt (strcat "\nModifying : " ObjName))
    ;; Optionally Autoconstrain the line. Comment out for now.
    (command "_autoconstrain" e "")
    (command "_delconstraint" e "")
    (if (f:IsLineOrtho obj)
        (progn
          (prompt (strcat "\nModifying : " ObjName))
          (setq p1 (vlax-get-property obj 'Startpoint)
                p2 (vlax-get-property obj 'Endpoint)
               )
          (f:CleanUp P1 prec)
          (f:CleanUp P2 prec)
          (prompt "...Fixed.")
        )
        (prompt "\nLine is at an angle to the current UCS. Not fixed.")
      ) ; End If
    ) ; end cond
  ) ; end AcDbLine
) ; end f:RndObject

At this point we’ll table the AutoConstrain option until later, as we run some tests and find out which methods are most efficient. Right now we want to just make sure the rounding works.

Here again we write in some stand-in functions for seeing if the line is orthogonal to the UCS using (f:IsLineOrtho), and if so, clean up the start and end points using (f:CleanUp).
Note that we may ultimately write in the code to clean up lines explicitly in this function instead of writing a separate function; regardless, this is a good way to functionally break down the code into smaller modules to execute and test later.

At this point I would write these as blank stand-in functions, which do nothing except maybe print an informative message that prints the point data passed to it.

```
(defun f:CleanUp (pt prec)
  (prompt ("\n\tProcessing point ") (princ pt))
  (princ)
)
```

Now if we run FIXUP on some lines in the drawing, we get this:

```
Command: fixup
Precision of vertices [1/8] :
Select objects: 2 found
Select objects:

Modifying : AcDbLine
  Processing point #<variant 8197 ...
  Processing point #<variant 8197 ...
Modifying : AcDbLine
  Processing point #<variant 8197 ...
  Processing point #<variant 8197 ...
FIXUP operation completed.
```

I strongly recommend development in this manner, where you code only so far, then test the code with dummy stand-in functions that simply report progress. Here I can tell I immediately what entities were selected and that each is picking up the properties from the master list correctly.

**Working with and Rounding Values of VLA-Objects**

The (f:RndObject) function will ultimately take a VLA-object and the precision, and round off the value of the important points and modify the VLA-object.

As indicated in the table on page 26, the important points it will modify are either a single 3D coordinate point for start/endpoints of lines, insertion points of blocks and text, centers of circles and arcs, etc.; a list of 2D points for Lightweight Polylines; a list of 3D points for 2D solids; or a standalone value for the radius of a circle or arc, or rotation angle of a block. For these values, we will modify them using the ActiveX API with VL- functions. For linear dimensions, we will need to alter DXF codes 10, 13, and 14 using the old-school functions (entget), (entmod), and (entupd).

In addition to this complexity, when working with 2D/3D coordinate points of VLA-objects, we need to account for the fact that the ActiveX API doesn’t expose these values as lists. Rather, they expose this coordinate data as a **Variant** data type, which is essentially a catch-all universal data type that reserves enough system resources to contain any other data type, such as numbers, strings, dates, or whatever. That’s what the #<variant 8197 ...> business is above.
In normal AutoLISP you never really have to worry about specifying data types for symbols as you do in other languages. The tradeoff is that, while coding is fast, the processing involved is not. When you work without explicitly defining data types, the AutoLISP language has to allocate the maximum amount required for a symbol as a “best guess” more than anything. As a result, many operations in VisualLISP are, from a computing standpoint, pretty expensive and take up much more memory than would happen if you were using a strongly typed language like C++ or Java, or even a weakly typed language like VB.net.

This is one of the primary reasons why entity data modification is so much faster than using DXF-style entity access, because the (entget), (entmod), and (entupd) functions are so slow. When processing thousands of entities at a time, speed in any one single function becomes critical, which is why we are taking the time to code it using ActiveX functions.

But the problem is that with the ActiveX API, we have this weird Variant data type to deal with. You can see it in action with this code:

```
Command: (vla-get-startpoint (vlax-ename->vla-object (ssname (ssget) 0)))
Select objects: [Select a line]
Select objects: #<variant 8197 ...>
```

The reasoning behind this highlights the differences between ActiveX programming and AutoLISP. ActiveX is an API that specializes in accessing the Object Model of an application, open documents and the data within those documents, and communicating to other applications. LISP, on the other hand, is fantastic at processing list data, and Autodesk adopted this language early on to create the AutoLISP API as a means of interacting with drawing element data as a series of lists. The various AutoLISP functions were written specifically to express internal DWG entity data in a form that can be processed quickly, and that form was a list. Before AutoCAD became an object-oriented program, this was a very valid was of expressing data, but you could not access properties of the application or open documents.

The ActiveX API stores data as Variants which allows it to be passed between AutoCAD and other applications, such as Microsoft Excel. For use in VisualLISP we can read, create, and modify Variant data types pretty handily; we just have to know some ways of doing this.

Whereas in normal AutoLISP we used functions such as (list), (car), (nth), (mapcar), (apply), and so on to handle list processing, we use a different set of functions for handling variants of “list” data types, called Safearrays. In other languages, arrays are like lists, in that they represent a series of data values in a single construct that can be incrementally accessed and modified. Arrays often contain nested arrays as well. Safearrays are called “safe” because, as part of their data structure, they include their length which is thus fixed, providing high level protected access to an array. This prevents errors when you attempt to assign or fetch members beyond the length of the array.

Most people get scared at having to deal with variants and safearrays; indeed, for what we do, it actually does simplify the programming quite a bit to use DXF codes, but this safearray stuff is fun. We'll press on.
For this class we are only going to worry about two things in regards to variants and safearrays: how to convert Variants into LISP values, and how to convert them back into variants that can be updated in VLA-objects.

Converting a Variant into a List involves first seeing what data type it is, and if it is a safearray, converting it to a safearray, then converting that safearray into a List. To access the value contained in a variant symbol, we use the `(vlax-variant-value)` function. When the variant data is of a safearray data type, we can then turn that Safearray symbol into a List using the `(vlax-safearray->list)` function.

We can use this function to get a proper LISP value from any symbol, whether it be of type variant or not, and whether that variant’s value is a safearray data type or not:

```lisp
(defun f:LispValue (v)
  ;;|Version: 2011.10.01
  ;;|Description:
  ;;|Recursively dig through a Variant or Safearray value to produce a LISP value (list)
  ;;|Parameters:
  ;;|v – Variant, Safearray, or simple value
  ;;|Notes:
  ;;|2011.10.01: AU 2011. Thanks to Vladimir Nesterovsky 2002 and Acadx.com
  ;;|Keywords: variant conversion point
  ;;|
  (cond
   ( (= (type v) 'variant)
     (f:LispValue (vlax-variant-value v))
   )
   ( (and (= (type v) 'safearray)
         (safearray-value v)
     )
     (mapcar 'f:LispValue (vlax-safearray->list v))
   )
   ( T v)
  )
)
```

This code highlights several things that are important:

1. This function is recursive; that is, it calls itself providing the values it returns. On first run it determines the data type of the `v` argument; if it is a variant, get the value using `(vlax-variant-value)` and send that back to the `f:LispValue` function for further processing.

2. If the value returned from that operation is a safearray AND the `(safearray-value)` function returns a value, use `(vlax-safearray->list)` to return the list.

3. Note that the `(safearray-value)` function is undocumented (I think it’s a holdover from when VisualLISP was called VitalLISP). The reason to use this function here is that sometimes the safearray value may be nil or have an element that’s nil, which could generate an error. Indeed, you could shorten this block to simply

```
( (= (type v) 'safearray)
  (mapcar 'f:LispValue (safearray-value v))
)
```

4. **ALWAYS** give credit where due when using someone else’s code.
Armed with this function, let's plug it into the (f:CleanUp) function and see what we get.

```lisp
(defun f:CleanUp (pt prec)
  (prompt ("\n\tProcessing ")(princ (f:LispValue pt))(princ)
)

Command: fixup
Precision of vertices [1/8] :
Select objects: Specify opposite corner: 9 found

Modifying : AcDbLine
  Processing (10.875 3.0 0.0)
  Processing (20.232 3.0 0)
Modifying : AcDbLine
  Processing (12.125 6.0 0.0)
  Processing (15.432 6.0 0)

FIXUP operation completed.

Utility Functions for Working with Vla-Objects
As a quick aside, it's helpful to write small command utilities to help you code faster when working with the ActiveX API. One of these is (c:get), which is used to get the vla-object of a selected entity:

```lisp
(defun c:get (/ e)
  (if (setq e (car (entsel)))
    (vlax-ename->vla-object e)
  )
)

To examine the properties and methods of a selected objects or symbols, use this:

```lisp
(defun c:dump (/ e sym)
  ;|Version: 2008.08.21
  ;|Description:  
  ;|   Programming tool: Utility DUMP function to list properties and methods.
  ;|Notes:  
  ;|   2008.08.21 : Added ability to specify symbol
  ;|Keywords: Programming Tools VLA Object
  ;|
  (if (not *sym*)
    (setq *sym* "")
  )
  (initget 128)
  (cond ( (setq e (entsel "\nSelect object or <> to enter symbol name: "))
    (vlax-dump-object (vlax-ename->vla-object (car e)) T)
  )
    ( = ""
      (setq sym
        (getstring
          (strcat "\nSymbol name to dump " (strcase *sym*) ":")
        )
      )
    )
  )
  (= 'VLA-OBJECT (type (eval (read *sym*)))))
```
Converting Safearrays for Coordinates

The problem we run into with Variants is that they store different kinds of data depending on the object type involved. When you use \texttt{(f:LispValue)} on the ‘Coordinates property, used for both Polylines and Solids, it returns a long stream of numbers, but they mean different things. For example, when entered at the command line:

\begin{verbatim}
Command: (setq cl (f:LispValue (vlax-get-property (c:get) ‘coordinates)))
Select object: <select a polyline>
(7.5 -0.125 10.125 -0.125 10.125 1.25 7.5 1.25)
\end{verbatim}

For Lightweight Polylines, the ‘coordinates property is a variant array of 2D points (note there is no Z=0 coordinate; this is governed by the Elevation property). Note that there are 8 elements to the list, two for each point on a polyline rectangle.

For all other objects that we are going to work with, this variant is an array of 3D points, so we have to account for these differences by object type.

To handle this, we need another function that will convert a safearray of 2D and 3D points properly into a list:

\begin{verbatim}
(defun f:vla-variant->coordlist (var n / f:2Dlist f:3Dlist r)
 ;|Version: 2011.02.12
 ;|Description:
 ;|Given a variant safearray array of doubles, return the list of 2D (n=2)
 ;|or 3D (n=3) WCS points in order. If the variant is a single point list,
 ;|return the single list (not a list of point lists)
 ;|Parameters:
 ;|var - Variant array of 2D or 3D doubles
 ;|n - 2 or 3, depending on whether the variant is from a LWPolylines or
 ;|other object type.
 ;|Returns: 2D or 3D list of points in the WCS
 ;|Thanks to: Doug Broad (and others on the Autodesk and Swamp forums)
 ;|Keywords: Entity Data Library Function

(if n
 (f:2Dlist)
 (f:3Dlist))
\end{verbatim}
(cons (list (car l) (cadr l)) (f:2Dlist (cddr l)))
)
(defun f:3Dlist (l)
  (if l
    (cons (list (car l) (cadr l) (caddr l)) (f:3Dlist (cdddr l)))
  )
  (if (= n 2)
    (setq r (f:2Dlist (f:LispValue var)))
    (setq r (f:3Dlist (f:LispValue var)))
  )
  (if (= 1 (length r))
    (car r)
    r
  )
)

This example shows a rarity (for me): defining a function inside of another function. Here, it is done so we can recursively propagate through the list and create a list of 2D or 3D point lists.

Now let’s revisit that rectangle with this:

Command: (f:vla-variant->coordlist (vlax-get-property (c:get) 'coordinates) 2)
Select object: ((7.5 -0.125) (10.125 -0.125) (10.125 1.25) (7.5 1.25))

For single 3D points (line endpoints, insertion points, circle and arc centers, etc.), as well as 2D solids, you want to use 3 as the last argument, e.g.

Command: (setq sl (f:vla-variant->coordlist(vlax-get-property(c:get)'coordinates) 3))
Select object: <select a 2D solid>
((7.81616 -1.42423 0.0) (9.56923 -1.53778 0.0)(8.33558 -2.55977 0.0) (9.73155 -2.54355 0.0))

Turning Lists back into Safearrays and Variants
Before we can get onto the rounding of our values, we need to understand how to get a modified list of points back into a safearray, back into a variant, and then can then be put back to the vla-object, updating its geometry.

This is done with the (f:vla-Coordlist->Variant) function, which takes as its single argument the (modified, rounded) point list, using functions like (vlax-make-safearray), (vlax-safearray-fill), and other generally nasty looking things.

We are not going to discuss the particulars of those functions; the AutoLISP Developer’s Guide in the online help is there and does a somewhat adequate job. Better still, mine the Autodesk Customization discussion forum at http://forums.autodesk.com/t5/Visual-LISP-AutoLISP-and-General/bd-p/130 which is stock full of many years’ worth of valuable programming knowledge on the subject. Personally, I think the best way to learn is to see something written and break it down to learn how it works. Dig it:
(defun f:vla-Coordlist->Variant (ptsList / ptsArray sArray)
  ;| Version: 2011.10.01
  Description:
  Create variant array of doubles from point list, reverse of
  (f:vla-variant->coordlist)
  Example:
  (setq v (f:vla-Coordlist->Variant '((0 0 0) (12 3 0) (121.34 343 0))))
  ==> variant
  (f:LispValue v) ==> '(0 0 0 12 3 0 121.34 343 0)
  Parameters:
  ptsList - a list of point lists
  Returns:
  A Variant of a Safearray type.
  Keywords: Entity Data Library Function
  ; Create a single list of coordinate numbers
  (if (= 'LIST (type (car ptsList)))
    (setq ptsList (apply 'append ptsList))
  )
  ; Make the safearray of the correct dimension (size)
  (setq ptsArray
    (vlax-make-safearray
      vlax-vbDouble ; element type
      ;'(0 . #) array dimension = length of points list
      (cons 0 (1- (length ptsList)))
    )
  )
  ; fill it with the value of the points list
  (setq sArray (vlax-safearray-fill ptsArray ptsList))
  ; turn it into a variant
  (vlax-make-variant sArray)
)

So, if we take the point list from the solid we picked previously:
(f:vla-coordlist->variant sl) => #<variant 8197 ...>

Now for the Rounding

Now that we have the object coordinates sorted out, and we can turn those values back into variants, we can go ahead and round off the values and modify the objects in one swoop. We'll take care of dimensions and DXF codes in a bit.

Previously, we coded out a stub function, (f:CleanUp), to be used to round off the values. However, we can process all of the data types we need to work on right in the (f:RndObject) function instead, and negate the need to come up with a complex function that could handle all of the different object types and data types. Therefore, we are going to wrap up this portion of the code in the (cond) statement we started in the (f:RndObject) function:
For handling the line object, which is by far the most common object type you will work on, the full code looks like this (the new code is bold):

```lisp
(cond ( (= "AcDbLine" ObjName)
   ;; Autoconstrain the line, then delete the constraint
   (command "_autoconstrain" e "")
   (command "_delconstraint" e "")
   (if (f:IsLineOrtho obj)
     (progn
       (prompt (strcat "\nModifying : " objname))
       (setq p1 (vlax-get-property obj 'Startpoint)
       p2 (vlax-get-property obj 'Endpoint)
       )
       (vlax-put-property
        obj
        'Startpoint
        (f:vla-coordlist->variant
         (f:RndPt
          (f:vla-variant->coordlist p1 3)
         prec
         )
        )
       )
       (vlax-put-property
        obj
        'Endpoint
        (f:vla-coordlist->variant
         (f:RndPt
          (f:vla-variant->coordlist p2 3)
         prec
         )
        )
       )
      )
      (prompt "...fixed.")
     ) ; end progn
   )
   (prompt "\nLine is at an angle to the current UCS. Not fixed.")
   ) ; end if
  ) ; end AcDbLine

Testing Orthogonality of Lines
When we select a line to clean up, we first need to test whether or not it is orthogonal to the current UCS.

To handle this, we need some utility functions to get the current angle of the current UCS to the WCS, then get the angle of the Line object, and see if its angle is orthogonal to the UCS angle.

The easiest way to get the current UCS angle to the WCS is to calculate the angle from WCS 0,0,0 to the value given by the UCSXDIR system variable:

```lisp
(defun f:GetUcsAngle ()
  (angle '(0 0 0) (getvar "ucsxdir"))
)
```
Then use this to see if a line object is orthogonal to that angle.

(defun f:IsLineOrtho (LineObj / a)
  ;| Version: 2010.10.01
  Description:
  Returns T if the angle of any AcDbLine object is orthogonal,
  i.e., 0, 90, 180, 270° to the current UCS angle
  Parameters:
  LineObj - AcDbLine VLA-object
  Notes:
  2011.10.01: AU 2011
  Keywords: Rounding Round
  |
  ; Set this as a global variable here the first time around
  (if (not *UCSAngle*)
    (setq *UCSAngle* (f:GetUCSAngle))
  )

  ;; Subtract angle of the current UCS from the angle of the line,
  ;; then divide this difference angle by pi for later comparison
  (setq a
    (/ (abs (- (vla-get-Angle LineObj) *UCSAngle*)) pi)
  )

  ;; Due to AutoCAD's high level of precision, you may generate very small
  ;; numbers in the angle ratio. Round it off to the nearest 0.00000001
  (setq a (f:Rnd a 0.00000001))

  ;; Test angle/pi ratio; if points are ortho, return T
  (or (= 0.0 a) (= 0.5 a) (= 1.0 a) (= 1.5 a) (= 2.0 a))
)

Note: Ensure you set *UCSAngle* to nil in the main program each time it is run.

Cleaning up other object types
For processing polylines, we do this:

( (= "AcDbPolyline" ObjName)
  (prompt (strcat \nModifying : " objname))
  (vlax-put-property
   obj
   'Coordinates
   (f:vla-coordlist->variant
    (mapcar
     '(lambda (pt) (f:RndPt pt prec))
     (f:vla-variant->coordlist
      (vlax-get-property obj 'Coordinates)
      2
     )
    )
   )
  )
  (prompt "...fixed.")
) ; end AcDbPolyline
The main difference here is that we use the (mapcar '(lambda (pt) ) cliché to round each 2D point list in the returned array of 2D points from the (f:vla-variant->coordlist) function.

Similarly, 2D Solids use this to modify their points:

```lisp
( (= "AcDbSolid" ObjName)
  (prompt (strcat "\nModifying : " objname))
  (vlax-put-property
   obj
   'Coordinates
   (f:vla-coordlist->variant
    (mapcar
     '(lambda (pt)
        (f:RndPt pt prec)
     )
    (f:vla-variant->coordlist
     (vlax-get-property obj 'Coordinates)
     3
    )
   )
  )
  )
  (prompt "...fixed.")
) ; end AcDbSolid
```

Note the (f:vla-variant->coordlist) function uses an argument of 3. Block References use this to clean up insertion points:

```lisp
( (= "AcDbBlockReference" ObjName)
  (prompt (strcat "\nModifying : " objname))
  (vlax-put-property
   obj
   'InsertionPoint
   (f:vla-coordlist->variant
    (f:RndPt
     (f:vla-variant->coordlist
      (vlax-get-property obj 'InsertionPoint)
      3
     )
     prec
    )
   )
  )
  )
  (prompt "...fixed.")
) ; end AcDbBlockReference
Arcs and circles share the same properties to adjust, namely the center and radius. Note how we are only using the \( f:rnd \) function on the radius property, because it’s a single Real value:

\[
\begin{align*}
&\text{( or (= "AcDbCircle" ObjName) (= "AcDbArc" objname))} \\
&\text{(prompt (strcat \"nModifying : \" objname))} \\
&\text{(vlax-put-property obj
}
\text{'center}
\text{ (f:vla-coordlist->variant}
\text{(f:RndPt}
\text{ (f:vla-variant->coordlist}
\text{(vlax-get-property obj 'Center)
}
\text{3}
\text{ )}
\text{ prec}
\text{ )})
\text{)}
\text{(vlax-put-property obj
}
\text{'radius}
\text{ (f:Rnd (vlax-get-property obj 'Radius) prec) }
\text{)}
\text{(prompt "...fixed."))}
\end{align*}
\]

; end AcDbCircle

**Handling AcDbRotatedDimensions with DXF codes**

For handling the AcDbRotatedDimension object type – linear dimensions to you and me – we need to revert back to our old-school DXF-processing entget handling ways. That’s because the ActiveX API doesn’t expose the definition points as properties, and those are the properties we need to clean up. Once processed our dimensions should clean up, value wise.

To handle DXF codes, it’s actually very simple – way simpler, as a matter of fact, than dealing with variants and safearrays and LISP values and whatever. It’s just somewhat slower.

The code in the \( f:RndObject \) function looks like this:

\[
\begin{align*}
&\text{( (= "AcDbRotatedDimension" ObjName))} \\
&\text{(prompt (strcat \"nModifying : \" objname))} \\
&\text{(setq ed (entget e))} \\
&\text{(setq dxf_list '(10 13 14))} \\
&\text{(setq ed (f:RndData ed dxf_list prec))} \\
&\text{(entmod ed) ;; Update drawing from new edata} \\
&\text{(prompt "...fixed."))}
\end{align*}
\]

; end AcDbRotatedDimension

The big story here is the \( f:RndData \) function, which simply takes an entity data list (acquired via \( \text{entget} \)), a DXF list to modify (10, 13, and 14 refer to a dimensions dimension line location, the first definition point, and the second definition point, respectively), and the precision to which you round those DXF values off.
Here's the (f:RndData) function:

```lisp
(defun f:RndData (ed dxf_list prec)
  ;; Version: 2011.10.01
  ;; Description:
  ;; Simple DXF-style data rounding function
  ;; Parameters:
  ;; ed - Entity data list
  ;; dxf_list - List of DXF codes to process, e.g. '(10 11 12)
  ;; prec - Precision as Real
  ;; Returns:
  ;; Modified (rounded) object edata for DXF codes specified by dxf_list
  ;; Notes:
  ;; 2011.10.01: AU 2011
  ;; Keywords: Rounding Round DXF
  ;;
  ;; (mapcar
  ;'(lambda (l)
    ;; (cond
    ;  ( (member (car l) dxf_list)
    ;    (cons (car l) (f:RndPt (cdr l) prec))
    ;  )
    ;  ( T l)
    ;)
    ;)
  ;ed
  ;)

Recall an entity data list looks like this:

```lisp
((-1 . <Entity name: 7fff605d50>) (0 . "LINE") (330 . <Entity name: 7fff6039f0>) (5 . "1CD") (100 . "AcDbEntity") (67 . 0) (410 . "Model") (8 . "0") (100 . "AcDbLine") (10 5.125 3.0 0.0) (11 10.875 3.0 0.0) (210 0.0 0.0 1.0))
```

What (f:RndData) does is go through the edata association list, cons by cons, and if the first number in each cons is a member of the dxf_list argument, round that value off and place it back in the entity data list, returning the same code but with modified points.

Because we are only using this for dimensions, we could have coded the (f:RndData) function using a classic (subst newlist oldlist edatalist) expression. However, in some cases entities have multiple instances of the same DXF code (e.g., Groupd Code 10 in a Polyline), which (subst) would ignore as it only substitutes the first association list that matches the DXF code.

Back in the (f:RndObject) function this list is updated in the drawing via (entupd ed).

### Handling AutoConstrain Options

The last thing to wrap up is to enable some options for handling AutoConstrain. To do this we will use this (f:GetBoole) function, which will ask the user a Yes No question and return T or nil based on the answer. This can be used to fork a program into a certain direction when required.
(defun f:getboole (msg dflt / r)
 |Version: 2011.10.01
 Description:
 Prompts the user with "<message>? <Y/N>".
 If DFLT is non-nil, the message will prompt with a Y.
 If DFLT is nil, it will use N.
 Parameters:
  msg - Message as String without the '?
  dflt - T ("Yes") or nil ("No")
 Return Value: T or nil
 Notes:
  Thanks to Tony Tanzillo
  2011.10.01: AU 2011
 Keywords:
  |
 ;(initget "Yes No")
 (setq r
   (getkword
    (strcat \n" msg ". [" (if dflt "Y" "N") "]: ")
   )
   ;; If the user entered a response, return it.
   ;; Otherwise return the default T/nil.
   (if r (eq r "Yes") dflt)
)

In the main program, we can do this to handle our AutoConstrain options:

;; Initialize a global variable for AutoConstraining each line.
;; Also reset the *UCSAngle* variable
(setq *ACLine* nil
  *UCSAngle* nil
  ss (ssget)
  ssl (sslength ss)
)

;; Ask the user if they want to AutoConstrain each line individually,
;; or temporarily AutoConstrain selected entities if the selset is < 100
(cond ( (f:getboole "AutoConstrain each line individually" nil)
   (setq *ACLine* T)
 )
 ( (< 100 ssl)
   (prompt
    "\nOver 100 entities selected. AutoConstrain option disabled."
   )
 )
 ( (f:getboole "Temporarily AutoConstrain all selected entities" nil)
   (command "._autoconstrain" ss "")
   (command "._delconstraint" ss "")
 )
 ( T (prompt "\nNo AutoConstrain option set."))
}
Back in the (f:RndObject) function, for the AcDbLine object, we have this:

```lisp
;; Add AutoConstrain behaviors before rounding anything
(cond ( (= "AcDbLine" ObjName)
    (if *ACLine*
        (progn
            (command "_autoconstrain" e "")
            (command "_delconstraint" e "")
        )
    )
    ;; Only clean up linework that is horizontal or vertical
    (if (f:IsLineOrtho obj)
        (...rest of code...)
    )
)
```

That pretty much explains the majority of the programming. Please download the source dataset file from the AU class website and explore the code. I’ve added some minor tweaks to it here and there, but you should get the gist of what we were trying to accomplish.

**Timing Fixup Operations**

One of the things you should do is, on intensive utilities like this, set up a timer to evaluate the impacts of certain options or operations, to see if they negatively affect performance and by how much. To to this we need a simple “start timer” function and an “end timer” function which bookends the selection set processing loop.

```lisp
(defun f:StartTimer (/ s)
  (setq *StartTime* nil)
  (setq s (getvar "DATE"))
  (setq *StartTime* (* 86400.0 (- s (fix s))))
)

(defun f:EndTimer (/ s EndTime)
  (if *StartTime*
      (progn
        (setq s (getvar "DATE"))
        (setq EndTime (* 86400.0 (- s (fix s))))
        (prompt
            (strcat "\nTime: " (rtos (- EndTime *StartTime*) 2 8) " seconds")
        )
        (prompt "\nError: No start timer initialized."))
      (princ)
  )
)
```

Then these will be called in the Main program function here:

```lisp
(f:StartTimer)
(f:ssmap
  '(lambda (e)
      (f:RndObject e *Prec*)
    )
  (ssget)
)(f:EndTimer)
```
How fast is FIXUP?
Here are some results for processing a selection set of 8,400 objects:

With no AutoConstraints applied: 2.281 seconds
With AutoConstraints applied to individual Line objects: 54.874 seconds
With AutoConstraints applied and prompts added for each object: 120.754 seconds

Using FIXUP in Practice - Tip and Tricks
Take it for a spin on the accompanying FixupFloorPlan.dwg floor plan drawing in the downloadable dataset, which has been purposefully adjusted to be out of whack. You’ll see that FIXUP can fix an entire floor plan up in mere seconds. It’s VERY fast. However, there are a couple of preparatory steps you want to take when using FIXUP on a floor plan:

1. Before you use FIXUP, be aware of what it does. It MOVES geometry around. Just like nuclear power, if misused it could lead to disaster. As you learn how it works, I suggest working on small selection sets (or whole copies of your drawings) as you test the waters.

2. FIXUP is based on the notion that if you have drawing entities at some nice X/Y point from the current UCS origin, they will all be in nice X/Y distances from each other. To make this happen easy, it’s best to “base” the drawing (plan, elevation, section, detail, whatever) at a nice distance from the UCS origin. To do this:
   a. Unlock all of the layers and turn on / thaw all layers. Turn SNAP on and set it to a large value – 1’ or 5’ . In Model Space, issue the MOVE command, and grab the entire extents of the drawing. For the base point, select a “good” point on the drawing – the center of a column grid intersection is ideal.
   b. Once the MOVE command is in play and asking for the target point, SNAP should become active and your cursor will jump around (this is a new AutoCAD 2012 behavior). Then place the point in WCS space some nice distance from the origin.
   c. To check how good or bad the drawing is dimensionally, issue the ID command and OSNAP to an exterior corner point, intersection of two orthogonal walls, or some other hard point. You should be coordinate values that end with X/8” or X/16” at the most. If you get X/256”, that is indicative of a problem.

3. If the plan has angled wings or oblique geometry, set a UCS oriented to that geometry using the UCS “Object” option to align with a wall or some other entity. Put the origin of this UCS at some point that is important to that UCS that could be used as a starting point for dimensioning, such as the outside corner of the wing of a building.

4. Issue the FIXUP command, set the precision to 1/8”, and select small portions of the drawing you wish to work on. Typically, you should select only those things to clean up, check them for accuracy, and move on to clean up additional portions of the plan.

5. For completely orthogonal buildings, you could select the entire drawing at once, and it should complete in less than two seconds if you do not AutoConstrain things. You should see any dimensions that were previously ugly instantly fix themselves to nice numbers.
They may not be precisely what values they should be, but any discrepancies are now easy to fix.

6. Because 1/256" is only about 0.003", you may want to temporarily set your UNITS to decimal with a precision of 8 decimal places. That will allow you to see minute discrepancies which need fixing, such as a 5" wall that now lists as 4.9996".

7. After you run FIXUP, use the Distance command (and check out the utilities later in this handout) to check geometry, such as distances between parallel lines and so on. You may see endpoints that no longer meet because FIXUP “squared up” the linework to be truly orthogonal before it fixed the endpoints. For most drawings any gaps are the precision amount, e.g. 1/8"

Here’s a floor plan before:
Additional Techniques to Improve Day to Day Accuracy
Using the ID, LIST, and DISTance Commands Effectively

As we have seen, understanding and exploiting the user coordinate system is a powerful weapon against inaccuracy. Using the ID command as you select endpoints or intersections is a valuable tool to identify potential issues, because once the plan is “based” at some nice distance from (0,0), any coordinate you get back with X/256” in them is indicative of Bad CAD™.

Here’s a good example of using ID (with LUPREC = 8 to report the highest level of precision):
Command: id Specify point:  X = 2'-3 3/8"     Y = 1'-2 1/4"     Z = 0'-0"

Here’s a bad example indicating something needs fixing (LUPREC = 8):
Command: id Specify point:  X = 4'-11 21/64"     Y = 1'-5 9/256"     Z = 0'-0"

LIST returns similar information but on the whole object. LISTing a badly drawn Line, for example would reveal something like this:

Select objects:

```
LINE   Layer: "0"
   Space: Model space
   Handle = 1d7
   from point, X=1'-1 43/256"  Y=2'-2 33/128"  Z=   0'-0"
to point, X=3'-2 105/256"  Y=2'-2 67/256"  Z=   0'-0"
Length =2'-1 31/128", Angle in XY Plane = 0.00886667
delta X =2'-1 31/128", Delta Y = 0'-0 1/256", Delta Z = 0'-0"
```
Here you have the evil trinity of bad start and end coordinates, a bad angle (very close but not quite 0), and a bad length to boot.

The DISTANCE command is one of the most popular day to day commands and is often used with the NEARest and PERPendicular Osnaps, which give you the distance between any two parallel lines, such as the width of a corridor or wall assembly. So, let’s put them into a LISP routine:

```lisp
(defun c:ds ()
  ;| Version: 1.0
  ; Description: Distance from NEARest to PERPendicular
  ; (distance between two parallel lines)
  ;
  (prompt "\nDistance Near - Perp")
  (command "_.dist" "near" pause "perp")
  (princ)
)
```

**Helpful CUI Customizations**

**F1 = FROM**

The FROM command modifier is possibly one of the least used features in AutoCAD but one of the most powerful. It allows you to locate a point from another reference point within a command. When prompted to enter a point, it temporarily suspends the command in progress and asks “How far and in what direction from one point do you want to specify another point?” and feeds the new point back to the command in progress.

Mapping it to the F1 key means it’s easily accessible and it keeps you from launching the HELP system every three seconds when you meant to hit ESC.

**F4 = PERPendicular OSNAP**

It’s often very useful to use the PERPendicular osnap by itself, instead of in a running osnap. In conjunction with FROM, you have a great tool to place objects a certain distance away from the face of another object, such as a wall, in one shot.

With FROM tied to F1 and PERPendicular tied to F4, you have an extremely powerful one-two punch to beat your daily architectural drafting into submission. Let’s say you have a desk in a floor plan, but need to accurately place the front edge 4” to the left from face of a vertically drawn wall. Some might offset a line of the wall 4” off, and place it using NEAR, PERP, or some other osnap. But you don’t have to draw any construction or temporary lines to do this right at the mouse in one shot:

First, turn on OTRACK and POLAR snap, and turn off ORTHO.

Command: **Move** *(Select the desk)*
Select objects: 1 found
Select objects: *(Enter to finish selection)*
Specify base point or [Displacement] <Displacement>:
  *(Pick a point on the desk nearest the wall, e.g. the midpoint)*
Specify second point or <use first point as displacement>: *(Hit F1) _from*
Base point: *(Hit F4) _per to (select PERP osnap on face of wall)*
<Offset>: *(drag to the left of the wall and type 4")*

When you hit F1 in response to the MOVE command’s request for a target point, the MOVE command is suspended. You first specify the nearest point on the wall by using the PERPendicular osnap from the point on the desk. Then drag to the left 4”. That new point is fed to the MOVE command as the target point, and the desk is dropped in place at exactly the correct location. No construction line needed.

To assign these keys to the Function keys, enter the CUI editor using the CUI command. In your Main CUI, search for the “Snap, From” command. Drag and drop it onto the Keyboard Shortcuts > Shortcut Keys node. In the Access > Keys section, hit the right hand button and hit the F1 key to assign it.

Note: AutoCAD is pretty finicky when redefining F1. If you have a single, custom CUI (no partial CUIs and no Enterprise CI), redefining F1 may not work at all – F1 will stay on Help. Sometimes the best way is to transfer it from another CUI file. See this thread on AUGI’s forum: http://forums.augi.com/showthread.php?t=69086 which has a downloadable CUI for transfer.

![Image of CUI editor showing shortcut keys assignment](image)

**Interactively Changing the SNAP Setting with a Menu Pick**

Part and parcel of using SNAP effectively is being able to change it on the fly within another command. This gives you the freedom to keep it on and knock it down to 1/8” to 1/16” to 1/32” when need be, such as within a 3”=1’-0” detail.

To make this work, we create a shortcut cursor menu that will appear when requested. By default, using the SHIFT or CTRL with the right-mouse button will display the same OSNAP context menu when a command is active. We can commandeer one of these to display our own interactive SNAP menu.
In the CUI dialog, create a new command called 10'-0. In the Properties, set the Command Macro to `\snap 120.
Repeat the procedure for any and all SNAP sizes you would commonly use, e.g. 6", 1", 1/2", 1/4", 1/8", 1/16", 1/32" and so on.

Then create a new Shortcut Menu and call it Drawing Snap. In the Properties, add a new alias called DWGSNAP in addition to the one that exists by default. Drag and drop all of the custom snap commands you made into this new menu node.

To assign it to a mouse button combination, we need another Command. Create a new Command called Drawing Snap Menu. In the properties, set the Macro to $P0=[CUI_MENUGROUP_NAME].DWGSNAP $P0=*
This is the menu syntax for a popup cursor menu. For example, if you are customizing the ACAD.CUIx, the menugroup is called ACAD, so the command is $P0=ACAD.DWGSNAP $P0=*

Go to the Mouse Buttons node, and expand either the Shift+Click or Ctrl+Click node. Button 2 is the right mouse button, and button 3 is the middle mouse button. By default, Shift+Button 3 is the transparent orbit command, which is rather useless when you are doing 2D drawings. Drag and drop the Drawing Snap Menu command to the Button 3 line.

Easily Restoring Named UCSes, Using the PLAN command, and Views

UCSes are extremely valuable tools for working with complex floor plans, particularly ones with many angles. Although designed to work in 3D space, use them to your advantage in 2D to orient your X and Y axes aligned with the geometry in the drawing. By doing so you now make it easy to use FIXUP on similarly-oriented entities, such as the architecture for a wing as seen project shown on p.14.

When working on complex plans, create a series of UCSes named 1 through 9. Then create these command macros C:U1…C:U9 that restore those UCSes and orientations instantly (I've removed the headers for brevity). This gives you instant recall of these UCSes.

```lisp
(defun c:u1 ()
  (prompt "\nUcs Restore 1")
  (if (tblobjname "ucss" "1")
    (command ".\_ucs" "r" "1")
    (prompt "\nUcs 1 not defined.")
  )
  (princ)
)
```
(defun c:u2 ()
  (prompt "\nUcs Restore 2")
  (if (tblobjname "ucs" "2")
    (command "_.ucs" "r" "2")
    (prompt "\nUcs 2 not defined.")
  )
  (princ)
)
Etc.

For each UCS set the Origin to a proper point on the building, such as an exterior corner or column intersection.

The PLAN command is an even more useful feature, particularly when used in conjunction with named UCSes. It rotates your view so that the UCS X and Y axes are positioned on screen the same as they are in the WCS, making it simple to work on the orthogonal portions of a plan at this particular orientation.

Similarly, you can create a series of C:P1...C:P9 command functions that restore the UCS # and issue the PLAN command for that particular UCS:

(defun c:p1 ()
  (prompt "\nPlan UCS 1")
  (if (tblobjname "ucs" "1")
    (progn (command "_.ucs" "r" "1")
            (command "_.plan" "")
    )
    (prompt "\nUcs 1 not defined.")
  )
  (princ)
)
(defun c:p2 ()
  (prompt "\nPlan UCS 2")
  (if (tblobjname "ucs" "2")
    (progn (command "_.ucs" "r" "2")
            (command "_.plan" "")
    )
    (prompt "\nUcs 2 not defined.")
  )
  (princ)
)
Etc.

Finally, create a series of C:V1...C:V9 command functions that restore any one of 9 preset views. Views store the UCS that was in place when they were created, so it makes it easy to not only restore a UCS, not only put it in the proper orientation for editing, but will also focus on a particular area.

(defun c:v1 ()
  (prompt "\nView Restore 1")
  (if (tblobjname "view" "1")
    (command "_.view" "r" "1")
    (prompt "\nView 1 not defined.")
  )
  (princ)
)

(defun c:v2 ()
  (prompt "\nView Restore 2")
  (if (tblobjname "view" "2")
    (command "_.view" "r" "2")
    (prompt "\nView 2 not defined.
"))
  (princ)
  )

Etc.

**Incremental MOVE and SHIFT Utility Commands**

Lastly, in the downloadable dataset I have provided a SHIFT.LSP utility which contains two utilities, C:SH and C:MM, for making repeatable adjustments to drawing objects using Stretch and Move.

The SH utility stretches objects by a selected incremental amount in an up, down, left, or right direction. Similarly, MM will incrementally move objects in the same manner. Both commands write the values the user enters for the incremental distance and the direction to the Registry, allowing those values to be reused between open drawing sessions.

Both are really useful for cleaning up some discrepancies after an extensive FIXUP session. Please refer to the class session on the AU web site for the source code.

The SH command basically does this:

1. Pick the area or objects you wish to stretch
2. Tell it what distance you want to stretch or move. This number will be saved to the Registry.
3. Tell it what direction – U, D, L, or R - it should stretch or move the objects.
4. It will repeat, so you can either:
   a. Hit ENTER to use the same distance and direction; just pick two additional points to define the stretch window;
   b. Type in U, D, L, or R to specify a new direction, hit ENTER to use the same distance, and select two points to stretch;
   c. Type in a new distance using the same direction; hit ENTER to use the same direction, and select two points to stretch;
   d. Type in a new distance and direction, and select two points to stretch.
   e. Note: The two points can be either left-to-right or right-to-left.

**Thank you for attending!**

Matt Stachoni
Senior AEC Software and Information Technology Specialist

**CADapult LTD**
(302) 733-0477 x121 Phone
(302) 547-0895 Mobile

Empowering Design With Integrated Solutions

matt.stachoni@cadapult.net
matt@stachoni.com
http://www.cadapult.net/AecBlog