A Guided Tour of the Common Lisp Interface Manager

Ramana Rao <rao@xerox.com>
Xerox Palo Alto Research Center

William M. York <york@ila.com>
International Lisp Associates, Inc.

Dennis Doughty <doughty@ila.com>
International Lisp Associates, Inc.

Abstract

The Common Lisp Interface Manager (CLIM) is a powerful Lisp-based system that provides a layered set of facilities for building user interfaces. These facilities include a portable layer called Silica that includes basic windowing, input, output services, and mechanisms for constructing window types and user interface components; stream-oriented input and output facilities extended with presentations and context sensitive input similar to the work pioneered in the Genera UI system; and a gadget-oriented toolkit similar to those found in the X world extended with support for look and feel adaptiveness. In this article, we present an overview of CLIM's broad range of functionality and present a series of examples that illustrates CLIM's power.

Introduction

Common Lisp is a language standard that has provided a broad range of functionality, and that has, to a large degree, successfully enabled the writing of truly portable Lisp programs. The emergence of CLOS and the cleanup efforts of Ansi X3J13 have further enhanced the utility and portability of Common Lisp. However, one major stumbling block remains in the path of those endeavoring to write large portable applications. The Common Lisp community has not yet provided a standard interface for implementing user interfaces beyond the most basic operations based on stream reading and printing.

The Common Lisp Interface Manager (CLIM) addresses this problem by specifying an interface to a broad range of services necessary or useful for developing graphical user interfaces. These services include low level facilities like geometry, graphics, event-oriented input, and windowing; intermediate level facilities like support for Common Lisp stream operations, output recording, and advanced output formatting; and high level facilities like context sensitive input, an adaptive toolkit, and an application building framework.

CLIM will eventually support a large number of window environments including Genera[16], X[12], the Macintosh[1], Microsoft Windows[5], SunView[13], and NextStep[6]. CLIM is designed to exploit the functionality provided by the host environment to the degree that it makes sense. For example, CLIM top level windows are typically mapped onto host windows, and input and output operations are ultimately performed by host window system code. Another example is that CLIM supports the incorporation of toolkits written in other languages (e.g. C-based toolkits that implement Motif[7] or OpenLook[15]). However, in both cases, a uniform interface provided by CLIM allows Lisp applications to deal only with Lisp objects and functions regardless of their operating platform (i.e. combination of Lisp system, host computer, and host window environment).

An important goal that has guided the design of CLIM has been to layer the specification into a number of distinct facilities. Furthermore, the specification doesn't distinguish the use of a facility by higher level CLIM facilities from its use by CLIM users. For example, the geometry substrate, which includes transformations and regions, is designed for efficient use by the graphics and windowing substrates as well as by CLIM users. This means that, in general, a CLIM user can reimplement higher level CLIM facilities using the interfaces provided by lower level facilities.

This modular, layered design has a number of benefits. The CLIM architecture balances the goal of ease of use on one hand, and the goal of versatility on the other. High level facilities allow programmers to build portable user interfaces quickly, whereas lower level facilities provide a useful platform for building toolkits or frameworks that better support the spe-

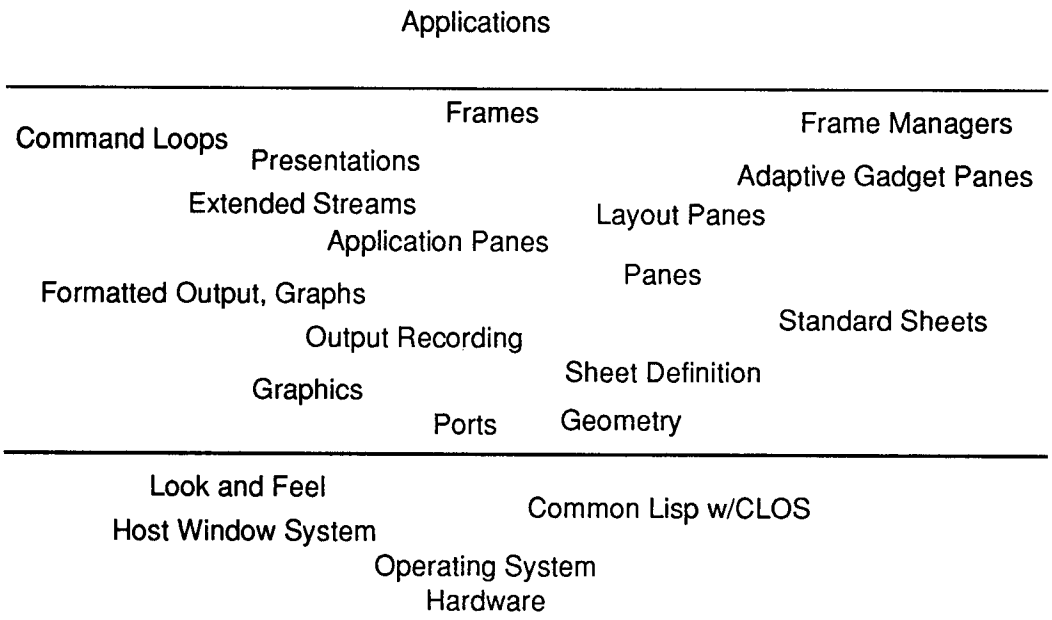

Figure 1: An Overview of CLIM functionality

cific needs or requirements of a particular application.

For example, CLIM's application framework and adaptive toolkit allow users to develop applications that automatically adopt the look and feel of the host's environment. (We often call this "adaptiveness," "look and feel independence," or occasionally more picturesquely, "chameleon look and feel".) However, many users may need or want to define a particular look and feel that stays constant across all host environments (we call this "portable look and feel"). Such users can circumvent the look and feel adaptiveness provided by CLIM, while still using most of the application framework facility and other high level CLIM facilities like context sensitive input. Furthermore, using the lower level facilities of CLIM, they can develop portable toolkit libraries that define and implement their own particular look and feel.

Another major benefit of CLIM's modular design is that application programs need only load CLIM facilities that they use. This is particularly important because of the extent of functionality provided by CLIM. For example, some applications may not use some CLIM facility, say presentations or output recording, and in such cases, these facilities need not be loaded. Similarly, other applications may not need all gadget implementations or any Lisp-based gadget implementations, since they are running in an environment in which C-based gadgets have been incorporated.

The next section presents an overview of the functionality provided by CLIM. The rest of paper describes Silica, application building functionality, and many of the high level facilities of CLIM in greater detail. These sections present a series of real code examples that gradually build up higher and higher CLIM concepts.

1 Overview of Functionality

Figure 1 shows the various aspects of a host environment in which CLIM lives as well as the various elements provided by CLIM. Below we briefly describe a number of CLIM's areas of functionality. Later sections will illustrate many of these components.

Geometry CLIM provides points, rectangles, and transformations; and functions for manipulating these object types.

Graphics CLIM provides a portable interface to a broad set of graphics functions for drawing complex geometric shapes. The CLIM drawing model supports a wide variety of drawing options (such as line thickness), a sophisticated inking model, color, and full affine transformations, which allows graphics to be arbitrarily translated, rotated, and scaled.

Silica CLIM provides a portable layer called Silica for implementing *sheets* (windows and other window-like objects). Silica specifies a uniform interface for creating and managing hierarchies

of sheets, regardless of their class or other properties. In addition, Silica defines a standard sheet class that implements a straightforward window that supports much of the functionality of an X[12] or NeWS[14] window.

Common Lisp Streams CLIM integrates the Common Lisp Stream I/O functionality with the CLIM graphics, windowing, and panes facilities.

Output Recording CLIM provides a facility for capturing all output done to a sheet and automatically repainting it when necessary. In addition, this facility provides a substrate for other CLIM facilities including formatted output and presentations.

Formatted Output CLIM provides a set of high-level macros that enable programs to produce neatly formatted tabular and graphical displays easily.

Presentations CLIM provides the ability to associate semantics with output, such that Lisp objects may be retrieved later via user gestures (e.g. mouse clicks) on their displayed representation. This *context sensitive input* is modularly layered on top of the output recording facility and is integrated with the Common Lisp type system. A mechanism for type coercion is also included, providing the basis for powerful user interfaces.

Application Building CLIM provides a set of tools for defining *application frames*. These tools allow the programmer to specify all aspects of an application's user interface, including pane layout, interaction style, look and feel, and command menus and/or menu bars. In addition, the application-building tools provide high-level facilities for linking user gestures with application commands.

Panes CLIM provides panes which are analogous to the gadgets or widgets of toolkits like the X toolkit[9] or InterViews[3]. Supported pane types include layout panes for arranging other panes, gadget panes for presenting users with feedback information or mechanisms for invoking application behavior, and application panes for displaying and allowing users to interact with application data.

Look and Feel Adaptiveness CLIM supports look and feel independence by specifying a set of abstract gadget pane protocols. These protocols define a gadget in terms of its function and not in terms of the details of its appearance or operation. Application that use these gadget types and related facilities will automatically adapt to use whatever toolkit is available and appropriate for the host environment. In addition, portable Lisp-based implementations of the abstract gadget pane protocols are provided.

2 Silica

The Silica layer of CLIM serves two primary functions. First, it provides a portable windowing model that insulates higher levels of CLIM and CLIM users from the details of the host window system. So from the perspective of most CLIM user's, Silica is the window system. However, on most platforms, Silica will use the services of a host window system to provide efficient windowing, input and output facilities. In this regard, Silica can be viewed as a portable CLX (the Lisp interface to the X11 protocol).

Second, Silica provides an extensible framework that can be used to explore alternative window system implementations. For example, two other Common Lisp window systems interfaces have been build within the Silica framework: the Deli Window System[8] and Common Windows[2].

Moreover, the extensible framework also allows sharing and reusing of functionality typically associated with window systems in a broader range of contexts. Silica allows uniform treatment of many of the objects provided by CLIM including the following:

- windows like those in X or NeWS that provide a surface or canvas for output
- lightweight gadgets typical of toolkit layers
- structured graphics like output records and application presentation objects
- objects that act as Lisp handles for windows or gadgets implemented in a different language (e.g. OpenLook gadgets implemented in C).

The central abstraction specified by Silica is the *sheet*. A sheet can be viewed either as a surface that can be painted on and to which input gestures can be directed or as a displayable object that lives in a hierarchy of other such objects.

The fundamental notion in the Silica window system model is the nesting of sheets within another sheet called a *windowing relationship*. In a windowing relationship, one sheet called the *parent* provides space to or groups a number of other sheets called

```
(defun test-silica ()
  (let* ((sheet
          (make-standard-sheet :parent (find-graft :orientation :graphics)
                               :transformation (make-translation 100 100)
                               :region (make-rectangle 0 0 200 200 )))
         (w (bounding-rectangle-width sheet))
         (h (bounding-rectangle-height sheet))
         (x1 (/ w 4))
         (y1 (/ h 4))
         (x2 (* 3 (/ w 4)))
         (y2 (* 3 (/ h 4))))
    (enable-sheet sheet t)                                          ; Draw This
    (draw-line sheet (make-point x1 y1) (make-point x1 y2))         ;  ------
    (draw-line sheet (make-point x1 y2) (make-point x2 y2))         ;  |\  /|
    (draw-line sheet (make-poiny x2 y2) (make-point x2 y1))         ;  | \/ |
    (draw-line sheet (make-point x2 y1) (make-point x1 y1))         ;  | /\ |
    (draw-line* sheet x1 y1 x2 y2)                                  ;  |/  \|
    (draw-line* sheet x1 y2 x2 y1)                                  ;  ------
    (medium-force-output sheet)
    (sleep 1)
    (disable-sheet sheet)
    (disown-child (sheet-parent sheet) sheet)))
```

Figure 2: This program illustrates the use of standard sheets, the simple sheet type provided by Silica.

children. The Silica sheet protocols specify functionality for constructing, using, and managing hierarchies of sheets. Silica sheets have the following properties:

- coordinate system—provides the ability to refer to locations in a sheet's abstract plane.
- region—defines an area within a sheet's coordinate system that indicates the area of interest within the plane that is a clipping region for output and input.
- parent—a sheet that is the parent in a windowing relationship in which this sheet is a child.[1]
- transformation—determines how points in this sheet's coordinate system are mapped into points in its parent.
- enabled flag—determines whether the sheet is currently considered to be actively participating in the windowing relationship with its parent and siblings.
- children—a set of sheets that are each a child in a windowing relationship in which this sheet is a parent.

Sheet hierarchies are displayed and manipulated on particular host window systems by establishing a connection to that window system and "attaching" them to an appropriate place in that window system's window hierarchy. *Ports* and *grafts* provide the functionality for managing this process. A *port* is a connection to a display service that is responsible for managing host window system resources and for processing input events received from the host window system.

A *graft* is a special kind of sheet that stands in for a host window, typically a root window (i.e. screen level). A sheet is attached to a particular host window system by making it a child of an associated graft. A host window will be allocated for that sheet; the sheet will then appear to be a child of the window associated with the graft.[2]

Silica provides a standard sheet type that provides the basic capabilities of lightweight windows like those provided by X11 or NeWS. The example

[1] Support for multiple parents (i.e. participation in multiple windowing relationships as a child) is currently an open design issue.

[2] For some host servers, particularly X, arbitrary points in the Silica hierarchy can be *mirrored* by host window objects. This allows interoperating with foreign applications and toolkits that rely on the existence of host windows.

program in Figure 2 illustrates the straightforward construction and use of a standard sheet.

This code starts by making a standard sheet that is 200 pixels wide by 200 pixels high and positioned at (100, 100) on the graft. The `:orientation` argument to `find-graft` in this call indicates a request for a graft whose origin is at its lower left corner and whose x coordinates increase to the right and whose y coordinates increase up.

The sheet is then enabled, thus making it visible on the screen associated with the default graft. The second argument indicates that the call shouldn't return until the host window indicates that the window has actually been "exposed".

Then six lines are drawn along the sides of a rectangle and through its diagonals. The two different calls to the line drawing functionality are equivalent, except for the arguments they take. CLIM provides "spread" and "unspread" versions of many geometry and graphics functions as a convenience to users. In addition, a number of functions have two different versions, one that returns a structured object (e.g. a point or rectangle), and another that returns multiple values. The CLIM convention is to indicate the spread argument and the multiple return value versions of a function by annotating the name with an asterisk.

`medium-force-output` is called to force the graphics output to the host window system in case it is buffered. A medium implements a graphics protocol, and in particular, the CLIM graphics protocol. A standard sheet automatically has a CLIM medium associated with it that is forwarded the graphics functions invoked on it. By separating the medium from sheets, it is possible for the medium classes to be host system specific, while sheet classes are host system independent. A user can extract a sheet's medium using `sheet-medium` for efficiency, when a large number of graphics calls are going to be made.

Finally after the output is forced out, the sheet is disabled and disowned, thus freeing all host window system resources allocated for the sheet.

This example illustrates the use of one sheet class. Most CLIM users need not concern themselves with the details of how sheet classes are defined. However, a number of the higher level CLIM facilities define sheet classes. These include the panes provided by the CLIM toolkit, output records, presentations, and special sheets that act as lisp handles to host toolkit objects. All of these higher level objects obey the sheet protocols so that their hierarchical, repaint, and input aspects can be treated uniformly.

Since many users may like to integrate their own objects into the Silica hierarchy, and sheet class definition is one area in which CLOS is used interestingly in CLIM, this issue will be discussed in a forthcoming paper[10].

3 Building Applications

In this section, we explain a number of the necessary elements for building applications in CLIM in greater detail, including frames, frame managers, panes, and simple commands. We will also illustrate these elements in a series of three examples: a simple application that paints a picture and reacts to a mouse button event, a color editor that has an adaptive look and feel, and a simple line and text drawing program.

Application Frames Frames are the central abstraction defined by the CLIM interface for presenting an application's user interface. Many of the high level features and facilities for application building provided by CLIM can be conveniently accessed through the frame facility. The examples in this section and the next will illustrate many of these facilities, including CLIM toolkit panes, look and feel adaptiveness, and generic command loops.

To build a user interface, an application programmer defines one or more frames classes. These frame classes define a number of frame properties including application specific state and a hierarchy of panes (i.e. user interface gadgets and regions for interacting with the users). Frame classes also provide a hook customizing application behavior during various portions of the frame protocol. For example, an `:after` method on generic functions in the frame protocol can allow applications to manage application resources when the frame is made visible on some display server.

Frame Managers Frames are typically displayed as top level windows on a desktop. Frame managers provide the machinery for realizing frames on particular host window systems. A frame manager acts as an mediator between the frame and what is typically called a desktop manager, or in X terminology, a window manager. One important function provided by a frame manager is to adapt a frame to the look and feel of the host window manager.

CLIM defines a number of generic functions for manipulating frames. Analogously to the Silica layer, where a sheet can be connected to a host window system by invoking `adopt-child` on a graft and that sheet, frames can be connected to a host window system by invoking `adopt-frame` on a frame manager and that frame. Then, `enable-frame` can be used

to make the frame visible. The `launch-frame` function packages together all the functionality needed to bring up a frame on a host window system.

Panes An application specifies a frame's content by supplying code that constructs a hierarchy of panes. CLIM panes are rectangular sheets that are analogous to the gadgets or widgets of other toolkits. Applications builders can construct a pane hierarchy using pane classes from a library of standard panes or by defining and using their own pane classes.

A frame's pane hierarchy is generated when it is adopted. The frame manager is responsible for attaching the pane hierarchy of a frame to an appropriate place in a Silica hierarchy (and therefore to a host window system window hierarchy) when the frame is adopted. The pane tree may or may not be attached directly to a graft, since the frame manager may choose to interpose its own sheets or panes to provide (or represent) additional desktop user interface gadgets and regions.

CLIM supplies three basic categories of panes: application panes, layout panes, and gadget panes. *Application panes* are panes that can be used to present application specific data and manage user interactions with this data. CLIM provides a number of application pane types that provide access to the CLIM graphics, input, and stream facilities. These include a basic pane that supports the CLIM graphics protocol and low level input services, a basic stream pane that supports the Common Lisp stream I/O functions, and an extended stream pane that supports the CLIM extended stream protocol.

Layout panes are composite panes that arrange their children according to the space requirements of their children and some composition rule. For example, CLIM provides two pane classes, `hbox-pane` and `vbox-pane`, that lay their children out in a horizontal row or a vertical column respectively.

Gadget panes are panes that implement common toolkit components like push buttons or sliders. Each gadget pane class has a set of associated generic functions which serve the role of callbacks in "traditional" toolkits. For example, a pushbutton has an "activate" callback function which is invoked when its button is "pressed."

Commands Most applications have a set of operations that can be invoked by the user. In CLIM, the *command* facility is used to define these operations. Commands support the goal of separation of an application's user interface from its underlying functionality. In particular, commands separate the notion of an operation from the details of how the operation is invoked by the user.

Application programmers define a command for each operation that they choose to export as an explicit user entry point. A command is defined to have a name and a set of zero or more operands, or *arguments*. These commands can then be invoked using a variety of interaction techniques. For example, commands can be invoked from menu, keyboard accelerators, direct typein, mouse clicks on application data, or gadgets. More sophisticated command reading facilities are discussed later.

3.1 A Simple Frame

In this first application building example, we illustrate how a user can define a frame class, and use an application pane for presenting information and interacting with the user. In the code in Figure 3, we define an application frame using `define-application-frame`. The syntax of this macro is the same as for `defclass`. It defines a new frame class which automatically inherits from the class `frame`. This CLIM class provides most of the functionality for manipulating frames.

Only one requirement must be fulfilled by all frame definitions: code to generate a pane hierarchy must be supplied. The `:pane` option is the simplest way to supply this code. The hello frame constructs a hierarchy with only one application pane. The arguments to the `make-instance` of the pane indicate that the pane should preferably be 200 pixels wide by 200 pixels high, but that it can stretch in either direction to absorb all available extra space.

The layout protocol supported by CLIM is somewhat similar to those of TEXor InterViews[3]. A request for space allocation is specified as a preferred size (i.e. hs, vs), an acceptable amount of extra space (i.e. hs+, vs+), an acceptable amount of space deficit (i.e. hs-, vs-). The special constant `+fill+`[3] is used to indicate a tolerance for an infinite amount of extra space or space deficit.

The `:settings` option is used to specified the default title of a hello frame. *Settings* are a general mechanism for specifying arguments to a frame that can be easily overriden either by the application at frame construction time or by the user with a preference file.

The example defines a specialized application pane using `define-application-pane`. Pane classes defined using `define-application-pane` inherit from the basic application pane class as a default. The

[3] CLIM follows the convention of surrounding constants with pluses.

```
(define-application-frame hello-frame ()
    ()
  (:pane (make-instance 'hello-data-pane
                        :hs 200 :hs+ +fill+ :vs 200 :vs+ +fill+))
  (:settings :title "Hello from Lisp"))

(define-application-pane hello-data-pane () ;; inherits basic-clim-pane
                                            ;; by default
    ())

;; Behavior defined via CLOS class specialization
(defmethod handle-repaint ((pane hello-data-pane) region &key &allow-other-keys)
  (declare (ignore region))
  (let* ((w (bounding-rectangle-width pane))
         (h (bounding-rectangle-height pane)))

    ;; Blank the pane out
    (draw-rectangle* pane 0 0 w h :filled t :ink (pane-background pane))

    ;; Center the label
    (draw-text* pane "Hello" (floor w 2) (floor h 2)
                :align-x :center :align-y :center)))

(defmethod button-release ((pane hello-data-pane) (button-name (eql :right))
                           &key x y &allow-other-keys)
  (draw-point* pane x y))
```

Figure 3: A simple frame implementation.

basic application pane supports graphics operations and invokes generic functions on the pane when input events are recieved.

Defining a special application pane allows us to providing repaint and input methods for the pane. We provide a `handle-repaint` method which clears the pane and prints "Hello" centered in the pane.

Finally, we provide a `button-release` method that draws a point at places in the pane where the right mouse button is released. This method illustrates one place where CLIM uses `eql` specializers to allow applications to attach application specific behavior conveniently.

To run the code in this example, a user need only execute `(launch-frame 'hello-frame)`. This interface hides the complications involved in finding a frame manager, adopting the frame into the frame manager, and enabling it.

3.2 Adaptive Look and Feel

In this next example, we define a frame which can be used to edit colors by manipulating their red, green, and blue components separately. This example illustrates the use of other kinds of panes provided by CLIM. In particular, we will illustrate the use of gadget panes with adaptive look and feel

The frame definition is provided in Figure 4. The color editor frame uses some application slots for storing the current rgb values as well as two panes that are used for giving the user feedback. The `with-frame-slots` macro is used to get access to them in code that is implicitly part of a method on the frame; for example, in a `:pane` option.

The color frame introduces another frame option, `:menu-group`, which we will discuss below.

The code provided in the `:pane` option in Figure 4 uses all three kinds of panes provided by CLIM. It uses two `basic-clim-pane` application panes to display colors. Unlike in the previous example, this pane type isn't specialized because its behavior serves this frames purposes adequately. In particular, input is ignored and its default repaint method fills the pane with a background color which can be changed.[4]

The color editor example also uses the two other

[4] Note that in the previous example the repaint method on `basic-clim-pane` could have been reused by the hello pane using method combination.

```
(define-application-frame color-editor-frame ()
    (current-color-pane
     drag-feedback-pane
     (red :initform 0.0)
     (green :initform 0.0)
     (blue :initform 0.0))
  (:pane
   (with-frame-slots (current-color-pane drag-feedback-pane red blue green)
     (vertically ()
      (bordering ()
       (setf current-color-pane         ; Pane to show selected color
             (make-instance 'basic-clim-pane :hs 200 :hs+ +fill+ :vs 50
                            :background (make-color-rgb red green blue))))
      (bordering ()
       (setf drag-feedback-pane         ; Pane to show color in real time
             (make-instance 'basic-clim-pane :hs 200 :hs+ +fill+ :vs 50
                            :background (make-color-rgb red green blue))))
      ;; Slider defaults min-value to 0 and max-value to 1
      (realize-pane 'slider :id 'red   :orientation :horizontal)
      (realize-pane 'slider :id 'blue  :orientation :horizontal)
      (realize-pane 'slider :id 'green :orientation :horizontal)
      +fill+)))
  (:menu-group ("Quit" :command '(com-quit-color-editor)))
  (:settings :title "Color Editor"))
```

Figure 4: A frame definition for a color editor.

kinds of panes: layout panes and adaptive gadget panes. Layout panes are used to arrange other panes spatially. Gadget panes are pre-wired interactive objects that can be used to provide users with a means for controlling application behavior or inputting data.

The color editor uses the vertical box and border layout pane classes. The `vertically` and `bordering` macros provide an interface to these panes classes. Most CLIM layout panes provide similar macro interfaces to improve the readability of the code and to provide the convenience of automatically constructing a list of subpanes.

The vertical box pane arranges its children in a stack from top to bottom in the order they are listed at creation in the `vertically` form. This pane type also supports inter-element space and "pieces of glue" at arbitrary points in the children sequence. In the color editor frame, the `+fill+` "glue" is used to absorb all extra space when too much vertical space is allocated to the vertical box. CLIM also provides a horizontal box which does the same thing except in the horizontal direction.

Now we are ready to turn to the gadget panes. The color editor uses three sliders to allow the user to specify red, green, and blue values between 0.0 and 1.0. The three calls to `realize-pane` construct horizontal slider panes that also show their current value as a numeric field.

The application calls `realize-pane` on an abstract gadget specification name rather than `make-instance` on a real gadget class name. This allows an appropriate CLIM object to participate in selecting a gadget class. In particular, the pane hierarchy construction code is invoked when the frame is being adopted into a frame manager, and the frame manager determines what real gadget class is instantiated by the calls to `realize-pane`. This enables the frame manager to pick a gadget class that has the proper look and feel for the host window environment.

Gadget panes notify their clients—in general, their frame—by invoking callback generic functions. All callback functions take at least three arguments: `gadget`, `client`, and `id`. The client argument enables using specialization on the frame as a means for specifying callback handlers for a frame's gadgets.

The `id` argument provides a means for distinguishing gadgets in a frame from one another or for associating data with a gadget. However, CLIM doesn't

```
(defmethod drag-callback ((slider slider) (client color-editor-frame) id)
  (with-slots (drag-feedback-pane red green blue) client
    (setf (pane-background drag-feedback-pane)
          (ecase id
            (red   (make-color-rgb (gadget-value slider) green blue))
            (green (make-color-rgb red (gadget-value slider) blue))
            (blue  (make-color-rgb red green (gadget-value slider)))))
    (repaint-sheet drag-feedback-pane +everywhere+)))

(defmethod value-change-callback ((slider slider)
                                  (client color-editor-frame)
                                  id
                                  (new-value value))
  (with-slots (current-color-pane red green blue) client
    (ecase id
      (red   (setf red new-value))
      (green (setf green new-value))
      (blue  (setf blue new-value)))
    (setf (pane-background current-color-pane)
          (make-color-rgb red green blue))
    (repaint-sheet current-color-pane +everywhere+)))
```

Figure 5: Callback methods for color editor frame.

enforce uniqueness across a frame's pane hierarchy. The id argument is provided as a required argument to the callback functions, so that specialization on the id can be used to partition callback handling code.

The slider gadget protocol defines two callback functions: `drag-callback` is repeatedly invoked while the slider is being "dragged" by the user, and `value-change-callback` is invoked when the slider is "released" in a new location. Notice that this specification is sufficiently abstract to allow a variety of different look and feels for a slider. For example, no guarantee is made as to whether the mouse button is held down during dragging, or whether the mouse button is pressed once to start and again to stop dragging.

Figure 5 shows methods for the slider callback functions specialized on the color frame. The `drag-callback` method resets the background color of and repaints the frame's drag feedback pane, and the `value-change-callback` does the same for the frame's current color pane. These methods use the id argument to determine which color component to change. In a more complicated situation or one where the frame may be subclassed, `eql` specialization on id could have been used instead of `ecase`.

An application can invoke a color editor frame by executing the following:

```
(launch-frame 'color-editor-frame
   :color (make-color-rgb 0 0 1)
   :wait-until-done t)
```

The `:wait-until-done` argument to `launch-frame` allows the frame to be invoked as if it were a function. The frame can then return values to its user. In this particular case, the frame will take an initial color (the details of parsing this color and setting the frame slots aren't shown here), edit it, and return it when the users quits the frame.

The color editor frame uses the `:menu-group` option to indicate that access to certain application operations in a manner appropriate to the host environment should be provided. In this example, only one such operation, named `com-quit-color-editor` and presented as `"Quit"` to the user, is required by the frame. The frame manager will determine how this command can be invoked by the user. The definition of `com-quit-color-editor` command (not provided here) would remove the frame from the host screen and return the current color value. In the next example, commands will be explored in greater detail.

3.3 Simple Commands

In this third application building example, we build a simple drawing program that can be used to draw

```
(define-application-frame draw-frame ()
    (
     ;; Lines and Texts of Drawing
     (lines      :accessor lines   :initform nil)
     (strings    :accessor strings    :initform nil)
     ;; Current Text Entry
     (cur-point  :accessor cur-point  :initform nil)
     (cur-string :accessor cur-string :initform nil))
  (:pane (make-instance 'draw-pane :hs 400 :vs 400
                        :text-style (intern-text-style :serif :bold-italic :large)))
  (:command-definer t)
  (:top-level t))

(define-application-pane draw-pane ()
    ())

(defmethod handle-repaint ((pane draw-pane) region)
  (declare (ignore region))
  (let ((frame (pane-frame pane))
        (medium (sheet-medium pane)))
    (call-next-method)                  ; Paints the background
    (dolist (line (lines frame))
      (draw-line medium (first line) (second line)))
    (dolist (pair (strings frame))
      (draw-text medium (cdr pair) (car pair)))
    (when (cur-string frame)
      (draw-text medium (cur-string frame) (cur-point frame)))))
```

Figure 6: A frame and pane definition for a line and text draw program

lines and insert text. This example further elaborates on the concept of command introduced in the last example. Our simple drawing program will define commands for various drawing operations and will "bind" specific input events to these commands.

The draw frame and pane are defined in Figure 6. The `handle-repaint` method for the draw-pane is straightforward. It fills the pane to the background using the default method provided for application panes, and then iterates through the lines and strings painting them. Finally it paints the current string being entered.

The draw frame uses two options to `define-application-frame` that we have not encountered before. The `:command-definer` option is used to specify a name for a command-defining macro for this frame class. Passing `t` to this option, as in the example, indicates that a name of the form `define-<frame-name>-command` should be used. The command-defining macro can then be used to define commands that are specialized to the defined frame class.

The second option, `:top-level`, specifies a special form that is used to initiate the top level command loop for the frame. If this option is specified in a multiprocessing environment, a process will be started that executes an appropriate top level run function. If this option is `t` then a default top level loop is used. The top level is responsible for dequeuing commands that have been invoked by the user and executing them. We will return to some details below.

Figure 7 shows the command definitions for this frame. Each of these commands will generate a method specialized on `draw-frame`. The `with-frame` and `with-frame-slots` macros can be used to access the frame or its slots within a command definition.

To complete this example, we need to attach these commands to particular input operations on the pane. Figure 8 defines some input methods specialized on the draw pane. Each of these methods invokes `execute-frame-command` passing in a special command invocation form. This eventually leads to each

```
(define-draw-frame-command com-draw-clear ()
  (with-frame (frame)
    (setf (lines frame)      nil
          (strings frame)    nil
          (cur-string frame) nil)
    (repaint-sheet (frame-pane frame) +everywhere+)))

(define-draw-frame-command com-draw-add-string (string point)
  (with-frame (frame)
    (push (cons point string) (strings frame))))

(define-draw-frame-command com-draw-update-string (string point)
  (with-frame (frame)
    (let ((pane (frame-pane frame)))
      (draw-text pane string point)
      (medium-force-output pane))))

(define-draw-frame-command com-draw-add-line (point event-x event-y)
  (with-frame (frame)
    (let ((pane (frame-pane frame)))
      (setf (lines frame)
            (push (list point (make-point event-x event-y))
                  (lines frame)))
      (draw-line* pane (point-x point) (point-y point) event-x event-y)
      (medium-force-output pane))))
```

Figure 7: The draw frame's command implementations.

command being invoked by the frame's top level loop. In a multiprocessing environment, this happens in a separate top level loop process associated with the frame. We do not have room in this paper to cover the more subtle details.

The `button-press` method uses the `tracking-pointer` macro to manager a "rubber-banding" line input loop. This macro allows the application to process all input, bypassing the normal input distribution channels. Note that a special ink, `+flipping-ink+`, is used to paint the line. This ink, analogous to using xor on a monochrome screen, provides a device-independent way to draw the line such that it can be easily "undrawn."

This example also provides methods for button click. A button click is an event that is generated when a mouse button is pressed and then released immediately afterwards. CLIM also supports double mouse button clicks. The default methods for these events decompose them into press and release events.

A final observation about this example is that it binds the specific user interface to the commands using methods on the draw pane. We believe that special pane classes could be provided that implement any number of existing or new declarative translation mechanisms for mapping events to command invocations. We await user feedback to guide further design work in this area.

4 High Level Facilities

In this section, we explain a number of higher level facilities provided by CLIM, including output recording, formatted output, presentations, context sensitive input, and command procesors. Many of these facilities are derived from work done at Symbolics on the Dynamic Windows (DW) project for Genera[16]. See [4] for more detailed information on the motivations and design details behind DW. Many of the original contributers to DW have participated in the redesign of these facilities for CLIM.

We illustrate these facilities in two complete examples: a directory lister and a simple schedule browser.

```
(defmethod key-press ((pane draw-pane) char
                      &key &allow-other-keys)
  (let ((frame (pane-frame pane)))
    (setf (cur-string frame)
          (concatenate 'string (or (cur-string frame) "")
                       (string char)))
    (execute-frame-command
     frame
     `(com-draw-update-string ,(cur-string frame) ,(cur-point frame)))))

(defmethod button-press ((pane draw-pane) (button-name (eql :left))
                         &key x y &allow-other-keys)
  (let ((frame (pane-frame pane)))
    (setf (cur-point frame) (make-point x y))
    (let (lastx lasty)
      (tracking-pointer (pane)
        (:pointer-motion (x y)
          (when lastx
            (draw-line* medium startx starty lastx lasty :ink +flipping-ink+))
          (draw-line* medium startx starty x y :ink +flipping-ink+)
          (setq lastx x
                lasty y))
        (:button-release (button-name x y)
          (when (eql button-name :left)
            (when lastx
              (draw-line* medium startx starty lastx lasty :ink +flipping-ink+))
            (execute-frame-command
             frame `(com-draw-add-line ,(cur-point frame) ,x ,y))
            (return)))))))

(defmethod button-click ((pane draw-pane)
                         (button-name (eql :left))
                         &key x y &allow-other-keys)
  (let ((frame (pane-frame pane)))
    (when (cur-string frame)
      ;; Complete the previous string
      (execute-frame-command
       frame
       `(com-draw-add-string ,(cur-string frame) ,(cur-point frame)))
      (setf (cur-string frame) nil))
    ;; Start the next string.
    (setf (cur-point frame) (make-point x y))))

(defmethod button-click ((pane draw-pane)
                         (button-name (eql :middle))
                         &key x y &allow-other-keys)
  (let ((frame (pane-frame pane)))
    (execute-frame-command frame '(com-draw-clear))))
```

Figure 8: The draw pane's input methods.

```
(defun show-packages (stream)
  (formatting-table (stream)
    (dolist (package (list-all-packages))
      (formatting-row (stream)
        ;; The first column contains the package name
        (formatting-cell (stream)
          (write-string (package-name package) stream))
        ;; The second color contains the symbol count,
        ;; aligned with the right edge of the column
        (formatting-cell (stream :align-x ':right)
          (format stream "~D" (count-package-symbols package)))))))
```

Figure 9: An output function that uses table formatting.

Output Recording Many of the higher level facilities in CLIM are based on the concept of *output recording*. The CLIM output recording facility is simply a mechanism wherein a window remembers all of the output that has been performed on it. This output history (stored basically as a display list) can be used by CLIM for several purposes. For example, the output history can be used to automatically support window contents refreshing (or "damage repaint" events). This database can be exploited in more sophisticated ways, as we shall see later. Of course, the application programmer has considerable control over the output history. Output recording can be enabled or suspended, and the history itself can be cleared or pruned.

Output records can be nested, thereby forming their own hierarchy. The leaves of this tree are typically records that represent a piece of output, say the result of a call to `draw-rectangle` or `write-string`. The intermediate nodes typically provide additional semantics to the tree, such as marking a subtree of nodes as resultant output of one particular phase of an application. CLIM provides support for defining new output record types, to allow for customization of such attributes as how the storage of inferior nodes is managed (e.g. a node that is going to store many graphical elements may want to sort them into an R tree for faster search and retrieval).

Output Formatting CLIM provides a convenient table and graph formatting facility, which is built on top of the output recording facility. The key to these formatting tools (as opposed to, say, `format`'s `X` directive) is that they dynamically compute the formatting parameters based on the actual size of the application-generated output.

The application programmer uses these tools by wrapping any piece of output-producing code with "advisory" macros that help the system determine the structure of the output.

For example, start with a simple output function that shows some information about the packages in the Lisp environment:

```
(defun show-package-info (stream)
  (dolist (package (list-all-packages))
    (write-string (package-name package)
                  stream)
    (write-string "    " stream)
    (format
      stream "~D"
      (count-package-symbols package))
    (terpri stream)))
```

Any attempt to fix this function to produce tabular output by building in a certain fixed spacing between the package name and symbol count will either get caught by an unexpectedly-long package name, or will have to reserve way to much space for the typical case.

The code in Figure 9 is an improved version of this function that produces a neatly formatted table for any set of package names,

Presentations The next step up from perserving the mere physical appearance of output done to a window is to preserve its semantics. For example, when an application displays a Lisp pathname on the screen via `(format t "~A" path)`, the string "/clim/demo/cad-demo.lisp" may appear. To the user this string has obvious semantic meaning; it is a pathname. However, to Lisp (and the underlying system) it is just a text string. Fortunately, in many cases the semantics can be recovered from the string. Thus the power of the various textual cut-and-

paste mechanisms supported by contemporary computer systems. However, it is possible to improve upon the utility of this lowest common denominator facility (i.e. squeezing everything through its printed representation) by remembering the semantics of the output as well as its appearance. This is the idea behind *presentations*.

A presentation is a special kind of output record that maintains the link between screen output and the Lisp data structure that it represents. A presentation remembers three things: the displayed output (by capturing a subtree of output records representing the output), the Lisp object associated with the output, and the *presentation type* of the output. By maintaining this "back pointer" to the underlying Lisp data structure, the presentation facility allows output to be "reused" at a higher semantic level.

An application can produce semantically tagged output by calling the CLIM function `present`. For example, to display the pathname referred to above as a presentation, the application would execute `(present path 'pathname)`. `present` captures the resulting output and the pathname object in a presentation of type `'pathname`.

Presentation Types CLIM defines a set of presentation types, which are arranged in a supertype/subtype lattice like the CL types. In fact, the presentation type hierarchy is an extension of the CL type hierarchy. The reason that this extended type system is needed is that the CL type system is "overloaded" from the UI perspective. For example, the integer 72 might represent a heart rate in one application and a Fahrenheit temperature in another, but it will always just be an integer to Lisp.

The application programmer can define the "UI entities" of the application by defining presentation types, thus extending the presentation type library. By defining a presentation type, the programmer can centralize all of the UI aspects of the new type in one place, including output appearance and input syntax. As an example, CLIM defines a `pathname` presentation type that defines how a pathname is displayed and how one is input. The pathname input side provides pathname completion and possibilities-display features. By defining this behavior in one place and using it in all applications that need to display or read pathnames, CLIM helps build consistent user interfaces.

Note that in the pathname output example given above, `present` invokes the standard pathname displayer defined by the presentation type. However, since the presentation facility is simply based on the output recording facility, presentation semantics can be given to any output. The following example shows how the pathname object could be associated with some graphics that were displayed on the screen.

```
(with-output-as-presentation
      (:object path
       :type 'pathname
       :stream s)
  (draw-rectangle* s 0 0 30 30))
```

Context-Dependent Input Once semantically-tagged output as been displayed on the screen, how can it be reused? The key is for the application to establish an *input context* whenever it is waiting for input from the user. The input context is specified in terms of the presentation type that is appropriate for the current input point. CLIM provides a simple interface for managing the input context in the form of the `accept` function. For example, if the application requires the user to input a pathname, it can execute `(accept 'pathname :stream s)`. Typically, this will invoke the input reader (or *parser*) that was defined for the pathname type and will establish an input context that indicates that it is waiting for a pathname.

Once the input context has been established, CLIM will automatically make any appropriate existing output available to the user via mouse gestures. That is, once an input context of "pathname" as been established, the user can move the mouse over any presentation that is of type pathname (or is a subtype of pathname), click on it, and the pathname object underlying the presentation is returned as the value of the call to `accept`.

Command Processors As discussed earlier, applications can define each user-accessible operation as a command that has a name and a set of zero or more arguments. For each argument the application programmer can specify a presentation type. This type information is used by CLIM's command processor facility to manage a dialog with the user in which arguments of the correct type are collected. In this way, previously "presented" data can be used by the user to supply arguments to the commands. Also, the body of a command is guaranteed to be invoked on argument values of the correct presentation type (and Lisp type).

Defining an application's operations as commands does not define or constrain the interaction style of the application. Once a command has been defined, the CLIM *command processor* facility can invoke several built-in mechanisms to read command "sentences" (a command name and its arguments)

from the user in a variety of ways. The command name and arguments may be typed by the user, or the command name might be selected from a menu and the arguments supplied by clicking on displayed presentations, or a single keystroke "accelerator" might invoke a command on a default argument set. In addition, the command processer is extensible by application programmers. For example, the command processor can be extended to support a "noun then verb" interaction style, where the user can first click on a displayed presentation and then invoke a command that is defined to take an argument of the selected presentation type.

4.1 A Directory Browser

To illustrate how all of these high level facilities come together, we define a simple example in Figure 10 that makes use of them all.

The `dirlist-frame` application is a very simple-minded file system browser. It defines two panes, an input pane to handle user typein and an output pane to display directory contents. The application defines a display function that is associated with the output pane. The function displays the contents of a directory, one file per line. Each line is a presentation of type `pathname`, so the association between the text lines on the screen and the Lisp pathname objects that produced them is maintained. The application defines one command that takes an argument of type `pathname` and displays the contents of the specified directory.

The `define-application-frame` form makes use of some new concepts and options:

- State variables. The application has two slots, one to hold the list of files contained in the current directory, and the other to hold on to the display pane.

- `making-application-pane`. This convenience macro packages up some useful application pane idioms. It controls the scrolling behavior of the pane, including whether it has a vertical scroll bar, a horizontal scroll bar, neither, or both.

- The `extended-stream-pane` type. This kind of pane supports the extended input and output protocols of CLIM, including output recording and the presentation type facility. Many of these facilities are defined as extensions of the Common Lisp stream protocol.

- The `:display-function` and `:display-time` options. These options provide a simple way to manage application information display in a pane. The display function is invoked by the CLIM command loop at the specified time. Typically, the display function utilizes application state information that is modified by the application's commands. A `:display-time` of `:command-loop` means to run the display function whenever an application command is executed.

- The `extended-top-level` function. This top-level supports the extended command loop, including the management of the command processer and display functions.

The `dirlist-display-files` display function is the heart of this application. It iterates over the contents of the current directory displaying the files one by one. Because it uses `present`, each displayed pathname is a presentation, and can automatically be selected by clicking on it with the mouse in an appropriate context.

The `com-edit-directory` command is defined to take one argument, of presentation type `pathname`. The command's body tries to interpret the pathname that it receives as a directory, obtaining a list of the files contained therein.[5] The command simply updates the `files` application state variable with the new list.

Since this application was defined with an interactor pane for user input, the user can invoke the sole command by typing its name, "Edit Directory." CLIM supports automatic command completion, so in this case only the first letter and the complete action need be typed. At this point the CLIM command loop will begin reading the arguments for that command, and will automatically enter a `pathname` input context. Thus, the user can fill in the required argument either by typing a pathname, or by clicking on one of the pathnames visible in the display pane.

In order to carry this interface one step further into an entirely mouse-driven interaction style, the application programmer must specify some association between a mouse gesture and the command it invokes. This is done via the `define-presentation-to-command-translator` form. The CLIM presentation substrate supports a general concept of presentation type "translation." This translation mechanism can be used to map objects of one type into a different presentation type, if appropriate. For example, it might be possible to satisfy an input request for a pathname by selecting a computer user's login ID and

[5] Common Lisp is weak in its support of hierarchical file systems so this code is somewhat contrived.

```
(define-application-frame dirlist-frame ()
    ((files :initform nil)
     pane)
  (:pane
   (with-frame-slots (pane)
     (vertically ()
      (making-application-pane (:hs 400 :vs 50)
       (make-instance 'extended-stream-pane :interactor t))
      (making-application-pane (:hs 400 :vs 400)
       (setf pane
             (make-instance 'extended-stream-pane
                            :display-function '(dirlist-display-files)
                            :display-time ':command-loop))))))
  (:command-definer t)
  (:top-level (extended-top-level)))

(defmethod dirlist-display-files ((frame dirlist-frame) pane)
  (clear-output-history pane)
  (with-slots (files) frame
    (dolist (file files)
      (present file 'pathname :stream pane)
      (terpri pane))))

(define-dirlist-frame-command (com-edit-directory :command-name "Edit Directory")
    ((dir 'pathname))
  (setq dir (make-pathname :directory (append (pathname-directory dir)
                                              (list (pathname-name dir)))
                           :name :wild :type :wild :version :wild
                           :defaults dir))
  (with-frame-slots (files)
    (setf files (directory dir))))

;;; This defines a "mouse" translator that says "when I click left
;;; on an object of type PATHNAME, run this body, which maps the
;;; object into a command that takes that kind of object as an argument
(define-presentation-to-command-translator edit-dir (pathname :gesture ':left)
  (object)
  `(com-edit-directory ,object))
```

Figure 10: A complete implementation of a directory listing frame.

```
(defvar *days* #("Sun" "Mon" "Tue" "Wed" "Thu" "Fri" "Sat"))

(define-presentation-type weekday ()
  :parser ((stream &key default)
           (declare (ignore default))
           (values
            (completing-from-suggestions (stream)
              (dotimes (i 7)
                (suggest (aref *days* i) i)))))
  :printer ((object stream &key acceptably)
            (write-string (aref *days* object) stream)))
```

Figure 11: The weekday presentation type definition.

returning the pathname of the user's home directory. This would be accomplished by defining a translator from a user-id presentation type to the **pathname** type that consulted the system's user database to retrieve the home directory information.

By far the most useful application of the translation mechanism is to define translators from arbitrary presentation types to the command presentation type. In our example, by defining a translator from the **pathname** type to the **command** type which maps the pathname into Edit Directory of that pathname, and assigning it to the left mouse button, CLIM will automatically invoke the Edit Directory command on a pathname that the user selects with the left mouse button.

4.2 Schedule Example

We present one last example to explore how an application might define its own presentation types, as well as take advantage of the output formatting facilities. In this example we build a simple appointment browser.

Since the user of this application will frequently be dealing with the days of the week, we start by defining a new presentation type, **weekday**. This simple presentation type, shown in Figure 11, represents a day of the week as a number from 0 to 6. Each day number has associated with it the abbreviated day name, "Mon," "Tue," etc.

The **weekday** presentation type defines two basic pieces of behavior: how a weekday is displayed, and how it is read as input. It does this by defining a *printer* and a *parser* function for the type. As is the case with most presentation types, the printer and parser are duals. That is, the printer, when given an object to print, produces output that the parser can interpret to arrive back at the original object.

We define an application frame for the appointment browser in Figure 12. The frame defines state variables to hold the list of appointments and the current day. This information is kept in slots on the frame (rather than global variables) so that multiple copies of the application can be run, each with its own appointment list. The application represents the appointment data as an alist containing an entry for each day of the week, with each entry containing a list of the appointments for the day. Test data is provided as a default initform.

Just as in the previous example, the appointment application defines two panes, an interactor and an output display pane, and the standard **extended-top-level** command loop is used.

The appointment application defines two commands. The "Show Summary" command resets the display back to the weekly summary mode by setting the **current-day** slot to **nil**. The "Select Day" command sets **current-day** to the value of an argument that is specified to be a weekday. This presentation type specification allows the command processor to make all presented weekday active when it is filling in this argument, as well as, provide completion assistance to the user.

As before, it would be a great user convenience to be able to invoke the "Select Day" command simply by clicking on one of the displayed weekdays. However, rather than using the **define-presentation-to-command-translator** form, this time we use a shortcut provided by the command definition macro. When defining the argument for the "Select Day" command, we not only specify that it is of type **weekday**, but that we would like to define that clicking left on a weekday presentation should invoke this command. This is done using the **:translator-gesture**

```
(defvar *test-data*    ;; Alist of day number and appointment strings
  '((0) (1 "Dentist") (2 "Staff meeting") (3 "Performance Evaluation" "Bowling")
    (4 "Interview at ACME" "The Simpsons") (5 "TGIF") (6 "Sailing")))

(define-application-frame schedule ()
    ((display-pane)
     (appointments :initarg :appointments :initform *test-data*)
     (current-day :initform nil))
  (:pane (with-frame-slots (display-pane)
           (vertically
            (making-application-pane (:hs 400 :vs 200)
              (setf pane (make-instance 'extended-stream-pane
                                        :display-function '(display-appointments)
                                        :display-time ':command-loop)))
            (making-application-pane (:hs 400 :vs 50)
              (make-instance 'extended-stream-pane :interactor t)))))
  (:command-definer t)
  (:top-level (extended-top-level)))

;;; Chooses which day to see in detail.
(define-schedule-command (com-select-day :command-name t)
    ((day 'weekday :translator-gesture ':left))
  (with-frame-slots (current-day)
    (setq current-day day)))

;;; Show weekly summary.
(define-schedule-command (com-show-summary :command-name t) ()
  (with-frame-slots (current-day)
    (setq current-day nil)))
```

Figure 12: The schedule frame definition and commands.

option to the command definer.

Finally, we turn to the display the appointment information. The display function, `display-appointments`, shown in Figure 15, is somewhat more complex than our earlier example. It can display two different sets of information: a weekly summary showing the days of the week and the number of appointments for each day, or a detailed description of one day's appointments. The two information displays are shown in Figure 13 and Figure 14.

`display-appointments` decides which set of information to display by examining the application state variable `current-day`. The table formatting facility to present the weekly summary information neatly organized. The daily appointment list, by contrast, is displayed "by hand". Note, however, that whenever a day of the week is displayed, it is done with a call to `present` using the `weekday` presentation type. This allows the printed weekdays to be selected either as a command or as a weekday argument.

This example illustrates how an applciation with interesting UI behavior can be constructed from a high-level specification of its functionality.

Conclusion

The series of examples presented in this article illustrate the broad range of functionality provided by CLIM. The later examples, especially, demonstrate that complex user interfaces can be built economically and modularly using CLIM. Many of the higher level facilities make it possible to separate the issues involved in designing an application's user interface from the functionality of the application.

On the other hand, many of these higher level facilities may not be appropriate for all users. CLIM's lower level facilities and clean modularization of the higher level facilities provide these users with portable platform and a framework for implementing their own

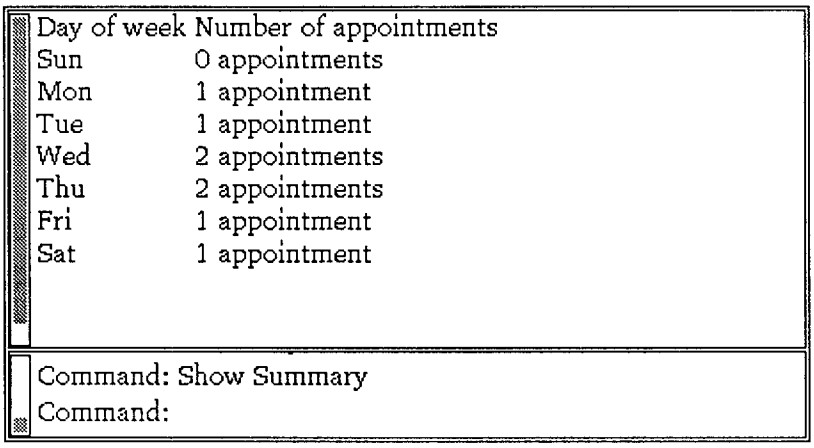

Figure 13: The Schedule example displaying schedule summary

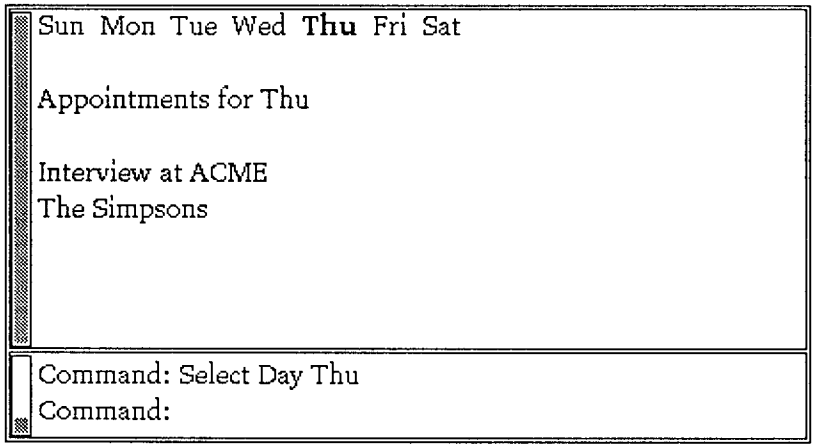

Figure 14: The Schedule example displaying schedule for a particular day, i.e. Thursday

```
;;; Complex display function, shows two completely different
;;; displays.
(defmethod display-appointments ((frame schedule) pane)
  (clear-output-history pane)
  (with-slots (current-day appointments) frame
    (if (null current-day)
        (show-weekly-summary pane appointments)
        (show-appointments pane current-day
                           (rest (assoc current-day appointments))))))

;;; Show a summary of the week, with an appointment count for
;;; each day.  You can see the appointments for a specific day by
;;; clicking on the day name.
(defun show-weekly-summary (pane appointments)
  (formatting-table (pane)
    ;; Table headings
    (formatting-row (pane)
      (formatting-cell (pane)
        (write-string "Day of week" pane))
      (formatting-cell (pane)
        (write-string "Number of appointments" pane)))
    (dolist (day appointments)
      (formatting-row (pane)
        (formatting-cell (pane)
          (present (first day) 'weekday :stream pane))
        (formatting-cell (pane)
          (format pane "~D appointment~:P" (length (rest day))))))))

;;; Show detailed appointment list for day
(defun show-appointments (pane current-day current-day-appointments)
  ;; Show all days at top so you can switch to another
  ;; day with one click.
  (dotimes (day 7)
    (with-text-face ((if (eql day current-day) ':bold ':roman) pane)
      (present day 'weekday :stream pane))
    (write-string "  " pane))
  (terpri pane)
  (terpri pane)
  ;; Show all the appointments, one per line
  (write-string "Appointments for " pane)
  (present current-day 'weekday :stream pane)
  (terpri pane)
  (terpri pane)
  (dolist (appointment current-day-appointments)
    (write-string appointment pane)
    (terpri pane))))
```

Figure 15: The schedule's display function.

user interface toolkits and frameworks. In addition, CLIM's use of CLOS to define explicit, documented protocols provides application programmers with the opportunity to customize CLIM and support interfaces not anticipated by the CLIM designers.

CLIM currently supports the Genera, X, SunView, and Macintosh host window environments. A developer's prerelease of CLIM is now available for Allegro Common Lisp, Lucid Common Lisp, CLOE-386, and Genera. In addition, prerelease versions of CLIM should soon be available for Harlequin Common Lisp and Macintosh Common Lisp.

We along with a number of other individuals of the CLIM consortium are actively working on writing a specification for CLIM[11].

Acknowledgements

CLIM represents the cooperative effort of individuals at several companies. These individuals include Jim Veitch, John Irwin, and Chris Richardson of Franz; Richard Lamson, David Linden, and Mark Son-Bell of ILA; Paul Wieneke and Zack Smith of Lucid; Scott McKay, John Aspinall, Dave Moon and Charlie Hornig of Symbolics; and Gregor Kizcales and John Seely Brown of Xerox PARC. Mark Son-Bell and Jon L. White have help us improve this paper.

References

[1] Apple Computer. *Inside Macintosh*, volume 3. Addison-Wesley, Reading, MA, 1985.

[2] Intellicorp, Mountain View, CA. *Common Windows Manual*, 1986.

[3] M. Linton, J. Vlissides, and P. Calder. Composing user interfaces with interviews. *IEEE Computer*, 22(2):8–22, Feb 1989.

[4] Scott McKay, William York, and Michael McMahon. A presentation manager based on application semantics. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 141–148. ACM Press, Nov 1989.

[5] Microsoft Corporation, Redmond, WA. *Microsoft Windows Software Development Kit*, 1985.

[6] Next, Inc, Redwood City, CA. *Next Preliminary 1.0 System Reference Manual: Concepts*, 1989.

[7] Open Software Foundation, Cambridge, MA. *OSF/MOTIF Style Guide*, 1989.

[8] Rob Pettengill. The deli window system: A portable, clos based network window system interface. In *Proceedings of the First CLOS Users and Implementors Workshop*, pages 121–124, Oct 1988.

[9] Ram Rao and Smokey Wallace. The x toolkit. In *Proceedings of the Summer 1987 USENIX Conference*. USENIX, 1986.

[10] Ramana Rao. Silica papers. In Preparation, 1991.

[11] Ramana Rao, Bill York, Dennis Doughty, John Aspinall, Scott Mckay, and Dave Moon. Common lisp interface manager specification. In Preparation, 1990.

[12] R.W. Scheifler and J. Gettys. The x window system. *ACM Transactions on Graphics*, 5(2), 1986.

[13] Sun Microsystems, Mountain View, CA. *SunView Programmer's Guide*, 1986.

[14] Sun Microsystems. *NeWS Technical Overview*, 1987.

[15] Sun Microsystems, Mountain View, CA. *OPEN LOOK Graphical User Interface*, 1989.

[16] Symbolics, Inc. *Programmer's Reference Manual Vol 7: Programming the User Interface.*