Chiron-1 User Manual

version 1.4

Kari Nies
Craig MacFarlane
Mary Cameron
Gregory Bolcer

Arcadia Document UCI-93-07

Department of Information and Computer Science
University of California, Irvine [1]

September 24, 1993

[1]This material is based upon work sponsored by the Defense Advanced Research Projects Agency under Grant Number MDA972-91-J-1010. The content of the information does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Revisions

Version 1.3 – June 1992

- First external distribution of Chiron 1.
- Major performance enhancement from Chiron 1.2
- Includes both Xview and Motif implementation of ADL

Version 1.3(b) – January 1993.

- Support for multiple servers on a single machine with use of the CHIRON_SERVER_NUMBER environment variable (optional)
- Modified artist template. Pre-existing .cal files must be regenerated or edited. (fixes Set_Behavior deadlock bug)
- New methods, *Load_File* and *Save_File*, for adl text_window class
- No-flicker redraw for canvases.
- Corrections to ADL Reference Manual (appendix E)

Version 1.4 – September 1993

- ADL
  - New spline class
  - New GIF class
  - New scrollable list class (Motif only)
  - New colormap (64 colors)
  - Support for multiple displays (Motif only)
  - New resize event (Motif only)
  - New set keyboard focus (Motif only)
  - Unselectable (grey) menu options (Motif only)
  - Select events detected anywhere on polylines and splines
  - One or two button notices for Motif
  - New get_display_depth method for base frames
- Client architecture
  - Universal artist specification
  - Artists can now monitor multiple ADTs
  - Extendible client event type definition

  - Dynamic configuration
  - Flexible dispatching architecture
  - Fine grain client event registration
  - Application main procedure now loaded as main executable
  - Support of artist and client shutdown

- System dependency upgrades
  - SunAda 1.1
  - gcc v.2.4.5
  - X11R5
  - Openwindows 3.0
  - Motif 1.2

- *Lots* of bug fixes

- New Makefiles for the installation

Contents

[1] The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

List of Figures

1 Introduction

Chiron is a User Interface Development System (UIDS) for software environments. A UIDS, as phrased by Myers [Mye89], is an integrated set of tools that help programmers create and manage many aspects of interfaces.

Chiron's focus is on user interface and application architectures. It takes a software engineering approach to user interface development by creating interface layers which are resilient to change. In particular, two main goals have been the separation of an application from its user interface code, as well as a separation between the user interface code and the underlying toolkit substrates. *Chiron* supports the construction of graphical user interfaces that provide multiple coordinated views of application objects and allows flexible restructuring of the configuration of those views.

This manual is intended as an aid to *Chiron* user interface builders. Although a brief description of the *Chiron* architecture is provided, this manual does not thoroughly discuss the design, rational, and philosophy behind *Chiron*. For a more in-depth discussion of *Chiron*, the reader is refered to [KCTT91, TJ93].

1.1 The Chiron Release

This manual is distributed with the *Chiron* 1.4 release, which runs on Sun 4 workstations with version X11 release 5 of the X Window System. This release uses both the XView 3.0 and Motif 1.2 toolkits, and therefore can support either an OPEN LOOK or a Motif look and feel. *Chiron* users will need at least one of these toolkits in order to install the system. *Chiron* has been built using the SunAda compiler, version 1.1, and version 2.4.5 of the GNU gcc compiler. The *Chiron* 1.4 release is available via ftp upon request.

1.2 Support Information

If you should have problems installing *Chiron*, or have any questions, suggestions or comments, please contact:

Arcadia Project
Att: Chiron Support
University of California
Irvine, CA 92717

email : *arcadia-chiron-bugs@ics.uci.edu*
email : *arcadia-chiron@ics.uci.edu*

The *arcadia-chiron-bugs* mailing list is intended for bug reports, and installation problems. The *arcadia-chiron* mailing list is intended for discussing design issues, suggesting new features, and general comments.

A bug report form is included with the release. Please use this form when reporting bugs in the system.

1.3 Organization of the Manual

This manual should give the reader a basic understanding of the *Chiron* user interface development system and should provide enough information for the user to build *Chiron* clients on their own. If you discover deficiencies in the manual please report them to us via the standard support channels.

Section 2 provides a brief overview of the *Chiron* approach to user interface development systems. Here we present the conceptual architecture of the system and describe the runtime components of a *Chiron* client.

The *Chiron* artist encapsulates knowledge about how to render the depiction of an object (or rather, an Ada ADT). The artist must respond to programmatic state changes in the object as well as to user manipulations of its depictions. Section 3 gives a quick overview of a *Chiron* artist, section 5 explains artist event processing, and section 6 describes the Lo-CAL artist language.

Chiron artists create graphical user interfaces by making calls into an Abstract Depiction Language, or ADL. The ADL is an object-oriented class hierarchy that defines all of the graphical objects that *Chiron* supports, and the methods available for creating, manipulating, and destroying those objects. An overview of the ADL hierarchy is given in section 4.

Section 7 gives detailed step by step instructions on how to write a *Chiron* artist, including source code examples, and section 8 gives instructions on how to generate, compile, and execute a *Chiron* client. Integration with the Sun Microsystems graphical user interface builder, DevGuide, is discussed in section 9.

Section 10 discusses known defects, unimplemented features and desired enhancements, and section 11 gives some assistance in trouble shooting the system.

Appendix A gives complete source code listing for the simple artist example discussed in section 3. Appendix B lists the *chiron_standard_library* package which artists must include in order to access the ADL hierarchy. Appendix C gives complete source code listing for the artist example discussed in section 7. Appendix D gives complete source code listing for an association tables generic specification for maintaining relations between graphical and program objects. Finally, appendix E is a listing of the *ADL Reference Manual*.

2 Chiron System Overview

The development of any large system poses a group of well-known software engineering problems including maintainability, resilience to change, and reusability. The objective of the *Chiron-1* project has been to develop a user interface technology that takes these concerns into consideration.

To address these issues, the *Chiron* architecture stresses a separation of concerns; both between the application and the user interface, and between the user interface and the underlying toolkits and windowing substrates. It is this separation that has motivated the *Chiron* design.

The benefits of this separation are many. The user interface and application can evolve independently of each other. Application programmers may develop and test their work without having to deal with user interface issues. The user interface developer's domain is encapsulated neatly within the confines of an *Artist*. In addition, the artist writer may focus on issues of presentation, choosing between different look-and-feels without having to be intimately familiar with various toolkits and windowing systems. The user interface developer is also insulated from changes in these underlying substrates.

2.1 Artists and ADTs

Before we can discuss the *Chiron* architecture, it is important to understand the assumptions *Chiron* makes about applications and the basic ideas behind artists and annotation.

The *Chiron* model requires that applications are ADT-based [Gut77]: the application's data and functionality should be organized as abstract data types (ADTs). In particular, objects should be explicitly created, modified, queried, and destroyed with procedure or functions calls. This prerequisite insures that any object modification occurs through a clean functional interface (and thus can be augmented graphically by the artist). Writing interfaces for ADT-based applications entails writing *artists* for the ADTs that make up the application.

Artists are generally written to depict ADTs; they encapsulate decisions on how to present and manipulate the ADTs graphically. More precisely, an artist maintains the bi-directional mapping between ADT objects within the application and visual objects on the screen. Thus, modifications to the ADT will be reflected on the display, and vice versa. Multiple artists may depict a single ADT, and a single artist may depict the state of multiple ADTs.

Artists are notified whenever an ADT is modified and respond by updating their depictions accordingly, in essence, annotating graphics to the operations on the ADT. The essential characteristic of annotation as a mechanism for binding artists to ADTs is that neither the semantics nor the syntax (signatures of operations) of the ADTs are changed. Application code need not be modified to add ADT visualization and ADT functionality. Thus, the interface to the functional parts of an application is not corrupted by the user interface. Annotation is achieved using *Chiron's* wrapping and dispatching mechanisms, described in the following sections.

It is important to note that an artist may receive events other than ADT state changes,

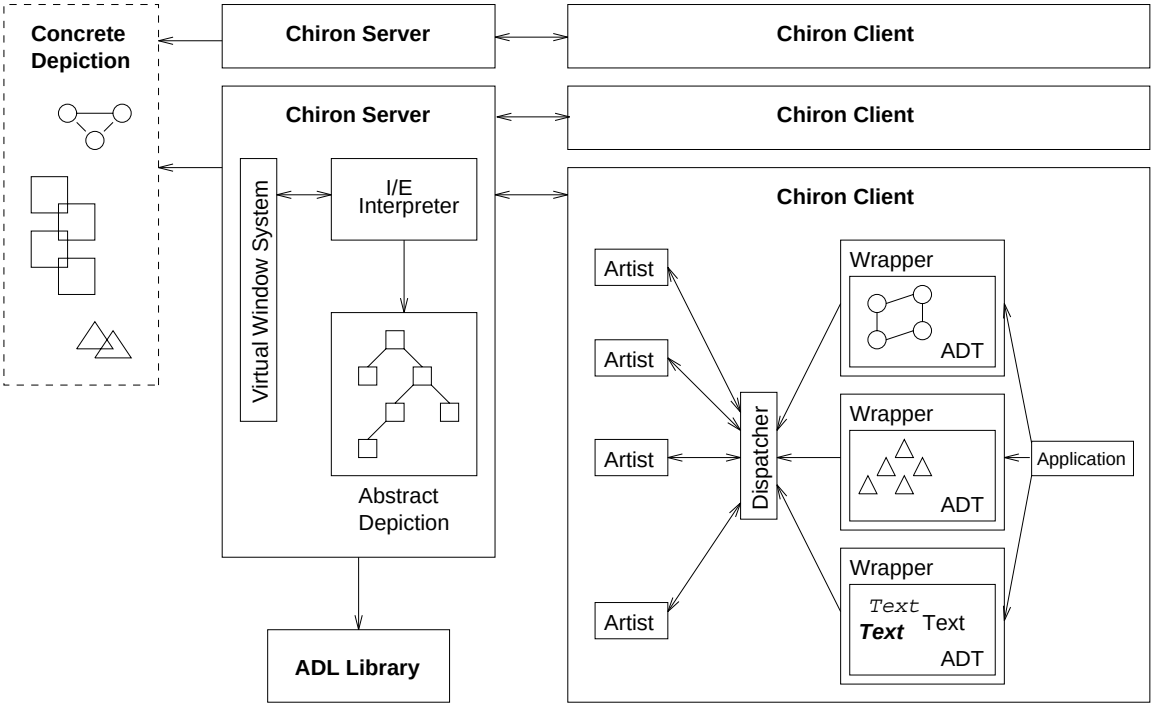

Figure 1: *Chiron's* Conceptual Architecture

including hand-coded events. In fact, an artist may not have any associated ADT. However, for object(ADT)-based applications, we strongly advise using an ADT-based design, as it fosters maintainability and reusability, and it is facilitated by the *Chiron* architecture.

2.2 Conceptual Architecture

Chiron's conceptual architecture is illustrated in figure 1. *Chiron* is a serverized system: the *Chiron* server and a *Chiron* client run in separate Unix processes. The *Chiron* server manages all aspects of a user interface that are not artist or application specific. It can be thought of as a virtual machine providing a high-level interface to artists in the form of an Abstract Depiction Language (ADL). It receives ADL instructions and uses them to create and manipulate an internal data representation called the abstract depiction. The abstract depiction is rendered to a concrete depiction (a user viewable image) via calls to the underlying window system. The server also listens for events from the window system (button push, menu selection, etc.) and translates them to *Chiron* events before shipping them back to the appropriate client.

A *Chiron* server can support up to ten clients running on separate machines. There can be multiple *Chiron* servers per machine. Each client, however, may communicate with only one server.

The *Chiron* server provides flexibility in terms of windowing systems and toolkits, application languages, and process inter-connection topology. Although we currently only

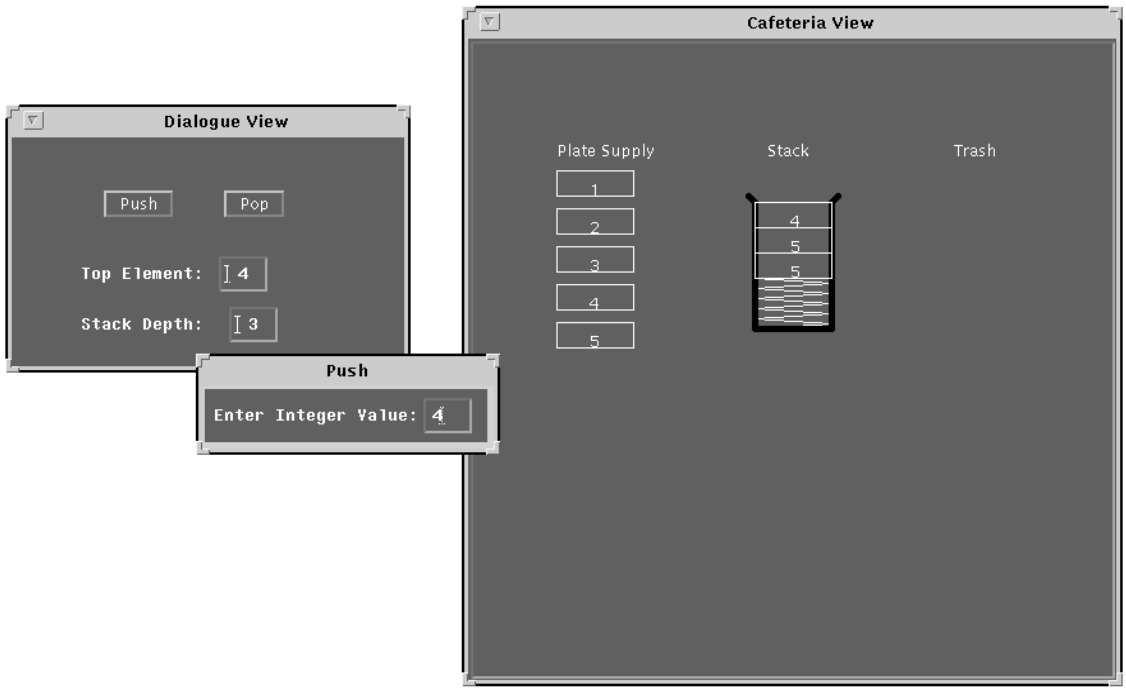

Figure 2: This *Chiron* user interface provides two coordinated views of the same stack, via two artists monitoring the same ADT instance. The leftmost artist (two windows) provides a dialogue view of the stack, while the rightmost artist provides a cafeteria view of the same stack.

provide support for Ada clients, it is feasible to provide other language-specific interfaces to the server's ADL.

A *Chiron* client is comprised of the application, the ADTs to be depicted and artists for those ADT's, and some coordinating runtime components. The *Chiron* method of annotation requires that each depicted ADT be "wrapped" by a listening agent. These listening agents, or *wrappers* intercept calls to the ADT, and perform the requested operation. The wrapper maintains the interface to the ADT so that the application need not be modified to accommodate the user interface. The application or artists make calls to the ADT, indirectly, through the wrapper. Once the wrapper has performed the requested ADT operation, it constructs an event that represents the operation and notifies the client dispatcher of that event. The client dispatcher, in turn, relays the event to any interested artists. Artist register their interests in specific events with the client dispatcher.

This design supports multiple coordinated views of instances of the same ADT. Figure 2 shows a *Chiron* user interface to an application that manipulates a stack. Two artists

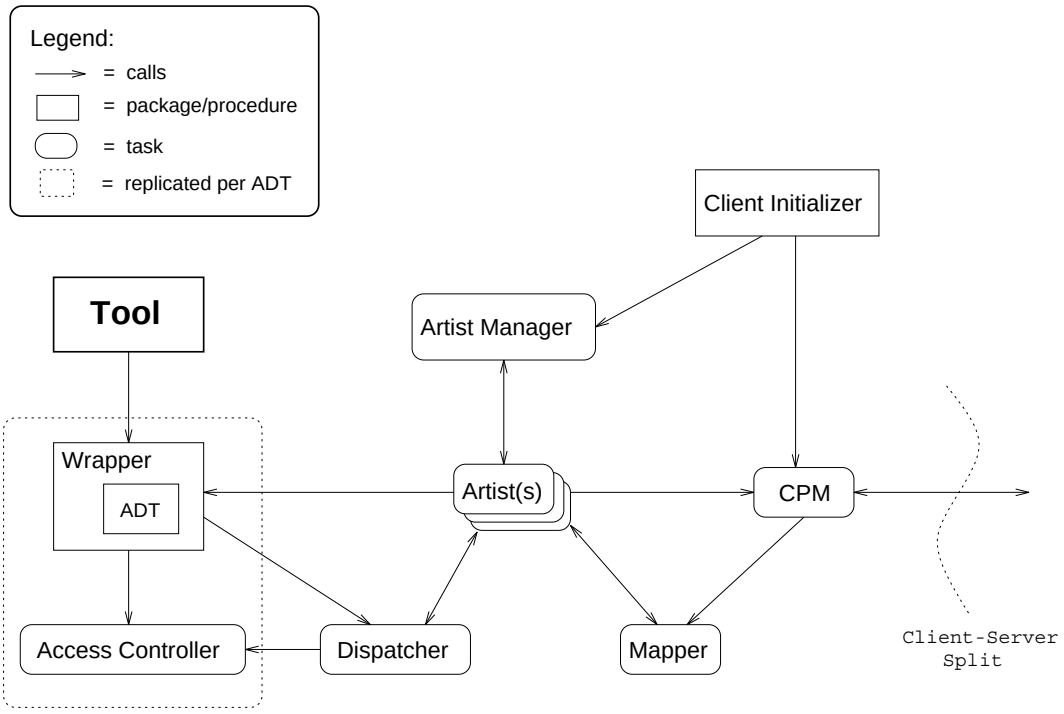

Figure 3: Runtime Architecture of a *Chiron* Client

provide two coordinated views into a single stack ADT instance. The artist on the left provides a dialogue view of the stack, while the artist on the right provides a cafeteria view of the same stack. The *Chiron* client architecture allows for easily reconfigurable user interfaces. Artists become plug compatible, highly reusable entities when bound to a single ADT. User interfaces for existing applications may be built with little or no restructuring of the application.

The *Chiron* model is highly concurrent. Most components maintain their own, and possibly multiple, threads of control. Thus, unlike most user interface callback architectures, we avoid imposing sequential control upon applications, allowing the application, the user interface, and the server, to run in parallel. Because of this pervasive concurrency, the client implement concurrency control for ADTs. For example, the wrapper must guard against simultaneous access of the ADT; it only allows concurrent read operations and exclusive write operations. Also, in order to maintain consistency between the ADT state and its depictions, the dispatcher ensures that all artists have been notified of a change in ADT state before another change of state can be performed.

2.3 Runtime Architecture of a Client

A *Chiron* client consists of an application program (including ADTs) along with a set of artists and several coordinating components. The client runtime architecture is illustrated in figure 3.

Below are brief descriptions of the main components. For a more detailed discussion of the *Chiron* client design see [For93].

Client Initializer The Client Initializer is responsible for bringing up the initial client configuration. This consists of starting up the CPM and invoking artist instances. The Client Initializer is a package that performs client initialization as part of its own elaboration. It will attempt to read a client configuration file at initialization, allowing configurations to be modified without recompilation. There is one Client Initializer per client.

Artist Manager The Artist Manager provides an interface through which new instances of artists can be invoked dynamically. It is used by the Client Initializer to construct an initial configuration, and may also be used by an artist to invoke other artists at runtime. The Artist Manager also includes a procedure to shutdown an executing artist. There is one Artist Manager generated per client.

Application This is the application for which the client is providing a user interface. It makes calls to its ADTs indirectly through an identical interface provided by the Wrapper. It must also *with* the Client Initializer package to allow for automatic initialization of the client components. Otherwise the application remains essentially unchanged. There is only one application per client.

Artist Artists may be used to maintain the graphical depictions of ADTs. They respond to notifications of client and server events. Notifications of changes in ADT state and possibly other events originating within the client are routed from a dispatcher and notifications of server events are routed from the Mapper. Artist manipulate their graphical depictions by making calls ADL calls which are routed through the CPM and subsequently passed to the server. Artists may receive ADT events originating from several ADTs, allowing a single artist to maintain a depiction of multiple ADTs. Artists may also be completely independent of ADTs, receiving only events from the server and possibly hand-coded, non-ADT, events. There may be multiple artists within a client.

Wrapper The Wrapper exports an interface identical to the ADT. It forms a software layer "wrapped" around the ADT which intercepts calls to the ADT, protects its data structures from concurrent access, invokes the requested operation, and notifies the Client Dispatcher of the event. Both the application and the artists call the ADT indirectly through the Wrapper. There is one wrapper for each depicted ADT.

Access Controller The Access Controller is used by the Wrapper to ensure CREW (Concurrent Read Exclusive Write) access to an ADT. The Wrapper obtains a read or write lock from the Access Controller before accessing the ADT. In addition, when a write operation is performed, all subsequent write operations are blocked until each notified artist has completed updating its depiction. There is one Access Controller for each depicted ADT.

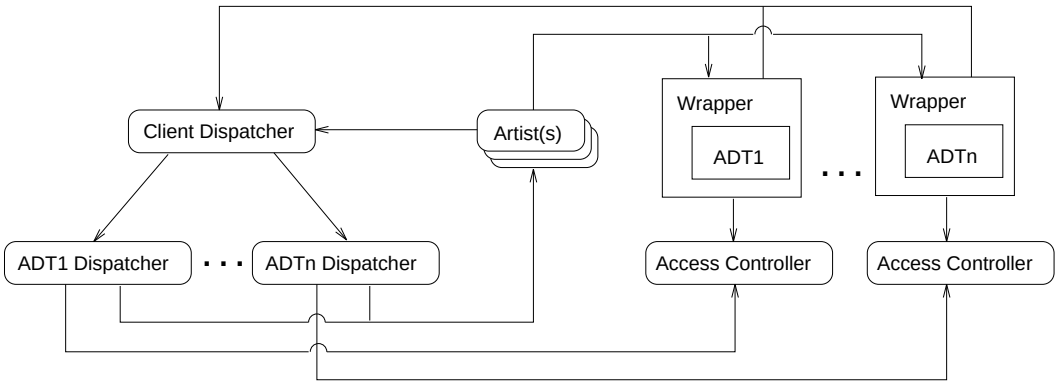

Figure 4: Chiron 1.4 dispatching architecture

Dispatcher The Client Dispatcher routes client events from Wrappers to the appropriate artists. Artists register directly with the Client Dispatcher for notification of specific events and a dispatcher only broadcasts events to artists that have registered for that event. Dynamic registration and de-registration is supported. The default architecture provides one centralized Client Dispatcher per client. However, it is possible to generate dedicated dispatchers for individual ADTs. The Client Dispatcher, will forward all calls pertaining to a particular ADT to its corresponding dispatcher if a dedicated dispatcher exists for that ADT. Figure 4 illustrates a dispatching architecture with a dedicated dispatcher for each depicted ADT. The Client Dispatcher maintains a stable dispatching interface, allowing the dispatching architecture to change with minimum impact on the rest of the client.

Mapper The Mapper routes server events to the appropriate artists. Server events are created when the user interacts with the graphical depiction, i.e. a button push or a menu selection. Based on the object of the event, the Mapper routes the event to the artist that created the object.

CPM The CPM, Client Protocol Manager, handles communication to and from the *Chiron* server.

Many of the components that are unique to each client are generated automatically by the *Chiron* client tool set. These include the Client_Initializer, Artist_Manager, dispatchers, wrappers, access controllers, and artist templates. The CPM and Mapper are included in the *Chiron* client library, and the application and its ADTs are provided by the user.

The only component that is of interest to the user interface builder in the *Chiron* server is the ADL Hierarchy. The ADL Hierarchy describes the graphical objects available when writing artists. The ADL Hierarchy is written in C++ and is implemented using both the XView and Motif Toolkits. It is accessed by an artist via a special interfacing language. See section 4 for a detailed discussion of the hierarchy and appendix E for an alphabetical listing of all ADL classes and their methods.

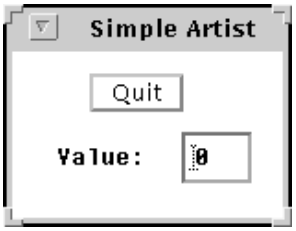

Figure 5: Simple artist example

3 Artist Overview

Artist writing is the key task in constructing a *Chiron* client. Before we go into great detail on this subject, we would like to first familiarize the reader with stucture of an artist, and the basic steps in writing an artist. As an example, we will refer to the implementation of a very simple *Chiron* artist illustrated in figure 5. This artist is composed of a base frame, a panel, a button, and a text field. The *value* field displays a value presumably modified by the executing application and the *quit* button causes the artist disappear.

There are three text files associated with each artist: a specification file, a template (or .cal) file, and a translated file. The artist specification file and the .cal file are generated for each artist. The .cal file serves as a template for the artist body. The artist writer fills in the template with code to create graphical depictions and define how these depictions will respond to certain events. The language used to accomplish this task is called Lo-CAL and it is a superset of the Ada language. The completed template file(s) are processed by the Lo-CAL compiler into legal Ada code. The Lo-CAL language is described in section 6.

3.1 Example artist specification

The specification for our example artist is listed in figure 6. Artists are implemented as task types, allowing multiple instances of artists to be dynamically created and destroyed. All artists share the same interface.

Start_Artist The Start_Artist entry is called only by the Artist_Manager package. This signals the artist to create its graphical objects and register for events.

Notify_Client_Event The Notify_Client_Event entry is called by a dispatcher to notify the artist of any client events for which the artist has pre-registered. Client events generally correspond to operations on ADTs, but they may also include hand-coded events.

Notify_Server_Event The Notify_Server_Event entry is called by the Mapper to notify the artist of server events. Server events are basically windowing system events that have been detected, but not handled, by the server. They include such things such as button and menu selections.

```
with Client_Events;  use Client_Events;
with Chiron_Standard_Library;
with System;

package Simple_Artist is

   package CSL renames Chiron_Standard_Library;

   task type Simple_Artist is

      entry Start_Artist (
            ID             : CSL.Artist_ID_Type;
            Aptr           : SYSTEM.ADDRESS;
            Display_Name   : CSL.Str);

      entry Notify_Client_Event (
            Client_Event   : Client_Events.Client_Event_Ptr;
            Handler_Routine : SYSTEM.ADDRESS);

      entry Notify_Server_Event (
            Object         : CSL.Object_Type;
            Server_Event   : CSL.Chiron_Event_Ptr;
            Handler_Routine : SYSTEM.ADDRESS);

      entry Terminate_Artist;

   end Simple_Artist;

   type Simple_Artist_Ptr is access Simple_Artist;

end Simple_Artist;
```

Figure 6: Simple artist specification

Terminate_Artist The Terminate_Artist entry is called by only by the Artist_Manager. This signals the artist to prepare for termination. This should include destroying graphical objects and unregistering for client events.

All artists must *with* the *Chiron_Standard_Library* (CSL) package. This package defines the Ada types and definitions necessary to interface with the ADL, including the definitions of a variable length string type, Str, and the server event definition. A complete listing of the *Chiron_Standard_Library* package is given in appendix B.

3.2 Example artist template

The artist template for our simple example is given in figure 7. The template is augmented with comment directives to instruct the artist writer on where various code segments should be inserted. A full listing of the completed artist may be found in appendix A.

There are six individual tasks that must be performed when writing an artist:

- Declare graphical objects.
- Declare event handler procedures.
- Register for client events

```
with Wrapper_Simple;
with Client_Dispatcher;

package body Simple_Artist is


   task body Simple_Artist is


      --<< declare artist objects here. >>
      Self_Ptr       : Canvas_Artist_Ptr;
      Local_Artist_ID : CSL.Artist_ID_Type;
      Local_Display   : CSL.Str;

      --<< declare handler routines for both client and server >>
      --<< events plus any auxilliary routines here.          >>
      --
      --server event handlers must have the signature:
      --procedure <handler_name> (Object : CSL.Object_Type;
      --Event  : CSL.Chiron_Event_Ptr);
      --
      --client event handlers must have the signature:
      --procedure <handler_name> (Event : Client_Event_Ptr);


   begin --task body

      accept Start_Artist (
             ID   : CSL.Artist_ID_Type;
             Aptr : SYSTEM.ADDRESS;
             Display_Name : CSL.Str) do
        Self_Ptr := address_to_artist(Aptr);
        Local_Artist_ID := ID;
        Local_Display := Display_Name;

        --<< register interests in client events with the >>
        --<< client_dispatcher here.                      >>

      end Start_Artist;


      --<< create initial graphical objects here. >>

      --<< set behaviors of graphical objects in response >>
      --<< to server events here.                         >>

      --<< call adl start_processing method here. >>

      loop
        select
          accept Notify_Client_Event (
                 Client_Event    : Client_Events.Client_Event_Ptr;
                 Handler_Routine : SYSTEM.ADDRESS);
        or
          accept Notify_Server_Event (
                 Object          : CSL.Object_Type;
                 Server_Event    : CSL.Chiron_Event_Ptr;
                 Handler_Routine : SYSTEM.ADDRESS);
        or
          accept Terminate_Artist;
        end select;
      end loop;

   end Simple_Artist;

end Simple_Artist;
```

Figure 7: Simple artist template

- Create graphical objects.
- Set object behaviors
- Call the *start_processing* method on the baseframe.

These individual tasks are discussed in detail in section 7. For our particular simple example, we will now briefly discuss the code segments that implement the above tasks, and define our simple artist.

Graphical objects declarations

In this section we declare the graphical objbects that make up our artist: a base frame, panel, button and text field. All available graphical objects are defined by the the ADL class hierarchy and described in section 4. The ~ is a directive to the Lo-CAL translator that it needs to do some translations on this type.

The behavior array is used to set the behavior of graphical objects. It will bewill be used below. For now, we just have to declare it.

```
artist_frame     : ~CSL.ADL_base_frame;
artist_panel     : ~CSL.ADL_panel;
quit_button      : ~CSL.ADL_button;
value_field      : ~CSL.ADL_text_fld;

quit_button_behavior : CSL.Behavior_Array_Type :=
                       (others => System.No_Addr);
```

This code is found at line 14 of the completed artist (appendix A).

Event handler procedures

The *Handle_Quit* procedure is a callback procedure for a server event. It will automatically get called when the quit button is selected (we set that up below). The ~*Apply* call is used to call the *set_show* method on our base frame that will cause our artist to disappear. Again, the ~ signals that this is outside of the scope of the language, so the Lo-CAL translator will translate this into low-level calls to the server.

```
procedure Handle_Quit (Object : CSL.Object_Type;
                       Event  : CSL.Chiron_Event_Ptr) is
begin
   ~Apply(ADL_base_frame,
          artist_frame,
          set_show,
          false);
end Handle_Quit;
```

This code is found at line 32 of the completed artist (appendix A).

The *Handle_Update* procedure is a callback procedure for a client event. It will automatically be called whenever the artist is notified that the ADT value has been updated. The *Event* parameter contains the new value, which we use to reset our text field value using the ADL *set_value* method on our value_field object.

```
procedure Handle_Update (Event : Client_Event_Ptr) is
begin
  --update depiction
  ~Apply(ADL_text_fld,
         value_field,
         set_value,
         to_str(integer'image (Event.Simple_Update_Value)));
end Handle_Update;
```

This code is found at line 42 of the completed artist (appendix A).

Registering for client events

Now we must inform the client dispatcher of which events the artist is interested in knowing about. In this case, the artist must be notified know when the ADT value changes. We also give it the address of the *Handle_Update* callback procedure, which will be invoked automatically whenever the artist is notified of this client event.

```
Client_Dispatcher.Dispatcher.Register_Event
  (Local_Artist_ID, Client_Events.Simple_Update_Value,
   Handle_Update'ADDRESS);
```

This code is found at line 65 of the completed artist (appendix A).

Creating graphical objects

In this section instances of the previously declared graphical objects must be created. This is accomplished by invoking the ADL create methods for each object.

```
artist_frame := ~Apply(ADL_base_frame,
                  create,
                  frame_label   => "Simple Artist",
                  x             => 200,
                  y             => 200,
                  width          => 200,
                  height         => 200,
                  foreground     => BLACK,
                  background      => WHITE);

artist_panel := ~Apply(ADL_panel,
                  create,
                  parent         => artist_frame,
                  width          => 200,
                  height         => 200);

quit_button := ~Apply(ADL_button,
                 create,
                 parent          => artist_panel,
                 button_id        => 1,
                 label           => "Quit",
                 auto_placement  => false,
                 x               => 40,
                 y               => 10);
```

```
value_field := ~Apply(ADL_text_fld,
                      create,
                      parent         => artist_panel,
                      label          => "Value: ",
                      text_value     => "0",
                      display_length => 3,
                      stored_length  => 3,
                      read_only      => true,
                      auto_placement => false,
                      x              => 20,
                      y              => 40);
```

This code is found at line 73 of the completed artist (appendix A).

We also invoke the *ADL_Window_Fit* method on our base frame and panel. This resizes the panel to fit nicely around its panel objects and then resizes the base frame to fit nicely around the resized panel.

```
~Apply(ADL_panel,
       artist_panel,
       ADL_window_fit);

~Apply(ADL_base_frame,
       artist_frame,
       ADL_window_fit);
```

This code is found at line 112 of the completed artist (appendix A).

Setting object behaviors

Setting object behaviors means specifying how the artist should respond to server events on particular objects. This is done by setting callback routines for object/event pairs. Since some objects can receive different kinds of events, we set these behaviors in an array (declared above), then associate that array with the object. Now if the server informs our artist of a Select_Event on the quit button, the *Handle_Quit* procedure will be invoked automatically.

```
quit_button_behavior(Select_Event) := Handle_Quit'ADDRESS;

~Set_Behavior(quit_button,
              quit_button_behavior);
```

This code is found at line 123 of the completed artist (appendix A).

Calling start_processing method

Finally, we have to call the *start_processing* method on our base frame. This tells the server to display our artist and to start receiving server events.

```
~Apply(ADL_base_frame,
       artist_frame,
       start_processing);
```

This code is found at line 131 of the completed artist (appendix A).

4 The ADL Hierarchy

The ADL Hierarchy defines the set of graphical objects available to artist writers. Figure 8 shows the hierarchy classes and their inheritance relationships. The hierarchy supports creation of buttons, menus, windows, icons, scrollbars, and many other graphical objects. Those classes appearing in bold text cannot be created. For example, t create an instance of the class **ADL_drawable** cannot be created. Such classes exist for hierarchical purposes as they define methods common to all sub-classes. Separate from the hierarchy are two auxiliary classes, **Object_List** and **Corner_List**, which are used only to help define other objects within the hierarchy.

The hierarchy is written in C++ and is accessible to artists written in the Lo-CAL language (see section 6). The ADL Hierarchy was originally implemented using the XView Toolkit which provides the OPEN LOOK look-and-feel. Since then, ADL Hierarchy has been reimplemented on top of the Motif Toolkit which provides the Motif look-and-feel. Chiron can support both the Motif look-and-feel as well as OPEN LOOK. The same *Chiron* client can be executed under either look-and-feel by simply resetting an environment variable (see section 8). Due to the nature of the toolkits, there are certain areas where complete duplication of functionality was infeasable. These areas are indicated as such in the following sections. Figure 9 illustrates two executions of a 16-tile puzzle client running under each look-and-feel.

To determine all the methods available to an instance of a given class, follow the parent-class chain all the way up to root class, **ADL_object**. For example, to determine all the methods available to an instance of a base frame, look at the description for class **ADL_base_frame**, **ADL_frame**, **ADL_window**, **ADL_drawable**, and **ADL_object**. Collectively, these classes provide the set of methods for a base frame instance.

The following sections describe the graphical objects provided by the hierarchy. To determine all the available class methods and their signatures, see the *ADL Reference Manual* listed in appendix E.

4.1 Frames

A frame is a container for other windows. It manages the geometry and placement of subwindows that do not overlap (they are tiled) and that are fixed within the boundary of the frame. Subwindows include canvases, text subwindows, tty windows, and panels. These subwindows cannot exist without a parent frame to manage them. A frame depends upon the window manager for its decorations and many basic operations.

There are two types of frames supported by the ADL Hierarchy: base frames and popup frames. A base frame is the main frame of an application. It exists throughout the execution of an application and can be iconified. Popup frames are frames that can come and go throughout the execution of an application. Examples of base frames and popup frames under both XView and Motif are pictured in figure 10.

A font and cursor can be specified for a given frame. They need to be created as instances of the **ADL_font** and **ADL_cursor** classes, respectively. The font specified will be the new default font of the frame and its subwindows. It should be iterated that some of

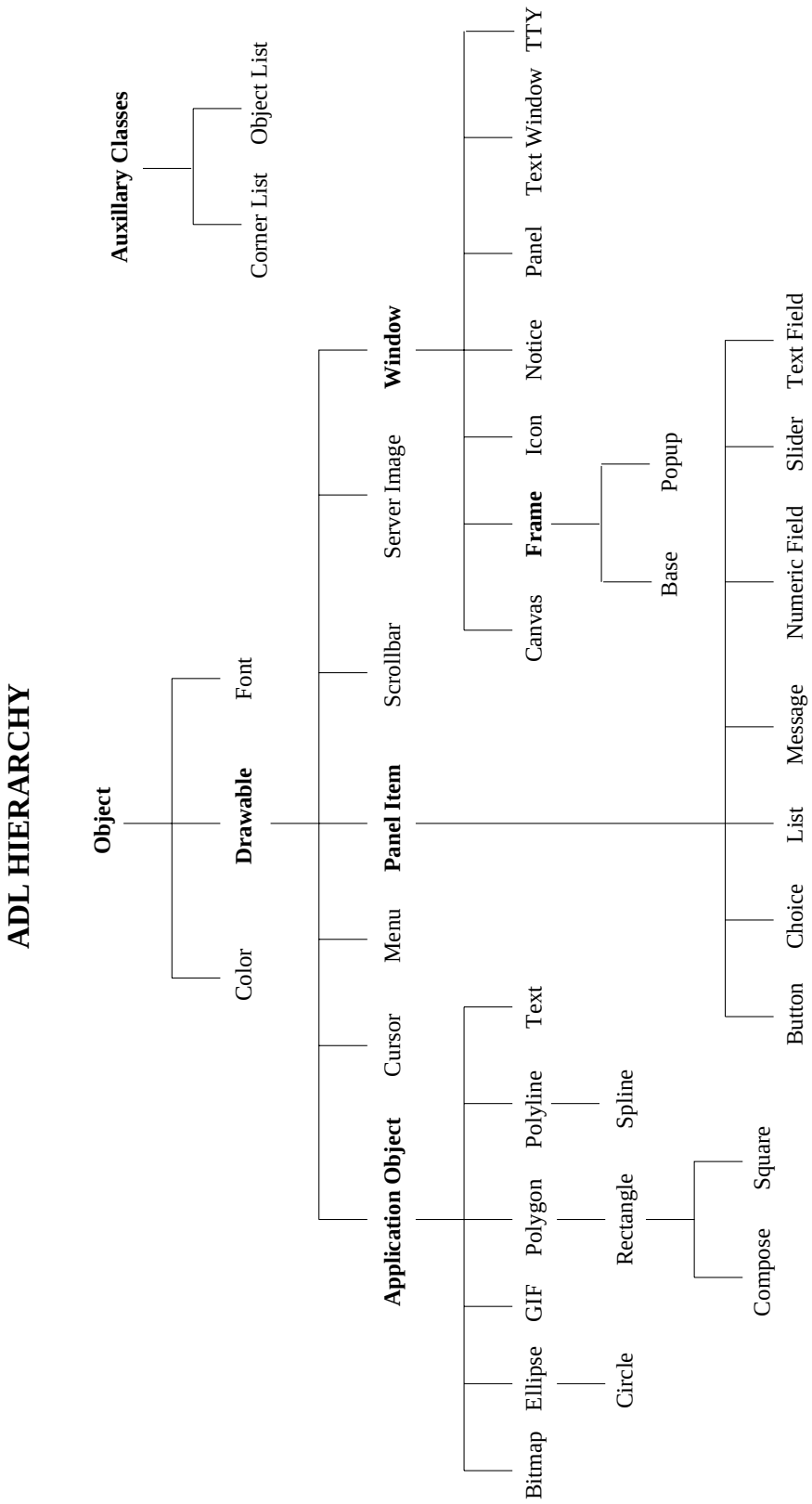

Figure 8: The ADL Hierarchy

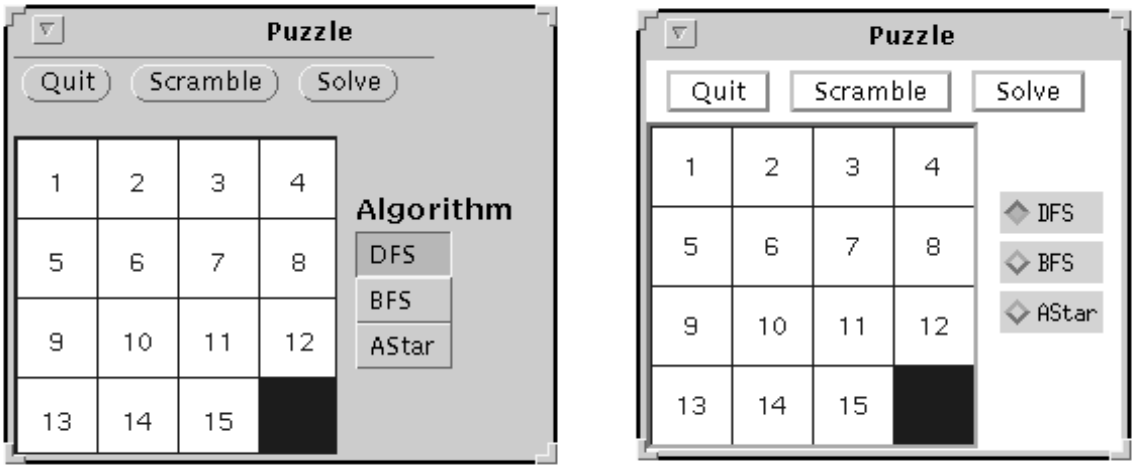

Figure 9: *Chiron* clients can execute under either an OPEN LOOK or a Motif look-and-feel. Here we have two executions of a 16-tile puzzle. The left puzzle is executing under the XView look-and-feel, while the right puzzle is executing under the Motif look-and-feel.

the text within the frame is unalterable (such as the title within the title bar). The cursor specifies the shape of the cursor when inside the frame. Note that due to the manner in which X11/Xt maps window ids, cursors created using the Motif server must be set using ADL_window's set_cursor method. The call to set_cursor must occur after the call to *start_processing* (see section 4.1.1).

In XView, currently cursors cannot be set for frames – only for panels and canvases. Motif allows a cursor to be associated with any window.

A frame has *width* and *height* attributes that can be set during frame creation. Oftentimes, however, an application might want the size of the frame to be just large enough to house the subwindows within it. *Chiron-1* provides a method which does just this. After creating the frame and all the frame's subwindows, applying the ADL_window_fit() method resizes the frame to fit its contents. There are variations on this method: ADL_window_fit_height() adjusts the vertical size of the frame to fit its contents and ADL_window_fit_width() adjusts the horizontal size of the frame to fit its contents. ADL_window_fit should be called from the innermost panel or frame first to assure proper sizing. See figure 11.

The class **ADL_frame** defines methods common to both base frames and popup frames.

4.1.1 Base Frames

Base frames are provided with titles and (optional) footers that display text. Footers, however, are only available under XView and are not supported under Motif. The title typically displays information such as the application name; the text is centered and its

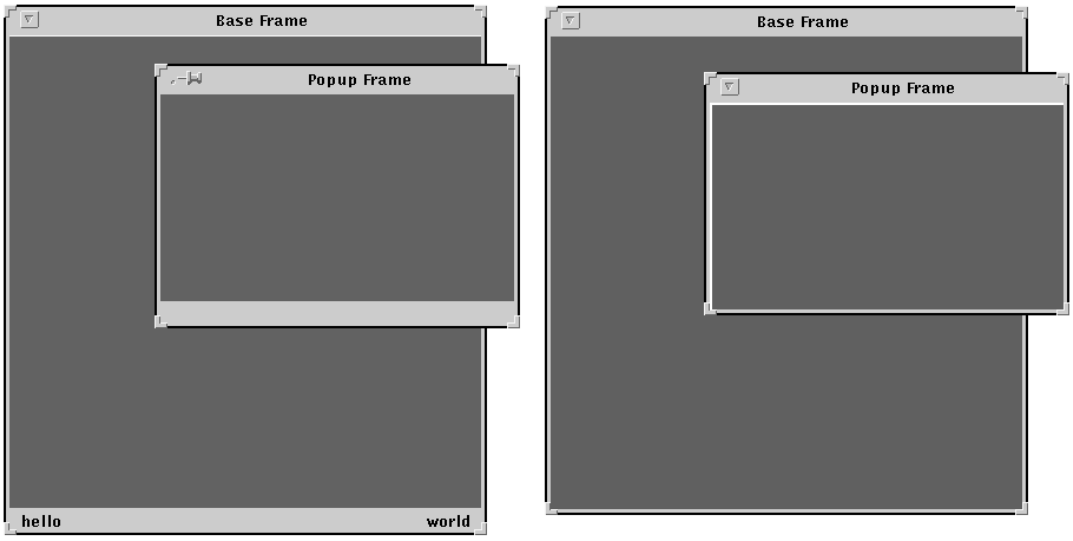

Figure 10: Examples base frame and popup frame under XView (left) and Motif (right). Close buttons are located in the upper left corner for base frames and Motif popup frames. XView popup frames provide push pins. XView supports arbitrary text display, called footers, in the lower left and right borders of a frame. Footers are not supported under Motif.

---

font is not alterable by the application. The title also has a Close button at the upper left corner; selecting the button causes the frame to iconify – that is, the frame turns into a graphical image, or icon, and is displayed (probably) elsewhere on the screen.

When the base frame is closed, an icon replaces the entire base frame, including all subwindows and all dependent popup frames. By default, no icon is associated with a base frame and the frame iconifies to a blank square somewhere on the screen. However, an icon can be defined for the base frame by creating an instance of the **ADL_icon** class. Once an icon is created, it can be attached to the frame using the attach_icon() method.

The footer of a base frame might display text such as error messages, a page number, the date, or other miscellaneous information. The footer is split into two parts: the left footer where text is left justified, and the right footer where the text is right justified. The visibility of the footer portion of the base frame can be toggled. **Note that footers are not available under Motif.**

Foreground and background colors can be specified for a base frame. If the application is executed on a monochrome machine, these colors will be mapped to black and white. The foreground and background color will be inherited by the objects within a frame, unless specifically overridden. For example, if the base frame is created with a foreground color of blue and a background color of red, then the base frame's panel will have those same colors unless the panel is specifically created with different colors.

```
--Create frame.

--Create panel.

--Create panel objects.

Apply(ADL_panel,
      panel,
      ADL_window_fit);

Apply(ADL_base_frame,
      frame,
      ADL_window_fit);
```

Figure 11: Resizing frames

The class **ADL_base_frame** defines the methods particular to a base frame instance, including creation and destruction. It should be noted that destroying a base frame destroys all objects within the base frame, essentially killing the application.

Note that a base frame will not appear until the start_processing() method is called. This method signals to *Chiron* that all initial graphical objects have been created and that the artist is ready to process events. If you would like the base frame to appear without triggering event processing, and you are using XView, then simply call the set_show() method on the base frame object. set_show() may not be invoked before start_processing() if using Motif. set_show() is defined in class ADL_drawable.

4.1.2 Popup Frames

Popup frames are defined as children of base frames. Usually, popup frames serve one function and then go away. Instead of having a Close button in the frame's title bar, the popup frame has a pushpin The pushpin governs whether the frame remains up after the user performs the functions that the popup frame provides. **Note that Motif does not support the notion of a pushpin.**

Popup frames are not automatically displayed when created. An application might create many popup dialogue boxes initially and then determine the appropriate time to actually display them. To make a popup frame visible, apply the set_show() method (defined in class **ADL_drawable**).

Popup frames can have footer areas (just like base frames). The footer of a popup frame might display text such as error messages, a page number, the date, or other miscellaneous information. The footer is split into two parts: the left footer where text is left justified, and the right footer where the text is right justified. The visibility of the footer portion of the popup frame can be toggled. **Note that footers are not available under Motif.**

Foreground and background colors can be specified for a popup frame. If the application is executed on a monochrome machine, these colors will be mapped to black and white. The foreground and background color will be inherited by the objects within the frame,

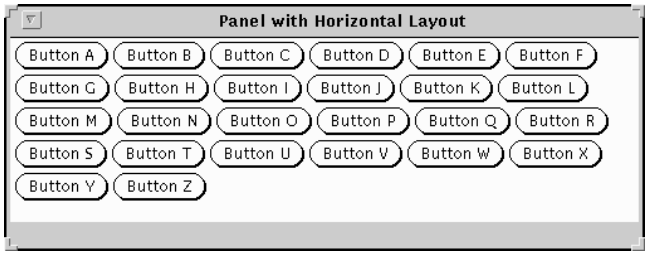

Figure 12: The panel above has set its layout scheme to *horizontal*, and all the buttons have been created with their *auto_placement* attributes set to true, thereby utilizing the layout scheme of the panel.

unless specifically overridden. For example, if the popup frame is created with a foreground color of blue and a background color of red, then the popup frame's panel will have those same colors unless the panel is specifically created with different colors.

The initial location of a popup frame is specified relative to that of its parent base frame. Popup frames can be dragged to new locations such that the next time the popup frame is displayed, it will appear in the new location.

The class **ADL_popup_frame** defines the methods particular to a popup frame instance, including creation and destruction.

4.2 Panels

Panels define control areas within a graphical application. Buttons, text and numeric input fields, lists, sliders, messages, and choice items all reside within a panel, and are sub-classed from the class **ADL_panel_item**.

A panel defines a default layout scheme for the placement of its panel items. This layout scheme is dependent upon a few attributes set when the panel itself is created. The *layout* attribute specifies whether the panel items are placed in row order (horizontal layout) or column order (vertical layout). For example, if two buttons are created within a panel, and the panel has specified a horizontal layout, then the buttons will appear side by side. If the layout is specified as vertical, then the second button will appear below the first.

There are two other attributes controlling the layout scheme: *x_gap* and *y_gap*. x_gap specifies the default distance between items in the horizontal direction (in pixels). y_gap specifies the default distance between items in the vertical direction (in pixels). Figure 12 shows the use of the default layout scheme for a XView panel with horizontal layout.

A panel is created as a child of a frame (either a base frame or a popup frame), and therefore is a subwindow of that frame. The location of the panel subwindow within the frame needs to be specified. This can be done explicitly, via x and y attributes which specify the location of the panel (in pixels) relative to the upper left corner of the parent frame. Or the panel location can be specified relative to other existing subwindows within the frame.

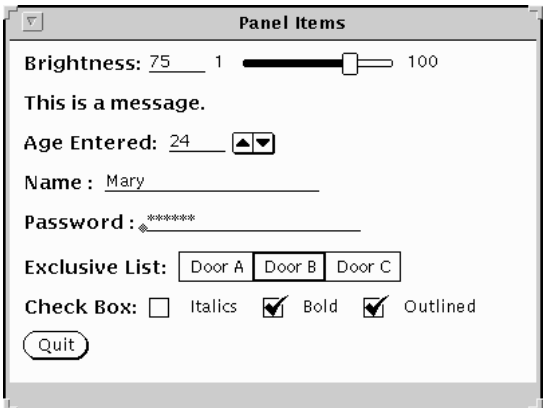

Figure 13: The above panel, created with the XView *Chiron* server, contains instances of the following panel items: sliders, messages, numeric fields, text fields, choice items, and buttons.

For example, it can be specified that the panel be placed below an existing subwindow and/or to the right of another existing subwindow. And finally, the panel location can assume its default location by setting the *auto_placement* attribute to true.

A panel has *width* and *height* attributes that can be set during panel creation. Oftentimes, however, an application might want the size of the panel to be just large enough to house the panel items within it. *Chiron* provides a method which does just this. After creating the panel and all the panel items within it, applying the ADL_window_fit() method resizes the panel to fit its contents. There are variations on this method: ADL_window_fit_height() adjusts the vertical size of the panel to fit the panel's contents and ADL_window_fit_width() adjusts the horizontal size of the panel to fit the panel's contents.

The default color for XView panels is grey. Panel colors may be changed, but panel items will remain grey. i.e. They will not inherit the different color. Therefore panels, using the XView based Chiron server, look best when the default grey is used.

The class **ADL_panel** defines the methods particular to a panel instance, including creation and destruction.

4.3 Panel Items

Panel items are objects like buttons, sliders, and choices which can only exist within a panel. A panel with the various panel items is shown in figure 13.

The class **ADL_panel_item** defines methods common to buttons, messages, sliders, text fields, numeric fields, and choice items.

4.3.1 Buttons

A button item allows the user to invoke a command or bring up a menu. Menu buttons differ in their creation and usage and are therefore defered to the discussion of menus in

section 4.9. The remainder of this section refers only to non-menu buttons.

Buttons can be textual or graphical. A textual button has a text label which might be the name of the command or menu invoked or displayed when the button is pressed. The font of a textual button cannot be changed or emboldened. A graphical button contains an image; the image typically describes the action invoked when the button is pressed. The image for a graphical button is of the **ADL_server_image** class. **The Motif version of the *Chiron* server does not support color for graphical buttons.**

The size of the button is automatically determined by the text or graphical image specified when the button is created. The button can be resized after creation using the set_width() and set_height() methods found in class **ADL_drawable**.

A button is a child of a panel. The placement of the button within the panel can be specified explicitly, via x and y attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the button to be placed according to the layout scheme of the panel.

Buttons can be defined to be active or inactive. An inactive button ignores button presses by the end user. Additionally, the visibility of a button can be toggled.

All buttons are given unique IDs upon creation. This allows the artist to identify particular buttons during button event notification.

Buttons generate a *select* event when the button is pressed (by clicking on it with the left mouse button). See section 5 for more information on events and object behavior.

The class **ADL_button** defines the methods particular to a button instance, including creation and destruction.

4.3.2 Messages

Messages display either text or a graphical image within a panel. Messages can be used to provide the end user with additional information, for instance, to characterize a group of buttons. Textual messages can appear plain or emboldened. They can be made visible or invisible (see the **ADL_drawable** class for the description of the set_show() method).

In order to create a graphical message, one must first create the image as an instance of the **ADL_server_image** class.

A message is a child of a panel. The placement of the message within the panel can be specified explicitly, via x and y attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the message to be placed according to the layout scheme of the panel.

The class **ADL_message** defines the methods particular to a message instance, including creation and destruction.

4.3.3 Lists

Lists provide the artist writer with a simple and convenient method of creating lists of items, allowing the user to select from the group of choices. Pressing the left mouse button on an item selects it and deselects any other item. Dragging the left button moves the selection as the pointer is moved. Releasing the left button over an item moves the location

cursor to that item. Moving the location cursor has the side effect of generating Chiron server event of type Select.

The Str_Val field of the Chiron server event contains the name of the item selected. The Num_Val field contains the numeric position of the item selected. For more information see section 5.

The class **ADL_list** defines the methods particular to a list instance.

4.3.4 Sliders

Slider items allow the graphical representation and selection of a value within a range. Sliders are appropriate for situations where it is desired to make fine adjustments over a continuous range of values. The user selects the slider bar and drags it to the value desired. A slider has four displayable components: the label, the current value, the slider bar, and the minimum and maximum allowable integral values (the range).

The label is the text preceding the actual slider bar. The text can be set to an empty string if no preceding text is desired.

Sliders may be horizontal or vertical. Horizontal sliders specify that the label and the actual slider bar appear side by side. Vertical sliders specify that the actual slider bar appears below the corresponding label.

A slider is a child of a panel. The placement of the slider within the panel can be specified explicitly, via x and y attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the slider to be placed according to the layout scheme of the panel. Since a slider has two parts: the textual label and the graphical slider bar, it should be noted that the placement of the slider object refers to the placement of the label part. The slider bar will be placed next to, or below, the label part depending upon the *layout* value.

A slider object has a numeric value. This can be initialized upon creation, altered programmatically via the set_value() method, and, of course, altered by the end user through graphical means. The value of the slider can be queried at any time using the get_value() method.

Sliders can generate *select* events when the slider bar has been dragged (using the left mouse button). See section 5 for more information on events and object behavior.

The class **ADL_slider** defines the methods particular to a slider instance, including creation and destruction.

4.3.5 Text Items

Text items describe input areas accepting textual input from the end user. A carat is used to indicate the insertion point where new text is added. You can type in more text than fits on the text field; it will cause the field to scroll to the left.

As with sliders, text fields have two parts: a label part and a field part. The label part might describe the kind of entry value expected, such as name or password. The field part is where the information is entered in by the end user.

Text fields can be layed out vertically or horizontally. A vertical layout specifies that the field will appear below the label. A horizontal layout specifies that the field will appear to the right of the label.

A text field object has a string value. This can be initialized upon creation, altered programmatically via the set_value() method, and, of course, altered by the end user through graphical means. The value of the text field can be queried at any time using the get_value() method.

The editability of a text field can be controlled via the set_read_only() method. Read only text fields do not respond to keyboard input.

It is sometimes desirable to have a protected field where the end user can enter confidential information. The *mask_char* attribute is provided for this purpose. When the user enters a character, the character specified as the value of *mask_char* will be displayed in place of the character the user has typed. If you want to disable character echo entirely so that the carat does not advance and it is impossible to tell how many characters have been entered, use the space character as the mask. If *mask_char* is set to NULL (the default) then the characters will appear as typed. It is important to note that the value of the text field is still the string that the user types, regardless of what is displayed. **The mask_char attribute not is not supported in the Motif server.**

Other attributes such as the number of textual characters displayed as well as the maximum number of textual characters accepted by the input field (independently of how many are displayable) can be controlled. **Note that text items may not exceed 100 characters.**

A text field is a child of a panel. The placement of the text field within the panel can be specified explicitly, via *x* and *y* attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the text field to be placed according to the layout scheme of the panel. Since a text field has two parts: the textual label and the input field, it should be noted that the placement of the text field object refers to the placement of the label part. The input part will be placed next to, or below, the label part depending upon the *layout* value.

Text fields generate *select* events when the keyboard return key has been pressed while the text field object has the keyboard focus. As other keys are pressed, the text field display is updated, but the application won't be notified until the return key has been pressed. See section 5 for more information on events and object behavior.

The class **ADL_text_fld** defines the methods particular to a text field instance, including creation and destruction.

4.3.6 Numeric Items

Numeric items describe input areas accepting numeric input from the end user. Numeric items include buttons, labeled with arrows, to increment or decrement the value displayed in the field (see figure 13).

As with text fields, numeric fields have two parts: a label part and a field part. The label part describes the kind of entry value expected, such as age or weight. The field part is where the information is entered in by the end user.

Numeric fields can be layed out vertically or horizontally. A vertical layout specifies that the field will appear below the label. A horizontal layout specifies that the field will appear to the right of the label.

A numeric field object has a numeric value. This value can be initialized upon creation, altered programmatically via the set_value() method, and, of course, altered by the end user through graphical means. The value of the numeric field can be queried at any time using the get_value() method.

The editability of a numeric field can be controlled via the set_read_only() method. Read only numeric fields do not respond to keyboard input.

Other attributes such as the number of characters displayed as well as the maximum number of characters accepted by the input field (independently of how many are displayable) can be controlled.

A numeric field is a child of a panel. The placement of the numeric field within the panel can be specified explicitly, via x and y attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the numeric field to be placed according to the layout scheme of the panel. Since a numeric field has two parts: the textual label and the input field, it should be noted that the placement of the field object refers to the placement of the label part. The input part will be placed next to, or below, the label part depending upon the *layout* value.

Numeric fields generate *select* events when the keyboard return key has been pressed while the numeric field has the keyboard focus. As the value is incremented and decremented via the numeric field buttons, the numeric field display is updated, but the application won't be notified until the return key has been pressed. See section 5 for more information on events and object behavior.

The class **ADL_num_fld** defines the methods particular to a numeric field instance, including creation and destruction.

4.3.7 Choice Items

Choice items provide a list of different choices to the end user in which one or more choices may be selected. A choice item has a label, a list of choices, and an indication of which choice or set of choices is currently selected.

Choice items can be defined as exclusive or non-exclusive. Exclusive items specify that only one choice may be set at a given time. Non-exclusive choice items indicate that multiple choices may be set at a given time.

With the OPEN LOOK look and feel, choice items may appear in a standard box-per-choice fashion, or in check box fashion (see figure 13). Note that if the check box presentation is desired, the choice item is automatically defined to be non-exclusive. The box-per-choice representation can be either exclusive or non-exclusive. In the Motif version, exclusive choices always appear as diamonds and non-exclusive choices as squares. Figure 9 illustrates exclusive choices for both XView and Motif.

A choice item may appear horizontally or vertically. A horizontal layout specifies that the label is followed by the various choices, all side by side. A vertical layout specifies that

the label is above the first choice, which appears above the second choice, etc. In other words, the choice item appears in a column.

A choice item is built up incrementally. First you create a choice item and then you add choices via the add_choice_string() method. There is no physical limit to the number of choices appearing in a choice item.

A choice item has a numeric value which can be initialized upon creation, altered programmatically via the set_value() method, and, of course, altered by the end user through graphical means. The value of the choice item can be queried at any time using the get_value() method. The semantics of this numeric value is dependent upon the type of the choice item: exclusive or non-exclusive. For exclusive choice items, the value of the item is given as an ordinal value. So if the first choice is chosen, the value of the choice item will be 0; if the second choice is chosen, the value of the choice item will be 1, and so on. For non-exclusive choice items, the value of the item is given as the binary value of a mask indicating those choices that are selected. That is, if there are three choices, and they are all selected, the mask looks like 111, so the value is 7. If only the first and third choices are selected, then the mask looks like 101, so the value of the choice item is 5. If only the third choice is selected, then the mask looks like 100, so the value is 4.

A choice item is a child of a panel. The placement of the item within the panel can be specified explicitly, via x and y attributes which specify the pixel distance from the upper left corner of the panel, or implicitly via the *auto_placement* attribute. The *auto_placement* attribute allows the choice item to be placed according to the layout scheme of the panel. Since a choice item has two parts: the textual label and the choice list part, it should be noted that the placement of the choice item refers to the placement of the label part. The choice list part will be placed next to, or below, the label part depending upon the *layout* value.

Choice items generate *select* events when a choice selection is made via the left mouse button. See section 5 for more information on events and object behavior.

The class **ADL_choice** defines the methods particular to a choice item instance, including creation and destruction.

4.4 Canvases

Canvases define drawing areas within an application. Bitmaps, circles, squares, etc., can be defined here, and are collectively referred to as *application objects*, as all are subclassed from **ADL_application**.

Canvases are scrollable and thus are defined with a virtual width and height as well as a physical width and height. Typically, the virtual dimensions are larger than the physical ones, although this is not required. The virtual dimensions are referred to as the paint width and height. The physical dimensions are referred to as the view width and height.

Foreground and background colors can be specified for a canvas. If the application is executed on a monochrome machine, these colors will be mapped to black and white. The foreground and background color will be inherited by the objects within the canvas (the application objects), unless specifically overridden.

A font and cursor can be specified for a given canvas. They need to be created as

instances of the **ADL_font** and **ADL_cursor** classes, respectively. The cursor specifies the shape of the cursor when inside the canvas.

Canvases can have menus associated with them. If the right mouse button is pressed within the canvas, the menu for the canvas will be displayed (if one exists). To associate a menu with a canvas, the menu itself needs to exist. The **ADL_menu** class, defines how to create menu instances. Applying the set_canvas_menu() method associates the menu with the canvas.

Canvases can have vertical and/or horizontal scrollbars. The Motif version of the Chiron server will display the scrollbars on an as-needed basis. If the either of the paint dimensions are larger then the view dimensions, then the appropriate scrollbar will be created and displayed automatically. There is no need to explicitly create and set the scrollbars for the Motif server. For the XView server it is necessary to create and attach the scrollbars to the canvas explicitly. The **ADL_scrollbar** class, defines how to create scrollbar instances. Applying the set_vertical_scrollbar() and set_horizontal_scrollbar() methods attaches scrollbars to the canvas.

A canvas is created as a child of a frame (either a base frame or a popup frame), and therefore is a subwindow of a frame. The location of the canvas subwindow within the frame needs to be specified. This can be done explicitly, via x and y attributes which specify the location of the canvas (in pixels) relative to the upper left corner of the parent frame. Or the canvas location can be specified relative to other existing subwindows within the frame. For example, it can be specified that the canvas be placed below an existing subwindow and/or to the right of another existing subwindow. And finally, the canvas location can assume its default location by setting the *auto_placement* attribute to true.

Canvases generate *adjust* events when the middle mouse button is pressed while over the canvas, and *key* events when keyboard keys are pressed while the mouse is over the canvas. When the right mouse button is pressed while over the canvas, a menu is displayed; the end user can make a menu choice and thus generate a *menu* event. Note that the canvas will only be the focus of events if the mouse is over the canvas, but not over any of the application objects within the canvas; otherwise, the application object will be the target of the event. See section 5 for more information on events and object behavior.

The class **ADL_canvas** defines the methods particular to a canvas instance, including creation and destruction.

4.5 Application Objects

Application objects are graphical objects that can appear within a canvas. Figure 14 shows a scrollable canvas with the various application objects currently supported. The **ADL_application** class defines attributes and methods common to all graphical objects, such as moving objects relative to one another from within the canvas.

Graphical objects within a canvas can overlap one another (in contrast, subwindows like panels and canvases must appear tiled). Since two or more objects can overlap, it is necessary to specify the *z-axis ordering* to determine which should appear to be the top-most item, the second top-most item, etc., down to the bottom-most item. This can be specified at object creation time: when a graphical object is created (for example, a rectangle),

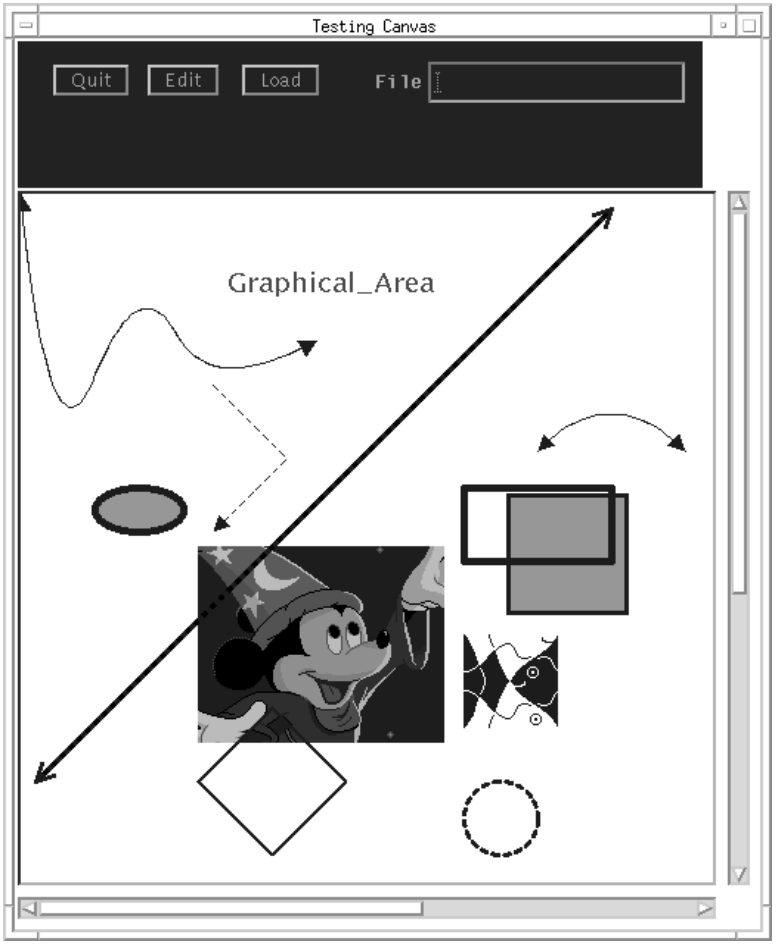

Figure 14: The above canvas contains (at least) one instance of the eleven application objects provided: polylines, polygons, rectangles, squares, ellipses, circles, bitmaps, and splines, gifs, text. The square was defined with a green background and therefore hides the part of the rectangle it overlaps. If the background was defined to be clear, the rectangle would show through.

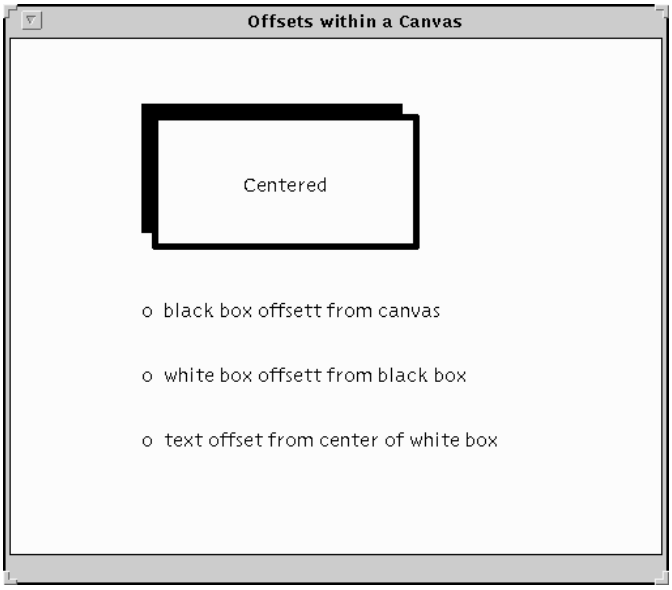

Figure 15: Application objects can be relative to the canvas or to each other. The black rectangle is relative to the canvas, the white rectangle is relative to the black rectangle, and the text is relative to the center of the white rectangle.

---

there are two attributes that can be specified, *above* and *below*. These attributes determine where the object being created should appear in relation to other existing objects within the canvas. So it can be specified that a rectangle should appear above an existing circle and polyline, and below an existing ellipse. Then as the rectangle is dragged about, its z-axis ordering will remain intact; if it is dragged such that it overlaps the ellipse, the rectangle will appear beneath the ellipse, and so on. If an object is created without specifying *above* and *below* attributes, the object will appear to be the top-most item (created thus far). The z-axis ordering can be modified dynamically by raise() and lower() methods defined in the **ADL_application** class.

Each application object, whether it be a polyline, text, or bitmap, has a virtual bounding box which can be found by drawing a box to encompass the object. When we talk about the position of an object, we are talking about the location of the upper left hand corner of the object's bounding box.

Application objects can have menus associated with them. Pressing the right mouse button over a graphical object with an associated menu causes the menu to be displayed. To associate a menu with a graphical object, use the set_application_menu() method.

4.5.1 Polygons

Polygons define a closed graphical object made up of a set of points. A polygon object can be drawn with a specified thickness, and the outline of the figure can be solid, dashed, or double-dashed. It can be drawn hollow or filled (specifying clear as the background color causes the object to appear hollow; any other color draws the object filled).

A polygon is a child of the canvas containing the object. The *corner_list* attribute specifies the shape of the polygon; see class **Corner_List** to define the set of points that make up the shape.

The placement of the polygon can be relative to the upper left corner of the canvas, or relative to another existing object within the canvas. To position the polygon relative to an existing object, specify that object by setting the *object* attribute. If you want the center of the polygon relative to the center of *object*, then set the *center* attribute to true; otherwise the upper left corner of the polygon's bounding box will be relative to the upper left corner of the existing object. The x and y attributes specify the offset of the polygon from the upper left corner of the canvas, or from the upper left corner of the existing object, or from the center of the existing object (depending upon the values of *object* and *center*). Figure 15 illustrates the use of the object positioning features.

The *line_color* and *fill_color* color of the polygon can be specified. The thickness of the outline of the polygon can also be specified by giving a positive integer value – the larger the number the thicker the line. *Chiron-1* provides three options for setting the style of the object's outline: solid, dashed, or double-dashed. The visibility of the polygon can be toggled as well as the selectability of the object. If a polygon is not selectable, then it cannot be the target for events.

Polygons can have menus associated with them. This means that pressing the right mouse button over the object causes the associated menu to be displayed. To associate a menu with a polygon, the menu itself needs to exist. The **ADL_menu** class defines how to create menu instances. Applying the set_application_menu() method associates the menu with the polygon.

Polygons can generate *select* events when the left mouse button is pressed over the object, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed over the object, *key* events when keyboard keys are pressed while over the object, and *move* events when the left mouse button is pressed while over the object, and then dragged to a new location. See section 5 for more information on events and object behavior.

The class **ADL_polygon** defines the methods particular to a polygon instance, including creation and destruction.

4.5.2 Rectangles

Rectangles are defined as specialized polygons. The functionality of rectangles is identical to that of polygons, only their creation is different. Rather than specifying a *corner_list* of points that make up the shape of the object, you only need to specify the size of the x and y axis. See the description of class **ADL_polygon** for information on other attributes and methods available to an instance of class **ADL_rectangle**.

The class **ADL_rectangle** defines the methods particular to a rectangle instance, in-

cluding creation and destruction.

4.5.3 Squares

Squares are defined as specialized rectangles (which are specialized polygons). The functionality of squares is identical to that of polygons and rectangles; only some of the creation attributes are different. Rather than specifying a *corner_list* of points that make up the shape of the object, you only need to specify the length of a side. See the description of class **ADL_polygon** for information on other attributes and methods available to an instance of class **ADL_square**.

The class **ADL_square** defines the methods particular to a square instance, including creation and destruction.

4.5.4 Compose

Compose objects are defined as specialized rectangles. They consist of a rectangle with a text label centered inside. The rectangle and text are treated as a single application object. Any events on the rectangle or text are returned on the compose object, and if the object is moved the label remains centered within the rectangle.

The class **ADL_compose** defines the methods particular to a compose instance, including creation and destruction.

4.5.5 Polylines

Polylines are graphical lines that are defined by two or more points. The line can be drawn with a specified thickness; the line can be solid, dashed or double-dashed; and the line may have arrowheads that are hollow or filled, appearing at either, both, or none of the two endpoints of the line.

A polyline is a child of the canvas containing the line. The *corner_list* attribute specifies the shape of the polyline. This is the list of points that define the polyline. By default, the first point given is the start point and the last point given is the end point of the polyline. See the specification of the **Corner_List** class for information on specifying the points of the polyline.

The placement of the polyline can be relative to the upper left corner of the canvas, or relative to another existing object within the canvas. To position the polyline relative to an existing object, specify that object by setting the *object* attribute. If you want the center of the polyline relative to the center of *object*, then set the *center* attribute to true; otherwise the upper left corner of the polyline's bounding box will be relative to the upper left corner of the existing object. The x and y attributes specify the offset of the polyline from the upper left corner of the canvas, or from the upper left corner of the existing object, or from the center of the existing object (depending upon the values of *object* and *center*).

The foreground color of the polyline can be specified. The default is the foreground color of the canvas itself. The thickness of the line can also be specified by giving a positive integer value – the larger the number the thicker the line. *Chiron-1* provides three options for setting the style of the line: solid, dashed, or double-dashed. Arrowheads for the

polyline can be set to hollow or filled and the arrowheads can appear at both endpoints of the polyline, at the start point or end point of the line, or not at all. The visibility of the polyline can be toggled as well as the selectability of the polyline. If a polyline is not selectable, then it cannot be the target for events.

Polylines can have menus associated with them. This means that pressing the right mouse button over any points on the polyline causes the associated menu to be displayed. To associate a menu with a polyline, the menu itself needs to exist. The **ADL_menu** class defines how to create menu instances. Applying the set_application_menu() method associates the menu with the polyline.

Polylines can generate *select* events when the left mouse button is pressed while over any point of the line, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed while over any point of the line, *key* events when keyboard keys are pressed while over any point of the line, and *move* events when the left mouse button is pressed while over any control point of the polyline, and then dragged to a new location. See section 5 for more information on events and object behavior.

The class **ADL_polyline** defines the methods particular to a polyline instance, including creation and destruction.

4.5.6 Ellipses

Ellipses define elliptical graphical objects. An ellipse object can be drawn with a specified thickness, and the outline can be solid, dashed, or double-dashed. The ellipse can be drawn hollow or filled (specifying clear as the background color causes the object to appear hollow; any other color draws the object filled).

An ellipse is a child of the canvas containing the object. The *size_x* attribute is used to specify the width of the ellipse and the *size_y* attribute is used to specify the height of the object. The *angle* attribute is used to specify the rotation of the ellipse in degrees relative to the three o'clock position.

The placement of the ellipse can be relative to the upper left corner of the canvas, or relative to another existing object within the canvas. To position the ellipse relative to an existing object, specify that object by setting the *object* attribute. If you want the center of the ellipse relative to the center of *object*, then set the *center* attribute to true; otherwise the upper left corner of the ellipse's bounding box will be relative to the upper left corner of the existing object. The *x* and *y* attributes specify the offset of the ellipse from the upper left corner of the canvas, or from the upper left corner of the existing object, or from the center of the existing object (depending upon the values of *object* and *center*).

The *line_color* and *fill_color* of the ellipse can be specified. The thickness of the outline of the ellipse can also be specified by giving a positive integer value – the larger the number the thicker the line. *Chiron-1* provides three options for setting the style of the object's outline: solid, dashed, or double-dashed. The visibility of the ellipse can be toggled as well as the selectability of the object. If an ellipse is not selectable, then it cannot be the target for events.

Ellipses can have menus associated with them. This means that pressing the right

mouse button over the object causes the associated menu to be displayed. To associate a menu with an ellipse, the menu itself needs to exist. The **ADL_menu** class defines how to create menu instances. Applying the set_application_menu() method associates the menu with the ellipse.

Ellipses can generate *select* events when the left mouse button is pressed over the object, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed over the object, *key* events when keyboard keys are pressed while over the object, and *move* events when the left mouse button is pressed while over the object, and then dragged to a new location. See section 5 for more information on events and object behavior.

The class **ADL_ellipse** defines the methods particular to an ellipse instance, including creation and destruction.

4.5.7 Circles

Circles are defined as specialized ellipses. The functionality of circles is identical to that of ellipses, only their creation is different. Rather than specifying a *size_x* and a *size_y*, you only need to specify the diameter of the circle. Also, there is not an *angle* attribute, as it makes no sense to rotate a circle. See the description of class **ADL_ellipse** for information on other attributes and methods available to an instance of class **ADL_circle**.

The class **ADL_circle** defines the methods particular to a circle instance, including creation and destruction.

4.5.8 Text

Text defines textual images displayed within a canvas. A text object is a child of the canvas containing the object. The *font* attribute is used to specify the font of the displayed text.

The placement of the text can be relative to the upper left corner of the canvas, or relative to another existing object within the canvas. To position the text relative to an existing object, specify that object by setting the *object* attribute. If you want the center of the text relative to the center of *object*, then set the *center* attribute to true; otherwise the upper left corner of the text's bounding box will be relative to the upper left corner of the existing object. The x and y attributes specify the offset of the text from the upper left corner of the canvas, or from the upper left corner of the existing object, or from the center of the existing object (depending upon the values of *object* and *center*). For example, if you want the text to appear centered inside a rectangle object, give the rectangle object as the *object* and set *centered* to true, leaving the x and y attributes set to zero.

The foreground and background colors of the text object can be specified. In most cases the background color will be ignored. However, if the foreground color of the text object is the same as the background color of the canvas, the object will not appear. To correct this, the text will be drawn inverted, meaning that the background color specified will shade the text, providing the necessary contrast.

The visibility of the text can be toggled as well as the selectability of the object. If a text object is not selectable, then it cannot be the target for events.

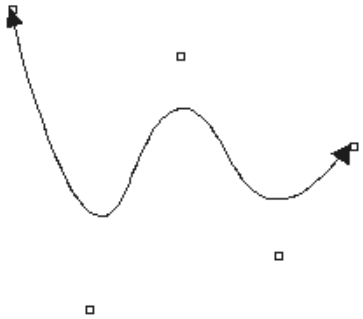

Figure 16: This figure shows a spline with five control points indicated by the squares

Text objects can have menus associated with them. This means that pressing the right mouse button over the object causes the associated menu to be displayed. To associate a menu with a text object, the menu itself needs to exist. The **ADL_menu** class defines how to create menu instances. Applying the set_application_menu( ) method associates the menu with the text object.

Text objects can generate *select* events when the left mouse button is pressed over the object, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed over the object, *key* events when keyboard keys are pressed while over the object, and *move* events when the left mouse button is pressed while over the object, and then dragged to a new location. See section 5 for more information on events and object behavior.

The class **ADL_text** defines the methods particular to an text instance, including creation and destruction.

4.5.9 Splines

The ADL_spline class offers a simple interface to create general B-splines. The natural cubic spline is an interpolating curve that approximates shapes similar to an elastic object stretched to form a certain outline. A spline is described by a set of control points where the objective is not to make the line pass through the points, but to use the points to control the course of the curve. It is not global in nature. e.g. the curve is only locally controlled by the points it interpolates or approximates.

Splines can generate *select* events when the left mouse button is pressed while over any point of the line, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed while over any point of the line, *key* events when keyboard keys are pressed while over any point of the line, and *move* events when the left mouse button is pressed while over any control point of the spline, and then dragged to a new location. See section 5 for more information on events and object behavior.

4.5.10 Bitmaps

Bitmaps define graphical images displayed within a canvas.

A bitmap is a child of the canvas containing the object. Its image is specified by creating an instance of **ADL_server_image**.

The placement of a bitmap can be relative to the upper left corner of the canvas, or relative to another existing object within the canvas. To position the bitmap relative to an existing object, specify that object by setting the *object* attribute. If you want the center of the bitmap relative to the center of *object*, then set the *center* attribute to true; otherwise the upper left corner of the bitmap's bounding box will be relative to the upper left corner of the existing object. The *x* and *y* attributes specify the offset of the bitmap from the upper left corner of the canvas, or from the upper left corner of the existing object, or from the center of the existing object (depending upon the values of *object* and *center*).

The foreground and background colors of the bitmap can be specified. The visibility of the bitmap can be toggled as well as the selectability of the object. If a bitmap object is not selectable, then it cannot be the target for events.

Bitmaps can have menus associated with them. This means that pressing the right mouse button over the object causes the associated menu to be displayed. To associate a menu with a bitmap object, the menu itself needs to exist. The **ADL_menu** class defines how to create menu instances. Applying the set_application_menu() method associates the menu with the bitmap.

Bitmap objects can generate *select* events when the left mouse button is pressed over the object, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed over the object, *key* events when keyboard keys are pressed while over the object, and *move* events when the left mouse button is pressed while over the object, and then dragged to a new location. See section 5 for more information on events and object behavior.

The class **ADL_bitmap** defines the methods particular to a bitmap instance, including creation and destruction.

4.5.11 GIF[2] Images

The ADL provides the artist writer with a GIF class that provides for simple display of GIF images within a canvas. These images have all of the attributes of the Bitmap class, with the exception of specification of the foreground and background color. When creating a ADL_gif object it is important to provide the full pathname to the file containing the GIF (see figure 17).

GIF objects can generate *select* events when the left mouse button is pressed over the object, *menu* events when the right mouse button is pressed, a menu is displayed, and a menu choice is made by the end user, *adjust* events when the middle mouse button is pressed over the object, *key* events when keyboard keys are pressed while over the object, and *move* events when the left mouse button is pressed while over the object, and then dragged to a new location. See section 5 for more information on events and object behavior.

[2]The Graphics Interchange Format(c) is the Copyright property of CompuServe Incorporated. GIF(sm) is a Service Mark property of CompuServe Incorporated.

```
canvas    : ~CSL.ADL_canvas;
world_gif : ~CSL.ADL_gif;

world_gif := ~Apply(ADL_gif,
                    create,
                    parent => canvas,
                    gif_file => "/usr/home/jdoe/world.gif");
```

Figure 17: GIF creation example.

The class **ADL_gif** defines the methods particular to a GIF object.

4.6 Text Windows

Text windows allow the end user to display and edit a sequence of ASCII characters. *Chiron-1* has imported XView's text editor for this purpose. The editor provides basic text editing features such as inserting arbitrary text into a file. It also provides more complex operations such as searching for and replacing a string of text.

Figure 18 depicts a text and tty window.

A text window is created as a child of a frame (either a base frame or a popup frame), and therefore is a subwindow of that frame. The location of the text subwindow within the frame needs to be specified. This can be done explicitly, via x and y attributes which specify the location of the text window (in pixels) relative to the upper left corner of the parent frame. Or the text window location can be specified relative to other existing subwindows within the frame. For example, it can be specified that the text window be placed below an existing subwindow and/or to the right of another existing subwindow. And finally, the text subwindow location can assume its default location by setting the *auto_placement* attribute to true.

A text window has *rows* and *cols* attributes which specify the width and height of the text window in terms of the given font. The *rows* attribute specifies the number of text rows that the window should display – the larger the font, the longer the window. The *cols* attribute specifies the number of text columns that the window should display – the larger the font, the wider the window.

A font and cursor can be specified for a given text window. They need to be created as instances of the **ADL_font** and **ADL_cursor** classes, respectively. The font specified plays an important part in the size of the resulting window (not just the font displayed in the window). The cursor specifies the shape of the cursor when inside the text window.

Methods are provided to set/read the textual contents of the window to/from either a string or a file. The class **ADL_text_window** defines the methods particular to a text window instance, including creation and destruction.

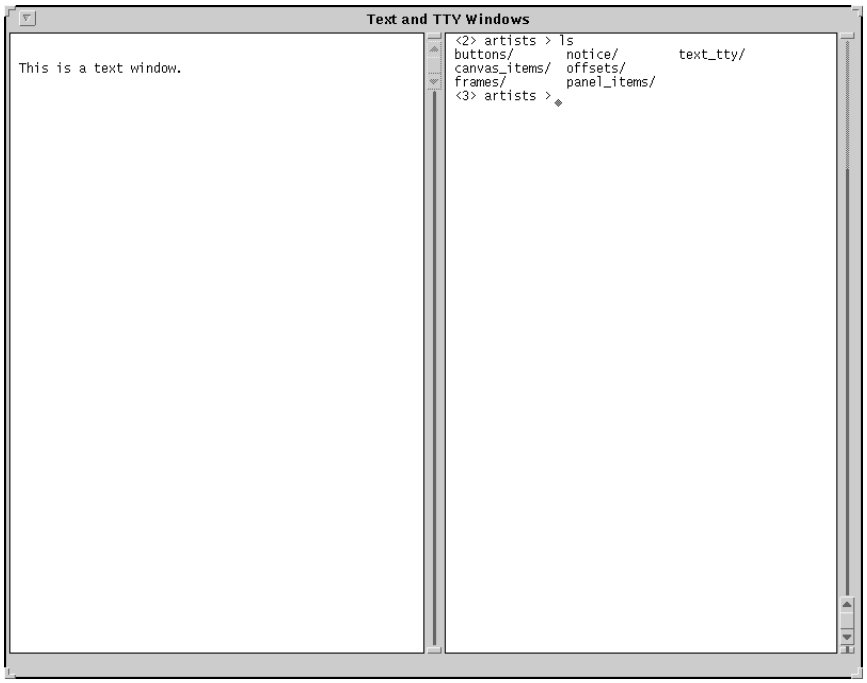

Figure 18: The window to the left is a text window which allows text to be displayed and edited. The window to the right is a tty window which emulates a standard terminal.

4.7 TTY Windows

A TTY (or terminal emulator) subwindow emulates a standard terminal, the principal difference being that the row and column dimensions of a tty subwindow can vary from that of a standard terminal. In a tty subwindow, you can run arbitrary programs, including a complete interactive shell. The tty subwindow accepts the standard ANSI escape sequences for doing ASCII screen manipulation, so you can use *termcap* or *termio* screen-handling routines.

Note that you can have only one tty subwindow per process.

A tty window is created as a child of a frame (either a base frame or a popup frame), and therefore is a subwindow of that frame. The location of the tty subwindow within the frame needs to be specified. This can be done explicitly, via x and y attributes which specify the location of the tty window (in pixels) relative to the upper left corner of the parent frame. Or the tty window location can be specified relative to other existing subwindows within the frame. For example, it can be specified that the tty window be placed below an existing subwindow and/or to the right of another existing subwindow. And finally, the tty subwindow location can assume its default location by setting the *auto_placement* attribute to true.

Figure 19: A notice object "grabs" the entire screen and queries the end user for input.

A tty window has *rows* and *cols* attributes which specify the width and height of the tty window in terms of the given font. The *rows* attribute specifies the number of text rows that the window should display – the larger the font, the longer the window. The *cols* attribute specifies the number of text columns that the window should display – the larger the font, the wider the window.

A font and cursor can be specified for a given tty window. They need to be created as instances of the **ADL_font** and **ADL_cursor** classes, respectively. The font specified plays an important part in the size of the resulting window (not just the font displayed in the window). The cursor specifies the shape of the cursor when inside the text window.

The class **ADL_tty** defines the methods particular to a tty window instance, including creation and destruction.

The Motif version of the *Chiron* server does not support the ADL_tty class.

4.8 Notices

A notice is a popup window used to notify the user of a problem or to ask him or her a question that requires an immediate response. The notice "grabs" the entire screen so no other windows or applications can receive input until the user responds to the notice. Figure 19 shows an example of a XView notice window. Your application has control over the messages that are displayed within the notice window as well as the choices available to the user as responses.

First you create a notice and then you invoke the notice at critical parts of the application. To create a notice, you need to specify the *message* to display as well as *yes* and *no* labels. For example, the *message* might be "Do you REALLY want to overwrite this file?" with the *yes* and *no* labels set to "Save" and "Cancel".

To display the notice and query for the necessary input from the end user, apply the invoke_notice() method. This method displays the notice, grabs the server, and waits for the user to make a selection on one of the available button choices. Once a choice has been

made, the notice disappears and an integer value is returned, indicating the choice made. A value of one indicates that the *yes* button was pressed while a value of zero indicates that the *no* button was pressed.

Using the Motif server, if the artist writer specifies a null string for either label, then the button for that label is not created and the server will return the value corresponding to the button that was created.

The class **ADL_notice** defines the methods particular to a notice instance, including creation and destruction.

4.9 Menus

Menus play an important role in an application's user interface. Menus may be attached to buttons, canvases, and application objects.

There are three different types of menus: popup, pulldown, and pullright menus. The general term popup menu may describe all three types in certain contexts since menus are "popped up". However, pulldown and pullright menus have distinct characteristics that make them unique.

4.9.1 Popup Menus

Popup menus are displayed when the user selects the right mouse button over objects such as canvases and application objects.

Creation. Popup menus are created via a call to create_popup(). The artist writer then fills the menu with options by calling add_menu_choice(), or with submenus by calling create_submenu(). The last step in creating a popup menu is attaching it to some object. Popup menus may be attached to an instance of a canvas class or an instance of any application object class. To attach a menu to one of these objects call the appropriate set_application_menu() method.

Event handling. An instance of a popup menu may be attached to many different application objects. Therefore, returning a menu selection event through the menu is not sufficient to determine which application object prompted the menu selection. Popup menus return their selection events through the object that prompted them. See section 5 for more information on menu events.

4.9.2 Pulldown Menus

Pulldown menus are associated with buttons. When a pulldown menu is created, a menu button is created automatically. Menu buttons have a set of choices associated with them that the user can access only via the pulldown menu. When the user presses the right mouse button over a menu button, the choices are displayed in the form of a pulldown menu.

Creation. Pulldown menus are created via a call to create_pulldown(). The artist writer can then fill in the menu with options by calling add_menu_choice(), or with submenus by calling create_submenu(). Unlike popup menus, pulldown menus are not attached to any other object. Pulldown menus always appear as a button in a panel.

Event handling. Since the artist does not have a handle to the menu button, when the user makes a menu selection, the event is returned through the menu. (Note that the ADL implements the Motif model for pulldown menus. In XView, buttons are created explicitly by the user and menus are associated with buttons after their creation. Also, in the XView model, menu events are returned through the button and not through the menu.) See section 5 for more information on menu events.

4.9.3 Pullright Menus

The ADL provides for items in either popup or pulldown menus to have pullright menus associated with them. Also known as sub-menus or cascading menus, these menus are activated by the user dragging the right mouse button to the right of a menu item that has an arrow pointing to the right. The cascading menu that results can also have menu items with pullrights attached.

Creation. Pullright menus are created via a call to the create_submenu() method. The create_submenu() method is provided with a parent parameter indicating to which menu the submenu should be attached. The artist writer can then fill in the submenu with options by calling add_menu_choice(), or with submenus by calling create_submenu().

Event handling. When a menu selection is made through a pullright menu, the select event is returned through the same object that events on the root menu are returned. See section 5 for more information on menu events.

There is no physical limit to the number of menu choices appearing in a menu instance, or the number of pullright menus in a sequence. The class **ADL_menu** defines the methods particular to a menu instance, including creation and destruction.

4.10 Scrollbars

Scrollbars are used to change what you view in a subwindow. For instance, in a text subwindow, scrollbars are used to scroll through a document. In a canvas subwindow, scrollbars can be used to see another portion of the paint window (the virtual area of the canvas) which can be larger than the canvas subwindow.

Scrollbars are described in OPEN LOOK using the visual metaphor of an elevator riding on a cable, which is attached at both ends to anchors. The elevator contains directional arrows and a drag box. A subwindow can have vertical or horizontal scrollbars. Vertical scrollbars are placed to the right of the subwindow while horizontal scrollbars are placed at the bottom.

The scrollbar's look-and-feel is related to the size of the object it scrolls. Attributes are associated with each of the following terms:

Orientation The orientation of a scrollbar indicates whether it is horizontal or vertical.

Object Length The length of the object is registered with the scrollbar. The proportional indicator (the darkened part of the elevator cable) uses this value. For example, the object length for a text subwindow is the number of lines in the editing buffer.

Page Length When the object length is larger than what the view window can contain, the overall area is broken up into pages. When the user selects the elevator cable, the scrollbar scrolls in page segments in the direction of the cursor (e.g., left, right, up or down) relative to the elevator.

Unit Length When the user clicks on the elevator arrows, the scrollbar scrolls one unit. Units are measured in pixels, so arbitrary or abstract objects that are to be scrolled should be measured in terms of pixels so that scrolling seems consistent with the object. For example, a text subwindow sets its scrollbar's unit length to the size of the characters in its font. Unit scrolling results in the window moving line by line up or down.

View Length The view length is the same size as the height or width of the subwindow the scrollbar is associated with depending on the scrollbar's orientation.

Once a scrollbar has been created, it needs to be attached to the object to be scrolled. Text windows automatically have vertical scrollbars associated with them. Canvases, on the other hand, need to have scrollbars explicitly created and attached before any scrolling can occur. To attach scrollbars to canvases, use the set_vertical_scrollbar() and set_horizontal_scrollbar() methods defined for the adl_canvas class.

It should be noted that *Chiron-1* handles the scrolling of the canvas. The application will never be notified that scrolling has occured.

The class **ADL_scrollbar** defines the methods particular to a scrollbar instance, including creation and destruction.

4.11 Server Images

A server image is a graphical image which can later be used as the image for a button, message, bitmap, icon, or cursor. Thus creating a server image by itself does not create a visual object – it must be used as input to other objects to appear on the screen.

There are three ways to create a server image:

- by specifying the image as an array of chars (X11 bitmap format)
- or as an array of shorts (Sun bitmap format) – *since Motif is not a Sun product, it does not support images in the form of short arrays.*
- or by specifying an X11 bitmap file name – the preferred method of creating a server image.

If creating the image either as an array of chars or as an array of shorts, you also need to specify the *width* and *height* of the resulting image.

The class **ADL_server_image** defines the methods particular to a server image instance, including creation and destruction.

4.12 Icons

An end user may close an application to save space on the display. The program will still be running and it may even be active, but it is not receiving input from the end user. In order to represent the application in its closed state, an icon is used. An icon is a small picture that represents the application.

An icon is a child of a base frame (the base frame the icon represents in its closed form).

The image of the icon is specified by first creating an instance of class **ADL_server_image**, and passing that in to be the image of the icon. The size of the icon is dependent upon the size of the server image created. It is also possible to specify the location of the icon using the x and y attributes.

The class **ADL_icon** defines the methods particular to an icon instance, including creation and destruction.

4.13 Cursors

A cursor is an image that tracks the mouse on the display. Each window (base frame, popup frame, panel, canvas, tty window, and text window) has its own cursor which you can change. There are some cursors defined by the window manager that correspond to specific window manager operations such as resizing or dragging windows. For these cases you cannot redefine a cursor. However, for windows in your application, you can assign any cursor image you like.

In XView, currently cursors cannot be set for frames – only for panels and canvases. Motif allows a cursor to be associated with any window.

To specify the image of the cursor, you can create an image using class **ADL_server_image**, or you can use a standard predefined cursor image. If you choose to create your own image, you need to specify the x and y hot spots of the image.

The class **ADL_cursor** defines the methods particular to a cursor instance, including creation and destruction; additionally, the class defines the predefined standard cursor images.

4.14 Fonts

In X, a large number of fonts are provided (on the server). Some of these are fixed-width fonts, such as Courier, and some are proportional fonts, such as Times-Roman. **The command 'xlsfonts' should tell you which fonts your environment supports.**

Within a font family there are often different styles, such as **bold** or *italic*, and different point sizes. For example, Helvetica bold 14 refers to the Helvetica font family; bold is the style, and 14 is the point size. Not all server fonts have a variety of styles and sizes. These special-purpose fonts are generally specified by name only – there are no corresponding

styles or families for these fonts. When accessing fonts, you typically want to specify a font either by name or by the family, style, and size of the font.

Specification of a font by family is only supported by XView. There is no functionality in the Xtoolkit or Motif to support this method of naming. Consideration is being given to building a font database in the ADL from which the family, style, size naming convention can be used. Artist writers are advised to use the create_by_name() method for portability.

In addition to font creation and destruction, the methods in the **ADL_font** class also return information on the size of a text string in the given font, as well as the size of a given character in a given font.

4.15 Colors

Chiron-1 provides a list of colors available to artist writers. These can be used to specify the foreground and background colors for various class instances. If the application is executed on a monochrome machine, these colors are mapped to black and white.

Many class objects have their foreground and background colors defaulted to ADL_color_not_specified. This means "inherit parent colors". So, if a frame's foreground color is set to blue, all its subwindows will have a blue foreground color, unless it is explicitly set otherwise.

The color clear is used by application objects to determine whether or not the object should be drawn filled or hollow. For example, if you specify that a circle should have orchid as its foreground color and clear as its background color, an orchid-colored hollow circle will be drawn ... allowing the objects beneath the circle to show through. Specifying orchid as its foreground color and white as its background color will draw a white circle outlined in orchid ... and the circle will be opaque, meaning objects beneath the circle will not show through.

The class **ADL_color** (which actually is not a class in C++ terminology) lists out all the colors supported by *Chiron-1*.

4.16 Objects

The **ADL_object** class is the root class of the hierarchy. It contains attributes and methods common to all hierarchy instances. For example, each object instance has a unique id, a parent (which may be NULL), etc. Although you cannot create an instance of class **ADL_object** explicitly, all of the objects within the hierarchy are sub-classed from class **ADL_object**, thus they inherit the attributes and methods defined in this class. Methods to query the id and parent of a given instance are defined here.

4.17 Drawables

Drawables, for the most part, are objects that appear on the screen. Although you cannot create an instance of class **ADL_drawable** explicitly, most of the objects within the hierarchy are sub-classed from class **ADL_drawable**, thus they inherit the attributes and methods defined in this class.

Methods to query and alter the location, size, color, and visibility of an object are defined here.

4.18 Windows

Windows, for the most part, are objects that appear as windows on the screen. Although you cannot create an instance of class **ADL_window** explicitly, some of the objects within the hierarchy are sub-classed from class **ADL_window**, thus they inherit the attributes and methods defined in this class. Methods to query and alter the window font and cursor are defined here.

4.19 Auxiliary Classes

Separate from the hierarchy are two classes, **Object_List** and **Corner_List** which are referred to as *auxiliary classes*. They are not connected to the hierarchy because they are used only to help define other objects within the hierarchy.

4.19.1 Corner List

Corner list is used to define a list of (x, y) points that make up the shape of an **ADL_polyline**, **ADL_polygon**, or an **ADL_spline**.

An **ADL_polyline** is made up of at least two points, including a start point and an end point. The first point given is, by default, considered to be the start point, and the last point given is, by default, considered to be the end point. The declaration of the points are relative to the first point. Typical use would declare the first point as (0, 0), and the successive points as offsets from (0, 0) to define the line. Once the line is defined it can be located anywhere with the set_x() and set_y() methods (defined in class **ADL_drawable**). If you look at the create function for **ADL_polyline**, you will see that the parameters include a **Corner_List** definition and an (x, y) location – thus specifying (0, 0) as the start point does not necessarily mean that the start of the line will be drawn at (0, 0); the placement of the line depends upon the x and y parameters of the **ADL_polyline** create routine.

An **ADL_polygon** is made up of a set of points. All the points specified are relative to the first point, which typically will be defined as (0, 0). Thus, a triangle might be specified as:

```
Corner_List *tri;
/*  3 points make up the shape  */
tri = Corner_List::create (3);
/*  setting zero-th index, etc  */
tri->set_pair (0, 0, 0);
tri->set_pair (1, 3, 3);
tri->set_pair (2, -3, 3);
```

to create an image like:

```
          *    . (0, 0)

(-3, 3) .        . (3, 3)
```

Note that the polygon will *close* itself. That is, you do not need to specify that the first and last points are the same.

Once the triangle is defined, it can be moved anywhere. Remember that the *location* of an object is the location of the upper left corner of the bounding box of the object. The * in the diagram above indicates the upper left corner of the triangle object above.

The class **Corner_List** defines the methods particular to a corner list instance, including creation and destruction.

4.19.2 Object List

Object list is used to define a list of **ADL_application** objects. This list is then used to specify the z-axis ordering of overlapping objects within the canvas, via the *above* and *below* attributes found in the create() routines for all application objects.

The class **Object_List** defines the methods particular to an object list instance, including creation and destruction.

5 Events and Event Processing

A main task for any artist is the processing of server and client events. Server events are a fixed set of events that can originate in the server. They correspond to window events, such as select and resize events. Client events are a more flexible set of events that originate in the client. In general, they correspond to notifications of ADT calls, but they can also be expanded to include user defined events. In the following sections we discuss both kinds of events in terms of their definition, registration, routing, and handling.

5.1 Server Events

5.1.1 Definition

Chiron defines the following six kinds of server events:

menu a menu associated with an object was displayed and a menu choice was made. Buttons, canvases, and application objects can have menus associated with them.

select the object was selected either by the left mouse button or by pressing the return key.

key a keyboard key was pressed while the mouse was over the object.

adjust the middle mouse button was pressed over the object.

move the left mouse button was pressed over the object, dragged, and then released.

resize the user has resized a base or popup frame.

Server events are reported in (object, event) pairs; an artist is notified of the event that occurred and the object which received the event. For example, if a button is pressed, the artist will be given both the event description and the button instance.

A server event is represented as a discriminated record and an artist may query the fields of this record to get information about the event. *Chiron* defines the *Chiron_Event_Type* record type as follows:

```
type Chiron_Event_Type(Kind : Chiron_Event_Kind) is
  record
    Mouse_X  : integer;
    Mouse_Y  : integer;
    Time     : integer;
    Num_Val  : integer;
    Text_Val : Str;
    case Kind is
      when Menu_Event =>
        null;
      when Select_Event =>
```

```
            null;
          when Adjust_Event =>
            null;
          when Key_Event =>
            Key_Code : character;
          when Move_Event | Resize_Event=>
            Dest_X : integer;
            Dest_Y : integer;
        end case;
      end record;
```

Descriptions of the fields of an event type are given below:

Mouse_X Defines the x position of the mouse relative to the window where the event occurred.

Mouse_Y Defines the y position of the mouse relative to the window where the event occurred.

Time Defines the time-stamp of the event (useful for comparison).

Num_Val Defines a value dependent upon the type of event reported and the type of object receiving the event. It defines the current value for **ADL_num_fld** objects, **ADL_choice** objects, and **ADL_slider** objects that are the target for *select* events. It defines the button id for **ADL_button** objects that are the target for *select* or *menu* events. It defines the index in the corner list corresponding to the point selected within a line for **ADL_polyline** objects that are the target for *select* and *move* events. In all other cases the value is zero.

Text_Val Defines a value dependent upon the type of event reported and the type of object receiving the event. It defines the text of the menu choice if the event is of type *menu*. It defines the value for **ADL_text_fld** objects that are the target of *select* events. In all other cases the value is an empty string.

Key_Code Valid for *key* events only, it is the ASCII character corresponding to the key pressed.

Dest_X Valid for *move* and *resize* events only. For a *move* event, it is the x position of the mouse when the button was released. For a *resize* event, it is the new width of the frame.

Dest_Y Valid for *move* and *resize* events only, For a *move* event, it is the y position of the mouse when the button was released. For a *resize* event, it is the new height of the frame.

Class	Menu	Select	Key	Adjust	Move	Resize
button		yes				
pulldown menu	yes					
number field		yes				
text field		yes				
choice		yes				
canvas	yes		yes	yes		
app. object	yes	yes	yes	yes	yes	
base frame						yes
popup frame						yes

Figure 20: This table illustrates which class objects are capable of receiving the various event kinds.

5.1.2 Registration

Server event registration can be thought of as associating behaviors to objects, or rather, specifying how an artist should respond to server event on particular objects. This is done by defining callback routines for object/event pairs.

Event Targets Not all class objects can be the target for events – the class might be too abstract (like fonts), or the event might be handled automatically by the window manager (like clicking on icons), or the event might be ignored if not channeled to a specific object (like clicking in a panel, but not on a panel item). Figure 20 provides a table showing the event kinds that the different class objects are capable of receiving.

In most cases, the object of an event is easy to determine. For example, if an application object is selected using the left mouse button, then the application object itself is the object of the event. If a number is entered into a number field, then the number field is the object of the event. However, in the case of menu selection, the object of the event varies according to the type of menu through which the selection was performed (see section 4.9 for a description of the different menu types).

For a menu selection that is performed through a popup menu, the object of the event is considered to be the object to which the menu is attached. Since a single instance of a popup menu may be attached to multiple objects, returning a menu select through the menu itself is not sufficient to determine which object prompted the selection.

For a menu selection that is performed through a pulldown menu, the object of the event is considered to be the menu itself and not the menu button. Remember that the menu button is created implicitly and the artist therefore does not have a handle to it.

For a menu selection performed through a pullright menu, also known as a sub-menu or a cascading menu, the object of the event is the same object through which selection events on the root menu are reported. In other words, if the root menu is a popup menu, then

selection events on a pullright menu within that menu are reported on the object to which the root menu is attached. If the root menu is a pulldown menu, then selection events on a pullright menu within that menu are reported on the root menu itself.

Setting object behaviors Since a single object may receive several different kinds of events, we set the behavior of an objects by declaring an array of callback routines indexed by the various event kinds. The artist writer essentially registers for an event, by providing the address of callback routines for that event within a behavior array. Finally, the behavior is associated to the object via a call to the Lo-CAL *Set_Behavior* procedure

For example, if our application creates a Quit button, we might define a behavior array as follows:

```
quit_behavior : Behavior_Array_Type
                := (others => system.no_addr);

quit_behavior(Select_Event) := handle_quit'address;
```

and then associate it with our button like this:

```
~Set_Behavior (quit_button,
               quit_behavior);
```

The *~Set_Behavior* procedure can be used to associate behaviors with an entire class of objects as well as with an instance of a class. In this case we give an ADL class as the first parameter, i.e.

```
~Set_Behavior (ADL_button,
               quit_behavior);
```

~Set_Behavior is described in greater detail in section 6.

Finally, an artist can "unregister" for server events by setting the callback routine for an object/event pair back to *System.No_Addr* explicitly.

Declaring callback routines Users may declare as many callback routines as they need. The only restriction is that they must have the following interface:

```
procedure <callback_proc_name>
    (Object : Chiron_Standard_Library.Object_Type;
     Event  : Chiron_Standard_Library.Event_Ptr);
```

The *Object* parameter is the object of the event as indicated by the server. It can be any the graphical objects that is eligible event target. If a callback is registered for an event on a whole class of objects, the user can use this parameter to distinguish between the various instances of the class. The *Event* parameter is a pointer to a server event record. It can be dereferenced for information about the event.

```
package Stacks is
   type Stack is private;
   Stack_Empty: exception;
   function Create (Size : INTEGER) return Stack;
   procedure Destroy (S : in out Stack);
   procedure Push (S : in out Stack;
                   X : in INTEGER);
   function Pop (S : Stack) return INTEGER;
   function Top (S : Stack) return INTEGER;
   function Depth (S : Stack) return INTEGER;
   function Max (S : Stack) return INTEGER;
private
            .
            .
            .
end Stacks;
```

Figure 21: Simple stack ADT

5.1.3 Routing

When an event arrives from the server, the *Mapper* component within the client routes it to the appropriate artist based on the object of the event. This would be the artist that declared the object. The *Mapper* also keeps track of server event registration so it must the address to the appropriate callback routine and it passes this information along with the object and event to the artist. The Mapper will first check for a callback associated with the object instance. If there is no callback associated with the instance, it will look for a callback associated with the object's class. If no callback is found, i.e. the artist has not registered for that particular event, the mapper will not notify the artist of this event.

5.1.4 Handling

Upon receiving the event, a server event handler within the artist invokes the callback routine located at the address given at registration.

5.2 Client Events

5.2.1 Definition

The client event type is generated from the set of ADTs that a client will be depicting (with one or more artists). An example of a client event type generated from the stack ADT given figure 21 is shown in figure 22.

Type *Client_Event_Kind* enumerates all possible client events. Each enumeration is given as the ADT name appended with the operation name. If a client depicts more than

```
with Stacks;  use Stacks;

package Client_Events is

   type Client_Event_Kind is (
      Stacks_Create,
      Stacks_Destroy,
      Stacks_Push,
      Stacks_Pop,
      Stacks_Top,
      Stacks_Depth,
      Stacks_Max,
      Null_Event
   );

   type Client_Event_Type (Kind : Client_Event_Kind := Null_Event) is record
      case Kind is
         when Stacks_Create =>
            Stacks_Create_Size: INTEGER;
            Stacks_Create_Result: Stack;
         when Stacks_Destroy =>
            Stacks_Destroy_S: Stack;
         when Stacks_Push =>
            Stacks_Push_S: Stack;
            Stacks_Push_X: INTEGER;
         when Stacks_Pop =>
            Stacks_Pop_S: Stack;
            Stacks_Pop_Result: INTEGER;
         when Stacks_Top =>
            Stacks_Top_S: Stack;
            Stacks_Top_Result: INTEGER;
         when Stacks_Depth =>
            Stacks_Depth_S: Stack;
            Stacks_Depth_Result: INTEGER;
         when Stacks_Max =>
            Stacks_Max_S: Stack;
            Stacks_Max_Result: INTEGER;
         when Null_Event => null;
      end case;
   end record;

   type Client_Event_Ptr is access Client_Event_Type;

            .
            .
            .

end Client_Events;
```

Figure 22: Client event type

one ADT, all of the operations from each ADT will be enumerated within this type. If the ADT overloads an operator, the second occurrence will be appended with a 2, and the third with a 3, and so on.

The *Client_Event_Type* is defined as a record discriminated by the *Client_Event_Kind*. Each variant of the record specifies the data associated with each event kind, where the data corresponds to the formal parameters of the corresponding ADT operation. Each field is given as the *Client_Event_Kind* name appended by the parameter name, and each field type is the type of the corresponding formal parameter.

The client event type may be extended by the artist writer to include any event. For example, if an artist needs to pass a canvas to another artist, it could define an event where the data passed is an ADL_Object. In fact, events need not be based on ADT operations at all. If no ADT is specified, the client event will include only the *Null_Event*. The type structure of the client will be satisfied, and no wrappers or access controllers need be generated.

In addition to the client event type, the Client_Events package also defines two arrays that hold useful information about the various client events. These definitions for the same stack ADT are given in figure 23. The *Client_Event_Source* array is used by the Client Dispatcher to determine which ADT an event originated from (if any).

The *Client_Event_Mode* is used by the Wrapper to determine whether an ADT event corresponds to a read or write operation. Since it would take a substantial amount of semantic analysis to determine this information, the values in the array are all initially set to Write. **It is the artist writers's responsibility to set these values correctly. Failure to do this could result in runtime deadlock.**

5.2.2 Registration

Artists express interest in various client events by registering with the client dispatcher.

Registering with the dispatcher Artists register and unregister for client events by calling the *Register_Event* and *Unregister_Event* entries in the client dispatcher. For example, suppose that an artist would like to to notified whenever an item is pushed onto a stack. It would register for the corresponding client event kind as follows:

```
Client_Dispatcher.Dispatcher.Register_Event
  (Local_Artist_ID,
   Client_Events.Stacks_Push,
   Handle_Stacks_Push'ADDRESS);
```

The artist passes the client dispatcher its artist id (used to locate the artist when notifying it of events), the event kind, and a callback routine. The callback routine will automatically be invoked if the artist is notified of the event.

To unregister for that event:

```
type Client_Event_Sources is (Stacks, None);

Client_Event_Source : constant array (Client_Event_Kind)
                          of Client_Event_Sources := (
  Stacks_Create => Stacks,
  Stacks_Destroy => Stacks,
  Stacks_Push => Stacks,
  Stacks_Pop => Stacks,
  Stacks_Top => Stacks,
  Stacks_Depth => Stacks,
  Stacks_Max => Stacks,
  Null_Event => None
);

type Client_Event_Modes is (Read, Write, None);

Client_Event_Mode : constant array (Client_Event_Kind)
                          of Client_Event_Modes := (
  Stacks_Create => Write,
  Stacks_Destroy => Write,
  Stacks_Push => Write,
  Stacks_Pop => Write,
  Stacks_Top => Read,
  Stacks_Depth => Read,
  Stacks_Max => Read,
  Null_Event => None
);
```

Figure 23: Addition client event declarations

```
Client_Dispatcher.Dispatcher.Unregister_Event
  (Local_Artist_ID,
   Client_Events.Stacks_Push);
```

Declaring callback routines Users may declare as many callback routines as they need. The only restriction is that they must have the following interface:

```
procedure <callback_proc_name>
    (Event  : Client_Event_Ptr);
```

The *Event* parameter is a pointer to a client event record. It can be dereferenced for information about the event.

5.2.3 Routing

When an ADT call is made, the ADT wrapper builds an instance of the appropriate event kind and notifies the client dispatcher of the event. The client dispatcher will notify any artists have have pre-registered for the given event kind. If, however there is a dedicated dispatcher for the ADT, the client dispatcher will immediately forward the event to that dispatcher. Non-ADT, or hand coded, events can be sent to the dispatcher from any entity. They can be particularly useful for communication between artists.

5.2.4 Handling

Upon receiving the event, a client event handler within the artist invokes the callback routine located at the address given at registration.

6 The Lo-CAL Language

Lo-CAL is a language which extends the Ada programming language to include an interface to the ADL Hierarchy (written in C++) defined in section 4. Two major requirements for this interface are that it must remain constant in spite of the fact that the ADL Hierarchy will evolve over time, and that the interface have an Ada-like syntax.

The Lo-CAL language extends the Ada language for three purposes: declaring graphical objects, accessing ADL methods, and setting the behaviors of graphical objects. The Lo-CAL language constructs for these purposes are described below. In all of these cases a ~ is used to indicate non-Ada code. This code is later translated by the Lo-CAL complier (*lcc*) into standard Ada.

6.1 Declaring graphical objects

All graphical objects are declared as ADL class types. The Ada equivalents for these class types are defined in the *Chiron_Standard_Library*. The type mark must be preceeded by a ~ because it requires special translation. For example, a declaration for a base frame would look something like this:

```
My_Base_Frame : ~CSL.ADL_base_frame;
```

Graphical objects declarations may not include explicit initializations, because they interfere with the Lo-CAL translation.

6.2 Accessing ADL Hierarchy via Lo-CAL

Artists access the ADL hierarchy using the Lo-CAL ~*Apply* calls. The ~*Apply* call is very general in that it can be used to access any ADL method and has a variable length parameter list. The parameter list can use named or positional notation, and parameter values may be unspecified if default values are given in the method's definition. For a listing of all ADL methods, their parameter list, return values, and defaults, the user is referred to the *ADL Reference Manual*, included in appendix E.

There are three forms of the ~*Apply* call. The signature of the first Apply call looks like this:

```
function ~Apply (class  : class_name;
                 method : method_name;
                 [parameters to class method])
    return object_type;
```

It is used to create instances of classes defined in the hierarchy. For example, to create a base frame, the Lo-CAL call might look like this:

```
frame := ~Apply (ADL_base_frame,
                 create,
                 frame_label => "My Artist");
```

The signature of the second Apply call looks like this:

```
procedure ~Apply (class    : class_name;
                  instance : object_type;
                  method   : method_name;
                  [<parameters to class method>]);
```

It is used to invoke non-create methods that do not return values. For non-create methods, class instance on which the method is to be applied must be provided. For example, to associate an icon (that has been created and assigned to the object, *new_icon*) to the base frame created above, the following call can be made:

```
~Apply (ADL_base_frame,
        frame,
        attach_icon,
        new_icon => icon);
```

Since **ADL_base_frame** is a sub-class of the **ADL_frame** class, all methods defined in the **ADL_frame** class (etc., on up to root class **ADL_object**) can be applied to an **ADL_base_frame** instance. So, for example, the method *set_left_footer* as defined in class **ADL_frame** can be applied like so:

```
~Apply (ADL_base_frame,
        frame,
        set_left_footer,
        new_left_footer => "Left Footer");
```

It is important to note that the class given in the *~Apply* call is the class that the instance corresponds to, not the class that the *method* corresponds to. For example, the method *set_left_footer* is defined in the class **ADL_frame**, but we still give **ADL_base_frame** as the class name because *frame* is of class **ADL_base_frame**. Also, for ~Apply calls that require an instance parameter, the given instance must be the same class as the given class parameter.

The signature of the third and final Apply call looks like this:

```
function ~Apply (class    : class_name;
                 instance : object_type;
                 method   : method_name;
                 [<parameters to class method>])
          return <return type of class method>;
```

It is used to invoke non-create methods that do return values. For example, to query the value of the frame_label, the following call can be made:

```
label := ~Apply (ADL_base_frame,
                 frame,
                 get_frame_label);
```

These three Apply calls are all you need to access all ADL class methods from within an artist.

There is one important restriction on the use of Lo-CAL calls – they cannot be nested. That is, you cannot use the value returned from one Lo-CAL call as direct input to a second Lo-CAL call. This will interfere with the Lo-CAL translation.

6.3 Setting Object Behavior

Chiron reports server events as event/object pairs, where *event* is one of the predefined server events (see section 5.1 and *object* is the graphical object on which the event occurred. How an object reacts to the set of all server events is known as the object's *behavior*.

Behaviors are defined within an array structure called a *behavior array*. A behavior array is indexed by the set of possible server events and the elements of the array are the addresses of handling procedures. For example, if an application creates a Quit button, the following behavior array might be defined:

```
quit_behavior : Behavior_Array_Type
                := (others => system.no_addr);

quit_behavior(Select_Event) := handle_quit'address;
```

This example, informs *Chiron* to call the *handle_quit* procedure (also defined within the artist) when a *Select_Event* is received, ignoring the all other event kinds.

Once the behavior has been defined, it must be associated with either an ADL class or with an instance of a class. For example, behaviors may be defined for all buttons or for a particular button instance.

The following defines the behavior for a button instance:

```
~Set_Behavior (quit_button,
               quit_behavior);
```

Alternatively, we could define the behavior for all buttons:

```
~Set_Behavior (ADL_button,
               quit_behavior);
```

The two *~Set_Behavior* calls above are two procedures defined in Lo-CAL. The first one accepts a class instance and the second one accepts a class name. The behavior of an instance takes precedence over the behavior of a class: as events are received, *Chiron* first checks if the particular instance has a defined behavior, and then it checks if a behavior has been defined for the instance's class.

For example, suppose we have defined the following:

```
quit_button_behavior : Behavior_Array_Type
                       := (others => system.no_addr);
button_behavior      : Behavior_Array_Type
                       := (others => system.no_addr);

quit_button_behavior(Select_Event)
                       := handle_quit_button'address;
button_behavior(Select_Event)
                       := handle_button'address;

~Set_Behavior (quit_button,
               quit_button_behavior);

~Set_Behavior (ADL_button,
               button_behavior);
```

If the quit button is pressed, the *handle_quit_button* callback routine will be called. If any other button defined in the application is pressed, the *handle_button* callback routine will be called.

7 Writing Artists

In this section we present detailed instructions on how to implement an artist. As an example, we will use the stack dialogue artist depicted in figure 2. The artist defines a main dialogue window from which a user may view and manipulate a stack ADT instance. Two fields display the top element of the stack and the current depth of the stack. The push button causes a popup window to appear that contains an field for the user to input the new value to be pushed onto the stack. And the pop button simply pops an element from the stack.

The source for this artist is listed in appendix C. We will refer to line numbers in this listing as examples of various aspects of artist writing.

The template for this artist will be essentially identical to that listed in figure 7, except the artist name will be *Dialogue_Artist*. In the following sections we will discuss the sections of code that must be written in order to define an artist, plus some additional topics, including associating graphical objects with ADT instances and avoiding deadlock.

7.1 Declaring graphical objects

Before we can create and initialize the graphical objects that make up our artist, we must declare them. The graphical object declarations for our example artist are found in lines 18-26. Here is the declaration for the base frame of the dialogue artist:

```
artist_frame: ~CSL.ADL_base_frame;
```

The typemark for all ADL objects must be qualified with the *~CSL* prefix. The ~ is an indicator to the Lo-CAL processor that it must perform some translation here. Do not assign a default value to any ADL object declarations as it will result in syntax error in the Lo-CAL translated code.

In this section of code, you can also declare any behavior arrays. Behavior arrays are used to define the behaviors (callback routines) of an class or class instance over a range of server events (see section 6.3.) The dialogue example declares one behavior array on line 27:

```
behaviors  : CSL.Behavior_Array_Type := (others => System.No_Addr);
```

A behavior array is indexed by the server event kinds. Each element is the address of a callback routine that should be invoked whenever the corresponding event is encountered. Users should always use an aggregate default assignment, as shown in our example, in order to initialize the array elements to no address specified.

Users may declare any useful variables in this section as well.

7.2 Writing event handling procedures

Once an artist has been initialized, it continuously processes client and server events. Handling procedures (or callback procedures) are associated with client and server events and are automatically invoked when the artist is notified of such events. Definitions for these event types are given in section 5.

Callback routines for server events must have the following signature:

```
procedure <callback_proc_name>
   (Object : Chiron_Standard_Library.Object_Type;
    Event  : Chiron_Standard_Library.Event_Ptr);
```

When an artist is notified of a server event, it is given both a description of the event (described in section 5.1) and the object on which the event occurred. If the callback is intended to be assigned to a whole class of objects, the *Object* parameter can be used to distinguish between various object instances. The *Event* parameter can be dereferenced to access various fields of the server event record. For example, it can be used to find the selection of a menu event, the integer value entered into a numeric field, the cursor position at the time of an event, etc.

Callback routines for client events must have the following signature:

```
procedure <callback_proc_name>
    Event  : Client_Event_Ptr);
```

When an artist is notified of a client event, it is passed an instance of a client event. For ADT-based operations, the fields of the client event can be used to obtain the values of the parameters and return values of the performed ADT operation.

The event handler routines for our dialogue example are declared on lines 41-119. Figure 24 shows a sample server event handler and client event handler from this example.

The Handle_Select procedure is a server event handler. This routine is intended to be a *select* event handler for multiple classes of objects. The *Object* of the event is used to determine which selection has occurred. If a push button is selected, we bring up popup window in which the user will enter the value to push onto the stack. If the pop button is pushed, we perform a pop operation through the stack wrapper package. All calls to depicted ADTs should occur indirectly through their wrapper. Notice that we do not update the graphical depiction at this point. We will wait to receive the client event that the wrapper will generate for the pop operation and it will be handled as a client event. *Artists are notified of client events that result from their own ADT calls.* If a value is entered into the popup field, a push is performed via a call to the stacks wrapper. The number to push onto the stack is obtained by selecting the *num_val* element of the event.

The Handle_Stacks_Push procedure is a client event handler. It is invoked whenever a *Stacks_Push* event is detected. We update the the top of stack field in our depiction using the *Stacks_Push_X* field of the client event (this corresponds to the value input parameter

```
procedure Handle_Select (object : CSL.Object_Type;
                         event  : CSL.Chiron_Event_Ptr) is
   value  : integer;
   result : integer;
begin
   if object = push_button then
      --show dialogue box:
      ~Apply(ADL_popup_frame,
            pop_up_frame,
            set_show,
            true);
   elsif object = pop_button then
      --if stack not empty:
      if ( wrapper_stacks.depth (artist_stack) > 0 ) then

         --pop:
         value := wrapper_stacks.pop (artist_stack);
      end if;
   elsif object = pop_up_field then
         --if stack not full:
         if ( wrapper_stacks.depth (artist_stack) <
              wrapper_stacks.max (artist_stack) ) then
           --push:
           wrapper_stacks.push (artist_stack, event.num_val);
         end if;
         --hide dialogue box:
         ~Apply(ADL_popup_frame,
               pop_up_frame,
               set_show,
               false);
   end if;
end Handle_Select;


procedure Handle_Stacks_Push (Event : Client_Event_Ptr) is
   value : integer;
begin
   --update depiction
   artist_stack := Event.Stacks_Push_S;

   ~Apply(ADL_text_fld,
         top_field,
         set_value,
         to_str(integer'image (Event.Stacks_Push_X)));
   value := wrapper_stacks.depth (artist_stack);

   ~Apply(ADL_text_fld,
         depth_field,
         set_value,
         to_str(integer'image (value)));
end Handle_Stacks_Push;
```

Figure 24: example event handler routines

for the Push operation of the Stack ADT). We get the new depth by querying the Stack ADT wrapper, then use that value to update the depth of stack field.

Here are a couple of reminders for *~Apply* call syntax when writing callback routines:

- *~Apply* calls cannot be nested.
- The *class* parameter given in *~Apply* calls must be the class of the given *instance*.

For a detailed description of how to access the ADL hierarchy using Lo-CAL *~Apply* calls see section 6.

It is important to note that the parameters of callback routines will be deallocated after the call completes.

7.3 Registering for client events

Artists may register or unregister for client events by making calls to the *Client_Dispatcher*. The *Client_Dispatcher* is a package that exports a task with the following two entries:

```
entry Register_Event (
   Artist_ID : CSL.Artist_ID_Type;
   Event_Kind : Client_Events.Client_Event_Kind;
   Handle_Routine : SYSTEM.ADDRESS);

entry Unregister_Event (
   Artist_ID : CSL.Artist_ID_Type;
   Event_Kind : Client_Events.Client_Event_Kind);
```

The Arist_ID parameter should take the value of the *Local_Artist_ID* which is declared globally in every artist and initialized within the *Start_Artist* accept call.

Initial client event registration should be placed within the body of the *Start_Artist* accept call. This will ensure that the artist will not miss any events that the application may otherwise generate before its full initialization. This is because the *Start_Artist* entry is called for each artist specified in the client startup configuration before the application is allowed to begin execution.

Client event registration is performed on lines 139-148 of our example artist. The call to register for a push event looks as follows:

```
Client_Dispatcher.Dispatcher.Register_Event
   (Local_Artist_ID, Client_Events.Stacks_Push,
    Handle_Stacks_Push'ADDRESS);
```

7.4 Creating graphical objects

Graphical objects may be created and initialized by making calls into the ADL hierarchy (discussed in section 4). An interface to this object-oriented graphics library is provided

via the Lo-CAL *~Apply* call (see section 6). Object creation is performed on lines 155-258. The call to create the artist base frame is given below.

```
artist_frame := ~Apply(ADL_base_frame,
                       create,
                       frame_label => "Dialogue View",
                       x           => 20,
                       y           => 20,
                       width       => 300,
                       height      => 275,
                       foreground  => WHITE,
                       background  => FOREST_GREEN,
                       show_footer => true);
```

Here we create an instance of a base frame using the *create* method for the *ADL_base_frame* class. The *~Apply* call uses Ada syntax to specify the arguments of a method. It supports both positional and named notation. Many parameters have default values and need not be explicitly initialized. For a list of all ADL methods, along with their parameters and defaults, refer to the *ADL Reference Manual* included in appendix E.

Lines 260-273 of our artist example make a series of calls to *ADL_window_fit* methods. This method makes it easier to size panels and windows. For example,

```
~Apply(ADL_panel,
       pop_up_panel,
       ADL_window_fit);

~Apply(ADL_popup_frame,
       pop_up_frame,
       ADL_window_fit);
```

resizes the panel in the popup window so that it fits evenly around its panel objects, then resizes the popup window so that it fits evenly around the resized panel.

7.5 Setting object behaviors

In this section we must specify how ADL objects should react to various server events (see section 6.3 for more detail). A behavior array is used to specify a callback routine address for each server event. If no address is specified the for a particular server event, that event will not be reported to the artist.

Once a behavior array is defined, it is bound to an ADL class or an ADL instance using the Lo-CAL *~Set_Behavior* command, passing it the class/instance and the behavior array. An instance behavior will override a class behavior.

In our example, object behaviors are set on lines 281-287,

```
behaviors(Select_Event) := Handle_Select'ADDRESS;

~Set_Behavior(ADL_button,
              behaviors);

~Set_Behavior(ADL_num_fld,
              behaviors);
```

We define a behavior for only *select* events and then assign that behavior to the classes ADL_button or ADL_num_fld. Now all instances of these classes will receive notice of *select* events and will invoke the same handling procedure.

7.6 Calling the start processing method

After an artist has created and initialized its graphical objects and set behaviors on those objects, it must call the *start_processing* (a method defined in the ADL_base_frame class) on its base frame. This will cause the server to display the artist and begin listening for server events. *Note that a base frame will not appear until the* start_processing *method is called.* The call to *start_processing* is on lines 293-295 of our example artist:

```
~Apply(ADL_base_frame,
       artist_frame,
       start_processing);
```

7.7 Terminating artists

When the *Artist_Manager* shuts down an artist, it will invoke the *Terminate_Artist* entry of the artist before aborting/deallocating it. In order to achieve smooth shutdown, the artist writer should provide an accept body to the *Terminate_Artist* accept statments that performs the following actions:

- unregister for any client events
- destroy all of the artist's graphical objects

Failure to unregister for client events could cause a tasking error if the dispatcher attempts to notify an aborted artist. If the graphical objects are not destroyed, they will remain visible even after the artist has terminated. Usually, destroying the base frame is sufficient.

In our dialogue artist example, the *Terminate_Artist* accept call is defined on lines 310-320. This code is given in figure 25. If it is difficult to determine which client events an artist is currently registered for, the artist can simply loop through all client event kinds as we do in this example. Unregistering for an event that was never registered with the dispatcher will have no ill-effect.

```
accept Terminate_Artist do
  --unregister for all client events
  for Events in Client_Event_Kind loop
    Client_Dispatcher.Dispatcher.Unregister_Event (
      Local_Artist_ID, Events);
  end loop;
  ~Apply(ADL_base_frame,
       artist_frame,
       destroy,
       artist_frame);
end Terminate_Artist;
```

Figure 25: Termination code for example artist

7.8 Associating graphical objects with ADT objects

Artists may need to associate ADT objects with their graphical representations. For example, if the server detects an event on a rectangle representing a node in a graph, the artist will need a handle to the corresponding graph ADT node in order to handle the event appropriately.

Chiron provides an *association table* generic that may be used to maintain a bi-directional relation between graphical objects and ADT objects. An entry in this table contains three elements: the ADT object, the ADL object, and any context information required by the artist. These three types are provided by the artist as generic parameters in the package instantiation. Lookup functions are provided to retrieve an entry given either the ADT or ADL object. The generic package *association_tables* is located in

`.../chiron.1.4/src/client/tables_spec.a`

A listing of the package specification is found in appendix D.

7.9 Avoiding deadlock

In order to ensure that all artists depict a consistent view of their ADT(s), whenever an ADT write-mode operation is performed, *Chiron* blocks any subsequent write-mode operations on the ADT until all artists have updated their depictions (completed executing their callback routines). This can lead to a deadlock situation if a callback routine for a write-mode event attempts to invoke a new write-mode ADT operation. The callback will block on the 2nd write-mode operation, thereby disabling its completion. The lock on the ADT for the 1st write-mode operation will never be released and the system will deadlock.

Therefore it is critical that artist writers never make calls to ADT write-mode operations from within callbacks which themselves handle write-mode events. Write mode operations can invoked by callbacks for read-mode events, and they

are always safe from server event handlers. Read-mode operations are always safe in any context.

If this proves to be too restrictive, artist writers can get around the consistency locks by declaring all ADT operations to be read-mode in the client event definition (see section 5.2) and by implementing their own access control on the ADT if necessary.

8 Building Chiron clients

This section describes how to generate a *Chiron* client using the client toolset. First we describe the *client configuration file*, in which the artist writer describes the structure of the client. We then describe the client generator tools and their usage. And finally, we give instructions on how to compile, load, and execute *Chiron* clients.

8.1 Client configuration

Users configure their clients by creating a *client configuration file* (ccf). The information in this file is used by the generator tools to create appropriate client components. It is also read during client initialization in order to determine the runtime configuration of a client. The runtime configuration defines which artists and how many instances of these artist should be invoked initially as well as which display they should use. An example of a client configuration file for the configuration of a flight simulator is given in figure 26.

A client configuration file contains the following information:

ADT specification list: A list of the filenames, each on its own line, of all ADT specifications for which client events should be generated. Each ADT specification filename can optionally be followed by the word dispatcher. If present, a dedicated dispatcher will be generated for that ADT (see section 2.3).

Artist descriptions: This portion of the configuration file describes each artist type in terms of its name and any ADTs for which it will want to receive client events. Each line contains an artist name followed by a list of ADT specification filenames. Each filename given must also appear in the ADT specification list.

Runtime configuration: Each line contains an artist name followed by the number of instances of that artist that should be invoked at client initialization, optionally followed by which display should be used. If no display is specified, unix:0 is the default. Each artist name given must appear on the artist description list.

8.2 Generating client components

The *Chiron 1.4* toolset includes a set of tools for the automatic generation of several client components. This set includes:

- Artist Manager generator
- Client Events generator
- Wrapper generator
- Dispatcher generator
- Artist template generator
- Client Initializer generator

```
1

        aileron_spec.a
        altitude_spec.a
        attitude_spec.a                 dispatcher
        psi_spec.a                      dispatcher
        speed_spec.a
        tail_spec.a                     dispatcher
        theta_spec.a                    dispatcher
        throttle_spec.a


        Aileron_Module_Artist           aileron_spec.a
        Tail_Module_Artist              tail_spec.a
        Throttle_Module_Artist          throttle_spec.a

        Pitch_Artist                    attitude_spec.a
        Roll_Artist                     theta_spec.a
        Altitude_Module_Artist          altitude_spec.a

        Altimeter_Module_Artist         altitude_spec.a
        Airspeed_Module_Artist          speed_spec.a
        Compass_Module_Artist           psi_spec.a
        Turn_Module_Artist              theta_spec.a
        Horizon_Module_Artist           attitude_spec.a theta_spec.a


        Aileron_Module_Artist           1       jasmin:0
        Tail_Module_Artist              1       jasmin:0
        Throttle_Module_Artist          1       jasmin:0

        Pitch_Artist                    0       jasmin:0
        Roll_Artist                     0       jasmin:0
        Altitude_Module_Artist          0       jasmin:0

        Altimeter_Module_Artist         1       jasmin:0
        Airspeed_Module_Artist          1       jasmin:0
        Compass_Module_Artist           1       jasmin:0
        Turn_Module_Artist              1       jasmin:0
        Horizon_Module_Artist           1       jasmin:0
```

Figure 26: Example client configuration file

With so many generated components, client construction can be complicated and confusing. To address this, we provide an all-encompassing generator tool called the *client_builder* which reads the client configuration file and invokes the generator tools in order to construct all of the initial components of the specified client.

Artist manager generator

```
usage: artist_manager_generator <file.ccf>
```

The artist manager generator, given a client configuration file, creates an artist manager (see section 2.3) specification and body in two files *artist_manager_spec.a* and *artist_manager_body.a* respectively. The artist descriptions portion of the ccf file is used to determine the set of artists that the artist manager can invoke.

Client initializer generator

```
usage: client_init_generator <file.ccf>
```

Given a client configuration file, the client initializer generator generates a client initialization package (see section 2.3) into the file *client_init.a*. The generated package, *Client_Init*, contains code to initialize the client during its own elaboration and must be *with*'ed by the application main program. The generated initialization code will, by default, bring up the configuration specified in the runtime configuration portion of the ccf file at the time the client initializer was generated. However, it will first try to locate the ccf file and configure the client dynamically based on it's current contents before resorting to the default configuration.

Client events generator

```
usage: client_events_generator <file.ccf> <[unconstrained_types_file]>
```

The client events generator, given a client configuration file, generates a client event type definition (see section 5.2) that is based on the ADT specification list into the file *client_events.a*. All ADT specification files that are listed in the ccf must be present in the directory from which the client event generator is executed.

Because the fields of a client event record copy the type marks of the corresponding ADT operation parameters, unconstrained parameter types present a problem. This is due to the fact that Ada does not allow an unconstrained type as a record field type mark. The client generators provide an automated solution to this problem by allowing the user to specify an optional file that contains a list of all unconstrained type names. The client event generator will create a new access type for each unconstrained type and used the access type in place of the unconstrained type within the client event record definition. Note that if there are unconstrained types, the wrapper generator must be passed the unconstrained types file as well so that both packages can handle any access types consistently.

The *Client_Event_Mode* array **must** be edited to indicate the Read/Write permissions of ADT operations before the client is compiled (see section 5.2.) Failure to do so could result in deadlock.

Wrapper generator

```
usage: wrapper_generator <adt_specification_file> <[unconstrained_types_file]>
```

Given an ADT specification file name, the wrapper generator generates a wrapper and an access controller for that ADT. Given a file that contains the ADT, <adt_name>, the wrapper generator produces three files: *wrapper_<adt_name>_spec.a*, *wrapper_<adt_name>_body.a*, and *<adt_name>_controller.a*. The given ADT specification file must be present in the directory from which the wrapper generator is invoked.

Like the client event generator, the wrapper generator must also be able to handle unconstrained types. If the client event type has generated special access types in place of unconstrained types, the wrapper must be able to allocate and initialize these new access types when creating event instances. This special code is automatically included if the unconstrained types file is given as a second argument to the wrapper generator. All wrappers in a client should be generated with the same unconstrained types file that was provided to the client events generator.

Dispatcher generator

```
usage: dispatcher_generator <file.ccf>
```

Given a client configuration file, the dispatcher generator generates the dispatching architecture as described in the ADT specification list portion of the ccf file. The Client Dispatcher is always generated to the files *client_dispatcher_spec.a* and *client_dispatcher_body.a*. Any dedicated ADT dispatchers are generated to files named *dispatcher_<adt_name>_spec.a* and *dispatcher_<adt_name>_body.a*. The specified ADT files must be present in the directory from which the dispatcher generator is invoked.

Artist template generator

```
usage: artist_template_generator <artist_name> [<adt_specification_file> ...]
```

Given an artist name followed by an optional list of ADT specification filenames, the artist template generator generates an artist template (see section 3) specification and body into the files *<artist_name>_spec.a* and *<artist_name>.cal*. Any ADT specification files listed on the command line must be present in the directory from which the artist template generator is executed.

Client builder

```
usage: client_builder [-ci] [-am] [-ce] [-w] [-d] [-at] \
                      [-types <unconstrained_types_file>] <file.ccf>
   -ci :  generate the client inializer
   -am :  generate the artist manager
   -ce :  generate the client events
   -w  :  generate wrappers for each adt
   -d  :  generate dispatchers
   -at :  generate artist templates
   default :  generate all of the above
   -types unconstrained_types_file :
          specify file containing information that the client events
          generator and the wrapper generator can use to handle
          unconstrained types correctly.  Only valid with -ce, -w
          or default options.
```

The *client_builder* tool is provided to help users generate all of a client's initial components. The artist description list is used to generate an artist manager, the runtime configuration is used to generate a client initializer, and the ADT specification list is used to generate a client event type and dispatchers. A wrapper and access controller are generated for each ADT in the ADT specification list, and an artist template is generated for each artist in the artist description list. Several command line options are provided to allow the user to invoke any subset of client generator tools, but the default is to generate

all components. The output for the *client_builder*, given the ccf file in figure 26 and no command line options, is listed in appendix D.1.

8.3 Translating artists

```
usage: lcc <file.cal>
```

Once the artist has been written, it needs to be run through the Lo-CAL compiler (lcc) in order to convert it into legal Ada. The Lo-CAL processor translates Lo-CAL calls (like *~Apply*) into an Ada acceptable form, and in doing so, generates calls to the Client Protocol Manager to ship the call to the server. The code produced is fairly low-level; Lo-CAL hides the communication protocol between client and server from the user interface developer. Only the artist body (or .cal file) needs to be processed by lcc.

The Lo-CAL processor has a special -S flag for *silent* mode. This will cause the Lo-CAL code to be translated without any additional code insertions. This enables artists to use Ada separate compilation for very large artists.

8.4 Compiling and loading clients

Once all necessary components have been generated, the client events file has been edited, and all artists have been processed by lcc, the order of compilation is as follows:

```
ADTs
client_events.a  (edited)
artist_manager_spec.a
controllers
adt dispatchers
client_dispatcher
wrappers
artists  (translated by lcc)
artist_manager_body.a
client_init.a
application main program
```

In order to compile, set the Ada path to search in the following Ada libraries [3]:

```
$SRC_PATH/utilities
$SRC_PATH/types
$SRC_PATH/client
$Q_PATH
$ADA_PATH/vads_exec
```

To load the client:

[3] "$SRC_PATH", "$Q_PATH", "$Q_LIB", and "$ADA_PATH" are used throughout this section to indicate that the full path is dependent upon where the *Chiron* sources, *Q* Ada library, *Q* c library, and *SunAda* compiler are installed in your environment). These values should be defined in `.../chiron.1.4/src/Makefile.config`.

```
a.ld <main_procedure_name>
   -o <executable_name> $Q_LIB
```

Adamakegen[4] can be used to generate Ada makefiles for artists. A couple of rules can be added to handle invoking lcc and loading the client. We suggest copying an existing makefile from the chiron installation applications and editing the main procedure and executable name and the path to the Makefile.config file. Then run Adamakegen on your makefile to generate dependencies for your own client. Figure 27 shows a sample makefile for an artist without the Adamakegen generated dependencies. Note that the Makefile.config file is included, ensuring that all paths and commands are consistent with the *Chiron* installation. The default Adamakegen commands have been modified to to use the *Chiron* defaults.

8.5 Executing clients

To run the client, it's important that the necessary environment variables are set before executing the application:

```
for the server:
   CHIRON_LIB_PATH  .../chiron.1.4/standard_library
   CHIRON_PID       .../chiron.1.4/pid/pid
   CHIRON_SERVER    (set to machine where server will be executing)
   CHIRON_LIB       (either "xview" or "motif")
for the client:
   CHIRON_LIB_PATH  .../chiron.1.4/standard_library
   CHIRON_PID       .../chiron.1.4/pid/pid
   CHIRON_SERVER    (set to machine where server will be executing)
```

An optional environment variable, `CHIRON_SERVER_NUMBER`, may be set that allows more than one server to run on a given machine. The default number assigned is 1. For more than one server, set this variable to the same integer value on *both* the client and server machine. If you have chosen a server number already in use, the server will issue a warning message and then terminate.

The pid directory contains a file called pid that is used for providing persistent identifiers for *Chiron*. Sometimes, due to an abnormal termination, a pid.LOCK file may be left in that directory. If this happens, it should be removed, or it could cause the system to hang. You may also create your own pid directory and use it in place of the pid directory in the chiron installation.

To execute the client:

- start chiron_server script (in bin directory)
- wait until you see the message "loaded libraries"
- run the application executable

[4]The *Adamakegen* utility generates a Makefile based on the dependencies of a set of Ada source files. It can be obtained via anonymous ftp. Contact omalley@ics.uci.edu for more information.

```
# Chiron Macros
include <chiron src path>/Makefile.config

# Ada Make Gen command
ADAMAKEGEN = $(CHIRON_ADAMAKEGEN)
# Additional commands to execute during make depend
DEPENDCMDS =
# Compilation command
# Test to determine whether a compile was successful
ERRORTEST1 = test ! -s
ERRORTEST2 =
# Library cleaning command
CLEANLIB = $(ADA_CLEANLIB)
# Library removing command
RMLIB = $(ADA_RMLIB)
# Library creation command
MAKELIB = $(ADA_MKLIB) . $(ADA_PATH)/standard; $(ADA_REMOVE_PATH) $(ADA_PATH)/standard
# Library path extension command
ADDLIB = $(ADA_ADD_PATH)
# Library path extension command
FIXLIB = sed 's# Object directory name
OBJDIR = .obj


.SUFFIXES: .cal .a
.cal.a: ; $(BIN_DIR)/lcc $<


# DO NOT DELETE THIS LINE --Ada Make Gen commands are below this line.
# DO NOT DELETE THIS LINE --Place user commands below this line.


all: load

install: ada.lib all

load: <executable name>

<executable name>: $(ROOTS)
        $(ADA_LINK) <main procedure name> -o <executable name> $(Q_LIB)

ada.lib:
        make -f Makefile.<name> adalib
adalib:
        $(RM) .vadsrc
        $(SYM_LINK) $(SRC_PATH)/.vadsrc
        $(ADA_MKLIB)
        $(ADA_ADD_PATH) $(SRC_PATH)/utilities
        $(ADA_ADD_PATH) $(SRC_PATH)/types
        $(ADA_ADD_PATH) $(SRC_PATH)/client
        $(ADA_ADD_PATH) $(Q_PATH)
        $(ADA_ADD_PATH) $(ADA_PATH)/vads_exec
        $(FIXLIB)

uninstall:
        make -f Makefile.<name> veryclean
        $(RM) <executable_name>
        $(RM) <translated artist(s)>

makedepend:
         $(ADAMAKEGEN) -fMakefile.colors -NADA_PATH=$(ADA_PATH) -NSRC_PATH=$(SRC_PATH)
-NQ_PATH=$(Q_PATH) -r .
```

Figure 27: Sample Adamakegen makefile for a chiron client

To shutdown the client/server:

- control-c in application window
- control-c in chiron_server window

Chiron script. This installation provides a script called *chiron* in its bin directory that can simplify the process of executing a *Chiron* client. It takes the client binary as its argument. This script will execute both the client and the server, set all necessary environment variables, check for pid locks, and kill both processes on a <cntrl-c>.

9 Integrating with DevGuide

To aid in the development of artist writing, *Chiron* allows the use of *Sun Microsystem's Developer's Guide (DevGuide)*. The *DevGuide* application is fairly intuitive to use due to its its use of direct manipulation (drop and drag) layout of graphical objects and property sheet windows. Almost all of *DevGuide's* objects and connections can be mapped easily into *Chiron* Lo-CAL statements with the few exceptions mentioned below. The use of DevGuide, its connections manager, and the format of its Graphical Intermediate Language (GIL) are outside the scope of this manual and are dealt with in the *OpenWindows Developer's Guide 3.0 User's Manual* [Sun91]. This section describes how to use the (dev)Guide Artist Builder tools, *Gil2Local* and *GAB*, in order to translate a graphical user interface design built using *DevGuide* into a *Chiron* client.

9.1 Guide Artist Builder tools

Gil2LoCAL

```
Usage: gil2local file.G [-d decls][-a applys][-b behaves]
```

Gil2LoCAL is the command used to do the actual translation from GIL into *Chiron* Lo-CAL calls. These calls are split into three separate files: a declarations file, an applys file, and a behaviors file, all of which can be specified on the command line. The default file names are "decls.cal", "applys.cal", and "behaves.cal". The files that are generated by *gil2local* are described below and the code examples that are given are from a simple interface consisting of a base frame with a callback defined for the "resize" server event:

- **decls.cal:** Lo-CAL declarations. This file will contain the Lo-CAL object declarations to be included in the declarations section of an artist. Object variable declarations as well as Server Event callback functions declarations are generated.

```
        -- This section contains the Lo-CAL object
        -- *declarations* to be used in conjunction with
        -- the artist_template_generator to fill in the
        -- Declarations Section of an artist.

    WINDOW1 : ~CSL.ADL_base_frame;

    Auto_Set_Behavior  : ~Behavior_Array_Type := (others => System.No_Addr);

        -- End of generated declarations

        -- This section contains the declarations
        -- of the callback procedures specified by
        -- user via the graphical front-end.
```

```
procedure WINDOW1_CALLBACK (object : CSL.object_type;
                event : CSL.chiron_event_ptr) is
  begin
    null;
  end WINDOW1_CALLBACK;

  – End of generated callbacks
```

- **applys.cal:** Lo-CAL Apply statements. This file will contain the object creation calls assigned to the appropriate objects.

```
  – This section contains the Lo-CAL ~Apply
  – calls to be used in conjunction with the
  – artist_template_generator to fill in the
  – Object Creations section of an artist.

WINDOW1 := ~Apply( ADL_base_frame, create,
        frame_label => "Base Window",
        x => 20,
        y => 20,
        width => 400,
        height => 150,
        show_footer => TRUE,
        left_footer => "",
        right_footer => "" );

  – End of generated ~Apply calls
```

- **behaves.cal:** Lo-CAL Set_Behavior statements. This file will contain the object behavior mapping to the correct behavior array depending upon event, instance, and callback. All behaviors that are automatically generated, use the Auto_Set_Behavior Behavior Array, and other behaviors should declare their own Behavior Arrays. It also contains the call to the artist's start_processing method.

```
  – This section contains the Lo-CAL ~Set_Behavior
  – calls to be used in conjunction with the
  – artist_template_generator to fill in the
  – Behaviors section of an artist.

 Auto_Set_Behavior := (others => WINDOW1_CALLBACK'ADDRESS);
~Set_Behavior( WINDOW1, Auto_Set_Behavior );
```

~Apply(ADL_base_frame, WINDOW1, start_processing);

– End of generated Behavior calls

GAB

```
Usage: gab [-d decls][-a applys][-b behaves] artist_file.cal
```

Once the *Gil2LoCAL* translation has been done, the code can then either be manually or automatically inserted into an artist. If you are creating a new artist, you can simply use the *GAB* application to insert the code into a blank artist template. If significant modifications have been made to your artist, or you are writing over changes made from a previous code generation, then the Lo-CAL file should be included into the artist manually. Once the Lo-CAL file have been included this is usually enough to compile and bring it up using *Chiron*.

9.2 Limitations

The Guide Artist Builder has several limitations.

1. ADL application (canvas) objects aren't supported by DevGuide, so they aren't generated. Popup menu's can only be attached to canvases; pulldown menus can only be attached to buttons.

2. There are layout differences and default font and size differences between the same client running with the Motif Chiron server, the XView Chiron server, and the DevGuide display mainly because of default font differences.

3. *Gil2Local* does not generate declarations for scrolling lists as the are a recent addition to *Chiron's* ADL and are not supported in the Xview server.

4. DevGuide only supports Sun format 'glyphs' for use with buttons and menus. *Chiron* uses Xbitmaps.

5. Most connections are supported with the exception of fine-grain graphical events such as "enter" or "leave" which are handled at the *Chiron* server level. Also "get_label" calls are stored in a Get_Str variable since no method for specifying where to get it are supplied.

6. *Gil2Local* only generates callback procedures for object instances, while in *Chiron*, object classes can be set to all use the same handler procedure. Also, in most cases, *Gil2LoCAL* does not discriminate between event types, so for an object that can receive different events you must explicitly declare and set its own behavior array.

7. Because *Chiron's* ADL is a super-set of the graphical objects supported, canvas objects can only be generated using DevGuide through calling the "Execute Code..." object connection as the result of an event. Some artist editing may still be required.

For specific installation hints and requirements, please refer to the source code files provided with the release which include several hints, examples, and requirements.

10 Known Defects

The current implementation of *Chiron* has several defects we know about, and probably more that we don't know about. This list of known defects serves two purposes: first, as a user of *Chiron*, you can consult the list to find out if a problem you are having is related to a known bug or shortcoming, and if so whether there is a known work-around. Second, this list serves as the basis of a "to do" list for the *Chiron* implementors.

When you identify a shortcoming which is not on this list, you should send a note to the developers describing the problem. Send bug reports to `arcadia-chiron-bugs@ics.uci.edu`. Send criticisms, suggestions, and contributions for general discussion, to `arcadia-chiron@ics.uci.edu`. Please include an indication of the relative importance of the problem so we can prioritize our "to do" list. Also, send a work-around if you have one, and any other information which you think may be useful for the developers or others encountering related problems. If an error message is printed, please send the entire text of the message. We will redistribute messages and responses to those who express interest.

The list is divided into the following categories, which are independent of priority:

Unimplemented Features: Features of the design described in *Chiron: Concept and Design,* [BCJ+89] which are not yet part of the implementation.

Desired Enhancements: Features which are not part of the current design, but are consistent with it.

Caveats: Warnings about implemented features that have not yet been thoroughly tested, or are known to have problems.

10.1 Unimplemented Features

Dynamism: *Chiron* currently does not support the dynamism characterized in the design document. Although adding artist instances dynamically is supported, and adding completely new artists (defined in a separate process) is not.

Extending Hierarchy: Adding to the hierarchy is currently not encouraged due to the fact that the server is currently not as dynamic as we would like. When introducing a new class object, the various types need to be defined on the Ada side. Additionally, the class needs to be entered into the symbol table and the instruction/event interpreter needs to include the various new methods into its call table.

Abstract Depiction Language: Originally we had intended to define our own graphics language to be interpreted (Doodle) [BCJ+89]. For the sake of time, we opted to use existing software (C++ on top of XView), which impacted the original design.

Persistence: *Chiron-1* does not support persistence. This has been deferred to the *Chiron-2* design.

10.2 Desired Enhancements

Canvas Redraw Algorithm: Currently the canvas is refreshed by redrawing all items within the canvas. We need to implement a smart redraw algorithm to cut down on the number of objects redrawn. The redraw occurs offscreen, however, to minimize flicker.

Bounding Boxes: Currently bounding boxes are used as a crude estimation of the region of an application object. We would prefer that each class contained it's own method for determining it's region.

Multiple ADT Instances: The current Chiron design does not handle the binding of multiple ADT instances to multiple artist well. The dispatching mechanism does not filter by instance and there is no explicit instance registration. Currently events must be filtered from within each artist when a client contains more than one ADT instance.

10.3 Caveats

Ellipses can't be Rotated: Due to a bug in the X11 library (which we use to render graphics), ellipse objects can not be rotated.

Cursors Limited (XView only): Currently, cursors can only be defined for panel and canvas subwindows. This we see as a bug in XView – we feel that you should be able to define cursors for base frames, popup frames, text windows and tty windows as well.

Server Image: Creation of an **ADL_server_image** class instance using Sun's bitmap format (specified as an array of shorts) currently does not work. However, the other two means of creating a server image (create_X_format() and create_by_file()) should work.

Files and Bitmaps: Remember when specifying paths to files and bitmaps that these paths will be interpreted relative to the server – not the application.

Motif Menu Warning: Applications containing a pulldown menu will get the following warning under the motif server: "Attempting to manage an incomplete menu.". This warning can be ignored.

Text Limit: ADL text types are limited to 100 characters. This includes text, text fields and messages. This effects any parameter or return value of an ADL apply call. This can be worked around by using a text_window that loads/stores text from/to a file.

Motif Auto Layout: Automatic layout under the Motif server is somewhat buggy. In particular, the horizontal layout on a panel does not support wrap-around. To ensure that xview and motif clients look consistent, absolute coordinates should be specified.

11 Troubleshooting Chiron

This sections is provided to give assistance in troubleshooting problems with the execution of *Chiron* applications.

The client registers with the server then hangs.

- Make sure your DISPLAY is set correctly in the server window.
- Make sure that you are calling the *start_processing* method on your base_frame.
- Make sure that the CHIRON_PID path is set correctly.
- Make sure that there is not a pid.LOCK file in the pid directory.
- Make sure that you have permission to read/write the pid file in the pid directory.

The server gives an *fopen* error.

- Make sure that the CHIRON_LIB_PATH path is set correctly in the server window.

The client does not register with the server.

- Make sure that the CHIRON_SERVER environment variable is set correctly in the client window.
- Make sure the CHIRON_LIB_PATH environment variable is set correctly in the server window.

The client raises a LINK_FAILURE.

- Make sure that the CHIRON_PID path is set correctly.
- Make sure that you have permission to read/write the pid file in the pid directory.
- Make sure that the CHIRON_SERVER environment variable is set correctly in the client window.
- Make sure the CHIRON_LIB_PATH environment variable is set correctly in the server window.

The client raises a TASKING_ERROR.

- Make sure that the CHIRON_PID path is set correctly.

The server gives a table key not found message. This means that the id numbers generated into your artist.a file by lcc are not consistent with your installed ADL library.

- Make sure that you are running a version of the server that is compatible with your application. Most important, make sure that the version of lcc you are using reads the same ADL library as the server.
- The ADL library may have been re-installed. Try re-lcc'ing your .cal file(s).

A canvas application is running painfully slow.

- Check the size of your canvas paint region. If it is very large, it could be causing the window manager to swap memory when it redraws the canvas.

- This problem is exacerbated when using a color monitor since they require more bits per pixel than monochrome monitor.

Your artist initializes, but then appears to deadlock.

- See section 7.9 for information on how to avoid client deadlock.

The server gives a server number already in use message.

There is already a chiron server running on your machine with the same id number. Note that if the CHIRON_SERVER_NUMBER environment variable is not set, a default of 1 is assigned for both the server and the client.

- Change the CHIRON_SERVER_NUMBER environement varable to the same number in both the server and client windows.

References

[BCJ+89] Gregory Alan Bolcer, Mary Cameron, M. Gregory James, Rudolf K. Keller, Richard N. Taylor, and Dennis B. Troup. Chiron-1: Concept and design. Arcadia Technical Report UCI-89-12, University of California, Irvine, October 1989. (Revised, January 19, 1990).

[For93] Kari Forester. Chiron 1.4 client design. Arcadia Technical Report UCI-93-02, University of California, March 1993. (Revised July 27, 1993).

[Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, June 1977.

[KCTT91] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. User interface development and software environments: The Chiron-1 system. In *Proceedings of the Thirteenth International Conference on Software Engineering*, pages 208–218, Austin, TX, May 1991.

[Mye89] Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[Sun91] SunSoft, 2550 Garcia Avenue Mountain View, CA 94043. *OpenWindows Developer's Guide 3.0 User's Guide*, 1991.

[TJ93] Richard N. Taylor and Gregory F. Johnson. Separations of concerns in the Chiron-1 user interface development and management system. In *Proceedings of the Conference on Human Factors in Computing Systems*, Amsterdam, April 1993. Association for Computing Machinery.

A Simple artist example

```
                                                                    1
with Client_Dispatcher;                                             2
                                                                    3
package body Simple_Artist is                                       4
                                                                    5
  task body Simple_Artist is                                        6
                                                                    7
    Self_Ptr       : Dialogue_Artist_Ptr;                           8
    Local_Artist_ID : CSL.Artist_ID_Type;                           9
    Local_Display   : CSL.Str;                                      10
                                                                    11
    --<< declare artist objects here. >>                            12
                                                                    13
    artist_frame    :  CSL.ADL_base_frame;                          14
    artist_panel    :  CSL.ADL_panel;                               15
    quit_button     :  CSL.ADL_button;                              16
    value_field     :  CSL.ADL_text_fld;                            17
                                                                    18
    quit_button_behavior : CSL.Behavior_Array_Type :=               19
                          (others => System.No_Addr);               20
                                                                    21
    --<< declare handler routines for both client and server >>     22
    --<< events plus any auxilliary routines here.          >>     23
    --                                                             24
    --server event handlers must have the signature:               25
    --procedure <handler_name> (Object : CSL.Object_Type;          26
    --Event  : CSL.Chiron_Event_Ptr);                              27
    --                                                             28
    --client event handlers must have the signature:               29
    --procedure <handler_name> (Event : Client_Event_Ptr);         30
                                                                    31
    procedure Handle_Quit (Object : CSL.Object_Type;                32
                           Event  : CSL.Chiron_Event_Ptr) is        33
    begin                                                           34
       Apply(ADL_base_frame,                                        35
             artist_frame,                                          36
             set_show,                                              37
             false);                                                38
    end Handle_Quit;                                                39
```

```
                                                                                41
   procedure Handle_Update (Event : Client_Event_Ptr) is                        42
   begin                                                                        43
     --update depiction                                                         44
      Apply(ADL_text_fld,                                                       45
           value_field,                                                         46
           set_value,                                                           47
           to_str(integer'image (Event.Simple_Update_Update_Value)));           48
   end Handle_Update;                                                           49
                                                                                50
                                                                                51
begin --task body                                                               52
                                                                                53
   accept Start_Artist (                                                        54
           ID   : CSL.Artist_ID_Type;                                           55
           Aptr : SYSTEM.ADDRESS;                                               56
           Display_Name : CSL..Str) do                                          57
     Self_Ptr := address_to_artist(Aptr);                                       58
     Local_Artist_ID := ID;                                                     59
     Local_Display := Display_Name;                                             60
                                                                                61
     --<< register interests in client events with the >>                      62
     --<< client_dispatcher here.                    >>                       63
                                                                                64
     Client_Dispatcher.Dispatcher.Register_Event                                65
       (Local_Artist_ID, Client_Events.Simple_Update_Value,                     66
        Handle_Update'ADDRESS);                                                 67
                                                                                68
   end Start_Artist;                                                            69
                                                                                70
   --<< create initial graphical objects here. >>                              71
                                                                                72
   artist_frame :=  Apply(ADL_base_frame,                                       73
                    create,                                                     74
                    frame_label    => "Simple Artist",                          75
                    x              => 200,                                      76
                    y              => 200,                                      77
                    width           => 200,                                     78
                    height         => 200,                                      79
                    foreground      => BLACK,                                   80
                    background      => WHITE);                                  81
                                                                                82
```

```
artist_panel :=  Apply(ADL_panel,                                    84
                   create,                                           85
                   parent        => artist_frame,                    86
                   width         => 200,                             87
                   height        => 200);                            88
                                                                     89
quit_button :=  Apply(ADL_button,                                    90
                   create,                                           91
                   parent         => artist_panel,                   92
                   button_id      => 1,                              93
                   label          => "Quit",                         94
                   auto_placement  => false,                         95
                   x              => 40,                             96
                   y              => 10);                            97
                                                                     98
                                                                     99
value_field :=  Apply(ADL_text_fld,                                 100
                   create,                                          101
                   parent         => artist_panel,                  102
                   label          => "Value:  ",                    103
                   text_value     => "0",                           104
                   display_length  => 3,                            105
                   stored_length   => 3,                            106
                   read_only      => true,                          107
                   auto_placement  => false,                        108
                   x              => 20,                            109
                   y              => 40);                           110
                                                                    111
 Apply(ADL_panel,                                                   112
      artist_panel,                                                 113
      ADL_window_fit);                                              114
                                                                    115
 Apply(ADL_base_frame,                                              116
      artist_frame,                                                 117
      ADL_window_fit);                                              118
                                                                    119
--<< set behaviors of graphical objects in response >>              120
--<< to server events here.                        >>              121
                                                                    122
quit_button_behavior(Select_Event) := Handle_Quit'ADDRESS;          123
                                                                    124
 Set_Behavior(quit_button,                                          125
           quit_button_behavior);                                   126
                                                                    127
```

```
      --<< call adl start_processing method here. >>                    129
                                                                        130
       Apply(ADL_base_frame,                                            131
             artist_frame,                                              132
             start_processing);                                         133
                                                                        134
      loop                                                              135
        select                                                          136
          accept Notify_Client_Event (                                  137
                   Client_Event    : Client_Events.Client_Event_Ptr;    138
                   Handler_Routine : SYSTEM.ADDRESS);                   139
        or                                                              140
          accept Notify_Server_Event (                                  141
                   Object          : CSL.Object_Type;                   142
                   Server_Event    : CSL.Chiron_Event_Ptr;              143
                   Handler_Routine : SYSTEM.ADDRESS);                   144
        or                                                              145
          accept Terminate_Artist;                                      146
        end select;                                                     147
      end loop;                                                         148
    end Simple_Artist;                                                  149
                                                                        150
end Simple_Artist;                                                      151
```

B Chiron-1 Standard Library

Below you'll find the package *Chiron_Standard_Library* which is the only package artists need to include to access the ADL Hierarchy. The package imports the *Var_String* package which is the string package artists must use; the package defines the *Chiron-1* event kinds and event type, the seven Lo-CAL subroutines, and finally, the Ada equivalent to the C++ types defined in the ADL Hierarchy.

```
--Copyright (c) 1992 Regents of the University of California.
--All rights reserved.

-------------------------------------------------------------------
--TITLE
--Chiron Standard Library Types
--
--LANGUAGE
--Ada
--Lo-CAL (user version only)
--
--PERSONNEL
--Authors: Rudolf Keller, Dennis B. Troup
--
--OVERVIEW
--This package has to be with-ed by any Chiron artist. It consists of
--the following sections:
--
--Variable_Length_String Specification
--general definitions
--class specific object types
--Lo-CAL specification (user version only)
--
--REFERENCES
--See: Chiron-1: Concept and Design, UCI 89-12
--
-------------------------------------------------------------------

with system;
with Lo_Cal;
with Interpreter_Objects;
with System_Dep; use System_Dep;
with Var_Strings;
package Chiron_Standard_Library is

-----------------------------
--Variable_Length_String Specification  --
-----------------------------

   Index_Out_Of_Range : exception renames Var_Strings.Index_Out_Of_Range;
   String_Too_Long : exception renames Var_Strings.String_Too_Long;

   subtype Str is Var_Strings.Str;
```

```
-----------------------------------
--Return a lower case version of S:
-----------------------------------
function Lower (S : Str)
  return Str
  renames Var_Strings.Lower;


-----------------------------
--Convert a Str to a string:
-----------------------------
function To_String (Line : Str)
  return string
  renames Var_Strings.To_String;


-------------------------------------------------
--Convert a regular type string to a "str" string:
-------------------------------------------------
function To_Str (Line : string)
  return Str
  renames Var_Strings.To_Str;


---------------------------
--Return length of string:
---------------------------
function Length (Line : Str)
  return integer
  renames Var_Strings.Length;


--------------------------------
--Assign ith postion to "char":
--------------------------------
procedure Set_Ith (Line : in out Str;
                   Pos  : integer;
                   Char : character)
  renames Var_Strings.Set_Ith;


---------------------------------
--Return ith character of string:
---------------------------------
function Get_Ith (Line : Str;
                  Pos  : integer)
  return character
  renames Var_Strings.Get_Ith;


-----------------------------------------------------------------
--Delete from Source:
--if How_Many = 0, then Delete from Start to Length (Source):
-----------------------------------------------------------------
procedure Delete (Source   : in out Str;
                  Start    : in natural;
                  How_Many : in natural := 0)
  renames Var_Strings.Delete;
```

```
-------------------------------------------------------------------
--Replace Source(Start .. Start + How_Many) with Alteration:
-------------------------------------------------------------------
procedure Replace (Source      : in out Str;
                   Alteration  : in Str;
                   Start       : in natural;
                   How_Many    : in natural := 0)
  renames Var_Strings.Replace;


procedure Replace (Source      : in out Str;
                   Alteration  : in string;
                   Start       : in natural;
                   How_Many    : in natural := 0)
  renames Var_Strings.Replace;


-----------------------------------------------
--Return the location of Fragment in Source:
-----------------------------------------------
function Location (Fragment    : in Str;
                   Source      : in string;
                   Start       : in natural := 1;
                   Occurrence  : in natural := 1)
  return natural
  renames Var_Strings.Location;


function Location (Fragment    : in string;
                   Source      : in Str;
                   Start       : in natural := 1;
                   Occurrence  : in natural := 1)
  return natural
  renames Var_Strings.Location;


function Location (Fragment    : in Str;
                   Source      : in Str;
                   Start       : in natural := 1;
                   Occurrence  : in natural := 1)
  return natural
  renames Var_Strings.Location;


function Location (Fragment    : in string;
                   Source      : in string;
                   Start       : in natural := 1;
                   Occurrence  : in natural := 1)
  return natural
  renames Var_Strings.Location;


------------------------------
--Return a substring of line:
------------------------------
function Slice (Line    : Str;
                Start   : integer;
                Finish  : integer)
  return Str
  renames Var_Strings.Slice;
```

```
---------------------------------------
--Return the concatenation of 2 strings:
---------------------------------------
function  "&" (Line1 : Str;
               Line2 : Str)
   return Str
   renames Var_Strings."&";

function  "&" (Line1 : Str;
               Line2 : string)
   return Str
   renames Var_Strings."&";

function  "&" (Line1 : string;
               Line2 : Str)
   return Str
   renames Var_Strings."&";

-----------------------------------
--Truncate length of line to "len":
-----------------------------------
procedure Truncate (Line : in out Str;
                    Len  : integer)
   renames Var_Strings.Truncate;


---------------------------------
--Return TRUE if line1 "=" line2:
---------------------------------
function Equal (Line1 : Str;
                Line2 : Str)
   return boolean
   renames Var_Strings.Equal;

function Equal (Line1 : string;
                Line2 : Str)
   return boolean
   renames Var_Strings.Equal;

function Equal (Line1 : Str;
                Line2 : string)
   return boolean
   renames Var_Strings.Equal;


---------------------------------
--Return TRUE if line1 "<" line2:
---------------------------------
function Less_Than (Line1 : Str;
                    Line2 : Str)
   return boolean
   renames Var_Strings.Less_Than;


---------------------------------
--Return TRUE if line1 "<=" line2:
---------------------------------
function Less_Than_Or_Equal (Line1 : Str;
                             Line2 : Str)
   return boolean
   renames Var_Strings.Less_Than_Or_Equal;
```

```
------------------------------------------------------------------
--general definitions
------------------------------------------------------------------


type Chiron_Event_Kind is (Menu_Event,
   Select_Event,
   Adjust_Event,
   Key_Event,
   Move_Event,
                        Resize_Event);


type Chiron_Event_Type(Kind : Chiron_Event_Kind) is
  record
    Mouse_X  : integer;
    Mouse_Y  : integer;
    Time     : integer;
    Num_Val  : integer;
    Text_Val : Str;
    case Kind is
      when Menu_Event =>
        null;
      when Select_Event =>
        null;
      when Adjust_Event =>
        null;
      when Key_Event =>
        Key_Code : character;
      when Move_Event | Resize_Event =>
        Dest_X : integer;
        Dest_Y : integer;
    end case;
  end record;

type Chiron_Event_Ptr is access Chiron_Event_Type;

------------------------------------------------------------------

package The_Lo_Cal is new Lo_Cal(Chiron_Event_Kind);

subtype Artist_Id_Type is The_Lo_Cal.Artist_Id_Type;

Null_Artist_Id : Artist_Id_Type renames The_Lo_Cal.Null_Artist_Id;

subtype Client_Id_Type is The_Lo_Cal.Client_Id_Type;

Null_Client_Id : Client_Id_Type renames The_Lo_Cal.Null_Client_Id;

------------------------------------------------------------
--Behavior_Proc: simulation of procedure pointers. The procedures pointed to
--are assumed to have the signature (Chiron_Event_Ptr, Object_Type)
--(see Set_Behavior ()).

subtype Behavior_Proc is The_Lo_Cal.Behavior_Proc;
--subtype Behavior_Proc is system.address

subtype Behavior_Array_Type is The_Lo_Cal.Behavior_Array_Type;
--type Behavior_Array_Type is array(Chiron_Event_Kind) of Behavior_Proc
```

```
------------------------------------------------------------------

type Interpreter_Data is
  record
    Artist : The_Lo_Cal.Artist_Id_Type;
  end record;

package Interpreter_Object_Type is
  new Interpreter_Objects(Interpreter_Data, (Artist => Null_Artist_Id));

subtype Library_Id_Type is Interpreter_Object_Type.Library_Id_Type;
subtype Class_Id_Type is Interpreter_Object_Type.Class_Id_Type;

subtype Instance_Type is Interpreter_Object_Type.Instance_Type;

subtype Object_Type is Interpreter_Object_Type.Object_Type;
subtype Object_Ptr is Interpreter_Object_Type.Object_Ptr;

--Useful constants

Null_Library  : Library_Id_Type
  renames Interpreter_Object_Type.Null_Library;

Null_Class    : Class_Id_Type
  renames Interpreter_Object_Type.Null_Class;

Null_Instance : Instance_Type
  renames Interpreter_Object_Type.Null_Instance;

Null_Object   : Object_Type
  renames Interpreter_Object_Type.Null_Object;

--Useful functions

function Form_Object(Library  : in Library_Id_Type;
                     Class    : in Class_Id_Type)
  return Object_Type renames Interpreter_Object_Type.Form_Object;

function Form_Object(Library  : in integer;
                     Class    : in integer)
  return Object_Type renames Interpreter_Object_Type.Form_Object;

------------------------------------------------------------------
--class specific object types
------------------------------------------------------------------
--In order to make the types of the Chiron Standard Library libraries
--accessible to artist programs, equivalent types in the Ada language
--are needed.
--This section provides the Ada definitions of all the types available
--in the Chiron Standard Library. This section has to be updated if the
--library changes.
------------------------------------------------------------------

  subtype ~ADL_application is Object_Type;
  subtype ~ADL_base_frame is Object_Type;
  subtype ~ADL_bitmap is Object_Type;
  subtype ~ADL_button is Object_Type;
  subtype ~ADL_canvas is Object_Type;
  subtype ~ADL_choice is Object_Type;
  subtype ~ADL_circle is Object_Type;
  subtype ~ADL_cmap is Object_Type;
  subtype ~ADL_compose is Object_Type;
```

```
  subtype ~ADL_cursor is Object_Type;
  subtype ~ADL_drawable is Object_Type;
  subtype ~ADL_ellipse is Object_Type;
  subtype ~ADL_font is Object_Type;
  subtype ~ADL_frame is Object_Type;
  subtype ~ADL_gif is Object_Type;
  subtype ~ADL_icon is Object_Type;
  subtype ~ADL_list is Object_Type;
  subtype ~ADL_menu is Object_Type;
  subtype ~ADL_message is Object_Type;
  subtype ~ADL_notice is Object_Type;
  subtype ~ADL_num_fld is Object_Type;
  subtype ~ADL_object is Object_Type;
  subtype ~ADL_panel is Object_Type;
  subtype ~ADL_panel_item is Object_Type;
  subtype ~ADL_polygon is Object_Type;
  subtype ~ADL_polyline is Object_Type;
  subtype ~ADL_popup_frame is Object_Type;
  subtype ~ADL_rectangle is Object_Type;
  subtype ~ADL_scrollbar is Object_Type;
  subtype ~ADL_server_image is Object_Type;
  subtype ~ADL_slider is Object_Type;
  subtype ~ADL_spline is Object_Type;
  subtype ~ADL_spring is Object_Type;
  subtype ~ADL_square is Object_Type;
  subtype ~ADL_text is Object_Type;
  subtype ~ADL_text_fld is Object_Type;
  subtype ~ADL_text_window is Object_Type;
  subtype ~ADL_tty is Object_Type;
  subtype ~ADL_window is Object_Type;
  subtype ~Corner_List is Object_Type;
  subtype ~Object_List is Object_Type;

-----------------------------------------------------------------
--adl_application.h
-----------------------------------------------------------------

  type Line_Style is (Line_Solid, Line_Double_Dashed, Line_Dashed);

-----------------------------------------------------------------
--adl_color.h
-----------------------------------------------------------------

  type ADL_Color is (
          BG0, BG1, --Private control colors: not for public use
          Clear,
          ADL_color_not_specified,
          Black, Dark_Slate_Gray,
          Dim_Gray, Gray,
          Light_Gray, White,
          Yellow, Gold,
          Khaki, Wheat,
          Tan, Goldenrod,
          Orange, Coral,
          Salmon, Red,
          Orange_Red, Indian_Red,
          Firebrick, Brown,
          Sienna, Maroon,
          Medium_Violet_Red,
          Violet_Red, Pink,
```

```
            Green, Spring_Green,
            Medium_Spring_Green,
            Green_Yellow, Pale_Green,
            Yellow_Green, Lime_Green,
            Medium_Sea_Green,
            Sea_Green, Forest_Green,
            Olive_Drab, Dark_Olive_Green,
            Dark_Green, Cadet_Blue,
            Medium_Aquamarine,
            Dark_Turquoise,
            Medium_Turquoise,
            Turquoise, Aquamarine,
            Cyan, Blue, Medium_Blue,
            Medium_Slate_Blue,
            Cornflower_Blue, Sky_Blue,
            Light_Blue, Light_Steel_Blue,
            Steel_Blue, Dark_Slate_Blue,
            Navy, Navy_Blue, Midnight_Blue,
            Dark_Orchid, Medium_Orchid,
            Orchid, Magenta,
            Violet, Plum, Thistle,
            Blue_Violet, Slate_Blue );

-----------------------------------------------------------------
--adl_cursor.h
-----------------------------------------------------------------

  Olc_Basic      : constant:= 0;
  Olc_Move       : constant:= 2;
  Olc_Copy       : constant:= 4;
  OLC_Busy       : constant:= 6;
  OLC_Stop       : constant:= 8;
  OLC_Panning    : constant:= 10;
  OLC_Navigation : constant:= 12;

  ADL_default_cursor: ADL_cursor:= Null_Object;
  --to be used as a constant

-----------------------------------------------------------------
--adl_drawable.h
-----------------------------------------------------------------

  Font_Family_Default           : constant Str :=
    To_Str("FONT_FAMILY_DEFAULT");
  Font_Family_Lucida            : constant Str := To_Str("FONT_FAMILY_LUCIDA");
  Font_Family_Roman             : constant Str := To_Str("FONT_FAMILY_ROMAN");
  Font_Family_Serif             : constant Str := To_Str("FONT_FAMILY_SERIF");
  Font_Family_Cour              : constant Str := To_Str("FONT_FAMILY_COUR");
  Font_Family_Default_Fixedwidth : constant Str :=
    To_Str("FONT_FAMILY_DEFAULT_FIXEDWIDTH");
  Font_Family_Lucida_Fixedwidth  : constant Str :=
    To_Str("FONT_FAMILY_LUCIDA_FIXEDWIDTH");


  Font_Style_Default     : constant Str := To_Str ("FONT_STYLE_DEFAULT");
  Font_Style_Normal      : constant Str := To_Str ("FONT_STYLE_NORMAL");
  Font_Style_Italic      : constant Str := To_Str ("FONT_STYLE_ITALIC");
  Font_Style_Bold        : constant Str := To_Str ("FONT_STYLE_BOLD");
  Font_Style_Bold_Italic : constant Str := To_Str ("FONT_STYLE_BOLD_ITALIC");
```

```
  Small_Font  : constant := 10;
  Medium_Font : constant := 12;
  Large_Font  : constant := 14;
  XL_Font     : constant := 19;

  ADL_default_font: ADL_font:= Null_Object;
  --to be used as a constant

-----------------------------------------------------------------
--adl_notice.h
-----------------------------------------------------------------

  Notice_Failed : constant := -1;
  Notice_Yes    : constant := 1;
  Notice_No     : constant := 0;

-----------------------------------------------------------------
--adl_panel.h
-----------------------------------------------------------------

  type Layout is (Layout_Vertical, Layout_Horizontal);

-----------------------------------------------------------------
--adl_polyline.h
-----------------------------------------------------------------

  type Arrow_Style is (Arrow_Filled, Arrow_Hollow);

  type Arrow_Location is (Arrow_None, Arrow_Start, Arrow_End, Arrow_Both);

-----------------------------------------------------------------
--adl_scrollbar.h
-----------------------------------------------------------------

  type Direction is (Vertical, Horizontal);

-----------------------------------------------------------------
--adl_server_image.h
-----------------------------------------------------------------

  type Short_Array is array(natural range <>) of Short;

  type Short_Array_Ptr is access Short_Array;

  type Character_Array is array(natural range <>) of Character;

  type Character_Array_Ptr is access Character_Array;

-----------------------------------------------------------------
--adl_utils.h
-----------------------------------------------------------------

  type Workstation_Type is (Monochrome_Workstation, Color_Workstation);
```

```
------------------------------------------------------------------
--corner_list.h
------------------------------------------------------------------

  Default_Corner_Size : constant := 10;

  type Pair is
    record
      X_Loc : integer;
      Y_Loc : integer;
    end record;

------------------------------------------------------------------
--object_list.h
------------------------------------------------------------------

  List_Size : constant := 10;


------------------------------------------------------------------
--Lo-CAL Specification
------------------------------------------------------------------
--
--The Lo-CAL software provides artist programs an interface to the
--abstract depiction language (ADL).
--Reference: Chiron-1: Concept and Design, UCI 89-12
------------------------------------------------------------------

  ----------------------------------------------------------------
  --Applies class-defined creation method to a class.
  --The specified Class must occur in the standard library.
  --
  function ~Apply (Class  : Class_Name;
   Method : Method_Name;
   [<parameters to class method>])
   return Object_Type;


  ----------------------------------------------------------------
  --Applies class-defined method to an instance of the class.  The
  --particular method is a *procedure* as opposed to a *function* (see below).
  --
  procedure ~Apply (Class    : Class_Name;
    Instance : Object_Type;
    Method   : Method_Name;
    [<parameters to class method>]);


  ----------------------------------------------------------------
  --Applies class-defined method to an instance of the class.
  --The particular method is a *function* as opposed to a *procedure*.
  --The value returned here is whatever the defined method returns.
  --
  function ~Apply (Class    : Class_Name;
   Instance : Object_Type;
   Method   : Method_Name;
   [<parameters to class method>])
   return <return type of class method>;


  ----------------------------------------------------------------
  --The Set_Behavior procedure tells the Chiron system what procedures
  --to call when server events happen to class instances. A behavior
```

```
    --is an array which is indexed by Chiron_Event_Type; the array contains
    --procedures addresses which are called when there is a object-event match.
    --Set_Behavior comes in two forms:
    --In the first form, the same behavior is defined for all instances
    --of the class given by the parameter "Class".
    --In the second form, a behavior is only defined for the class
    --instance given by the parameter "Instance".  The second form overrides
    --the first form: instance behaviors will be used before class
    --behaviors.
    ------------------------------------------------------------------
    procedure ~Set_Behavior (Class    : Class_Name;
     Behavior : Behavior_Array_Type);

    ------------------------------------------------------------------
    procedure ~Set_Behavior (Instance : Object_Type;
     Behavior : Behavior_Array_Type);

  end Chiron_Standard_Library;
```

C Stack artist example

```
                                                                                    1
with Wrapper_Stacks;                                                                2
with Client_Dispatcher;                                                             3
                                                                                    4
with TEXT_IO;                                                                       5
                                                                                    6
package body Dialogue_Artist is                                                     7
                                                                                    8
                                                                                    9
  task body Dialogue_Artist is                                                      10
                                                                                    11
    Self_Ptr       : Dialogue_Artist_Ptr;                                           12
    Local_Artist_ID : CSL.Artist_ID_Type;                                           13
    Local_Display   : CSL.Str;                                                      14
                                                                                    15
    --<< declare artist objects here. >>                                           16
                                                                                    17
    artist_frame    : ~CSL.ADL_base_frame;                                          18
    artist_panel    : ~CSL.ADL_panel;                                               19
    push_button     : ~CSL.ADL_button;                                              20
    pop_button      : ~CSL.ADL_button;                                              21
    top_field       : ~CSL.ADL_text_fld;                                            22
    depth_field     : ~CSL.ADL_text_fld;                                            23
    pop_up_frame    : ~CSL.ADL_popup_frame;                                         24
    pop_up_panel    : ~CSL.ADL_panel;                                               25
    pop_up_field    : ~CSL.ADL_num_fld;                                             26
    behaviors       : CSL.Behavior_Array_Type := (others => System.No_Addr);        27
    artist_stack    : wrapper_stacks.stack;                                         28
                                                                                    29
                                                                                    30
    --<< declare handler routines for both client and server >>                     31
    --<< events plus any auxilliary routines here.          >>                     32
    --                                                                             33
    --server events handlers must have the signature:                              34
    --procedure <handler_name> (Object : CSL.Object_Type;                         35
    --Event  : CSL.Chiron_Event_Ptr);                                              36
    --                                                                             37
    --client events handlers must have the signature:                              38
    --procedure <handler_name> (Event : Client_Event_Ptr);                        39
                                                                                    40
    procedure Handle_Select (object : CSL.Object_Type;                              41
                             event  : CSL.Chiron_Event_Ptr) is                      42
      value  : integer;                                                             43
      result : integer;                                                             44
    begin                                                                           45
```

```
  if object = push_button then
    --show dialogue box:
    ~Apply(ADL_popup_frame,
          pop_up_frame,
          set_show,
          true);
  elsif object = pop_button then
    --if stack not empty:
    if ( wrapper_stacks.depth (artist_stack) > 0 ) then

      --pop:
      value := wrapper_stacks.pop (artist_stack);
    end if;
  elsif object = pop_up_field then
      --if stack not full:
      if ( wrapper_stacks.depth (artist_stack) <
           wrapper_stacks.max (artist_stack) ) then
        --push:
        wrapper_stacks.push (artist_stack, event.num_val);
      end if;
      --hide dialogue box:
      ~Apply(ADL_popup_frame,
            pop_up_frame,
            set_show,
            false);
  end if;
end Handle_Select;


procedure Handle_Stacks_Create (Event : Client_Event_Ptr) is
begin
  --update depiction
  artist_stack := Event.Stacks_Create_Result;
end Handle_Stacks_Create;


procedure Handle_Stacks_Push (Event : Client_Event_Ptr) is
  value : integer;
begin
  --update depiction
  artist_stack := Event.Stacks_Push_S;

  ~Apply(ADL_text_fld,
        top_field,
        set_value,
        to_str(integer'image (Event.Stacks_Push_X)));
```

```
                                                                    94
    value := wrapper_stacks.depth (artist_stack);                   95
    ~Apply(ADL_text_fld,                                            96
           depth_field,                                             97
           set_value,                                               98
           to_str(integer'image (value)));                          99
  end Handle_Stacks_Push;                                           100
                                                                    101
                                                                    102
  procedure Handle_Stacks_Pop (Event : Client_Event_Ptr) is         103
    value : integer;                                                104
  begin                                                             105
    --update depiction                                             106
    artist_stack := Event.Stacks_Pop_S;                             107
    value := wrapper_stacks.top (artist_stack);                     108
    ~Apply(ADL_text_fld,                                            109
           top_field,                                               110
           set_value,                                               111
           to_str(integer'image (value)));                          112
                                                                    113
    value := wrapper_stacks.depth (artist_stack);                   114
    ~Apply(ADL_text_fld,                                            115
           depth_field,                                             116
           set_value,                                               117
           to_str(integer'image (value)));                          118
  end Handle_Stacks_Pop;                                            119
                                                                    120
                                                                    121
                                                                    122
                                                                    123
begin --task body                                                   124
                                                                    125
  TEXT_IO.PUT_LINE("Begin dialogue artist.");                       126
                                                                    127
  accept Start_Artist (                                             128
         ID   : CSL.Artist_ID_Type;                                 129
         Aptr : SYSTEM.ADDRESS;                                     130
         Display_Name : CSL.Str) do                                 131
    Self_Ptr := address_to_artist(Aptr);                            132
    Local_Artist_ID := ID;                                          133
    Local_Display := Display_Name;                                  134
                                                                    135
    --<< register interests in client events with the >>           136
    --<< appropriate dispatchers here.              >>             137
                                                                    138
    Client_Dispatcher.Dispatcher.Register_Event                     139
      (Local_Artist_ID, Client_Events.Stacks_Create,                140
       Handle_Stacks_Create'ADDRESS);                               141
```

```
   Client_Dispatcher.Dispatcher.Register_Event                          143
     (Local_Artist_ID, Client_Events.Stacks_Push,                       144
      Handle_Stacks_Push'ADDRESS);                                      145
   Client_Dispatcher.Dispatcher.Register_Event                          146
     (Local_Artist_ID, Client_Events.Stacks_Pop,                        147
      Handle_Stacks_Pop'ADDRESS);                                       148
                                                                        149
end Start_Artist;                                                       150
                                                                        151
                                                                        152
--<< create initial graphical objects here. >>                         153
                                                                        154
artist_frame := ~Apply(ADL_base_frame,                                  155
                  create,                                               156
                  frame_label => "Dialogue View",                       157
                  x           => 20,                                    158
                  y           => 20,                                    159
                  width       => 300,                                   160
                  height      => 275,                                   161
                  foreground  => WHITE,                                 162
                  background  => FOREST_GREEN,                          163
                  show_footer => true);                                 164
                                                                        165
artist_panel := ~Apply(ADL_panel,                                       166
                  create,                                               167
                  parent          => artist_frame,                      168
                  layout          => Layout_Horizontal,                 169
                  width           => 300,                               170
                  height          => 300,                               171
                  auto_placement  => true,                              172
                  win_below       => false,                             173
                  below_sibling   => null,                              174
                  win_right_of    => false,                             175
                  right_of_sibling => null,                             176
                  x               => 0,                                 177
                  y               => 0,                                 178
                  x_gap           => 40);                               179
                                                                        180
push_button := ~Apply(ADL_button,                                       181
                  create,                                               182
                  parent        => artist_panel,                        183
                  button_id     => 1,                                   184
                  label         => "Push",                              185
                  auto_placement => false,                              186
                  foreground    => WHITE,                               187
                  x             => 70,                                  188
                  y             => 40);                                 189
                                                                        190
```

```
pop_button := ~Apply(ADL_button,                                    192
                create,                                             193
                parent        => artist_panel,                      194
                button_id     => 2,                                 195
                label         => "Pop",                             196
                foreground    => WHITE,                             197
                auto_placement => true);                            198
                                                                    199
top_field := ~Apply(ADL_text_fld,                                   200
                create,                                             201
                parent        => artist_panel,                      202
                label         => "Top Element: ",                   203
                text_value    => "",                                204
                layout        => layout_horizontal,                 205
                display_length => 3,                                206
                stored_length  => 3,                                207
                read_only     => true,                              208
                mask_char     => ascii.nul,                         209
                foreground    => WHITE,                             210
                auto_placement => false,                            211
                x             => 50,                                212
                y             => 90);                               213
                                                                    214
depth_field := ~Apply(ADL_text_fld,                                 215
                create,                                             216
                parent        => artist_panel,                      217
                label         => "Stack Depth:  ",                  218
                text_value    => "",                                219
                layout        => layout_horizontal,                 220
                display_length => 3,                                221
                stored_length  => 3,                                222
                read_only     => true,                              223
                mask_char     => ascii.nul,                         224
                auto_placement => false,                            225
                x             => 50,                                226
                y             => 130);                              227
                                                                    228
pop_up_frame := ~Apply(ADL_popup_frame,                             229
                create,                                             230
                parent      => artist_frame,                        231
                frame_label => "Push",                              232
                x           => 275,                                 233
                y           => 50);                                 234
                                                                    235
pop_up_panel := ~Apply(ADL_panel,                                   236
                create,                                             237
                parent => pop_up_frame,                             238
                layout => Layout_Vertical);                         239
```

```
                                                                    241
pop_up_field := ~Apply(ADL_num_fld,                                 242
                  create,                                           243
                  parent        => pop_up_panel,                    244
                  label         => "Enter Integer Value:",          245
                  numeric_value  => 0,                              246
                  layout        => Layout_Horizontal,               247
                  min_value      => 0,                              248
                  max_value      => 100,                            249
                  display_length => 3,                              250
                  stored_length  => 3,                              251
                  read_only      => false,                          252
                  foreground     => WHITE,                          253
                  auto_placement => true,                           254
                  x             => 0,                               255
                  y             => 0,                               256
                  foreground     => ADL_color_not_specified,        257
                  show          => true);                           258
                                                                    259
~Apply(ADL_panel,                                                   260
     pop_up_panel,                                                  261
     ADL_window_fit);                                               262
                                                                    263
~Apply(ADL_popup_frame,                                             264
     pop_up_frame,                                                  265
     ADL_window_fit);                                               266
                                                                    267
~Apply(ADL_panel,                                                   268
     artist_panel,                                                  269
     ADL_window_fit_height);                                        270
                                                                    271
~Apply(ADL_base_frame,                                              272
     artist_frame,                                                  273
     ADL_window_fit);                                               274
                                                                    275
                                                                    276
                                                                    277
--<< set behaviors of graphical objects in response >>             278
--<< to server events here.                        >>             279
                                                                    280
behaviors(Select_Event) := Handle_Select'ADDRESS;                   281
                                                                    282
~Set_Behavior(ADL_button,                                           283
          behaviors);                                               284
                                                                    285
~Set_Behavior(ADL_num_fld,                                          286
          behaviors);                                               287
                                                                    288
```

```
                                                                        290
      --<< call adl start_processing method here. >>                    291
                                                                        292
      ~Apply(ADL_base_frame,                                            293
           artist_frame,                                                294
           start_processing);                                           295
                                                                        296
                                                                        297
                                                                        298
      loop                                                              299
        select                                                          300
          accept Notify_Client_Event (                                  301
                  Client_Event    : Client_Events.Client_Event_Ptr;     302
                  Handler_Routine : SYSTEM.ADDRESS);                    303
        or                                                              304
          accept Notify_Server_Event (                                  305
                  Object          : CSL.Object_Type;                    306
                  Server_Event    : CSL.Chiron_Event_Ptr;               307
                  Handler_Routine : SYSTEM.ADDRESS);                    308
        or                                                              309
          accept Terminate_Artist do                                    310
            --unregister for all client events                          311
            for Events in Client_Event_Kind loop                        312
              Client_Dispatcher.Dispatcher.Unregister_Event (           313
                Local_Artist_ID, Events);                               314
            end loop;                                                   315
            ~Apply(ADL_base_frame,                                      316
                 artist_frame,                                          317
                 destroy,                                               318
                 artist_frame);                                         319
          end Terminate_Artist;                                         320
        end select;                                                     321
      end loop;                                                         322
                                                                        323
    end Dialogue_Artist;                                                324
                                                                        325
  end Dialogue_Artist;                                                  326
```

D Association tables package specification

```
--Copyright (c) 1992, 1993 Regents of the University of California.
--All rights reserved.
--
--This software was developed by the Arcadia project
--at the University of California, Irvine.
--
--Redistribution and use in source and binary forms are permitted
--provided that the above copyright notice and this paragraph are
--duplicated in all such forms and that any documentation,
--advertising materials, and other materials related to such
--distribution and use acknowledge that the software was developed
--by the University of California, Irvine.  The name of the
--University may not be used to endorse or promote products derived
--from this software without specific prior written permission.
--THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY EXPRESS OR
--IMPLIED WARRANTIES, INCLUDING, WITHOUT LIMITATION, THE IMPLIED
--WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE.


------------------------------------------------------------
--TITLE
--Association_Tables
--
--LANGUAGE
--Ada
--
--PERSONNEL
--Author:  Mary Cameron  2/90
--
--OVERVIEW
--This generic package contains routines to support the artist
--builder in maintaining association tables of ADT objects
--and their graphical representations.  The artist should
--instantiate this package foreach "type" in the ADT.
--
--NOTES
--This package is referenced by artists.
--
------------------------------------------------------------
generic
  type ADT_Type    is private;
  type Artist_Type is private;
  type Context     is private;    --allows additional information to be
                                  --associated with (ADT, Artist) pairs.

  with Function ADT_Equal    ( X,Y: in ADT_Type    ) return BOOLEAN is "=";
  with Function Artist_Equal ( X,Y: in Artist_Type ) return BOOLEAN is "=";

package Association_Tables is

  ---------
  --Types:
  ---------
  type Association_Table is private;
  type Table_Entry is private;
```

```
----
--Exceptions:
----
Entry_Not_Found      : exception;


Table_Does_Not_Exist : exception;
Invalid_Entry        : exception;


-------------------------------------------------------------------
--TITLE
--Create_Table
--
--OVERVIEW
--Creates and returns an Association_Table.
--
--PARAMETERS
--return => the created table
--
-------------------------------------------------------------------
function Create_Table
  return Association_Table;


-------------------------------------------------------------------
--TITLE
--Destroy_Table
--
--OVERVIEW
--Destroys an Association_Table.
--
--PARAMETERS
--Table => the table to destroy
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--
-------------------------------------------------------------------
procedure Destroy_Table (Table : in out Association_Table);


-------------------------------------------------------------------
--TITLE
--Insert_Entry
--
--OVERVIEW
--Inserts (ADT, Artist, Context) tuples (or entries) into table.
--
--PARAMETERS
--Table => the table in which to insert the entry
--ADT   => a handle to the ADT object
--Art   => a handle to the artist object
--Label => any other information that may be necessary
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--
-------------------------------------------------------------------
procedure Insert_Entry (Table : in out Association_Table;
                        ADT   : ADT_Type;
                        Art   : Artist_Type;
                        Label : Context);
```

```
-------------------------------------------------------------------
--TITLE
--Delete_Entry
--
--OVERVIEW
--Deletes an entry when given the ADT handle.
--
--PARAMETERS
--Table => the table in which to delete the entry
--ADT   => the handle to the ADT object
--

--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--Entry_Not_Found
--
-------------------------------------------------------------------
procedure Delete_Entry (Table : in out Association_Table;
                        ADT   : ADT_Type);


-------------------------------------------------------------------
--TITLE
--Delete_Entry
--
--OVERVIEW
--Deletes an entry when given the Artist handle.
--
--PARAMETERS
--Table => the table in which to delete the entry
--Art   => the handle to the Artist object
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--Entry_Not_Found
--
-------------------------------------------------------------------
procedure Delete_Entry (Table : in out Association_Table;
                        Art   : Artist_Type);


-------------------------------------------------------------------
--TITLE
--Is_Entry_In_Table
--
--OVERVIEW
--Determines if the specified ADT object is in the table.
--
--PARAMETERS
--Table  => the table in which to search for the entry
--ADT    => the handle to the ADT object
--return => TRUE if found
--FALSE otherwise
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--
-------------------------------------------------------------------
function Is_Entry_In_Table (Table : Association_Table;
                            ADT   : ADT_Type)
  return BOOLEAN;
```

```
------------------------------------------------------------------
--TITLE
--Is_Entry_In_Table
--
--OVERVIEW
--Determines if the specified Artist object is in the table.
--
--PARAMETERS
--Table  => the table in which to search for the entry
--Art    => the handle to the Artist object
--return => TRUE if found
--FALSE otherwise
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--
------------------------------------------------------------------
function Is_Entry_In_Table (Table : Association_Table;
                            Art   : Artist_Type)
  return BOOLEAN;


------------------------------------------------------------------
--TITLE
--Get_Entry
--
--OVERVIEW
--Searches the table for the specified ADT object, and returns
--the entry for further inquiry.
--
--PARAMETERS
--Table  => the table in which to retrieve the entry
--ADT    => the handle to the ADT object
--return => the matching entry
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--Entry_Not_Found
--
------------------------------------------------------------------
function Get_Entry (Table : Association_Table;
                    ADT   : ADT_Type)
  return Table_Entry;


------------------------------------------------------------------
--TITLE
--Get_Entry
--
--OVERVIEW
--Searches the table for the specified artist object, and returns
--the entry for further inquiry.
--
--PARAMETERS
--Table  => the table in which to retrieve the entry
--Art    => the handle to the artist object
--return => the matching entry
--
--POSSIBLE EXCEPTIONS
--Table_Does_Not_Exist
--Entry_Not_Found
--
------------------------------------------------------------------
```

```
function Get_Entry (Table : Association_Table;
                    Art   : Artist_Type)
  return Table_Entry;


------------------------------------------------------------------
--TITLE
--Get_ADT_Object
--
--OVERVIEW
--Given the entry, returns the handle to the ADT object.
--
--PARAMETERS
--E      => an entry in the table (begotton from GET_ENTRY)
--return => the ADT object
--
--POSSIBLE EXCEPTIONS
--Invalid_Entry
--
------------------------------------------------------------------
function Get_ADT_Object (E : Table_Entry)
  return ADT_Type;


------------------------------------------------------------------
--TITLE
--Get_Artist_Object
--
--OVERVIEW
--Given the entry, returns the handle to the artist object.
--
--PARAMETERS
--E      => an entry in the table (begotton from GET_ENTRY)
--return => the artist object
--
--POSSIBLE EXCEPTIONS
--Invalid_Entry
--
------------------------------------------------------------------
function Get_Artist_Object (E : Table_Entry)
  return Artist_Type;


------------------------------------------------------------------
--TITLE
--Get_Context
--
--OVERVIEW
--Given the entry, returns the context.
--
--PARAMETERS
--E      => an entry in the table (begotton from GET_ENTRY)
--return => the context
--
--POSSIBLE EXCEPTIONS
--Invalid_Entry
--
------------------------------------------------------------------
function Get_Context (E : Table_Entry)
  return Context;
```

```
private

   type Association_Table_Body;
   type Association_Table is access Association_Table_Body;

   type Table_Entry is record
      ADT_element    : ADT_Type;
      Artist_element : Artist_Type;
      Label          : Context;
   end record;

end Association_Tables;
```

D.1 Execution of client_builder for flight simulator ccf example

```
<<Generating an Artist Manager>>

Files Created:    artist_managager_spec.a
                  artist_manager_body.a

<<Generating a Client Initializer>>

Files Created:  client_init.a

<<Generating Client Events>>

Parsing file: aileron_spec.a ...
#  0 error(s) found.
Parsing file: altitude_spec.a ...
#  0 error(s) found.
Parsing file: attitude_spec.a ...
#  0 error(s) found.
Parsing file: psi_spec.a ...
#  0 error(s) found.
Parsing file: speed_spec.a ...
#  0 error(s) found.
Parsing file: tail_spec.a ...
#  0 error(s) found.
Parsing file: theta_spec.a ...
#  0 error(s) found.
Parsing file: throttle_spec.a ...
#  0 error(s) found.

File Created:  client_events.a

<<Generating Wrappers>>

Parsing file: aileron_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_aileron_module_spec.a
                  wrapper_aileron_module_body.a
                  aileron_module_controller.a

Parsing file: altitude_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_altitude_module_spec.a
                  wrapper_altitude_module_body.a
                  altitude_module_controller.a

Parsing file: attitude_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_attitude_module_spec.a
                  wrapper_attitude_module_body.a
                  attitude_module_controller.a

Parsing file: psi_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_psi_module_spec.a
                  wrapper_psi_module_body.a
                  psi_module_controller.a
```

```
Parsing file: speed_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_speed_module_spec.a
                  wrapper_speed_module_body.a
                  speed_module_controller.a

Parsing file: tail_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_tail_module_spec.a
                  wrapper_tail_module_body.a
                  tail_module_controller.a

Parsing file: theta_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_theta_module_spec.a
                  wrapper_theta_module_body.a
                  theta_module_controller.a

Parsing file: throttle_spec.a ...
#  0 error(s) found.

Files Created:    wrapper_throttle_module_spec.a
                  wrapper_throttle_module_body.a
                  throttle_module_controller.a

<<Generating Dispatchers>>

Files Created:    dispatcher_attitude_module_spec.a
                  dispatcher_attitude_module_body.a
                  dispatcher_psi_module_spec.a
                  dispatcher_psi_module_body.a
                  dispatcher_tail_module_spec.a
                  dispatcher_tail_module_body.a
                  dispatcher_theta_module_spec.a
                  dispatcher_theta_module_body.a
                  client_dispatcher_spec.a
                  client_dispatcher_body.a

<<Generating Artist Templates>>

File Created:   aileron_module_artist.cal

File Created:   tail_module_artist.cal

File Created:   throttle_module_artist.cal

File Created:   pitch_artist.cal

File Created:   roll_artist.cal

File Created:   altitude_module_artist.cal

File Created:   altimeter_module_artist.cal

File Created:   airspeed_module_artist.cal

File Created:   compass_module_artist.cal

File Created:   turn_module_artist.cal

File Created:   horizon_module_artist.cal
```

E ADL Reference Manual

The remainder of this manual is a listing of the *ADL Reference Manual*. The ADL Reference Manual lists all of the available classes in the heirarchy along with their methods and signatures.