Autodesk University 2007

Advanced Visual LISP®

Speaker: Doug Broad, Nash Community College

CP311-4 Put the power of Visual LISP in your hands to create, compare, and edit drawings; interact with Excel; access the Windows registry; and work with reactors. This class demonstrates how to use the active document, other open documents, and documents using ObjectDBX, and how to interact with other applications. We'll discuss tips for using VLIDE (Visual LISP Integrated Development Environment), demystifying ActiveX documentation, using ObjectDBX, accessing the Windows registry, and putting reactors to work. Sample programs will be included. Participants should know LISP but need not know Visual LISP

About the Speaker:

Doug Broad has taught Architectural Technology at Nash Community College, Rocky Mount, NC and has been an Architect since 1986. An AutoCAD user since R2.6, an AutoLISP user since R9, and a Visual LISP user since R2000, he has developed many programs to help in his Architectural practice. Currently using AutoCAD Architecture 2008, LISP and Visual LISP continue to give him a competitive edge.
dbroad@nashcc.edu

Autodesk®


Autodesk
University
2007

Make Sure Vlisp Is Loaded First With (VL-LOAD-COM)

For some past versions, the loading of the visual lisp system has been automatic. AutoCAD 2008 does not load the system automatically.

Rule: Assume the system will not be loaded before your program loads.

Advice: Put (vl-load-com) at the top of any file to be loaded. There is no need to include them inside any user defined function except when posting code to newsgroups.

Otherwise: Error messages are generated indicating functions aren't defined:
$ (vlax-get-acad-object)
; error: no function definition: VLAX-GET-ACAD-OBJECT

A Comparison of LISP Methodologies. Use Which One is Best In Context	Good/Bad	Command Methods		ActiveX Methods		DXF Methods		
		Command	SendCommand	VLA-	VLAX-	Entmod	Entget	Entmake
Reactor Callback Capable	+			●	●			
Best raw speed	+					●	●	●
Best access to active drawing	+					●	●	●
ObjectDBX Capable	+			●	●			
Object access is self commenting	+	○	○	●	●			
Supports selection sets.	+	●	●	●	●	●	●	●
Can be used with other open drawings.	+		●	●	●			
Can be used to control other applications	+			●	●			
Works in R14 and earlier	+	●				●	●	●
Is affected by System Variables	-	●	●					
Depends on Command Prompt Sequence	-	●	●					
May require complex error handlers	-	●	●					

Why Learn Vlisp?

1. To allow your LISP programs to interface with other documents and with other applications.
2. To use reactors to automate certain tasks. (Note: Most reactive tasks have been made unnecessary due to fields, dynamic blocks, groups, palettes, the design center and formulas in tables.
3. To better isolate your programs from other LISP programs.
4. To make your code more readable.

Short Review of Vlide Features

Autodesk has provided VLIDE (Visual Lisp Integrated Development Environment) to assist with LISP coding. Since other AU classes have focused entirely on these features, we will only review its primary productivity features:

1 ***Syntax coloring:***

a. Reserved words: blue

i. Indicate functions protected by Autodesk. Help identify correct spelling and protect mistaken reassignment as user variables.

b. Parentheses: Red

c. Nothing is worse than misplaced, missing, or extra parentheses.

d. Comments: Grey

i. The editor ignores these but you shouldn't. Be overabundant with explanations about the intentions of your code.

e. Strings: Magenta

i. A misplaced double quote can really mess up a program. If large parts of your program are pink, look for the beginning and decide where to end the string.

f. Constants: Green

i. The letters I, l, and 1 look alike. The color here makes the choice obvious.

2 ***Auto-indenting and parentheses matching***

a. Indicates how many parentheses remain unclosed

b. Each right parenthesis flashes its corresponding opening parenthesis.

c. Double-clicking on a parenthesis highlights the entire expression.

3 ***AutoComplete*** 

a. Type enough of a built-in function or reserved word and you can finish by using CTRL+Spacebar.

b. A short list of choices can be selected from pop down list. See video.

4 *Appropos*

a. Forget exactly how to spell something or need to look it how to use a function. No worries. Use apropos. Help is a quick jump from there.

5 *Database exploration.*

a. View -> Browse drawing database

b. This is great for finding out about objects. Avoid the need to use:

   i. (setq entinfo (entget(car(entsel))'("*")))

   ii. (setq tblinf (entget (tblobjname "layer" "targetlayer")))

6 *Debugging Tools*

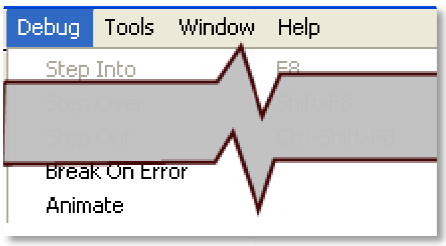
a. Ability to partially load.

   i. Double-click a parenthesis in the code editor and choose "load selection". After looking at the results in the console and inspecting variable values, load the next expression, etc.

   ii. Reveals errors by showing the intermediate values. Reduces the need to add debug printing statements to log variable values.

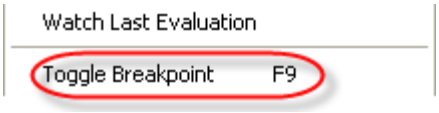
b. Break on error.

   i. Don't know where problems lie? Use this to hold the value of local variables in a known state and open the error trace window.



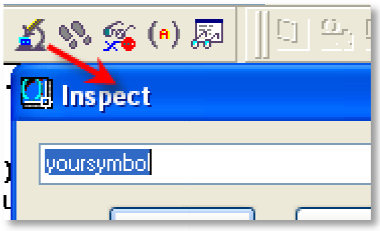
c. Code animation

   i. Find out where your code is spending its time and where execution stops.



d. Breakpoints:

   i. Stop at a predetermined point to learn the values of your variables at any point. Code step-through can be performed.

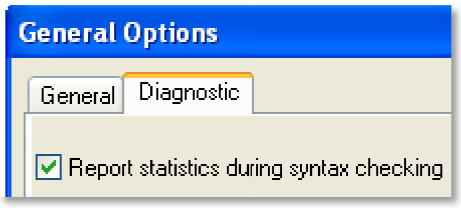


7 Object inspector:

a. Beginning with a value or symbol(you can enter an expression like (vlax-get-acad-object), you can begin exploring any related thing.

8 Syntax checking

a. Aren't sure whether you have localized your variables? Choose to report statistics during syntax checking



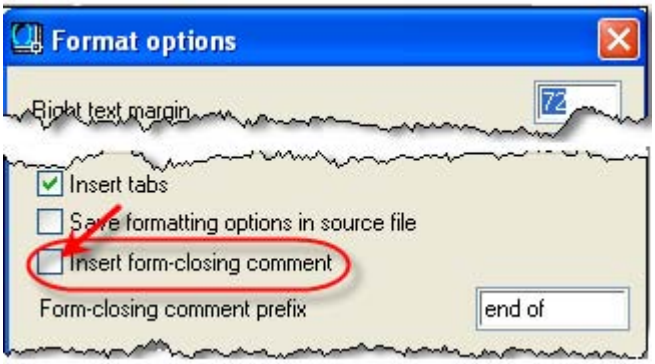
9 Form closing comments

a. Think you know where each expression ends? Bet you don't.

10 Last Break Source

a. CTRL+9

b. Jump straight to the problem.



11 Error Trace

a. CTRL+SHFT+R

b. Identify what is broken. See missing variable values – NIL ---

12 Developer Help F1

a. This really should be first rather than last since it has continued to be my most valuable programming resource.

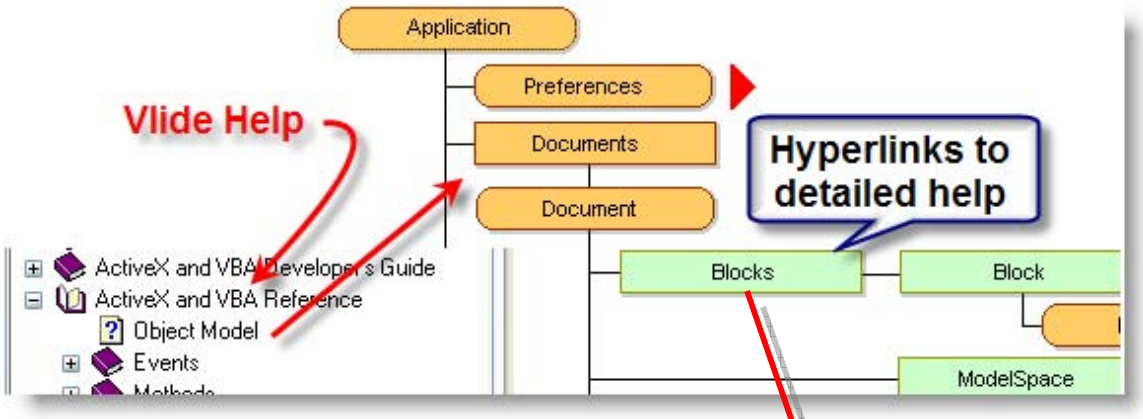
b. In 2008, the search features were mistakenly omitted (bug). To be able to search, go to the AutoCAD window and choose Help->Developer help from the menu. The search features there are still intact.

c. Find the object model under the ActiveX and VBA help area. This is most important for the Vlisp programmer.

Vlide's Object Model Documentation

1. What is it?

a. A color coded outline block diagram starting from the application level down to individual objects.

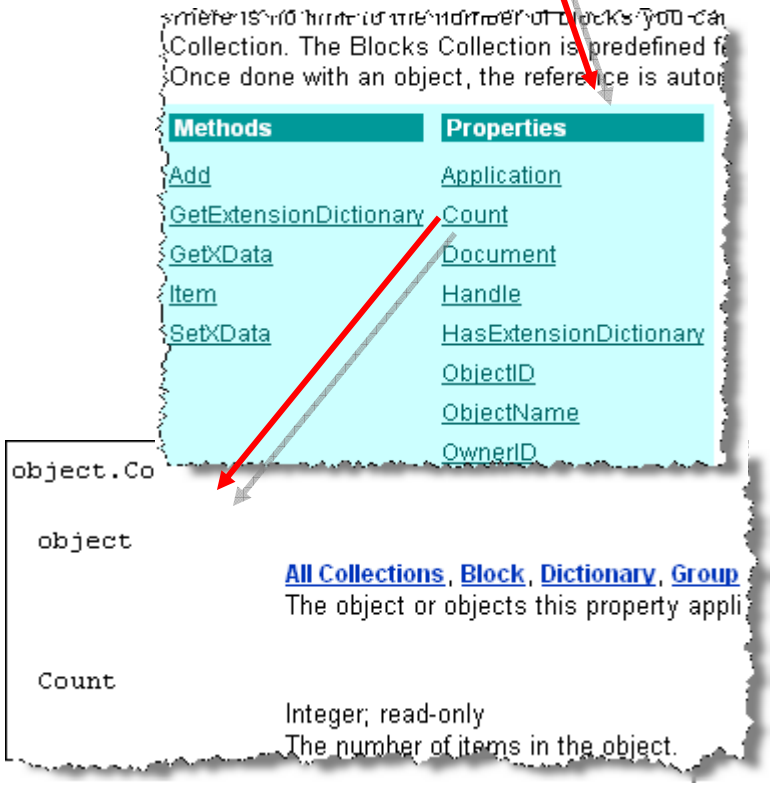


2. Why use it?

a. To understand the structure of the system.
b. To quickly access object methods and properties.
c. To quickly access help files.
d. To learn the access function names and their arguments.

3. How to get to it?

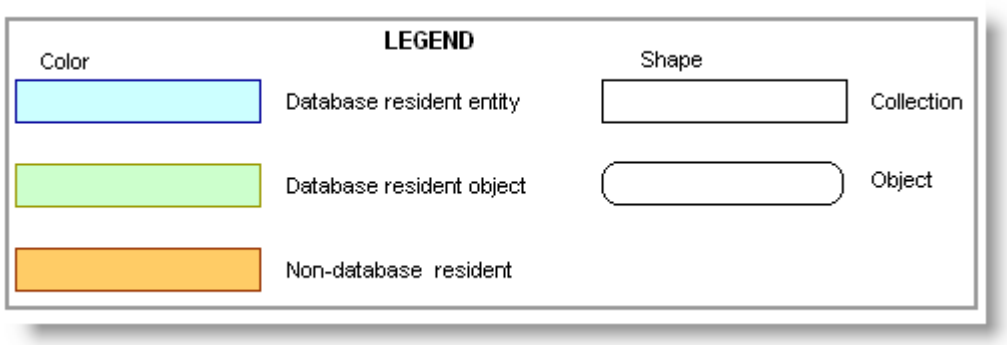
a. In Vlide hit F1.



Component Object Model Terminology

Collection: A group of objects associated with one another and sharing common characteristics and methods

- Plural node names are usually collections.
- The shape of the node (rectangles are collections). Slots are objects.



Legend of Object Model

Object: A single indivisible part (Application, Document, Block, Line, Dictionary, Xrecord) Objects usually belong to collections but they could be properties of Objects or Collections.

Selectionsets: similar to collections but are considered objects. They have slightly different capabilities and methods. For example, most collections have an add method. Selectionsets have an additem method. You can still use vlax-for and vlax-map-collection on them.

Property: a characteristic of an object or collection. Some properties are read-only. A **property** and a **collection** can refer to the same thing. Blocks collection is a property of a document. So is the layers collection.

Method: an action applied to a collection or object. Some methods are inherited but are not explicitly stated. To remove an object from a block, you delete the object.

Referenced: Objects cannot be deleted if they are referenced by others objects.

Owner: an object that possesses or has responsibility of other objects or collections. Model space objects are owned by the model space.

Accessing the Object Model with Vlisp

The **application** object is the root object (top level). Since it contains all other objects, you have access to everything below. Starting from the application object however, is not usually the most efficient way to lower level elements.

Object Access from Application Object		
Level	Access To	Use
1	Application	(vlax-get-acad-object)
2	ActiveDocument	(vla-get-activedocument (vlax-get-acad-object)))
3	Modelspace	(vla-get-modelspace(vla-get-activedocument(vlax-get-acad-object)))
3	Paperspace	(vla-get-paperspace(vla-get-activedocument(vlax-get-acad-object)))
2	Preferences	(vla-get-preferences(vlax-get-acad-object))
3	ActiveSelectionSet	(vla-get-activeselectionset(vla-get-activedocument(vlax-get-acad-object)))

Methods To Process A Collection	
Basic Format	Description of Action
(vlax-for Item Collection Expressions)	Processes every object in the collection. Equivalent to Foreach
(vlax-map-collection Collection Function)	Applies the function to every object in the collection. Equivalent to Mapcar
(setq i 0) (while condition (setq Item (vla-item Collection n) i (1+ i)) Expressions)	Allows partial processing of collection until condition no longer applies

Object Access From a Line Object	
Access To	Use
Application	(vla-get-application line)
Document	(vla-get-document line)
Block Containing Line	(vla-ObjectIDToObject(vla-get-document line)(vla-get-ownerid line))
Modelspace	(vla-get-modelspace(vla-get-document line))

Addressing low-level items like the Modelspace collection by starting from the application object is tedious. Therefore, it is good practice to save either the active document or the active Modelspace object and then to reference lower level objects starting from there.

```
(setq radius 5 center (vlax-3d-point '(0 0 0)))
(setq ThisDoc (vla-get-activedocument (vlax-get-acad-object)))
(setq blocks (vla-get-blocks ThisDoc))
(setq mspace (vla-get-modelspace ThisDoc))
(vla-addarc mspace center radius 0 pi)
```

Such code may be more readable and easier to test than the more direct:

```
(vla-addarc
  (vla-get-modelspace
    (vla-get-activedocument (vlax-get-acad-object))
  )
  (vlax-3d-point '(0 0 0))  5  0  pi)
```

The direct approach has one advantage: **there are no variables stored or named**. The purposes for each argument can be commented instead.

By using separate setq statements, the programmer can individually test and debug the process.

The following example is a library access function which uses techniques some find debatable. It can, however speed access and can shorten other functions, improving readability.

Example:

```
(defun ThisModelSpace ()
  (cond (*ModelSpace)
        ((setq *ModelSpace
                (vla-get-modelspace
                  (vla-get-activedocument (vlax-get-acad-object))
                )))))
```

The library function “ThisModelSpace” above stores a global variable to make successive access to the object faster. Caution should always be exercised when using global variables however, since they are outside the scope of the functions that use them and since other functions might use the same symbol names for other purposes. For Vlisp, also, there can be a problem when objects aren’t released. Generally it is safe to keep the active document, the application object, and the model and paper space collections as globals. Never assume lower level objects should be stored since they can change or disappear.

Given a line stored in variable “a”. Let’s say the line gets erased. An attempt to access object data from an erased line leads to an error:

(vla-get-startpoint a)

; error: Automation Error. Object was erased

Using local variables to hold low level vla-objects is generally safe and convenient.

Another way to access objects is by going up or down from an object already known. Each object has a single owner and may own multiple objects. By accessing the owner of an object, you can go up the model.

```
(setq blockid (vla-get-ownerid line))
(setq *doc (vla-get-document line))
(setq blockobj (vla-ObjectIDToObject *doc blockid))
```

At this point blockobj points to the owner object of the line.

Who Owns This Object? - A library function could be written as follows:

```
(defun GetOwnerObj (object)
  (vla-ObjectIDToObject
    (vla-get-document object)
    (vla-get-ownerid object)))
```

Access to the object's owner is gotten by (GetOwnerObj Line)

Library functions, if employed, should be independent, containing all the code to do their job without additional functions. Library functions should also be well written and thoroughly tested. If not self-evident, they should be well commented.

Which layout block is current?

Once a function is passed an entity object, the owner object (its block) can be determined with GetOwnerObj. To get the current layout block without having an object to refer to, use properties of the activedocument and activespace or rely on system variables.

```
(defun GetCurrentSpace (Doc)
  (if
    (= (vla-get-ActiveSpace Doc) 1) ;like tilemode
    (vla-get-modelspace doc)
    (vla-get-paperspace doc)))
```

The GetCurrentSpace function takes a document object. This makes the function, non-document specific. This makes it useful for working in other documents. It won't work on ObjectDBX documents though.

Lists, Safearrays, and Variants

Try to avoid safearrays if possible

It is often possible to avoid using safearrays and variants. Doing so can simplify programming tasks, improve readability, and speed execution. Use Vital lisp functions rather than those generated by vlax-import-type-library automatically for you.

Example: To get the start and endpoints of a line by accessing the line object.

$(setq sp1 (vla-get-startpoint *lineobject)*)

#<variant 8197 ...>

$(setq sp2(vlax-get *lineobject* 'startpoint))

(893.563 535.5 0.0)

SP1 is in a form that can be used as input to another Vlisp function but which can't be used directly by legacy lisp functions as a list. To get the list form given SP1, use

(Safearray-value (variant-value sp1))

Caution: The safearray-value function only works for single dimension safearrays. For more general translation, use vlax-safearray->list. For one dimensional safearrays, safearray-value executes 20% faster.

SP1 can however be used directly as

(Vla-put-startpoint *lineobject2* sp1)

SP2 can be used to set the startpoint by

(Vlax-put *lineobject2* SP2)

So use the vla-get form in tandem with the vla-put form but use the vlax-get form if the program needs to process the list.

To create a Safearray

1. Know your data type
   a. Example data types are shown. See others in the help files.
      i. unknown or mixed vlax-vbVariant
      ii. integers: vlax-vbInteger
      iii. long integer vlax-vbLong
      iv. real numbers: vlax-vbDouble
      v. string: vlax-vbString
2. Create a container for the data that uses that data type.
   (setq Pt1 (vlax-make-safearray vlax-vbdouble '(0 . 2)))

3. Fill the container with your data.
   (vlax-safearry-fill Pt1 '(1.0 2.0 0.0))
   Please note here that Pt1 will change without reassigning it by using setq.
4. Convert to a variant if necessary for function.
   (setq pt1v (vlax-make-variant pt1))

What type is it?

Use the type function to determine what type of data is bound to a variable.

(type x) will return **safearray** if it is a safearray**, variant**, if it is a variant, and **list**, if it is a list.

If the safearray to be made is a point, keep it simple.

(Vlax-3d-point '(10.0 12.0 0.0))

The vlax-3d-point creates a 3 item safearray, fills it, and wraps it into a variant.

Safearrays can be multidimensional

Vlax-make-safearray takes a variable number of arguments. The following makes an array with 3 columns and 4 rows.

(Setq matrix (vlax-make-safearray vlax-vbdouble '(0 . 3) '(0 . 2)))

(Setq mlist '((1 1 0)(2 2 0)(3 3 0)(4 4 0)))

(vlax-safearray-fill matrix mlist)

#<safearray…>

The order for a two dimensional array is (vlax-make-safearray *type rows columns*)

For 3d arrays the order is *planes rows columns*.

Other Safearray functions

(vlax-safearray-get-dim matrix) ;;returns the number of dimensions of a safearray

2

(Vlax-safearray-get-element matrix 3 2) ;;obtains a particular element of the safearray

0.0

The sequence below demonstrates that you can change an individual element of a safearray without converting it to a list and then converting back again.

$(vlax-safearray->list matrix);;a handy function for converting safearrays

((1.0 1.0 0.0) (2.0 2.0 0.0) (3.0 3.0 0.0) (4.0 4.0 0.0))

```
$ (vlax-safearray-put-element matrix 3 2 25.0)

25.0

_$ $(vlax-safearray->list matrix)

((1.0 1.0 0.0) (2.0 2.0 0.0) (3.0 3.0 0.0) (4.0 4.0 25.0))

(setq dim 1)

(vlax-safearray-get-u-bound matrix dim) ;dim = 1 tests first dimension.

3

$(vlax-safearray-type matrix)          ; returns the type of data contained in the safearray

5

$vlax-vbdouble   ; each data type variable is bound to an integer.

5
```

Unfilled Safearrays Are Not Empty

They are initialized. Initialization depends on the data type and is predictable. For Boolean arrays each element is initialized to :vlax-vb-false (vlisp semi-equivalent to nil)

For strings, each element is initialized to an empty string. See help files for other types.

Vlax-Curve Functions

These make some otherwise difficult geometric operations easy.

Vlax-Curve Functions	
Function Format	Returns
(vlax-curve-getClosestPointTo curve-obj givenPnt [extend])	Closest point on the curve to input point
(vlax-curve-getClosestPointToProjection curve-obj givenPnt normal[extend])	Closest point (in WCS) on a curve after projecting the curve onto a plane
(vlax-curve-getDistAtParam curve-objparam)	Length of the curve's segment from the curve's beginning to the specified parameter
(vlax-curve-getDistAtPoint curve-obj point)	Length of the curve's segment from the curve's beginning to the specified parameter
(vlax-curve-getEndParam curve-obj)	Parameter of the endpoint of the curve
(vlax-curve-getEndPoint curve-obj)	Endpoint (in WCS) of the curve
(vlax-curve-getFirstDeriv curve-obj param)	First derivative (in WCS) of a curve at the specified location
(vlax-curve-getParamAtDist curve-obj dist)	Parameter of a curve at the specified distance from the beginning of the curve
(vlax-curve-getParamAtPoint curve-obj point)	Parameter of the curve at the point
(vlax-curve-getPointAtDist curve-objdist)	Point (in WCS) along a curve at the distance specified by the user
(vlax-curve-getPointAtParam curve-obj param)	Point at the specified parameter value along a curve
(vlax-curve-getSecondDeriv curve-obj param)	Second derivative (in WCS) of a curve at the specified location
(vlax-curve-getStartParam curve-obj)	Start parameter on the curve
(vlax-curve-getStartPoint curve-obj)	Start point (in WCS) of the curve
(vlax-curve-isClosed curve-obj)	Is the start point is the same as the endpoint? (T, nil)
(vlax-curve-isPeriodic curve-obj)	Curve has infinite range in both directions and there is a period value dT, such that a point on the curve at (u + dT) = point on curve (u), for any parameter u? T or nil
(vlax-curve-isPlanar curve-obj)	Can a plane that contains the curve? T or nil

Curve parameters should not be interpreted as having a meaning apart from the vlax-curve functions. Autodesk's commands generate curve objects that have predictable parameters. After editing, though, the parameter's meaning may be unclear. Do not assume that a parameter means anything.

Vlax-curve-getfirstderiv returns the endpoint of a vector whose startpoint is 0,0,0. The length of the vector depends on whether the part of the curve picked is straight or curved. If curved, its length is the radius of the curve. If straight, it is the length of the line segment.

This example will draw a tangent line at a point picked:

```
(defun c:drawtangent (/ ename pt1 curveobj deriv1 pt2)
  (mapcar 'set '(ename pt1) (entsel))
  (setq curveobj (vlax-ename->vla-object ename))
  (setq pt1 (vlax-curve-getclosestpointto curveobj pt1))
  (setq deriv1 (vlax-curve-getfirstderiv
                 curveobj
                 (vlax-curve-getparamatpoint curveobj pt1)
               ))
  (vlax-invoke
    (vla-objectidtoobject
      (vla-get-document curveobj)
      (vla-get-ownerid curveobj))
    'addline pt1 pt2 ))
```

Vlax-curve-getsecondderiv returns the endpoint of a vector whose startpoint is 0,0,0 that points in the direction of the center and whose length is the radius of the cuve at the param. If the curve object is straight at the point and has no curve, then it returns a second derivative of (0.0 0.0 0.0)

```
(setq derive2 (vlax-curve-getsecondderiv curveobj
                (vlax-curve-getstartparam curveobj)))


(setq radius (sqrt (apply '+ (mapcar '* derive2 derive2))))
```

This example will draw a radius line from the picked point:

```
(defun c:drawradius (/ ename pt1 curveobj deriv2 pt2)
  (mapcar 'set '(ename pt1) (entsel))
  (setq curveobj (vlax-ename->vla-object ename))
  (setq pt1 (vlax-curve-getclosestpointto curveobj pt1))
  (setq deriv2 (vlax-curve-getsecondderiv
                 curveobj
                 (vlax-curve-getparamatpoint curveobj pt1)
               ))
  (setq pt2 (mapcar '+ pt1 deriv2))
  (vlax-invoke
    (vla-objectidToobject
      (vla-get-document curveobj)
      (vla-get-ownerid curveobj))
    'addline pt1 pt2))
```

ObjectDBX™

What is ObjectDBX™?

An ActiveX service to access and change the contents of AutoCAD drawing files directly without opening them in the AutoCAD editor. Many built-in type library wrapper functions (vla-xxx) will work with an ObjectDBX document (though not officially supported).

Why use ObjectDBX™?

It's fast. Using ObjectDBX™ can speed processing of multiple documents because they do not need to be loaded into the application. You save all the regeneration time, time to load menus, time to initialize lisp, etc.

Are there any drawbacks?

When saving changes, you lose thumbnails.

How is it accessed?

Getinterfaceobject allows in-process execution.

Notice the versioning of the program id.

```
;;;Getting ObjectDbx Method 1
(defun ObjectDBXDocument ()
  (vla-GetInterfaceObject
    (vlax-get-acad-object)
    (strcat "ObjectDBX.AxDbDocument."
            (substr (getvar "acadver") 1 2))
    )
  )
;;Note:  Earliest version did not support
;;the number suffix.
```

(setq odoc (ObjectDBXDocument)) ;uses function above (the recommended method)

```
;;;Getting ObjectDBX Method 2
(defun ObjectDBXDoc ()
  (vlax-get-or-create-object
    (strcat "ObjectDBX.AxDbDocument."
            (substr (getvar "acadver") 1 2))
    )
  )
;;Note:  Earliest version did not support
;;the number suffix.
```

(setq odoc2 (ObjectDBXDoc)) ;uses function above (works but may require extra resources)

What are the limitations?

1. **Commands can't be used.** Therefore vla-sendcommand function will not work.
2. **No access to system variables.** (Vla-get-variable and vla-set-variable aren't supported.)

3. **Object selection doesn't work**. Forget about selection sets. This means you must process entire collections filtering by using object properties.
4. **Entity access functions don't work**.
    a. Don't try to use entget, vlax-vla-object->ename, entmod, ssget, etc.

What methods and properties are supported?

```
(vlax-dump-object odoc t)
; IAxDbDocument: IAxDbDocument Interface
; Property values:
;   Application (RO) = Exception occurred
;   Blocks (RO) = #<VLA-OBJECT IAcadBlocks 0972bc24>
;   Database (RO) = #<VLA-OBJECT IAcadDatabase 09508ee4>
;   Dictionaries (RO) = #<VLA-OBJECT IAcadDictionaries 0972bbd4>
;   DimStyles (RO) = #<VLA-OBJECT IAcadDimStyles 0972bae4>
;   ElevationModelSpace = 0.0
;   ElevationPaperSpace = 0.0
;   FileDependencies (RO) = #<VLA-OBJECT IAcadFileDependencies 095119b4>
;   Groups (RO) = #<VLA-OBJECT IAcadGroups 0972bb84>
;   Layers (RO) = #<VLA-OBJECT IAcadLayers 0972bcc4>
;   Layouts (RO) = #<VLA-OBJECT IAcadLayouts 0972bd14>
;   Limits = (0.0 0.0 12.0 9.0)
;   Linetypes (RO) = #<VLA-OBJECT IAcadLineTypes 0972bd64>
;   Materials (RO) = #<VLA-OBJECT IAcadMaterials 0972bdb4>
;   ModelSpace (RO) = #<VLA-OBJECT IAcadModelSpace2 0972be04>
;   Name = ""
;   PaperSpace (RO) = #<VLA-OBJECT IAcadPaperSpace2 0972be54>
;   PlotConfigurations (RO) = #<VLA-OBJECT IAcadPlotConfigurations 0972bea4>
;   Preferences (RO) = #<VLA-OBJECT IAcadDatabasePreferences 0951198c>
;   RegisteredApplications (RO) = #<VLA-OBJECT IAcadRegisteredApplications
0972bef4>
;   SectionManager (RO) = Exception occurred
;   SummaryInfo (RO) = #<VLA-OBJECT IAcadSummaryInfo 095119dc>
;   TextStyles (RO) = #<VLA-OBJECT IAcadTextStyles 0972bf44>
;   UserCoordinateSystems (RO) = #<VLA-OBJECT IAcadUCSs 0972bf94>
;   Viewports (RO) = #<VLA-OBJECT IAcadViewports 0972bfe4>
;   Views (RO) = #<VLA-OBJECT IAcadViews 0972c034>
; Methods supported:
;   CopyObjects (3)
;   DxfIn (2)
;   DxfOut (3)
;   HandleToObject (1)
;   ObjectIdToObject (1)
;   Open (2)
;   Save ()
;   SaveAs (2)
```

Note: You can't access the ObjectDBX application itself.

Notice flat list for two 2d points (llx lly urx ury)

New drawing

Indicates Single Document Mode

Notice the lack of any property beginning with Active*, such as ActiveDimStyle, ActiveLayer, ActiveLinetype, etc. Managing object properties is therefore not affected by any of these active objects.

Looking at the dump above, (RO) means the property is read only. Numbers in parentheses indicate the required number of arguments. Compare properties and methods with those of the ActiveDocument.

What can ObjectDBX be used for?

- Scanning and reporting statistics
- Standardizing layer names, text styles, dimension styles etc.
- Batch spell checking.
- Standardizing object properties across a project.
- Updating block definitions in the current drawing from a library.
- Importing styles and standardizing open documents to the ObjectDBX document

Let's Get Specific

Assuming you already have accessed ObjectDBX and saved it in Odoc.

- How to open a drawing
  - (vla-open odoc "filespecification" [read-only])
  - Note: you can't open password protected files with ObjectDBX.
- How to save an opened drawing
  - (vla-save odoc)
- How to save a new drawing.
  - (vla-saveas odoc "[path/]name")
  - File extension is optional
- How to close a drawing (2 methods)
  - (vlax-release-object odoc) ;be sure to save first.
  - (vlax-open odoc "filespecification")
  - You basically have an SDI document interface with each ObjectDBX document.
- Make a copy of a block in the current drawing with a different name.

```
;;;Wrapper for error trap.  Allows
;;;one to try to directly access a
;;;member of a collection that may
;;;not exist without generating an
;;;error.  If misused, this could
;;;cause pernicious bugs.
(defun safe-item (coll key / rtn)
  (if (null
        (vl-catch-all-error-p
          (setq rtn (vl-catch-all-apply
                      'vlax-invoke
                      (list
                        coll
                        'item
                        key
                      )
                    )
          )
        )
      )
    rtn
  )
)
```

```
;;;Redirect block references to
;;;a new block definition.
(defun remapinsertions (old new)
  (vlax-for n (blocks)
    (vlax-for m n
      (if
        (and
          (= "AcDbBlockReference"
             (vla-get-objectname m)
          )
          (= old
             (vla-get-effectivename m))
        )
         (vla-put-name m new)
      ))))
```

- How to copy a block definition to the current drawing. It is complicated in Vlisp to replace block definitions with those stored in other drawings and to refresh existing insertions. Steps are 1) open the source drawing, 2)Temporarily rename the block definition in the activedocument,

3)use copyobjects to transfer the block definition, 4) give block references that have the temporary name the new name, 5)delete the temporary block, and 5) close the source drawing. Safe-item allows the item method to be used where there is a possibility that the reference won't exist. RemapInsertions iterates through the entire block collection replacing every reference to the temporary block with the new block. Notice the nested vlax-for statements. This allows every object in every block to be checked and replaced. Another wrapper, probably unnecessary is blocks which obtains the block collection. GetBlockFrom branches depending on whether the source block exists in the activedocument. By temporarily renaming the existing block definition, it allows the process of replacement to preserve dynamic block information from the source blocks by copying the definitions whole. By using vlax-invoke, the list of objects to be copied can be ordinary rather than a Safearray.

GetBlockFrom is just a start. Improvements could include working with a list of block definitions from the same source drawing, cleaning up attributes, and preserving dynamic block properties.

```
;;GetBlockFrom renames existing
;;blocks, copy block definition from
;;source drawing, remap existing
;;blocks, and clean up.  Does not
;;handle blocks with attributes or
;;preserve dynamic block properties.
(defun GetBlockFrom (source bname /
    blk blks odoc oblks oblk tmpname)
  ;;Get ObjectDBX document
  (setq odoc (ObjectDBXdocument))
  ;;Open Source Drawing as DBX read only
  (vla-open odoc
            (findfile source)
            :vlax-false)
  ;;Get source drawing blocks collection
  (setq oblks (vla-get-blocks odoc))
  ;;Get source block
  (setq oblk (vla-item oblks bname))
  (cond
    ((setq blks (blocks) ;;block exists
           blk  (safe-item blks bname)
     )
     ;;Temporary block name
     (setq tmpname "GetBlockFromTemp")
     ;;Rename blocks old contents
     (vla-put-name blk tmpname)
     ;;Copy block definition in
     (vlax-invoke
       odoc
       'copyobjects
       (list oblk)
       blks
     )
     ;;Refresh all insertions
     (remapinsertions tmpname bname)
     ;;Delete unreference block
     (vla-delete (vla-item blks tmpname))
    )
    (t
     ;;Block will be new
     (vlax-invoke
       odoc
       'copyobjects
       (list oblk)
       blks
     )))
  ;;Close the source drawing
  (vlax-release-object odoc)
)
```

```
;;;Block collection in
;;;Activedocument
(defun Blocks ()
  (vla-get-blocks
    (vla-get-activedocument
      (vlax-get-acad-object)))
 )
```

VLR-Reactors - An Introduction

Detour Ahead – Are you sure reactors are your best route?

Our daily commute is sometimes blocked by traffic snarls. Just so, our programming goals have destinations that must be arrived at on-time and without accidents. Reactors give some choices but AutoCAD has given us some awfully good toll roads to compete with them.

Destinations	Alternatives To Reactors
Automatic layering	Palettes, Design Center, CUI, Toolbar Macros, Command Redefinition.
Custom objects	Dynamic Blocks
Text reporting object properties	Fields
Annotation Scaling	Annotation Features
Automatic Saving	savetime
Keeping things together	Groups
Scheduling	Tables, Data Extraction
Layer Standards	Layer state manager
Managing Title-block Info	Sheet Set manager
Lock Z value	Buy a vertical
Cleanup variables after lisp execution	Pause loops in programs for input
Launch a command by double-clicking on an object	CUI
Bring up a context sensitive menu at right click	CUI

Don't Do That! – Things you shouldn't use reactors for.

Reactors provide some powerful capabilities but they shouldn't be misused.

Inappropriate Reactor Uses	It is Better to
Cancel commands	Alert users and let them cancel
Prevent editing objects	Put on locked layers, Use file permissions, Send DWF or PDF, Train workers
Protect the drawing from changes	Send DWF or PDF, Use file permissions, Archive safe copies, buy Cadlock. Notice I didn't promote passwords

The Cast – Types of Reactors

Reactor Type	Their Specialties
:VLR-AcDb-Reactor	Monitor changes in the database. They are very active and should be used sparingly.
:VLR-Command-Reactor	They monitor commands, when they start, when they end, if they are cancelled and if they are unknown or failed.
:VLR-DeepClone-Reactor	Excels in tracking copied objects
:VLR-DocManager-Reactor	Tracks when documents are opened, destroyed, locked, became current, will be activated, or will be deactivated. When adding entities, it triggers often. It watches when you pop to another application.
:VLR-DWG-Reactor	Tracks when the drawing will be closed, deleted, opened or saved. Whether some events will trigger depends on whether SDI=0 or 1
:VLR-DXF-Reactor	Tracks operations related to dxfin and dxfout
:VLR-Editor-Reactor	A combination of the capabilities of VLR-Command-Reactor, VLR-DWG-Reactor, VLR-Lisp-Reactor, and the VLR-Sysvar reactor. I suggest using the others. You should avoid using both for the same event.
:VLR-Insert-Reactor	Tracks operation related to inserting blocks or drawings. When inserting other drawings, there is a translation matrix involved that controls scaling, translation, and rotation.
:VLR-Linker-Reactor	Tracks when ObjectARX application are loaded and unloaded
:VLR-Lisp-Reactor	Tracks when lisp expressions entered at the command line begin, end, and are cancelled. Useful for cleanup behind lisp commands that change variables and turn over control to AutoCAD commands for prompting.
:VLR-Miscellaneous-Reactor	Tracks when the pickfirst selection set changes and when the layout changes
:VLR-Mouse-Reactor	Tracks double-clicks and right-clicks. Prior to AutoCAD adding these events to the CUI, this was the only way to customize these actions in relation to object types.
:VLR-Object-Reactor	A watch selected Owner objects and notifies when those objects are copied, erased, un-erased, deleted from memory, opened for modification, modified, un-appended, re-appended and unmodified. Formerly, this was the only way LISP programmers had to implement custom objects.
:VLR-Sysvar-Reactor	This is a very active reactor and may trigger multiple times for each command. Can be very useful.
:VLR-Toolbar-Reactor	Tracks when toolbar icon sizes will change or has been changed.
:VLR-Undo-Reactor	Tracks subcommands of the undo command (Auto, Control, Begin, End, Mark, Back, and Number)

:VLR-Wblock-Reactor	Tracks Wblock – when it is about to start, when it is being performed, when it has ended.
:VLR-Window-Reactor	Tracks when Document window or application window is resized or moved. Ignores window tiling operations.
:VLR-XREF-Reactor	Tracks XREF operations, including when they are attached, restored, bound, and unloaded

Put On Your Safety Belt – Expect crashes during testing!

Have you ever ridden with someone you didn't trust? Perhaps they loved driving fast, popped wheelies or such? You're not exactly in the driver's seat with reactors and they can run off the road and take all your work with them. So don't wake up in the recovery room without anything to show? Bank your success by saving everything before testing anything. While preparing for this lecture, I crashed numerous times. Be warned!

It's alive! – How to build a reactor

The reactors listed above are constructed by LISP functions that look very similar to the reactors themselves. The only difference is that the functions don't have the starting colons. All of the functions that build reactors (except those that make object reactors) take 2 arguments.

Generalized Format**: (VLR-xxxx-reactor *Data Callbacks*) returns** *the reactor object <vla-object>*

Object reactor Format: **(VLR-Object-Reactor *Owners Data Callbacks*)** *returns the reactor*

Example: **(VLR-command-reactor nil '((:VLR-commandended . *CleanUpHandler*)))**

Data is a placeholder for anything you want stored with the reactor itself. This eliminates the need to maintain global variables to maintain transitions between events.

Callbacks is a quoted association list that contains dotted pairs in the form (*event . callback*) where callback is the name of a user defined function usually accepting 2 arguments (object reactors 3).

Event is a specially named variable that is bound to its name. Example names are :VLR-CommandWill-Start, :VLR-CommandEnded, or :VLR-ObjectModified. These names are not case sensitive but are written that way to improve readability.

For most reactors the first argument is the calling reactor and the second argument is a parameter list that varies for each reactor type and each event type within that reactor type. The help files are the best source of information when writing the callback functions.

Can I Call You Back?

All reactors require callback functions to work. If the callback function doesn't exist, it generates an error. It's up to the programmer to provide those functions. So the programmer should 1) make sure to define the callback function in the same file that contains the reactor code 2)should define it before the reactor becomes operational, and 3)use a name unique enough to

be reasonably stable. Since callbacks are so important, it might be better to define them in a separate namespace VLX application (beyond the scope of this course).

Generalized format: **(defun AAA_callback (*reactor params*) …..)**

Object reactor callback format: **(defun OBJ_was_modified (*Notifier Reactor Params*)**

Example:
```
(defun CleanUpHandler (reactor params)
  (foreach n (VLR-data reactor) (setvar (car n) (cadr n)))  )
```
If you didn't understand any of that, don't worry. Important things to note are the argument list of the example excludes its use as an object reactor.

> ***Important! Only ActiveX object access is allowed in callbacks. They cannot contain: 1) interactive prompts, 2) dialog box calls, 3) command function calls, 4) entmod or entmake or entget statements.***

The process of calling the callback function is called notification.

Don't Fall In Love with a Crash Test Dummy

Callbacks take time to develop. So speed the development of the overall application by using built-in callback functions initially. Using these dummies will answer the most important question: Will the reactor trigger when it is needed? These ready-to-use callbacks accept a variable number of arguments so they can be used for all kinds of events and for every kind of reactor.

VLR-Trace-Reaction Prints a nice diagnostic message in the trace window of Vlide that describes the calling reactor and the argument list passed to it.

> ***Warning! Set up the Vlide trace window to be visible before using the reactor and then do not click in the trace window. Some reactors like VLR-DocManager-Reactor will lock up AutoCAD if you do.***

VLR-Beep-Reaction Beeps when it's triggered. Often that is just enough to let you know that something is going on and that the reactor type will work or not for what you want to do. When a reaction causes a drum roll, you realize how active reactors can be.

User defined dummies: Not satisfied with the built-in dummies? Write your own. Any defun that accepts the correct number of arguments will work. Use alert boxes and princ statements to let you know what is going on without resorting to the trace window. You can also dummy out any callback by leaving the body empty or using alert or princ statements. Doing this eliminates the need to change things in two places (at the reactor and at the callback) Example: (defun mydummy (react data) (princ react)(princ data)(princ "\n")).

The Owners Club

Objects tracked by object reactors are called Owners. Object reactors continuously monitor those objects and notify the callback functions when selected events occur. Objects keep Object reactors on their toes by notifying them of what is happening to them. Object events that are tracked are deletions, un-deletions, modifications, and copying actions.

Owner management functions

Function Name	Format	Action	Returns
VLR-owner-add	(VLR-owner-add reactor ownerobject)	Adds an owner to the reactor	Ownerobject
VLR-owner-remove	(VLR-owner-remove reactor ownerobject)	Removes an owner from the reactor	Ownerobject
VLR-Owners	(VLR-owners reactor)	Identifies Owners of a reactor.	List of owners or nil

Object Callbacks are powerless over their notifiers: They receive notifications but they cannot change the state of the notifying objects (called notifiers). Callbacks for :VLR-openedForModify cannot even access an object's properties without causing errors. There only power with respect to the notifiers is to pass references to them on to other reactors that have power to change.

They Work Well Together

Since object reactors are helpless they need help from unrelated reactors. Command reactors for instance can be used to revise the object state after the object modification has occurred. The commandended event is generally a safe event during which to modify objects.

Callbacks can disable and enable reactors. They can pass data to other reactors. In order to implement this kind of teamwork, reactor objects need to be stored either in each other's data area or in global variables. Without this kind of inter-process communication, the teamwork of different functions acting together at different times is impossible.

They Can Be Persistent!

Object reactors can be made persistent. Persistent reactors are saved with the drawing. The opposite of persistent is transient. Transient reactors exist only during the current drawing session. Problems with persistent reactors occur when a drawing containing persistent reactors is opened unless the callback functions that they use are loaded early in the startup Relevant functions and their uses are:

VLR-Function	General format	What it does	Returns
VLR-Pers	(VLR-pers reactor)	Makes a reactor persistent	reactor
VLR-Pers-p	(VLR-pers-p reactor)	Determines if a reactor is persistent	T or nil
VLR-pers-release	(VLR-pers-release reactor)	Makes a reactor transient	Reactor or nil

What Triggered That?

Events trigger reactors that monitor those events. The help files identify for each reactor, a list of events that it tracks and a description of what those events are. Below each event table is a callback table that describes the parameter list provided to the callbacks during each event. In some cases, no parametric information is sent. In other cases the information is very important.

Keep Me Informed

Reactors normally fire only for events that relate to the current database. It is possible to have a document level reactor be notified by changes/events occurring in other documents. Experiment with notification settings when working with dummy callbacks to understand the differences.

VLR-notification Determines whether or not a reactor will fire if its associated namespace is not active. The return value is either 'all-documents or 'active-document-only.

VLR-set-notification Defines whether a reactor's callback function will execute if its associated namespace is not active. Format: (VLR-set-notification reactor 'range) where 'range is either 'all-documents or 'active-document-only.

Be careful using reactors that respond to other documents. Each document has its own namespace and variables in one drawing are not available to the reactor being triggered in another document. In fact most lisp code is suspended until a document is reactivated. It is entirely possible that reactors whose notification setting is 'all-documents might continue to execute after their host document is closed.

Who's on First?

When designing your reactor system do not depend on the order of the reactions or on the order of events. When object reactors fire, they generate a callback call for each tracked object and for each tracked event.

Regressive Behavior

Don't forget the importance of handling undo/redo events. If ignored or improperly handled the system will report errors. In the worst case, you can end up with unpredictable events. The object reactors are capable of notifying during undo events. At the very least, you should dummy out the undo callbacks to understand when things happen. Some experts believe that

building a reliable reactor system that accounts for undo/mredo is beyond the capabilities of LISP. Of course if your drafters never change their minds or make mistakes its unimportant.

You Are Dismissed

It is often wise to disable events during a callback function to the possibility of endless loop behavior. You don't want a situation where a reactor callback triggers the calling reactor. More generally, assume that there might be other reactors that should be temporarily turned off.

Function	Format	Use	Returns
VLR-remove	(VLR-remove reactor)	Disables this reactor	Reactor (a vla-object)
VLR-remove-all	(VLR-remove-all)	Disables all reactors	Reactor list

Get Them Back

Function	Format	Use	Returns
VLR-add	(VLR-add reactor)	Re-enables this reactor	Reactor (a vla-object)
VLR-added-p	(VLR-added-p reactor)	Is this reactor enabled?	T or Nil

Too Much to Do

Reactors, if used sparingly can be a blessing. It is possible however to completely shut down the system with reactor calls. Be sure to turn reactors on and off when you need them.

Send for the Understudy!

Sometimes you need a substitute for a callback. This could be an alternative to enabling and disabling a reactor. To do it, use (VLR-reaction-set *reactor event function*)

Conversely if you need to know the names of the reactions associated with a reactor, use

(VLR-reactions reactor)

To get all the reaction names for a particular reactor-type use (VLR-reaction-names *reactor-type*)

Alternatively, since this is lisp, a program can just change the definition of the callback function by using setq. (setq mycallback <newcallbackfunction>). Mycallback is now different.

Be Sure to Thank the Little People

These functions are important support functions. VLR-data and VLR-data-set avoid the use of global variables in reactor communications.

Function	Calling Format	Why use?	Returns
VLR-data	(VLR-data-reactor)	To access data stored in reactor	Expression or nil
VLR-data-set		To store data in a reactor	Expression or nil
VLR-type	(VLR-type reactor)	Returns a symbol representing the reactor type.	Symbol
VLR-types	(VLR-types)	Returns a list of symbols representing all reactor types	List

Tablets of Stone - Reactor Use Guidelines

1. Don't rely on the sequence of reactor notifications. A single command triggers multiple events. Yet the order of those events cannot be relied on.
2. Don't rely on the sequence of function calls between notifications. If two :VLR-erased notifications occur on separate objects, the order of the notifications does not indicate which was erased first.
3. Don't use interactive functions in callbacks: getpoint, getstring, get…, entsel, etc. will not work. Don't launch a dialog box. This is interactive
4. Don't try to change an object that issued the event notification. You can read properties of objects but not change them.
5. Avoid actions in callbacks that trigger the same event. Such actions may cause an infinite loop. Example: opening a drawing within a BeginOpen event.
6. Verify that a reactor is not already set before setting it, or you may end up with multiple callbacks on the same event. This is so basic but so important. See code examples for good practice.
7. No events will be fired while AutoCAD is displaying a modal dialog.
8. You cannot use the command function in callbacks
9. You cannot use entget or entmod inside callbacks. Use ActiveX instead.

Where's the Show?

Let's put it all together. Following is a function that will quickly demonstrate the reactors and give them a chance to strut their stuff. See the show at AU 2007 or try it yourself at work.

```
;;;test reactors
;;;Due to the possiblity of overloading the system
;;;the first statement disables all previous reactors
;;;D. C. Broad, Jr.  10/28/2007

;;Automatically create test reactions for a particular
;;reactor type.
(defun testreact  (rtype)
  (vlr-remove-all);dump other reactors first
  (eval
    (list
      (read
        (substr
          (vl-symbol-name
            rtype)
          2))
      nil
      (quote
        (mapcar
          '(lambda (x)
             (cons x 'callback))
          (vlr-reaction-names rtype)
          )))))

;;Stop all reactors
(defun c:stop () (vlr-remove-all))
;;Default to beeps
(setq callback vlr-beep-reaction)
;;Beep reactions
(defun c:beeps()(setq callback vlr-beep-reaction))
;;Trace reactions
(defun c:traces()(setq callback vlr-trace-reaction))
;;List all reactions
(defun c:reactions()
  (mapcar
    '(lambda (x)
       (vlr-reactions
         (cadr x)))
    (vlr-reactors)))
```

This code will automatically create dummy reactors for every event that the specified reactor has. It can quickly identify opportunities to take advantage of the reactor system.

In conclusion, reactors can be useful but should only be considered if other built-in offerings of AutoCAD are missing the capabilities you desire.

My main uses for reactors have been 1)to serve as cleanup functions for Lisp implemented commands that seek to auto layer and 2) to modify textsize, dimscale, and ltscale as part of a custom annotation management system (This use should now be obsolete as well).

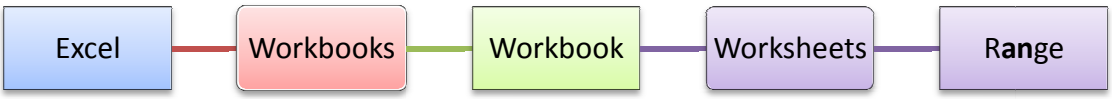
Accessing Other Applications

To access an application object other than AutoCAD, use:

(setq MyApp (vlax-get-or-create-object prog-id))

In the context of Microsoft Excel, this becomes:

(setq XLApp (vla-get-or-create-object “Excel.Application”))



The Excel Object Model Hierarchy

Excel has such good developer documentation that it is easy to prototype the application in VBA and then to translate into Vlisp. The need for such custom access to Excel however has greatly diminished with recent advancements in tables and in data extraction. In fact, it’s difficult to imagine a task involving Excel that AutoCAD itself can’t do well. The following sample illustrates VBA vs. LISP methods to access a value in a single cell named range in an Excel Workbook. The initial setqs above defun are for testing and debugging as illustrated in class.

Sample VBA Code:

```
Sub test()
Dim wbk As Workbook
Dim sht As Worksheet
Dim rng As Range
Dim val As Integer
Set wbk = Excel.Workbooks("test.xlsx")
Set sht = wbk.Sheets("Sheet1")
Set rng = sht.Range("livingroomarea")
val = rng.Value
Debug.Print val
End Sub
```

LISP Code Translation

```
;;Vars above defun for testing
(setq FileSpec "c:\\test.xlsx
 (setq ShtName "Sheet1")
(setq RngName "LivingRoomArea")
(defun c:GetVALue (FileSpec ShtName
RngName /   RNG SHT SHTS VAL WBK
WBKS XL)
  (setq XL (vlax-get-or-create-object
"Excel.Application"))
  (setq WBKS (vlax-get-property XL 'workbooks))
  (setq WBK (vlax-invoke WBKS 'open FileSpec))
  (setq SHTS (vlax-get-property WBK 'worksheets))
  (setq SHT (vlax-get-property SHTS 'item ShtName))
  (setq RNG (vlax-get-property SHT 'RngName
RngName))
  (setq VAL (vlax-get RNG 'value))
  (princ VAL)
  (mapcar 'vlax-release-object (list RNG SHT SHTS))
  (Vlax-invoke WBK 'close) ;Steps of closing
  (vlax-invoke XL 'quit)
  (vlax-release-object XL) )
```

Some Visual Lisp Errors

1. Attempting to get properties and methods not applicable to an vla-object
2. Using vla-item, vlax-for, or vlax-map-collection on a vla-object that is not a collection.
3. Using list methods on safearrays.
4. Attempting to access erased objects.
5. Attempting to access closed documents.
6. Not accounting for the possibility that an object type might change (circle to arc once broken), (polyline to lines once exploded)
7. Not releasing objects
8. Attempting to delete referenced objects.

Ways to avoid errors

1. Verify a method or property is applicable before trying to use it or access it.

```
(vlax-method-applicable-p object)
(vlax-property-available-p object)
```

2. Classify objects by type and direct the appropriate actions by conditional branching:

```
(vlax-for n *modelspace
  (setq on (vla-get-objectname n))
  (cond
    ((= on "AcDbLine")
     ;;do line stuff
     )
    ((= on "AcDbArc")
     ;;do arc stuff
     )
    (t ;;otherwise do this
     )))
```

3. Exclude objects which do not apply by filtering:

```
(if (ssget "x" '((410 . "Model")(0 . "Line")))
  (vlax-for n (vla-get-activeselectionset
                (vla-get-activedocument
                  (vlax-get-acad-object)))
    ;;do stuff with lines
    ))
```

4. Mistaking objects for collections can be avoided by looking at the object model, by using help files, by dumping object properties and by checking whether a count property or an item method is applicable.

5. Check to see if objects are referenced prior to deleting them. Since this is difficult to do using vla-methods, trap the inevitable error. See next section.

6. Verify data type or use function to ensure results.

```
;;takes a variety of input and makes it a list
(defun makelist (input)
  (cond
    ((= (type input) 'variant)
     (makelist (variant-value input)))
    ((= (type input) 'list)
     input)
    ((= (type input) 'safearray)
     (vlax-safearray->list input))
    ((listp input) input)
    (t (list input))))
```

   Using library catchall conversion routines is generally bad practice. Know what is coming and use the built-in functions to work with the data.

7. To avoid errors caused by accessing erased objects, either test to see if they were erased or don't keep the objects long term. (vlax-erased-p object) can be used to test before accessing the potentially erased object.

A Way to Ignore Errors:

Warning! The technique demonstrated in this section is dangerous and may hide unexpected bugs. The function is useful, however, since it can simplify processes when ignoring "expected" errors is harmless. Feel free and guiltless to use them.

```
(vl-catch-all-apply
                '(lambda (x) (vla-put-layer x newlayer))
                objectlist
              )
```

The vl-catch-all-apply process above is poorly written mistaken code. Mistakes are:

1. It assumes that vl-catch-all-apply is like mapcar rather than apply.
2. It does not save the results of the vl-catch-all-apply operation. So there will be no future opportunity to test the error.

```
(setq results (vl-catch-all-apply 'vla-put-layer (list object newlayer)))

(if (vl-catch-all-error-p results)
  (princ (vl-catch-all-error-message results))
;;remain quiet if successful
)
```

The second try above improves the process. The code will report errors. Errors it ignores, however are numerous and include: object doesn't exist, bad argument type, object on locked layer, layer doesn't exist….

It is better to check beforehand that a layer exists, that an object is not erased (if stored at an earlier time), and that a layer is unlocked. If the programmer is willing to forgo knowing the errors, then vl-catch-all-apply can simplify. It is similar to the VBA method "On Error, resume next"

Prior to using (vl-catch-all-apply...), thoroughly test by using (apply…) instead. The errors will be noticeable and silly coding mistakes can be discovered. Once the vl-catch-all-apply wraps problem code, it hides blunders.

Error handlers

Each susceptible user defined function should contain an error handler that cleans up after itself and reports the errors.

```
(defun MakeThingsBetter (/ oce clayer *error*)
  ;;save variables to be changed during function
  (setq oce    (getvar "cmdecho")
        clayer (getvar "clayer")
  )
  (defun *error* (msg)
    (princ msg)
    (setvar "cmdecho" oce)
    (setvar "clayer" clayer)
    (princ)
  )

  ;;do the rest of the code
  (*error*)
  ;;end quietly, restoring everything
)
```

In MakeThingsBetter, see above, system variables are saved first (stored locally) and an error handler is set up to clean up after itself at the end. *Error* serves a dual purpose of exiting gracefully and serving as a cleanup routine normally. Since the ***error*** function is stored locally, it does not interfere with user defined error handlers in other functions.

Regular Errors affect Visual Lisp Reliability

1. Bad parenthesis matching. Vlide usually can catch it but not always. Double click on each parenthesis to see the enclosing code. Check each step.

2. Data type problem affect regular expressions:

   a. (/ 2 3) = 0 (a hidden error if unexpected)

   b. (strcat 1 "this") doesn't work

3. Lists aren't always easy to traverse. Step and check. Test theory with real lists.

Releasing objects – Our Final Topic

In general, programs should release the objects bound to variables by that program.

Exceptions:

1. Symbols bound to the AutoCAD application object and the activedocument.
2. Objects not bound to variables do not need to be released.
3. Local variables and loop variables do not need to be released. The garbage collector is smart enough to recover space and to reset pointers when local variables go out of scope.

How to release one object:

(vlax-release-object object)

How to release a list of objects

(mapcar 'vlax-release-object *objectlist*)

or

(foreach n *objectlist* (vlax-release-object n))

When setting up an *error* handler, use something similar to this:

```
(defun *error* (msg)
  (if msg (princ msg))
  (mapcar '(lambda (x)(if x (vlax-release-object x))) (list dbx blks doc aco)))
```

The code above checks to see if the variable is bound to a value before attempting to release it. An attempt to release an object not bound to an object will generate an unnecessary error.

When in doubt, release the object!

In conclusion: 1) Use VLIDE, 2) Keep the Help files open, 3)Program for single line testing, 4)Dump and examine objects, 5) Don't program before you explore current AutoCAD features, 6) Comment thoroughly, 7) Use error handlers ,8) Clean-up after execution, 9) Be careful using reactors, and 10) Command and DXF methods have their place. Don't expect Vlisp to do everything.

Thank you for attending this Class! Hope you've enjoyed it. Please don't forget to fill out your feedback report.