An Andrew Toolkit Interface to the Athena On-Line Consulting System

by

Gregory M. G. Belaus

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Electrical Science and Engineering

at the

Massachusetts Institute of Technology

May 1988

Signature of Author ____

Department of Electrical Engineering and Computer Science
May 16, 1988

Certified by ____

Jerome H. Saltzer
Thesis Supervisor

Accepted by ____

Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

An Andrew Toolkit Interface to the Athena On-Line Consulting System

by

Gregory M. G. Belaus

Submitted to the
Department of Electrical Engineering and Computer Science

May 16, 1988

In Partial Fulfillment of the Requirements for the Degree of Bachelor of Science in
Electrical Science and Engineering.

Abstract

The Andrew Toolkit is an object-oriented programming interface to the X Window System. Using this toolkit, the user-interface to the Project Athena On-Line Consulting System client programs is transformed from a single command-line to a multi-windowed environment sporting buttons, panels, and scrollbars. Through this transformation, the Andrew Toolkit is analyzed in the creation of a visual user-interface.

Thesis Supervisor: Jerome H. Saltzer
Title: Professor of Computer Science and Engineering

Dedication

To Mom and Dad, for all their support and love.

To Jerry, for giving me the opportunity that others would not give.

To the brothers of ΘΔΧ, for the last four years.

Table of Contents

Abstract **2**
Dedication **3**
Table of Contents **4**
List of Figures **5**

1. Introduction **6**
1.1 The Andrew Toolkit 6
1.2 The On-Line Consulting System 7
1.3 Scope 8

2. Creating the User-Interface **9**
2.1 Programming 9
2.1.1 Application Specific Objects 9
2.1.1.1 TitleBar 10
2.1.1.2 Buttons 10
2.1.1.3 EditWindow and PanelWindow 13
2.1.1.4 OlcWindow and OlcrWindow 16
2.1.2 The OLC System Code 16
2.2 Dividing the Real Estate: Iteration of Design 17
2.3 Controlling the State in an Event-Driven System 18
2.4 Working with the Users 20
2.5 Speed 21

3. Analysis of the Andrew Toolkit **23**
3.1 The Extensive Expansive Toolkit 23
3.2 The Object Class Hierarchy 24
3.3 The X Window System Interface 26
3.4 Documentation 26

4. Conclusion **28**

Appendix A. The Andrew Toolkit Object System **30**
A.1 The Interaction Manager 31
A.2 View and DataObject 31
A.3 Lpair and Frame 32
A.4 Text and Textview 32
A.5 Formatting the Text: Stylesheet, Style, Device, and Context 33

Appendix B. Class Header File and C Source Code for Buttons Object **34**

List of Figures

Figure 2-1:	The OLC Application Window	11
Figure 2-2:	The OLCR Application Window	12
Figure 2-3:	The OLCR Application Window: Topic Window Displayed	15
Figure 2-4:	Early Design of the OLC Application Window	19

Chapter 1

Introduction

The Andrew Toolkit is an object-oriented event-driven high-level programming interface to the X Window System. Toolkit objects are arranged in a hierarchy, such that new types of objects are created through modifications to existing objects. This thesis explores the use of the Andrew Toolkit in the creation of a window-based user-interface to the Project Athena On-Line Consulting [OLC] System. Through the creation of Toolkit objects, a window-based interface to the OLC system is developed without changes to the underlying OLC transport mechanism. Care is taken in the design of the interface so that the user is always aware of his options and what he is expected to do next.

This thesis is an evaluation of the Andrew Toolkit through the creation of a window-based user-interface to the OLC clients. Its purpose is to determine how successful the use of the toolkit is in the transformation of the command-line interface of the OLC clients to this window-based interface. Through the analysis of its success, the classes of user-interface applications that the Andrew Toolkit appears best suited for are recommended.

1.1 The Andrew Toolkit

> The Andrew Toolkit forms the base environment for all user interface and application program development in the Information Technology Center's (ITC) Andrew system. Andrew is a collaborative project between IBM and Carnegie Mellon University (CMU) to create a computing and communication system that meets the needs of universities. [Reference 88, p. 4]

"The Toolkit's high-level programming interface provides a framework for developing and combining components into user interface applications." [Overview 88, p. 3] It is through the careful arrangement of existing components in the toolkit, combined with the creation of new components based on existing components, that the OLC interface is created.

> Support for the use of multiple components within applications comes from the Andrew Class System (Class), the object-oriented system on which the Andrew Toolkit is built. Class is a C language-based system, consisting of a small run-time library and a simple preprocessor that only preprocesses class header files. [Overview 88, p. 4]

The first part of Appendix B is the class header file for the buttons object (buttons.H). C++[1] programmers will find the format of this file similar to the definition of C++ objects.[2]

> Class provides an object-oriented environment with single-inheritance. The Class language permits the definition of object methods and class procedures.... C procedures for controlling the initialization and disposition of objects are created by the Class preprocessor. [Overview 88, p. 4]

1.2 The On-Line Consulting System

The Project Athena On-Line Consulting System[3] provides a method for users to receive help from consultants who are currently not logged-in at the user's location. This system currently handles over 3500 questions per academic term. *olc*[4] and *olcr*[5] are the client programs of the system for the user and the consultant, respectively. A third program, *olcd*, is the OLC daemon program that runs on a server machine. This daemon provides an asynchronous queuing mechanism for user questions.

The existing user-interface to the client programs is based entirely on command-line processing. Each command-line interaction with the system returns a simple account of the interaction, usually displaying a particular state of the system. No state is kept between interactions, and no attempt is made to preserve the displayed information retrieved from previous interactions.

---

[1]© 1987 AT&T

[2]C++ is an object-oriented programming language compatible with C. [Cox 86, p. 42]

[3]© 1985, 1986, 1988 Massachusetts Institute of Technology

[4]On-Line Consulting program

[5]On-Line Consultant's Reply program

1.3 Scope

This is an analysis of the transformation of the command-line interface of the OLC clients to an intuitive window-based interface created with the Andrew Toolkit. Also, this is a exercise is user-interface design. It is important to note that this transformation occurs in the context of the running OLC system. No changes are made to the underlying transport mechanism of OLC. Likewise, no changes are made to the *olcd* program. The final product, the modified versions of *olc* and *olcr*, are able to interface directly to this daemon in harmony with the old olc clients. This allows the new clients to be tested and experimented with on real OLC questions in the daemon's queue. No *stub* interfaces are required for this testing.

Chapter 2

Creating the User-Interface

The creation of the window-based user-interface to the OLC clients was the simultaneous programming of the interface and the development of its design. The programming combined the creation of custom Andrew Toolkit objects with small modifications to the OLC system code. The development of the design was constrained by the environment into which the user-interface was to be placed. These design constraints were parameterized by many issues including window real-estate, the event queue of the toolkit, the end-users of the clients, and the interaction between the toolkit and the supporting window system. Each constraint, seemingly minor, played an important role in the creation of a smooth and user-intuitive user-interface.

2.1 Programming

The C-code developed in the transformation of the user-interface of the OLC clients is a combination of custom Andrew Toolkit objects and the OLC system code. The creation of custom objects forms most of this code. A discussion of the Andrew Toolkit object system and some of the objects directly used in this implementation are found in Appendix A. Minor changes to the OLC system code are made to incorporate the new interface.

2.1.1 Application Specific Objects

A vast array of both high and low level objects are provided by the Andrew Toolkit. Designing a user-interface with this toolkit requires a keen understanding of which objects should be used *as is*, and which should be subclassed, and, in effect, modified. Using objects *as is* helps to provide a consistent user-interface across multiple applications, e.g. providing the same keystroke mechanism for basic editing. Objects must be subclassed, though, if extra state information needs to be attached to an object or more specific methods for working with this data are required.

Creating the window-based interface required the instantiation or subclassing of the Andrew Toolkit objects described in Appendix A. The final product, the arrangement of these new objects, is shown for each of the clients in Figures 2-1 and 2-2. The new objects created from the Andrew Toolkit are described in the next four paragraphs. Although "the design process is *nonhierarchical*, neither strictly bottom-up nor strictly top-down" [Carroll and Rosson 85, p. 27], this is a bottom-up presentation, mimicking the actual implementation process of the objects.

2.1.1.1 TitleBar

The titleBar is the simplest of the custom created objects. It is the rectangular region found at the top of the window for each application containing the title of the application, e.g. "Project Athena On-Line Consulting System", as well as the gray rectangular region that separates the *olc* and *olcr* windows into three or four distinct subwindows, respectively. A 200 character string (the title) and a font are the data associated with a titleBar object. The titleBar is a subclass of view that displays the string in the assigned font in the center of the visual rectangle associated with the titleBar.

2.1.1.2 Buttons

The buttons object is used extensively in the window-based interface to the OLC clients. The OLC client, shown in Figure 2-1, has three distinct instances of the buttons object displayed, while the OLCR client, Figure 2-2, has four. Each buttons object, located on the right hand side of the client window, has a series of buttons associated with it. Each of these buttons is either ENABLED, DISABLED, SELECTED, ALLWAYSENABLED,

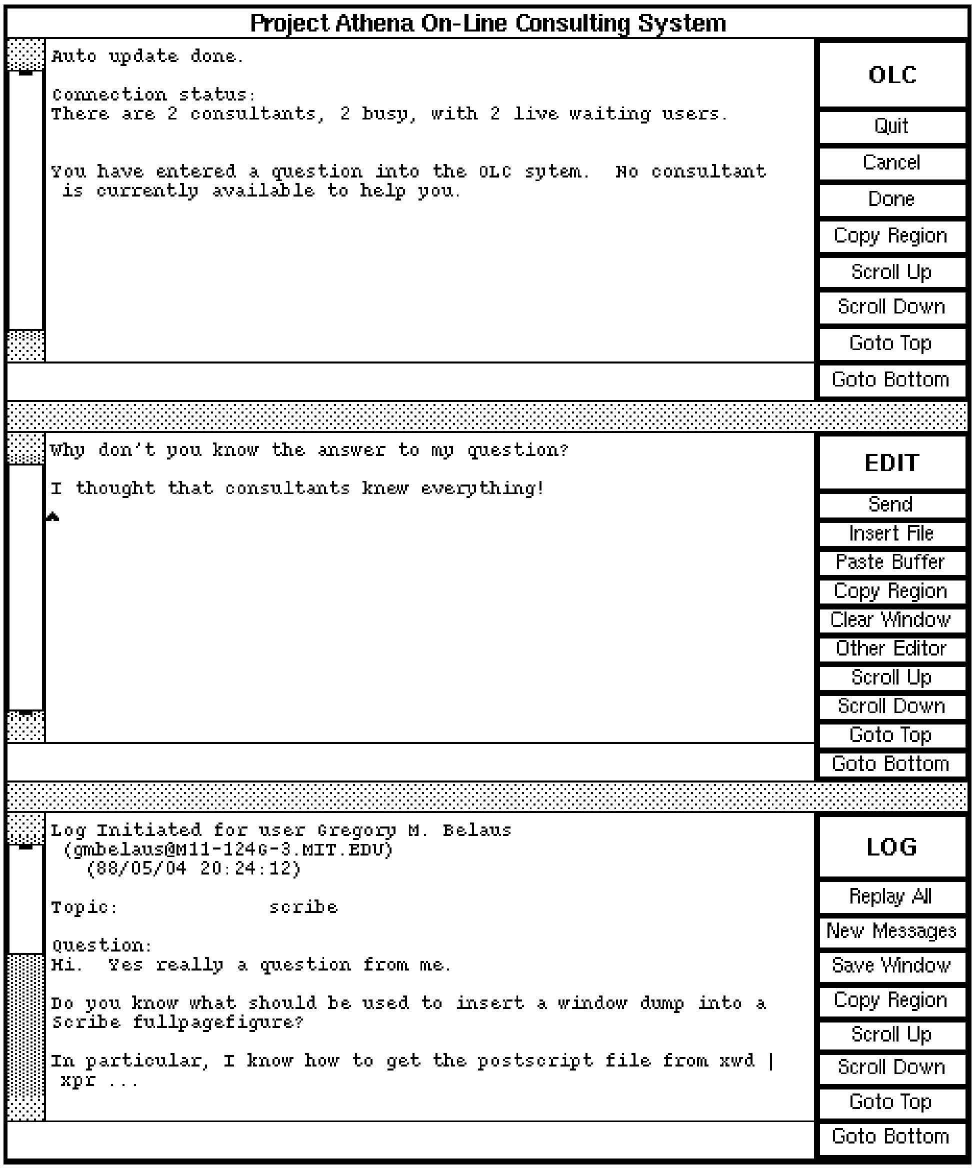

Figure 2-1: The OLC Application Window

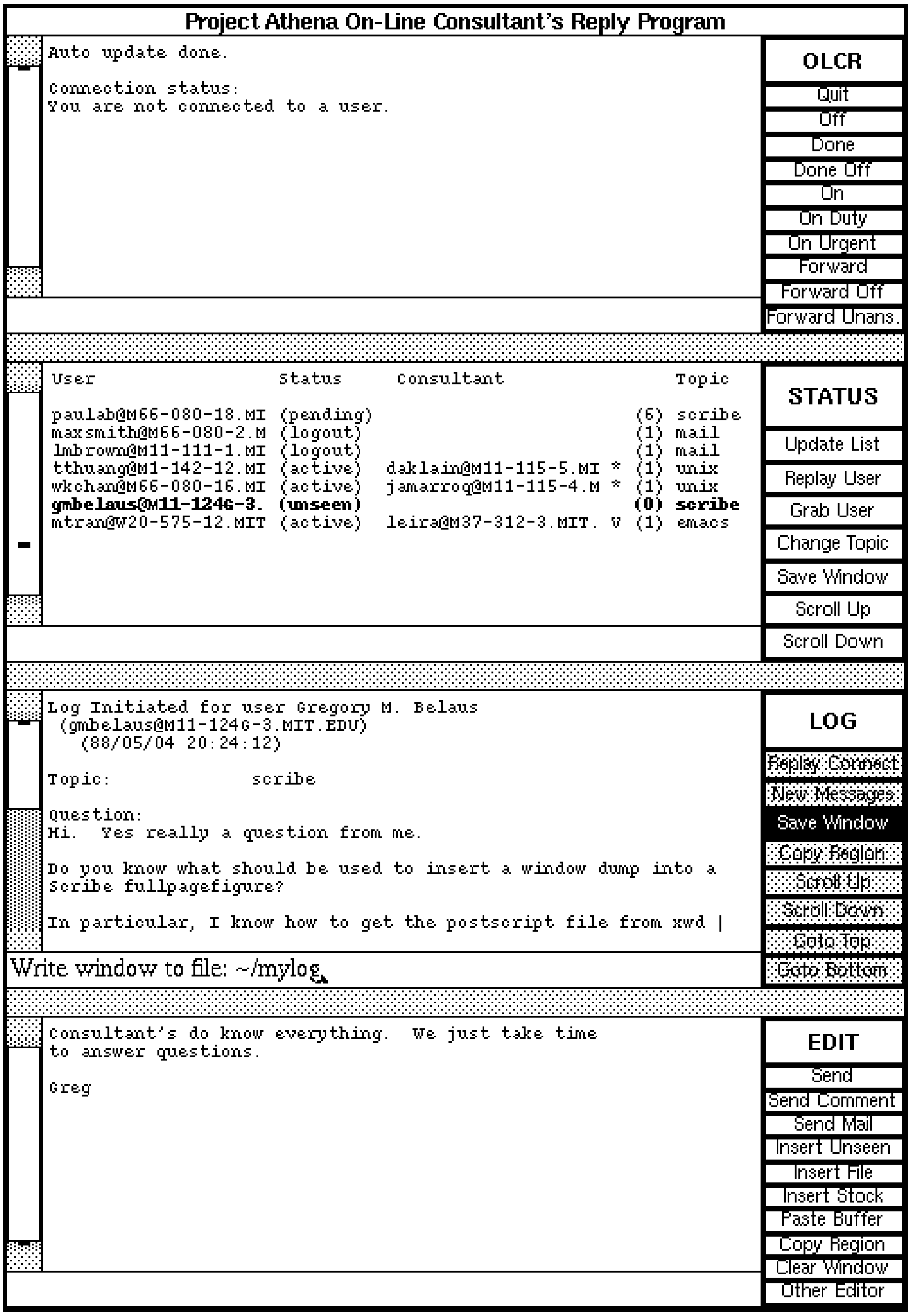

Figure 2-2: The OLCR Application Window

or a TITLE. The *olc* client shows all buttons ENABLED, except the TITLE buttons ’OLC’, ’EDIT’, and ’LOG’. Each of these TITLE buttons is at the top of a separate buttons objects. Figure 2-2 shows that the *olcr* client has many buttons in a different state than ENABLED: the ’Save Window’ button is SELECTED (in the LOG portion of the window), while the ’Replay Connect’, ’New Messages’, ’Copy Region’, ’Scroll Up’, ’Scroll Down’, ’Goto Top’, and ’Goto Bottom’ buttons are DISABLED. The ’Quit’ button is ALWAYSENABLED for both applications, meaning that it never changes from the ENABLED state.

Like the titleBar object, the buttons object is a subclass of view. All buttons are given equal height, except for TITLE buttons, which are given double the height of all other buttons. It does not matter what is the size of the visual rectangle for a buttons object, as the individual buttons will shrink or expand to fit the available region. Each button can be assigned a callback procedure that will be called when the left or right mouse button is pressed while the mouse pointer is in the button’s rectangle. A reference object is sent along with this call, as well as the text of the button. Only ENABLED and ALWAYSENABLED buttons will have their callback procedure called when pressed. The pressed button will invert, i.e. be SELECTED, until the called procedure returns.

The Class header file and C source code for the buttons object can be found in Appendix B. This serves as an an example of an object containing state information, dynamically allocated data, and a method for handling mouse events. The buttons object is a linked-list of button structures, affording a single object the ability to contain an arbitrary amount of information.

2.1.1.3 EditWindow and PanelWindow

The editWindow and panelWindow objects form the large-scale building blocks of the window-based interface to the OLC clients. Both of these objects are the same, except for one detail. The *olc* client continuously displays three instances of these objects (three editWindows, or two editWindows and a panelWindow), while the *olcr* client displays four (at least two of which are always editWindows). The object instances are arranged vertically, on top of each other, giving the characteristic subwindow effect of the visual display of the clients.

The editWindow object is a subclass of lpair. This lpair is a vertical split between a titleBar object (on top) and another lpair (on the bottom) . The second lpair is a horizontal split between a frame (on the left) and a buttons object (on the right). The frame, as stated above, is a subclass of the lpair object, with a vertical split between a view (on top) and a message object (on the bottom). This view is a textview object, containing a scrollbar (on the left) and an associated text object (its dataObject). The (sub)windows associated with the TITLE buttons OLC, EDIT, LOG, and OLCR are editWindows.

The panelWindow object is similar to the editWindow object. The only difference is that the frame contains a panel[6] (on top) instead of a textview. A panel is a subclass of textview. Like the buttons object, it has the provision for the context-associated execution of procedures. The panels are the lines of text displayed. Pressing the left or right mouse button with the mouse on one of these lines of text executes an associated callback procedure. Like a button, this region is highlighted after being selected (although this highlight does not disappear after the procedure returns). The STATUS window of the OLCR object, as well as the TOPIC windows of both clients, are panelWindows.

The TOPIC windows are not normally displayed be either of the clients, but replace part of the display window when their function is required. Figure 2-3 shows the TOPIC window being displayed in place of the EDIT window.

---

[6]The panel object is not a documented object in the Andrew Toolkit. It has been copied from the source to the hz program, and modified slightly.

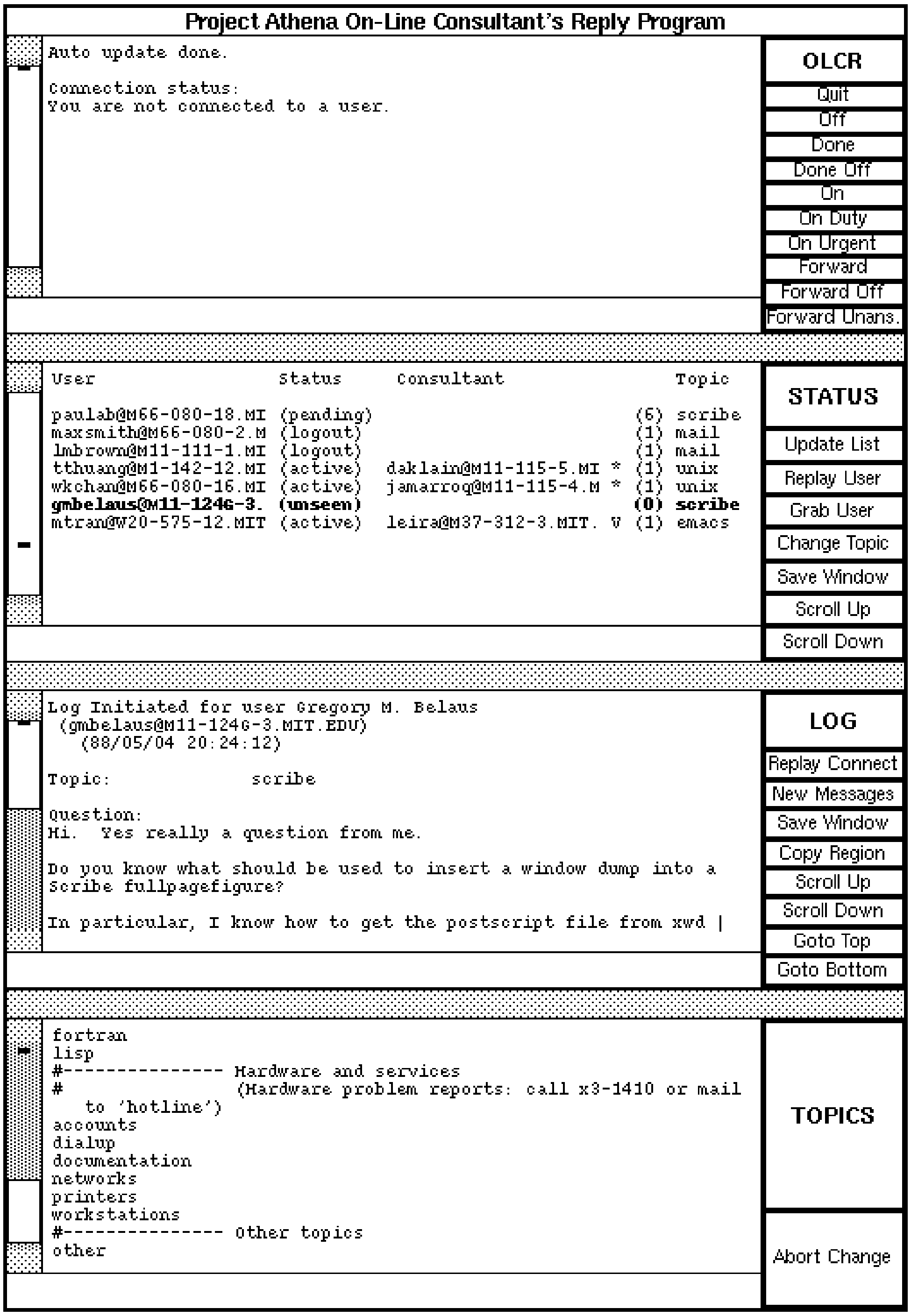

Figure 2-3: The OLCR Application Window: Topic Window Displayed

2.1.1.4 OlcWindow and OlcrWindow

The olcWindow and olcrWindow objects represent the display portion of the window-based clients shown in Figures 2-1, 2-2, and 2-3. These subclasses of lpair combine the editWindows and panelWindows appropriately, setup the text of the buttons as well as the callback procedures, and provide the methods for displaying and removing the topic window. They are the final glue that combines the toolkit objects with the custom created objects.

2.1.2 The OLC System Code

The transformation of the user-interface to the OLC clients *olc* and *olcr* required minor changes to the OLC system code, which consists of three parts:

1. **The Front-End User Interface**

   As stated, this interface is transformed from a command-line parser to a window-based system consisting of editWindows and panelWindows. The front-end code to a client sets up the interaction manager with default parameters, opens the window, and displays the proper object, either an olcWindow or olcrWindow. The *olc* client leads a user who is currently not asking a question through the selection of a topic and the input of an initial question.

   Both clients eventually enter a main loop that continuously monitors the state of both the OLC system and the user-interface. In this *main loop*, interaction manager events are processed until an interrupting event occurs. Pressing a button or panel jumps the interface to a callback routine. In this routine, the display is usually changed to limit the actions of the user and the proper command is processed. Control eventually returns to the *main loop*.

2. **Command Processing**

Minor modifications were required to all command procedures in the transformation from the command-line to the window-based system. These modifications all occurred at the user-interface level. The primary source of the changes is the extensive use of the top window in each client as a message area. Care was taken to make sure that the user is always aware of what is happening, and what he is expected to do (or can do) next. The interaction manager event queue is often manually flushed inside a command procedure so that information can be displayed at different stages of the execution of the command. This is particularly relevant in routines that require user input from the message line in each frame.

3. **The Transport Mechanism**

   The underlying transport mechanism of the OLC clients remains unchanged through the transformation of the user-interface. This allows for the simultaneous use of the new clients and the old ones, with both *talking* to the same daemon program on the OLC server machine and both usable as working clients.

2.2 Dividing the Real Estate: Iteration of Design

The creation of a window-base user-interface requires careful consideration of the visual presentation. The establishment of visual goals on paper serves as a first step in this consideration. However, "...the development of a set of design specifications from starting objectives must be a highly iterative process." [Carroll and Rosson 85, p. 15] Not until the first time the completed visual presentation appears on the screen can a final goal be pictured. Tweaking of the overall presentation and division of window real-estate to the final satisfaction of both programmer and user becomes a formidable task.

The transformation of the OLC client program user-interface required this same iteration of design. "Early interface testing need not be confined to paper and pencil tasks. ...for instance, the process of moving through a menu tree could be simulated quite easily in advance of a working prototype." [Carroll and Rosson 85, p. 24] Figure 2-4 shows an early attempt at the visual design of the *olc* client, sporting such problems as overly cluttered buttons, and textviews stacked on top of each other without intervening titleBars. This method further supports the bottom-up implementation presented above, without circumventing the concept of nonhierarchical design.

The final design of the OLC clients provides a common interface among the subwindows: buttons on the right and scrollbars on the left. The message line in the frames of the editWindows and panelWindows are used only for displaying/receiving transient information about the status of the subwindow, e.g. a query for a file name for a 'Save Window' command. The top editWindow is reserved for program and system status, as well as connection messages.

2.3 Controlling the State in an Event-Driven System

The Andrew Toolkit is event driven. This places limitations on the scope of the user-interface as defined by what events are generated. Although the supporting window system, the X Window System[7] supports the type of events desired for a particular part of the user-interface, the interaction manager may obscure these events from the programmer through its maintenance of the object hierarchy. Also, the user-interface requires that the event queue is manually flushed whenever it needs to guarantee that a user has seen particular information before continuing on for further processing.

Consider the buttons object. Implemented as a view with rectangles and text drawn

---

[7]The X Window System, Version 11 Release 1, © 1988 Massachusetts Institute of Technology

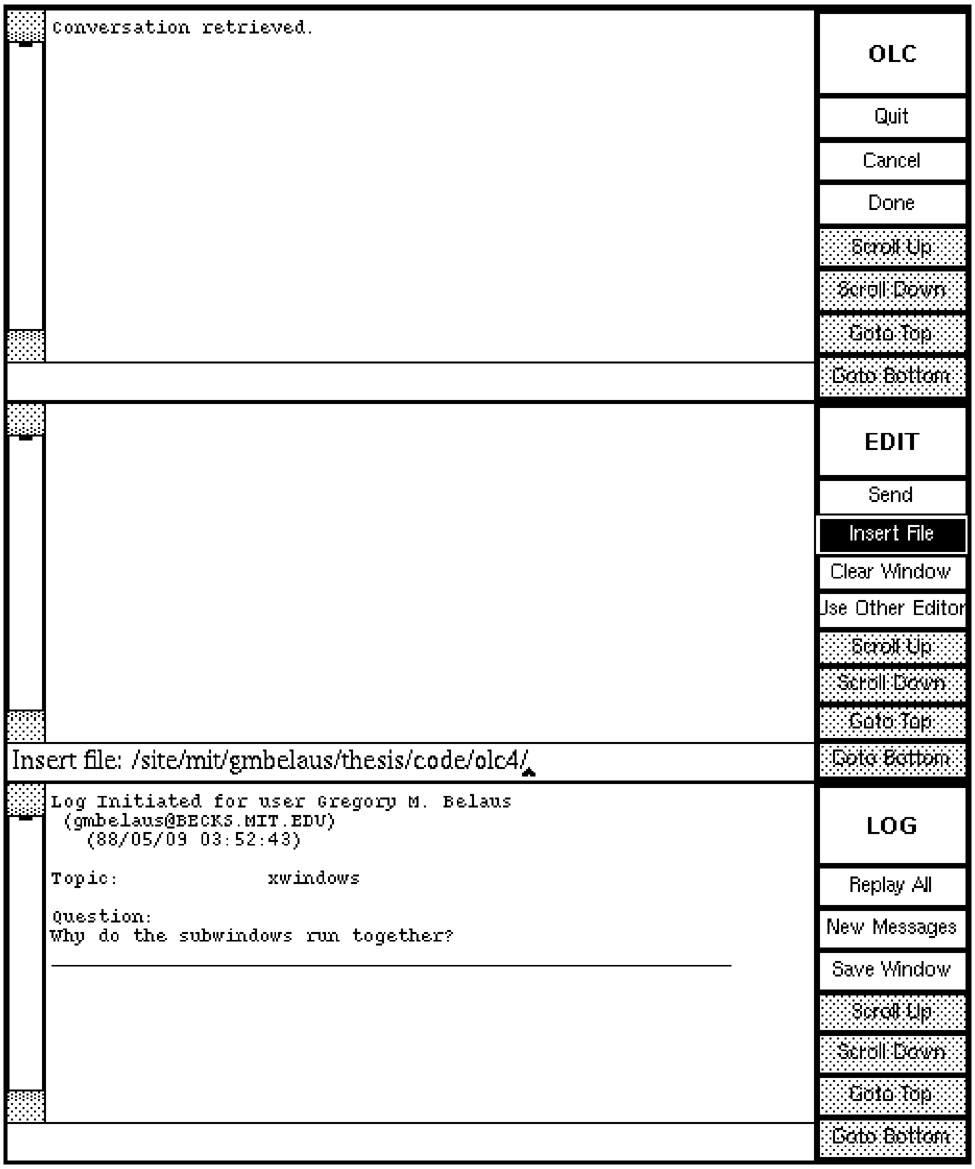

Figure 2-4: Early Design of the OLC Application Window

in, it is impossible to tell which button the mouse is currently pointing to without the press of a mouse button. This information would be useful in adding to the user-interface the feature that the button that the mouse is pointing to is always highlighted. The toolkit only provides mouse events when there is a transition in a mouse button. The X Window System will provide events whenever a subwindow is entered or exited. It is obvious that the Andrew Toolkit is taking advantage of these window system events for its control of cursors.[8] However, because of the layers of abstraction built into this object-based toolkit, it is impossible to extract this same function built-in to the cursor object without deeply exploring the cursor object source code.

The Andrew Toolkit event queue needs to be manually flushed whenever it is required that the user be aware of changes to the state of the system before the program continues the execution of a command. This provides a burden on the interface designer to always be aware of what is being displayed in the client window while programming the system. This is particularly troublesome in code that is entered from various parts of the system. Enough state information must always be known so that the manual flushing of the event queue occurs at the most opportune time.

2.4 Working with the Users

The end-user of a user-interface should be consulted for aid in its design. The designer has limited ideas on the overall function of any user-interface. He may be biased, or only represent a certain class of users. Those who are to use the final product should be sought out. Interaction with the end-user should extend from the beginning of the design phase through the completion and testing of the final implementation. [Carroll and Rosson 85, pp. 14-15] Besides providing alternative views in the design, the end-user may also be able to break the system through actions that the designer did not anticipate.

[8]The Andrew Toolkit allows for the creation of view specific mouse cursors. The mouse cursor changes appropriately when one view is exited and another is entered.

The transformation of the user-interface of the OLC clients represents a radical change in the manner in which the clients will be used. State information is kept track of and updated between command interactions, at the expense of speed. Feedback from users of the new system varies. Most users of the new *olc* client welcome the new interface and are not bothered by the degradation in performance. Because of their occasional use of this client, they are inclined to believe that the new interface makes the entire OLC system easier to use, and their time more productive. Thus, these *average* users prefer the button-driven visual interface.

Users of the new *olcr* program are more critical of the user-interface. Most comments from these users focus on the performance degradation from the original command-line interface. The bottom line is that although the system provides them with more continuous information, it is faster for them to retrieve the information that they *need* by using the old interface. In contrast with the *average* user, these *experts* prefer the speed and flexibility of the command-line interface.

2.5 Speed

Finally, the speed with which the toolkit can change the client display window plays a part in the actual design of the interface.

> ... systems should be designed to respond to the user within about two seconds for 80-90% of all requests. For other requests, the program should provide some intermediate output indicating that the system is working and that some progress is being made on the user request. [Wasserman 81, p. 379]

Care must be taken that interface enhancements do not degrade from the performance of the client such that updating the user-interface display takes more time than completing the normal tasks of the clients. Users of a program slowed considerably by the user-interface will cry out for a simpler interface, even if it requires more work on their part to use.

The Andrew Toolkit relies heavily on an intricate system of pop-up menus. These menus are associated with the context of the mouse at the time they are invoked, allowing for the dynamic use of new objects that bring with them associated tools for self-modification. Because of the slow speed with which these menus appear (when the middle mouse button is pressed)[9], all pop-up menus have been turned off in the OLC clients interfaces. Their function has been completely replaced by the buttons objects.

[9]Some members of the Project Athena System Development staff suggest that a rework of the Andrew Toolkit interface to the X Window System is required to hasten the mapping of menus to the display.

Chapter 3

Analysis of the Andrew Toolkit

The Andrew Toolkit provides an excellent array of tools available in an environment conducive to the production of a well-designed user-interface. The tools are the hierarchical Class system and the objects available for use within it. New custom objects can be created through simple modifications to existing objects.

However, there are a few things to consider when programming with the toolkit:[10]

3.1 The Extensive Expansive Toolkit

> The Toolkit provides the usual set of simple components (menu, scroll bars, etc) and several higher-level editable components including multi-font text, tables/spreadsheets, drawings, equations, rasters and simple animations. The text and table components are multi-media components that allow the embedding of other components within themselves. [Overview 88, p. 3]

Thus, the Andrew Toolkit is an extensive collection of objects and support tools. This collection does not come without price. Toolkit libraries, and the applications created with them, tend to be large. The basic libraries are comprised of megabytes worth of information, making it difficult to use debugging tools and interpreters in the process of application development. The OLC clients approach 500Kbytes as optimized and stripped executables, an average increase of almost 450Kbytes per client. This increase is not an anomaly to the Andrew Toolkit, but is in accord with figures from similar uses of comparable programming environments. "The moral is clear. Iconic interfaces do need lots of memory." [Cox 86, p. 195]

---

[10]All information in this chapter is based entirely on what information was available at the time of this writing, using the documentation distributed with the toolkit. No attempt has been made to thoroughly research the source code or contact those who have designed the toolkit.

To aid this last concern, a run-time system called *runapp* allows for text sharing of most of the common objects found in the toolkit between multiple applications. [Overview 88, p. 4] Although 500Kbytes large, this program allows compiled executables to be small and text-share many of the toolkit objects in the virtual memory system of the computer. This method reduces virtual memory usage and swapping on systems running multiple toolkit applications. However, with only one toolkit application running at a time, as was the case in the test use of the new OLC clients, no gains in virtual memory usage are realized.

3.2 The Object Class Hierarchy

An object-oriented hierarchical toolkit provides an excellent match to the task of user interface design.

> An object-oriented programming language encapsulates data with the operations on that data inside a well-defined tangible interface, the object. An iconic user interface uses the same principle with the operations that can be applied to the data manages in an application. The same principle that reduces the surface area between programmers serves to decrease the amount that a novice must learn to use an application.... Iconic user interfaces are far more complex than command-oriented interfaces, making it more important that ever to control complexity, provide reusability, and encapsulate change. [Cox 86, pp. 157-158]

Although the Andrew Toolkit is object-oriented, it fails to control complexity and encapsulate object data.

Uncontrolled data and method inheritance in the Andrew Toolkit increases the complexity of bookkeeping in the creation of an application. The class hierarchy of the toolkit provides a single inheritance model object-oriented system for the creation of a user-interface. Instantiations of objects inherit all the data and methods associated with its own object type, as well as that of its superclass. This superclass has inherited the data and methods from its superclass as well, and thus the instantiation receives the data and methods of its ancestors.[11] This large quantity of inherited data and methods becomes difficult to to keep track of, making bookkeeping a programming chore.

[11]ancestors = the classes in the path from the current class back to the top of the tree

The text object, used throughout the OLC clients, provides a simple example of the increased bookkeeping of method inheritance. This object provides no direct self-modifying method for the insertion of characters. However, simpletext, its superclass object, does provide a method for doing this. To use the inherited method, *InsertCharacters*, requires knowing about the simpletext object, reading the documentation about this object, and possibly analyzing its source code. The extra bookkeeping of knowing the details of the text object's superclass adds to the complexity of programming with this object.

> Encapsulation is the foundation of the whole approach [to object-oriented programming]. Its contribution is restricting the effects of change by placing a wall of code around each piece of data. All access to the data is handled by procedures that were put there to mediate access to the data. [Cox 86, p. 9]

The Andrew object class hierarchy violates this principle by providing unrestricted scoping of class data. Object designers are not forced to provide convenient access to all object data, although the low-level implementation of the class provides paths to it. This allows for the the creation of objects that are difficult to port to applications not thought of by the original designer.

The lack of complete encapsulation of class data made the panels object difficult to port to its application in the panelWindow object. The panelWindow object needs to make changes to the appearance of the panel text in both of the OLC clients. Changing the text of a panel from bold-type to plain-type requires access to the text object embedded within the panels object. Because the designer of the panels object did not provide a method that returns a pointer to this text object, a complex structure path had to be traced through the underlying implementation of the panels to find and use the text. Thus, a required use of the panels by the panelWindow object was burdened by the lack of complete data encapsulation.

3.3 The X Window System Interface

The Andrew Toolkit is arranged such that all toolkit events are appended to the global interaction manager event queue. When an event at the top of the queue that requires interaction with the X Window System is processed by the interaction manager, the proper *X* events are sent to its own event queue.

This queuing mechanism, at times, makes it difficult to guarantee that a particular event has been displayed. Even though it is known that the event is no longer in the interaction manager's queue, it may not have been processed by *X* yet. A simple example of this problem involves the opening of the original application window. Often the programmer will desire to know at what point the window is mapped to the display and is ready to receive further display instructions, so that a start-up screen can be displayed before the program continues on for further processing. Because the two queues are asynchronously linked, it is impossible to tell when this mapping has occurred.

A final comment on the interaction of the Andrew Toolkit with the X Window System involves the speed and method with which update and exposure events are handled. At present, Andrew Toolkit developers at CMU are debating whether the code for this interaction needs to be rewritten because it is slow. It appears that there is also a problem with exposure events being passed down to toolkit objects under certain conditions using an overlapping window manager.

3.4 Documentation

Documentation should be the primary interface between the programmer and the Andrew Toolkit. Hierarchically arranged objects in such a system should be arranged with an opaque box surrounding them. Documentation, thus, would be the writing on the box explaining the possible interfaces to that objects. Unfortunately, the most extensive programming documentation on the toolkit, [Reference 88], fails in this respect. The programmer is required to analyze the source code and Class header file for an object before being able to understand what the object's limitations are and if his needs require it to be subclassed. Sometimes this extra work is even required to simply understand how to use an object method. The following are two examples of these problems:

- Like Smalltalk-80[12], the Andrew Toolkit does not manage rectangles and points as objects, but as C structures. The toolkit documentation clearly states this in the graphic object section, but fails to delineate the associated fields of a rectangle or point. Thus, the user must search the appropriate object source code to find the definitions of these structures.

- The graphic object provides methods to draw and fill rounded rectangles. As part of their formal parameters, each requires pointers to two rectangle structures, *innerRect* and *outerRect* to be passed to them. However, there is no indication how a rounded rectangle is formed from these two rectangles. After examining the source code for the graphic object, it becomes apparent that the *innerRect* provides a radius for corner arcs within the *outerRect* and that only two of its four rectangle parameters are even used.

Examples like these abound and are not limited to the graphic object. For instance, distance parameters in the context object used for creating text margins are specified as long-int parameters. In reality, "The constant passed into context calculations is a fixed point number with the binary point 16 bits in, e.g., the 32 bit value is 16 bits of whole number followed by 16 bits of fractional number."[13] Unfortunately, small (petty) examples like this make it difficult to use the Andrew Toolkit as intended, without looking inside.

---

[12] "...an integrated programming language and programming environment" [Cox 86, p. 30] developed at Xerox PARC

[13] response received via e-mail from CMU on this problem

Chapter 4

Conclusion

Through the transformation of the command-line interface of the OLC client programs, *olc* and *olcr*, to an intuitive window-based system that supports buttons, panels, and scrollbars, the Andrew Toolkit is analyzed in the context of the creation of a user-interface. This transformation was an iterative process of simultaneous programming and design. Analysis of the toolkit occurs in the context of this process.

Iteration of design was required to determine which objects in the toolkit were needed for this interface, and which were to be custom made. Although the design process was nonhierarchical, the programming of the interface occurred in bottom-up procession. Application specific objects were created by subclassing toolkit objects. The final interface is the amalgamation of these custom objects with the OLC system code.

In general, an object-oriented programming interface is a good match for the task of creating a user-interface. However, the characteristics of the Andrew Toolkit and the end-users of the clients determined the actual design constraints. Strictly speaking, the toolkit is not a true object-oriented programming interface because it does not provide complete data encapsulation, and the required bookkeeping of inherited data and methods adds greatly to programming complexity. The lack of thorough documentation further complicates the lack of data encapsulation by often making it necessary for the programmer to examine source code to understand the use of object data and methods.

The final product of the design process, the transformed OLC clients, provides a valuable example of the creation of a user-interface with the Andrew Toolkit. The new window-based interface to these programs can be tested in parallel with the command-line interface of the old clients, using the same OLC daemon. Feedback from users of both systems suggest that *average* users of the *olc* program prefer the new interface. They are undaunted by the decrease in general performance of the clients from the old interface to the new. Because the new interface is easier to use, they feel that their time is spent more productively in the use of the new *olc* client. For this class of users, and hence this class of applications, the creation of window-based *olc* clients is worth the increased design effort because the final products are easier to use.

Users of the *olcr* client, the *experts*, are less satisfied with the new interface. They are more affected by the performance degradation associated with the transformation to the window-based interface. The speed and flexibility of the command-line interface better serves their needs. For this class of user, the creation of a window-based client is not worth the extra design effort unless the tools supporting this transformation are suitably efficient. Thus, the creation of window-based interfaces to be used by *experts* is a more difficult task than the creation of similar interfaces for *average* users of non-*expert* applications. The Andrew Toolkit does not perform well in this task.

Despite several drawbacks, the Andrew Toolkit provides the necessary components for the creation of a working user-interface to the OLC clients. The design process requires substantial iteration. Through feedback from the users of both the new and old clients, it is determined that the class of user corresponds to the type of interface desired. *Average* users prefer a window-based interface, and are willing to ignore the associated decrease in performance. Therefore, the Andrew Toolkit is well suited for applications to be used by this class of user. *Experts* desire a more efficient interface, like the command line. The Andrew Toolkit is unable to to provide the efficiency desired by these users for a window-based interface.

Appendix A
The Andrew Toolkit Object System

"The Andrew Toolkit is based on the development of components that can be used as building blocks for applications or other, more complex components. These components are frequently referred to as objects." [Overview 88, p. 5] The transformation of the user-interface of the client programs to a window-based interface requires the careful use of existing objects in the toolkit, as well as the creation of new objects. All objects are arranged in a tree-like hierarchy. Objects further down the tree, towards the leaves, are called *subclasses* of the object directly above them. In turn, the object directly above an object in the hierarchy is an object's *superclass*. An object inherits all the data and methods from its superclass object, which in turn has inherited all the data and methods from its superclass object as well.

The actual working copy of an object is called an *instance*. Thus you can have more than one instance of a particular object, and therefore have more than one object of any particular type to work with. For example, there can be more than one text editor object displayed on the screen. Work on text in each of these editors editors can occur simultaneously, with different text being edited in each object.

A large quantity of object types built-in to the Andrew Toolkit will be used in most Toolkit applications. However, most of these objects are commonly used indirectly as they are incorporated into high-level objects such as panels or editors. The following sections describe several objects that are used directly in the creation of the custom objects for the window-based interface to OLC.

A.1 The Interaction Manager

"The interaction manager handles all the 'administrative' duties in the event-driven Andrew Toolkit system." [Reference 88, p. 156] It is the highest level object in the Andrew Toolkit application object hierarchy. The interaction manager is the only object that *talks* directly to the window-system that is supporting the toolkit. Each top-level[14] window in an application has its own interaction manager.

The interaction manager has many class procedures and methods for dealing with the toolkit event-queue and the supporting window manager. There are provisions for handling one queue event at a time, as well as flushing the entire queue. Many global defaults are set with the interaction manager, including window dimensions, the application name for the window manager, and the default cursor. Window system cut buffer interactions are controlled from this object as well.

A.2 View and DataObject

"Two of the most important object types within the Andrew Toolkit are dataObjects and views.... The dataObject contains the information that is to be displayed, while the view provides display and editing features (a user interface) suited to that type of dataObject." [Overview 88, p. 5] These objects are the cornerstones of any application developed with the toolkit.

The top-level view of the toolkit is the bridge between the application and the window created by the associated window-system.[15]. Within this view, the application

---

[14]top-level = child of the root window of the physical display

[15]Multiple windows can be created by an application through calls to the host window system. Each window created has its own top-level view associated with it, either another instance of the view object or subclass or the same view shared with another window.

presents itself. Subviews (other instances of the view object or its subclasses) are used to partition a window into working regions.

DataObject and its subclasses represent the information that is displayed in views. The text associated with a message line is a dataObject. The internal representation of a graphic image is a dataObject as well. Views are used to look at a particular dataObject. Most often, views and dataObjects are referred to together as pairs.

A.3 Lpair and Frame

The lpair object is a subclass of view; the frame object is a subclass of lpair. Both of these object types are used to organize the visual display within a window. An lpair, a layout pair, is a view that wholly incorporates two subviews. The two views can be arranged either horizontally or vertically. The split between the two subviews can be determined as a percentage of the size of lpair, or at a fixed quantity of display pixels. This split can be user-movable or fixed.

The frame object is the combination of a subview and a message line. The message line, used for displaying simple one-line messages and for short information queries, is placed vertically below the subview. Its size is a fixed quantity of pixels.

A.4 Text and Textview

The text and textview objects are a specific case of the dataObject/view pair discussed above. The text object is an internal representation of text characters. It provides methods for the insertion and deletion of characters, as well as for writing/reading portions of the text to/from a file. Text may be formatted [see next section]. Other objects, as well as views, can be inserted into a text object. This allows for the creation of multi-purpose editors that allow the incorporation of text, graphics, tables, and animations.

The textview object is a subclass of view. It is a specialized object designed for the display of text objects. Formatted text is displayed in semi-WYSIWYG fashion[16], including the display of other objects that have been inserted into the text. GNUEmacs[17] style editing is the default, although there is a provision for the creation of custom keymaps. Textview supports a text cursor as well as a method for selecting regions of text with the mouse. When incorporated into an application, textview supports a full-function scrollbar mechanism by default.

A.5 Formatting the Text: Stylesheet, Style, Device, and Context

The four toolkit objects, stylesheet, style, device, and context, provide the working set required to apply formatting to text. Although seemingly cumbersome to work with, these objects are arranged such that maximum application portability can be achieved with the least amount of effort. "A stylesheet is a collection of the styles which have been defined for use in a particular object.... A style is a collection of attributes which can be applied to a region of data." [Reference 88, p. 189] Device and context provide portability. A style can appear differently when displayed on a different device or found in another context within a data object.

---

[16]Currently, there is no attempt to display a text object exactly as it will appear on a printed page.

[17]© 1988 Free Software Foundation

Appendix B

Class Header File and C Source Code for Buttons Object

buttons.H

```
#include "rectangle.h"

                                /* entry types */
#define ENABLED 1
#define DISABLED 2
#define TITLE 3
#define SELECTED 4
#define ALWAYSENABLED 5

                                /* button definition */
struct button {
  struct button *next;          /* linked list of buttons */
  char name[50];                /* it’s display name and call argument*/
  int type;                     /* type of entry */
  struct point nameCurPoint;    /* current point for string drawing */
  struct rectangle buttonRect;  /* rectangle for button */
  int (*callbackProcedure) ();  /* callback procedure */
  char *referencePointer;       /* pointer to reference object, for callback */
};

class buttons : view {
 overrides:
  FullUpdate(enum view_UpdateType, long left, long top, long width, long right);
  Hit(enum view_MouseAction action, long x, long y,
      long numberOfClicks) returns struct view *;
 methods:
  AddButton(char *name, int type,
            char *callbackProcedure, char *referencePointer) returns boolean;
  DeleteButton(char *name) returns boolean;
  SetType(char *name, int type) returns boolean;
  EnableAllButSelected() returns boolean;
  DisableAllButSelected() returns boolean;
  SetEntryFont(struct fontdesc *font) returns boolean;
  SetTitleFont(struct fontdesc *font) returns boolean;
 classprocedures:
  InitializeObject(struct buttons *bts) returns boolean;
  FinalizeObject(struct buttons *bts) returns boolean;
 data:
  int qtyButtons;               /* number of buttons */
  int qtyTitleButtons;          /* number of title buttons */
  struct button *buttons;       /* first button */
  struct fontdesc *entryFont;   /* font for normal entries */
  struct fontdesc *titleFont;   /* font for title entries */
  struct pixmap *enabledPixmap; /* white pixmap for enabled entries */
  struct pixmap *selectedPixmap;/* black pixmap for selected entries */
  struct pixmap *disabledPixmap;/* grey pixmap for disabled entries */
};
```

buttons.c

```
#include <class.h>

#include "buttons.eh"
#include "graphic.ih"
#include "fontdesc.ih"

#include "rectangle.h"

                                /* CLASS PROCEDURES */
boolean buttons__InitializeObject(classID, bts)
     struct classheader *classID;
     struct buttons *bts;
{
 bts->qtyButtons = 0;           /* no buttons */
 bts->qtyTitleButtons = 0;      /* obviously, no buttons with titles */
 bts->buttons = (struct button *) 0; /* no first button */
 bts->entryFont = fontdesc_Create("andysans", fontdesc_Plain, 8);
 bts->titleFont = fontdesc_Create("andysans", fontdesc_Bold, 10);
 bts->enabledPixmap = NULL;     /* null for now, set in FullUpdate */
 bts->selectedPixmap = NULL;    /* ditto */
 bts->disabledPixmap = NULL;    /* ditto */


 return TRUE;
}

boolean buttons__FinalizeObject(classID, bts)
     struct classheader *classID;
     struct buttons *bts;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {
    nextButton=thisButton->next;
    free(thisButton);
  }

  return TRUE;
}

                                /* OVERRIDES */
void buttons__FullUpdate(bts, type, argleft, argtop, argwidth, argheight)
     struct buttons *bts;
     enum view_UpdateType type;
     long argleft;
     long argtop;
     long argwidth;
     long argheight;
{
  struct button *thisButton, *nextButton;
  struct rectangle thisVisRect;
  struct rectangle roundedButton;
  long top;
  long left;
  long height;
  long width;
  long delta;                   /* height of a non-title rectangle */
  float realDelta;              /* real (float) delta */
  long currentY = 0;            /* current y that we're at */
  float realY = 0;              /* real y we should be at */

  long heightFudge = 0;         /* fudge factor for next time around */
```

```
                              /* can't set pixmaps in InitializeObject */
bts->enabledPixmap = (struct pixmap *) buttons_WhitePattern(bts);
bts->selectedPixmap = (struct pixmap *) buttons_BlackPattern(bts);
bts->disabledPixmap = (struct pixmap *) buttons_GrayPattern(bts, 1, 7);


buttons_SetTransferMode(bts, graphic_COPY);

buttons_SetLineWidth(bts, 3);

buttons_GetVisualBounds(bts, &thisVisRect);
top = rectangle_Top(&thisVisRect);
left = rectangle_Left(&thisVisRect);
height = rectangle_Height(&thisVisRect) - 1; /* - 1 to comp. for linewidth */
width = rectangle_Width(&thisVisRect);

delta = (long) (realDelta = (float) height/(bts->qtyButtons
                                     + bts->qtyTitleButtons));

for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {

  nextButton=thisButton->next;

  thisButton->buttonRect.top = top + currentY;
  thisButton->buttonRect.left = left;
  thisButton->buttonRect.width = width;

  if (thisButton->type == TITLE) {
    if (thisButton->next == NULL) /* force last rectangle to line up */
      thisButton->buttonRect.height = top + height - currentY;
    else thisButton->buttonRect.height = delta * 2 + heightFudge;
    thisButton->nameCurPoint.y = top + currentY + delta + heightFudge;
    currentY = currentY + delta * 2 + heightFudge; /* y for next button */
    realY = realY + realDelta * 2; /* real y for next button */
    buttons_SetFont(bts, bts->titleFont);
    thisButton->nameCurPoint.x = left + width/2; /* string in center */
    buttons_MoveToPt(bts, &thisButton->nameCurPoint);
    buttons_FillRect(bts, &thisButton->buttonRect, bts->enabledPixmap);
    buttons_DrawString(bts, thisButton->name,
                       graphic_BETWEENTOPANDBASELINE |
                       graphic_BETWEENLEFTANDRIGHT);
    buttons_DrawRect(bts, &thisButton->buttonRect);
  }
  else {
    if (thisButton->next == NULL) /* force last rectangle to line up */
      thisButton->buttonRect.height = top + height - currentY;
    else thisButton->buttonRect.height = delta + heightFudge;
    thisButton->nameCurPoint.y = top + currentY + delta/2 + heightFudge/2;
    currentY = currentY + delta + heightFudge; /* y for next button */
    realY = realY + realDelta; /* real y for next button */
    buttons_SetFont(bts, bts->entryFont);
    thisButton->nameCurPoint.x = left + width/2; /* string in center */
    buttons_MoveToPt(bts, &thisButton->nameCurPoint);
    if (thisButton->type != SELECTED) {
      buttons_FillRect(bts, &thisButton->buttonRect, bts->enabledPixmap);
      buttons_DrawString(bts, thisButton->name,
                         graphic_BETWEENTOPANDBASELINE |
                         graphic_BETWEENLEFTANDRIGHT);
    }
    buttons_DrawRect(bts, &thisButton->buttonRect);
  }

                              /* fudge factor to counter rounding */
  heightFudge = (int) (realY - ((float) currentY) + .5);

  switch (thisButton->type) {
```

```
    case ENABLED:               /* leave here, in case we want rounded ones */
      break;
    case TITLE:
      break;
    case DISABLED:
      buttons_SetTransferMode(bts, graphic_OR);
      buttons_FillRect(bts, &thisButton->buttonRect, bts->disabledPixmap);
      buttons_SetTransferMode(bts, graphic_COPY);
      break;
    case SELECTED:              /* shouldn't get here */
      break;
    case ALWAYSENABLED:
      break;
    }

  }
}

struct view *buttons__Hit(bts, action, x, y, numberOfClicks)
     struct buttons *bts;
     enum view_MouseAction action;
     long x;
     long y;
     long numberOfClicks;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {

    nextButton=thisButton->next;

    if ((thisButton->buttonRect.top <= y) &&
        ((thisButton->buttonRect.top + thisButton->buttonRect.height) > y) &&
        (thisButton->buttonRect.left <= x) &&
        ((thisButton->buttonRect.left + thisButton->buttonRect.width) > x) &&
        ((action == view_LeftDown) || (action == view_RightDown)))
      break;
  }

  if ((!thisButton)
      || (thisButton->type == TITLE) || (thisButton->type == DISABLED))
    return (struct view *) 0;

                              /* invert it's rectangle */
  buttons_SetTransferMode(bts, graphic_INVERT);
  buttons_FillRect(bts, &thisButton->buttonRect, bts->selectedPixmap);
  buttons_FlushGraphics(bts);
  thisButton->type = SELECTED;

                              /* call procedure here */
  if (thisButton->callbackProcedure)
    (*thisButton->callbackProcedure) (thisButton->name,
                                      thisButton->referencePointer);

                              /* reinvert rectangle */
  buttons_SetTransferMode(bts, graphic_INVERT); /* just in case... */
  buttons_FillRect(bts, &thisButton->buttonRect, bts->selectedPixmap);
  buttons_SetTransferMode(bts, graphic_COPY);
  buttons_DrawRect(bts, &thisButton->buttonRect);
  thisButton->type = ENABLED;
  return (struct view *) 0;

}

                              /* METHODS */
boolean buttons__AddButton(bts, name, type, callbackProcedure, referencePointer)
```

```
     struct buttons *bts;
     char *name;
     int type;
     int (*callbackProcedure) ();
     char *referencePointer;
{
  struct button *thisButton, *nextButton;
  struct button *buttonToAdd;

  buttonToAdd = (struct button *) malloc(sizeof(struct button));
  buttonToAdd->next = NULL;
  strcpy(buttonToAdd->name, name);
  buttonToAdd->type = type;
  buttonToAdd->callbackProcedure = callbackProcedure;
  buttonToAdd->referencePointer = referencePointer;

  if ((thisButton=bts->buttons) == NULL)
    bts->buttons = buttonToAdd; /* if this is #1, add to link to front */
  else {
    while(thisButton->next) {   /* easier to think of as while */
      nextButton=thisButton->next;
      thisButton=nextButton;
    }
    thisButton->next = buttonToAdd; /* not first button, add to end */
  }

  bts->qtyButtons++;
  if (type == TITLE)
    bts->qtyTitleButtons++;

  return TRUE;
}

boolean buttons__DeleteButton(bts, name)
     struct buttons *bts;
     char *name;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {
    nextButton=thisButton->next;

    if (!strcmp(nextButton->name,name))
      break;
  }

  if (!thisButton)
    return FALSE;              /* didn't find it */

  thisButton->next = nextButton->next; /* link out entry */
  free(thisButton);            /* free entry */

}

boolean buttons__SetType(bts, name, type)
     struct buttons *bts;
     char *name;
     int type;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {
    nextButton=thisButton->next;

    if (!strcmp(thisButton->name,name))
      thisButton->type = type;
```

```
  }
}

boolean buttons__EnableAllButSelected(bts)
     struct buttons *bts;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {
    nextButton=thisButton->next;

    if (thisButton->type == DISABLED)
      thisButton->type = ENABLED;
  }

  buttons_WantUpdate(bts, bts);

  return TRUE;
}

boolean buttons__DisableAllButSelected(bts)
     struct buttons *bts;
{
  struct button *thisButton, *nextButton;

  for(thisButton=bts->buttons; thisButton; thisButton=nextButton) {
    nextButton=thisButton->next;

    if (thisButton->type == ENABLED)
      thisButton->type = DISABLED;
  }

  buttons_WantUpdate(bts, bts);
  return TRUE;
}

boolean buttons__SetEntryFont(bts, font)
     struct buttons *bts;
     struct fontdesc *font;
{
  bts->entryFont = font;

  return TRUE;
}

boolean buttons__SetTitleFont(bts, font)
     struct buttons *bts;
     struct fontdesc *font;
{
  bts->titleFont = font;

  return TRUE;
}
```

References

[Carroll and Rosson 85]
Carroll, John M. and Rosson, Mary Beth.
Usability Specifications as a Tool in Interative Development.
In Hartson, H. Rex (editor), *Advances in Human-Computer Interaction*. Ablex Publishing Corporation, Norwood, NJ, 1985.

[Cox 86] Cox, Brad J., Ph.D.
Object-Oriented Programming, An Evolutionary Approach.
Addison-Wesley Publishing Company, Reading, MA, 1986.

[Overview 88] Langston, Dianne and Grantham, Dennis, editors.
A Brief Overview of the Andrew Toolkit.
In *Introducing the Andrew Toolkit*, pages 3-10. Information Technology Center (ITC), Carnegie Mellon University, Pittsburgh, PA 15213, February, 1988.
Draft Version. This section excerpted from a longer article, *The Andrew Toolkit--An Overview*, presented a the 1988 USENIX Conference in Dallas, TX.

[Reference 88] Neuwirth, Christine M. and Ogura, Ayami.
The Andrew System Programmer's Guide to the Andrew Toolkit, Volume II: Reference
Information Technology Center (ITC), Carnegie Mellon University, Pittsburgh, PA 15213, 1988.
Draft Version corresponding to the IBM Andrew Toolkit software release of January 8, 1988.

[Wasserman 81] Wasserman, A. I.
User Software Engineering and the Design of Interactive Systems.
In Murray, G. (editor), *Infotech State of the Art Report*. Volume 4: *User Friendly Systems*. Pergamon Infotech Limited, Maidenhead, Berkshire, England, 1981.