A Guided Tour of CLIM, Common Lisp Interface Manager

2006 Update
Clemens Fruhwirth <clemens@endorphin.org>
The McCLIM Project

Original article*
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.

October 14, 2019

Abstract

The Common Lisp Interface Manager (CLIM) provides a layered set of facilities for building user interfaces. These facilities include a portable layers for 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;[1] 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. The article originally appeared in Lisp Pointers in 1991 and was updated in 2006 by Clemens Fruhwirth.[2] All examples in this article have been run with McCLIM[McC], a free CLIM implementation, as of January 2006.

*Published in Lisp Pointers 1991

[1]Similar to the work pioneered in the Genera UI system

[2]The CLIM 2 specification changed significant parts of CLIM rendering legacy code unusable. Clemens Fruhwirth has rewritten all examples and the corresponding text sections for the CLIM 2 specification. In addition, he has restructured the whole article, adding sections to provide additional insights into CLIM concepts.

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.[3]

The Common Lisp Interface Manager 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 such as geometry, graphics, event-oriented input, and windowing; intermediate level facilities such as support for Common Lisp stream operations, output recording, and advanced output formatting; and high level facilities such as context sensitive input, an adaptive toolkit, and an application building framework.

[3]Notice that this sentence was written in 1991; it is still true 15 years later.

CLIM implementations will eventually support a large number of window environments including X Windows, Mac OS X and Microsoft Windows. 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. CLIM supports the incorporation of toolkits written in other languages. A uniform interface provided by CLIM allows Lisp application programmers to deal only with Lisp objects and functions regardless of the operating platform.

An important goal that has guided the design of CLIM was to layer the specification into a number of distinct facilities. Furthermore, the specification does not 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 programmers. This means that, in general, a CLIM programmer 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 specific needs or requirements of a particular application.

For example, CLIM's application framework and adaptive toolkit allow programmers 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 is 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. For instance, the CLIM programmer can implement new gadget types on top of the drawing primitives and treat them equally to the built-in gadget types.

We will use the term *CLIM implementor* for the party implementing low-level and high-level parts according to the CLIM specification, *CLIM programmer* for the party that will use the facilities provided by the implementor, and *CLIM user* for the party that will use the programs provided by the programmer.

The next section presents an overview of the functionality provided by CLIM facilities.

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 substrate CLIM provides a portable interface to a broad set of graphics functions for drawing complex geometric shapes.

Windowing substrate CLIM provides a portable layer for implementing sheets (windows and other window-like objects).

Extended Streams CLIM integrates the Common Lisp Stream I/O functionality with the CLIM graphics, windowing, and panes facilities. In addition to ordinary text, the programmer can send a button, a picture or any other arbitrary widget to a CLIM output stream and CLIM will display the widget in the sheet associated with the output stream.

Output Recording CLIM provides output recording for capturing all output done to an extended stream and automatically repainting it when necessary.

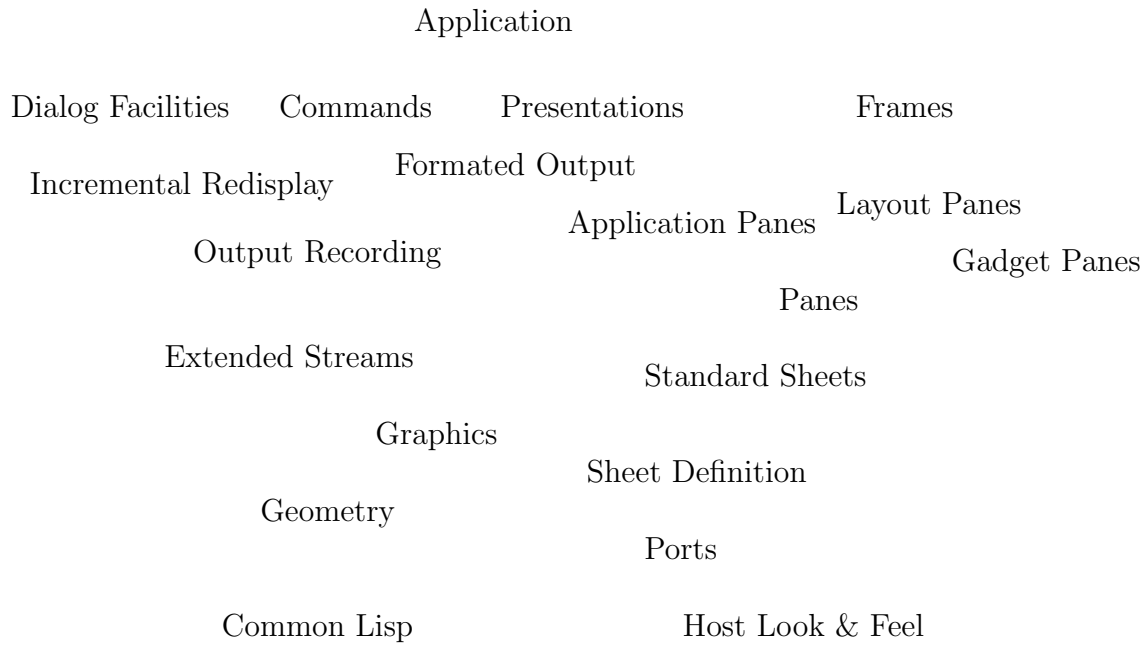

Figure 1: An Overview of CLIM facilities

Formatted Output CLIM provides a set of high-level macros that enable programs to produce neatly formatted tabular and graphical displays easily.[4]

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.

Panes CLIM provides *panes* that are analogous to the gadgets or widgets of toolkits like the X toolkit, GTK or Mac OS X's toolkit.

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. Applications that use these gadget types and related facilities will automatically adapt to use whatever toolkit is available on and appropriate for the host environment. In addition, portable Lisp-based implementations of the abstract gadget pane protocols are provided.[5]

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.

[4] This also includes Graph Formatting. Graph formatting is only partly implemented in McCLIM at this date (March 2006).

[5] McCLIM does not support look and feel adaptiveness at the moment. Hence, McCLIM mostly uses this portable Lisp-based implementation.

Commands CLIM supports the separation of command execution and command invocation. A CLIM user can invoke commands either via typing it into an interactor, clicking on a menu entry or implicitly invoking a presentation-translator by clicking on a presentation. Commands can also be invoked explicitly by the programmer.

Dialogs and Incremental Update/Redisplay Incremental Redisplay goes a bit further than Output Recording. With Incremental Redisplay, an output record can not only reproduce content that was written to a stream, the CLIM programmer can also attach the code that generated the content to the content itself. Whenever necessary, the application programmer can ask an output stream to update itself. CLIM will query all drawn elements for obsolescence, and if necessary, rerun the code to produce fresh output.

This is a large number of facilities to explore. The most systematic way – progressing from the lowest-level to the highest – would also be the lengthiest. Therefore, we start with showing several facilities in action with the most fundamental example in the realm of programming: Hello World.

2 Our first application

We will start with a few lines of code for the trivial Hello World example to give the reader a test case to verify his CLIM setup. It also serves as a point of reference from where the reader can start his explorations of more challenging CLIM facilities. We do not try to elaborate the CLIM concepts in detail here, but simply use them with a brief discussion. The confused reader may hope for a more in-depth explanation in the following section. Please regard *pane*, *application frame*, *sheet*, *sheet hierarchy*, *graft* and *top-level loop* as terms we will discuss later.

We provide extensive CLIM specification references in footnotes. The motivation for this is to show that all the relevant information can be found in the CLIM 2 specification[MY05]. Before a good CLIM programmer can master any CLIM concept, he has to get used to the style of writing of the specification, as this is the most relevant work for CLIM. The best we can do in this short paper is provide pointers and references and hope that the interested reader starts to explore the surrounding text sections on his own.

After loading a CLIM implementation, the package :clim-user is available to absorb user code. This package is a good start for experimentation and first steps. When proper packaging is required, simply include the packages :clim and :clim-lisp in your new package's :use list.

The central element of CLIM application programming is the *application-frame*. An application frame is defined via define-application-frame.[6] Here is the application frame definition for Hello World:

```
(define-application-frame hello-world ()
  ((greeting :initform "Hello World"
             :accessor greeting))
  (:pane (make-pane 'hello-world-pane)))
```

define-application-frame's basic syntax is similar to defclass because define-application-frame also generates classes. In this case, it creates a frame class hello-world that has no superclass except frame (which is added automatically).

With :pane, we define a *top-level-pane* that becomes the content of a fresh window that belongs to an application frame. Although the usual case is for an application frame to correspond to a top level window, sometimes an application frame is swallowed by another application and only space in another existing window is reserved. For instance, a web site management tool might swallow a text editor, so that the user has the option to edit web sites without switching to another application.

The list given after the :pane option is a form which is evaluated when an instance of the hello-world class is created. We use make-pane to construct a pane as the top-level-pane for frame instances. make-pane is a constructor for panes.[7] We can treat it as an analog to make-instance especially made for pane classes. Let us have a look at the definition of hello-world-pane.

```
(defclass hello-world-pane
  (clim-stream-pane) ())
```

The one and only superclass of hello-world-pane is clim-stream-pane.[8] As there are no additional slots,

[6] See Section 28.2 "Defining and Creating Application Frames" in [MY05].

[7] See Section 29.2 "Basic Pane Construction" in [MY05].

[8] See Section 29.4 "CLIM Stream Panes" in [MY05].

```
;;; Behavior defined by the Handle Repaint Protocol
(defmethod handle-repaint ((pane hello-world-pane) 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))
    ;; Draw greeting in center of pane
    (draw-text* pane
                (greeting *application-frame*)
                (floor w 2) (floor h 2)
                :align-x :center
                :align-y :center)))
```

Figure 2: handle-repaint for hello-world-pane

an experienced CLOS programmer might guess that we will use hello-world-pane solely for method specialization. Before doing so, let us have a look what we have actually inherited from clim-stream-pane:[9]

```
CLIM-USER> (describe (find-class
                      'clim-stream-pane))
DIRECT-SUPERCLASSES:
PERMANENT-MEDIUM-SHEET-OUTPUT-MIXIN
STANDARD-REPAINTING-MIXIN
STANDARD-EXTENDED-INPUT-STREAM
STANDARD-EXTENDED-OUTPUT-STREAM
STANDARD-OUTPUT-RECORDING-STREAM
SHEET-MULTIPLE-CHILD-MIXIN
BASIC-PANE
```

basic-pane is the foundation of all pane classes. It provides reasonable defaults for all protocol methods and inherits from the protocol class pane. In turn, pane inherits from basic-sheet. Hence, all panes we construct via basic-pane automatically adhere to the sheet protocol.

Our hello-world-pane needs some methods to be useful. With handle-repaint in Figure 2, we participate in the *repaint protocol*.[10] This method has two tasks: paint the pane with the pane's background and draw the greeting of *application-frame* in the center of the pane. The hello-world frame instance is automatically available in the context of handle-repaint via the dynamically scoped variable *application-frame* and so it is possible to use the accessor greeting to obtain the instance's slot content.

Two functions are needed to see this code in action: make-application-frame and run-frame-top-level. The former is a constructor for instances of frame classes, the latter for running the *top-level loop* of the application frame. The top-level loop is used for command reading, execution and displaying results. But all we need to know for the moment is that it makes the frame visible before entering the loop.

```
(run-frame-top-level
 (make-application-frame 'hello-world))
```

This completes the Hello World example. In the next section, we catch up to the details we skipped in this example.

3 Basic Facilities

3.1 Geometry

To CLIM, geometry means *regions*. A region is either bound or unbound and has a dimensionality of either zero, one or two. That corresponds to a point, a path or an area respectively. Regions can be compared (region predicate protocol[11]) and

[9] Internal classes removed from listing.

[10] See Section 8.4 "Repaint Protocol" in [MY05]

[11] CLIM relies heavily on CLOS. In CLOS, the term *protocol* means a set of generic functions that together form a coherent interface. A protocol specification not only includes the syntactic details of the function names and the number of function arguments, but also the functions' purpose and the semantics of the return values (and/or side effects) must be given in a textual specification.

composed (region composition protocol).

Every bounded region has a *bounding rectangle*. It is the smallest rectangle that contains every point in the region. The bounding rectangle of every bounded region can be accessed via the *bounding rectangle protocol*.

CLIM supports *affine transformations* of regions. Such transformations can move, stretch and rotate a region. A transformation is affine when every straight line remains straight after transformation. Transformations can be composed arbitrarily. The programmer can attach transformations to mediums and panes. In layout panes, CLIM uses transformations to map the coordinates of child panes to the coordinate system of their parents.

All drawing settings can be changed either permanently, or temporarily in the context of the with-drawing-options macro.

3.2 The Windowing Substrate

CLIM does not directly talk to the window system. Instead, CLIM is layered on top of a windowing substrate.[12] This substrate is a middleware between CLIM on one side and the host window system on the other. This middleware layer provides a portable windowing model that insulates higher levels of CLIM and CLIM programmers from the details of the host window system. From the perspective of a CLIM programmer, talking to the window system is equal to talking to this abstraction layer. The implementation details are hidden in *backends*, like in McCLIM the CLX backend, which hides X11 details. These backends will use the services of a host window system to provide efficient windowing, input and output facilities. Thanks to this middleware, CLIM is portable across different host windowing systems.

This framework allows uniform treatment of the following GUI building blocks:

- Windows like those in X, Mac OS X and Microsoft Windows.
- Gadgets typical of toolkit layers, such as Gtk+, QT or Mac OS X's toolkit. The backend provides a frame manager which takes care of mapping the abstract gadget types found in CLIM to appropriate gadget with a native look and feel.
- Structured graphics like output records and an application's presentation objects

The central abstraction specified by CLIM is the *sheet*. A sheet is a surface that can be painted on and to which input gestures can be directed, and that lives in a hierarchy of other such objects. To get used to the notation of sheets, you can think of them as swallowable windows.

The fundamental notion in CLIM is the nesting of sheets within another sheet called a windowing relationship. In a windowing relationship, a parent sheet provides space to or groups a number of other children sheets. The *sheet protocols* specify functionality for constructing, using, and managing hierarchies of sheets.

Sheets have the following properties:

parent/children: a sheet is part of a windowing hierarchy and maintains links to its parent and its children.

coordinate system: a sheet has its own coordinate system that might have a different scaling or orientation than its parent.

transformation: via a sheet transformation these differing sheet coordinate systems are mapped into coordinate system of their parents.

clipping region: defines an area within a sheet's coordinate system that indicates the area of interest.

Window hierarchies Sheets participate in a number of protocols. The windowing protocols describes the relationship between sheets. Every sheet has a parent, a sheet might have one or more children. A sheet can be adopted by a parent sheet and disowned later. A sheet is grafted when it is connected to a *graft* either directly or through it's ancestors. 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.

[12] formerly known as Silica.

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.

Window input The input architecture of the windowing substrate allows sheets to receive any window event from the host windowing system. The event class hierarchy descends from the class event to device-event (including all keyboard and mouse actions), window-event (window-size-changed and window-repaint events), window-manager-events (window-deleted) to artificial timer-events. See 8.2 "Standard Device Events" in [MY05].

The function pair dispatch-event and handle-event is the center of all event handling. dispatch-event is intended to be called when an event is to be dispatched to a client either immediately or queued for later processing in an event queue. On the other side, handle-event is intended for further specialization, so the application developer can implement special policies for selected events. For instance, when a sheet notices through a window-configuration-event that the sheet's size has changed, it might redo its layout for its children.

Window output All drawing operation happen on a *medium*. This object captures the state of the drawing like foreground, background, line style, and the transformation which is applied to the coordinates before drawing. Every medium is associated with a sheet. In turn, a sheet must be associated with a medium whenever drawing operation should be executed. Many CLIM sheets are associated with a medium permanently. sheet-medium obtains the medium associated with a sheet.

The graphic output capabilities of sheets range from simple line style and text style customization over rendering various geometrical shapes, a color model capable of doing alpha blending, composable affine transformations to pattern, stencil and tiling filling, and pixmaps usage. The features of the *output protocol* are specified briefly in Section 8.3 "Output Protocol"and more precisely in Chapters 10-14 of [MY05].

CLIM lives in an idealized world in terms of graphics operations. A CLIM programmer can use an infinitely long and wide drawing pane with arbitrarily precise resolution and continuously variable opacity. As rendering devices with these properties are rare, we need to render the idealized graphic description to a device with finite size and a fixed drawing precision. The rendering rules are specified in Section 12.4 "Rendering Conventions for Geometric Shapes"of [MY05].

4 Building Applications

In this section, we explain a number of the necessary ingredients for building applications in CLIM. We will illustrate frames, panes, and simple commands with two examples: a color editor and a simple line and text drawing program.

4.1 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.

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. The *frame manager* is responsible for attaching the pane hierarchy of a frame to an appropriate place in a sheet hierarchy (and therefore to a host window system window hierarchy) when the frame is adopted.

To build a user interface, an application programmer defines one or more frame 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 hooks for 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.

CLIM is able to show dialog windows, but the code that brings them up is usually quite different from the code that is used to generate the content of application frames. This is unusual for a windowing toolkit as most of them unify the generation of dialog content and content of other window types.

CLIM generates a dialog with the appropriate input gadget as consequence of a series of input requests. Thanks to the stream facility, the programmer can actually request input synchronously with a blocking read request. He does not have to take care of asynchronously handling confirmation or cancel button clicks. For instance, the programmer requests a string from the user and the user is presented with a prompt, an editable text field, and two buttons for confirmation and canceling. The string requesting function returns only after the user hits the confirmation button. The programmer can directly use the function's return value which is the string provided by the user. Clicking the cancel button is dealt with by signaling an abort-gesture condition.

From the caller's perspective, an attempt to separate application frames and dialogs could be: a dialog window itself is side-effect free with respect to the application state and therefore the whole sense of calling a dialog creation routine must arise from the values it returns. For example, the code that modifies the state of the text editor in a text-replace operation does not rest with the callback code of the Ok button in the dialog. This task rests with the code that is executed after the dialog returns its values, namely the code of the Search/Replace command.

An intermediate dialog is something that is brought up to collect additional information for (or before) an operation. When the user selects the "Search" command, he is queried for a search string in an additional dialog window; probably offering other search options like case-insensitive search or backwards search. This is done in a synchronous manner, blocking until the requested input is made available by the user.

An application frame is an interface that provides the user with a variety of commands to choose as his next step. For instance, the user may choose from commands like Open, Save, Search, or Quit. The frame is a long-living GUI object compared to dialogs, and there is no linear execution path as there is in after a dialog as the user is free to select any commands he likes as his next action.

Hence, the synchronous programming pattern for dialogs is more convenient because after dialog confirmations there is a predetermined path of execution, while an application frame has to be prepared to handle an arbitrary sequence of commands.

4.2 Panes

An application frame constructs its pane tree by specifying a top-level pane in define-application-frame. This pane is usually a layout pane that contains more gadget and/or layout panes as its children. With the help of layout panes, a pane hierarchy can be constructed. The top-level pane (and the whole hierarchy when it is a layout pane) is created when the application frame is adopted by a frame manager and made visible to the user. The programmer can compose an interface consisting of pre-defined gadget panes, layout panes, or application-specific panes. CLIM panes are rectangular sheets that are analogous to the gadgets or widgets of other toolkits.

Panes and sheets as defined by the windowing substrate have in common that they are associated with a region on screen, a parent, and optional children. They differ in their usage of the input and output capabilities. A sheet is passive and intended to be used by other, active components, while a pane already contains this active part. For this reason, panes are implemented as subclasses of basic-sheet augmenting the class with an active part. For instance, a button-pane actively draws its own button representation on its allotted screen area and a click on the correct button area triggers a callback for the button. A composite pane lays out its child elements and requests them to draw themselves onto specific screen regions.

CLIM comes with a set of predefined gadget panes. They consist of push-button, toggle-button, slider, radio-box, text-field, text-editor panes ready for use by the CLIM application programmer. These gadgets might be remapped to native system gadgets by the frame manager, so a native look and feel is possible.

Each gadget pane class is associated with a set of

generic functions that act as callbacks do in traditional toolkits. For example, a pushbutton has an "activate" callback method which is invoked when its button is pressed. For this particular callback, a method named activate-callback is invoked by default, and a CLIM programmer can provide a specialized method to implement application-specific behavior for a subclassed button-pane. But except in the case where the programmer needs a lot of buttons with related behavior, creating a subclass for changing a single specific callback is not economical. Hence upon gadget creation, the programmer can specify an alternative callback method for any callback available. For example, by providing the :activate-callback initarg, the programmer can change the callback to any regular or generic function. By convention, any callback can be changed by providing an initarg keyword equal to the callback's name. See Chapter 30 in [MY05] for a listing and description of available callbacks.

CLIM also provides composite and layout panes. These pane types are used for aggregating several child panes into a bigger single pane that has a layout according to the requested directives. For example, CLIM provides two pane classes, hbox-pane and vbox-pane, that lay out their children in horizontal rows or vertical columns respectively. The parent/child relations are managed via the sheet's windowing protocol. If the user interface does not change in ways unpredictable in advance (as in a user interface builder for instance), the program does not have to do hierarchy management via the windowing protocol. He is provided with a set of convenience macros that allows elegant interfaces composed simply by wrapping the respective pane construction code into the convenience macros.

Application pane classes can be used for subclassing. They can be used to present application specific data – for instance by specializing handle-repaint – and to manage user interactions – for instance by specializing handle-event.

4.3 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 separating 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 menus, keyboard accelerators, direct typein, mouse clicks on application data, or gadgets.

The commands are processed in a REPL-like loop. Every application frame has its own running top-level loop specified via :top-level in define-application-frame. For a CLIM application, it is rarely necessary to change the default top level loop.

The top-level loop becomes visible when an interactor-pane is added to the user interface. Here the CLIM user gains direct access to the command loop. The loop steps are similar to read-eval-print:

1. Read a command.
2. Execute the command.
3. Run the *display function* for each pane in the frame associated with the top-level loop as necessary.

Whenever a command is invoked other than by typing, an appropriate command-invoking text appears after the command prompt nonetheless. Here, the user can directly see how his commands are synthesized from other invocation methods like pointer clicks or menu item selections.

5 Simple applications

5.1 Color Editor

In this next example, we define a frame for color selection by manipulating their red, green, and blue components separately. This example illustrates the use of gadget panes provided by CLIM. In the code in Figure 3, we define an application frame using define-application-frame. As said before, the syntax of this macro is similar to that of defclass. It defines a new frame class which automatically inherits from the class frame which provides most of the functionality for handling frames.

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-world frame constructs a hierarchy with only one application pane via make-pane. The pane hierarchy of color editor is more interesting.

Every pane can request space via its initarg. A request for space allocation is specified as a preferred size (:height, :width), a maximum size (:max-height, :max-width), and a minimum size (:min-width, :max-width). The special constant +fill+ can be used to indicate the tolerance for any width or height.

The color editor frame uses three application slots for storing the current RGB values as well as two panes for showing the currently selected colors. The variable *application-frame* is dynamically scoped and is defined in any event handler as well as for all code that is evaluated in the context of the :pane(s) frame option.

The code provided in the :pane option in Figure 3 uses all three kinds of panes provided by CLIM. It uses two application-panes to display colors, two layout panes (vbox-pane and hbox-pane) and three gadget panes (slider-pane).

In contrast to hello-world, we do not need a specialized pane class here. We can use an application-pane for displaying chosen colors. An application pane supports graphics operations and invokes generic functions on the pane when input events are received. For the color editor, we only need its ability to refresh its background color.

The vertically and horizontally convenience macros provide an interface to the hbox-pane and vbox-pane classes. Most CLIM layout panes provide similar convenience macros. 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.

In the horizontally macro in Figure 3, we do not supply forms to be evaluated directly. Instead, horizontally processes a form wrapped up in a list. The first list element is a rational number which denotes the amount of space the pane generated by the following form is allowed to occupy in the resulting horizontal layout. In our code in Figure 3, hbox-pane generated from the horizontally macro has a space requirement on its own. The whole composite pane is forced to have a size 200 pixels high.

Now we are ready to turn to the gadget panes. The color editor uses three sliders one for each component of RGB. make-color-slider creates all three sliders that differ by id, initval and label. The first two variables are handed to make-pane to construct the slider panes. The remaining initargs for make-pane are shared across all three entities. To decorate each slider-pane with a proper label, each of them is wrapped up in a label-pane via the labelling convenience macro.

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.

We use the gadget ID to distinguish between the red, green and blue slider in the callback code. We could use three different callback functions here, but such callbacks would have much more similarities than differences, thus we do not do that here. Instead, we distinguish the gadget by their id.

The drag-callback method resets the background of drag-feedback-pane and delegates its redrawing to redisplay-frame-pane, while value-change-callback does the same for the frame's current color pane. These methods use the id argument of the gadget to determine which color component was changed.

The color editor frame uses the :menu-bar option to indicate that a menu-bar should be added to the application frame. Frame commands defined with :menu t are accessible from the menu-bar. In this example, we define only one command named com-quit which is presented as "Quit" to the user. com-quit is defined via define-color-editor-command. This macro is generated automatically along define-application-frame from the application frame's name. com-quit simply closes the applica-

```
  (labelling (:label label)
    (make-pane :slider :id id :orientation :horizontal :value initval
                       :max-value 1 :min-value 0
                       :show-value-p t :decimal-places 2
                       :drag-callback #'color-slider-dragged
                       :value-changed-callback #'color-slider-value-changed)))

(define-application-frame color-editor ()
  (current-color-pane
   drag-feedback-pane
   (red :initform 0.0)
   (green :initform 1.0)
   (blue :initform 0.0))
  (:pane (with-slots (drag-feedback-pane current-color-pane red green blue)
           *application-frame*
           (vertically ()
             (setf current-color-pane
                   (make-pane 'application-pane
                              :min-height 100 :max-height 100
                              :background (make-rgb-color red green blue)))
             (horizontally (:min-height 100 :max-height 100)
               (1/2 (make-color-slider 'red red "Red"))
               (1/4 (make-color-slider 'green green "Green"))
               (1/4 (make-color-slider 'blue blue "Blue")))
             (setf drag-feedback-pane
                   (make-pane 'application-pane
                              :min-height 100 :max-height 100
                              :background (make-rgb-color red green blue))))))
  (:menu-bar t))

(defun color-slider-dragged (slider value)
  (with-slots (drag-feedback-pane red green blue) *application-frame*
    (let ((color (ecase (gadget-id slider)
                   (red (make-rgb-color value green blue))
                   (green (make-rgb-color red value blue))
                   (blue (make-rgb-color red green value)))))
      (setf (pane-background drag-feedback-pane) color
            (medium-background drag-feedback-pane) color))
    (redisplay-frame-pane *application-frame* drag-feedback-pane)))

(defun color-slider-value-changed (slider new-value)
  (with-slots (current-color-pane red green blue) *application-frame*
    ;; The gadget-id symbols match the slot names in color-editor
    (setf (slot-value *application-frame* (gadget-id slider)) new-value)
    (let ((color (make-rgb-color red green blue)))
      (setf (pane-background current-color-pane) color
            (medium-background current-color-pane) color))
    (redisplay-frame-pane *application-frame* current-color-pane)))

(define-color-editor-command (com-quit :name "Quit" :menu t) ()
  (frame-exit *application-frame*))
```

Figure 3: Color Editor

tion frame causing the termination of the frame's top-level-loop. In the next example, commands will be explored in greater detail.

We can invoke the color-editor with the regular run-frame-top-level/make-application-frame combination.

```
(run-frame-top-level
  (make-application-frame 'color-editor))
```

5.2 A simple drawing application

We move on to a simple application that draws lines and inserts text interactively. Our simple drawing program defines commands for various drawing operations and binds specific input events to these commands.

The application frame is defined in Figure 4. A different approach is used to generate the pane hierarchy: instead of mixing the layout and pane information, we use the :panes keyword to list all panes that should be available in frame layouts. The :layouts keyword combines them into several layouts. It is possible to define multiple layouts with this option. We define two simple layouts, default-layout and alternative.

There are two options to define-application-frame that we have not seen before, :command-definer and :top-level (both with there defaults). :command-definer is used to specify a name for a command-defining macro for this frame class. Passing t to this option (its default) generates a command definer named define-<frame-name>-command. We can use the command-defining macro as a convenience macro for define-command which is used to define frame commands, and will see it in action before long. :top-level specifies a special form that is used as top level command loop. The top level is responsible for dequeuing and executing commands that have been invoked by the user. This loop is to a application frame what the REPL is to a terminal.

This is the first example that does not use clim-stream-pane (or one of its subclasses) as a pane class.[13] Instead, we compose our own drawing pane using standard-extended-input-stream, basic-pane and permanent-medium-sheet-output-mixin. The first class is used to provide the application programmer with stream properties for the draw-pane that are required for the convenience macro tracking-pointer. basic-pane is responsible for handling all requests belonging to the sheet and pane protocols in a standard manner. The last class is a mixin to tag the pane as permanently visible on screen during its instances' lifetime. Most static user interfaces use this mixin. Note that McCLIM is sensitive to the order of the superclasses.[14]

For draw-pane, the handle-repaint method shown in Figure 4 is straightforward. It delegates background filling to the next less specific method and then iterates through the lines and strings painting them. Here, we implicitly defined the format of the lines slot and the strings slot of the application frame class. The elements of the list stored in lines are pairs of points, namely the start and end point for a line. In strings, we store the text's position as the car of a cons and the text as its cdr.

Figure 5 shows the command definitions. We have commands for adding a text string, for adding a line and resetting the draw pane. After every command, we update the drawing pane via the auxiliary method update-draw-pane.[15]

At this point, we can actually use the application; although not very conveniently. The interactor pane can be used to invoke one of three commands, either by typing the complete command including all its parameters or by typing only the command name, then a dialog queries the user for the missing command argument(s). Clicking on the menu bar entries is another way to invoke commands. When commands are invoked through the menu bar, the user will be queried for the missing argument(s) in the same way as if a command had been typed into the interactor pane.

But drawing by typing coordinates is not convenient. Therefore, we attach these commands to other user interactions. Figure 6 defines input methods (methods for handle-event) for pointer button presses as well as key presses on the draw pane. Each handler invokes a tracking function

[13] When using McCLIM, we have to do this as there are bugs in the behavior of clim-stream-pane that have not been fixed yet.

[14] All stream classes like standard-extended-input-stream must be listed before basic-pane. Otherwise, no stream handling facilities will be available.

[15] An experienced CLIM programmer would define a display-function for draw-pane. This function is run after a command is executed, and causes the display pane to be updated with any changes. We will save this technique for later examples.

```
(define-application-frame draw-frame ()
  ((lines :accessor lines :initform nil)            ; lines of drawing
   (strings :accessor strings :initform nil))       ; texts of drawing
  (:panes (draw-pane (make-pane 'draw-pane))
          (interactor :interactor))
  (:layouts (default-default (vertically ()
                               (:fill draw-pane)
                               (1/4 interactor))))
  (:menu-bar t)
  (:command-definer t)
  (:top-level (default-frame-top-level)))

(defclass draw-pane
    (standard-extended-input-stream ; must have precedence over basic-pane
     basic-pane
     clime:always-repaint-background-mixin
     permanent-medium-sheet-output-mixin)
  ())

(defmethod handle-repaint ((pane draw-pane) region)
  (with-application-frame (frame)
    (call-next-method)                  ; Paints the background
    (dolist (line (lines frame))
      (draw-line pane (car line) (cdr line)))
    (dolist (pair (strings frame))
      (draw-text pane (cdr pair) (car pair)))))
```

Figure 4: define-application-frame for draw-frame

(track-line-drawing and track-text-drawing) that uses tracking-pointer to bypass the regular input distribution channels and to dispatch events to user defined handlers.

For pointer-button-press-event, which is used to draw lines, the input loop manages a "rubber-banding" line. The :pointer-motion is invoked whenever the mouse pointer is moved by the user. The code attached to :pointer-motion clears the previously-drawn line and draws a new line with the new pointer position. It can easily undraw the old line by the using the special ink +flipping-ink+. When the user confirms the line by releasing the pointer button, a command to the application is synthesized via execute-frame-command supplying all required parameters.

We provide a similar input facility for text input. Whenever the user hits a key in the draw-pane, the respective handle-event calls track-text-drawing which attaches the character entered to the mouse pointer. As with the rubber-banding line, the user can move the displayed string around while he is free to append additional string characters by additional key presses. He can confirm the text position with a mouse click causing a command to be dispatched to the application frame that will add the text to the application frame permanently.

As each of these methods invoke execute-frame-command passing in a special command invocation form, this naturally leads to a separation between the code that specifies how a command is invoked (menu-bar click, click on draw-pane or typing in the interactor pane) and the code for command execution (code bodies of define-command).

6 High Level Facilities

In this section, we explain a number of higher level facilities provided by CLIM, including output recording, formatted output, presentations, context

```
(define-draw-frame-command (com-draw-add-string :menu t :name "Add String")
    ((string 'string) (x 'integer) (y 'integer))
  (push (cons (make-point x y) string)
        (strings *application-frame*))
  (update-draw-pane))

(define-draw-frame-command (com-draw-add-line :menu t :name "Add Line")
    ((x1 'integer) (y1 'integer) (x2 'integer) (y2 'integer))
  (with-slots (lines) *application-frame*
      (push (cons (make-point x1 y1) (make-point x2 y2))
            lines))
  (update-draw-pane))

(define-draw-frame-command (com-draw-clear :menu t :name "Clear") ()
  (with-slots (lines strings) *application-frame*
    (setf lines nil strings nil))
  (update-draw-pane))

;; Auxilary Method
(defun update-draw-pane ()
  (repaint-sheet (find-pane-named *application-frame* 'draw-pane) +everywhere+))
```

Figure 5: Commands for draw-frame

sensitive input, and command processors.[16] We illustrate these facilities in two examples: a directory lister and a simple schedule browser.

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). 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 rearranged.

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, for instance row and column formatting information. Chapter 16 in [MY05] has all the details.

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 T 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)
```

[16] Many of these facilities are derived from work done at Symbolics on the Dynamic Windows (DW) project for Genera[Sym]. See [MYM89] 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.

```
(defmethod handle-event ((pane draw-pane) (event pointer-button-press-event))
  ;; Start line tracking when left pointer button is pressed
  (when (eql (pointer-event-button event) +pointer-left-button+)
    (track-line-drawing pane
                        (pointer-event-x event)
                        (pointer-event-y event))))

(defmethod handle-event ((pane draw-pane) (event key-press-event))
  (when (keyboard-event-character event)
    (multiple-value-bind (x y) (stream-pointer-position pane)
      ;; Start with empty string, as a key release event will be received anyway
      (track-text-drawing pane "" x y)))
  (update-draw-pane))

(defun track-line-drawing (pane startx starty)
  (let ((lastx startx)
        (lasty starty)
        endx endy)
    (block nil
      (with-drawing-options (pane :ink +flipping-ink+)
        (draw-line* pane startx starty lastx lasty)
        (tracking-pointer (pane)
          (:pointer-motion (&key x y)
            (draw-line* pane startx starty lastx lasty) ; delete old
            (draw-line* pane startx starty x y)         ; draw new
            (setf lastx x lasty y))
          (:pointer-button-release (&key event x y)
            (when (eql (pointer-event-button event) +pointer-left-button+)
              (draw-line* pane startx starty lastx lasty) ; delete old
              (setf endx x endy y)
              (return))))))
    (execute-frame-command
     *application-frame* `(com-draw-add-line ,startx ,starty ,endx ,endy))))

(defun track-text-drawing (pane current-string current-x current-y)
  (tracking-pointer (pane)
    (:pointer-motion (&key x y)
      ;; We can't use flipping ink for text, hence redraw.
      (handle-repaint pane +everywhere+)
      (setq current-x x current-y y)
      (draw-text* pane current-string x y))
    (:keyboard (&key gesture)
      (when (and (typep gesture 'key-release-event)
                 (keyboard-event-character gesture))
        (setf current-string
              (concatenate 'string
                           current-string
                           (string (keyboard-event-character gesture))))
        (handle-repaint pane +everywhere+)
        (draw-text* pane current-string current-x current-y)))
    (:pointer-button-release (&key event x y)
      (when (eql (pointer-event-button event) +pointer-left-button+)
        (execute-frame-command *application-frame*
          `(com-draw-add-string ,current-string ,x ,y))
        (return-from track-text-drawing nil)))))
```


Figure 6: User Interfaces

```
  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 too much space for the typical case. In CLIM, we can use the code in Figure 7 to produce a neatly formatted table for any set of package names.

Presentations The next step up from preserving 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, 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 insufficient 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 be just 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 display of possibilities. 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 output is semantically-tagged, it can be reused as semantically-meaningful input. To achieve this, the application does not only have to tag its output but it also has to provide additional semantic information when doing input operations. For

```
(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 7: An output function that uses table formatting.

instance, whenever a pathname is required, the program has to use a special method in contrast to reading a string of characters.

The counterpart to present is accept. It is used to establish an *input context*. The input context is a presentation type that is appropriate for the current input point. For example, if the application requires the user to input a pathname, it can trigger an appropriate prompt, input parser and input context with:

```
(accept 'pathname :stream s)
```

Typically, this invokes the input reader (or parser) that was defined for the pathname type and establishes an input context that indicates that it is waiting for a pathname.

Once the input context is established, CLIM automatically makes any appropriate existing output available to the user via mouse gestures. After calling accept as shown above, 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 CLIM promotes the separation of command execution code and command invocation code. The command facility aids this task. Every application frame can define application commands that are accessible via various input methods. The most common are clicking on the command's entry in the menu bar or a context menu, typing the command in the interactor pane, or the application dispatches a command to itself (maybe triggered by other ways of user input or network interactions, etc.). The latter case is seen in track-line-drawing and track-text-drawing of the draw-frame example. These methods generate "Add Line" and "Add String" commands as the result of event-handling on the draw-pane.

A command has a name specified as string and a set arguments specified as presentation types. Looking back at the command "Add String" of draw-frame, we see that this command takes a string and two integer as arguments. This type information is useful for partial command parsing. Assume the user clicks on a menu entry or types only the command name in the interactor. CLIM notices that there are arguments missing in the command and requests the missing ones in a dialog via calls to accept. Mentioned early, accept establishes an input context and so the user is able to fill the missing argument with clicks on appropriate visible output objects. Also keyboard users will find semantically-tagged input methods convenient as implementations of presentation types can provide the user with a completion facility.

In addition, the command processor 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.

```
(define-application-frame file-browser ()
  ((active-files :initform nil :accessor active-files))
  (:panes
   (file-browser :application
                 :display-function '(dirlist-display-files)
                 ;; Call the display-function whenever the command
                 ;; loop makes a ``full-cycle''
                 :display-time :command-loop)
   (interactor :interactor))
  (:layouts (default (vertically ()
                       (:fill file-browser)
                       (1/4 interactor))))
  (:menu-bar nil))

(define-presentation-type dir-pathname ()
  :inherit-from 'pathname)

(defmethod dirlist-display-files ((frame file-browser) pane)
  ;; Clear old displayed entries
  (clear-output-record (stream-output-history pane))

  (dolist (file (active-files frame))
    ;; Instead of write-string, we use present so that the link to
    ;; object file and the semantic information that file is
    ;; pathname is retained.
    (present file
             (if (cl-fad:directory-pathname-p file) 'dir-pathname 'pathname)
             :stream pane)
    (terpri pane)))

(define-file-browser-command (com-edit-directory :name "Edit Directory")
  ((dir 'dir-pathname))
  ;; the following was a previous attempt to deal with the oddities of
  ;; CL pathnames.  Unfortunately, it does not work properly with all
  ;; lisp implementations.  Because of these oddities, we really need
  ;; a layer like cl-fad to keep things straight. [2007/01/05:rpg]
;;;  (let ((dir (make-pathname :directory (pathname-directory dir)
;;;                            :name :wild :type :wild :version :wild
;;;                            :defaults dir)))
    (setf (active-files *application-frame*)
          (cl-fad:list-directory dir)))

(define-presentation-to-command-translator pathname-to-edit-command
    (dir-pathname                          ; source presentation-type
     com-edit-directory                    ; target-command
     file-browser                          ; command-table
     :gesture :select                      ; use this translator for pointer clicks
     :documentation "Edit this path")      ; used in context menu
    (object)                               ; argument List
    (list object))                         ; arguments for target-command

(define-file-browser-command (com-quit :name t) ()
  (frame-exit *application-frame*))

(defmethod adopt-frame :after (frame-manager (frame file-browser))
  (declare (ignore frame-manager))
  (execute-frame-command frame
        `(com-edit-directory ,(make-pathname :directory '(:absolute)))))
```


Figure 8: File Browser

6.1 A Directory Browser

The dirlist-frame application is a very simple file system browser using presentation types. It defines two panes, an output pane to display directory contents and an input pane to handle user typein. For the output pane, the application defines a display function. dirlist-display-files works in conjunction with the top level command loop. By default, the command loop calls all display functions after a command was processed to make all changes in the application's data structures visible that the command execution might have caused. When all display functions have updated their panes, the command loop displays a prompt in the interactor[17] and waits for the next command.

The dirlist-display-files display function iterates over the contents of the current directory displaying the files one by one. Each output line is produced by a call to present. present creates the association between the text lines on the screen and the Lisp pathname objects.

draw-frame has a single command "Edit directory". The command's body interprets the pathname that it receives as a directory, obtaining a list of the files contained therein. The command simply updates the application variable active-files with the new list.

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 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. The translator would consult the system's user database to retrieve the home directory information to achieve its conversion task.

The command loop of an application frame is regularly requesting commands for processing and it uses accept for its request. Hence, an input context for reading an object with the presentation type command is established and all objects that can be used as commands will become clickable on screen. Here, we can define a presentation-to-command translator that translates a pathname into the edit directory command.

The presentation-to-command translator in Figure 8 is very simple. It has the name pathname-to-edit-command and converts the presentation type pathname to the command com-edit-directory for the command table of file-browser. The gesture option specifies that it works with the select gesture, while the string supplied to :documentation is used in the context menu.

This command translator has little work to do. The body of the translator has to return a list of arguments that are handed to the command com-edit-directory. But we do not need to any conversion of the supplied object, as it is already a pathname. Thus, the object is wrapped in a list and returned to the caller which will apply com-edit-directory to the list.

A simple trick is used to dispatch an initial command to the application frame. An after-method is provided for adopt-frame which runs after the frame manager has adopted the frame for displaying. This is different for an after-method on initialize-instance as an after-method on adopt-frame runs later, when the application frame instance has already a command queue associated with it.

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." Here, CLIM supports for automatic command completion becomes visible. Only the first letter has to be typed followed by the complete action (usually Tab) and the command completion facility will complete the input to "Edit Directory". 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.

6.2 Schedule Example

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 9, represents a day of the week as a number from 0 to 6. Each day number is associated with an abbreviated day name, "Mon," "Tue," etc.

[17] Omitted when there is no interactor pane.

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

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

(define-presentation-type weekday ())

(define-presentation-method accept
    ((type weekday) stream (view textual-view) &key)
  (values (completing-from-suggestions (stream)
            (dotimes (i 7)
              (suggest (aref *days* i) i)))))

(define-presentation-method present
    (daynumber (type weekday) stream (view textual-view) &key)
  (write-string (aref *days* daynumber) stream))


(define-application-frame scheduler ()
  ((appointments :initarg :appointments :initform *test-data*)
   (current-day :initform nil))
  (:panes (scheduler-display :application
                             :display-function '(display-appointments))
          (interactor :interactor))
  (:layouts (default-layout
             (vertically ()
               scheduler-display
               interactor))
            (alternative-layout
             (horizontally ()
               interactor
               scheduler-display)))
  (:menu-bar t))

;;; Chooses which day to see in detail,
(define-scheduler-command (com-select-day :name t :menu t)
    ((day 'weekday :gesture :select))
  (with-slots (current-day) *application-frame*
    (setq current-day day)))

;;; Show weekly summary.
(define-scheduler-command (com-show-summary :name t :menu t) ()
  (with-slots (current-day) *application-frame*
    (setq current-day nil)))

(define-scheduler-command (com-toggle-layout :name t :menu t) ()
  (with-accessors ((layout frame-current-layout)) *application-frame*
    (setf layout (if (eq layout 'default-layout)
                     'alternative-layout
                     'default-layout))))
```

Figure 9: Scheduler: application frame, presentation type and commands

```
;;; Complex display function, shows two completely different displays.
(defmethod display-appointments ((frame scheduler) pane)
  (clear-output-record (stream-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)
      (with-drawing-options (pane :text-face :bold)
        (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 (pane (if (eql day current-day) ':bold ':roman))
       (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 10: Scheduler: display functions

The weekday presentation type defines two basic pieces of behavior: how a weekday is displayed, and how it is read as input. The macro define-presentation-method is intended for specializing methods on presentation types. Via this macro, we define a printer and a parser function for the type weekday. 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. accept does not need a parser in every case, and for simplicity the programmer might choose not to provide a parser at all.

We took the easy way out with our parser. CLIM provides a completion facility and instead of reading the day name of the stream and parsing it into an integer, we provided an exhaustive set of input to object mappings. The completing-from-suggestions macro collects all suggestions made by the parser via suggest. suggest takes an input and object as suggestion. In our case, day name taken from *days* is the input and the number of the week i is the object. CLIM will match the input given by the user with the suggestions and in case of a match will return the corresponding object to the caller.

We define an application frame for the appointment browser in Figure 9. The frame defines state variables to hold the list of appointments and the current day. This information is kept in slots on the frame, 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. 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.

The command "Select Day" has the following argument specification:

```
((day 'weekday :gesture :select))
```

It takes an argument day of type weekday. Up until now, our command arguments have looked similarly to specialized lambda lists, but next to the type information the arguments of commands are also allowed to have various keyword-value pairs for selection further options. In this case, we supplied the value :select for the option :gesture. The macro define-command parses this keyword-value pair and generate a presentation translator. More precisely, a presentation-to-command translator is defined that is equal in functionality to the one we have seen in the file-browser example. Whenever a presentation of the type pathname is selected (e.g. with pointer clicks) in a command input context, it is translated into a command invocation of "Select Weekday".

Finally, we turn to the display the appointment information. The display function, display-appointments, shown in Figure 10, 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.

display-appointments decides which set of information to display by examining the application state variable current-day. The table formatting facility is used to present the weekly summary information neatly organized. The daily appointment list, by contrast, is displayed using write-string. 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 application with interesting UI behavior can be constructed from a high-level specification of its functionality.

7 Conclusion

The series of examples presented in this article illustrates the broad range of functionality provided by CLIM. The later examples, especially, demonstrate that complex user interfaces can be built econom-

ically and in a modular fashion 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, these higher level facilities are not appropriate for all programmers. CLIM's lower level facilities and clean modularization of the higher level facilities provide these programmers with portable platform and a framework for implementing their own 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.

A free CLIM implementation is available as McCLIM, found at `http://common-lisp.net/project/mcclim/`. In addition to the caveats in this document, note that McCLIM only works with an X windows backend as of January 2006.

Acknowledgments

Original article 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.

2006 update Clemens Fruhwirth thanks the developers for McCLIM for producing a free CLIM implementation, and especially Robert Strandh for answering so many questions in conjunction with McCLIM.

References

[McC] McCLIM. A free CLIM implementation.

[MY05] Scott McKay and Wiliam York. Common lisp interface manager specification, 2005.

[MYM89] Scott McKay, William York, and Michael McMahon. A presentation manager based on application semantics. In *Proceedings of the ACM SIG-GRAPH Symposium on User Interface Software and Technology*, pages 141–148. ACM Press, November 1989.

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