Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.3.1

CLOS dynamics and

CLOS Performance

Some CLOS Review

- Methods allow customizing behavior to classes.
- You can define primary methods:
```
(defmethod foo ((arg1 class1) (arg2 class2)…) …)
```
- You can define :before, :after, and :around methods. This define a before method:
```
(defmethod foo :before
   ((arg1 class1) (arg2 class2)…) …)
```

Method types

- All before methods run first, most-specific first. Before methods are typically used for argument checking and setup
- Then the most-specific primary method runs (call-next-method will run next-most-specific primary method).
- Then all after methods run, least-specific first. After methods do cleanup, recording, etc.

Around methods

- Around methods wrap around all other methods. The most-specific :around method runs. Addition :around methods are run and the before/primary/after method run only if (call-next-method) is used. :around methods allow inserting logic that will determine whether methods run at all.

Methods allow any level of complexity

- Mixing before, after, primary, and around methods appropriately, along with defining subclasses and superclasses appropriately, programmers have great flexibility in determining behavior.

CLOS is dynamic 1

- Methods (even new generic functions) can be defined at any time, even while and application is running.
- Methods and modifiable aspects of generic functions can be redefined at any time.
- Note that if methods named **foo** are defined

  ```
  (fmakunbound 'foo)
  ```

  destroys all methods and the generic function (in case you want to start over).

CLOS is dynamic 2

- You can change the class of an instance at any time:

```
cg-user(7): (defclass c1 ()
              ((x :initform 10 :initarg :x)
               (y :initform 55 :initarg :x)
               (z :initform 99 :initarg :z)))
#<standard-class c1>
cg-user(8): (defclass c2 ()
              ((a :initform "slota" :initarg :a)
               (y :initform "sloty" :initarg :y)
              (c :initform "slotc" :initarg :c)))
#<standard-class c2>
```

Changing class of an instance 2

```
cg-user(9): (setq ins (make-instance 'c1))
#<c1 @ #x20b6762a>
cg-user(10): (list (slot-value ins 'x)
                   (slot-value ins 'y)
                   (slot-value ins 'z))
(10 55 99)
cg-user(11): (change-class ins 'c2 :a 777)
#<c2 @ #x20b6762a>
cg-user(12): (list (slot-value ins 'a)
                   (slot-value ins 'y)
                   (slot-value ins 'c))
(777 55 "slotc")
cg-user(13):
```

Performance Considerations

- How does one decide when to use:
  - Generic function versus function
  - Standard-class, structure-class, built-in-class
- Appropriate comparisons require looking at
  - Required functionality in your program
  - Availability of functionality in different styles
  - Cost of functionality in different styles

Measuring Performance

```
(defun try ()
  (time
   (dotimes (I 100)
     (make-instance 'point :x 1 :y 1) )))
(compile 'try)
```

- Choose a number of iterations such that the loop takes at least a second
- Calculate iterations per second
- Replace make-instance with other things

Make-instance Performance

- The compiler does a lot at compile time to optimize make-instance
- Avoid creating instances if you don’t have to because
  - it increases the size of the process and may cause needless paging
  - it increases the frequency and duration of garbage collection

Instance Creation

- Make-instance supports
  - Multiple inheritance and initarg defaulting, making it very flexible
  - Automatic combination of initialization protocols
  - Layered protocols
- Making a struct is similar to making a vector
  - structs are smaller and can be created faster than class instances

Performance Comparison

```
(defstruct ship1
  (x-position 0)
  (y-position 0))

(defclass ship2 ()
  ((x-position :initform 0 :initarg :x-position)
   (y-position :initform 0 :initarg :y-position)))

;; 24 bytes per ship1.
;; (make-ship1) 200,000 per second
(defun try1 (n)
  (time (dotimes (i n) (make-ship1))))

;; 48 bytes per ship2
;; (make-instance 'ship2) 200,000 per second,
;; except for the first one.
(defun try2 (n)
  (time (dotimes (i n) (make-instance 'ship2))))
```

Unoptimized Make-instance

```
;; (make-instance class-name) 20,000 per second
(defun try3 (n class-name)
  (time (dotimes (i n) (make-instance class-name))))
```

Slot-value Performance

- Slot-value can be as fast as two arefs
  - First aref looks up position of slot value in instance vector
  - Second aref accesses the instance vector
- Standard slot reader, writer, and accessors are nearly as fast as slot-value

Slot-value Vs. Structure Accessors

- Standard-class slot-value supports:
  - Multiple inheritance
  - Class redefinition
  - Change-class
  - Error-checking (type, bound)
  - Specialization in metaclass (slot-value-using-class)
- Structure accessors do not support these

Slot-value Vs. Structure Accessors

- Defstruct accessors compile in-line
  - Single memory reference
- Slot-value is optimized when object type can be inferred
  - But not completely in-lined (instance-read-1)
  - 2 memory references
- Unoptimized slot-value is quite slow

Performance Comparison

```
;; 2.2 million struct accesses per second
(defun try4 (n)
  (let ((ship (make-ship1)))
    (Time (dotimes (i n) (setq *junk* (ship1-x-position ship))))))

;; 700,000 slot accesses per second
(defun try5 (n)
  (let ((ship (make-instance 'ship2)))
    (Time (dotimes (i n) (setq *junk* (slot-value ship 'x-position))))))
```

Standard-class Slot Accessors

- Slot accessor functions are generic functions
  - Additional methods can be defined
  - Performance is similar to slot-value when there are no additional methods
- Defstruct accessor functions get compiled in-line
  - Callers must be recompiled if the struct is redefined

Accessor Performance

```
(defclass buick ()
  ((color :initform :red :accessor buick-color)))

(defun try7 (n)
  (let ((buick (make-instance 'buick)))
    (Time (dotimes (i n)
            (setq *junk* (buick-color buick))))))
```

Generic Function Call

```
(defun seize (lock)
  (etypecase lock
    (simple-lock . . .)
    (null-lock . . .)))

versus

(defmethod seize ((lock simple-lock)) . . .)
(defmethod seize ((lock null-lock)) . . .)
```

- Should be roughly equivalent except for the first call to SEIZE.

Performance Comparison

```
(defun invert (value) (not value))

;; 3.5 million calls per second
(defun try8 (n)
  (time (dotimes (i n) (setq *junk* (invert t)))))

(defmethod nada (value) (not value))

;; 3.5 million calls per second
(defun try9 (n)
  (time (dotimes (i n) (setq *junk* (nada t)))))

(defmethod flip ((value ship2)) (not value))
(defmethod flip ((value buick)) (not value))

;; 1.8 million calls per second
(defun try10 (n)
  (let ((ship (make-instance 'ship2)))
    (time (dotimes (i n) (setq *junk* (flip ship))))))
```

Generic Function Call

- Suppose one represented objects using sequences, with symbols for type codes

```
(defun seize (list-lock)
  (if (listp list-lock)
      (ecase (car list-lock)
        (simple-lock . . .)
        (null-lock . . .))
      (error  . . .)))
```

- Slower than either typecase or GF dispatch

Avoid Keyword and Optional Arguments

- Function calling and method dispatch is much slower
- Much more work at run time to analyze the argument list

Performance Comparison

```
(defmethod negate ((value ship2) &key arg) (not value))
(defmethod negate ((value buick) &key arg) (not value))

;; 300,000 calls per second
(defun try10 (n)
  (let ((ship (make-instance 'ship2)))
    (time (dotimes (i n) (setq *junk* (negate ship))))))

;; Remove keyword processing to increase performance by
;; a factor of 6x.
```

Method Dispatch

- Method dispatch is usually slow the first time (unless there is only one method)
- For a given set of argument types, the applicable methods are cached
- CLOS generally builds the dispatch incrementally, making for a slow start

Multiple Dispatch

```
(defmethod add1 ((a fixnum) n) (+ a n))
(defmethod add1 ((a single-float) n) (+ a n))

;; 1.1 million per second
(defun try11 (n)
  (time (dotimes (i n) (setq *junk* (add1 2 1)))))

(defmethod add2 ((a fixnum) (n fixnum)) (+ a n))
(defmethod add2 ((a single-float) (n single-float)) (+ a n))

;; 625 thousand per second
(defun try12 (n)
  (time (dotimes (i n) (setq *junk* (add2 2 1)))))
```

Performance - Use Defstruct Rather Than Defclass

- In performance critical code you are better off without objects and the overhead of message dispatch.
- Only do this if metering determines it is necessary.

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.3.2

Garbage Collection

Why Have Garbage Collection

"explicit memory management has proved to be a fruitful source of bugs, crashes, memory leaks and poor performance."

Java Language Specification

"When (not if) garbage collection becomes available, we will have two ways of writing C++ programs."

Bjarne Stroustrup

What is garbage collection?

- Automatic reclamation of no longer needed memory
- An object can be freed if no live object points to it
- Objects are not freed immediately

Lisp Spaces

- Lisp heap space is divided into two parts
  - Newspace - where new objects live
  - Oldspace - where old objects live
- Newspace is managed with scavenges
  - Newspace is divided into two halves, and only one half is in use at any time
  - When that half gets full, all live objects are copied to the other half and packed consecutively
    - garbage is left behind
  - Scavenges are normally fast and not noticeable

Tenuring to Oldspace

- The "age" of an object is tracked by counting the number of scavenges it survives
  - called its "generation"
- When an object reaches a certain age, it is moved to oldspace
  - called "tenuring" the object
- Oldspace is not garbage collected very often
  - objects that survive a while are likely to survive a long while, perhaps forever

Garbage in Oldspace

- Managed by a Global GC
- Only performed occasionally, but all other processing and signal handling stops
- Duration is noticeable
- Frequency and behavior of a global GC can be controlled
  - it must be controlled when responding to events in real time, such as with an equipment controller
- (Frequency and behavior of the scavenger cannot be controlled)

Oldspace and Newspace Grow

- When oldspace gets full, an additional oldspace segment is carved out of newspace
  - many oldspaces
- When newspace is full, it may
  - grow incrementally by asking for a large block of memory from virtual memory
  - become oldspace and create a whole new newspace

Information on Memory Spaces

```
USER(3): (room)
area  address(bytes)        cons          symbols          other bytes
                        8 bytes each  24 bytes each
                        (free:used)   (free:used)       (free:used)
Top #x205e0000
New #x204c0000(1179648)   710:3366      239:15           986496:100392
New #x203a0000(1179648)    -----         -----               -----
Old #x20000d58(3797672)   815:54211     231:14007       2122272:883616

;;One newspace, one oldspace
```

Recommendation

- Tweaking the garbage collector settings should be done as a last resort
- The best solution is to limit garbage creation
- Interactive programs can trigger a Global GC at convenient times to improve response times

Triggering a Scavenge

- Triggering a scavenge
  - (excl:gc)
- Finding out when scavenges happen
  - (setf (sys:gsgc-switch :print) t)
- Determining scavenge efficiency
  - (setf (sys:gsgc-switch :stats) t)
  - Efficiency should typically be at least 75%
    - less than 25% of your time is spent scavenging

Triggering a Global GC

- Triggering a global gc
  - (excl:gc t)
- If :print and :stats switches are true (previous slide), you can see how many bytes are tenured

Basic Control

- Excl:*global-gc-behavior*
  - :auto - gc automatically after exceeding threshold (excl:*tenured-bytes-limit*)
  - :warn - warning only after exceeding threshold
  - :auto-and-warn - warning and gc after exceeding threshold
  - nil - no warnings and no gc
  - (300 2.0) - gc after
    - threshold is exceeded and lisp is idle 300 seconds
    - 2.0 times threshold is exceeded

Advanced Control

- `(setq excl:*tenured-bytes-limit* 10000000)`
  - Automatic global gc after 10 Mb (default is 5)
  - Don't set it below 1 Mb
- `(setf (excl:gcgc-parameter :generation-spread) 10)`
  - objects that survive 10 generations are tenured
  - default is 4
- `(setf (excl:gsgc-switch :gc-old-before-expand) t)`
  - before expanding oldspace, do a dynamic gc in case the expansion is not necessary

Advanced Control

- When building a lisp application (excl:build-lisp-image)
  - you can set the initial size of oldspace and newspace
- Use excl:resize-areas to restructure oldspaces and newspaces in a running image

Immediate Tenuring

- (excl:tenuring (excl:load-system 'macsyma))
- All objects within the scope of forms will go straight into oldspace
- Avoids work for the scavenger

Weak-Vector

- (excl:weak-vector length)
  - Creates a vector
  - The GC will "collect" elements that have no other references (element becomes NIL)
- (excl:schedule-finalization x function)
  - GC applies function to x at the point x is identified as garbage
- Use together to verify that the GC is collecting large objects you think should be collected

GC Errors

- STORAGE-CONDITION
  - A type of error condition is signalled when Lisp cannot get more memory from the operating system
  - Normally, lisp does not immediately exit, since there is usually a bit of space left in newspace
  - Only real solution is to add virtual memory or kill other processes competing for virtual memory

GC Errors

- Corrupted memory will cause the GC to signal a fatal error and exit lisp immediately
- There is no recovery
- Typically caused by
  - bugs in a foreign function interface
  - highly optimized code whose type declarations were violated

The GC and Foreign Code

- Pointers to Lisp objects can be passed to foreign code (C, C++, Fortran, …)
- BUT if the GC runs, the object may move, invalidating the pointer
- Typical symptom is a SEGV in foreign code that is otherwise working

The GC and Foreign Code

- Remedies:
  - Prevent the GC from running when calling foreign code
    - Define foreign function with :release-heap :never option
  - Allocate objects in static space
    - using keyword arguments to ff:allocate-fobject
    - using the function excl:make-static-array

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.3.3

Conditions and Error Handling

Unavoidable Error Conditions

- Not all errors are bugs, for example database is down or permission to save file is denied
- Production code must keep the user out of the debugger
- Trapping for errors and providing users with options for continuation or recovery is much better

Defensive Programming

- Assume you (and your peers) will make programming errors
- Write functions that can recognize and handle bad input
  - Prefer etypecase over typecase and ecase over case when the set of cases is fixed
  - Use ASSERT- and CHECK-TYPE-like forms at key points where performance is not critical

Signaling Errors

- Use the function ERROR
  - (error “connection to server ~A is down” server)
- Use the function CERROR
  - (cerror “continue anyway” “connection to server ~A is down” server)
- Use the function SIGNAL
  - First argument names a class, the rest of the argument list is used in a call to make-instance
  - (signal ‘network-down :type :LAN)

Signaling Errors

- Use the function CHECK-TYPE or ASSERT

```
(defun nth-character (n s)
  (check-type s string)
  (check-type n fixnum)
  (assert (<= 0 n (1- (length s))))
  (char s 0))
```

Signaling Errors

- (break)
- Put it in your source code temporarily
- Causes you to explicitly go directly into the debugger, without being intercepted by error handlers

Signaling Warnings

- (warn “You are running out of table space”)
- Ordinarily, prints the message to *error-output* and returns NIL
- Used for “benign” or
- (setq *break-on-signals* t)
  - In this case, WARN behaves like BREAK

Error Conditions are Objects

- Different errors have different types
- Represented as a CLOS object
- Error classes defined using DEFCONDITION
- Error handlers usually apply to specific error classes
- Handlers on the class ERROR apply to all error conditions

Ignoring Errors

- Easy to understand the concept but only sometimes a good idea

```
(defun safe-division (a b)
  (ignore-errors (/ a b)))
(safe-division 4 2)
   2
(safe-division 4 0)
   NIL
   #<DIVISION-BY-ZERO @ #x204d9562>
```

More on ignoring errors

```
(defun safe-division (a b)
  (if (zerop b) nil (/ a b)))
```

Ignore-errors useful when doing something useful but not necessary, and in the code fragment (read-user-init-file not defined here)

```
(multiple-value-bind (a b)
  (ignore-errors (read-user-init-file))
  (if b (format t "Problem reading init file, ~
            Skipped.~%")))
```

Error Handling

```
(defun slope (x1 y1 x2 y2)
  (/ (- y2 y1) (- x2 x1)))
(defun print-slope (x1 y1 x2 y2)
  (handler-case
      (print (slope x1 y1 x2 y2))
    (division-by-zero ()
      (print :infinite))
    (error (c)
      (princ c))))

(print-slope 0 5 0 10) -> :infinite
(print-slope 0 5 NIL NIL)  -> “nil is not a number”
```

Example User-Defined Condition

```
(defcondition network-down (error)
  ((network-type :initarg :type))
  (:report
    (lambda (condition stream)
       (format stream
               “The ~A network is down”
               (slot-value condition
                           ‘network-type)))))
(defun check-network ()
  (or (test-local-area-network)
      (signal ‘network-down :type :LAN)))
```

Condition class

- errors are a subclass of condition
- you can signal a condition
- each signal results in a make-instance of one object of the particular error or condition class
- system will search for a handler for that condition
- handler of last resort is the debugger

“Serious” conditions

- The class ERROR is a subclass of SERIOUS-CONDITION
- there are other conditions which are error-like called serious-conditions
- ignore-errors specifically doesn’t ignore them
- example: stack overflow, or running out of virtual memory

Handling Errors Sometimes

```
(defun database-connect (name)
  (loop
   (catch :retry
    (handler-bind ((network-down #'fix-net))
      (return
        (open-database (find-database name)))))))

(defun fix-net (condition)
  ;; If this function ever returns,
  ;; you get debugger.  This happens if
  ;; the network is not fixable.
  (when (network-is-fixable)
      (fix-network)
      (throw :retry)))
```

handler-bind vs. handler-case

- handler in handler-bind run in context of error
  - can decline to handle the condition
  - OR can fix things up and continue
- handler in handler-case runs when stack already unwound
  - In this case, the handler always applies, it cannot decline to handle the condition

restarts

- A restart establishes a means to recover from error conditions
- You see them as “continue” options when you land in the debugger.
- Restarts differ from error conditions in that
  - Invoking a restart explicitly transfers control to another part of the program, as if doing a THROW
  - Corrective action or recovery is implicit in the act of invoking a restart

With-simple-restart

```
(with-simple-restart (abort "Close Connection")
  (process-header socket (read-header socket)))
```

- First arg is symbol naming the restart
- Second arg is documentation string

- ABORT is a standard restart, but you can have others of your choosing

Invoking Restarts

```
(defun abort-if-abortable (value)
  (when (find-restart ‘abort)
    (invoke-restart ‘abort value)))
```

- This is similar to the lisp function ABORT, except ABORT signals an error if there is no restart named ABORT.
- “Simple” restarts are exactly like throw and catch except that you can test for them with FIND-RESTART.

restart-case

- establishes a context in which one or more restarts are active
- (restart-case <form> <restarts>)
- anonymous restarts (name is NIL) can only be taken by debugger
- named restarts -- handler can call find-restart, invoke-restart

debugger-hook

- Global variable, normally its value is NIL
- You can set it to point to a function of two arguments
  - First arg receives an error condition
  - Second arg is value of *debugger-hook*
- Immediately prior to landing in the debugger, this function is called
  - You might use it to pop up a menu of restarts to the user (a “menu debugger”) rather than letting the user land in the full-blown debugger

Allegro CL Certification Program

Lisp Programming Series Level 2

Session 2.3.4

Interface Development with IDE

IDE - Interface Dev. Env.

- Graphical user interface for developing user interfaces
- You create windows, dialog boxes, menu bars with the user interface
- You use a text editor to add methods that customize the behavior of your application

Franz Inc.

CG - Common Graphics

- A package of functions for creating windows and controls, drawing graphics, and receiving and generating events
- Available only on Windows

Projects

- Collects all files associated with an application
- Contains dialogs and other windows that are part of the user interface
- Dialogs and other windows are designed with forms
- Create a New Project (File | New Project)
- Project Manager displays information about a project

Forms

- Add a form with File | New Form

You are asked what kind of window. Choices (initially bitmap-window, frame-window, dialog, text-edit-window, etc.) incllude new window classes you have created.

- Always create your own subclasses of windows so your methods affect your windows only

When you have a new form

- Double-click to inspect it with an inspector window
- Typically change the class, name, and title
- Also menu, scrollbars, background- and foreground-colors

Typical window class definition

```
(defclass paint-buffer (frame-with-single-child)
  ())

(defclass paint-pane (bitmap-pane)
  ((ischanged :initform nil :accessor buffer-ischanged)
   (file :initform nil :accessor buffer-file)
   (objects :initform nil :accessor buffer-objects)
   (selections :initform nil :accessor buffer-selections)
   (mouse-x :initform 0 :accessor mouse-x)
   (mouse-y :initform 0 :accessor mouse-y)))

;;; Associate the two
(defmethod default-pane-class ((obj paint-buffer))
  'paint-pane)
```

Windows Messages

- Operating system sends your application messages via SendMessage()
  - Key press or button click
  - Window needs redisplay
  - Color palette has changed
- Common Graphics translates most messages into generic function calls that you can hook into

Redisplay-Window

```
(defmethod redisplay-window ((pane paint-pane)
                              &optional box)
  ;; This method is called automatically whenever the
  ;; window needs to be redisplayed.
  (cg:erase-contents-box pane
     (or box (cg:page-box pane)))
  (dolist (object (buffer-objects pane))
    (draw-object object pane))
   (dolist (object (buffer-selections pane))
    (highlight-object object pane)))
```

Some Event Handler Functions

- cg:redisplay-window, cg:erase-window
- cg:resize-window, cg:move-window
- cg:mouse-left-down, cg:mouse-left-up, etc.
- cg:virtual-key-down
- cg:user-close
- cg:mouse-moved
- Must first create your own window subclass to specialize on
- You can call them yourself

User-Close

```
(defmethod user-close ((window paint-pane))
  (if (not (buffer-ischanged window))
      (progn (call-next-method)
        (user-close (parent window)))
    (case (pop-up-message-dialog window "Close"
           "The file has changed. Save the changes?"
           warning-icon "Yes" "No" "Cancel")
      (1 ;; Save
       (call-next-method)
       (when (buffer-ischanged window)
          (user-save-file window))
       (user-close (parent window)))
      (2 ;; Discard
       (call-next-method)
       (user-close (parent window)))
      (3 ;; Cancel
       nil))))
```

Common Dialog Boxes

- Pop-up-message-dialog
  - Yes/No/Cancel type dialogs
  - Warning/abort/info icons
  - One to four buttons
- Ask-user-for-new-pathname
  - Used for Save As type commands
- Ask-user-for-existing-pathname
  - Used for Open type commands

invalidate

- Call “invalidate” to force a window or component to redisplay
- Invalidate calls redisplay-window
- Don’t call redisplay-window directly

Intro To Standard Widgets

- Drag and Drop “GUI builder”
- Use “events” to add behavior

Standard Widgets, cont’d

- For the most part, properties like size and position can only be set during the design stage
- You can (setf RANGE) and (setf VALUE) at run time. Use cg:find-component to get the appropriate object.

Find-component

- Given a window or dialog box, finds the component having a certain name
- Relies on the “name” property of a component
- The name property should be a symbol
- Tip: if you rename your component, you have to update the name in the various calls to find-component.

Component Events

- Double-click on a widget to bring up its property sheet
- Select the "Events" tab
- Select the event of interest (on-click, on-change, on-mouse-in, etc.)
- Note that it writes an empty event handler for you
- Events naming convention:
  - formname-gadgetname-eventname
  - form1-checkbox1-on-change
  - on-change args: (widget newvalue oldvalue)
- on-change and on-click are the main ones to worry about

Some notes on the grid-widget

- The grid-widget is a complex table or spreadsheet widget written wholly in Lisp (does not correspond to a standard Windows widget)
- It is very powerful and thus very complex
- There are examples in CG Example set (click on Help | CG Examples
- New documentation sent to you

Must make appropriate subclasses

- Step one is making your own subclasses of all relevant grid-widget classes (such as grid-widget, grid-row, grid-column, etc.)
- grid is divided into sections (blocks of rows and columns). Each section is customizable.
- A cell is the intersections of a row and column. It is not an individual lisp object. Instead, methods specialized to the row and column define behavior for the cell.

Rows and columns

- Rows and columns are known collectively as subsections.
- Grid sections and subsections can be customized with many properties (resizable, border-color, scroll-bars, movable, deletable, etc.)

Displaying data

- Read-cell-value gets the value for a cell (whatever kind of lisp object it is). The cell is identified by its row and column.
- Draw-cell displays data in a cell. It is called automatically whenever the cell is uncovered or invalidated. You write methods for it but do not call it directly (you call invalidate-cell instead, for example).
- Default draw-cell is princ-to-string.

Responses to events

- Cell-click methods respond to mouse clicks.
- Write-cell-value modifies a value in a cell.

Higher level functions

- Could provide data-object method for each row and data-reader method for each column (row represents an employee and column represents some aspect, like hire date or salary). Default read-sell-value calls data-reader.

Grid-widget examples

Three examples provided:

- Simple color editor
- Replicated editable-text columns
- Complex employee example

Some notes on AllegroServe

- AllegroServe is a web application server that allows you to create and publish web pages
- Its has two components: a web server and a html generator
- You can create pages dynamically, using the html generator to create a page based on current data

Dynamic vs. static pages

- A static page has content which changes rarely and in any case does not depend on current data or user input (welcome pages, product listings, other static data listing)
- A dynamic page is generated in real time based on current information and user data (search results, shopping carts)

Allegro Webactions adds more dynamic capability

- Webactions allows using special html tags which trigger running Lisp functions
- This allows a html designer to call functions written separately by a programmer

AllegroServe and Webactions documentation and tutorials

- Available on Franz Inc. website (www.franz.com/support/documentation/7.0/doc/introduction.htm, search for aserve and webactions)
- Also available for 6.2

training@franz.com

http://www.franz.com.lab/