November 30 – December 3, 2004 ◊ Las Vegas, Nevada

All You Need to Know About Good Programming You Learned in Kindergarten

Phil Kreiker – Looking Glass Microproducts

CP32-2 Learn how to apply some simple concepts to produce friendly, courteous AutoLISP® programs. It isn't hard to write good programs; it's just a little more work than you expect. All you need to know about good programming you learned in kindergarten. These concepts include cleaning up after yourself; not taking things that don't belong to you; and putting things away when you're done with them. To maximize your benefits from this class, you should have an interest in programming AutoCAD® and a dislike of unfriendly, discourteous programming and programmers.

About the Speaker:
Phil is Megabyte Master at Looking Glass Microproducts, an Autodesk® Registered Developer, where he is responsible for AutoCAD® training, customization, and support. He taught AutoCAD at the Colorado School of Mines from 1998 to 2003. Phil wrote the —CAD Cookbook— column for 10 years for CADalyst, and was technical editor-at-large for CADENCE magazine from 1996 to 1997. He is the author of several books including his latest Visual LISP®: A Guide to Artful Programming which is part of the Programmer's Series from Autodesk® Press. He received his Bachelor of Engineering degree from Cooper Union for the Advancement of Science and Art and his M.S. degree from Lowell Technological Institute.
phil@lookingglassmicro.com

There are two ways to write error-free programs; only the third one works.

What is an Error?

Before we can start writing error-free programs, we must first define what we mean as an error.

> **A software error occurs when the software does not perform according to its specifications**

This definition assumes that the specifications are correct. This is rarely, if ever, a valid assumption

Consider the Ballistic Missile Early Warning System (BMEWS), a complex of warning and tracking radars based at sites in Alaska, Greenland, and the United Kingdom. This equipment can detect a missile as far as 4800 km away and provide a 15-minute warning of an attack on North America.

According to the initial program specification, any object detected by radar whose trajectory does not correspond to a scheduled international flight must be a missile. All went well until the first time the moon came over the horizon.

It is easier to change the specification to fit the program than vice versa.

> **A software error occurs when the software does not perform according to its specifications, providing that it is used within its design limits**

This definition ignores the necessity that if a program is accidentally used beyond its design limits, it must exhibit some reasonable behavior.

Consider an Air Traffic Control system designed to handle 100 airplanes simultaneously. It is unacceptable for the system to simply drop plane number 1 when plane number 101 appears.

Open the pod bay door, HAL.

> **A software error occurs when the software does not perform according to the official documentation supplied to the user**

This definition ignores the possibility that both the software and the documentation could be in error, and ignores the fact that user documents tend to describe only the expected use of the software. Your system shouldn't crash if you double-click on a digitizer button.

Every program has (at least) two purposes:
The one for which it was written, and another for which it wasn't.

This leaves us with only one workable definition for a software error:

> **A software error occurs when the software does not do what the users reasonably expect it to do**

In order to design a successful system, the software developer must always understand what the users "reasonably expect."

What do users expect?

- They expect the system not to lock-up or reboot, or unexpectedly return to the C:> prompt
- They expect to be given a list of choices (if they ask)
- They expect commands to be case insensitive
- They don't expect to see nil.
- Users do NOT expect

```
; error: bad argument type: numberp: nil
```

Testing for Errors

Programmers are not to be measured by their ingenuity and their logic, but by the completeness of their case analyses

Your assignment:

Test a program that determines if the three numbers you give it describe an equilateral, an isosceles, or a scalene triangle.

You test the following inputs:

2,2,2

3,4,5

1,2,2

2,1,2

2,2,1.

But did you test these?

1

-2,-2,-2

3,4,8

BUFFALO BUFFALO BUFFALO BUFFALO

QUIT

HELP

!@&^*%%##.

Creating Courteous Programs

Everything you need to know about programming, you learned in kindergarten.

Put things back the way you found them

The code that follows restores selected system variables when the command either finished normally or is cancelled.

```
;;; Save System Variables

(defun pushvars (vlist / pair name)
   (foreach pair vlist
      (setq name (strcase (car pair) T))
      (setq
         SYSVARS (cons
                     (cons name (getvar name))
                     SYSVARS
                 )
      )
      (If (cdr pair)
         (Setvar name (cdr pair))
```

```
      )
   )
)

;;; Restore System Variables

(defun popvars (/ pair)
   (foreach pair SYSVARS
      (setvar (car pair) (cdr pair))
   )
   (setq *error* old_error)
   (setq SYSVARS nil)
)


;;; Error Handler

(defun error (s)
   (if (not (member s
            '("function cancelled"
              "console break")))
      (princ (strcat "\n; error: " s "\n"))
   )
   (setvar "cmdecho" 0)
   (command "._undo" "_end")
   (if undoit
      (progn
         (prompt "\nUndoing...")
         (command "._undo" 1)
         (setq undoit nil)))
   (command "._undo" "_auto" "_on")
   (popvars)
   (vl-exit-with-value (princ))
)


;;; program skeleton

(defun c:fubar (/ old_error sysvars undoit)
  (setq
     old_error *error*
     *error*   error
   )
   (pushvars
      '(("cmdecho" . 0)
        other system variables to save
       )
   )
   (command "._undo" "_auto" "_off")
   (command "._undo" "_group")
   (fubar)
   (command "._undo" "_end")
   (command "._undo" "_auto" "_on")
   (popvars)
   (princ)
)
```

The code that follows restores system variables as expected for user input.

```
;;; Restore one system variable

(defun restore (varname / old-value)
   (if (setq
          old-value (cdr
                      (assoc
                        (strcase varname t)
                        sysvars
                      )
                    )
       )
      (setvar varname old-value)
   )
)
```

Here's an example of an input loop using restore:

```
;;; Input loop

(while again
   (restore "blipmode")
   (restore "osmode")
   (setq p0 (getpoint "\nTo point: "))
   (if p0
      (setvar "lastpoint" p0)
   )
   (setvar "osmode" 0)
   (setvar "blipmode" 0)
   do something
)
```

Don't take things that don't belong to you

- Make all your local functions and variables local to your command.
- Document your your non-local functions and variables, and give them unique names:

  LGM_PUSHVARS

  LGM_POPVARS.

Ask Politely and say Thank you

- Emulate the AutoCAD command prompts
- Allow for defaults and remember previous responses
- Validate and filter inputs
- Highlight selected entities
- Honor noun-verb object selection
- Create a 'Previous' selection set and LASTPOINT.
- Emulate the AutoCAD Command prompts
- Start each prompt on a new line and end with a space

```
(GetString "\nLayer name: ")
```

- Find and imitate a command similar to the one you're writing

  A STAR command should look and feel like the POLYGON command.
- Allow for defaults and remember previous inputs

```
;;; getint with default

 (defun xgetint (prmpt default)
   (cond
      ((getint
          (strcat prmpt " <" (itoa default) ">: ")
       )
      )
      (default)
   )
)
```

```
;;; Example

(initget 7)  ; no null, no zero, no negative
(setq star:npoints
  (xgetint "\nNumber of points: " star:npoints)
)
```

```
;;; getkword with default

 (defun xgetkword (prmpt default)
   (cond
      ((getkword
          (strcat prmpt " <" default ">: ")
       )
      )
      (default)
   )
)
```

```
;;; getstring with default

(defun xgetstring (cr prmpt default / temp)
   (if (= default "")
      (setq default ".")
   )
   (setq
      temp (getstring
              cr
              (strcat prmpt " <" default ">: ")
           )
   )
   (if (= default ".")
      (setq default "")
   )
   (cond ((= temp ".") "")((= temp "") default) (temp))
)
```

```
;;; getreal with default

(defun xgetreal (prmpt default)
   (cond
      ((getreal
          (strcat prmpt " <" (rtos default 2 4) ">: ")
       )
      )
      (default)
   )
)
```

```
;;; getdist with default

(defun xgetdist (pnt prmpt default / msg)
   (setq msg (strcat prmpt " <" (rtos default) ">: "))
   (cond
      ((if pnt
           (getdist pnt msg)
           (getdist msg)
       )
      )
      (default)
   )
)
```

- Validate and Filter Inputs

```
;;; prompt for a line

(while (and
          (setq
              esel (entsel "\nselect first line: ")
          )
          (/= "LINE"
             (cdr (assoc 0 (entget (car esel))))
          )
       )
   (prompt "\nNot a line."))
)
```

- Remember What You Did Last Time

If you're going to use defaults, you must decide where to store them.

Location	Persistence	Scope
In Local Variable	While command is in progress	Current Command
In Global Variables	Until application is reloaded	Current Editing Session
In the Drawing	Until changed	Current Drawing
In the Environment	Until changed	Current User on Workstation

In the Registry	Until changed	Current User on Workstation
In the CFG file	Until changed	All users on Workstation

Local Variables

Your defaults are maintained only while the command is active. The defaults revert each time you evoke your command.

```
;;; Variables and Defaults

(setq params '((an_int . 1) (a_real . 3.14) (a_string . "")))


;;; Local Variables

(defun c:fubar (/ an_int a_real a_string)
  (mapcar 'set-default params)
  ;; ...
)


;;; set-default

(defun set-default (pair)
  (if (null (vl-symbol-value (car pair)))
    (set (car pair) (cdr pair))
  )
)
```

Global Variables

Your defaults are maintained in the current editing session. The defaults are maintained between commands.

```
;;; Global Variables

(defun c:fubar ()
  (mapcar 'set-default params)
  ;; ...
)
```

In the Drawing

Your defaults are remembered in the drawing file, and are maintained between editing sessions for that drawing.

```
;;; In the drawing

(defun c:fubar (/ dict)
  (setq dict "fubar")
  (mapcar 'ldata-get defaults)
  ;; ...
  (mapcar 'ldata-put defaults)
)


;;; ldata-get
```

```
(defun ldata-get (pair)
  (vlax-ldata-get dict (vl-symbol-name (car pair)) (cdr pair))
)

;;; ldata-put

(defun ldata-put (pair)
  (vlax-ldata-put
    dict
    (vl-symbol-name (car pair))
    (vl-symbol-value (car pair))
  )
)
```

In the Environment

Your defaults are remembered for each user, and apply for all editing sessions.

```
;;; environment

(defun c:fubar (/ dict)
  (setq dict "fubar")
  (mapcar 'env-get params)
  ;; ..
  (mapcar 'env-put params)
)


;;; env-get

(defun env-get (pair / key value valuetype)
  (setq key (strcat dict ":" (vl-symbol-name (car pair))))
  (setq value (getenv key))
  (setq valuetype (type (cdr pair)))
  (cond
    ((null value) (setq value (cdr pair)))
    ((equal 'str valuetype))
    ((setq value (read value)))
  )
  (set (car pair) value)
)

;;; env-put

(defun env-put (pair / key value valuetype)
  (setq key (strcat dict ":" (vl-symbol-name (car pair))))
  (setq value (vl-symbol-value (car pair)))
  (setq valuetype (type value))
  (cond
    ((equal 'int valuetype)
     (setq value (itoa value))
    )
    ((equal 'real valuetype)
     (setq value (rtos value 2 8))
    )
  )
  (setenv key value)
```

```
)
```

In the Registry

Your defaults are remembered for each user, and apply for all editing sessions.

```
;;; Registry

(defun c:fubar (/ dict)
  (setq dict
         "HKEY_CURRENT_USER\\Software\\Looking Glass Microproducts\\fubar"
  )
  (mapcar 'reg-get params)
  ;; ..
  (mapcar 'reg-put params)
)

;;; reg-get

(defun reg-get (pair / key value valuetype)
  (setq key (vl-symbol-name (car pair)))
  (setq value (vl-registry-read dict key))
  (if (null value)
    (setq value (cdr pair))
  )
  (set (car pair) value)
)


;;; reg-put

(defun reg-put (pair / key value valuetype)
  (setq key (vl-symbol-name (car pair)))
  (setq value (vl-symbol-value (car pair)))
  (vl-registry-write dict key value)
)
```

In the CFG file

Your defaults are remembered for all users, and apply for all editing sessions.

```
;;; CFG

(defun c:fubar (/ dict)
  (setq dict "fubar")
  (mapcar 'cfg-get params)
  ;; ..
  (mapcar 'cfg-put params)
)


;;; cfg-get

(defun cfg-get (pair / key value valuetype)
  (setq key (strcat "AppData/" dict "/" (vl-symbol-name (car pair))))
  (setq value (getcfg key))
  (setq valuetype (type (cdr pair)))
```

```
    (cond
      ((null value) (setq value (cdr pair)))
      ((equal 'str valuetype))
      ((setq value (read value)))
    )
    (set (car pair) value)
  )

  ;;; cfg-put

  (defun cfg-put (pair / key value valuetype)
    (setq key (strcat "AppData/" dict "/" (vl-symbol-name (car pair))))
    (setq value (vl-symbol-value (car pair)))
    (setq valuetype (type value))
    (cond
      ((equal 'int valuetype)
       (setq value (itoa value))
      )
      ((equal 'real valuetype)
       (setq value (rtos value 2 8))
      )
    )
    (setcfg key value)
  )
```

- Highlight/Dehighlight Selected Entities

```
(defun sshighlight (ss mode / n)
   (setq n (if ss (sslength ss) 0))
   (while (> n 0)
      (setq n (1- n))
      (redraw (ssname ss n) mode)
   )
)
```

- Create a 'previous' selection set and lastpoint

```
(defun c:fubar ()
  ;; body of command
  ...
  (setvar "highlight" 0)
  (command "._select" ss "")
  (setvar "lastpoint" p)
  (command "._undo" "_end")
  (command "._undo" "_auto" "_on")
  (popvars)
  (princ)
)
```

- Check your Spelling

  If there is a spelling error in a user prompt, there is at least one error in the program

  There usually is another.

Why are there so few courteous programs on the market today?

My theory is that it goes against the grain of modern education to teach children to program, since it requires making plans, acquiring discipline, organizing thoughts, devoting attention to details, and learning to be self-critical.