User Interface Development and Software Environments: The Chiron-1 System

Rudolf K. Keller,
Mary Cameron, Richard N. Taylor, and Dennis B. Troup

Department of Information and Computer Science
University of California
Irvine, CA 92717

Technical Report 90-25

September 1990

Abstract

User interface development systems for software environments have to cope with the broad, extensible and dynamic character of such environments, must support internal and external integration, and should enable various software development strategies. The Chiron-1 system adapts and extends key ideas from current research in user interface development systems to address the particular demands of software environments. Important Chiron-1 concepts are: separation of concerns, dynamism, and open architecture. We discuss the requirements on such user interface development systems, present the Chiron-1 architecture and a scenario of its usage, detail the concepts it embodies, and report on its design and prototype implementation.

Keywords: User Interface Development Systems, Software Engineering Environments, Concurrent Systems.

1 Introduction

There is a great need for tools to help design, implement, and maintain user interfaces. Human interface development strategies such as those involving iterative design tend to be very expensive, and interface software seems to be inherently difficult to write. Furthermore, the trend towards ever more sophisticated interfaces is continuing, imposing new challenges on interface developers. User interface tools promise relief in that the quality of the interfaces is ensured, or even improved, while the effort to create and maintain them is kept under control and possibly radically reduced.

Myers [Mye89b] distinguishes two types of user interface tools: user interface toolkits and user interface development systems. A *user interface toolkit* is a "library of interaction techniques, where an interaction technique is a way of using a physical input device to input a value, along with the feedback that appears on the screen." A *user interface development system* (UIDS) is "an integrated set of tools that help programmers create and manage many aspects of interfaces" [Mye89b]. These systems are often referred to as user interface management systems, but we prefer the more general notion of UIDS, and use that term to characterize our work.

Early software environments typically supported a narrow range of activities, most notably programming, or have been restricted to a single "hard-wired" software development process. Consequently, user interface tools dealt with limited or relatively static tasks, which are typical features of most of the currently available systems. Moreover, many systems constrain the interface between application and user interface code by imposing, for example, a dispatch structure (often found with toolkits), or a transition-diagram-driven control structure. The emergence of much more general environments calls for novel user interface tools which are more powerful, more adaptive, and less restrictive than existing ones.

The *Arcadia* research project [TBC+88] is investigating the construction of software environments that are integrated, yet flexible and extensible enough to support experimentation with alternative software processes and tools. A major sub-project of Arcadia is *Chiron*. The goal of the Chiron project is to explore the particular demands and constraints on a UIDS for such environments, the interrelation between the environment architecture and the UIDS, and the user interface development strategies most appropriate for such environments.

This research is being validated by a series of prototypes of environments and UIDSs. A first effort resulted in *Chiron-0*, a UIDS which reflects the user interface research of the first phase of the Arcadia project and which is described in [YTT88]. The lessons learned from Chiron-0, the progress made in the Arcadia prototype environments, and the ambitious goals of the second phase of the Arcadia project inspired the *Chiron-1* UIDS project.

The proposed design of the *Chiron-1* UIDS has been described in [KCTT90]; in this paper we present the system as it has been built. We first discuss the special requirements software environments impose on UIDSs. Then, an overview of the Chiron-1 system is given. Thereafter follows a description of how Chiron-1 applications are built. Next, we point out some important Chiron-1 concepts and show how they make Chiron-1 a powerful platform for user interface development in software environments. Finally, we give an evaluation of our

design, and present the current status and future research directions of our project.

2 UIDS Requirements for Software Environments

Arcadia research indicates that software environments must be broad in scope, highly flexible and extensible, and very well integrated. This set of characteristics drives special demands on the environments' user interface facilities.

Below we briefly discuss these characteristics, together with the requirements they impose on supporting UIDSs. We have structured the discussion into five sections, claiming that a UIDS should support a broad and extensible scope, dynamism, tight integration, and various development strategies.

These requirements are not disjoint, and there exist tensions between the different items (see below). Furthermore, we do not claim that the list of requirements is complete. It rather shows the major additional requirements that should come together with the requirements for "standard" UIDSs (cf. [Loe88]).

2.1 Broad and Extensible Scope

In our estimation software environments should support the full spectrum of software development and maintenance activities: from requirements definition and test planning, to modeling and analysis, to preparation and maintenance of user manuals. There is also enormous variation in the objects utilized (e.g., graphs, structured text, running programs, etc.). The scope of such environments is therefore much broader, for example, than the scope of programming environments.

A UIDS must not limit the scope of environments by making assumptions about the types of objects to be manipulated. It should rather allow the presentation of a variety of objects, in various depictions, whether or not they are static or dynamically created. UIDS architectures are thus required to be open in respect to control models and tool interfaces. Many current UIDSs deal with restricted application areas, making, for instance, assumptions on the static nature of applications or relying on certain application models such as editor models, e.g., *Dost* [DS86]. For UIDSs in extensible software environments such limitations are not acceptable.

Furthermore, software environments should allow adding new capabilities to address new needs. New development and analysis techniques appear regularly; it is only sensible to integrate them into an environment. Doing so can be difficult, however, unless an environment is designed with this kind of growth and change in mind.

2.2 Dynamism

Many software development activities are dynamic, especially when executing code is involved. Typical examples are runtime analysis or interactive debugging.

A UIDS should support the visualization of such activities. When objects are dynamically created, the UIDS should provide means to depict them in a way which is dependent upon

the current state of the execution. In many cases the depictions *have* to be created or adapted at runtime. Compelling examples are: long running processes such as system design, during which new tools are brought in and utilized, debuggers for concurrent programs where an error case might not be readily reproducible upon re-start, applications where types to be depicted are dynamically created, or applications which deal with a lot of data and which lack a means for persistent storage.

2.3 Tight Integration

A software environment should be both externally and internally integrated. By *external integration* we mean that the user's interactions with all of the tools and facilities of the environment are as uniform and comfortable as possible. *Internal integration* deals with the tool builder's perspective: an environment is internally integrated if there are uniform mechanisms for tool invocation and combination, data sharing, persistent object management, etc. The addition of new tools to the environment, a quite common process in extensible environments, should hence be seamless for both the user and the tool builder.

Current UIDSs often promote external integration; in fact, external integration is one of the fundamental goals of UIDSs. Internal integration, however, is sacrificed (or ignored) in many UIDSs in that the interface to the functional portion of applications is restricted, twisted or hampered by properties of the UIDS.

2.4 Various Development Strategies

We assume that in broad-scope software environments a variety of development strategies is adopted. These strategies can have heavy impacts on UIDS issues as the three examples below suggest.

First, in large software projects, many developers are usually involved, and *cooperative development* might be applied. This can involve simultaneous access and manipulation of shared data, use of project management tools, use of various communication media, etc. (cf. [Gib89]). A UIDS should support cooperative work by providing, for instance, user interface consistency (possibly at different levels) among users working on the same data or running the same application (cf. section 3.3).

Second, development might be *bi-directional* in the sense that either the user interface or the functional parts of an application are the starting point of an application's development. An example of the former case is the technique of evolutionary prototyping (cf. [BK89]), where the user interface is iteratively specified, refined, implemented and finally extended to a full-blown application. The latter case can occur when the functional parts of an application have already been implemented, and a user interface should be attached without changing the already existing parts. It's clear that a UIDS should support both directions of development.

Finally, a UIDS is used by a variety of people, acting in *different roles*, such as application programmers, user interface designers, etc. A UIDS should also take into account that users perceive their needs differently, according to their level of skill and experience.

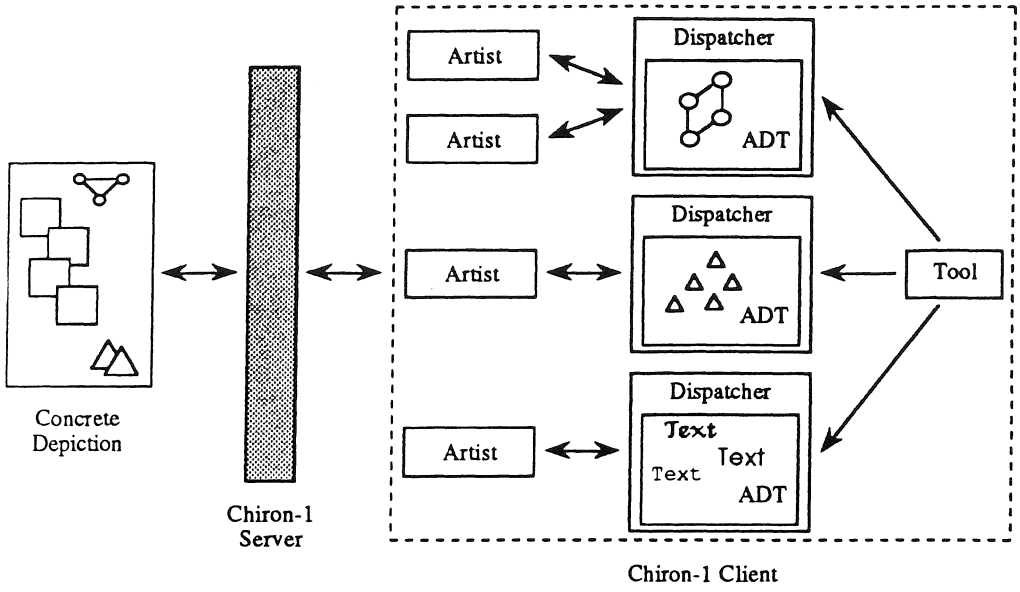

Figure 1: A tool and its user interface in the Chiron-1 model

This list of requirements is very ambitious; quite possibly, a single UIDS cannot cover all above mentioned issues. With Chiron-1 we have tried to investigate the most challenging ones, the results being presented in the rest of this paper. Other significant UIDSs include *Dynamic Windows* [MYM89], *Garnet* [Mye89a], *Higgens* [HK88], and *Serpent* [BCH+90]; space limitations do not permit a comparison of their capabilities and Chiron-1's in this paper.

3 Chiron-1 System Overview

To give an overview of Chiron-1, we first discuss the UIDS model on which the system is based. Then we describe its language interface. Finally, we present the Chiron-1 architecture. Since Chiron-1 at runtime is a system comprised of a server and an arbitrary number of clients, we have divided the discussion into a client and a server section.

3.1 Chiron-1 UIDS Model

Faced with the requirements discussed in the previous section and given the positive experiences gained from the Chiron-0 model, we decided to adopt and substantially extend the Chiron-0 model. Chiron-1 uses an *artist-based*, *concurrent* UIDS model, which makes a clear separation between the applications' functional and user interface parts, while still guaranteeing effective and efficient communication between those parts. Moreover, it distinguishes application-specific and general-purpose user interface parts, allowing applications to be organized accordingly.

Artists for data structures were introduced by Myers [Mye83] in the *Incense* symbolic debugging system. Artists are associated with abstract data types (ADTs), and encapsulate decisions about how instances of those ADTs are depicted. In terms of Smalltalk's model-view-controller paradigm, an artist maintains a bi-directional mapping between model objects and corresponding view objects. *Loops* [SB86, SBK86] binds the equivalent of artists to objects using a specialized form of inheritance called *annotation*. Chiron-1 adopts a more formal version of annotation for binding artists to ADTs. An annotation of an ADT may add new operations, extend existing operations, and add local state. New operations and extensions to existing operations may modify only local state. An artist adds local state to keep track of the depiction of an object, and extends existing operations to update the depiction when the object is modified. Annotations are further discussed in [YTT88].

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 ADT are changed. Application code need not be modified just because the object it is manipulating is depicted on the screen. Thus, the interface to the functional parts of an application is not corrupted by the user interface.

Since artists do not alter the syntax or semantics of the underlying ADT, multiple artists may be attached to a single ADT. This makes it possible to display *multiple, coordinated views* of instances of the same ADT.

The artist approach has some implications for the functional parts of applications (henceforth referred to as "tools") to which Chiron-1 attaches user interfaces. Chiron-1 requires that the tools be ADT-based: all references and manipulations of objects should be performed by procedures and functions. In particular, objects should be explicitly created and destroyed with procedure or function calls. This gives the artists the opportunity to augment those operations (visually displaying their effects).

Figure 1 illustrates the Chiron-1 model. The tool meets Chiron-1's requirements and is based on ADTs. Each ADT is wrapped into a *dispatcher*, a software layer that dispatches incoming calls from both the tool and the artists. When the tool calls an ADT (via the associated dispatcher), the call is propagated to all attached artists, which update the *concrete depiction*, i.e., the display. The double arrows indicate that information also flows from the display to the artists, and back to the ADTs.

Chiron-1 manages all the parts of user interfaces which are not artist-specific (and not application-specific) in the *Chiron-1 server* (drawn as a black box in figure 1). The server can be viewed as a virtual machine which provides a high-level interface to the artists in the form of the *Abstract Depiction Language* (see below). Thus, lower-level user interface software, such as an underlying user interface toolkit and the window system, are hidden from the artists. As the name "server" suggests, several tools can communicate with the same Chiron-1 server. Accordingly, a tool together with its objects, dispatchers, and attached artists is called a *client*. Servers and clients run as separate processes (see section 3.4).

The tool, the artists, and the server are all active and maintain their own, possibly multiple, threads of control. This *concurrent model* avoids imposing sequential control upon tools, and can exploit potential parallelism in distributed implementations.

3.2 Chiron-1 Language Interface

The Chiron-1 system contains an object-oriented language called the *Abstract Depiction Language* (ADL), together with predefined ADL class libraries. ADL provides a framework in which the client developer programs the user interface parts of artists. Rather than being written directly in ADL, artists are programmed in an artist language, which is the language of the tool, supplemented with an interface to ADL. Below, we provide a description and rationale of these languages, discuss their impact on the design of Chiron-1, and show how they are used in concert for specifying event handling. An example of their usage is presented in section 4.1.

3.2.1 Abstract Depiction Language

ADL is used to describe and manipulate the artists' user interface objects. It is called ADL, because at runtime, the server stores information provided by ADL programs in *abstract depiction* data structures (see below). There are high-level ADL classes which handle default objects and behavior. The client developer specializes these classes to deal with objects and behavior specific to the user interface under consideration, using the inheritance mechanism built into ADL.

Unlike many other UIDS languages, which are poorly structured (non-local control flow, use of global variables, etc. [Mye89b]), ADL should be a high-level programming language. It should be object-oriented, since this feature makes a language especially well-suited for the UIDS domain [LVC89]. Furthermore, an "ideal" ADL should be small, expressive, easy to use, and have fast execution.

As a vehicle to specify ADL, we developed a language called *Doodle* which borrows ideas from other object-oriented languages and systems, including Smalltalk, Eiffel, Trellis/Owl, Coral, and C++. It supports constraints and persistence and contains a predefined class library. A preliminary description of Doodle can be found in [BCJ+89].

The Chiron-1 design is ADL-independent in that ADL can represent different object-oriented languages such as C++, Eiffel, and Doodle.

3.2.2 Artist Language

In the current design, we call the artist language *Lo-CAL*. "Lo-CAL" stands for lo(w) Chiron Artist Language[1]. Lo-CAL is a superset of Ada (we assume, for the Arcadia context, tools are written in Ada) that contains an interface to ADL, allowing artists to manipulate ADL class objects. At runtime, ADL instructions are sent to the server where they are interpreted. Thus, Lo-CAL bridges the gap between the "tool language" (Ada) and the "server language" (ADL).

Artists are components which connect the tools to the server. Thus, prime candidates for an artist language were ADL and the tool languages. We have not chosen ADL as the artist

[1] "low" was adopted for historical reasons: at an earlier stage of development, we had planned to define an artist language which would have included the complete functionality of ADL. Since we called that language "CAL", we call the present, much simpler language "Lo-CAL".

language, because artist writers would possibly have to learn a new language and because there would be an ADL/tool language interface in the clients. Furthermore, we did not want to restrict the design to one particular tool language such as ADL.

A natural solution was to adopt Ada as an artist language and to provide an Ada/ADL interface. Instead of coercing legal Ada to be both the ADL and the artist language (faking features such as inheritance and dynamic binding as was done in Chiron-0), we decided to define Lo-CAL.

Extending Chiron-1 in the future to support another tool language will mean that a new artist language, for instance, "Hi-CAL", has to be defined. Like Lo-CAL, Hi-CAL would be a superset of its tool language, containing an interface to ADL. Chiron-1 would have to adapt the standard client runtime system (see below) and the client building tools (see section 4). The server, however, would remain unchanged.

With the ADL/Lo-CAL languages, a two-level mechanism for specifying event handling is provided: ADL classes may contain their own event methods. These methods can be used to specify (by a special command) that events should be propagated to the artists. In order to handle incoming events, artist programs have to specify their own event handlers. This is done by a Lo-CAL procedure which takes as parameters a class name, an event, and the associated event handler. If an event is handled neither at the ADL level nor at the Lo-CAL level, it is ignored.

3.3 Client Architecture

Figure 2 shows the client side of an executing application using Chiron-1. The figure is simplified in that it contains only one ADT. Ellipses represent components running in parallel; rectangles represent data, as well as components that are invoked by other components (i.e., that do not run continuously on their own).

The *tool* makes calls into the dispatcher.

The *dispatcher* is generated from the ADT specification by the dispatcher generator (see section 4). It wraps around all procedures and functions of the ADT, and presents an interface that is identical to the ADT's interface. The tool replaces all calls to the original ADT with calls to the dispatcher. The bodies of the wrapping procedures and functions contain calls to the original ADT, followed by a sequence of calls to the artists associated with the dispatcher (one call per artist). With this scheme, the artists are informed of each procedure call to the ADT, with only a minor modification to the tool's code. This scheme is a decentralized relative of the message-passing mechanisms used in Field [Rei90].

When called by the dispatcher, the *artists* may need to update their graphical depiction of the ADT, by sending instructions to the *client protocol manager*. The client protocol manager encodes the instructions into a protocol form and sends them to the server.

The server sends Chiron-1 events to the client protocol manager, which acts as a decoder. *Chiron-1 events* describe the object receiving the event and information about the event. The client protocol manager forwards them to the *client event manager*, which calls the

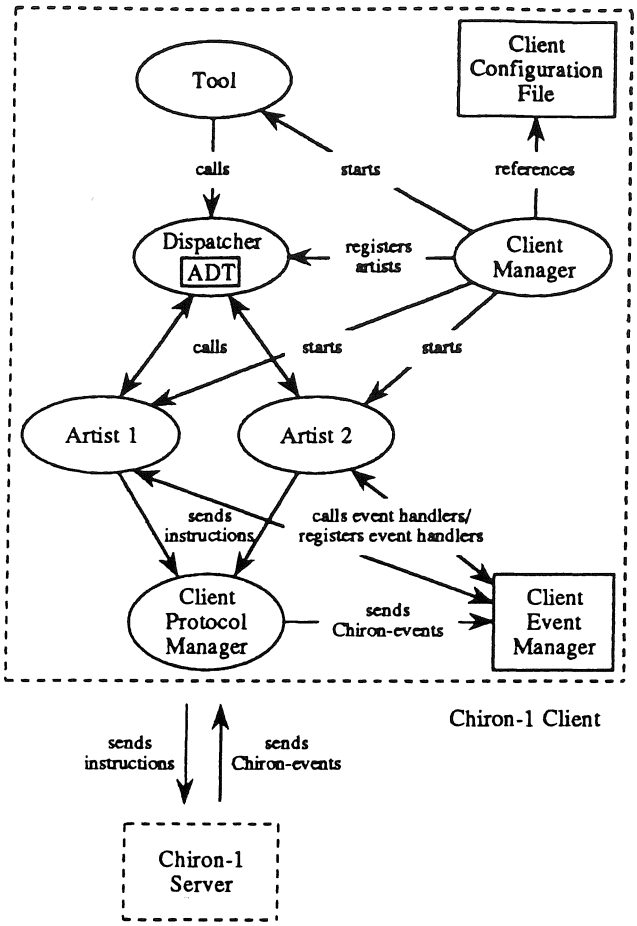

Figure 2: Chiron-1 client at runtime

appropriate event handlers in the artists (artists register event handlers with the client event manager).

The event handlers (or any part of an artist) may call the dispatcher to update the ADT instance and notify the other artists. If, for example, an artist deletes a view object after a user interaction, it might call the dispatcher to perform a delete operation on the corresponding ADT object. The dispatcher would then call the other artists which might in turn delete their corresponding view objects.

The *client manager* is responsible for coordinating the startup of the client, which involves starting the tool, starting the artists, and establishing the dispatcher-to-artist communication paths by registering artists with the dispatcher. The information needed to perform this job is contained in the client configuration file.

The *client configuration file* is a text file that contains the client's ADT names (and, im-

plicitly, dispatcher names), the artist names, and numbers indicating how many instances of each artist to start up initially. To change the client configuration at runtime, the Chiron-1 system provides a client configuration package, which consists of procedures to create and delete artist instances. This package can be called directly by the artists and the tool. Moreover, the client manager provides a graphical interface to this package, allowing the end user to alter the configuration.

The client protocol manager, the client event manager, and the client manager are all client-independent. They are provided by the Chiron-1 system and make up the *standard client runtime system.*

3.4 Server Architecture

The server architecture is illustrated in figure 3.

Instructions are sent from the clients to the server. The *server protocol manager* decodes them from the protocol form and forwards them, along with their corresponding client identifications, to the scheduler. The *scheduler* buffers and prioritizes incoming instructions and events for processing by the interpreter.

The *interpreter* is at the heart of the server, as it is responsible for controlling most of the functionality of the server. It gets commands (instructions and Chiron-1 events) from the scheduler. The instructions are references to ADL classes which were previously written and stored in *ADL class libraries.* Upon receipt of these instructions, the interpreter may either load in class code (as it maintains a cache of library information) or execute the loaded class code. It is the execution of this code that causes the abstract depictions to be maintained, that is, created and updated. The interpreter is also responsible for rendering the abstract depictions, i.e., it *references* the abstract depictions and invokes the drawing primitives. Upon getting a Chiron-1 event, the interpreter determines whether the event can be handled within the server, or whether it is *sent* to the clients.

Abstract depictions are the internal representation of the objects to be displayed. There is one abstract depiction per one client. They allow quick, incremental drawing and updating of the displays, and support the event translator.

The *event translator* gets window system events from the window system and translates them to Chiron-1 events. The translation is necessary because the window system events are, in general, too low-level and are targeted to windows as opposed to graphical objects. The event translator *references* the abstract depictions to correlate the event to the intended ADL object. The resulting Chiron-1 event is then *sent* to the scheduler.

To render the abstract depictions onto the concrete depictions, the interpreter calls upon the *drawing primitives.* The drawing primitives provide a window system independent interface, and thus hide the underlying window system from the interpreter.

Conceptually, Chiron-1 can be built on any state-of-the-art *window system.* The window system interacts with the *concrete depictions* and allows the event translator to get window system events.

We call the system comprising the abstract depictions, the event translator, and the drawing primitives the *ADL system.*

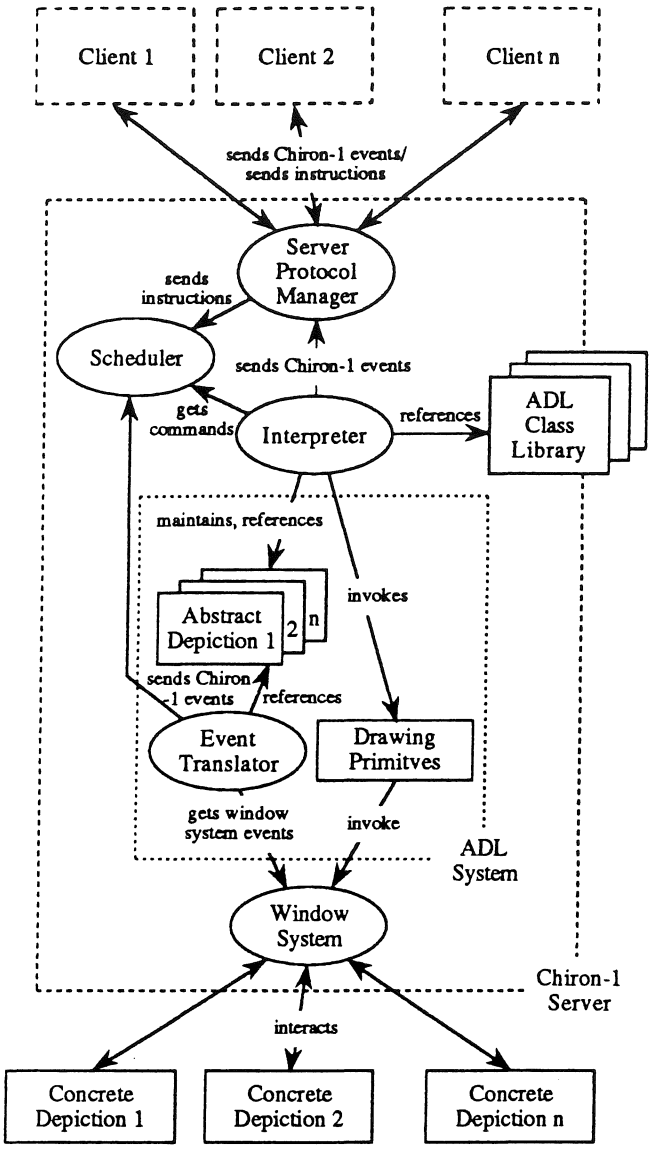

Figure 3: Chiron-1 server at runtime

4 Building Chiron-1 Applications

Building a Chiron-1 application involves in general three steps: specializing ADL classes, writing artists, and creating an executable client. In this section, we detail these steps along with the tools that Chiron-1 provides to perform them. The construction process is illustrated with a sample application. We conclude the section with a discussion of extensions to the Chiron-1 system that allow for the generation of certain artist classes.

4.1 Building a Sample Application

As an example, we have chosen the ubiquitous stack. We assume a trivial tool which is based on the following ADT:

```
package Stack_ADT is
  procedure Push (x: integer);
  function  Pop return integer;
  function Top return integer;
  function Depth return integer;
end Stack_ADT;
```

We want to build an application which allows the user to view and manipulate the stack in two different, coordinated representations (see figure 4). In the "cafeteria view", we use the well-known plate holder metaphor: the stack is represented as a pile of plates resting on a spring. Dragging plates onto/from the top of the pile corresponds to push/pop operations. (For brevity, we have omitted a plate supply and a plate "trash can", two of the mechanisms which would give this metaphor full functionality.) In the "dialog view," information on the status of the stack is displayed in numerical fields, and manipulation is triggered by push buttons.

To build the application (in the example, a client comprising a cafeteria and a dialog artist), one performs the following steps:

Specializing ADL Classes The artist developer might specialize the predefined ADL classes to cope with the special needs of the artists under consideration. Chiron-1 provides an *ADL processor*, which checks new classes syntactically and translates them into the internal form which is required by the Chiron-1 server. Once a class is in the internal form, it can be inserted into a class library by using the *librarian*. The librarian is an interactive tool that controls all access to class libraries: it creates and destroys class libraries, inserts and deletes classes, and it allows class libraries and classes to be browsed.

Chiron-1's predefined class libraries will eventually be quite rich. Thus, this step may often be skipped. In the currently available "Chiron-1 standard library", dialog views are fully supported. To implement the cafeteria view, one could use the predefined ADL_polyline, ADL_rectangle, and ADL_text classes. To simplify the artist, we defined two new classes, ADL_spring and ADL_spring_stack (ADL_spring_stack was implemented with the ADL_spring class).

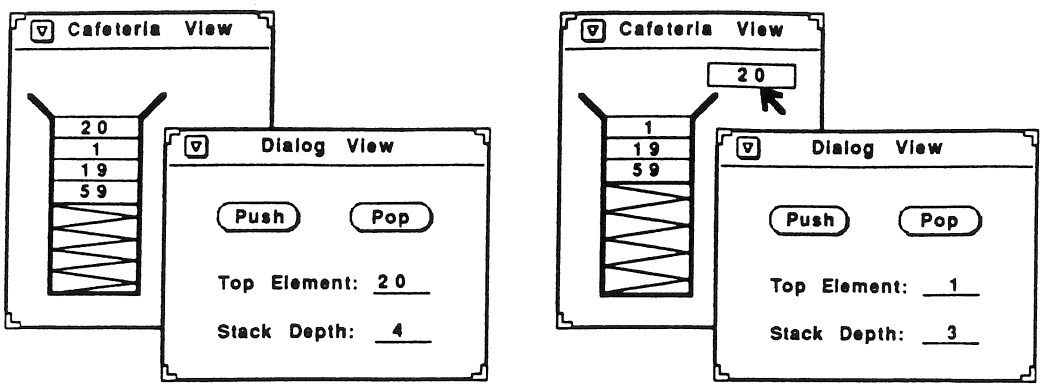

Figure 4: Sample Chiron-1 application: Stack views before (left) and after (right) popping the top plate

Writing Artists The artist developer writes the artist programs in Lo-CAL. (Figure 5 shows excerpts of the cafeteria artist Lo-CAL program.) Lo-CAL differs from legal Ada by having a set of six predefined procedures and functions that might be called with a varying number of parameters, of distinct type. They allow the developer to open and close ADL class libraries (lines 30, 47), to create and delete ADL class instances (lines 31, 34), to apply ADL class methods (lines 13, 16, etc.), and to specify (artist) event handlers (lines 35, 36).

To bridge the ADL and Lo-CAL type systems, the Chiron-1 system provides packages which contain Ada mappings of the ADL type definitions (lines 2, 9, 10, 28). These packages are generated upon insertion of ADL classes into class libraries.

Since the artists' package specifications are determined by the ADTs (or dispatchers) to which they are attached, the Chiron-1 system provides an *artist template generator*. It generates complete artist package specifications and templates of the artist bodies.

Artist programs are processed by the *Lo-CAL processor*, which checks them syntactically, and converts them into legal Ada.

Creating an Executable Client Each of the tool's ADTs which will be annotated by artists, needs to be wrapped into a dispatcher. The dispatchers are generated by the *dispatcher generator*, which takes ADTs as its input. The tool, the ADTs, the dispatchers, and the artist programs (in Ada) are compiled and linked in with the *standard client runtime system*. Finally, to run the client, a *client configuration file* must be provided (cf. section 3.3).

Figure 4 shows a snapshot before and after a pop operation. After the top plate (label "20") has been dragged outside the loading area (the region above the plate holder from where plates "drop" onto the holder), the cafeteria artist's event handler is called (figure 5, line 9), which places the plate at its new location (line 20) and which calls the Stack_ADT's dispatcher (line 21). The dispatcher calls the Stack_ADT's Pop function and broadcasts a pop message to all attached artists, i.e., to the dialog and the cafeteria artist (line 41). The

```
task type Cafeteria_Artist is
  entry Start (ID: Artist_ID);
  entry Push (x: integer);
  entry Pop (result: integer);
  entry Top (result: integer);
  entry Depth (result: integer);
end Cafeteria_Artist;
...
  procedure Handle_Move (Object: Object_Ptr;
                         Event: Chiron_Event_Ptr) is
    ...
  begin
    if Apply (ADL_spring_stack, Stack,
              is_in_loading_area, Event.Dest) then
      ...
    elsif Apply (ADL_spring_stack, Stack,
                 is_in_loading_area, Event.Orig) then
      -- top plate was moved outside loading area:
      -- complete the move and call dispatcher
      Apply (ADL_Rectangle, Object, set_xy, Event.Dest);
      Stack_ADT_Dispatcher.Pop;
    else
      null; -- spring stack not affected
    end if;
  end Handle_Move;

begin
  accept Start (ID : Artist_ID) do
    Artist_ID:= ID; end Start;
  Library_ID:= Open_Library ("My_Own_Library");
  Frame:= Apply (ADL_base_frame, create,
                 "Cafeteria View", ...);
  ...
  Stack:= Apply (ADL_spring_stack, create, ...);
  Behaviors(Move_Event):= Handle_Move'address;
  Set_Behavior (ADL_rectangle, Behaviors);
  Apply (ADL_base_frame, Frame, start_processing);
  loop
    select
      ...
    or accept Pop (result: integer) do
        Apply (ADL_spring_stack, Stack, pop_from_stack);
      end Pop; -- stack is redrawn without top plate
      ...
    end select;
  end loop;
  Close_Library (Library_ID);
end Cafeteria_Artist;
```

Figure 5: Cafeteria artist code (excerpts)

dialog artist then updates its numerical fields. The cafeteria artist calls the ADL_spring_stack object's pop_from_stack method to update its stack display (line 42).

4.2 Artist Generation

The presented scheme of application building is completely language-based (ADL, Lo-CAL). A major extension to this scheme are *front-ends*, tools which allow for the development of Chiron-1 applications on higher levels of abstraction. There is no doubt that front-ends are highly desirable, in particular for certain development strategies.

The front-ends of many state-of-the-art UIDSs support application development either by automatic creation in the sense of Foley, Olsen, and Singh [Fol87, Ols86, SG89], or by graphic specification [Hen86]. In the former case, functional descriptions are used to specify the application's semantic procedures and to automatically generate the user interface. Chiron-1 will eventually comprise front-ends of both types.

The first front-end which has been developed for Chiron-1 is the *PGraphite/artist generator* (see figure 6). It is an extension of the *PGraphite* system [WWFT88]. PGraphite takes a *GDL specification* as its input. GDL stands for "Graph Description Language" and allows for the description of the high-level structure of attributed, directed graphs. PGraphite then produces a *graph ADT* (interface package) that can be used to manipulate and persistently store graphs of that class. The PGraphite artist generator produces, in addition to the graph ADT, a *graph artist* which can be used to browse and manipulate the graph. For the purpose of artist generation, GDL was extended by a set of pragmas (in the sense of Ada). These pragmas allow for the specification of graph (artist) features such as the shape of nodes and edges, the layout algorithm, and the growth direction. To generate an executable *graph client*, the graph ADT and artist together with a *tool* are processed with the *Chiron-1 client building tools* (as described in section 4.1).

An example of a front-end based on graphic specification is a dialog box artist builder: the user specifies the layout and the items of a dialog box with a graphic editor. The tool then generates an artist and an ADT. Its design is currently underway.

5 Important Chiron-1 Concepts

In this section we discuss some distinguishing features of Chiron-1: separation of concerns, various levels of dynamism, and open architecture. They are related to each other; the sum total of their impacts gives strong support to the requirements expressed in section 2.

5.1 Separation of Concerns

Separation of concerns is a main feature of the Chiron-1 model presented in section 3.1. The clear separation of the tool, artists, and server components is accompanied by powerful mechanisms for communication between these components (dispatchers, ADL), and by pervasive concurrency.

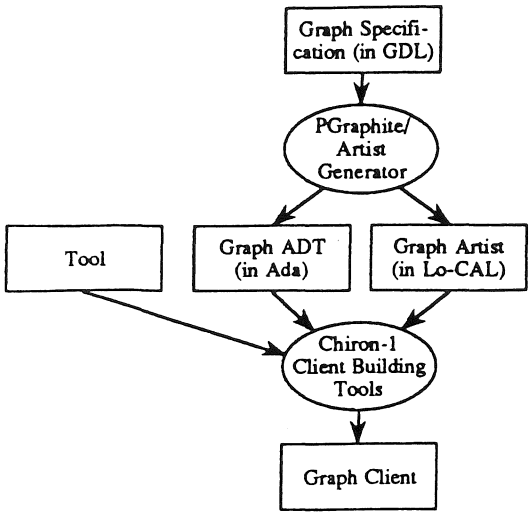

Figure 6: Graph artist/client generation

The tool-artist separation is a consequence of Chiron-1's annotation scheme. Tools need not be designed towards the UIDS, and thus, internal integration is promoted. The requirements on the tools' ADTs imposed by Chiron-1 are basically the same as on any ADT (see [GHM78]). The dispatcher mechanism allows communication between the tool and the artists via the shared ADTs. It also provides a means to coordinate artists depicting different views of the same ADT.

The artist-server separation promotes a clean design of the user interface components and runtime efficiency: the artists will usually take care of the top-level presentation decisions such as the relative positioning and selectability of figures, whereas decisions that require extensive calculations or knowledge of exact coordinates are left to the server. The interface between the artists and the server is defined by the ADL and its class libraries. Together with the event handling mechanism discussed in section 3.2, ADL is a flexible mechanism to distribute workload on the artists and the server.

These separations are complemented by concurrency. Several control models for UIDSs have been proposed, most notably, the internal (or prompting) model, and the external (or dispatcher) model. While these models might be appropriate in certain fields, a concurrent model is best suited for the broad scope of applications Chiron-1 encompasses: both the tool and the artists have their own threads of control. In addition, the client and the server run in parallel (see below).

5.2 Various Levels of Dynamism

Chiron-1 gives strong support for dynamism at various levels: the artist instance level, the artist level, and the configuration level.

At any time during execution, new artist instances can be created, and existing ones can be destroyed. This can be done by the end users via the client manager, or by running artists calling appropriate procedures (see section 3.3). At this level, new instances of a certain artist can only be created if the artist has previously been linked into the running client.

Chiron-1 also allows adding new artists at runtime (that is, artists which have not been linked into the running client), and allows for their seamless integration, later on. The basic idea of our approach is to have new artists as separate processes which communicate with the client process. These artist processes are developed using "standard" artist programs, i.e., artist programs that do not differ from artist programs written to be compiled and linked into clients. This idea is realized by furnishing the artist and client processes with a pseudo dispatcher and a pseudo artist, respectively, the pseudo entities having an inter-process communication back-end. The details are described in [BCJ+89].

In general, an executing Chiron-1 system consists of many servers and many clients, with each client communicating with at least one server. Dynamism at the configuration level means that at any time, such systems can be changed by adding and deleting servers and clients, and by establishing new communication paths.

5.3 Open Architecture

Chiron-1 is characterized by an architecture which is open towards customization and extensibility. The underlying window system can be replaced by any state-of-the-art window system. Chiron's look and feel is encapsulated in the ADL system. It can be changed by changing the structure and implementation of the ADL class libraries. A change of the ADL language requires adjustments to the interpreter, the ADL processor, and the Lo-CAL processor. Changing the artist language involves the steps described in section 3.2.2. Finally, as long as the clients obey the protocol rules, Chiron-1 allows them to abandon the artist structure altogether (although we do not recommend this kind of use).

Furthermore, the organization of Chiron-1 as a client/server system makes it open for various client/server configurations. For example, a single client can communicate with several servers. The users interacting with the different servers then interact with the same client and access the same ADTs. While there is full data consistency, that is, the users share the same model data, viewing consistency is only partial: the viewing state of the client's artists is common to all users, whereas the viewing states of the servers might differ from user to user. Mechanisms to fully exploit these flexible configurations are being explored (protection and locking schemes for cooperative workers, means to control different levels of consistency, etc.).

6 Evaluating the Design

The Chiron-1 design has undergone several refinement cycles. Using an incremental development strategy, we have produced a series of prototypes of high-risk components. In this section, we briefly detail some important issues which we have come across when realizing the

concepts of section 5.

In mapping Chiron-1's annotation concept to a design, we could profit from the lessons learned from Chiron-0 [YTT88]. When approximating annotation with Ada in Chiron-0, code duplication from the ADTs into the artists is, in certain cases, inevitable, e.g., when annotating recursive data structures. To ease this problem, Chiron-1 introduces a package to support the artists in maintaining *association tables* of ADT objects and artist objects.

Special care was taken in designing an efficient client/server and client/artist process communication. Considering the eventual support of clients written in different languages, we adopted the Q system [MS89]. Q provides support for interprocess communication between several languages, such as Ada, C, and Lisp. A series of prototypes, simulating different client/server configurations, has shown promising results.

Given the dynamic character of the Chiron-1 system, processing ADL is a highly dynamic activity. This leaves two approaches to design the ADL processor, i.e., the server's interpreter: either as an ADL interpreter or as a dynamic linker with an interpretation component (to interpret incoming events and client instructions). To ensure portability, we have decided to go with the former approach. If execution is slowed down intolerably, as it has been reported for other UIDSs containing an interpretation layer [Mye89b], we might reverse this decision. Our first server prototype has been implemented with an ADL processor containing a fixed set of ADL libraries.

Since we are not interested in toolkit development per se, nor in establishing a new look and feel, we have evaluated existing systems, e.g., *ET++* [WGM88], *InterViews* [LVC89], *XView* [Hel89]. Our first implementation supports *Open Look*, uses C++ as the ADL, and the *XView toolkit* as the basis for the ADL libraries. The ADL-independent design of Chiron-1 will allow us to use Chiron-1 as a testbed for the concepts currently being explored with the Doodle language.

7 Current Status and Future Work

A prototype implementation of Chiron-1 has recently been completed. A series of different artists is being developed which allows for extensive testing of the system. Along with the artist development, the ADL class libraries are being populated.

Among the activities which have yet to come to full fruition is development of the front-ends (cf. section 4.2). We plan to provide a suite of front-ends which supports all major client building activities: an ADL class builder, an artist builder, and a client builder. These tools will eventually be combined into one front-end system, which will also comprise a regular Chiron-1 server. This system will exploit Chiron-1's dynamic features to allow users to experiment with new artists, and to design clients iteratively.

We will also be promoting integration with the Arcadia environment, most notably with its object management facilities and with the Appl/A process programming language [TBC+88]. We want to use its object management components as vehicles to investigate persistence issues in Chiron-1. Appl/A will be supported as a tool and artist language.

Another activity we will be pursuing is validating the openness of the Chiron-1 architec-

ture. We want to support ADLs such as Doodle which incorporate advanced features, e.g., persistence and constraints. We also want to study the impacts of different tool languages on the client design.

Finally, we would like to generalize our mechanisms for client/server configurations and use Chiron-1 as a system for supporting cooperative work.

Conclusion

We have discussed a list of requirements which should be met by UIDSs to address the special demands of software environments. An overview on the Chiron-1 UIDS has outlined its UIDS model and language interface, and described its architecture. We have described how applications are built with Chiron-1, and have illustrated the process with an example. Important Chiron-1 concepts have been presented, and their significance in the software environment context has been explained. Finally, we have detailed some design issues and reported on the status of our work and the plans for future research.

Acknowledgement

We appreciate the comments and suggestions provided by our other colleagues in the Arcadia consortium.

In addition to the authors, the Chiron-1 system is being designed and implemented by Ken Anderson, Gregory Bolcer, Sheng-Lian Fu, Gregory James, Wai-Hung Lee, and John Self.

References

[BCH+90] L.J. Bass, B.M. Clapper, E.J. Hardy, R.N. Kazman, and R.C. Seacord. Serpent: A user interface environment. In *Proceedings of the Winter 1990 USENIX Technical Conference*, Washington D.C., January 1990.

[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).

[BK89] Walter R. Bischofberger and Rudolf K. Keller. Enhancing the software life cycle by prototyping. *Structured Programming*, 10(1):47–59, January–March 1989.

[DS86] Prasun Dewan and Marvin Solomon. Dost: An environment to support automatic generation of user interfaces. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 150–159, Palo Alto, California, December 1986. Appeared as *SIGPLAN Notices 22*(1), January 1987.

[Fol87] J. Foley. Transformations on a formal specification of user computer interfaces. *Computer Graphics*, 21(2):109–112, April 1987.

[GHM78] J. V. Guttag, E. Horowitz, and D. R. Musser. Abstract Data Types and Software Validation. *Communications of the ACM*, 21(12):1048–1064, December 1978.

[Gib89] Simon J. Gibbs. Liza: An extensible groupware toolkit. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 29–35, Austin, May 1989. Association for Computing Machinery.

[Hel89] Dan Heller, editor. *XView Programming Manual*, volume 7 of *The X Window System Series*. O'Reilly & Associates, Inc., Sebastopol, CA, 1989.

[Hen86] C. Austin Henderson. The Trillium user interface design environment. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 221–227, Boston, April 1986. Association for Computing Machinery.

[HK88] Scott E. Hudson and Roger King. Semantic feedback in the Higgens UIMS. *IEEE Transactions on Software Engineering*, 14(8):1188–1206, August 1988.

[KCTT90] Rudolf K. Keller, Mary Cameron, Richard N. Taylor, and Dennis B. Troup. Chiron-1: A user interface development system tailored to software environments. Arcadia Technical Report UCI-90-06, University of California, Irvine, June 1990.

[Loe88] J. Loewgren. History, state and future of user interface management systems. *SIGCHI Bulletin*, 20(1):32–44, July 1988.

[LVC89] Mark A. Linton, John M. Vlissides, and Paul R. Calder. Composing user interfaces with InterViews. *IEEE Computer*, 22(2):8–22, February 1989.

[MS89] Mark Maybee and Stephen D. Sykes. Q: Towards a multi-lingual interprocess communications model. Arcadia Technical Report UCI-89-06, University of California, Irvine, February 1989.

[Mye83] Brad A. Myers. Incense: A system for displaying data structures. *Computer Graphics*, 17(3):115–125, July 1983.

[Mye89a] Brad A. Myers. Encapsulating interactive behaviors. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 319–324, Austin, May 1989. Association for Computing Machinery.

[Mye89b] Brad A. Myers. User-interface tools: Introduction and survey. *IEEE Software*, 6(1):15–23, January 1989.

[MYM89] S. McKay, W. York, and M. McMahon. A presentation manager based on application semantics. In *Proceedings of the Symposium on User Interface Software and Technology*, pages 141–148, Williamsburg, VA, November 1989. Association for Computing Machinery.

[Ols86] Dan R. Olsen, Jr. MIKE: The menu interaction kontrol environment. *ACM Transactions on Graphics*, 5(4):318–344, October 1986. Special Issue on User Interface Software—Part 3.

[Rei90] Steven P. Reiss. Connecting tools using message passing in the field environment. *IEEE Software*, 7(4):57–66, July 1990.

[SB86] Mark Stefik and Daniel Bobrow. Object-oriented programming: Themes and variations. *AI Magazine*, 6(4):40–62, Winter 1986.

[SBK86] Mark J. Stefik, Daniel G. Bobrow, and Kenneth M. Kahn. Integrating access-oriented programming into a multiparadigm environment. *IEEE Software*, 3(1):10–18, January 1986.

[SG89] Gurminder Singh and Mark Green. Chisel: A system for creating highly interactive screen layouts. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology*, pages 86–94, Williamsburg, Virginia, November 1989.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, November 1988. Appeared as *Sigplan Notices 24*(2) and *Software Engineering Notes 13*(5).

[WGM88] A. Weinand, E. Gamma, and R. Marty. ET++ - an object-oriented application framework in C++. In *Object Oriented Programming Systems, Languages and Applications '88 Conference Proceedings*, pages 46–57, September 1988.

[WWFT88] Jack C. Wileden, Alexander L. Wolf, Charles D. Fisher, and Peri L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 130–142, Boston, November 1988.

[YTT88] Michal Young, Richard N. Taylor, and Dennis B. Troup. Software environment architectures and user interface facilities. *IEEE Transactions on Software Engineering*, 14(6):697–708, June 1988.