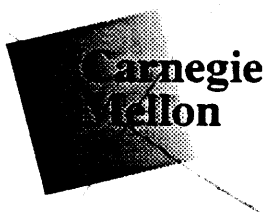
Programmable Command Languages for Window Systems

Richard Joel Cohn
CMU-CS-88-139
June 1988



Programmable Command Languages for Window Systems

Report number CMU-CS-88-139

This dissertation was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at Carnegie Mellon University.

The research in this dissertation was supported by the Information Technology Center, a joint project of Carnegie Mellon University and International Business Machines Corporation.

This dissertation was produced with Scribe. Illustrations were produced with Pic, Troff, and TranScript.

Abstract

Programmable command and macro languages have long served as important tools for users of computer systems. This thesis describes the design and implementation of a general-purpose command language for a window system. This has never been tried before because these systems are new and because the problems inherent in providing a command language for a window system are hard.

The thesis describes the requirements for a command language that can drive a wide variety of window system applications. Comparing these requirements with the properties of window systems and direct manipulation user interfaces shows why these requirements are difficult to meet and why some obvious approaches to the command language problem fail.

The thesis presents a practical strategy for providing a command language based on a compromise approach. This strategy defines an architecture in which developers provide programmable interfaces to their applications. Interfaces specify, at a *semantic* level, the objects understood by applications and the operations that manipulate these objects. The system incorporates these operations into a system-wide command language. The command language interpreter communicates with applications using remote procedure calls (RPC), enabling users to write command language programs that access operations of one or more applications transparently. The thesis describes a prototype implementation for Andrew, a multiple-process, multiple-window environment based on the UNIX operating system. While the prototype uses Lisp as the command language and C as the application implementation language, the RPC mechanism makes possible the use of multiple command and implementation languages.

This architecture simplifies the mechanics of providing interfaces to applications, but the task of interface design is intrinsically difficult. The thesis examines interfaces built for a variety of applications and extracts from this experience a set of guidelines for designing command language interfaces and interactive applications.

Acknowledgments

I still don't quite believe that I've actually finished—rather, almost finished—I still have to write the acknowledgments. Completing my doctorate was much more difficult than I imagined, but I have gotten much more help along the way than I ever could have hoped for. Now I get to thank everyone for their help. Thanking is easy; repaying is going to take a lot longer. But I'm working on it.

My committee provided guidance and good ideas. My advisor, Jim Morris, patiently wandered with me along the path to a proposal and eventually to the real thing. It was a learning experience for both of us. Gene Ball, my advisor and boss in a former life, made many helpful suggestions, especially to and from the airport. My other committee members, Dario Giuse and Ed Smith, provided comments that helped improve the thesis as well.

Having read one or two theses myself, I greatly appreciate the time spent by those who read mine and helped make it better: Mike Horowitz, Pedro Szekely, Edith Eligator, Claire Bono, and Mitch Silverstein. They weren't on my committee; they didn't have to read it.

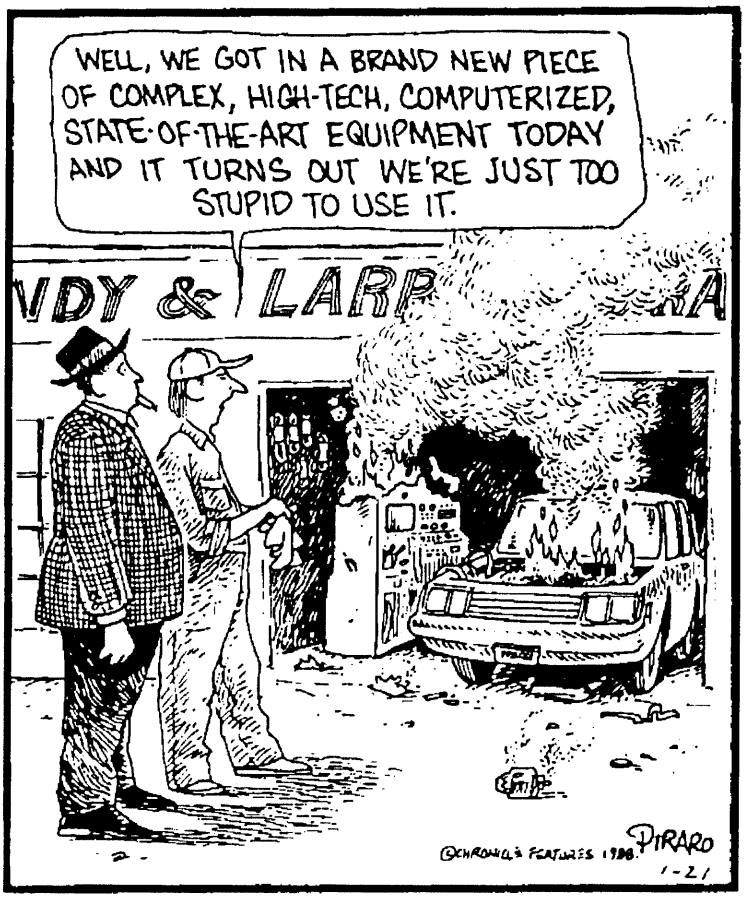
While I often wondered if I really should be working on a PhD, I never doubted that Carnegie Mellon, with its Computer Science Department and Information Technology Center, was the right place to do it. The department has an amazing combination of great people and great support that makes the environment around here very special. I just hope it stays that way. The people at the ITC made my stay at CMU much more pleasant than it otherwise would have been.

My officemates over the years have taught me a lot. Watching Satish, Andy, Ed, and Carl graduate showed me that it was hard, but not that hard. David suggested working with Jim, helped me get going, and most importantly, helped me stop.

I came to Pittsburgh to go to school. I ended up experiencing much more. The people who I want to acknowledge most are the friends I've made along the way. Very special thanks to Sharon, Suzanne, Mitch, Laurie, Kathy, and Claire. I would never have finished without you.

The more important the contribution, the less I have to say. I wouldn't have finished without the support of my friends; I wouldn't have gotten here in the first place without my family.

BIZARRO By DAN PIRARO



Contents

Illustrations

Tables

To my family

1 Introduction

The limits of my language mean the limits of my mind.
— Ludwig Wittgenstein

Command languages have long served as an important tool for users of computer systems. This thesis examines the use of a command language in a window environment and presents an architecture that supports its development.

1.1 MOTIVATION

The computer science community has devoted much attention to the design, implementation, and evaluation of user interfaces. System designers are concerned with how a user interacts with a program as well as with what the program can do. Hardware advances over the past ten years, especially the migration to personal computers, have allowed programmers to expend more resources on a system's user interface. With the exception of the most powerful supercomputers, the truism that computer time is more expensive than people's time no longer holds.

Designers are still experimenting, trying to determine how best to take advantage of increased resources and new technology. New input devices, such as mice and touch tablets, give a designer much more flexibility in obtaining responses from users. High-resolution bitmap displays permit a designer to provide much more information much faster than was previously possible. The amount of computing power available to a single user has increased tremendously in terms of processor speed and physical memory.

Greater resources give the software designer the opportunity to invent new techniques and implement ideas that were previously impractical. The set of interaction techniques collectively referred to by the term *direct manipulation* has greatly simplified the interface presented to the user by many application programs. Direct manipulation interfaces represent objects graphically and replace textual commands with direct manipulation of objects on the screen using a pointing device.

Another significant advance in user interface development is the multiple-window paradigm. A *window manager* sits between users and their applications, providing one or more virtual screens for each program. The user directs input to a particular window with a mouse or similar device. The term *window system* will be used here to describe a computer system where a user interacts with applications through a window manager and where many of these applications have direct manipulation interfaces.

A window system is an example of an enhancement to existing techniques—it has not replaced standard operating system interfaces. On the other hand, some new tools are expected to supersede existing ones. One traditional system element that has been neglected by system developers is the programmable command language. Command languages have been found to be extremely useful in conventional text-based environments, sometimes serving as the primary programming tool for users [Dolotta 80, Cowlishaw 84]. However, some user interface designers feel that a command language is an anachronism. Command languages were once needed because operations were invoked by lengthy and hard-to-remember commands. Use of the mouse, particularly in the domain of text editing, simplifies or eliminates many commands. Given these improvements, some argue that a tool to combine commands is no longer necessary.

The reports of the demise of command languages are greatly exaggerated. Combining commands in a macro is useful no matter what input mechanisms are available to the user. The introduction of the mouse has not eliminated all tedious and repetitious operations. Command language programs encapsulate a sequence of actions that are time-consuming (a sequence of file transfer commands, for example), need to be repeated exactly (a demonstration or test sequence), or are difficult to recall. Command languages enable users to customize and extend their environment by adding new operations. A command language is especially useful within an editor, where it can serve as an extension language, providing new commands for specialized kinds of editing.

The desire for some kind of command language is apparent in the marketplace, even for computers targeted towards relatively unsophisticated users. Several macro recorders have been introduced for the Apple Macintosh, allowing a user to easily replace a sequence of mouse clicks and keys by a single command. The inclusion of a macro facility in the Excel spreadsheet package has been a key factor in its dominance of the Macintosh spreadsheet market [Reich 85].

1.2 THE THESIS

A window system should provide a tool that satisfies these needs, allowing users to automate tasks within any application and those tasks that involve more than one application. The tool should deal gracefully with the special features of window system applications, particularly those with direct manipulation interfaces. It should be easy for users to write macros and easy for developers to incorporate the command language into their applications.

Existing techniques do not meet these objectives. Command languages designed for conventional text-based environments cannot easily be extended to work with window systems, especially with applications that manipulate graphical data. Other potential solutions are inflexible, restricting the developer's choice of interface style or implementation language.

This thesis presents a strategy for implementing command languages for window systems and describes a prototype design and implementation. This strategy defines an architecture in which software developers provide programmable interfaces to applications. An interface specifies, at a *semantic* level, the objects understood by applications and the operations that can manipulate these objects. These interfaces are incorporated into a system-wide general-purpose command language. A command language interpreter, implemented as a stand-alone process, is responsible for executing command language programs. It relies on remote procedure calls to invoke application functions.

Taking advantage of the operations made available by application interfaces, command language programs can carry out the same tasks that users perform interactively. The remote procedure call mechanism enables users to write command language programs that transparently combine commands from multiple applications. By permitting interfaces to be specified in terms of application-level objects and operations, this approach deals more effectively with the high-bandwidth communication typical of window systems than do alternatives that function at a lower level of abstraction. At the same time, the architecture is flexible enough to support a wide range of applications in a heterogeneous environment.

While this design simplifies the mechanics of providing an interface to an application, the task of interface design is intrinsically difficult. The thesis examines interfaces built for a variety of applications and extracts from this experience a set of guidelines for designing interfaces. Following these guidelines will lead to more flexible and more modular systems, even where a command language is not planned. While the thesis addresses the specific problem of providing a command language for window systems, this research has also served as a vehicle for studying more general user interface issues, particularly those relating to the structuring of applications.

1.3 THESIS OVERVIEW

This chapter has served to introduce and motivate the topic of command languages for window systems, suggesting that while useful, they are difficult to implement. It has briefly discussed the strategy adopted by this thesis. The remaining chapters elaborate and justify the claims made here.

The next chapter examines the issues to be addressed by a command language for window systems. It explains what a command language should provide in a window environment and identifies some of the difficulties faced by the command language designer.

Chapter 3 discusses some approaches to the problem, including text-based languages and input recorders. It evaluates these alternatives based on criteria established in Chapter 2.

Chapter 4 presents a new approach to the command language problem based on the strategy outlined in Section 1.2. It describes a prototype command language system called *Whisper* that demonstrates the viability of this approach. The chapter also describes some shortcomings of the prototype and explains how they could be addressed by extensions to the system.

Chapter 5 describes a few application interfaces, including a drawing editor, a text editor, and a mail reader. These case studies provide evidence that the Whisper design supports a variety of applications, including those that manipulate graphics as well as text.

The final chapter critically evaluates the prototype system and the research underlying it. The chapter explains how the techniques used here can be applied in other contexts, presents some suggestions for future research, and summarizes the contributions of the thesis.

The appendices contain some details about the prototype system. Appendix A describes the command language itself. Appendix B describes the language used to specify Whisper interfaces. The actual specifications for the applications discussed in Chapter 5 appear in Appendix C. Appendix D contains most of the code of a simple application, a desk calculator.

1.4 A NOTE ON DICTION

I have tried in vain to avoid the use of "he" and "his" as generic pronouns. As Mary-Claire van Leunen notes in *A Handbook for Scholars,* no one has yet found a reasonable substitute.

2 Command Languages

In 1973, C. J. Stephenson wrote, "Advances in computer languages during the past fifteen years have not generally been reflected in command and control languages. Those in common use are for the most part primitive and awkward" [Stephenson 73]. The same can be said today with respect to user interface design. The introduction of windowing and direct manipulation interfaces has made many systems easier to use. However, these systems do not provide command languages that meet the needs of their users. In fact, some provide no command language at all.

This chapter explains what a command language for a window system should provide and the issues that must be addressed in designing such a language. Given the vagueness of the term "command language," the chapter first provides a concise definition and describes some important characteristics of this class of computer language. The following section introduces the notion of *completeness* as a way of measuring the power of a command language. Subsequent sections examine completeness in text-based systems and window systems.

2.1 COMMAND LANGUAGE BASICS

A *command language* is a computer language that provides an interface to an application or operating system. The language consists of the commands of the underlying system. This definition uses the word "language" in its most general sense, meaning the symbols and gestures used for communication, in this case between a user and a program. This thesis focuses on *programmable* command languages. A program is a set of instructions that automate some process; a programmable command language enables a user to specify the process in a form that can be executed by the computer. Command language programs, in effect, extend the interface between the user and the system, allowing the user to create new commands and modify existing ones.

The term *command language* is also used to describe a particular type of user interface, one which is text-based, as distinguished from a menu-driven or visual interface. I will not use this latter definition here, and unless otherwise noted, command language will imply programmable command language. Throughout the thesis, "program" will refer to a command language program written by a "user," while "application" will denote a program written in a general-purpose programming language by an application "developer" or "designer."

Extension languages are one type of command languages. An extension language is an application-specific command language, providing a programmable interface to a single application. While an operating system has a command language, an editor has an extension language. Part of my thesis is that the same language should provide an interface to applications and the system as a whole, eliminating the need for this distinction. In the discussion that follows, I will not distinguish extension from command languages.

As Maurice Wilkes first observed, a computer language consists of "two languages, one inside the other; an outer language that is concerned with the flow of control and an inner language which operates on the data" [Wilkes 68]. Table 2.1 lists a number of command languages and the inner languages they support. While the distinction between inner and outer languages exists in any computer language, it is especially visible in command languages. Although programming languages rarely support more than one inner language, a few command languages do provide interfaces to a number of systems. Rexx and its predecessors, Exec and Exec 2, are prominent examples [Cowlishaw 84].

The distinction between inner and outer languages is often blurred. Most manuals, in fact, present the two languages together.[1] However, separating them allows developers to use the outer language to support multiple applications (inner languages), reducing implementation costs for themselves and learning time for users.

The outer language determines the form of command language programs. A program may consist of a list of input events, user commands, or procedures with associated parameters. It may be a linear sequence of instructions or a full-fledged program, including control structures, variables, and subroutines. The most common form of a command language program is the *macro*, a command that replaces an unconditional sequence of commands. Command languages include keyboard macro facilities and programming-by-example systems as well as general-purpose programming languages.

A system that provides a command language allows a user to be replaced by a program. The myriad of command languages share one common feature: each permits the user to

---

[1] See, for example, the Bourne shell and UNIX Emacs manuals [Bourne 78, Gosling 82].

Table 2.1. Example command languages

Command Language	Inner Language	Description
Bourne shell	UNIX	The first widely used UNIX command language [Bourne 78].
C Shell	UNIX	A Bourne shell descendant with a C-like syntax [Joy 79].
BAT files	MS-DOS	A simple command language for creating batch files [DOS 86].
Rexx	VM/CMS XEdit GDDM	A command language that supports multiple applications including the operating system, a text editor, and a graphics system [Cowlishaw 84, XEdit 83, GDDM 83].
PCL	Tops-20	An extension to the Tops-20 Exec that supports a programmable interface to applications as well as the operating system [PCL 82].
Awk	text manipulation	A language that associates actions with patterns found in lines of a text file [Aho 87].
Teco	Tops-20 Emacs	Teco, a line editor, was the implementation and extension language for the original Emacs [Stallman 81].
MLisp	UNIX Emacs	The first version of Emacs implemented on UNIX provided a Lisp-like extension language [Gosling 82].
SpiceLisp	Hemlock	Hemlock is an Emacs-like editor whose implementation and extension languages are both Lisp [MacLachlan 84].

capture in written form what can be done interactively and then to execute the stored program. An application often makes no distinction between commands issued directly by the user and those issued by a command language program. This is not by accident—applications are easier to build if they only need to provide a single interface. This explains the bizarre syntax of some command languages. While programming languages seem to be designed to be aesthetically pleasing, command languages are often designed to generate commands for unsuspecting applications.

The similarity between the interactive and programmable form of a command language makes the language easier to learn and use. The elements of the inner language are those users employ interactively everyday. Users find they have been writing (unrecorded) command language programs every time they use the system. They can move gradually from simple macros to complex programs as they learn the non-interactive components of the command language.

2.1.1 Uses of command languages

Command languages are flexible tools that support a variety of uses. This section describes the most common uses of these languages.

Keyboard macros are the simplest yet most common use of command languages. Many applications provide a facility which allows a user to record a sequence of keystrokes that can be played back later. The language constructs are exactly the interactive commands available to the user, and so there is little that even the casual user must learn to begin using keyboard macros. Although some systems allow macros to be saved for use in future sessions, in practice few macros are saved since most are written to perform a very specific task. It is frequently easier to rewrite a macro than to design a general one. Since macros are so easy to construct, it is often worthwhile to create one that will be used just a few times and then discarded.

Command languages frequently serve to customize an environment. Many interactive applications allow a user to define a personal profile which changes default parameter settings, declares command synonyms, and issues start-up commands. Applications with more powerful command languages extend the purpose of the profile, permitting a user to modify or add new commands. Support for customization does have some drawbacks, including increased difficulty in training new users, in transfer of knowledge between users, and in maintenance of extensions as the underlying system changes [Betts 87].

Command languages are also used to write programs to carry out tasks too complex for a keyboard macro, yet simple enough not to require the use of a general programming language. An example is a program that removes backup and temporary files from a user's directories. These programs are more difficult to construct than keyboard macros, and so are less likely to be written for a specific task and then thrown away. However, tasks that fall into this category, sometimes called "personal programming" or "programming in the small," do not require the efficiency or robustness that general-purpose programming languages provide. In many cases, the command language program is easier to write than the corresponding program in an ordinary programming language because it can make use of higher level primitives provided by some applications. The languages listed in Table 2.1 are all used for this type of programming. In an operating system command language, the primitives are programs, and the command language serves as a "glue" language, tying together independently written applications.

Some command languages are powerful enough to support the development of applications usually implemented in a general programming language. A wide range of programs have been written using MLisp, the UNIX Emacs extension language, although most of these programs seem to be mail readers [Borenstein 88a]. In fact, for the past

few years (1985–1987), the most widely used electronic bulletin board reader inside the Computer Science Department at Carnegie Mellon University has been one written in MLisp [Borenstein 85]. As is true for personal programming, the command language approach is attractive because of the powerful primitives available. An interpreted command language is especially suited for the iterative nature of user interface development [Sheil 83].

Yet another use of command languages is to test programs. *Regression testing* is a method of determining if changes to a program have introduced bugs into supposedly unmodified parts of the program. To see if the program has regressed, a tester records the response of the original program to a set of inputs, subjects the modified program to the same inputs, and compares the responses to the original output [Petschenik 85]. A tester can automate the process by recording inputs in a command language script. If the command language can record output, then testing can be completely automated.

If the command language includes a recording mechanism, it can be used for another kind of testing—human factors evaluation. Observing real users trying new applications and new application features helps user interface specialists and application developers learn what works and what needs to be revised. Recording user input in a script is often more practical than the two most common alternatives—videotaping and recording user sessions by hand—since the data is saved in a form that can be easily analyzed. For example, one study of UNIX analyzed frequency and grouping of operating system commands [Kraut 83]. A problem with this approach is the huge amount of information recorded. The UNIX study limited keystroke recording to just a few subjects and depended on semantic information (a record of process invocations) for the rest of the group. Another study depended entirely on semantic information [Hanson 84].

2.1.2 Evaluation

A command language, like any other tool, is successful if it makes its users more productive. For any given task, four factors determine command language productivity:

- *Feasibility*. Can the language be used to perform the task?
- *Ease of use*. How easily can the language be used to perform the task?
- *Efficiency*. How quickly can the task be performed?
- *Generality*. What constraints does the language impose on applications and the underlying system?

This section discusses each factor and its importance in determining the quality of a command language.

Feasibility

Determining the feasibility of using a command language for a range of tasks is the first step in the language's evaluation. A command language does not stand alone. If a task requires the use of one or more applications, can the command language be used with these applications? Even when the command language is available within an application, its use may be limited. For each interactive command, is there a corresponding function in the command language?

Ease of use

Ease of use is critical to the success of a command language. Command languages rarely provide functionality otherwise unavailable; usually a task performed by a macro can be done by hand or by a program written in a standard programming language. Given a particular task, a command language is useful only if it is faster to write and execute a macro than it is to do the task by hand or to write and execute a program using a general-purpose programming language.

A user must estimate the cost of each approach, with the best solution determined by the amount of use expected, as illustrated by Figure 2.1. Two factors determine cost: the time required to write the program (zero when executed by hand) and the time required to execute it. If the task will be repeated, the value of the command and programming language solutions increases since the cost of writing the program is amortized over the number of executions. As the number of repetitions grows, it becomes more likely that the programming language approach will be the most efficient since it is usually fastest per repetition. The usefulness of a command language cannot be predicted without knowing the environment in which it will be used. Command languages are less valuable in easy-to-use systems and in systems with easy-to-use programming languages.

A language's primitives—the components of its outer and inner languages—largely determine its ease of use. While feasibility addresses the question of whether the language supports an operation, ease of use determines how appropriate the language primitives are for solving particular problems. A command language with powerful but low-level primitives may be too difficult for many people to use.

Additional criteria, outside the language itself, need to be examined as well. These considerations center on the language's development environment and include ease of creation, turn-around time for making changes, and the availability of debugging tools. For example, keyboard macros are easy to create but difficult or impossible to edit, while more traditional command languages require more work to create but are easily modified and tested.

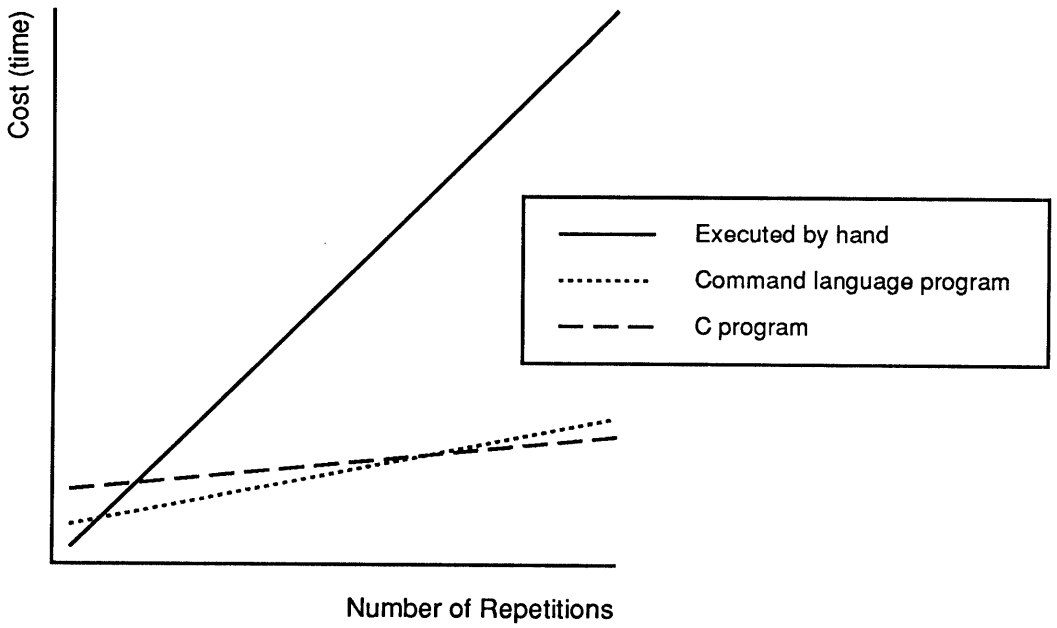


Figure 2.1. Cost comparison of programming approaches

Efficiency

While command language efficiency is determined by the time required to prepare a program and the time required to execute it, preparation time is often more important. One study of UNIX shell programs found that many are three orders of magnitude less efficient than equivalent C programs [Dolotta 80]. Users found this to be an acceptable trade-off of "machine efficiency" for "people efficiency" (ease of use).

Even when using a relatively inefficient command language, most of the time needed to execute a macro is spent inside inner language primitives. The primitives of the inner language of a command language are usually much more powerful than those of the outer language and take correspondingly longer to execute. This difference does not usually exist for programming languages.

Generality

Generality measures how a command language constrains its client applications. A command language is a tool that works through the cooperation of a set of independent applications. Cooperation requires that applications accept certain restrictions. These restrictions can have a least-common-denominator effect, preventing applications from using certain features that cannot be incorporated across the system, and reducing the choice of user interface techniques. For example, to avoid dependencies on particular window managers, a user interface toolkit may prevent the use of operations that are not provided by all window managers. In this way, constraints affect user productivity by limiting the capabilities of the system and its components.

All command languages impose constraints. All impose some computation model on programs, although application developers can decide how strictly to follow this model. The less closely they follow the model, the less system functionality they can make use of, and the less their application's user interface matches the rest of the system.

The UNIX shell, for example, models an application as a filter acting on a stream [Pike 84]. Applications read data from *standard input,* write data to *standard output,* and send special messages to *standard error.* Applications that follow these conventions can read input from a file or another process, as well as interactively. The same is true for writing data. Treating input and output symmetrically makes it possible for a shell programmer to compose functions using pipes.

While successful, the UNIX model demonstrates the problems inherent in imposing constraints on applications. Numerous exceptions to the model exist since it does not handle interactive applications well. Applications can be driven by scripts as long as they read from standard input. However, output is more problematic. Many applications act on a database (mail readers, text editors, and bibliography managers as well as general databases) rather than generating output. In these cases, standard input consists of control commands only, without any data. Standard output is used for feedback as well as data. Simply dividing output into standard output and standard error does not provide fine enough control. Some applications never fit the model. The Diff file comparator, for example, requires two inputs. The UNIX model, while simple and powerful, is too restrictive for many applications.

2.2 COMPLETENESS OF COMMAND LANGUAGES

While the four factors discussed above can be used to evaluate a command language, they cannot be measured precisely. This section presents the concept of *completeness,* which, though still qualitative, can be more rigorously evaluated. Completeness is closely related to both feasibility and ease of use. Understanding completeness also makes it easier to determine what a command language should provide in a window system. It helps to identify the services that existing command languages provide and the analogous services that command languages for window systems should provide.

A command language permits users to write programs that can do much of what can be done interactively. A system is *complete* only if users are able to accomplish in a command language program everything they are able to do interactively.

Both the inner and outer languages affect completeness. A user interacts with a system by invoking operations, examining the results of these operations, and invoking more operations based on analysis of these results. The inner language must make available a

full set of operations for completeness to hold. The outer language must permit a program to manipulate operation results and make decisions based on these results.

Completeness should be evaluated with respect to a level of abstraction in the user interface. Consider a drawing editor such as MacDraw for the Apple Macintosh. MacDraw allows the user to draw geometric objects on a page. It represents an object as a sequence of pixels determined by the position of the object and the resolution of the display. To add a box to a drawing, the user presses the mouse button at one corner and drags the cursor to the location of the other corner. As the user drags the cursor, the box is shown as it would appear if the user released the mouse button at that point. This type of interaction, called *rubber-banding,* is an example of *semantic feedback.* The definition of completeness seems to imply that the command language should be able to specify the meanderings of the mouse as the user determines the final box corner. The definition further implies that a command language program should be able to determine exactly what pixels represent the box since a careful user could do the same. While this information may be important in some contexts, it is more often irrelevant and is outside the underlying model of the application. Identifying layers of user interaction makes it possible to precisely state what information should be accessible from a command language program.

Many researchers have proposed models of human-computer interaction [Foley 82, Card 83, Shneiderman 87]. The model I present here, examining user interaction at four levels, is based on these earlier ones.

- The *conceptual* layer defines the task concepts understood by the user. At this level, the user can describe a task independent of any computer system or application program.
- The *semantic* layer specifies functionality in terms of a specific application's implementation of task concepts. Both the conceptual and semantic layers describe classes of objects and operations on these objects.
- The *syntactic* layer defines the structure of tokens used to invoke operations and to create output.
- The *lexical* layer maps physical actions into tokens and tokens into display effects. In most systems, a window manager or graphics package handles this mapping.

Depending on what is of interest, rubber-banding of a box can be analyzed at any of the four levels. At the conceptual level, the user creates a box in a drawing. At the semantic level, the user invokes MacDraw's create-box operator. In this example, there is little difference between the conceptual and semantic levels. If MacDraw did not have a create-box operator, the user would have to map the conceptual task into a sequence of create-line operations. At the syntactic level, the application receives input events cor-

responding to pressing the mouse button, moving the mouse, and releasing the button. The mapping between the lexical and syntactic levels is trivial here, but it need not be. The same syntactic tokens could be generated by another source, such as a macro recorder. Note that in this example, the path of the mouse and the time delay between input events are important only at the two lowest levels. This is usually true. Furthermore, in applications that do not provide feedback, the lexical layer can serve as a filter, preventing mouse movement or time delays from entering the stream of syntactic tokens.

The effect of the user's actions on the display can also be analyzed at each level. At the conceptual level, the user has created a geometric entity of a certain length and width. At the semantic level, MacDraw defines the box internally using some data structure. At the syntactic level, the program draws the box by issuing commands to the low-level graphics package. At the lexical level, MacDraw represents the box by a set of pixels on the screen or by a box drawn on paper.

The importance of completeness differs at each level of user interaction. Completeness at the conceptual level is critical for a command language. If the language is incomplete at this level, the user may need to rely on interactive operations to perform a task. At the semantic level, completeness is desirable but operations may be replaced with equivalent ones without affecting the conceptual level. The syntactic level is usually invisible to the user but does affect application programs. This is the most natural point to plug in a command language. If the command language can mimic the stream of tokens that the user produces interactively and consume the output the application produces, no changes are needed in the application to support the command language. As the next two sections will explain, lexical completeness is easy to attain in text-based systems but presents problems in other systems. Lexical and semantic completeness determine ease of use of a command language, while conceptual completeness determines feasibility.

2.3 COMPLETENESS IN TEXT-BASED SYSTEMS

Completeness is not difficult to achieve in text-based systems. Every action a user can carry out consists of pressing keys. Although the user may view a command language program as producing a sequence of commands for the environment, in practice most command languages provide applications an uninterpreted stream of characters. The user sees no difference between commands issued from a program and those invoked interactively. The application sees no difference either—it receives the same input stream in both cases.[2]

---

[2]This is not strictly true. In UNIX, for example, an application can determine if its input source is a program and alter its behavior accordingly.

Some of the most widely used text-based operating systems, including UNIX, MS-DOS, VM/CMS, and Tops-20, include nearly complete command languages [Bourne 78, DOS 86, Rexx 83, PCL 82]. In these systems, users enter commands by typing a line of text followed by a termination character. A command language can provide lexical completeness by adopting the same approach, sending lines of text to applications. Simple command language programs can be created by placing a list of commands in a file as they would have been typed to the system executive. The languages all provide some way of gathering output. Command language programs can evaluate this output and alter execution through the use of conditionals, gotos, variables, and, in some cases, subroutines.

Because so much of a user's time is spent inside applications, a command language should be able to execute subcommands within an application as well as invoke the application itself. A powerful command language will not increase productivity if a user is rarely in the environment in which the language works.

However, completeness does not often hold at the application level for existing command languages. In UNIX and MS-DOS, for example, the system command language is unavailable inside applications. The Tops-20 command language, PCL, does allow a script to send input to an application and to read the output generated. Rexx permits applications to supply their own interface, but few do. In many systems, a few applications fill this void by supplying their own command language.

While the UNIX shell language cannot be used inside an application, it can send input to a program and record the program's output. The output can be examined to determine the next program to run. String pattern matching programs can aid output processing. However, this style of command language programming doesn't work with screen-based applications. Rather than assuming the output device is a teletype, these applications support character-addressable displays. The command language can generate input for this type of application but cannot evaluate output. The output is too low-level, containing device-dependent terminal instructions interspersed with data. This same problem appears in window systems but is more severe since the use of screen-based applications is much greater.

2.4 COMPLETENESS IN WINDOW SYSTEMS

The introduction of workstations with pointing devices and bitmap displays has led to numerous advances in user interfaces. Mice and other pointing devices have greatly changed the nature of input a user can supply. New displays have made it possible to display graphics as well as more text than before—potential output bandwidth is orders of magnitude greater. While character-addressable displays made it feasible to move away from the simple teletype style of interaction, the combination of mouse and all-

points-addressable display have accelerated this trend, making computers much easier to use.

This section reviews the differences between window systems and the more traditional systems for which most command languages were designed, and then draws some conclusions about what should be provided by a complete command language for a window system. It also notes some problems that arise in handling window system features.

2.4.1 Visual applications

An obvious change made possible by new hardware is the introduction of applications whose underlying data is inherently visual. While Ivan Sutherland created the first interactive drawing editor over twenty years ago [Sutherland 63], only recently have drawing programs become widely available. First vector and now bitmap displays support the manipulation of graphical objects. Bitmap displays in particular allow applications to display a tremendous amount of data, much more than was ever practical in text-based systems.

Applications vary greatly in the amount of data represented by the same bitmap. By examining application output at an appropriate level of abstraction, the data bandwidth becomes more manageable. Applications exist within a hierarchy of abstraction, some of whose levels are shown in Figure 2.2.

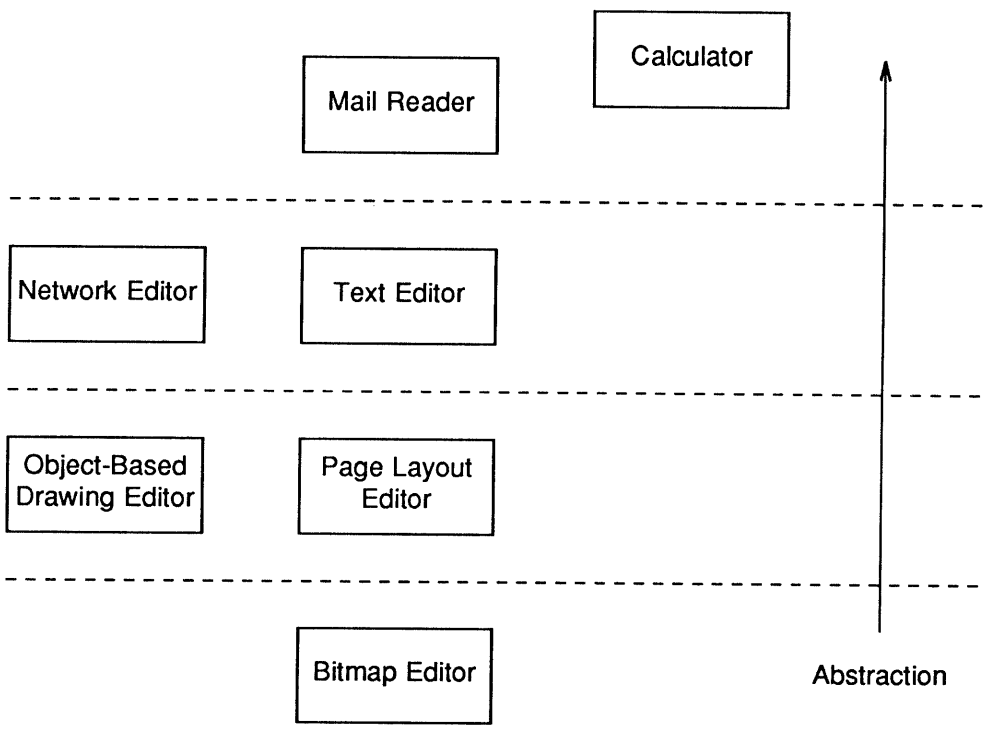


Figure 2.2. A hierarchy of applications

While all the applications pictured present information using a bitmap, the underlying data varies:

- The data managed by a bitmap editor such as MacPaint is just a matrix of zeros and ones, represented by a bitmap whose pixels are white or black. The user has control over every bit. The mapping between internal data and the display is one-to-one: "What you see is all you've got."
- The drawing editor, whose display is superficially similar to a bitmap editor's, manipulates a set of geometric objects. The actual bits that represent a line are not part of the application's data but just an effect of the presentation. The user can specify the location of the line but not its bit-by-bit representation. The program has complete control over how to represent the line at the limited resolution provided by the display.
- A page layout program, such as PageMaker, provides the user with the ability to specify the exact location of text, but not at the pixel level. The program exists midway in the hierarchy.
- The network editor, an application that models electrical circuits and logic diagrams, only stores objects and their connectivity. The layout of the objects is not part of the data and not under user control.
- A text editor with automatic formatting, such as MacWrite, is similar to the network editor. The user specifies what a document contains but not how it is laid out.
- The mail reader's display (see Figure 5.10 on page 89) has little connection to its data, compared to the other applications. Users have little control over layout, although they do control the content of messages, just as they determine the contents of documents in a text editor. Other objects, such as the list of message titles, cannot be changed. While implemented as documents managed by the text editor, these objects have more structure than ordinary text.
- Of all the applications shown in Figure 2.2, the calculator is the most abstract. Its array of buttons (see Figure 4.5 on page 45) is part of its input interface. Only the number display represents output. The bitmap displays a string which corresponds to an internal number.

The more abstract the application, the less inherently visual is its data, and the greater is the distance from the data's internal representation to its external representation. While the distance from the bitmap editor's data to its display is small, the distance for the calculator is much greater.

The inability of a user to control details of the presentation characterizes a more abstract interface. The difference between the text and page layout editors, for example, is that a user cannot specify page locations in the text editor. The key step in placing an application in the hierarchy of Figure 2.2 is classifying the information provided by the application as part of the presentation or as part of the underlying data. In the text editor, text

location is part of the presentation, while in the page layout editor, it is part of the data. Similarly, in the network editor, object location is part of the presentation, while in the bitmap editor, it is part of the data. Intuitively, data is what gets written to the disk, and the presentation is everything else.[3] For tools that produce documents (including text, drawing, table, equation, and graph editors), an alternative to the disk rule is that something is data if it affects the final product—the printed page. More generally, for any application that produces multiple views—on paper and on the screen, or all on the screen—data is what is common to all views.

The great amount of data presented by visual applications creates a problem for a command language. The type of data is just as much a problem. A user often examines the output of one command to determine the next command to invoke. In a text-based system, a command language can simulate this examination by performing string-matching operations to compare actual output against expected results. This programmed pattern recognition may not be possible given the output of visual applications. It is much more difficult to recognize patterns in graphic commands than it is in text strings.

Recognizing the distinction between data and its presentation can make a command language for a window system more usable, at least for more abstract applications. Completeness requires that application data be accessible. For a drawing editor, this means the user must have access to the objects in the drawing, and for each object, to its defining points and attributes. An application whose command language interface provides access to this information but not to the raw display is semantically but not lexically complete. For a more abstract application, this strategy both reduces the sheer amount of data and converts it to a more manipulable form. For an application as concrete as the bitmap editor, there is no distinction between its internal data and its display, and so the only interface that can be provided is the raw bitmap.

While this discussion has focused on output, much the same is true for input. Consider the example presented in Section 2.2: a drawing editor that allows a user to create a box by dragging a corner to its final destination. Even if the command language interface to this operation only specifies two corners, no semantic information is lost. The level of abstraction should be decided on a command-by-command basis. A bitmap editor might provide the same box creation operation and use the same command language interface. However, it might also have a sketch operation that paints each pixel crossed by the path of the mouse cursor while it is held down. In this case, the input cannot be condensed if semantic completeness is to be maintained.

---

[3]There is an exception to this rule: a program may choose to preserve some presentation information such as the current location in the document or the current selection.

The idea of condensing input is not specific to window systems. Most existing textual command languages do not give access to low-level keyboard interfaces, instead providing some more convenient interface. The UNIX teletype driver, for example, supports a limited editing capability. While an application may obtain input as the user enters it, most applications read a line at a time, permitting the user to edit the line of text until an end-of-line character is entered. The application program is unaware of any editing, only seeing the final result. The system eliminates timing data in the same way. While this information is available, most applications never see it.

The choice of abstraction level depends on the objectives of the command language as well as on the application. For example, if a command language is used for regression testing, it may be necessary to record and replay user commands exactly and to examine application output at the graphic command (syntactic) or bitmap (lexical) level. For most command language uses, however, semantic completeness of output is "good enough."

2.4.2 Direct manipulation interfaces

Many applications, graphical as well as non-graphical, take advantage of new technology to provide the user a more direct interface to their underlying data and operations. The features of *direct manipulation* interfaces [Shneiderman 87] include:

- Constantly visible objects and actions of interest.
- Rapid, reversible, incremental actions.
- Replacement of textual commands by direct manipulation of objects of interest.

This style of interaction, pioneered by researchers at the Xerox Palo Alto Research Center [Goldberg 84, Teitelman 85], was introduced into the computer mainstream by the Apple Macintosh. Figure 2.3 shows a typical direct manipulation display, a Macintosh program that allows users to learn about and change system parameters. The control panel constantly displays the range of legal settings for each parameter as well as its current value. A user sets a parameter by clicking the mouse button, not by entering values from the keyboard.

Hutchins, Holland, and Norman describe these interfaces at a more abstract level [Hutchins 86]. Direct manipulation interfaces provide a "feeling of directness" characterized by:

- *Engagement.* A feeling of communicating with the objects of interest, not a computer.
- *Articulatory distance.* Closeness of task and system objects at the lexical level. Articulatory distance measures the degree to which the form of communication reflects the task objects and operations.

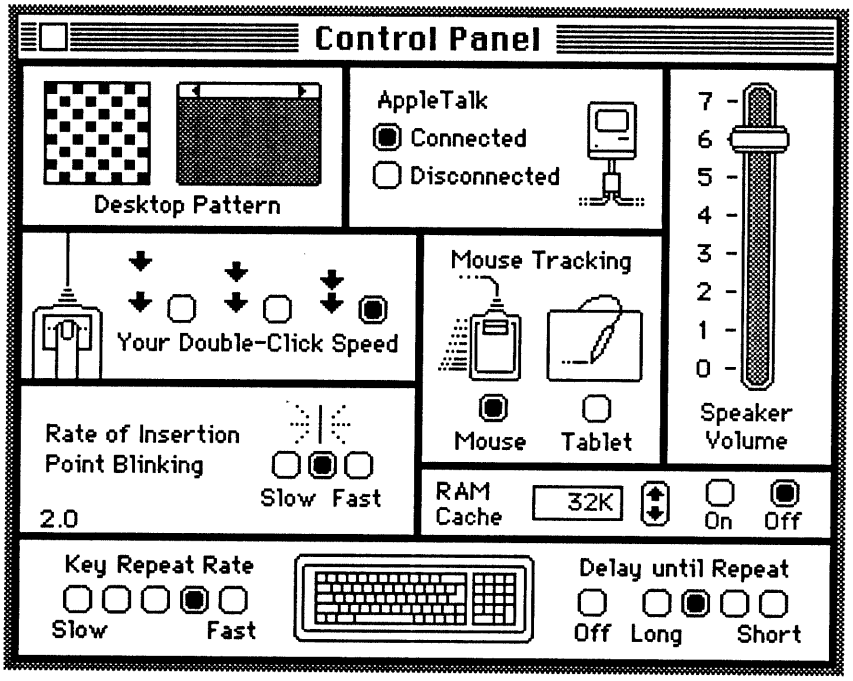


Figure 2.3. The Macintosh control panel

- *Semantic distance.* Closeness of task and system objects at the semantic level. Semantic distance measures the degree to which the semantic concepts presented by the interface correspond to the user's concepts and goals.

Direct manipulation interfaces provide more feedback than other interface styles, often at a semantic level. An example of semantic feedback is the use of "gravity" in a drawing editor. As the user rubber-bands a line, the editor may restrict the placement of the line's endpoint to existing points whenever the cursor is close to one of these points. The use of gravity assumes that the user rarely wants to place two points very close to one another. If they are close, the user most likely wants the points to coincide.

The features that make direct manipulation interfaces easy to use are the same that make integration with a command language difficult: continuous visibility of objects, rapid actions, and replacement of text commands by graphical manipulation.

- *Implicit computation.* In a text-based system, the user must make explicit queries to determine the state of the system. In a direct manipulation interface, the user can obtain information just by examining the display. Often, a user will choose data by directly selecting a displayed item rather than computing the selection. The user performs a critical part of the computation in his head. It is difficult to automatically record this type of information in a command language script. For example, while writing a macro in a text editor, the user may select a word. The system cannot determine the *search method* used to choose the word: its position or its content.
- *Level of abstraction.* Rapid incremental actions are easy to carry out but

hard to understand in retrospect. It is difficult for a person to read and edit a macro that contains many simple low-level operations.

- *No textual interface.* A command language interface is trivial to define when a user interface is text-based. The commands of the user interface can serve as the inner language of the command language. However, the user interface is only the starting point for the design of the command language interface for an application with a direct manipulation interface.

Semantic feedback causes problems related to these issues. For example, when users take advantage of gravity while drawing lines, they are actually querying the application about the location of existing lines during the create operation. Semantic feedback embeds high-level query operations within simple low-level operations.

In sum, direct manipulation interfaces greatly increase the bandwidth of communication between the user and the system for non-graphical as well as graphical applications. Not coincidentally, this communication is no longer purely textual and sequential. Both of these developments pose problems for command languages, which tend to be oriented towards low-bandwidth text-based applications.

2.4.3 Window managers

One of the ideas resulting from the development of powerful workstations is the multiple window paradigm. By placing concurrent applications in physically distinct virtual terminals, this paradigm has made multi-tasking easier to understand and to use.

While multi-tasking operating systems have existed for quite some time, only sophisticated users had been able to take advantage of this feature and only in a limited fashion in text-based systems. In standard UNIX, for example, multiple programs can run simultaneously, but the system displays output as it is generated, often interleaving the results of the executing programs. There is no physical separation of the output of different applications. Furthermore, because the user cannot easily specify input destination, only one program can actively seek input at a time. For these reasons, the use of multiple processes tends to be limited to simple batch processes such as compilations and print requests.

In a window system, even novices are likely to execute multiple interactive programs simultaneously. The multiple window paradigm has significantly changed the way users interact with the system. As shown in Figure 2.4, windows divide information provided by applications into physically distinct regions of the screen. Many common tasks may involve more than one application. While using one application to carry out a task, the user may need to access information via a different application, perhaps to obtain help or to find a file. The use of another window to execute the second application allows the

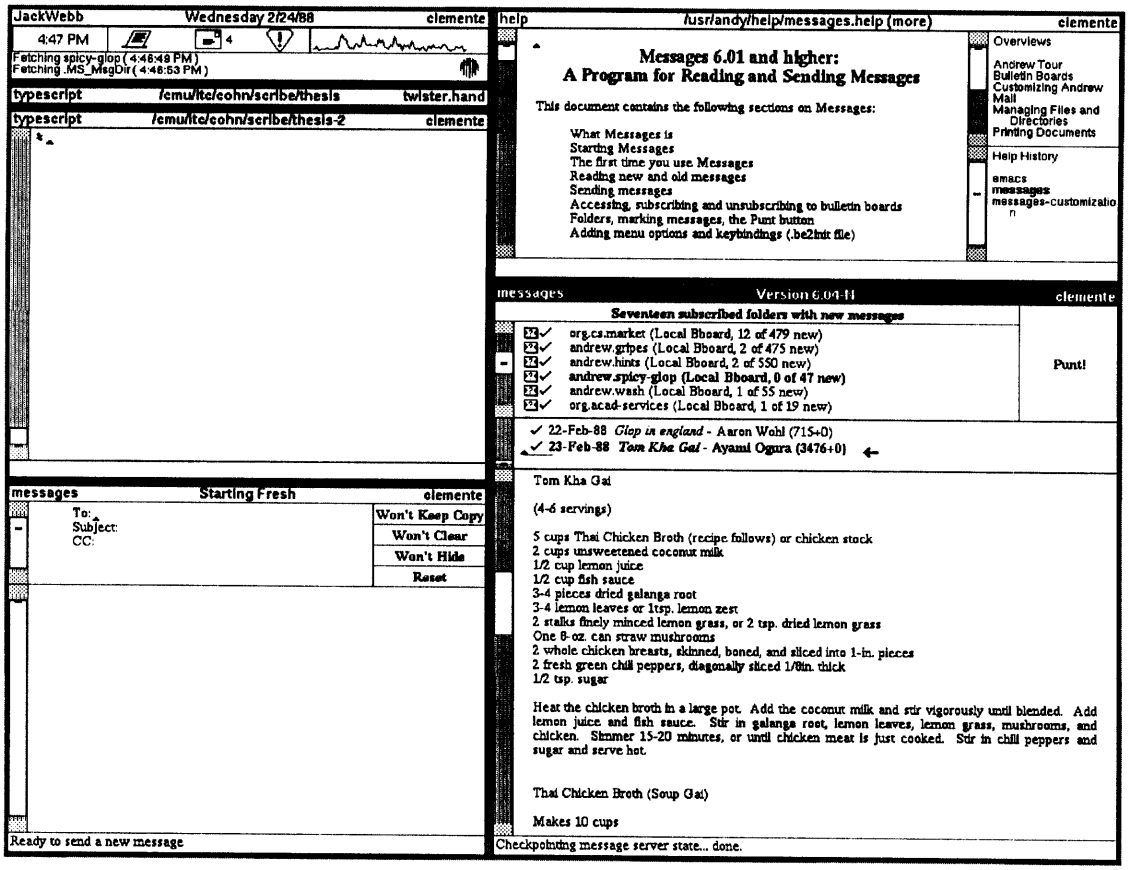


Figure 2.4. A window system display

user to obtain this information without altering the context developed in the original application. Sometimes the user may only need to confirm some facts without explicitly moving data between the two applications, while in other situations the information itself must be transferred. Some systems provide tools that encourage this style of interaction. For example, a few support a *cut and paste* facility that allows the user to select and move information from one application to another without using an intermediary file. Traditional command languages do not support this kind of interaction between applications.

The window manager has taken over some of the responsibilities traditionally held by the command language. In window systems running on top of UNIX, for example, users often change or kill the active process using window manager commands rather than shell commands. Cut and paste replaces, to some extent, the features in the shell that allow the user to move data between applications. The Macintosh provides a more extreme example. With no textual command language available, users must rely exclusively on the window manager (the Finder) and cut and paste.

A command language for a window system must incorporate comparable functionality.

A command language program, like the user, should be able to communicate with multiple active applications at the same time. Given this capability, a program could perform some actions within one application, examine the results, and carry out operations in another application based on these results. Within this framework, the command language could easily support tools already available interactively that allow a user to move data from one application to another, including cut and paste.

2.5 SUMMARY

After defining some basic concepts and describing uses of command languages, this chapter established four evaluation criteria: feasibility, generality, ease of use, and efficiency. Feasibility determines whether a command language can be used to perform a task, while generality determines the constraints the language places on the underlying system and applications. Ease of use and efficiency determine how well suited a command language is for a task.

While these criteria are helpful in comparing languages, the notion of completeness provides a more objective measure of the power of a command language. A language is complete, at a given level of abstraction, if a command language program can do what a user can do interactively. If a command language is lexically complete, users may write programs with the same commands that are available interactively. Semantic completeness ensures that any function provided interactively has an equivalent in the command language.

Lexical completeness is desirable since it greatly simplifies learning a command language and enables users to create programs quickly and easily. It also increases generality, since, in some implementations, command language programs can be substituted for an interactive user without the application's knowledge. However, characteristics of window systems and direct manipulation user interfaces often make semantic completeness a more appropriate objective. Operating at a semantic level allows a command language to condense the high bandwidth of input and output typical of window systems. It also allows programs to manipulate graphical data more conveniently. Moreover, semantic descriptions of operations produce more readable and more easily modifiable command language programs. Understanding of this trade-off will aid in the evaluation of command language alternatives to be presented in the next chapter.

3 Approaches to the Problem

Command languages have a long history, beginning with job control languages that managed tasks in early batch computer systems. The introduction of interactive computing led to a new generation of more powerful languages. More recently, the introduction of direct manipulation interfaces has spurred the development of input recorders and programming by example. This chapter reviews these tools, emphasizing their use or potential use in window systems. It begins with traditional text-based languages, and then moves on to input recorders—command languages in which users create programs by executing them rather than explicitly writing them down.

3.1 TEXT-BASED COMMAND LANGUAGES

Most command languages are text-based, even in window systems. The obvious reason is that alternatives have become generally available only in the past few years. However, there is a more compelling reason for the predominance of text-based languages: written language is more expressive for many purposes than images. Fine details and large amounts of data can often be manipulated more easily. For both these reasons, current window systems provide a wide range of text-based command languages.

3.1.1 Application-specific languages

One of the most successful approaches to the command language problem has been the development of extension languages—command languages within applications. Although most common for text editors, particularly the Emacs family,[1] command languages have been written for many applications, including drawing editors and spreadsheet programs [Raker 85, Hergert 86].

---

[1]Several distinct text editors share the name "Emacs" [Stallman 81, Gosling 82, Stallman 86]. The key feature common to all instances of Emacs is extensibility. Unless otherwise noted, comments about Emacs apply to all versions of the editor.

For example, Excel, a Macintosh spreadsheet program, provides a macro language that is as powerful as some programming languages [Hergert 86]. A macro is a column in a spreadsheet; each cell in the column contains a statement of the macro. Users write macros on the spreadsheet and can refer to an expression or statement in the program by its location in the spreadsheet. Statements use the same syntax as cell formulas, enabling users to avoid learning a new language. However, since the cell language was not designed as a programming language, it has some unusual and inconvenient features. For example, storage for variables must be explicitly allocated since a variable is just another cell in a spreadsheet. Despite these problems, Excel has become the most popular spreadsheet for the Macintosh, in large part due to its macro capabilities.

UNIX provides many application-specific languages as well as tools for building new ones [Bentley 86a]. The UNIX pipe mechanism makes it convenient to build languages implemented as preprocessors for other languages by enabling the output of one process to be tied to the input of another. For example, preprocessors for Troff, the UNIX document formatter, allow users to create pictures, tables, equations, and graphs [Kernighan 84, Kernighan 82, Bentley 86b]. All these "little languages" fill a specific niche yet make use of facilities in more general languages. For example, text in pictures may include ordinary formatting commands passed through to Troff. The graph language uses the picture language to generate Troff commands that place graph elements on the page. Make, Awk, and Sed are examples of little languages outside the domain of document production [Kernighan 84]. Each of these languages is tailored towards the particular kind of object it can manipulate.

HyperTalk, the language embedded in the Macintosh's HyperCard, is another example [Goodman 87]. HyperCard is an application that allows users to build *stacks* of *cards* containing text and graphics. Each card is a window full of information. A card may contain *buttons* that are links to other cards or that trigger the execution of HyperTalk programs. HyperCard has been used to implement address books and calendars, a car repair manual, and a HyperCard programming manual that includes working examples of HyperTalk programs.

HyperTalk scripts create pictures by describing, in an English-like syntax, interactive commands that would draw the desired graphics. For example, the following script will draw a circle:

```
choose select tool
doMenu select all
doMenu clear picture          -- delete everything
reset paint                   -- set standard conditions
choose oval tool
set lineSize to 2             -- double line width
set  dragSpeed to 0           -- draw as fast as possible
drag from 10, 10 to 250, 250  -- drag as user would
```

HyperTalk's inner language mimics the interactive interface as closely as possible in order to ease the transition from user to programmer.

An application-specific language can be a very effective tool. It is often easier to use and more efficient than a system-wide language. Generality is not an issue since the language is designed for the application. The language can be tailored to the task at hand, providing constructs useful for the particular application and dispensing with costly but inappropriate features.

Several disadvantages can outweigh these considerations. The ease of use of a task-specific language must be compared to the cost of learning and remembering yet another language. As is true for interactive interfaces, consistency across applications is important. Secondly, it is more efficient to provide a single command language than to implement one for each application. Savings can be achieved in development time, maintenance costs, and system size. Finally, a set of application-specific command languages cannot be used to program tasks that involve more than one application.

3.1.2 Package-specific languages

A related yet more general solution is to provide a programmable interface within a library or support package. For example, Grits is a database package running on the Andrew system that has been used to build a variety of applications, from an on-line questionnaire to a data retrieval and analysis program called The Great American History Machine [Miller 86].

From one perspective, UNIX Emacs is also a support package, serving as a base for many applications, from mail readers to directory editors to cheese-ordering programs [Borenstein 88a]. Its extension language serves as both a language for writing macros and as a language for implementing user interface applications. The editor's ability to tie processes to text buffers enables it to function as a window manager for character-addressable terminals.

Another example is NeWS, the Network Extensible Window System [NeWS 86]. NeWS includes the PostScript page description language [Adobe 85] as an extension language, permitting clients to define new commands. These commands, actually PostScript programs, can be downloaded into the window manager to improve performance. The NeWS extension language, unlike a typical command language, is intended for use by application developers, not end users. While available to all window manager clients, the language supports a narrowly defined range of tasks.

A package-specific language has the same advantages as the application-specific approach without all the disadvantages. Since the domain of these packages is limited, the

language can still be tailored as appropriate. Furthermore, the cost of implementation and learning can be amortized over a set of applications. If all applications in a system use the package, the language is effectively a system-wide command language. However, most package-specific languages do not permit mixing of operations from multiple applications.

User interface management systems

One particular kind of support package is the *user interface management system.* A UIMS provides the programmer with a variety of tools that are designed to simplify the creation of high quality user interfaces [Thomas 83, Olsen 84, Pfaff 85]. A UIMS introduces a higher level of abstraction for input and output so that the developer can take advantage of the bitmap display without dealing with formatting details.

According to the model of Tanner and Buxton [Tanner 85], the components of a typical UIMS correspond to two phases of user interface development: preparation and run-time. (See Figure 3.1.[2]) In the preparation phase, the application developer specifies an application interface—the operations and data of the underlying functionality—and a dialogue description—how the application is presented to the user. The UIMS uses the dialogue and application interface specifications and a library of interaction techniques to generate a user interface specification (Figure 3.2). The run-time phase supports the execution of this specification. Some UIMSs support a follow-up phase by collecting data about user interaction for later analysis.

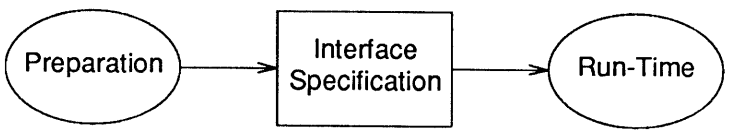


Figure 3.1. A model of user interface management systems

The style of programming mandated by the typical UIMS makes it easy to add a command language, although no UIMS described in the literature now supports one. By requiring the developer to specify the application interface and the dialogue description, most UIMSs force the programmer to separate the user interface from the base application. The code for the user interface is generated automatically by the system from these specifications. The application interface could be extended to provide a programmable interface for users of the application. Since the user interface would access the application functionality through the same interface, the command language programmer would be guaranteed the same functionality as the interactive user.

---

[2]Figures 3.1 and 3.2 are modified versions of diagrams that originally appeared in Hill's thesis [Hill 87a].

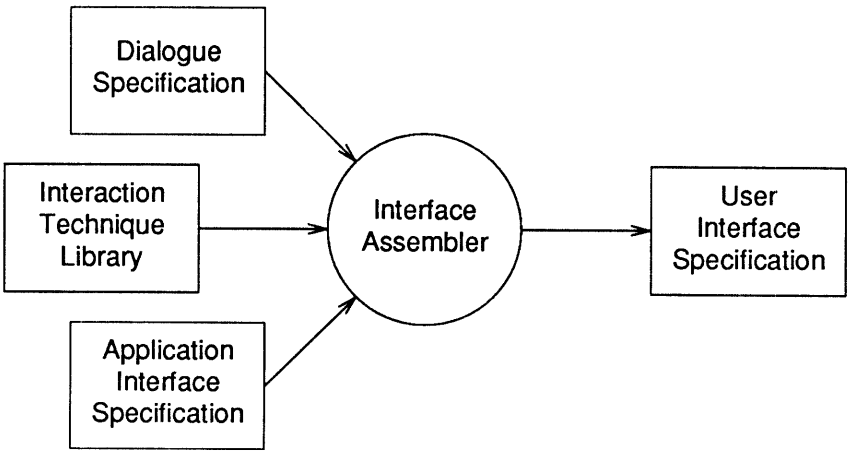


Figure 3.2. UIMS preparation phase

3.1.3 System-wide languages

While application- and package-specific approaches solve the command language program in a limited domain, only a system-wide command language can provide a total solution. A system-wide language, by definition, supports a diverse set of applications. Most implementations obtain this generality by positioning the language at a very low level so that it can see and control all application input and output without the application's knowledge. In text-based systems, this strategy can achieve lexical completeness. As Chapter 2 explained, this approach is not effective in window systems. System designers have tried several new tacks, including extending existing languages, adopting an object-oriented approach, and providing an open system with a single language for both developers and users.

Traditional languages and extensions

Operating system command languages such as the Bourne Shell and the C Shell have proven themselves by being of practical use for thousands of users over the course of several years [Dolotta 80]. Even in a window system, these languages can be used to automate many of the tasks that a user might want to do. However, the command language programmer has no control over windows and other new objects. The programmer does not have the same control over simultaneously executing programs that is available interactively. Mouse input cannot be provided to programs. Straightforward extensions can overcome some of these problems, but others cannot be disposed of as easily. More than trivial improvements are needed to enable these languages to handle graphical output, to invoke and receive results from application subcommands, and to mix subcommands from different programs. However, while weak in terms of completeness, most operating system command languages impose few constraints on applications.

Other work has concentrated on particular aspects of interaction in window systems.

Gill, for example, designed an interwindow communication facility analogous to pipes [Gill 86]. Window-based processes can be dynamically linked by *window channels*. Filters can be inserted in channels to modify the data stream. This system has only been used with terminal emulator windows and does not deal with a key issue, the processing of graphical output.

Object-oriented languages

Much of the research on command languages in the past few years has focused on presenting an object-oriented model to the user. One of the first object-oriented command languages was COLA, a command language for the multi-processor C.mmp [Snodgrass 83]. Smallworld is another experiment in object-oriented programming in a typescript-based system [Laff 85]. Its developers have extended the Rexx command language to include several of Smalltalk-80's key concepts. The principal goal of the system is not to provide another programming language, but to allow users to better organize files, programs, and system commands into one manageable whole. Smallworld has also been used to explore the utility of the object-oriented paradigm in a conventional setting. Ewing has developed a Smalltalk-80 interface to UNIX, including a window interface to the shell [Ewing 86].

Single-language systems

Another candidate for a command language is a programming language whose instructions can be invoked interactively. Smalltalk-80 and Lisp are examples of programming languages that possess some of the most desirable characteristics of programmable command languages. Programs are easy to create, and execution is immediate. They become first-class citizens of the environment, invoked just as system commands are. Smalltalk-80, in particular, is well-suited for window environments. It is embedded in an interactive programming environment with a powerful graphical interface. Its object-oriented paradigm meshes well with the style of interaction presented by the Smalltalk-80 tools [Goldberg 84]. Systems have been introduced commercially based on both Lisp and Smalltalk-80.

However, any single-language system possesses one major disadvantage. Like package-specific languages, these monolingual systems have too restrictive a domain to meet the goals for a command language for a window system. In some cases, the language is available only within a closed system that does not provide access to needed facilities. Sometimes, a single-language system is not appropriate or not available for non-technical reasons. While unsuitable for a system that must support a heterogeneous set of applications, under some circumstances a user interface management or single-language system may be the solution of choice.

Other approaches

Keedy and Thomson have implemented a system in which the command language interface is at the procedure rather than program level [Keedy 85]. The language can call any entry point in a module. Programs are special only in that they are modules with a single entry point. Typed parameters may be passed to procedures invoked from within command language programs. The system is similar to the one proposed in Chapter 4. However, its emphasis is on providing a flexible programming environment in a text-based system.

Gates has suggested that applications communicate with a command language via a common application protocol [Gates 87]. One example of a protocol is the MS Windows Dynamic Data Exchange, based on shared memory. Existing applications must be extended to take advantage of this strategy. Developers would be willing to pay this price, Gates claims, if their programs are able to remain autonomous and distinct and if the programming interface does not require a complete redesign. This proposal focuses on text-based applications and suggests that there should be a one-to-one correspondence between the user and programming interface.

Notkin, Griswold, and Donner have investigated a similar strategy [Notkin 87]. Their major concern is not command languages but application enhancement—how to manage the extension of functionality. They list three requirements for extension mechanisms: procedure-based interfaces, support of multiple extension languages, and speed. In their approach, applications are described as abstract data types. Applications and extensions are compiled and dynamically linked into a process called the Extension Interpreter (EI), just as new functions can be loaded into a programming language interpreter. By using a remote procedure call interface, EI is able to support multiple extension languages—the current system supports C and Icon. EI permits any application to support extensions by simply defining an abstract data type interface. All applications share the same extension mechanism.

Tinylisp, a small lexically-scoped dialect of Lisp, serves as both an application extension language and a system command language for a Modula-2+ environment [Ellis 87]. Tinylisp is written as a Modula-2+ module that can be bound into any application. Both languages can call functions in the other language. Tinylisp overcomes most of the drawbacks of extension languages. By providing one language that can be used by any application, it reduces implementation costs for developers and learning costs for users.

CUSP was a command language expressly designed for a window system, the Xerox Star Information System [Smith 82]. The language failed as a commercial product. Its major problems included an arcane syntax and its failure to permit the user to specify all the objects available interactively [Halbert 85].

3.2 INPUT RECORDERS

The simplest approach to the command language problem is to provide an input recorder that saves commands in a script as the user carries them out. The script can later be played back as if the user had re-entered the commands. The principal advantage of this approach is its simplicity for both users and application developers. A user can easily learn the macro language since its interface is exactly the interactive interface. A recorder provides the user with a visual language—a command language with a direct manipulation user interface [Chang 87]. The command language implementor can install the replay mechanism at a very low level, with little or no change to most parts of the system.

However, input recorders have serious drawbacks, particularly in window systems:

- Recorders cope poorly with the high bandwidth of interaction in window systems.
- Recorders usually do not permit conditional execution because they ignore output of commands.
- While easy to record a simple macro, it is difficult to design general ones.

Most input recorders save all input while in record mode. This practice works well in text-based systems. In an application with a direct manipulation interface, small incremental actions are more common. Commands that require the user to drag the mouse are difficult to record unless the macro facility can determine the exact input events that the application receives. Similarly, the use of time in direct manipulation user interfaces causes problems. A common practice in some highly interactive applications is to implement buttons that carry out some action until the button is released. For example, an editor may scroll text in a window while the user holds a mouse button. The input recorder must obtain the number of input events received and, possibly, the time between these events in order to faithfully reproduce the command. When a system permits applications to poll for input events, these problems worsen since the system (and the recorder) cannot determine what events the application received and processed.

Macro recorders permit the user to record any command invoked interactively. However, as noted in Section 2.4.2, macros in a direct manipulation interface cannot be completely specified automatically because part of the computation—the search method used to select data—is implicit. The user often bases actions on the current state of the application. Because many application objects are continuously visible, no explicit command is required to determine this state. Most recorders do not permit a macro to determine application state nor to invoke commands conditionally based on it.

The greatest drawback to macro recorders is that it is difficult for users to write general-

ized programs. Users are never interested in writing a macro to do *exactly* what they have just done. Even the simplest macro facilities implicitly parameterize some operands. For example, macro recorders in screen-based text editors such as Emacs treat the current position in the text as a macro parameter.

Most recorders permit only "straight-line code," a sequence of commands executed by the user. Macro facilities rarely permit macros to contain explicit parameters, conditional statements, or other control structures. The command language programmer is rarely able to generalize macros or to automate decisions made interactively.

Given the limited generalization available, a macro writer must choose the commands to be invoked in the macro to obtain the degree of generalization desired. In a text editing macro, this often means choosing between cursor positioning commands and search commands. Consider a macro to insert a space at the beginning of a line of text. If the user wants to insert the space at the beginning of the current line, and the text cursor is at the third character, he might use the backward-character command to get to the beginning of the line. If writing a macro, he would probably use the beginning-of-line command instead. If he wanted to add the space to all lines containing a certain word, he might use a combination of a search command and the beginning-of-line command. While the input recorder mechanism is simple to use, designing a macro can be a complex task.

Macro recorders determine the amount of generalization available to the macro writer by deciding how much *context* to save in a macro [Horn 87]. Context is the collection of information that determines the interpretation of a command, and can include cursor location, selection, display layout, and active modes. The simplest approach is to save no context at all. This works well for Emacs because most user commands are relative, not absolute. The same approach can fail miserably in a direct manipulation application that depends on mouse input. Many programs interpret events according to the location of the mouse cursor with respect to regions of the screen. If a macro records absolute cursor location or location relative to a window, then the layout of the display is part of the application context, and the macro may behave differently depending on the screen layout when the macro is executed. For example, the Macintosh permits users to invoke applications by double-clicking on an icon representing the application (Figure 3.3). A macro that performs the same action will fail if the icons are rearranged.

Despite these problems, many researchers and developers have implemented systems that record input. The remainder of this section reviews some of these systems and explains how they have dealt with the problems inherent in macro recorders.

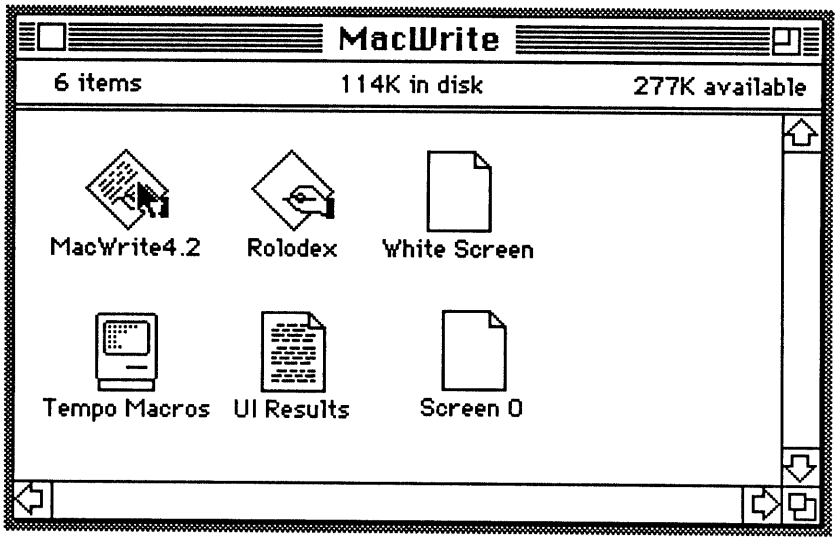


Figure 3.3. Invoking a Macintosh application

3.2.1 Simple recorders

Simple recorders are so easy to implement and use that their use is widespread. Even editors such as Emacs that have powerful extension languages provide macro recorders as well. Input recorders can also be found in user interface management systems and toolkits, providing applications with a simple command language at little cost.

Whimsy is a script recorder for the Andrew window manager [Cohn 87]. The window manager is an attractive level to install a recorder. On Andrew, all programs use the window manager directly or interact with the user through a Typescript window, an editor window that contains a UNIX shell (see Figure 2.4). Therefore, a window manager script can be used to drive any program that runs on the system. Installing the macro facility inside the window manager guarantees that applications will be unaffected by the recording or replaying of a script. In either case, an application sees the usual stream of input events.

The recorder reduces the amount of data recorded by taking advantage of the communication protocol between the window manager and applications. An application specifies to the window manager what kinds of mouse input it is interested in: only down button transitions, down and up, or mouse movement as well as button presses. Whimsy records only the input of interest to the application. Since window manager applications cannot poll for input, most timing problems disappear. However, some applications behave differently if the time between input events changed. For these applications, Whimsy provides a "real-time" mode which sends input events to the application at the rate they were recorded. This mode is not always reliable. Because Whimsy is part of the window manager, it can observe when input events are generated, but not when they

are received by an application. Even in real-time mode, applications may receive events with slightly different time delays, causing subtle changes in behavior.

Whimsy goes beyond input recording in order to support the examination and manipulation of command results. Users can record window manager output commands generated by the application. These commands tell the window manager to write text or draw a line or some other graphic. Even though these commands are at the syntactic rather than lexical level, they are still too low-level and too great in number to be useful in scripts.

Whimsy's biggest problem is that it cannot reliably execute scripts because different screen layouts would produce different results. Because the Andrew window manager tiles windows rather than overlaps them, windows are often different sizes. (In most overlapping window managers, applications can specify that they need a specific size window.) Moreover, applications attempt to format themselves according to the size of their window by adjusting the size of regions and fonts.

3.2.2 Recorders with editing

Several systems have attacked the generalization problem by letting users edit a macro once it is recorded. The editor should present scripts in an understandable and easily modifiable format. The system must provide control structures that allow users to parameterize macros and test conditions. If the editor's user interface is complex, then the user may be better off writing scripts by hand rather than using the recorder.

Halbert implemented a system called SmallStar as part of his thesis on programming by example [Halbert 84]. Users record macros that consist of straight-line code. Later, they can edit their programs to parameterize operations and add conditional statements.

SmallStar presents a recorded script to the user as a form (Figure 3.4).

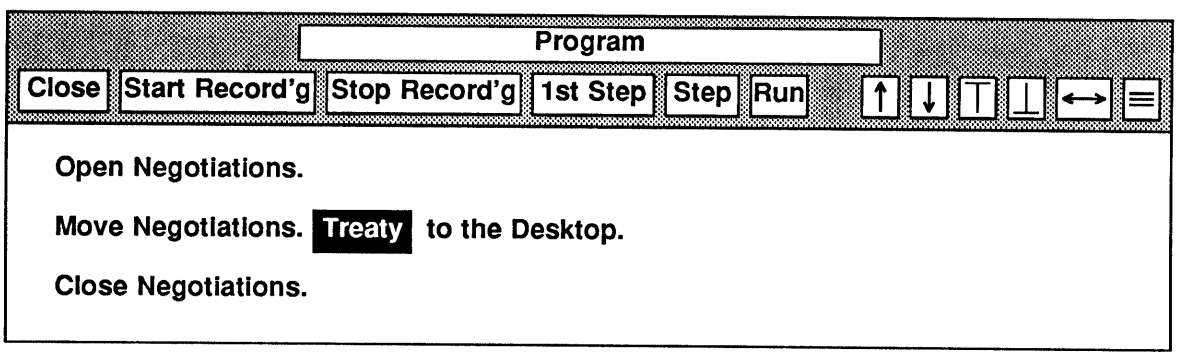


Figure 3.4. A SmallStar macro program

The system displays parameters, called *data descriptions,* and predicates of conditional

statements as property sheets associated with the program. A data description presents a set of search methods that can be used to specify an object. Each method includes one or more parameters that determine the particular data. For example, Figure 3.5 shows a data description for the Treaty document selected in Figure 3.4. The description includes both the pattern matched, version, and age of the file, and identifies the active method. The user can modify the description to generalize the pattern or make the data a macro parameter by choosing the prompt method.

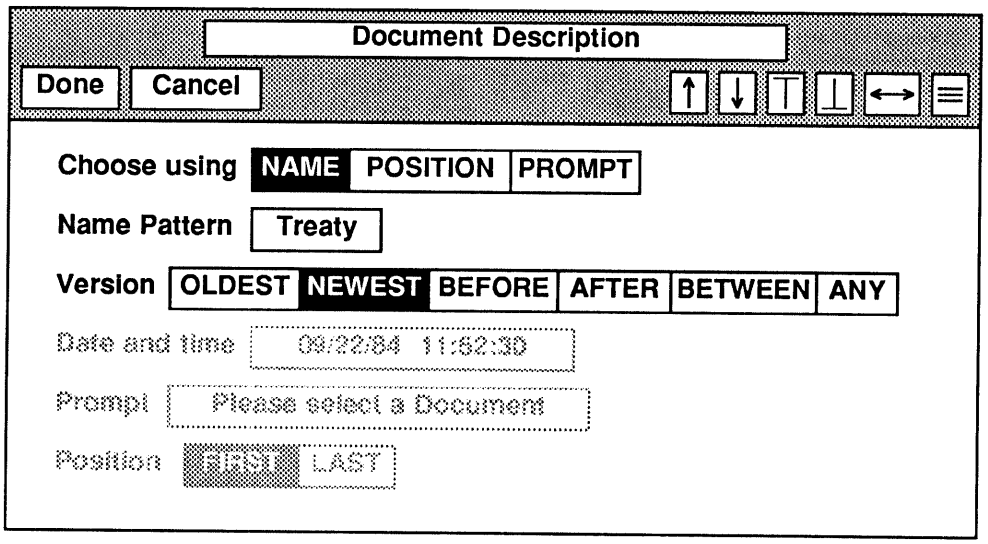


Figure 3.5. A SmallStar data description

Halbert dealt with the context problem by recording semantic rather than lexical information. Note that the operations recorded in the script in Figure 3.4 do not specify keystrokes or mouse button transitions or locations. Halbert modified each application in his system to translate low-level input data into SmallStar virtual machine commands.

Tempo is a commercially available input recorder for the Macintosh [Tempo 86]. As in SmallStar, users may edit recorded macros. Rather than examining a static description of the program, a user plays back the macro, stopping at the point at which he wants to make a change. He can add additional commands, an interactive prompt, a conditional test, or a loop. A test examines the contents of the Clipboard, a temporary storage location for text and graphics. Figure 3.6 shows the tests Tempo provides.

Unlike SmallStar, Tempo operates at the lexical level. Its designers had little choice since generality was a key goal—the system is intended to work with arbitrary Macintosh applications. Use of the Clipboard permits users to automate some decision-making, but tests are limited to text visible on the screen that can be copied to the Clipboard. Graphics are off-limits. Furthermore, as in other lexically-based recorders, macros fail when a user replays location-dependent actions in a window whose contents have been rearranged.

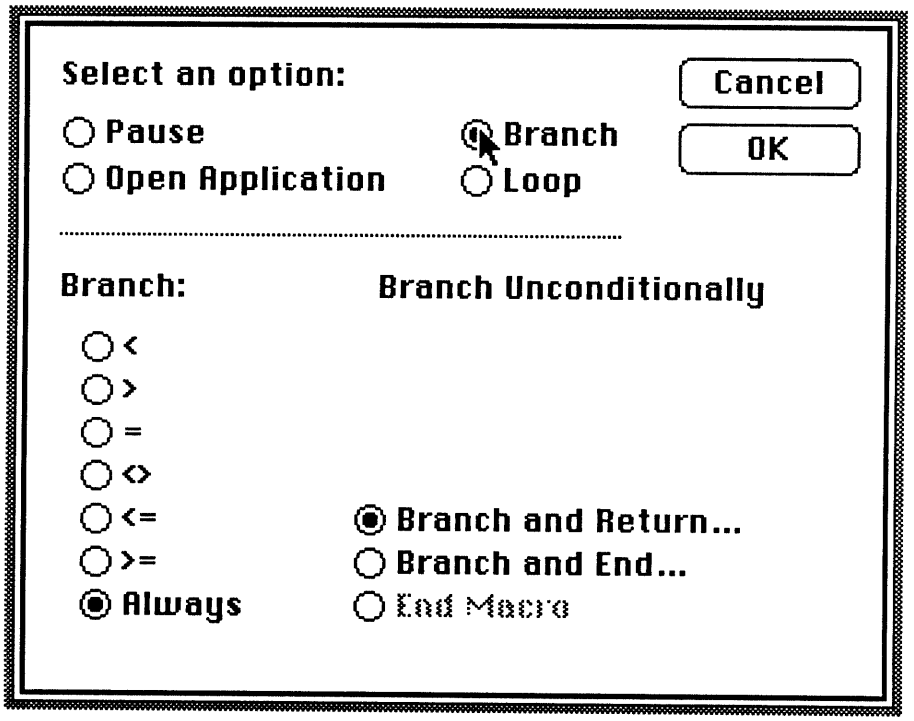


Figure 3.6. Tempo's conditional tests

3.2.3 Recorders with inferencing

While editing example scripts has been successful in permitting macro writers to express conditionals and generalize programs, it is a compromise between pure programming by example and normal programming. Some systems have tried to provide generalization by inferring the user's intentions. Typically, the user records several examples of an action, and the system determines what items are parameters by finding differences in the examples. Editing by Example is a system that allows users to create text editing scripts following this approach [Nix 85]. Because its domain is limited, Editing by Example is more successful than most inferencing systems. These systems are often hard to use because the user must guess at the kind of example required to build a program, making programming by example as difficult as normal programming.

3.3 SUMMARY

This chapter has examined a variety of approaches to the command language problem. Each approach has sought and achieved some balance of feasibility, generality, ease of use, and efficiency. While some systems have been more successful than others, all have been stymied by the difficulties described in Chapter 2.

The analysis in Chapter 2 and the systems in this chapter have shown that feasibility and

ease of use can be obtained only at the cost of generality. Furthermore, only a system that functions at a semantic level will be able to achieve completeness, manipulating graphics as well as textual data, capturing the kinds of decisions users make interactively, and combining commands from multiple applications. Requiring semantic completeness does not rule out the use of input recorders but does require a mapping from the lexical input provided by a recorder to a semantic representation. The next chapter will make use of these observations to present a new approach.

4 Whisper

The goal of this thesis is to devise a practical strategy for implementing *complete* command languages for window systems. Previous chapters have expanded on this goal, detailing the requirements of such a command language and the shortcomings of existing systems. This chapter describes Whisper, a new approach to the problem.

All command languages provide some interface between the command language system and the application. Command languages usually place this interface at the lexical level, allowing applications to accept input from the command language transparently—the application cannot distinguish a command language program from an interactive user. The previous chapter showed that this approach fails for direct manipulation applications. Another weakness of existing systems is that they do not support the extended functionality provided by window systems. As described in Chapter 2, users of window systems typically execute multiple programs concurrently and transfer data between running applications. A command language for a window system should support similar functionality. At the same time, the system should impose minimal requirements on applications, making it possible to support a variety of programs with different styles or interaction techniques, possibly written in different languages. Only by providing an open system will it be practical to modify existing applications to use the command language.

Whisper is a prototype command language system that demonstrates the viability of a command language in a window environment in which most applications support direct manipulation interfaces. Its most significant features are that:

- It supports command language interfaces defined at the *semantic* rather than lexical level.
- It permits a command language program to combine subcommands from multiple applications.

While other systems support the definition of application interfaces at the semantic level,

none have used them to provide a command language interface. Previous attempts to provide command languages for window systems have operated at the lexical level. Moreover, none have supported multiple, active, heterogeneous applications within a single command language program.

This chapter describes Whisper in greater detail. The next section introduces its basic components and the model of computation underlying Whisper. The following section presents a simple example. Then, the prototype system is examined from the user's and developer's perspectives. Some details of the implementation follow. The chapter concludes by identifying extensions to the prototype system, the most important being a programming-by-example front-end.

4.1 A MODEL OF COMPUTATION

While the demonstration system consists of several distinct parts, Whisper is really a framework for applications, the definition of a structure that supports a command language in a window system. This framework is independent of a particular user environment and outer language, and the inner language is determined by participating applications. However, it does rely on a specific model of computation—the client-server paradigm.

Under the client-server model, a process provides resources to clients who make contact with it. For example, on many UNIX systems, an "FTP server" allows users on remote computers to request the transfer of files between the server's machine and the user's machine. Users execute a client program that contacts the server, carries out the user's requests using a protocol defined by the server, and terminates the connection. The operating system supports the establishment of well-known addresses for standard services, allowing clients to find the correct server.

Whisper provides users with an interpreted command language. The interpreter is a client process of server applications. These servers provide access to data or special forms of computation. For example, a mail reader provides access to a user's mail database, while a text editor makes available documents and generic editing functions. A simple calculator provides arithmetic services, while a calculator with memory provides access to data as well as to functions. A statistical function package is no more than a sophisticated calculator. The Polylith, a research project at the University of Illinois, supports a similar model with emphasis on mathematical computation [Purtilo 85].

A Whisper application defines its services with an interface specification that describes, at a semantic level, the application's objects and operations. Command language programs access these services using interface operations. The interpreter is a separate

process, independent of any application, and can easily communicate with multiple applications. This structure, shown in Figure 4.1, makes it possible for the user to write macros that require the services of more than one application. In this respect, Whisper differs from application-specific extension languages.

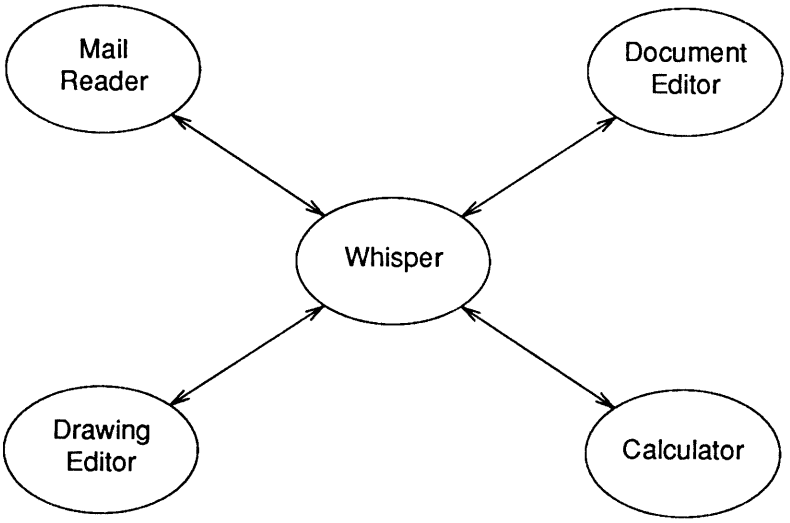


Figure 4.1. The Whisper process structure

Application developers can view the system in either of two ways. As in programs written on top of user interface management systems, the services provided by the application can be separated from the user interface. In this view, shown in Figure 4.2, the user interface and command language are both clients of the application, although the user interface will most likely be part of the same process as the application.

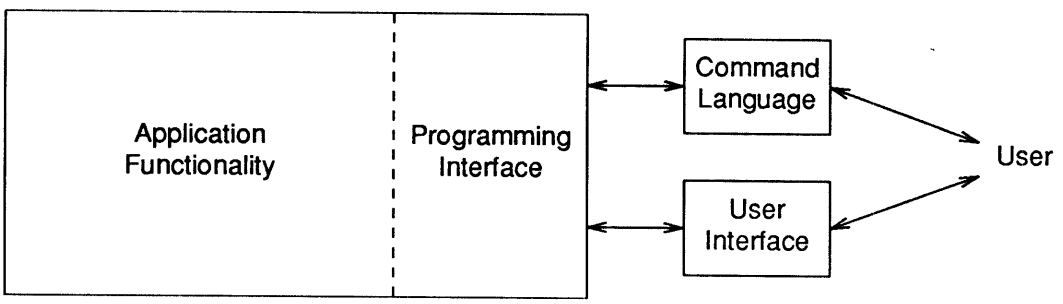


Figure 4.2. User interface and command language as clients

Given an appropriate programming interface, the command language can serve as the development language for the application's user interface (Figure 4.3). Alternatively, the user interface and programming interface can be viewed as completely independent (Figure 4.4). Unlike the first approach, this design does not guarantee completeness, but it is the most flexible and will sometimes be the only option for applications retrofitted to use Whisper, as was the case for the applications to be discussed in Chapter 5. As the

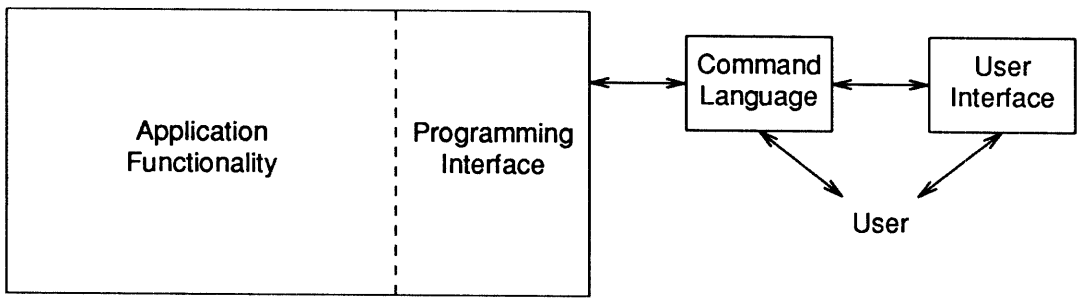


Figure 4.3. User interface implemented by command language

principle of separation of user interface and application functionality gains wider acceptance, more applications will fit the first model even when designed without Whisper in mind [Szekely 88].

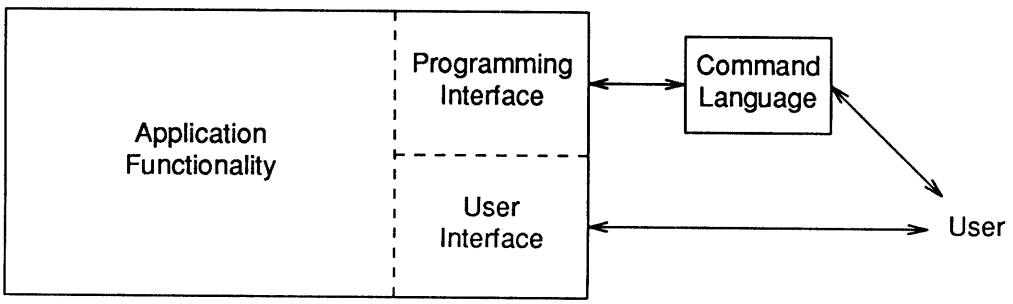


Figure 4.4. Separate programming and user interfaces

There are some important differences between Whisper and the standard client-server model. Server programs usually have a single interface—a network connection. There is no direct user interface. Instead, users must communicate with servers via client programs that do have user interfaces. Servers frequently are designed to provide service to more than one client at a time by providing a separate connection for each client. A connection is nothing more than a collection of data that the server uses that is specific to a client. If multiple clients access independent data, clients usually cannot detect that they are sharing the server. If shared data is involved, the server must ensure that client requests do not conflict.

Interactive users do not expect to share an application with other clients, unless they specifically invoke those clients. Few workstation applications support multiple connections—general databases are the most important exceptions. Because applications are written to support a single client, it is not possible to isolate enough state per client to make multiple connections transparent to users. (One source of errors in the implementation of Whisper applications has been an implicit assumption in these applications that there is only one thread of control.)

Despite these differences, the client-server paradigm is a useful model of computation for

Whisper. A developer structures an application so that it can provide a service in the form of access to functions and application-specific data. A command language programmer sees a language with access to a set of services.

4.2 AN EXAMPLE OF WHISPER USAGE

This section provides a quick introduction to Whisper by describing its use with a single application, a four-function calculator. The calculator, whose display is shown in Figure 4.5, emulates a real calculator without operator precedence. The user operates the calculator by clicking on the desired button or pressing the corresponding key.

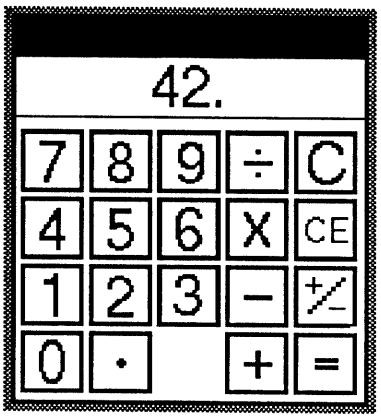


Figure 4.5. The Andrew calculator display

The user interface for the prototype Whisper is a Lisp interpreter. A special function allows the user to start up an application from inside the interpreter. The application behaves as usual, except that it can accept input from the interpreter as well as from its user interface. Once the application is running, Lisp programs that contain application calls can be loaded and evaluated as if they were ordinary Lisp functions. Invisible to the user is a mechanism that executes a remote procedure call (RPC) in order to evaluate application functions. As calculator functions are executed, the calculator display changes to reflect its current state.

The command language interface to the calculator is straightforward: there is a one-to-one mapping between the semantic-level user interface operations and application interface functions. Functions permit the command language programmer to enter digits and apply the four arithmetic functions, invoke the equals operation, and clear the display. The interface also includes a function to enter a number instead of a single digit. The following is an excerpt from the interface:[1]

---

[1]Section 4.4 describes the interface specification language. Appendix C.1 contains the complete calculator interface.

```
(defop calc-digit KeyDigit (current-calc character) () float)
; (calc-digit digit-char) → current-value

(defop calc-function KeyFunction (current-calc character) () float)
; (calc-function function-char) → current-value
; function-char may be one of +, -, x, /

(defop calc-equals KeyEqual (current-calc) () float)
; (calc-equals) → current-value

(defop calc-clear KeyClear (current-calc) () float)
; (calc-clear) → nil

(defop calc-enter Enter (current-calc float) ())
; (calc-enter number) → nil
; This is a function written only for the programming interface.
; It is more convenient than using calc-digit, calc-point, and calc-negate.
```

Each defop statement lists the name of the operation, the C function that implements it, the in parameter types, the out parameter types, and the return type, if any. Current-calc is a special parameter that is automatically supplied to the C routine. Current-value is the number displayed by the calculator. Comments (prefixed by semi-colons) describe how each function is called and what is returned.

Suppose a command language programmer, unaware that Lisp includes arithmetic functions, decides to implement his own set. The following program packages the add function in a more convenient form than is provided by the raw calculator interface.

```
(defun calc-add (&rest args) (calc-add-1 args))   ; &rest packages arguments into a list

(defun calc-add-1 (args)
  (calc-enter (float (car args)))                 ; enter the first argument
  (if (cdr args)                                  ; if there are more args,
    (progn                                        ; invoke + and recursively call calc-add-1
      (calc-function #\+)                         ; #\+ represents the character '+'
      (calc-add-1 (cdr args)))
    (calc-equals)))                               ; else invoke = and return the sum
```

Calc-add returns the sum of its arguments. Calc-add-1 is a helper function that enters each argument and invokes the interface's add function. After the last argument is entered, calc-add-1 invokes the equals function and returns the sum.

This example has briefly shown what a Whisper interface and program look like. The next two sections explain Whisper further, in terms of how end-users and application developers use the system.

4.3 USER'S VIEW

The foundation of the system is XLisp, a small Common Lisp subset [Betz 86]. Modifications to XLisp allow users to access application programs through the interpreter. A few special functions provide control over starting up new applications, switching between active ones, and exiting applications. The start-up command begin-application loads the command language interface, invokes the application, and creates a connection between it and the interpreter. The user may have multiple applications active at the same time and use the select-application command to switch between them. The currently active application receives all application interface function calls. The end-application command sends a special quit message to the application. Appendix A describes the complete set of XLisp extensions.

Associated with each application is a programming interface. An interface defines functions and variables implemented by an application. The interface functions can be those normally available via menus, mouse clicks, or keys, as well as lower-level functions not otherwise accessible. Functions may have in and out parameters; functions with multiple out parameters return a list of results. Parameters may be integers, floats, characters, strings, booleans, or *objects*. Whisper objects are similar to operating system capabilities [Strom 83] and Smalltalk objects [Goldberg 83], providing controlled access to private data.[2] A command language program can manipulate this internal data by using objects as parameters in interface operations. For example, a drawing editor may export objects that represent the graphical entities in a drawing and provide a set of operations on these objects.

Whisper users may treat the command language as the extension language for a particular application, as the calculator example demonstrates, or they may use it to communicate between applications by passing information from one active application to another. Of course, they may still take advantage of existing communication methods such as cut and paste.

Applications need not be started up from the interpreter; they can initiate communication themselves and request the evaluation of XLisp functions. If an application chooses to do this, it will usually provide the user a way to execute Whisper commands directly. The calculator, for example, could allow the user to create new buttons that invoke functions defined with the command language. In the course of evaluating a function, the interpreter will send back requests to the application to execute operators it finds. The func-

---

[2]Unlike Smalltalk-80, Whisper does not provide an inheritance mechanism for defining new classes of objects.

tion may contain calls to other applications as well, allowing one application to use the services of others. This capability is the key to the implementation of the mail reader described in Section 5.5.

4.4 DEVELOPER'S VIEW

Whisper requires that an application developer: (1) specify a programming interface, and (2) modify the application so that it can process interface requests. The difficult (and interesting) task is designing the Whisper interface. Once this is done, the developer can trivially incorporate the interface into the application.

The quality of application interfaces determines the success of a Whisper implementation. A developer has great latitude in specifying the programming interface to an application. The system only provides a mechanism for interface specification; this thesis provides guidelines that can aid developers in defining good interfaces. Chapters 5 and 6 discuss this subject, while the rest of this section describes the mechanism Whisper supports.

An application developer specifies an interface by defining its constituent functions and variables. Using a simple Lisp-like language, the developer defines each external operator and variable name, the corresponding implementation routine or variable, and the routine parameter and return types. For example, the full calculator interface, found in Appendix C.1, includes two variables and eight operators. (Appendix B spells out the details of the interface specification language.) Whisper generates the code needed to implement the necessary remote procedure calls based on the interface description. While C is the only implementation language currently supported, the system is designed to support any number of implementation languages.

The interface specification is similar in form to that required by user interface management systems such as MIKE [Olsen 86]. In addition to the standard parameter descriptions, the interface may specify that a parameter is provided automatically by the application or that it should assume a default value if the user does not specify it. This reduces the number of trivial routines that the developer must write that only serve to package up lower-level routines.

The prototype system supports a few basic types of parameters and variables, and maps these types between the command and implementation languages. If developers need to export a function that uses unsupported parameter types, they must wrap this function inside another that translates a legal type into the one used. Alternatively, a function can export an indirect reference to an application data type through the use of objects.

To export an object, the application interface must first declare its class, or type. Once the object type is declared, it may be used wherever a standard type is used: as the type of a variable, function parameter, or function return value. The system guarantees that an input parameter that is an object is valid, that is, that it corresponds to the previous value of an exported variable, an output parameter, or a return value that is an object of the same type. Applications may invalidate objects that have been exported. Once the object value is invalidated, Whisper will not accept it as the value for a variable or input parameter. An application may find it necessary to invalidate an object if the corresponding application object is deleted.

A stand-alone program, Wgen, reads the interface specification and produces two code fragments: one that can be read by XLisp and one that should be included in the application. The interface generator enables the system to hide much of the underlying remote procedure call mechanism. It is similar to tools such as Courier [Birrell 84] and Matchmaker [Jones 86] that automate the implementation of generic remote procedure calls.

The Lisp code generated by Wgen defines macros that make the interface operators appear to be ordinary Lisp functions. It also declares external application variables and identifies the version of the interface that it defines.

The C fragment created by Wgen defines an initialization procedure that should be called by the application. The code defines data structures used by Whisper library routines to respond to function calls and variable accesses by checking the number and types of input arguments and returning the requested data—operator results or variable values. The code fragment is included in the C module that implements the interface. For example, the code produced by Wgen given the calculator interface specification can be found in Appendix D.1.

A set of library routines handle the house-keeping necessary to process incoming function requests and to package up outgoing requests. The application must call a special routine that checks if a request is waiting to be processed just as it must check if user input is pending. However, if the application uses a user interface toolkit that provides *external control* [Green 85], this call can be hidden in the toolkit. In a toolkit with external control, the user interface calls application functions, and the toolkit hides the "main loop" of the program inside the user interface. Both the checks for user input and for Whisper requests can be placed in this main loop. The toolkit used by the calculator does in fact provide external control; the Whisper routine that checks for requests is not visible in the calculator implementation shown in Appendix D.

Whisper interfaces can mimic the module structure of an application. If the functions

that define the external operations occur in several modules, a sub-interface can be specified for each module, and all will be loaded when the application is invoked. If modules are shared by applications, each application automatically inherits the interface to each shared module. For example, the calculator, as well as the drawing and document editors described in Chapter 5, inherit interfaces defined by the underlying toolkit.

4.5 IMPLEMENTATION DETAILS

I implemented the prototype system in the Andrew environment developed at Carnegie Mellon University's Information Technology Center [Morris 86]. Andrew is a UNIX-based window environment for workstations. It includes two levels of user interface tools: a network-based window manager provides low-level access to input devices and the display, while a toolkit called the Base Editor handles many of the details of the user interface for applications. I used an experimental version of the Base Editor known as BX. The window manager, BX, the XLisp interpreter, Wgen, and the Whisper library are all written in C. Whisper can be incorporated into any C application on Andrew, but some features are useful only in BX applications.

There are two major components of the Whisper system. The first, described in the previous section, is Wgen, the program that translates interface specifications into the C and Lisp code fragments needed to support communication between the command language interpreter and applications. The other, and more significant, component of Whisper is its run-time system. This section explains some details of the run-time system and the implementation of objects.

4.5.1 The run-time system

The Whisper run-time system consists of a command language interpreter and the Whisper library included in participating applications. As noted above, the command language interpreter is a version of XLisp extended to provide functions that initiate and terminate connections to external applications, and invoke operators and access variables defined by external applications. An RPC package called R [Kazar 87] handles the low-level communication details. R transmits data between processes using the External Data Representation protocol [XDR 86], providing machine and language independence. Both the XLisp interpreter and the Whisper library make use of LWP, a light-weight process package [Rosenberg 86]. This allows both XLisp and Whisper applications to process RPC requests while waiting for user input. It also permits an application to process Whisper requests even while its own request is pending.

The figures in this section illustrate interaction between a user, the XLisp interpreter, and

a typical Whisper application called Delta, an Andrew drawing editor.[3] An arrow represents communication between two entities: user input, a remote procedure call, or a system call. The diagrams show the relative ordering but not the absolute duration of events. Return values are not shown in the diagrams.

The first three figures describe what occurs when the user interacts with the interpreter, either interactively or by submitting a program to it. This corresponds to the use of Whisper as a command language for the entire system. The remaining figures show the user interacting directly with an application. In this situation, Whisper functions as an extension language for the application. The user interacts indirectly with the interpreter by invoking functions in the application that happen to be implemented in XLisp rather than in C.

Figure 4.6 describes the sequence of events that occurs when an application program is begun from within XLisp. To start up an application, the user issues the begin-application command. In response, XLisp uses the standard UNIX call system to actually create the application process. It then waits for a ready-to-interact message from the application. Before Delta issues this message, it initializes its Whisper interface. Its interface consists of several subinterfaces. Each interface sends its own load-new-interface message to XLisp, prompting XLisp to load the Lisp file that corresponds to the C interface defined within the application. Once XLisp receives a ready-to-interact message, the original begin-application call returns.

Figure 4.7 describes the evaluation of a function defined by the application's interface. When the user types an expression to the XLisp interpreter that includes an interface function, XLisp sends a message containing the name of the operator and any arguments the user has supplied. The application processes the arguments, invokes the corresponding C routine, and returns a success code and the routine's out parameters and return value. Accessing a variable value is much like a function call with no arguments and a single return value. Setting a variable is similar to a function call with a single input argument.

The C routines provided by the Whisper library carry out the bulk of the work needed to execute remote operations. These routines unpack the message, determine if the operator is legal, compare the types of the arguments found to those expected,[4] execute the application routine corresponding to the function name, and pack up and send back the results or an error. Under ordinary circumstances, the application routine cannot distinguish between an ordinary invocation and invocation by Whisper.

---

[3]Refer to Section 5.3 for a description of Delta.

[4]More extensive (semantic) checking must be done in the application routine itself.

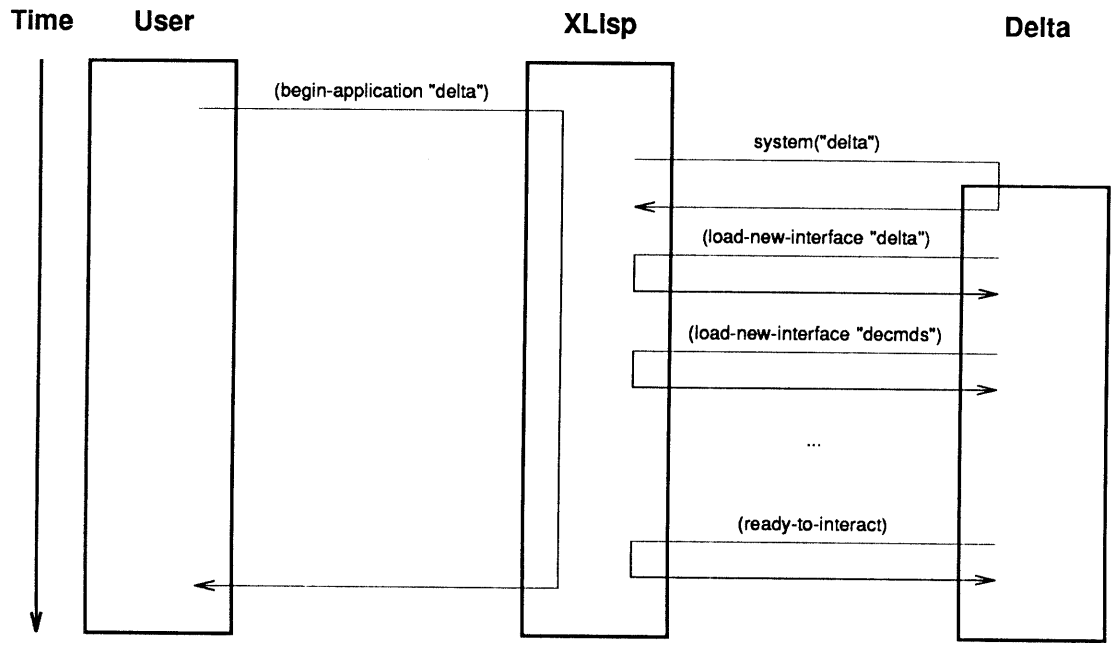


Figure 4.6. User invokes Delta from XLisp interpreter

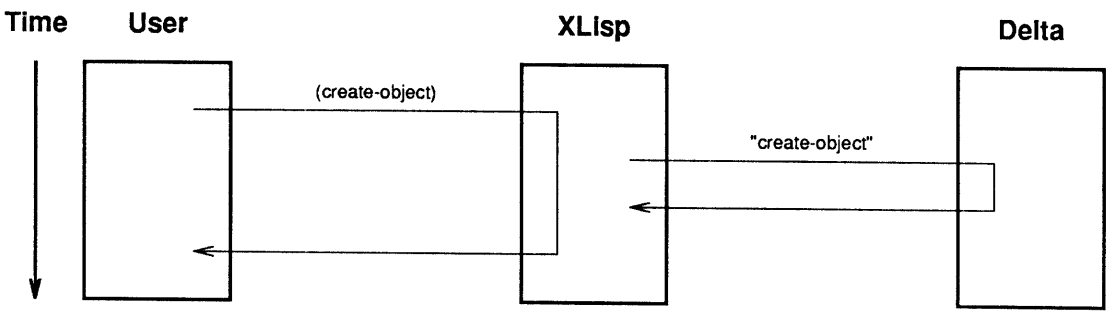


Figure 4.7. User types (create-object) inside XLisp interpreter

Figure 4.8 shows how a user exits an application from inside the XLisp interpreter. Whisper defines the command end-application that invokes a quit operation defined by the application. (All applications must define a quit function which is expected to clean up the program's state before exiting.) If the operation is successful, XLisp marks the application as inactive.

In Figure 4.9, the user starts up the application as he would in a system without Whisper, in this case by invoking the delta command in the Typescript. The Typescript, a terminal emulator that runs the UNIX shell, starts up Delta much like XLisp did in the first example. Interspersed with Delta's standard initialization sequence are remote procedure calls to XLisp, which is listening for messages sent to a well-known address. (The Whisper library determines this address.) The hello-application message informs XLisp that a new Whisper application is starting up. As in the previous example, Delta then sends a sequence of load-new-interface messages that describe its interface.

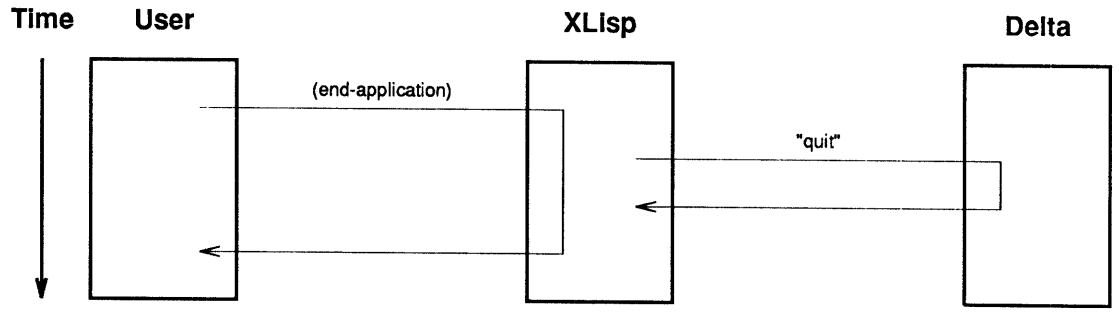


Figure 4.8. User exits Delta from inside XLisp interpreter

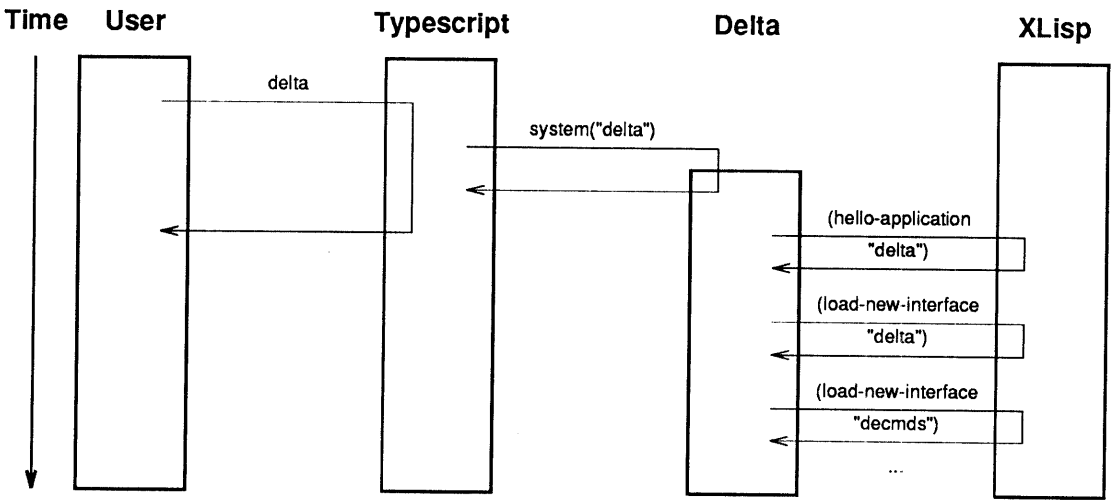


Figure 4.9. User invokes Delta from the Typescript

The next figure shows the user invoking a command written in XLisp that includes several functions defined by Delta's interface. This operation may have been a menu option or bound to a key. XLisp evaluates the expression that Delta sends to it. In the course of evaluation, XLisp sends several function calls back to Delta. A light-weight process created by the Whisper initialization routine handles the execution of these functions within Delta while another awaits the return of the original draw-ellipses function. Note that it is also possible for the command to cause XLisp to invoke functions in other applications.

The last figure shows what occurs when the user exits Delta. As part of the application's termination, a message must be sent to XLisp. This message enables the interpreter to destroy data structures that are no longer necessary since this instance of Delta will not communicate further with XLisp.

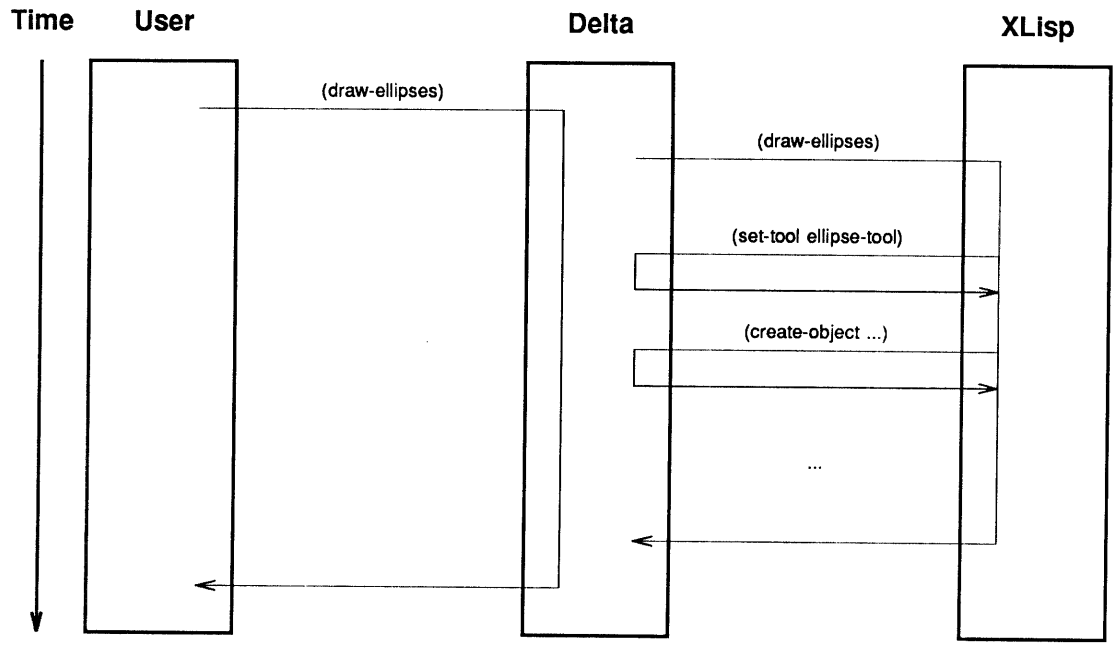


Figure 4.10. User executes (draw-ellipses) inside Delta

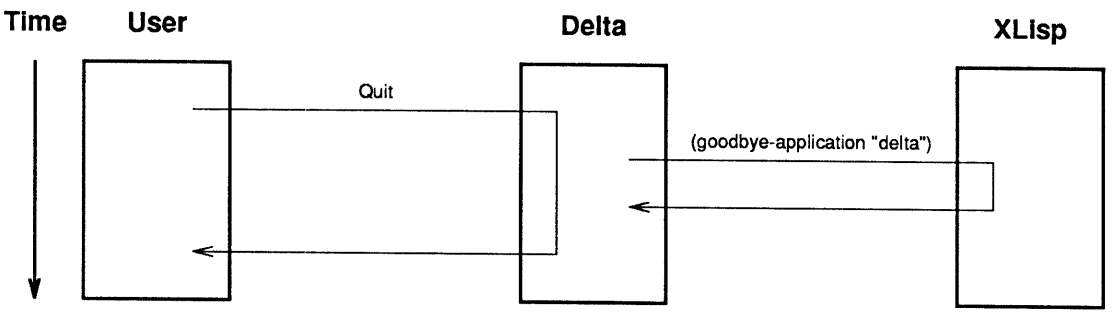


Figure 4.11. User exits Delta from inside Delta

4.5.2 Managing object references

As described in Section 4.3, *objects* provide applications a type-safe method of exporting references to internal data. Each time an application returns a value to the interpreter that is a object, Whisper stores the object and its type in a hash table. Whenever an application function requires an input parameter that is an object or an application variable is set whose type is an object type, Whisper looks up the incoming value in the hash table. If the value is not found or is of the wrong type, the call fails and the Whisper library returns a special error code to the interpreter. Under normal circumstances, the application is unaware of the object table. However, if an application needs to invalidate an object, it calls a Whisper routine that removes the object from the table.

In the current implementation, an object is just the address of the internal data. Returning

an index to the table rather than the address itself would permit an application to move the data without invalidating the object. This is useful since there are situations, especially in C programs, where an object may be repositioned in memory.

4.6 A PROGRAMMING-BY-EXAMPLE INTERFACE

An important feature of many command languages is that the interface provided by the command language is similar to the user interface, enabling users to transfer their knowledge of interactive commands to command language programming. A full implementation of Whisper would provide this feature by making available a programming-by-example front-end to the system. This front-end would enable naive users to treat Whisper as a simple keyboard macro system. The recorder would generate Lisp code that corresponds to the sequence of input events invoked by the user. More sophisticated users could use this mechanism as a starting point for complex programs.

This section discusses two extensions to Whisper: a simple recorder that saves and replays macros, and a more sophisticated recorder that supports macro editing. While I did not implement either extension, the Whisper design provides the lower level facilities required by these recorders.

4.6.1 A simple recorder

A Whisper recorder would appear to the unsophisticated user no different from an ordinary recorder. The user enters a remembering mode during which all input is recorded. Recorded macros can be played back at any time. Working in conjunction with the system's window manager, the recorder issues commands to one or more applications in multiple windows.

While most recorders save raw input events, the Whisper recorder saves the application interface commands bound to these events. This assumes, of course, that there is a well-defined mapping from key and mouse events to interface functions. This mapping could be provided if more information is included in Whisper interface specifications. The product of a recording is an ordinary Whisper program that defines an XLisp function. The user can modify the function by editing its text and adding commands, parameters, and control structures.

Consider a Delta user who wants to record a macro that creates a box inside an existing one. Given a selected box (Figure 4.12), he uses the Duplicate operator (Figure 4.13), which creates a new box slightly offset from the original (Figure 4.14).

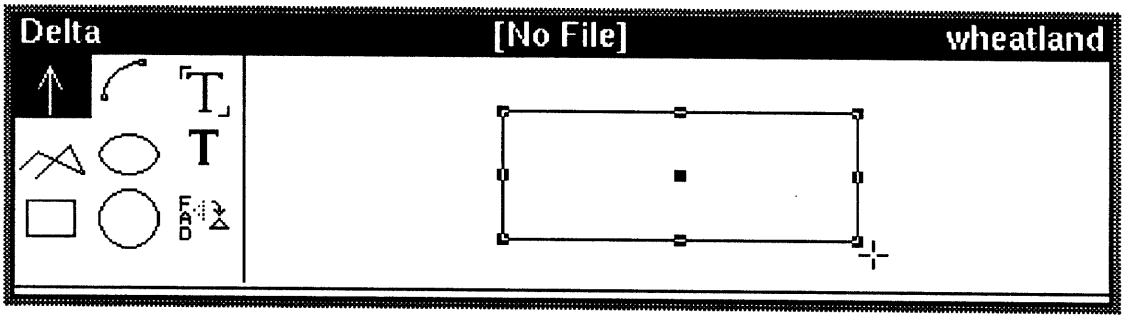


Figure 4.12. A selected box

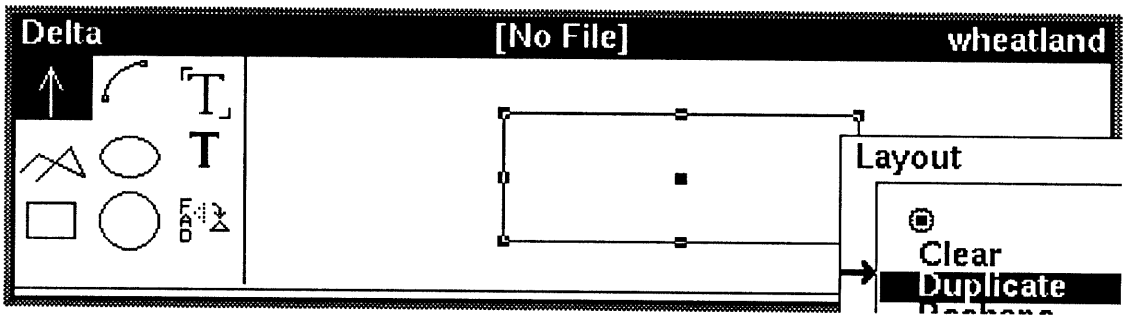


Figure 4.13. Duplicating the box

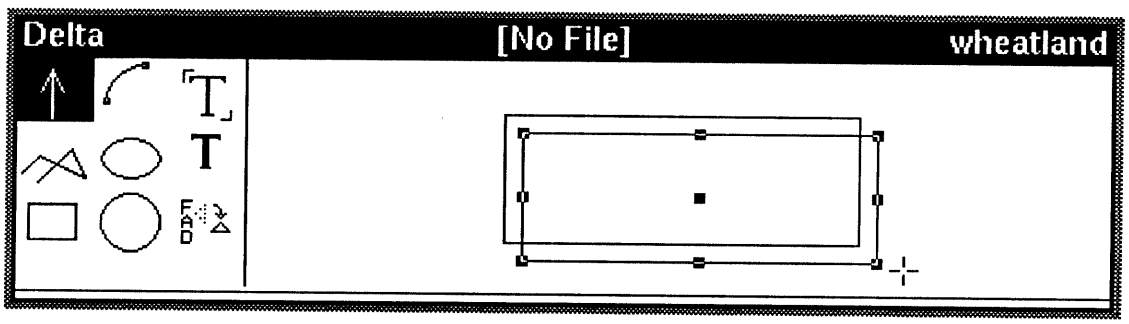


Figure 4.14. The box and its copy

He then moves the bottom right corner of the new box inside the original so that the distance between the two boxes is constant on all sides (Figure 4.15).

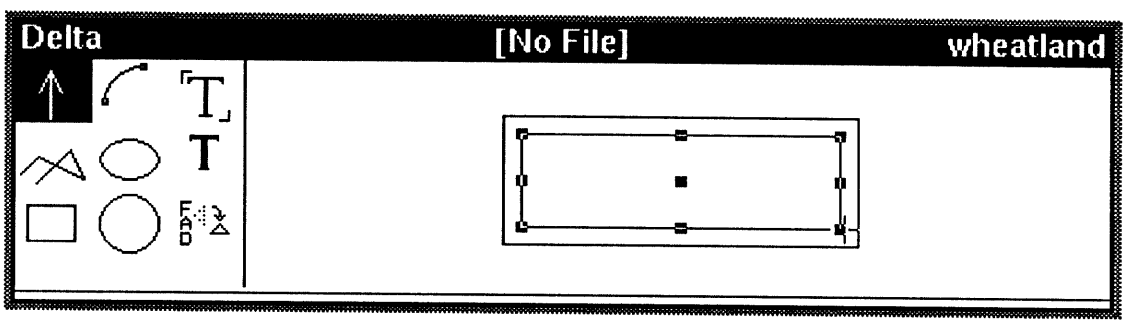


Figure 4.15. Adjusting the new box

Finally, he deselects the new box (Figure 4.16).

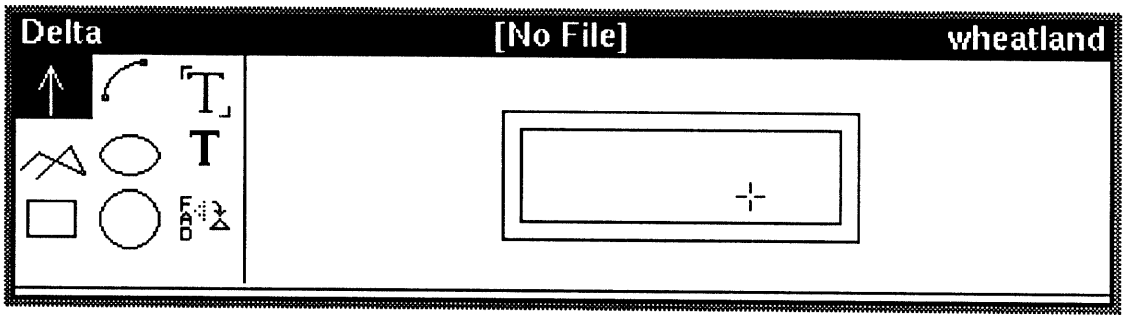


Figure 4.16. The final result

Delta records the following program in response to this series of steps:

```
(defun create-double-box
  (ccmds-duplicate)                    ; create a copy of the selection (the box)
  (ccmds-move-handle 4 -10.0 10.0)     ; move southeast handle left, up 10 units
  (ccmds-deselect-object))
```

The user supplied the function name. Menu commands such as Duplicate are translated into the corresponding interface function. Delta translates mouse commands according to the specific function carried out. Rather than recording a generic mouse-operation function, Delta records move-handle. This translation is application-specific and must be done by Delta, which passes the information on to the Whisper interpreter. The move-handle command specifies which handle (4 corresponds to the southeast corner) and the distance to move in Delta drawing units.

4.6.2 A recorder with editing

The Whisper user interface could be further enhanced by incorporating a programming-by-example system more sophisticated than a simple recorder. A system like Halbert's SmallStar, described in Section 3.2.2, meshes well with Whisper. The addition of some SmallStar features would provide Whisper users with a direct manipulation interface to the command language, yet allow them to change macros, to generalize them, and to apply conditional tests and iteration. Halbert's techniques can be applied to Whisper because SmallStar and Whisper have a similar view of applications. Both represent the user interface at a semantic level, recording application functions rather than lower-level lexical information.

This section describes how aspects of SmallStar can be added to Whisper: data descriptions, control structures, and command elision. It discusses one area where the Whisper design diverges from SmallStar—the selection of objects. Finally, this section presents an implementation of recording and replay.

SmallStar features

More information must be included in Whisper interface specifications for Whisper to support data descriptions. Descriptions can be generated automatically for some common objects, including text strings and documents. Others, such as Delta's drawing elements, must be defined by the interface. The extended interface must describe the search methods for each type of object that will generate a data description. The additional information needed consists of a display format, C code to generate parameter values from an object instance, and C code to generate a set of objects from user-supplied values.

Figure 4.17 shows a prototype Delta data description. The description could have been generated when the user selected a line while recording a macro. Position in Drawing is computed to be the center of the line. The Box is defined to be a bounding box. These two search methods are analogous to operations available to the user:

- selecting an object by clicking on it, and
- selecting objects by enclosing them in a box.

The Prompt search method, if chosen, will ask the user to select an object when the macro is executed. The interface specification includes the name of a C routine that will handle this selection. Section 5.3.1 further discusses the design of Delta data descriptions.

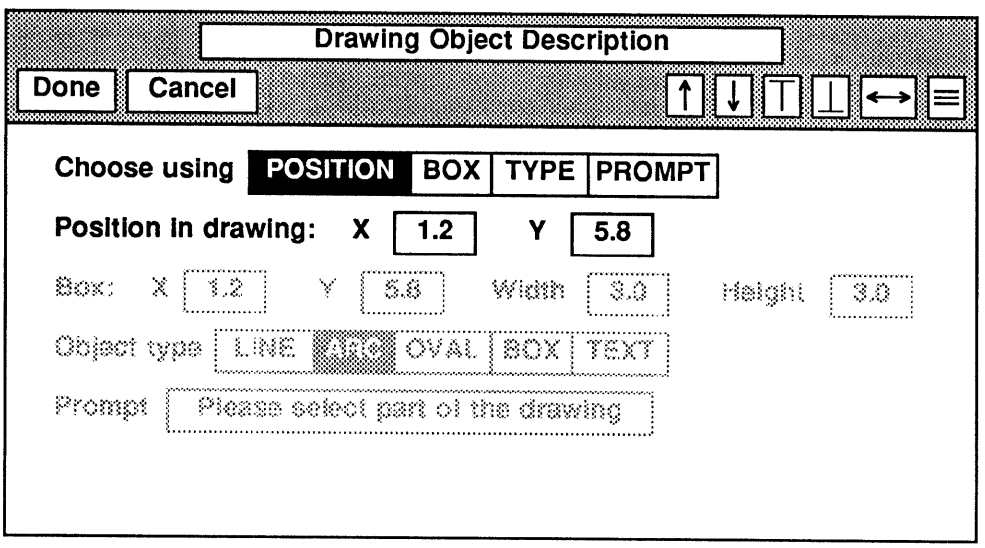


Figure 4.17. A Delta data description

The mechanism developed by Halbert allows a user to add control structures, including conditional tests, to a recorded macro. Incorporated in Whisper, these tests would be able to examine values returned by any function in the command language interface. In most

editing applications, however, almost all interface functions that return information are inquiries about the object being edited. For example, most Delta functions with return values provide data about objects in the user's drawing. In these applications, data descriptions, which can be considered implicit conditionals, will be used much more frequently than explicit tests (as Halbert observed).

While recording semantic rather than lexical information is critical to obtaining compact, readable macros, it is not enough. *Elision*, the reduction of a sequence of commands to a single command, also leads to condensed macros. For example, if a series of insert-character commands of a text editor are replaced by a single insert-string, the size of a macro can be greatly reduced. Eliding of commands is even more important for operations invoked by the mouse. For example, Delta provides a move command similar to the one provided by Apple's MacDraw. The user can move the cursor to an object, push down a mouse button, and drag the object to a new location by moving the mouse with the button down. Delta implements this operation with three separate functions corresponding to the down-press, drag, and release of the mouse button. The user, however, considers the operation to be a single atomic command. A sequence of commands should be elided when the user considers them to be a single operation. In this example, the interface specification would describe the mapping of the three operations into a single command. The application must provide an implementation of the merged command.

Object selection

In the SmallStar system, the act of selection is never explicitly recorded in a macro. Instead, each object that uses a selection explicitly references it, and a data description is generated for it. If the same object or objects are referenced several times, a single description is used. While Whisper could adopt this approach, it is better to treat selection as an ordinary operation and record the operation in the macro.

There are two differences in Whisper that make it more appropriate to treat selection as an ordinary operation. The first is that Andrew applications have no concept of a standard global selection. Each application is free to implement the concept of selection or not. Whisper cannot make assumptions about the properties of a selection. The second difference is that Andrew applications support a wider range of selection techniques than do SmallStar applications. The drawing editor has several selection methods and more could be added. The user can create a selection using a sequence of commands. While this command sequence could be specified by a data description, it would be easier to modify if Whisper treated it like other sequences of commands and not as data.

Selection is actually just one example of the translation of mouse coordinates into information meaningful to the application, and, more generally, an example of turning lexical information into semantic information. Rarely should a macro record window coor-

dinates associated with mouse input since this data will usually not be valid between invocations of the macro and is not meaningful to the user who wants to edit the macro. Each interface operator that uses mouse input should provide a function (specified in the interface) that translates coordinates into the arguments it needs. Table 4.1 shows some example translations. This information, like other data in the macro, would be represented by a data description. While the data actually recorded still may not be what the user wants, it is more easily modified in this form. For example, the user will rarely want a macro that applies some function to the 322nd character in a document. By providing a data description that refers to a character rather than a window position, Whisper makes it possible for the user to generalize the description into something more reasonable, such as current character or first character of the current paragraph.

Table 4.1. Example translations of mouse coordinates

Program	Translation
Drawing editor	World coordinates of the drawing (units might be inches, for example).
Text editor	Index to a character in the document.
Scrollbar	Vertical distance in units appropriate to object being scrolled (lines of text or distance in drawing units).
Mail reader	Index to a mail message.
Bitmap editor	Coordinate of the bitmap (an identity transformation).

In order to provide this facility, application functions that use mouse coordinates must obtain this data in a standard way, and the function that uses the information must be separated from the one that converts the coordinates into a more appropriate form. This separation is much the same as what must be done for commands that require other kinds of data, such as text strings and numbers. Interactive commands often prompt the user for a string. Two examples are Save As in MacWrite and write-named-file in Emacs. The corresponding interface function should take a string as an input argument. To avoid forcing the user to write an interface function that simply prompts for a string and calls the real application function, a standard function can be provided that prompts for a string when a function is called interactively but uses the supplied argument when the function is called from a program. This technique is similar to that used by Gnu and UNIX Emacs. In addition to a get-string function, the system can provide more specialized functions such as get-file-name and get-function-name. While some standard functions would be provided, the application could include its own. These functions can be viewed as type-checking functions.

Implementation

The programming-by-example interface would be provided by extensions to the Whisper library. To record a macro, a user chooses Record from an application menu. This action invokes a Whisper library routine that notifies the interpreter that a recording should begin. Figure 4.18 shows the sequence of events that occurs in response to user input once a user begins recording a macro. The window manager sends a stream of input events to Delta to be processed by the BX input handler. The handler maps events to application functions as usual, but while recording, sends both the input event and application function to a Whisper library routine as well. The Whisper routine maps this information into an interface function and data descriptions which it sends to the Whisper interpreter. Normally, this mapping is one-to-one, but it will sometimes use elision to compress multiple input events into a single recorded function. The application's Whisper interface specification defines the mapping between interactive functions and command language functions. The interpreter records the information passed to it by the application for future playback.

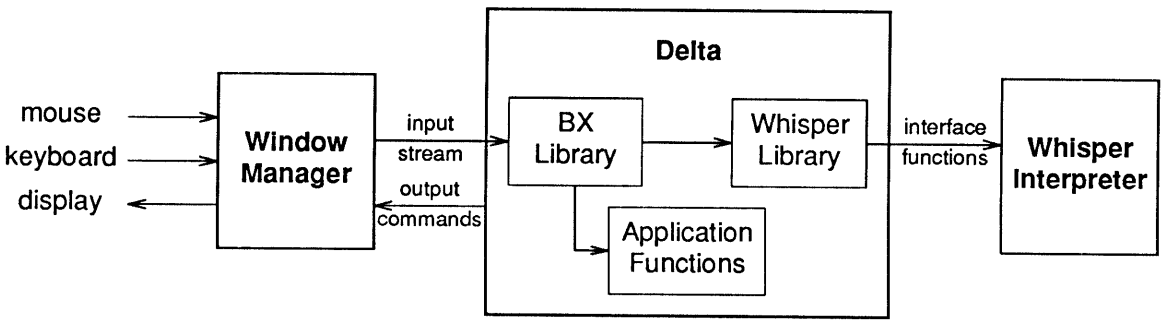


Figure 4.18. Implementation of macro recording

A similar sequence of events, shown in Figure 4.19, occurs when a user plays back a macro. The application provides a list of macros to choose from. Once a user selects a macro, a Whisper library routine asks the interpreter to execute it. The interpreter executes the macro as if it were an ordinary XLisp function, sending interface functions to the application for evaluation. Function side effects result in changes to application data which in turn cause the application to generate window manager output commands that update the display.

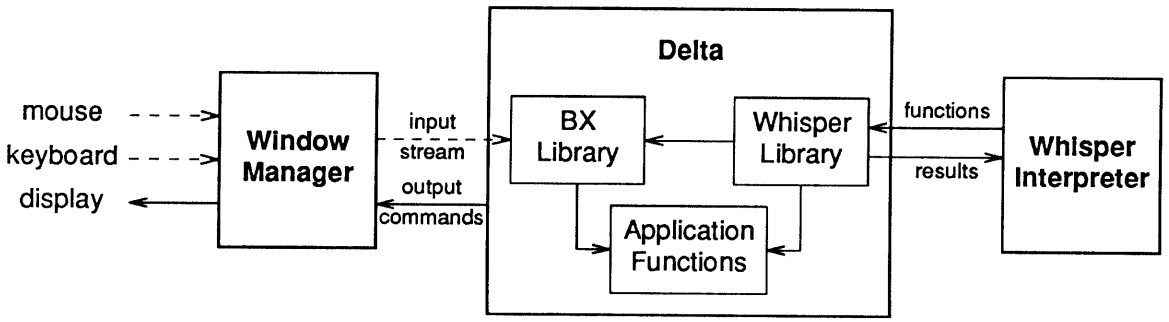


Figure 4.19. Implementation of macro replay

4.7 FURTHER EXTENSIONS

While a programming-by-example user interface is the most important missing component of Whisper, other extensions are important for a production system. These additions include a command language interface to the window manager, separate name spaces for applications, better handling of synchronization issues, and more general argument types for application interface functions.

A user of a windowing system interacts with the window manager's user interface more than any other user interface. A production version of Whisper should include an interface specification for the window manager. In Andrew, the specification would include functions that create, modify, and destroy windows and save a snapshot of the screen. Andrew's window interface is minimalist: it provides nine functions on a menu and three available using the mouse, and most of these twelve functions only serve to move windows around on the screen. On other systems, the window manager is combined with other functionality, making its application interface more important. For example, the Macintosh Finder combines window and file management.

The current version of Whisper installs all names in a single name space. External names can be shared by applications because names are not associated with applications. A function name is handed to the current application for evaluation. However, it is not possible for a name to represent a variable in one application and a function in another. Moreover, name conflicts can occur within an application since interfaces may be loaded by library files as well as by user modules. Using a more object-oriented approach, in which name bindings are determined by the object on which the named function operates would simplify command language programming.

Although Whisper permits users to interact with applications at the same time command language processing is carried out, it provides no synchronization of this activity. The current system relies on friendly users to avoid disaster, but a real system should protect the application.

Whisper provides a basic set of argument types for application functions. The more types Whisper provides, the more likely that the interface can access existing functions, making it easier for developers to incorporate Whisper into their application. Whisper should provide array and structured types as well as variants of the standard types (such as short integers).

4.8 SUMMARY

This chapter has described the design and implementation of Whisper, a prototype command language for window systems. Unlike most command languages, Whisper operates at a semantic level, overcoming many of the problems found in other approaches, as described in Chapter 3. Application developers define the inner language of Whisper by specifying programmable interfaces that describe application objects and operations. Whisper cannot guarantee completeness even at the semantic level; it is determined by the application interfaces.

Users may treat Whisper as the extension language for a particular application or as a general-purpose command language that can communicate with multiple applications. Given the ease of use of input recorders, it is worth providing a macro interface to Whisper even if the interface hides some of the command language's power. This chapter has described how a programming-by-example system, similar to SmallStar could be incorporated into Whisper.

5 Case Studies

Meaningful testing of a command language system is a difficult task. For any particular task, a command language is useful only if it is faster to write and execute a macro than it is to perform the task by hand or to write and execute a program in a programming language. To provide statistically meaningful results, experiments would require a group of subjects to perform several tasks using three methods: execution by hand, a Whisper program, and a C program.

I did not have the resources needed to carry out this kind of extensive testing. Because the applications for which I planned to provide interfaces were undergoing rapid change while I was building Whisper, I used a more stable but abandoned version of the system. It became impractical to recruit even a small number of users, since I would be asking them to use not just Whisper, but unsupported applications as well. As a partial substitute for user trials, I provided interfaces for more applications and wrote more command language programs than I had originally intended. This chapter describes my experience writing interfaces and macros for several applications.

5.1 THE TESTBED SYSTEM

As described in the previous chapter, I implemented Whisper in the Andrew environment. In this environment, almost all interactive applications do not interact directly with the display and mouse or even the window manager, but with the Base Editor, a user interface toolkit. While Whisper does not depend on the toolkit, all applications for which I created interfaces use an experimental version of the Base Editor (called BX), and some features were added to Whisper to make it more convenient to write interfaces for BX applications. While restricting the tested applications to BX programs limited the range of user interface styles Whisper had to cope with, the use of BX enabled me to learn about the interactions of a command language system with a user interface toolkit, experience that is applicable to other toolkits.

The Andrew window manager [Gosling 86a] is an autonomous process that mediates client processes' access to input/output devices: the keyboard, mouse, and display. Its process structure, shown in Figure 5.1, is similar to that of other network-based window managers such as X and NeWS [Scheifler 86, Gosling 86b, NeWS 86]. The window manager provides clients with a stream of input that includes both mouse events and keystrokes. An RPC interface specifies output commands that a client can use to draw graphics and text in a window on the screen. The window manager also notifies the client whenever the user reshapes or destroys the client window.

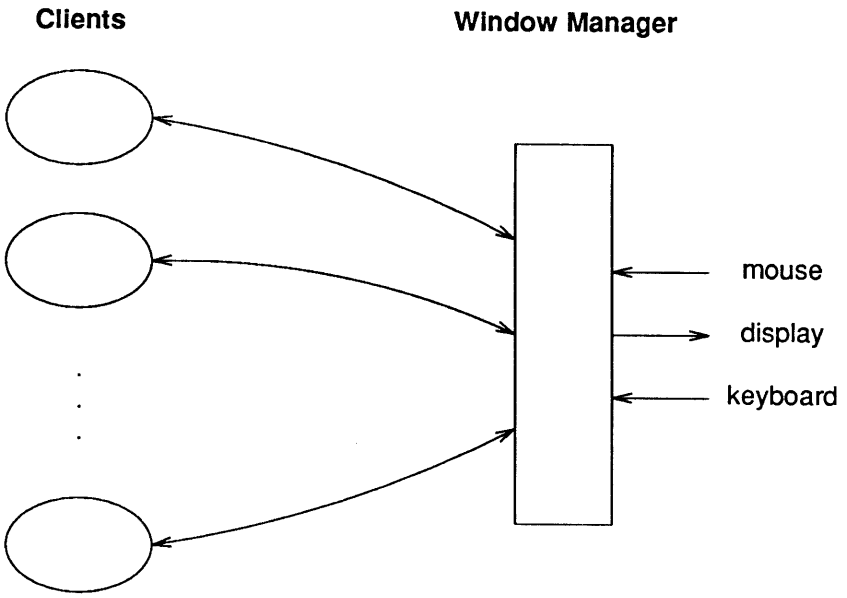


Figure 5.1. The window manager process structure

The Base Editor toolkit is a library of routines included in client applications. It provides a higher-level interface to programs, handling many of the details of communication with the window manager. It also makes available some powerful data types including *insets*,[1] *documents*, *scrollbars*, and *buttons*. An inset displays a *data object* in a rectangular region of a window. A document, one example of a data object, is used to manipulate text, and can range in size from a short string to an entire PhD thesis. A scrollbar, displayed adjacent to an inset, provides the user with the ability to view different parts of the inset's data object.

BX is an object-oriented system with only two classes of objects: insets and data

[1]In the current version of the Base Editor, called the Andrew toolkit or BE2, a "view" is the equivalent of a BX inset [Palay 88].

objects.[2] A data object is a permanent entity such as a document, drawing, or table data, while an inset provides a view of a data object. Insets often contain other insets and can be organized in a tree structure. In fact, the principal motivation behind BX's insets is to support editing of structured objects, such as documents that contain figures and equations. A BX application usually consists of a small amount of code that creates a window for use by an inset, while the inset itself implements most of the application functionality. Control is maintained by BX. The main program, insets, and data objects provide routines that are invoked by the BX main loop.

BX requires insets and data objects to provide a standard set of methods that the toolkit uses to manage user input, screen update, and reading and writing of data. The most important standard methods are described in Tables 5.1 and 5.2.

Table 5.1. Standard Base Editor inset methods

Method	Description
New	Create and initialize an instance of the inset.
Destroy	Destroy an instance of the inset, releasing its memory.
FullUpdate	Ask the inset to redraw itself.
Update	Ask the inset to selectively update itself based on changes since the last update.
KeyIn	Inform the inset that a key has been received.
Hit	Inform the inset that a mouse event has been received.
DataChanged	Inform the inset that its data object has changed.
Print	Ask the inset to print itself.

Table 5.2. Standard Base Editor data object methods

Method	Description
NewData	Create a new instance of the data object.
FreeData	Destroy an instance of a data object.
Read	Read a description of a data object from a file.
Write	Write a description of a data object to a file.
GetModified	Ask if the data object has been modified.
SetModified	Inform the data object that it has been modified.
InsetName	Ask for the name of an inset that can view the object.

[2]The Andrew toolkit uses a C preprocessor similar to C++ to provide a full object hierarchy.

The BX main loop processes user input by calling an inset's Hit or KeyIn routine. When no input is available, the main loop initiates screen update by calling the Update routine of the main inset. Commands called by the Hit and KeyIn routines usually do not update the screen directly but save information that to be used by the Update procedure, reducing the number of updates when the user provides more than one input event in a short time.[3] This program structure, in which control is external to the application, made it possible to easily incorporate Whisper into the toolkit. The delayed update mechanism ensured that most commands separated output from data structure changes, allowing most Whisper macros to be executed with only a single update.

5.2 A SIMPLE CALCULATOR

In the first test of Whisper I added a programmable interface to a simple calculator. This calculator, briefly discussed in Section 4.2, consists of three insets. A calc inset contains two children insets: an array of buttons and a label. Labels display a single line of text. The calculator uses a label to display its current the value.

The calculator interface is split into two parts. The main program provides a quit operator and the calc inset provides functions corresponding to the button operations. Tables 5.3 and 5.4 list the interface operations and variables, while Appendix C.1 contains the interface specification provided to Whisper.[4] BX library modules, included as part of the calculator, provide additional utility functions.

Even an interface as simple as the calculator's provides insight into the advantages and disadvantages of the Whisper approach. While it was easy to create a *complete* Whisper interface, the process demonstrated some of the problems that occur when an application that is designed to support a direct manipulation interface is extended to support a command language. Several issues arose that reappeared in other applications as well: choice of interface level, interface safety, exposure of internal parameters, and mapping of mouse input to semantic level functions.

The calculator is an application that can easily be provided with a provably complete interface. Each input function available to the user has an analogue in the command language interface. The only output of the program is the current display value. Since the interface provides the user access to this value, the interface is complete. While com-

---

[3]Application Hit routines will sometimes issue output commands directly in order to provide dynamic feedback. Section 5.3.2 discusses some implications of this implementation technique.

[4]By convention, operations and variables in Whisper interfaces are prefixed by the name of the module that implements them. Quit is an exception.

Table 5.3. Operations defined by the calculator interface

Operation	Description
quit	Exit the program.
calc-enter	(number) Enter number into the calculator.
calc-digit	(digit) → current-value Concatenate digit onto the calculator's current value and return the new value.
calc-point	Concatenate a decimal point onto the calculator's current value.
calc-function	(function) → current-value Apply the pending function and make function, one of +, -, *, or /, the new pending function and return the result.
calc-clear-entry	Set the current value to zero.
calc-clear	Reset the calculator.
calc-negate	→ current-value Negate the current value.
calc-equals	→ current-value Apply the pending function to the current value and return the result.

Table 5.4. Variables defined by the calculator interface

Variable	Description
calc-display	The current (displayed) value of the calculator.
calc-register	The value of the calculator's internal register. When calc-equals is invoked, the pending function is carried out on calc-register and calc-display.

pleteness seems trivial in this example, it demonstrates the difference in power between the Whisper approach and keyboard macros. In an ordinary keyboard macro system, macros would not be able to examine the calculator output. In a system that captured the output at the window manager level, the macro would have to distinguish between window manager operations that redrew the calculator itself and those that updated the display value. The value update in Andrew is done by drawing a string in the window. This command would have to be watched for and its argument converted to a number. While this kind of manipulation of the output stream is imaginable for the calculator, it is not feasible for more complex applications.

While completeness requires that functionality provided by the user interface be available through the command language, it does not dictate the interface level. An operation that is appropriate in an interactive interface may be too low-level in a command language interface. Entering numbers in the calculator provides one example. Users operate the calculator by clicking on buttons or by typing keys that correspond to the buttons. To enter the number -4.2, the user can type the four corresponding keys. However, a

program in which four functions must be called to enter this number would be clumsy to write and difficult to read. To make command language programming easier, an enter function was added that took a number as an argument. In the programming-by-example system, elision could be used to automatically simplify calculator macros, converting a series of steps into a single enter operation.

The simplest implementation of the calc-digit operator illustrates the interface safety problem. This function is used to enter digits into the calculator. The corresponding C function, KeyDigit, accepts an input parameter of type character. KeyDigit performs no checks to ensure that the character is a digit since all invocations of KeyDigit guarantee that the argument is a digit. However, once this function becomes public, some kind of check must be done. One solution is to implement calc-digit by a public function that checks its argument before calling the private KeyDigit. Another is to add the check to KeyDigit itself. A third solution is to extend the interface language to support input arguments with restricted types. The interface generator would automatically construct the code needed to enforce the restrictions, guaranteeing, as in the first option, that KeyDigit would only be called with legitimate input. In general the third approach is best since it imposes no unnecessary overhead on the user interface and minimizes the changes that must be made to the original program. This approach, however, is limited to cases where the restrictions do not change during the application's execution.

The Base Editor requires operations bound to keys and standard inset methods to accept the inset instance as their first argument. However, the command language programmer gains nothing from this flexibility if an application contains only one instance of an inset. The calculator application contains a single calc inset. To hide this additional complexity from the command language programmer, Whisper provides automatic parameters that are supplied to implementation routines by the interface but are never seen by the command language programmer.

Section 4.6.2 explained that mouse as well as keyboard input should be converted from the lexical to the semantic level. In the calculator, the mouse can be used to invoke an operation that is bound to one of the visible buttons. This mapping, handled by the button inset, is transparent to the calc inset. In the programming-by-example version of Whisper, the button inset would report the results of its mapping to Whisper. Button operations would then be recorded in macros when the user either clicked a calculator button or typed the corresponding key.

5.3 A DRAWING EDITOR

Delta is an object-based drawing editor with an interface similar to MacDraw's (Figure 5.2). The user may draw lines, curves, or text by first clicking on a button in the *palette*, an array of buttons on the left-hand side of the window, and then clicking at two points that define the object's location. After choosing the top-most button (the up arrow), the user can select objects by clicking on them and can move the selected objects by dragging them to a new location with the mouse button held down. Menu commands enable the user to copy, delete, and scale objects. The horizontal and vertical scrollbars allow the user to view different parts of a drawing. An area at the bottom of the window displays messages and prompts the user for text input.

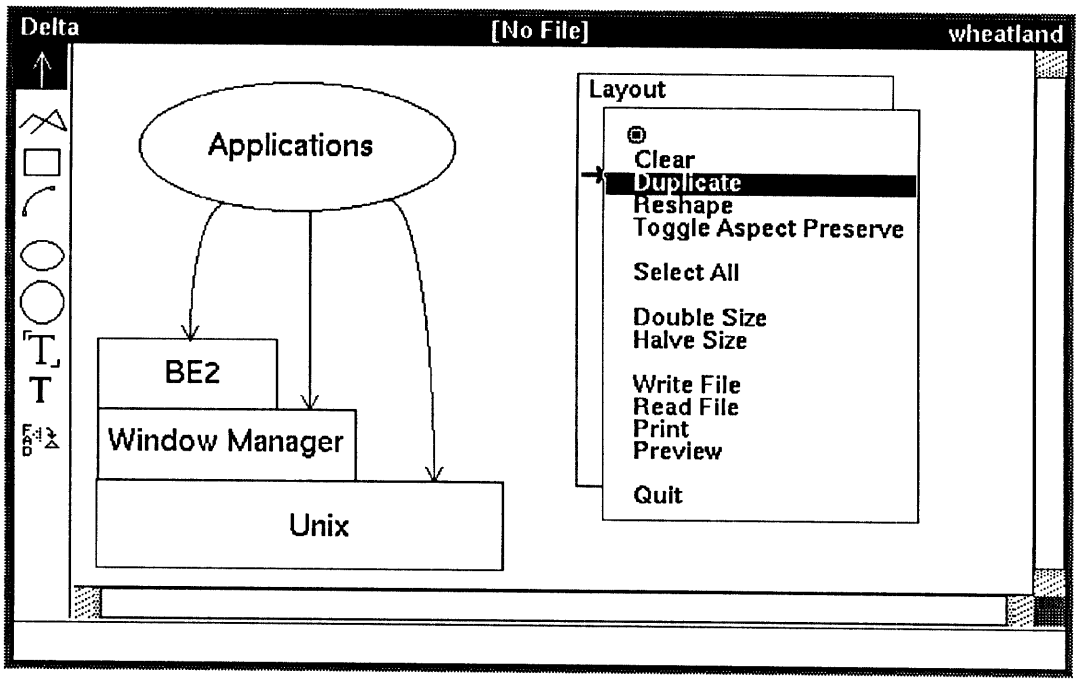


Figure 5.2. The Delta display

Delta, like the calculator, consists of a main program that creates and installs an inset in a window. Almost all of the application code is actually part of the inset. This inset can be inserted into a document and will provide the same functionality there as it does when editing a stand-alone drawing. The Delta inset de contains children insets that provide the scrollbars, message line, palette, and drawing surface, or *canvas*.

Delta's Whisper interface is split into three modules: the main program contains the quit operator, the decmds module contains operators that turn on and off the display of children insets (the palette and scrollbars), and the ccmds (canvas commands) module contains drawing operators. Table 5.5 summarizes and Appendix C.2 contains the full Whisper interface.

Table 5.5. Operations defined by the Delta interface

Operation	Description
quit	Exit the program.
decmds-toggle-palette	Toggle the palette on or off.
decmds-toggle-scrollbar	Toggle the scrollbars on or off.
decmds-get-state	→ (palette-state scrollbar-state) Return the status of the palette and scrollbars (t or nil).
ccmds-set-tool	(tool) Change the current "tool," one of the buttons in the palette. The interface defines names for the tools.
ccmds-create-object	(x2 y2 x1 y1) → created-object Create an object of the current type (defined by the current tool) using the two points as if they were generated using the first and second clicks of the mouse.
ccmds-select-by-location	(x y) → object Return the object at the location or nil.
ccmds-extend-by-location	(x y) → object Extend the selection to the location. Return the first object in the selection or nil.
ccmds-move-selection	(x y) Move the selection by (x,y).
ccmds-move-handle	(handle x y) Move the handle of the selection by (x,y). The interface defines names for the nine handles (shown in Figure 5.5).
ccmds-first-object	(object-type) → object Select all objects that match the string object-type (or all objects if the string is ""). Return the first object or nil.
ccmds-next-object	→ object Cycle through a list of selections created by ccmds-first-object. Return the next object or nil if there are no more.
ccmds-select-object	(object) Add object to the selection.
ccmds-deselect-object	(object) Remove object from the selection.
ccmds-get-n-vertices	(object) → n-vertices Return the number of vertices the object has.
ccmds-get-vertex	(vertex object) → (x y) Return the location of the object's numbered vertex.
ccmds-get-width	(object) → width Return the width (thickness) of the object.
ccmds-set-width	(object width) Set the width (thickness) of the object.

Table 5.5. Operations defined by the Delta interface (cont.)

Operation	Description
ccmds-get-visible-box	→ (xmin ymin xmax ymax) Return the extent of the drawing currently visible.
ccmds-show-point	(x y) Scroll the drawing so that (x,y) is in the center of the visible box.
ccmds-set-zoom	(zoom-factor) Set the current zoom factor.
ccmds-get-zoom	→ zoom-factor Get the current zoom factor.
ccmds-clear	Delete the selection.
ccmds-duplicate	(x y) Duplicate the selection, offset (x,y) from the original.
ccmds-reshape	Toggle reshape mode.
ccmds-select-all	Select all objects in the drawing.
ccmds-read-drawing	Prompt for a file-name and read a drawing from that file.
ccmds-read-named-drawing	(file-name) Read a drawing from file-name.
ccmds-write-drawing	Prompt for a file-name and write the drawing to that file.
ccmds-write-named-drawing	(file-name) Write the drawing to file-name.

Providing a programming interface to Delta was the most difficult test of Whisper. As discussed in Section 2.4, the class of programs that includes Delta is among the least abstract in a hierarchy of applications. Not only does a drawing editor such as Delta use a graphical direct manipulation interface, but the underlying objects manipulated by the program are graphical as well, and more difficult to describe in a command language. Many text-based applications have programmable interfaces. Text-based applications with direct manipulation interfaces pose additional problems, but despite surface changes the applications still can be described by a few concepts and operations that are expressible in a written language. For example, there are only minor differences between the Whisper interface to the calculator described in the previous section and one that is completely text-driven. However, the semantic information contained in a drawing editor is less easily expressed in textual language. The information content of the Delta display is much greater than the calculator display's.

5.3.1 Access to data

The most important decision to be made in designing the Delta interface was determining how to provide access to drawing state, permitting macros to learn about a drawing's contents. In Delta, as in most direct manipulation interfaces, the interface designer must provide explicit operations that correspond to the implicit commands a user executes while examining the display. (See Section 2.4.2.) I chose to provide a generator function, next-object, that cycles through the current selection, returning a reference to another selected object each time it is called. Additional functions, such as get-n-vertices, return information about an object's attributes. Another interface function allows the command language programmer to deselect a particular object. These functions allow the programmer to select all the objects in the drawing or in a region, examine each object, and apply editing operations to "interesting" objects. For example, the following program selects all double-width lines in the drawing:

```
(defun select-double-lines ()
  (let (object)
    (ccmds-select-all)                          ; select all objects in the drawing
    (setq object (ccmds-first-object "line"))   ; get the first line
    (do () ((null object))                      ; quit when no more lines
      (if (/= (ccmds-get-width object) 2)       ; if line width NOT 2
        (ccmds-deselect-object object))         ; deselect it
      (setq object (ccmds-next-object "line")))))   ; get the next line
```

While this interface is suitable for programming in the command language directly, in a programming-by-example system the use of data descriptions would simplify the selection process. To create the same macro using the programming-by-example interface, a user would record a macro in which he selects a line. He would then edit the macro, specifying line width 2 in the data description. Delta data descriptions would have two levels. The first describes the group of objects as a whole (Figure 5.3), and the second is available to specify details about the particular type of object (Figure 5.4). Here, the first level specifies that the macro operates on lines, and the second level specifies double-width lines.

5.3.2 Interface to mouse operations

The second major step involved in designing the Delta interface was deciding how to package the operations provided by the mouse. Depending on the program's mode, the mouse may be used to create, select, move, or scale objects or move selected vertices. Moreover, these operations are actually broken up into separate phases associated with pressing down, dragging, and releasing the mouse button.

The interface could have provided a single function corresponding to all these operations that bases its actions on current state—this is how Delta's user interface is implemented.

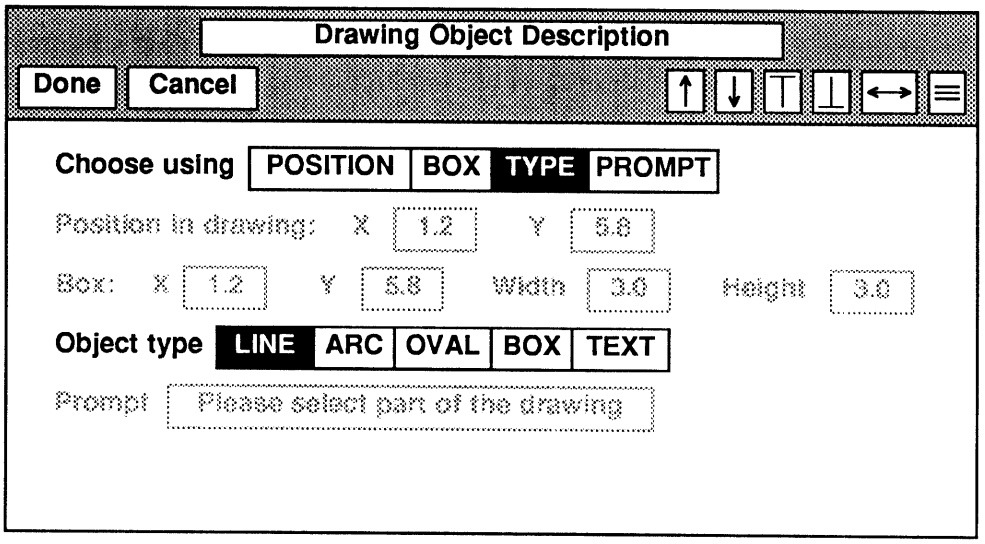


Figure 5.3. Top-level data description for double-width lines

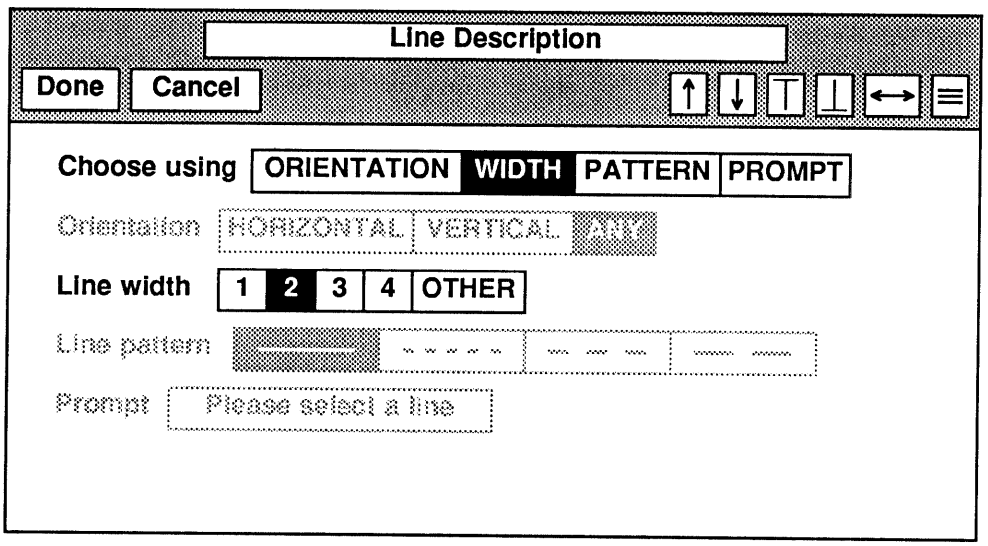


Figure 5.4. Second-level data description for double-width lines

The canvas_Hit function handles all mouse events inside a drawing. It chooses a specific action to carry out depending on:

- *Active tool.* The user chooses a *tool* from the palette to the left of the drawing. (See Figure 5.2.) The first tool (the up-arrow) represents selection. If it is chosen, mouse clicks invoke select, move, and edit operations. The other tools correspond to objects that the user may create. When one of these is selected, mouse clicks create objects.
- *Cursor location.* In select mode, if the cursor is over an object when the user presses a button and drags the mouse, the object will move to the location of the cursor when the button is released. If the cursor is not over an object, the initial and final locations determine a box, and Delta selects all objects inside the box.

- *Drawing state.* As in MacDraw, when the user selects an object, handles appear that can be dragged to change the object's shape. If the user presses a button over the handle of a selected object, that handle will move. If the object was not selected before the operation, the whole object will move.
- *Mouse button.* Andrew workstations have mice with two or three buttons. In select mode, the left button begins a new selection, while the right button extends an existing one.
- *Button action.* The operation Delta carries out depends on whether the user has depressed the button, is dragging the mouse with the button held down, or has released the button.

The advantage of providing a single command language function is that the mapping from user action to command language program is trivial. However, this approach is really lexical data recording in disguise and suffers from the same drawbacks. If a user edited a macro that recorded this level of information, he would have to figure out the action corresponding to the function by determining the state of Delta. This state is complex and not easily extracted from the command language program. In fact, the behavior of the program will change depending on Delta's state when the macro is executed.

An alternative approach is to provide the semantic action associated with the mouse function. Even here, there are two choices. High- or low-level semantic operations can be provided. Figure 5.5 shows the effect of dragging the northeast handle of an ellipse up and to the right. By moving a handle, the user scales the object. However, he need not understand that scaling is the underlying operation. The Whisper interface provides a function called move-handle, rather than a scale-selection function. The principle followed is that the command language interface should be at the *semantic* level yet *visible* to the user. The overriding goal is to enable the user to write easily understandable macros. While general programming interfaces are often designed to be as high-level and flexible as possible, the constraints of command language programming force Whisper interfaces to be lower-level and more baroque than the equivalent general programming interface. A command language interface represents a compromise between the user interface and a pure programmable interface.

5.3.3 Other interface decisions

Even when the function to provide is clear, its parameters may not be. Direct manipulation interfaces such as Delta's are frequently modeless. Delta's duplicate command places a copy of the selection slightly to the right and below the original. While this location is rarely the one desired by the user, it enables the copy operation to occur without prompting the user for a destination. In command language programs, however, it is more convenient to specify a final destination rather than move the duplicated objects from the temporary destination. The Delta command language interface deals with this

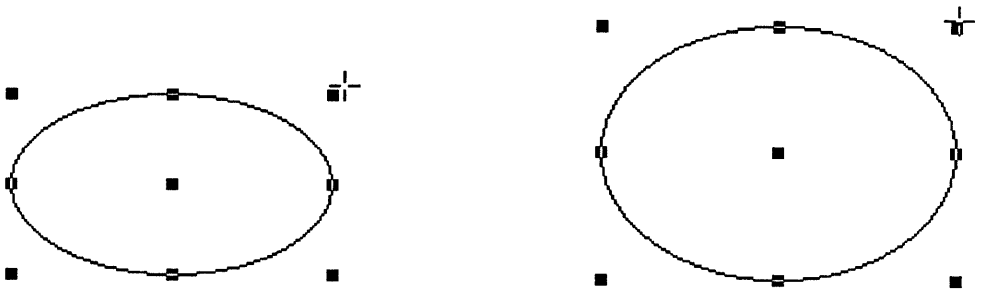


Figure 5.5. Editing a Delta object

problem by defining default parameters for many operations. As in the previous example, the designer must compromise between the user interface and the ideal programming interface.

While modeless operations use implicit operands that are made explicit in the programming interface, operations that prompt for arguments must also be modified. Non-interactive versions of operations such as read-drawing and write-drawing must be provided. If the implementation of read-drawing consists of two functions, a non-interactive one that takes a file name as an input parameter and returns a status code, and a wrapper function that prompts for the name and prints a message based on the return code, then it is easy to provide the necessary interface functions. Note that both the interactive and non-interactive functions must be provided since the macro writer may want to use either function. While this solution is sufficient, a simpler and more efficient solution is to provide a way for functions to prompt for arguments that are not provided, as described in Section 4.6. In the macro that records a read-drawing operation, for example, a data description specifies the name provided by the user. However, this description may be modified to specify a different name or to interactively request a new name. Since I didn't implement the argument-handling routines, the Delta interface simply provides both versions of each operator that prompts for parameters.

The effect of the user interface on the command language interface can be subtle. For example, Delta defines any object that the user can create by two points, determined by the location of the cursor when the mouse button is depressed and released. Objects whose geometry requires more than two points to define are fixed in some dimension, forcing the user to adjust the object using editing commands after creating it. Arcs are initially quarter-arcs, for instance. While it would have been convenient to parameterize these fixed values in the interface, this would have been impossible without major changes to Delta. Here the user interface, although separate from the application, strongly influenced it, limiting the functionality that could be provided by the command language interface.

In fact, even though one of Delta's design goals was to separate the user interface from the core application, interactions between the two often caused problems in the command language interface. Several examples have already shown how the choice of user input functions affected the command language interface. Implementation of output also affected the interface. Ideally, core application functions should not produce output directly but only make changes to underlying data that are noticed by the user interface, which can update the display at a convenient time [Szekely 87]. However, Delta sometimes violates this strategy. In part, the problem is efficiency. In order to support fast dragging of objects, Delta uses a fast but inaccurate redisplay algorithm while objects are being moved. Once the operation is completed, the screen is updated more slowly but correctly. To be fast, the quick redisplay code is not separated from the object moving code. This works well when the functions are invoked interactively, but not when they are called by a macro. The correctness of the macro is unaffected, but execution is slower.

Recently, user interface researchers have concluded that this problem is inherent in direct manipulation interfaces [Dance 87, Hill 87b, Hudson 87]. Object dragging in Delta is just one example of the semantic feedback typical of this style of user interface. Semantic feedback tends to bind the user interface to the core functionality since knowledge of application-specific data is needed to update the display.

5.3.4 Completeness

The Delta command language interface is complete, but the proof of its completeness is not obvious.

As is often the case, completeness for input operations is clear. For each menu operation the Whisper interface provides a corresponding function. As Section 5.3.2 described, interface functions provide the same editing capability that is available interactively using the mouse. The choose-tool operator corresponds to selecting a palette tool. The scrollbars allow the user to adjust the view of the drawing horizontally or vertically by a fixed amount. The interface function show-point does the same, although the mapping is not simple. The semantic distance between the interactive scrolling operations and the interface functions is a weakness of the interface.

Delta provides two classes of output. The first, discussed in Section 5.3.1, describes the underlying data object, the drawing. As explained in that section, interface operations allow command language programs to examine each object in a drawing. The second class of output represents the presentation of the data—the inset itself. This output consists of viewing information: the extent of the drawing currently displayed and the zoom factor. The Whisper interface also provides access to this data.

5.3.5 Use of the Delta interface

A programmable interface to Delta or any drawing editor can add considerable functionality. Programmability here is useful in the same way that a macro language is useful in a text editor. Repetitive tasks, such as changing the width of all lines in a drawing, are more easily done by a macro than by hand. Creation of structured drawings may also be more easily done in a program. The drawing in Figure 5.6, for example, was generated by the following program:

```
; draw n nested ellipses with initial center (xc,yc) and initial radius (xr,yr)
(defun do-ellipses (xc yc xr yr n)
  (ccmds-set-tool ellipse-tool)                    ; create-object will make ellipses
  (dotimes (i n)                                   ; repeat n times
    (ccmds-create-object (+ xc xr) (+ yc yr) xc yc)  ; create an ellipse
    (setq xr (* xr 0.75))                          ; shrink radii by 75%
    (setq yr (* yr 0.75)))
  (ccmds-deselect-object))

; draw n sets of nested ellipses with initial center (xc,yc) and initial radius r
(defun many-ellipses (xc yc r n nested)
  (dotimes (i n)                                   ; repeat n times
    (let ((xr r) (yr r))                           ; define x- and y-radius initialized to r
      (do-ellipses xc yc xr yr nested)             ; draw one set
      (setq xc (+ xc 100.0))                       ; move center to the right
      (setq xr (* xr 0.75)))))                     ; shrink x-radius

(many-ellipses 100.0 200.0 50.0 5 4)
```

The drawing contains five sets of concentric ellipses, with each ellipse's axes 75% of the size of the enclosing ellipse's. Note that this example could not be easily generated by a programming-by-example system.

Command language programming inside Delta could be used for more complex tasks as well, transforming the editor into a base for special-purpose editors, for example, supporting architectural and VLSI design.

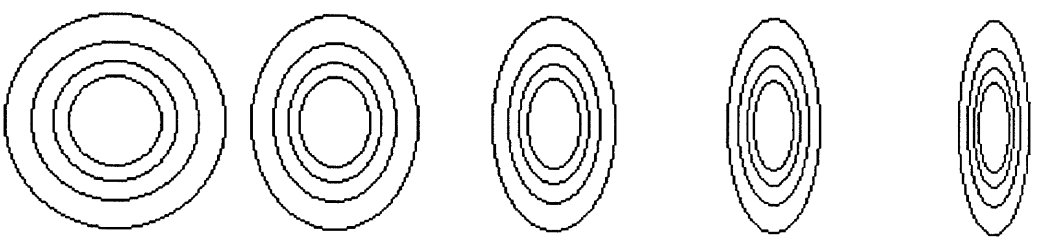


Figure 5.6. A macro-generated drawing

5.4 A DOCUMENT EDITOR

BrandX is a text editor that combines many of the features of Emacs and MacWrite. It provides selection and cut, copy, and paste commands similar to MacWrite's, as well as a full set of Emacs-like editing commands bound to keys. Like Emacs, BrandX supports the use of multiple *buffers*, each containing a view of a document. Like MacWrite, the editor allows the user to create a document with multiple fonts and typefaces. Blocks of text can be modified by *styles* that are defined by a set of attributes. Text is dynamically formatted according to the style information. BrandX also supports the insertion of insets such as drawings, equations, and tables into a document. Figure 5.7 shows an example of a BrandX document that includes a bitmap and drawings generated by Zip, another BX application.

BrandX is similar in structure to Delta and, in fact, served as a model for Delta. The text view inset provides most of the functionality of the editor. This inset is grouped with a scrollbar and a message line, which itself is a refinement of the text view. BrandX's insets, unlike Delta's, are part of BX and are available for use by all BX applications. The BX library contains all but one of the modules that make up BrandX. Partly for this reason, the BrandX Whisper interface is huge compared to Delta's, with about 180 operators defined. Appendix C.3 contains the complete interface. The interface is broken up according to the internal structure of BrandX—each interface module corresponds to a single C file. Whenever a BX application makes use of a library module, it automatically imports the corresponding Whisper functions. For example, if an application uses the standard BX message line, its Whisper interface will include message line functions.

5.4.1 Text editing

Programmable text editors have been in use for many years, the most popular of which have been the various versions of Emacs. Design of programming interfaces for text editors has gradually improved as developers learned from experience with earlier editors. Rather than design from scratch, I used the UNIX Emacs interface as the basis of BrandX's Whisper interface. This section discusses issues concerning the basic text editing interface. The next section examines a less explored area of editor interfaces—operations for editing and manipulating insets, objects inserted in documents.

Some of the problems that arose in the Delta interface design reappeared in the BrandX design. As in Delta, many interactive BrandX commands take no arguments since there is no convenient way to supply them. BrandX, like Emacs, does support a provide-argument operator that may be used with any command. BrandX operators behave as if they were supplied with an argument whose value is one if not specified by

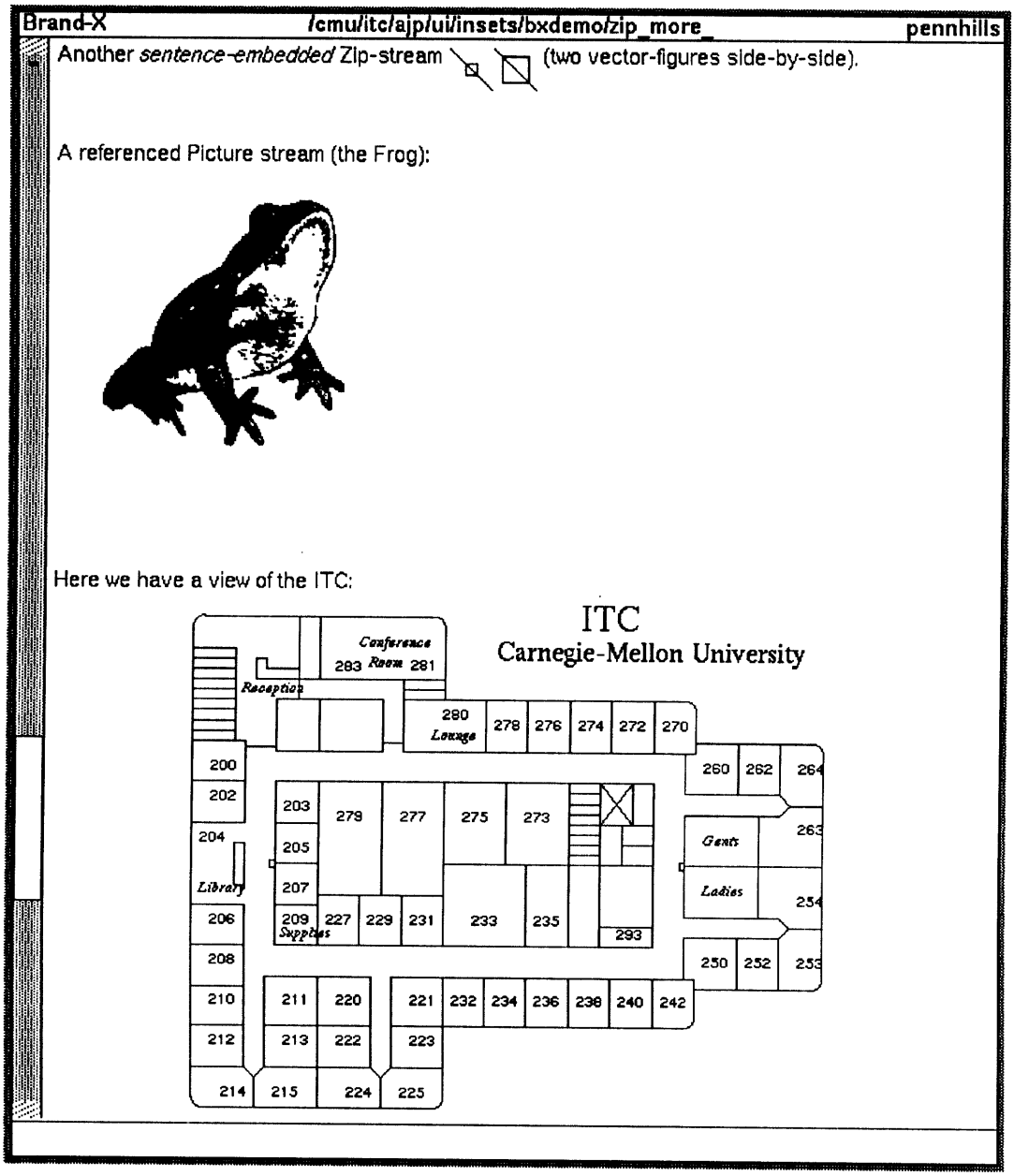


Figure 5.7. The BrandX display

provide-argument. The Whisper interface should work this way as well. However, provide-argument does not literally provide an argument to a subsequent function but, instead, sets a variable that the function can examine. This underlying implementation must be changed if the cleaner solution is to be provided by the interface.

Several commands in BrandX, as in Delta, prompt the user for a string argument. The Whisper interface for BrandX adopts the Delta solution. Each function is available in two forms: interactive and non-interactive. In a production version of Whisper, the necessary changes should be made to BX, BrandX, and Delta to support the layered ap-

proach described in Section 5.3. However, when a command prompts for arguments in a non-standard way, two forms of a command must be provided. The BrandX query-replace command, for example, first prompts for a search string and its replacement. It then displays each match of the search string, allowing the user to selectively replace just a few matches. Here, the interactive and non-interactive versions of the command cannot easily be combined.

A new problem that surfaced in the BrandX interface design concerns storage management. Internally, the editor uses *marks* to remember positions in a document. Even after a series of insertions and deletions, a mark points to the "same" location as long as the text at that location was not deleted. The Whisper interface exports operators that create marks and bind them to text positions. Since the create operation dynamically allocates marks, they should be deallocated when no longer needed. In the current Whisper implementation, the command language programmer is responsible for calling a free-mark operator. The command language garbage collector should handle this automatically. This problem does not appear in a system like Tinylisp where the command language and application exist in the same address space and use the same garbage collector.

5.4.2 Inset editing

BrandX is distinguished from other text editors by its ability (provided by the Base Editor) to edit arbitrary types of objects inside documents. When the user inserts an object into a document, the editor dynamically loads the code needed to view and modify it. This code is no different from the code that defines the inset managing the object in a stand-alone program. The drawing editor has the same user interface whether it is run stand-alone as Delta or within BrandX. BrandX allows the user to create, move, and destroy insets and to invoke inset functions. Once the user clicks the mouse inside the area of the document that "belongs" to the inset, the *input focus* switches from the enclosing document to the inset. All subsequent menu and keystroke commands apply to the inset. Using the appropriate inset methods, BrandX can read, write, and display documents that contain objects. While Whisper provides the ability to add new commands, the inset paradigm provides a different kind of extensibility—the ability to add new kinds of editable objects.

Supporting a programmable interface to arbitrary insets and to the editor's inset manipulation operations is an important test of Whisper. BrandX provides a uniform interface to a collection of user interfaces, one for each type of editable object, and the inset/data object organization encourages programmers to make these interfaces consistent. Whisper must extend this consistency to the command language interface.

BrandX's user interface to inset manipulation is less refined than its other functionality.

At the time I incorporated Whisper into the editor, BrandX's developers and users had had little experience with creating insets in documents. The user interface is missing some important operators, some of which I added to carry out my experiments.

Just as a command language consists of general language constructs (outer language) and application-specific functions (inner language), the BrandX interface contains general inset-manipulation functions and application-specific functions. The inset functions provide generic operations such as traversing the inset tree, changing the input focus, and determining an inset's type. These functions are part of the core BX library and have no specific knowledge of the text inset. The application-specific functions manipulate BrandX's refinement of the generic inset (vinset), for example allowing the user to search a document for an inset or create an inset at a particular character position. Delta, like BrandX, provides its own inset (dinset) manipulation functions. Any inset, not just the BrandX view and Delta canvas, may allow the user to include subinsets in the principal inset. Each must provide interface functions that support manipulation of these subinsets. Table 5.6 lists the inset operations supported by the Base Editor, BrandX, and Delta.

Table 5.6. Inset operations defined by applications

Operation		Description
Base Editor		
window-top-level	→ inset	Return the top-level inset in the window.
window-current-inset	→ inset	Return the inset that is the input focus.
window-goto-parent		Give the input focus to the current inset's parent.
im-inset-type	(inset) → inset-type	Return the type of the inset (a string).
im-goto-inset	(inset)	Give the inset the input focus.
BrandX		
vcmds-create-vinset	(inset-type) → vinset	Create a vinset of the specified type (a string).
vcmds-delete-vinset	(vinset)	Delete the vinset.
vcmds-first-vinset	(vinset-type) → vinset	Return the first vinset of the specified type.
vcmds-next-vinset	(vinset-type) → vinset	Return the next vinset of the specified type.
vcmds-inset-to-vinset	(inset) → inset	Convert an inset into a vinset.
vcmds-vinset-to-inset	(vinset) → inset	Convert a vinset into an inset.
Delta		
ccmds-first-dinset	(dinset-type) → dinset	Return the first dinset of the specified type.
ccmds-next-dinset	(dinset-type) → dinset	Return the next dinset of the specified type.
ccmds-inset-to-dinset	(inset) → dinset	Convert an inset into a dinset.
ccmds-dinset-to-inset	(dinset) → inset	Convert a dinset into an inset.

The Base Editor's framework for insets makes it possible to write macros that deal with multiple kinds of objects. For example, the following Whisper program, vcmds-spell-check-all, invokes a spelling checker on all text in a document, including that found in drawings.

```
(defun vcmds-spell-check-all ()
  (view-apply-global-command (vcmds-spell-check)))

; apply command to this document and text in sub-drawings
(defun view-apply-global-command (cmd)
  (let (vin)                                    ; declare a local variable
    (eval cmd)                                  ; do command to this document
    (setq vin (vcmds-first-vinset "de"))        ; get first drawing in document
    (do () ((null vin))                         ; quit when no more drawings
      (im-goto-inset (vcmds-vinset-to-inset vin))  ; give input focus to drawing
      (de-apply-text-command cmd)               ; apply command to text in drawing
      (window-goto-parent)                      ; give input focus to enclosing doc
      (setq vin (vcmds-next-vinset "de")))))    ; get next drawing in document

; apply command to all text in drawing
(defun de-apply-text-command (cmd)
  (let (din)                                    ; declare a local variable
    (ccmds-select-all)                          ; select all objects in drawing
    (setq din (ccmds-first-dinset "view"))      ; get first text in drawing
    (do () ((null din))                         ; quit when no more text
      (im-goto-inset (ccmds-dinset-to-inset din))  ; give input focus to text
      (eval (view-apply-global-command cmd))    ; apply command to drawings in text
      (window-goto-parent)                      ; give input focus to enclosing drawing
      (setq din (ccmds-next-dinset "view")))))  ; get next text in document
```

This program illustrates a weakness in the current implementation of Whisper and BX. All functions defined by Whisper interface specifications in BrandX exist in a single name space, just as all functions defined by the corresponding C code do. The functions defined by interfaces must be unique, even if they belong to different insets. To ensure uniqueness, I adopted the convention that interface function names are prefixed by the module that defines them. This forces the command language programmer to be aware of all the possible types of objects (insets) a macro might encounter and each object's corresponding operations. A pure object-oriented approach would make command language programming much simpler. The function invoked by a name would be determined by the type of object it was applied to. The functions used by vcmds-spell-check-all could be replaced by a single generic function, apply-global-command, that applies a command to itself and all its subinsets.

```
; apply command to this inset and its sub-insets
(defun apply-global-command (cmd)
  (let (in)                                  ; declare a local variable
    (eval cmd)                               ; do command to this inset
    (setq in (first-inset))                  ; get first inset in this inset
    (do () ((null in))                       ; quit when no more insets
      (goto-inset in)                        ; give input focus to child inset
      (apply-global-command cmd)             ; apply command to child inset
      (goto-parent)                          ; give input focus to enclosing inset
      (setq in (next-inset)))))              ; get next inset in this inset
```

5.4.3 Completeness

Because of its size and complexity, the completeness of the BrandX interface is difficult to evaluate. Since the interface closely follows the Emacs MLisp interface, most of its problems occur where the two applications' models of editing diverge.

As was true for the calculator and Delta, semantic completeness for input operations is easy to ensure for BrandX. Each operation provided interactively by keys or menus is included in the Whisper interface. BrandX supports just two mouse operations:

- *Selection.* Users can begin a selection at a character and extend the selection to another character. Interface functions support these operations.
- *Change of input focus.* By clicking the mouse in a sub-inset, the user gives that inset the input focus. In addition to the equivalent of this function, the Whisper interface provides functions that allow a command language program to traverse a document's inset tree.

BrandX output, like Delta's, can be classified as representing either the data object (the document) or its presentation. Like Emacs, the BrandX interface allows a command language program to obtain the contents of a portion of a document as a string. The BrandX interface also allows a program to determine the styles in effect at a position. Similarly, a program can determine the inset at a position or the next inset in a document. One way in which BrandX differs from Emacs is that in BrandX, line breaks are part of the presentation, not the data. BrandX adjusts the length of each line according to the window width, while Emacs breaks lines whenever it finds a newline character. Newline characters serve as paragraph breaks in BrandX. The Whisper interface provides functions that determine the beginnings and ends of both lines and paragraphs.

5.5 A MAIL READER

One use of a command language is as a general-purpose programming language. Flexible interpreted languages like Lisp are especially suited for developing user interface programs. Although the original purpose of MLisp, the extension language for UNIX Emacs, was to enable users to add editing functions and special modes, it was later used to implement applications such as mail readers and directory editors [Borenstein 88a]. One test of Whisper is whether it can serve the same purpose: is it powerful enough to support user interface development, in particular the construction of an electronic mail and bulletin board (bboard) reader?

As an experiment, I ported Batmail, an existing Emacs mail and bboard reader for Andrew, to Whisper. The experiment's hypothesis was that the Emacs/MLisp interface to the message server could be replaced by BrandX/XLisp. The experiment would be considered successful if the new program provided the basic functionality of the old given similar development effort.

Messages, the Andrew mail and bboard system [Borenstein 88b], is based on a model similar to Whisper's. It provides a well-defined programming interface that can be used to support different user interfaces. Messages is the basis of seven programs running in four environments on five machines including UNIX workstations, IBM PCs, and Macintoshes.

5.5.1 Emacs Batmail

Batmail, one of these programs, is a mail and bboard reading program that uses Emacs to provide a user interface. Batmail can process incoming mail, read messages in mail classes and bboards, and send mail. Like other Emacs-based mail programs [Borenstein 85], Batmail provides two windows, one for message captions and one for bodies. Figure 5.8 shows a typical Batmail display.

Batmail consists of a set of MLisp functions executed inside Emacs and a separate C program called Robin. The MLisp code implements the user interface, while Robin handles message processing. Robin is a small program that provides access to the Andrew Message Server (AMS) library. Figure 5.9 illustrates this process structure.

In Emacs, external programs communicate with MLisp functions using UNIX standard input and output. Emacs commands allow a function to start up a program, send data to it, and asynchronously receive data from it. Functions send and receive data as character strings. Robin, for example, accepts as input a set of commands that correspond to AMS library routine calls. It parses its input, calls the appropriate AMS routines, and outputs a sequence of MLisp commands to be carried out by the Batmail program. For example,

```
Emacs                          V2.10.37                          wheatland
Date: 31 Aug 1987 1306-EDT
From: Patricia Ann Fitzgerald <PF0A@TB.CC.CMU.EDU>
Subject: Turkey/heat transfer
To: Cooking@TB.CC.CMU.EDU

The article on the heat transfer when cooking a turkey is:

Ruebush, John and Fisk, Robert.  "Cooking a Turkey"  Mathematics Magazine.
v.53, no.4, September 1980, pp.237-239.

It is available in the E&S Library.

---- BatMail body of message 5
--- ext.cc.cooking - Since 1 July 1987 (5 messages)
1        2-Aug-87                                Michael Charles Wien@TD.  (578)
2        3-Aug-87  Cook Book                     Jeffery Paul Hansen @TB.  (304)
3       25-Aug-87  theory of turkey roasting     M014BL02@VB.CC.CMU.EDU (390)
4       29-Aug-87  Turkeys/heat transfer         Michael Charles Wien@TD.  (514)
5       31-Aug-87  Turkey/heat transfer          Patricia Ann Fitzger@TB.  (212)
---- BatMail captions    [? for help]  (82%)
```

Figure 5.8. The Batmail display

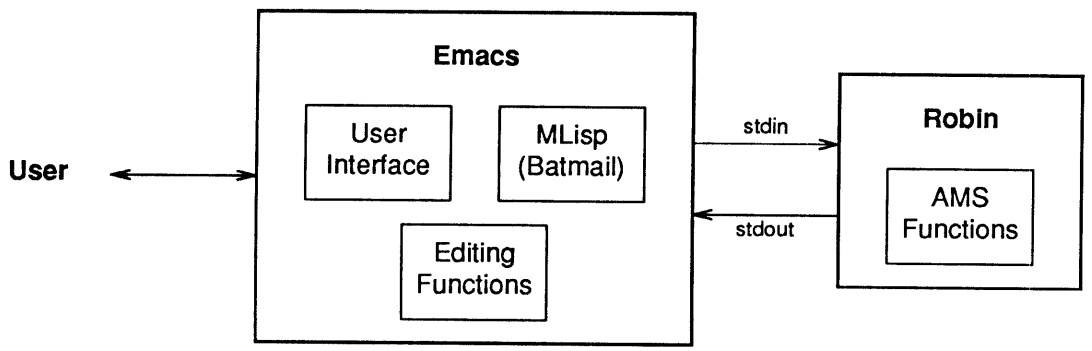


Figure 5.9. MLisp Batmail process structure

the Robin command delete-message, when successful, returns a MLisp command that updates the corresponding message caption, marking it as deleted.

This communication mechanism constrains application interfaces. All data must be passed as text strings in an uninterpreted stream of bytes. Communication is asynchronous, so while the MLisp code can model program input as function calls, no return values can be provided. Batmail solves this problem by implementing a *callback* mechanism. Operations are split into two parts: first input is prepared for Robin, and then the MLisp program waits for a call back from Robin to tell it what to do next. The callback informs the MLisp code that the function has completed and provides output results.

While Emacs itself does not deal with status and error messages, the Batmail callback mechanism handles them gracefully. At any time during a long operation, Robin can output a status command that, when executed within Emacs, displays a message to the user. Error messages can be issued the same way. Errors that should abort the current

operation are more complicated. Robin provides an error trapping mechanism that forces it to ignore new input until a special reset command is seen.

The constraints imposed by Emacs are exactly the constraints imposed by standard command languages, and the resultant problems are the same that a macro writer faces when creating a UNIX shell script. If a command language program is to invoke an interactive application, the program must communicate with the application via standard input and output. Like an MLisp program, the data passed to and received by the application is limited to text, and the program is unable to synchronize input and output. The Batmail solution is not available—usually the command language programmer is unable to modify the underlying application to provide a more powerful interface. Emacs uses the same model of applications as the UNIX shell languages and suffers the same limitations.

5.5.2 Whisper Batmail

Creating a BrandX/XLisp version of Batmail consisted of several tasks:

1. Defining a Whisper interface.
2. Modifying Robin to support this interface.
3. Modifying Batmail to call the new interface functions.
4. Modifying BrandX to support new functions needed by Batmail.
5. Translating the MLisp code into XLisp.

The user interface is essentially the same as the original; its display is shown in Figure 5.10.

Some changes were due to the Whisper process structure, shown in Figure 5.11, in which the Lisp interpreter is separate from the text editor. A user invokes a Batmail command inside BrandX just as he would inside Emacs. BrandX sends the command to XLisp, which executes it, making calls as needed to both Robin and BrandX. BrandX maintains two logical connections with XLisp, one to issue Whisper commands and one to receive and execute editor commands.

The key difference between the original Robin interface and the Whisper version is that input commands became interface functions. Callbacks were removed from all Robin routines. The information provided by the callbacks were turned into output parameters. Functions that issued a variable number of callbacks were split into stages. For example, the Robin function CaptionsSince asks for the new message captions in a folder. The Emacs version iterates through the new captions, issuing a callback for each one. In the revised program, the XLisp code does the iteration, repeatedly calling a function until all the new captions have been received. Making these changes to Robin can be viewed as

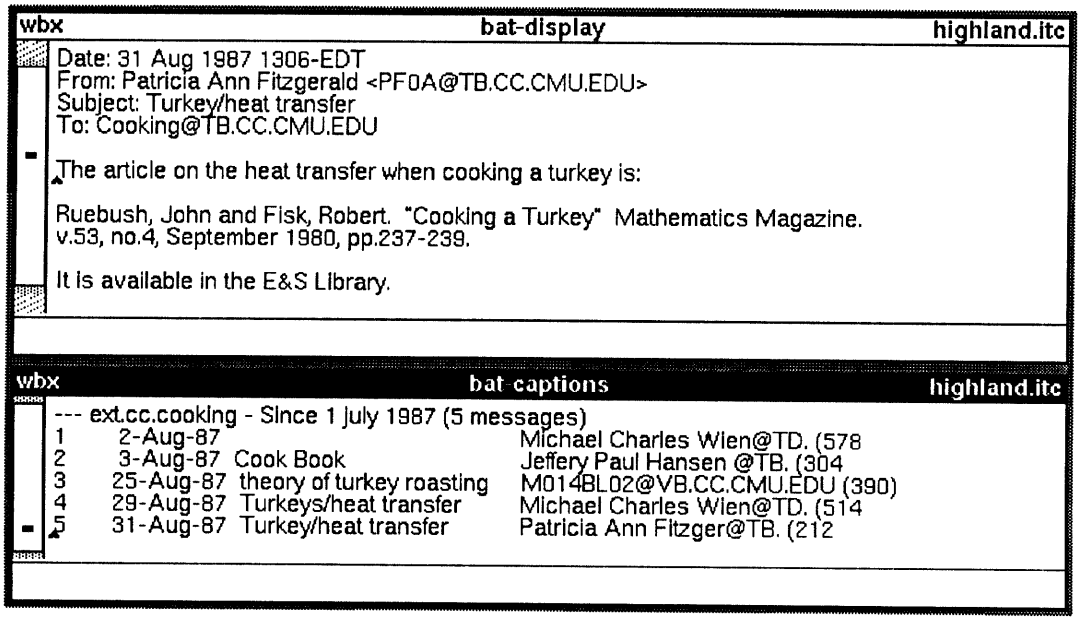


Figure 5.10. The Whisper Batmail display

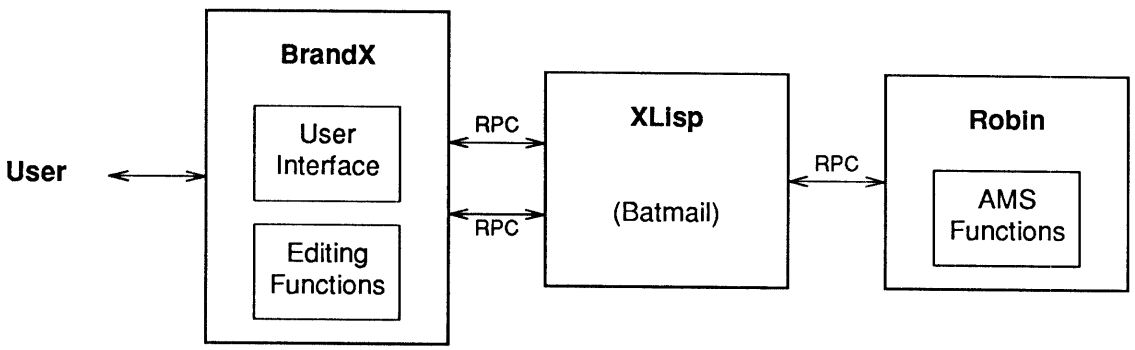


Figure 5.11. Whisper Batmail process structure

separating the database actions from its output commands, for Robin can be considered to be a simple mail program whose output is disguised as MLisp.

The changes made to the MLisp code to support the new Robin interface were straightforward. In most cases the changes were purely syntactic. In others (where functions were split into stages), code that was formerly in Robin was moved into Batmail. Functions that were callbacks now are called directly by the Lisp code rather than indirectly by the Lisp function that handles Robin output.

The final step of the project was to translate the Batmail MLisp code into XLisp. This was done by first writing a MLisp compatibility package in XLisp and then editing the Lisp code where there was not a one-to-one correspondence between BrandX and Emacs functions or where MLisp and XLisp used different syntax. BrandX and Emacs have similar models of text editing, and the programming interfaces provided by the two

programs are comparable,[5] although Emacs contains many functions not available in BrandX. However, the number of missing functions needed by Batmail were fewer than expected, and many of these were easily written in Lisp. Forty-five MLisp functions corresponded to existing XLisp and BrandX functions, while twenty-five Emacs operators were implemented in XLisp using lower level BrandX operators. Ignoring the Emacs operators used to communicate with external programs, only twelve operators were not converted to BrandX/XLisp. Using the compatibility package meant that much of the original Batmail code could be used without modification.

5.5.3 Evaluation

The tangible result of this experiment is Whisper Batmail (WMail), a mail and bboard reading program that behaves much like the original Batmail. The user can read a collection of mail classes and bboards, iterating through each folder and, within a folder, each message. He can reply to messages or send new mail. Because the underlying editor is BrandX and not Emacs, he can add BrandX styles and even insets to outgoing mail. Only a few Batmail features were left unimplemented, and all of these remaining functions could easily be added.

WMail is not nearly as fast or reliable as the original Batmail. While the application's performance could be improved, the implementation of the Lisp interpreter as a separate process ensures that WMail will be slower than Batmail. Each interface operation is a remote procedure call in Whisper, while only Robin (mail) commands are non-local in Batmail. Moreover, the number of editor operations is much greater than the number of mail operations. The number of context switches also slows down WMail since three processes are involved, not just two. However, if the interpreter were part of BrandX, performance would be as good or better than in Batmail. Then, the time required by an editor function call would be about the same for both Emacs and BrandX. Calls to Robin would use the current mechanism, which can be made faster than Emacs's RPC. (Numerical data need not be converted into text strings and back again as in Emacs. Also, Robin calls return typed values rather than produce string output that is interpreted as Lisp functions.)

It is difficult to compare the development process in building MLisp and Whisper applications because much depends on the functionality and reliability of the underlying applications. UNIX Emacs is a commercial product,[6] while BrandX is an unsupported, never-to-be-released program, and Whisper has only been subjected to limited testing by

---

[5] See Section 5.4.

[6] Unipress Software, Inc. maintains the version of UNIX Emacs available on Andrew.

a single user. The remainder of this section examines the development of WMail, taking these constraints into account.

The Robin component of WMail was produced almost trivially from the original program. Much of the conversion was a mechanical task, and the entire program was converted in less than two days. Most of the MLisp compatibility package was also written in just a day or two. However, the conversion of Batmail to WMail took about ten days, although this time included making changes to BrandX and XLisp.

Debugging was much slower than it is for typical MLisp programs, in part due to (fixable) bugs in the component programs, but in part due to the structure of Whisper. Some operations that are straightforward in Emacs are significantly more complex in Whisper. For example, keys in Emacs are bound to MLisp functions, which can be immediately evaluated by the MLisp interpreter. Under Whisper, a key in BrandX may be bound to an XLisp function. Here, the function must be sent to the XLisp interpreter by the light-weight process that handles user input. The evaluation of the function will often result in BrandX operator calls. A separate light-weight process in BrandX handles these requests, which may also require waiting for input (for example, if the operator prompts the user for a string), invoking yet another light-weight process. Bugs may occur in any of the three UNIX processes and in one of several light-weight processes in each UNIX process.

User error and status notification is more difficult in Whisper than in Emacs because the interpreter and the applications are not integrated. Although the interfaces I defined did not use it, Whisper does provide an exception handling capability. Rather than depending on return values to signify errors, an interface operator may signal an error that causes the XLisp interpreter to abort normal processing and enter an error state. BrandX functions can provide status information, just as Emacs functions do. However, other applications, such as Robin, have no direct link to the editor. Robin commands can issue calls back to the XLisp interpreter that signify intermediate states. For example, Robin issues bat-info commands during long operations. Bat-info is a Lisp function that calls a BrandX operator that displays a message in the current BrandX window. If Robin was used with an application other than BrandX, bat-info could be redefined appropriately.

Despite some problems, the Batmail experiment has shown that Whisper is as powerful as MLisp as a language for programming user interfaces, and is superior to it in several respects. Unlike MLisp, Whisper is not tied to a particular application. It provides a better outer language and a more flexible interface to external applications. Whisper has the potential to be an important application development tool in window systems.

5.6 SUMMARY

This chapter has discussed four case studies of Whisper use, from both the developer's and user's viewpoint. The emphasis of this chapter has been on interface design—what issues arise when a developer sits down to define a command language interface for an application. Much of the process of building an interface is independent of the particular applications involved. This section summarizes the steps to follow when designing a command language interface.

The application's interactive commands serve as a starting point for the Whisper interface. Mouse operations are often a collection of commands, of which one is chosen based on cursor location, active modes, and number of clicks of the mouse button. If this is the case, the mouse commands should be implemented as separate operations in the programming interface. In other situations, multiple commands can be reduced to a single parameterized command. Operations, both keyboard- and mouse-based, may have implicit parameters which should be made explicit in the command language interface in order to make macro generalization easier. Delta and the calculator provide examples of separation, reduction, and parameterization of commands.

The application's output as well as input must be examined. Some user commands return information, while most cause changes reflected in the application's display. Interface operations must be added that allow command language programs access to the same information. Delta, for example, provides an iterator that allows a program to examine each object in a drawing.

Together, the set of input and output operations should be checked for completeness at the semantic and, if possible, lexical level. While the existence of a programming-by-example front-end encourages the designer to provide an programming interface similar to the user interface, there are some operations that should be replaced by semantically equivalent ones. Elision can often bridge the gap between the user and programming interfaces in these instances. When a designer chooses to provide a lower-level interface to operations or data, care must be taken to ensure the safeness of the interface. Ideally, a command language program should not be able to cause a fatal error in the underlying application.

6 Evaluation and Conclusions

This chapter evaluates both the Whisper system and the research underlying it. Given the problem described in Chapters 1 and 2 and the system presented in subsequent chapters, how well does the solution fit the problem? I implemented the prototype system in order to test the design more thoroughly than would otherwise be possible. While previous chapters described the system implementation and a few examples, this chapter assesses Whisper as a whole and reviews some key design choices. It reviews the lessons learned from the design and implementation and suggests directions for further exploration. The final section summarizes the contributions of the thesis.

6.1 PROTOTYPE EVALUATION

This section examines Whisper based on the evaluation criteria presented in Chapter 2: completeness, feasibility, ease of use, efficiency, and generality.

6.1.1 Completeness

A command language should be complete, that is, at a given level of abstraction, it should be able to capture in written form what a user can do interactively. A user of a window system is able to interact concurrently with several applications and to transfer data between them. The command language must be able to do the same. The user carries out tasks by what can be a complex sequence of operations, each based on the output of previous steps. The command language must provide access to equivalent input operations and to the information, both textual and graphical, that the user obtains from these operations.

Whisper's completeness with respect to a system or set of applications is determined by the applications' interface specifications. Whisper provides application developers with the tools to achieve a complete system. It supports the definition of interfaces consisting

of objects and operations defined at the semantic level. While individual application interfaces define the inner language, Whisper provides an outer language that enables command language programmers to compose programs.

Whisper does not prescribe the contents of an application's interface and so cannot guarantee completeness. Instead, this thesis provides guidelines and examples that aid in designing good command language interfaces. The same strategy that is required to build high quality user interfaces must be used to build good (complete) command language interfaces. Careful analysis of the task leads to an initial design and implementation which then must be iteratively improved. Ultimately, the quality of a command language system lies in the hands of its application developers and not the implementor of the command language itself.

6.1.2 Feasibility

As described in Section 2.1.1, a command language can support a wide range of tasks. Feasibility measures the extent of this range. This section evaluates Whisper's appropriateness for several classes of tasks.

The simplest use of a command language is for writing macros, short unconditional sequences of commands. With a programming-by-example front-end, Whisper could be used for this purpose, although in some circumstances it will be more efficient to provide a separate input recorder. For example, UNIX Emacs provides a keyboard macro facility in addition to its extension language.

Many applications support customization by allowing users to change default settings in a personal profile read at application start-up. Users specify these settings with a language, most often an application-specific one. Whisper can substitute as a common language and, in fact, provides profile support for BX applications. Whisper provides the developer with a uniform and more powerful language, but its features may be superfluous if the application supports only limited customization.

Command languages also support "personal programming," the creation of programs to accomplish tasks that are too complex to write a keyboard macro for, yet simple enough not to require the use of a general programming language. Whisper is especially suited for this class of task. However, some tasks remain difficult because the interfaces provided by applications cannot be made semantically complete. An example is the use of macros inside applications, such as Telnet, that allow users to login to remote computer systems. A user might want to write a macro that provides his login name and password to the remote system. Many systems will not accept a password until they process the login name and print a password prompt. Whisper cannot be used to write a

macro that handles this task correctly. The difficulty here is that the activity in the window cannot be described by a Whisper interface and so must be dealt with lexically rather than semantically. The commands that may be executed are not part of the definition of the application. Telnet understands the initiation and termination of a connection to a remote machine, but once a connection is established, it views all user interaction as just a flow of characters to and from the remote host. Telnet could provide Whisper operators that permitted examination of the transmitted characters, but a macro would still be limited to looking for output patterns. When an application does not process input and output at a semantic level, it is unable to provide a semantic-level command language interface. This holds for any system, not just Whisper.

Some command languages can be used for more intensive programming. Whisper can be used for this type of task, as the Batmail case study showed. With some tuning, Whisper would perform well enough to serve as an implementation language for interactive applications.

Another use of command languages is to perform application testing. A tester could create Whisper programs that test application functions by performing operations and examining changes. These programs could not test the user interface, but this is often an advantage. Direct testing of the underlying semantic operations lessens the sensitivity of the evaluation. Small changes in application layout and display will not invalidate the test as long as the underlying operations do not change.

At first glance, Whisper does not appear to be suited for human factors testing. This class of test usually requires lexical recording—saving exactly what the user does, not a distillation of it. However, recording every user input event creates a huge amount of data which can be difficult to interpret. Some researchers have suggested that annotating the recorded data with semantic information can significantly aid analysis [Dance 87]. The Whisper programming-by-example front-end maps lexical input into semantic operations. By extending Whisper to save the original input along with these semantic operations, the system could produce just such a record.

6.1.3 Ease of use

A functional system is a necessary but not sufficient condition for a useful system. A user will carry out a task by hand or with a conventional programming language if the available command language is too difficult to use. Whisper is a tool for developers as well as users. A developer will choose to implement application-specific command languages or none at all if Whisper is hard to use. This section evaluates the ease of use of Whisper for both the end-user and the developer.

Whisper simplifies command language program by providing a common user interface to all participating applications. A programming-by-example front-end, such as the one described in Section 4.6, would further ease command language programming. In the simplest and perhaps most common case, users would never see the underlying command language. They would just record macros and play them back, and the command language would appear to consist of the commands typed interactively. A more sophisticated user could use the programming-by-example interface to generalize a recorded macro or add conditionals and iteration. More complex programs would be written as they are now, by creating programs using a text editor.

The design of Whisper ignores user assistance, a facility that would be important in a production version of the system. However, by extending the interface specification to include a description of each function, it would be easy to provide the command language programmer access to documentation.

Even with its textual user interface, writing Whisper programs is still simpler than writing programs in a general programming language such as C. Development is much more pleasant in an interpretive environment. Accessing application operators is easy, as is switching between applications. The model that the Whisper user is presented is that a set of interfaces is available, all from within the Lisp interpreter. However, because applications are separate processes, when things go wrong the user is forced to learn about failure modes that are not possible in a single-process system, as will be discussed in Section 6.2.3.

The interface Whisper presents to the developer is more refined than its basic user interface. Application interfaces are easy to specify, and the system provides a tool that translates specifications into C code. In most cases, only minor changes must be made to an application to incorporate Whisper. Because the BX toolkit uses external control, its clients handle most Whisper interaction transparently.[1] Other applications must add a small amount of code to permit the Whisper library to process incoming operator requests.

The bulk of the work for the application developer consists of defining the application's interface. This is a hard problem, and little can be done to automate it. Guidelines, such as those provided by this thesis, make the task of specifying an application interface easier.

The remote procedure call model hides many of the details of Whisper's multi-process implementation from application developers. And as is true for the user, when things go

---

[1] Section 4.4 explains external control.

wrong, developers must deal with the added complexity that multiple communicating processes entails. The light-weight process package that Whisper uses to manage application communication with the interpreter helps to separate the developer's code from the Whisper code.

6.1.4 Efficiency

Whisper's performance is acceptable as a prototype. Measurements and user experience show that some tuning and a few changes would significantly improve the system. This section reports on some measurements of the current system, while a discussion of performance enhancements can be found in Section 6.2.3.

Table 6.1 shows the time required to execute various function calls in XLisp and UNIX Emacs MLisp. Each entry reports the elapsed time required to evaluate a function on a Sun-3/50 with four megabytes of memory. The results were computed by measuring the time required to execute between 1,000 and 100,000 of the given calls using a stopwatch. The results show that a typical XLisp function evaluation is about an order of magnitude faster than the execution of a call to a Whisper interface operator. The time required to execute a Whisper operator is comparable to a call from XLisp to the Andrew window manager that returns data. The window manager uses an optimization that can greatly reduce the amount of time required by calls that do not return results. Using this optimization, calls can be as fast as an ordinary XLisp call or as slow as a Whisper call depending on the current state of the connection. This optimization is explained in Section 6.2.3. The table also shows that MLisp is about three times as fast as XLisp.

Table 6.1. Function call timings (msec/call)

Type of Call	Real Time
XLisp function (+)	1.3
Application function	18
Wm (with results)	19
Wm (input only)	1–19
MLisp function (+)	0.4

For many Whisper tasks, lack of speed is not a problem. Many command language programs are very short. In larger programs, the bulk of execution time often is spent inside application operations, and so a sluggish outer language does not significantly affect overall performance. In fact, this is one reason that command languages are used despite the availability of faster programming languages. However, the inner language of Whisper consists of application procedures, not whole programs, and therefore Whisper programs tend to consist of more small operations than do typical command language

programs. Also, as discussed in Section 6.1.2, execution speed is an issue when the command language is used to implement applications. These observations agree with those of Notkin and Griswold, who feel that speed is a key requirement for an application extension mechanism [Notkin 87].

6.1.5 Generality

The design of Whisper trades off system completeness for freedom provided to application developers. As demonstrated, Whisper can provide interfaces to a wide range of applications, including those most frequently used in the Andrew environment: text editors, drawing editors, and mail readers. The case studies emphasize direct manipulation interfaces since direct manipulation is a primary characteristic of user interfaces in window systems and are more difficult to handle than text-driven interfaces. Whisper can easily handle textual interfaces as well.

All of the applications for which I provided interfaces were written before Whisper existed, so it is clearly possible to add a Whisper-like tool to an existing system. As will be described below in Section 6.3, the structure of an application determines the ease with which Whisper can be incorporated.

Although the design and implementation of Whisper provides a command language for a heterogeneous environment, it also shows that homogeneity of applications can make a command language more powerful. While Whisper provides a mechanism for applications to share data, the applications themselves must agree on a common data format. One example is the standard definition of objects provided by the BX toolkit. BX, like other object-oriented environments, guarantees that certain operations will be available on a given set of objects. This guarantee makes it possible to write programs that manipulate objects inside documents (or drawings) without knowledge of the kind of object.

The generality of Whisper can, at times, be a drawback. Languages designed for specific applications can contain language features as well as functions that are especially tailored for the application. For example, the many variants of Emacs provide variables whose scope is limited to a specific file. Juno, a constraint-based drawing editor, allows the user to create procedures whose "statements" are constraints and whose "parameters" are unconstrained variables [Nelson 85]. A procedural approach to interfaces will not always serve as a satisfactory substitute for the abstractions used by a particular application. In part, this is the rationale behind the "little languages" approach discussed in Section 3.1.1.

6.2 REVIEW OF KEY DESIGN DECISIONS

This section examines several of the important decisions made during the design and implementation of Whisper. It explains the alternatives available, why the particular choices were made, and how each decision affected the final system. The four issues discussed are the choice of testbed system and base command language, the decision to implement the system using separate heavy-weight processes, and the strategy chosen to provide access to application data.

6.2.1 Testbed system

While the choice of Andrew, UNIX, and C as the testbed for Whisper affected many design decisions, the ideas behind this research can easily be applied in other environments.

In a Lisp or Smalltalk system, the programming language or a subset of it can double as the command language. All applications share a single address space, and the command language can provide access to applications' internal data and operations. Under these circumstances, several design issues in the current system become moot. Because the command language and applications exist within a single address space, the complexities of inter-process communication can be avoided. Some modes of failure are no longer possible. Communication bandwidth between applications and the command language is higher. Garbage collection does not need to be coordinated between separate programming and command languages.

Despite these simplifications, a basic problem remains. Providing access to applications' internals does not guarantee a usable command language for a window system. Developers must still supply appropriate application interfaces. While some details of an interface are language-dependent, the problems and guidelines identified in this thesis apply to the design of text-based interfaces for any graphical application.

Similarly, using a user interface management system or a toolkit other than BX, multiple toolkits, or no toolkit at all would have changed some of the implementation details but not the overall design of Whisper. As noted in Section 5.1, three policy decisions in the Base Editor facilitated the addition of Whisper interfaces to applications:

- BX enforces separation of output from core functionality by restricting output operations to the Update and Redraw methods.[2] This rule improves the efficiency of Whisper programs since display updates occur only when a command language program completes execution or waits for input.

---

[2]Table 5.1 described the standard BX methods including Update and Redraw.

- BX uses external control, allowing Whisper to hide most of its changes from clients by placing them inside the toolkit's main loop.
- BX defines a standard structure for application components, allowing developers to construct applications by composing these objects. This standard structure, extended by Whisper to the command language interface, allows command language programs to pass data between components.

Supporting Whisper on a system without these properties would have been more difficult.

6.2.2 Base command language

While the Whisper architecture imposes certain constraints on the command language included in the system, it does not define much of the outer language's specific syntax and semantics. The architecture requires the command language to accept as its inner language a collection of functions implemented by independent applications. The language should provide a standard set of data types including numbers and strings and should include control structures for conditional tests and loops.

The decision to use Andrew as the testbed for Whisper limited the choices for base command language. I considered the UNIX C shell and two dialects of Lisp, XLisp and the extension language for GNU Emacs. I chose XLisp because it is small and easy to modify.

Since the base language for Whisper is a Lisp dialect, it easily supplies the programming features needed by a command language. Others have made this same observation—several shell languages are based on Lisp [Ellis 80, Giuse 85]. The language's expressiveness makes many macros easier to write. The results of the Batmail experiment were made clearer because it was possible to separate the effect of the underlying language (essentially the same) from the differences in interfaces.

While Lisp provides a few advantages to Whisper, the system is not dependent on it. The RPC mechanism provides a clean, language-independent interface between applications and the command language. The Whisper architecture can easily support multiple outer languages.

6.2.3 Multiple process architecture

A fundamental decision in the Whisper design was determining how to structure the system's processes—whether to implement the interpreter as a separate heavy-weight process or link it into each application. Designing the interpreter as a separate entity fundamentally changed the character of the system, trading off performance for functionality.

In the Andrew system, most applications exist as separate UNIX processes. UNIX processes are considered *heavy-weight*—each process has its own address space and requires a significant amount of time to start up. Processes communicate via files or special objects called *sockets* that asynchronously send and receive arbitrary data. While the basic operations on sockets do not reliably transmit data, a remote procedure call package can be used to provide the semantics of procedures. The RPC package allows the programmer to model communication with other processes as ordinary procedure calls. The caller passes arguments as for a local procedure and execution halts until the procedure returns. Three major differences exist between local and remote procedure calls:

- Remote calls are significantly slower than local ones.
- A remote call can fail in ways that a local one cannot.
- Because the caller and receiver exist in separate address spaces, call-by-reference cannot be provided in remote procedure calls.

This section discusses why a multiple process architecture was chosen despite the loss of performance and added complexity, and how these problems can be minimized.

Placing the command language interpreter in its own process makes possible one of Whisper's key features: the ability to write programs that involve multiple applications. The Whisper mail reader, for example, contacts the mail application to obtain messages and then uses the text editor to display them. Macros can be written that transfer data between applications or use the results of an operation in one application to determine the next action in another. This feature corresponds to the user's ability to interactively transfer data between windows. If Whisper were only available within applications that linked in the interpreter, it would be only an extension language for particular languages, not a general-purpose command language. The multiple process structure also allows Whisper to support applications written in different languages and that exist on different computers, even different types of computers.

Given that multiple processes are so useful, what can be done to reduce the costs? As noted earlier, command language macros are often quite short, and sluggish RPC performance is not noticeable in these macros. In other cases, the cost of communication is small compared to the total computation cost. Other researchers have also noted this effect [Fitzgerald 86, Lantz 85]. In the long run, faster RPC mechanisms due to improvements in both hardware and software will help the most. Nelson has demonstrated an RPC system two orders of magnitude faster than the one used for Whisper [Nelson 81], and Tanner's Switchboard program [Tanner 86] uses multiple processes in which the round-trip call time is 1 millisecond rather than Whisper's 18 milliseconds. In the short run, there are several optimizations that can help Whisper.

As was shown in Table 6.1, a Whisper remote procedure call takes about 18 milliseconds

of which 1 millisecond is XLisp function call overhead. Of the remaining time, approximately 6 milliseconds are used to execute system calls, 6 milliseconds are used by the low-level RPC package, and the remaining time is used by Whisper to pack and unpack arguments and to find the function to be executed. Because the applications that will be accessed by the command language are not known in advance, this information cannot be compiled into the interpreter. However, it can be compiled into the application interface, resulting in larger but faster applications.

Another approach to improving performance is to group a set of calls into a single packet, amortizing the cost of each call over the group. The Andrew window manager and several other programs use this technique, called *batching*, to greatly improve performance [Gosling 86a, Gosling 86b, Scheifler 86]. Batching can only be employed if procedures do not return results. Otherwise, a call must be executed as soon as it is issued. This technique works well for window managers, where most interface procedures are graphics commands. This explains the entries for the Andrew window manager in Table 6.1. When a call returns data, a real remote call is made, taking about the same time as a Whisper call. When no output is involved, the call may be as trivial as copying a single byte into a data structure and may be as fast as a local function evaluation. Whisper interfaces have many more functions that return data, but this number can be reduced by careful interface design.

The added complexity of remote procedure calls must also be dealt with. Using RPC forces the language and the programmer to consider errors that are impossible in a single process system. The most common problem is that the remote process has experienced a fatal error. The easiest thing to do in this case, and Whisper's default action, is to force this to be a fatal error for the command language program. More sophisticated programs can catch this error and perform whatever action may be appropriate.

A simpler approach is possible in some environments. For example, in the latest version of Andrew, the text and drawing editors, the shell interaction window, and the spreadsheet program can all be part of the same process. In a system such as this one where a single application predominates, and in systems where applications share one address space, the interpreter can be linked in to the principal application, greatly increasing the performance of command language programs that access only that application. Other applications can still be accessed using remote procedure calls. Not only is no functionality lost, but the language can be tailored to the principal application. An example of a feature that cannot be provided by the current implementation of Whisper is the use of variables associated with windows. The principal application need not be an editor. It could be a CAD program, a database manager, or a spreadsheet.

6.2.4 Object references

One of the most difficult aspects of Whisper's design was deciding how application data should be accessed from the command language. This section explains why access should be provided and discusses some alternatives. While the use of multiple processes affected the choices made, data reference problems are independent of process structure.

To ensure completeness, the programmable interface must provide functions that correspond to the interactive user's ability to identify objects of interest. In a text-based interface, the user must explicitly describe objects. In a direct manipulation interface, the mouse can be used to identify text to be deleted, a mail message to be examined, or a rectangle to be filled. The user interface provides mechanisms to display objects and choose from among them. The programmable interface must provide equivalent functionality.

Interface operations that simply return the state of objects are also important. These operations correspond to the interactive user's ability to examine the application display. A side effect of almost every command in a text editor is a change to the visible document. The programmable interface must provide access to the contents of the document to enable a command language program to make decisions according to the document's state.

Providing object references involves three separate decisions:

- How are objects accessed?
- Are object references explicit or maintained inside applications?
- How is the validity of references checked?

The two editors examined in Chapter 5, Delta and BrandX, provide interface functions that enable a program to iterate through a set of objects. The caller of these functions may specify parameters that narrow the selection of objects, for example, limiting the set of drawing objects to lines or circles. These operations can be considered a primitive form of database query.

These operations can be extended to provide a more general database capability, supporting uniform access to a wide range of data. Each application stores data in its own format, while the Whisper interface provides a query language that is the same across applications. SmallStar data descriptions could provide a front-end for data access.

References to objects can be made implicit by forcing interface functions to operate on default objects or explicit by using parameters in interface functions. As noted in Chapter 5, while Whisper application interfaces use both styles, explicit references make macros easier to generalize.

If a system supports explicit object references, tools should be provided to help applications manage them. Functions that use references should be protected against illegal values. Section 4.4 described the mechanism used by Whisper to manage object references.

6.3 IMPLICATIONS FOR INTERACTIVE APPLICATION DESIGN

Much of this research has focused on the design of command language interfaces for representative applications and on the relationship between these interfaces and the applications' implementation. The quality of application interfaces determines the success of Whisper; an application's design limits the range of choices in design of its interface. This section discusses some of the implications of the thesis research for the design of interactive applications.

A Whisper interface possesses characteristics of both user and programming interfaces. It provides a text-based user interface whose purpose is to support command language programming. To design a good Whisper interface, a developer must apply principles of good programming interface design *and* good user interface design.

Good user and programming interface designs share some common properties. Designers of both user and programming interfaces can adopt an object-oriented or abstract data type approach, organizing an interface around the objects provided by the application's underlying functionality and the operations available on these objects. Consistency is desirable in both kinds of interfaces—similar functions should behave similarly, allowing a user of the interface to apply knowledge of how one function works to others. As in any design task, simplicity is important.

While Whisper interface designers can and should follow these common principles, they must also balance some contradictory ones. The users of the two kinds of interfaces can be quite different. Presumably more sophisticated than ordinary users, programmers are more likely to accept a complex interface as the price for a more powerful interface.

This trade-off appears in several forms. User interfaces are usually more closely tied to the conceptual model behind the application. In fact, the key principle behind direct manipulation user interfaces is that the user should directly interact with these underlying objects and operations. While a programming interface may contain a minimal set of operations, a user interface usually contains a larger set of higher-level, less orthogonal operations. A designer often limits the flexibility of a user interface in order to make an application easier to use. Modeless interaction forces many operations to be parameterless, forcing the designer to choose default values for operands. Unlike user interface operations, the operations provided by programming interfaces are building blocks, not ends in themselves.

Given these general guidelines and the ones specific to command language interfaces discussed in Chapter 5, the remainder of this section addresses the impact these goals have on application structure.

The designer of a Whisper application has a great deal of latitude in determining the structure of the application. This flexibility is intentional, for one of the design goals of Whisper was to minimize the constraints the command language imposes on applications. In practice, however, Whisper encourages a layered approach which increases consistency and minimizes duplication of code.

An application designer can implement command language interface support independently of the application's user interface, as shown in Figure 6.1.[3] This approach places few limits on the application, permitting the user interface and core functionality to be implemented together or separately.

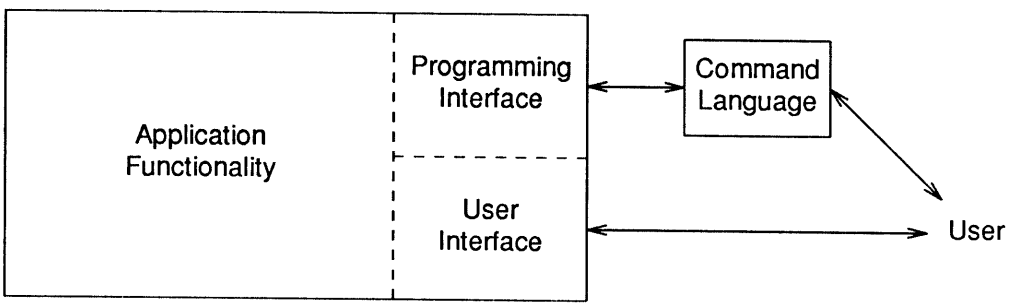


Figure 6.1. Separate programming and user interfaces

An alternative approach is to implement the command language and user interface as clients of a single programming interface (Figure 6.2), separating the user interface from the core application functionality. This layered approach minimizes the overhead of using Whisper by sharing as much code as possible. Separation has other advantages as well, as many user interface researchers have pointed out [Tanner 85, Olsen 83]. Separation supports experimentation with user interface designs, multiple user interfaces, and reusable user interface tools.

Many user interface management systems and toolkits enforce separation [Dance 87, Szekely 88]. A developer provides a description of the application's functionality and its user interface, and the UIMS generates a specification that can be executed by the UIMS run-time support modules. (See Section 3.1.2.)

Whisper can piggy-back onto a UIMS, replacing its own interface specification language

[3]Figures 6.1 and 6.2 appeared earlier as Figures 4.4 and 4.2, respectively.

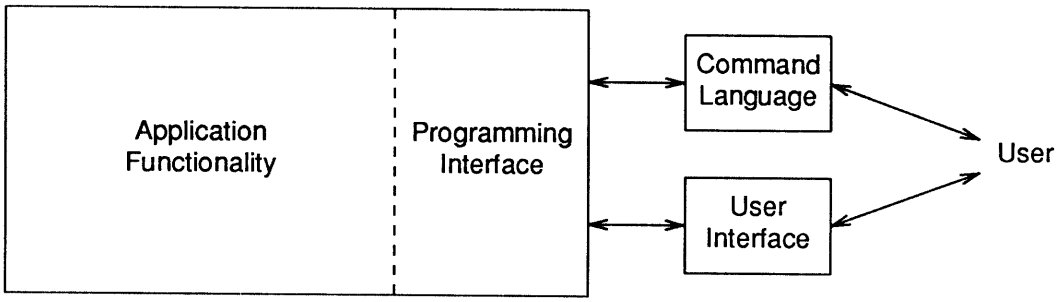


Figure 6.2. User interface and command language as clients

with the UIMS's. Tinylisp (described on page 33) takes a similar approach. Core application functionality is implemented as a collection of modules. Each module provides an interface which defines the operations available for use by other modules *and* by the command language. However, even when the interface mechanism supports the use of a common definition to provide interfaces to both the application and the command language, developers still must take into account the differences between the two kinds of interfaces.

Beyond the difference in the type of interface—general-purpose programming versus command language programming—is a difference in what the interfaces separate. While a UIMS divides the application from its user interface, a command language interface must separate the application from a part of the user interface, from user interaction. According to the UIMS perspective, all user interface activity is beyond the scope of the application, but for a command language, this is not the case. The command language interface should provide control over the user interface, both input and presentation. For example, the drawing editor interface discussed in Chapter 5 permits the command language programmer to determine the portion of the drawing to be displayed. Similarly, the interface to the Base Editor library permits command language programs to bind keys to functions and display messages to the user. The command language must provide I/O functionality as well as core application functionality since the objective of a command language program may be to alter the presentation of application data as well as to change the data itself.

These principles, separation of core functionality from user interaction, and command language control of user interface operations, have two further consequences, both of which can be applied to any interactive application in order to provide greater modularity:

- Input should be separated from output.
- Parameters of interactive commands should be explicitly identified and controllable from the command language interface.

These guidelines assume that code that is nominally part of the user interface performs

functions useful in command language programs. Most such functions map user interface (lexical) concepts to and from application (semantic) concepts, as shown in Figure 6.3.[4] Examples include the translation of mouse coordinates to application-specific data and the translation of error and status information into a message readable by the user. Even the writer of a non-interactive command language program may want to use these functions rather than invoke the application functionality directly.

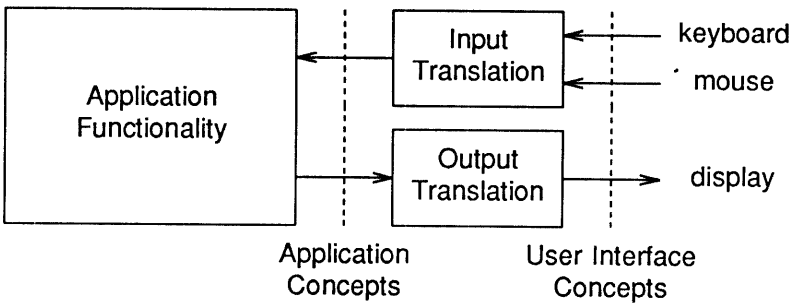


Figure 6.3. Input and output translation

Separation of input and output is required in order to perform some basic command language tasks. One of the most common uses of a command language is to package a sequence of interactive commands. Even though the macro replaces the keyboard and mouse as the source of input, the command language programmer still wants to produce the usual output on the display. Conversely, the programmer may want to invoke interactive functions but intercept their output. Separation permits macro writers to selectively make use of input and output translation.

Related to this separation is a refinement of the definition of input and output. Some display effects, although literally output, should be considered part of the process of obtaining input. Pop-up menus and dialogue boxes are two examples. A command language program may want to display application data while disabling prompts used to gather input.

Like the separation of input and output, control of interactive command parameters provides the command language programmer greater control over an application and its user interface. It also makes possible the mapping from user interface commands to command language interface commands required by a programming-by-example front-end to the command language. Parameters include objects and coordinates selected using the mouse as well as text answers to prompts.

---

[4]This diagram is based on the structure for interactive applications proposed by Szekely [Szekely 88].

While few researchers would argue against modularity, separation has proved difficult to achieve in practice. Direct manipulation interfaces in particular require greater linkage between applications and their interfaces in order to provide semantic feedback and rapid update of continuously visible objects. Experience with Whisper confirms these problems. (In particular, see Section 5.3.3.) Current research has made some progress [Szekely 88, Cockton 87], but no one has found a definitive answer. Only continued experimentation and evaluation will make separation easier.

6.4 FUTURE WORK

This research has suggested several directions for future work.

I intentionally left unimplemented some components of Whisper in order to limit the scope of the thesis. I presented a design for a programming-by-example front-end in Section 4.6. Another part of the user interface that would be useful to implement is the set of generic data query functions briefly described in Section 6.2.4.

The ultimate test of software is acceptance by users. Whisper should be incorporated into a live system, one currently in use. The obvious candidate is the current Andrew toolkit, since it is quite similar in goals and design to the BX toolkit used by the prototype Whisper. Once the system moves to a new base, more applications can become programmable, leading to further understanding of application interface design. One key application that should provide a command language interface but does not in the current system is the window manager itself. A wider range of applications could be supported as clients, including some that already implement their own language. Whisper should be able to provide a standard outer language for these applications. Increased usage of the system will make more extensive evaluation possible and will help find weaknesses in the current design.

This chapter has identified some key design decisions that determined the shape of the prototype, including choice of a particular testbed system and base command language, and the use of a multiple process architecture. Moving to a new system and command language would test Whisper's portability and provide further understanding of the interaction between Whisper and user interface toolkits. A new single-process implementation of Whisper would help determine the importance of efficient execution of the command language.

One of the goals of Whisper is to capture in a command language the interaction that the user carries out with a mouse. As new input devices become generally available, the command language should be able to cope with these as well.

The Whisper approach pushes the burden of representing output in a meaningful way onto the application interface writer. Two issues should be explored in this area: achieving a better understanding of the pattern recognition process that users carry out when viewing program output, and developing more structured output techniques that would make interface specification less ad hoc.

While the emphasis of this work has been to support macros and "personal programming," the mail reader experiment has shown that Whisper can be used for user interface development. Further investigation, including a comparison of Whisper with other development tools, would lead to an improved system for building user interfaces.

A command language provides an easy-to-use interface to an underlying system. This work has sought to enhance command languages to deal with a set of new resources, window systems and applications with direct manipulation interfaces. Command languages should be extended to support other developments as well, for example, providing simpler interfaces to multiprocessors and multicomputer environments.

6.5 THESIS CONTRIBUTIONS

The objective of this work was to build a general command language for a sophisticated multi-window, multi-processing system. This has never been tried before because these systems are new and because the problems inherent in providing a command language for a window system are hard. This thesis has examined the requirements of command languages and has discussed the properties of window systems and direct manipulation interfaces that conflict with these requirements. These conflicts explain why some obvious approaches fail.

I have described a practical strategy for providing a command language and have demonstrated its potential by implementing a prototype system. Application developers specify interfaces that define the objects and operations of their applications at a semantic level. The system incorporates these operations into the command language, enabling users to write command language programs that access operations of one or more applications. The thesis outlines the design of a direct manipulation user interface to the system that allows users to create command language programs by example.

Providing a programmable command language interface which captures the semantics of an interactive application can be difficult. The thesis has presented example interfaces for a set of applications and extracted from these case studies guidelines for both application developers and implementors of toolkits and user interface management systems. These guidelines, in addition to supporting command language interfaces, aid in the construction of more flexible and more modular user interface software.

Over the past few years, computers have become ubiquitous. As the range of users grows ever wider, it becomes increasingly difficult to design user interfaces that are appropriate for all uses and all users. While relatively few people will ever become programmers or even write macros, tools that support extension and customization can simplify the process of providing interfaces tailored towards specific classes of users. By investigating one set of tools, this thesis has contributed to the achievement of a major goal of computer science research, the development of easy-to-use computer systems.

Appendix A
Additions to XLisp

This appendix describes the user-level additions to XLisp that implement Whisper. Most of these functions are defined in Lisp and depend on a small number of primitives written in C. (These primitives are not described here.)

The following functions provide the user the ability to manage the current active application:

begin-application *application* { *argument* }* &key :display
Loads the interface file for the named application. If the interface file is found, the application is invoked. If the application starts up successfully, the name of this instance of the application is returned. This application is pushed onto a stack of active applications. If the keyword :display has a nil argument, then the application will not display itself. The function returns nil if the application does not successfully start up.

select-application *application*
Changes the current active application. The function returns t if successful.

end-application &optional *application* &key :quitop
Terminates the named application (default is the current one) by sending a quit message to it. All applications must include quit as an interface function. If the quit operation is successful (returns true), the application is popped from the stack of active applications. If the keyword :quitop is specified, its argument is used instead of quit to end the application. The function returns t if successful.

The following function is used in macros declared in interface files to hide the distinction between ordinary and external function calls. The user never calls directly this function.

execute-op *op-name* { *argument* }*
Apply the named operator to the specified arguments by sending a message to the current application that includes the operator name and arguments.

Application variables are referenced using the same syntax as ordinary variables. No special functions are required to access them.

The following functions are used in interface files to specify an interface. These functions are never invoked by the user.

application *application* &optional *program-name version*
Define a new application. An application must be defined before it can be invoked (using begin-application).

module *module* &optional *version*
Define a new module in the current application.

defavar *variable*
Declare *variable* as a variable in the current application interface.

defop *operator*
Declare *operator* as a function in the current application interface.

The following functions are invoked by applications. They cannot be called by users.

application-hello *application* &optional *version*
Inform the interpreter that the named application has been executed. Applications that want to execute Whisper functions initiate communication with the interpreter using this call. The interface file for the application is loaded. If a version number is specified, it must match the one contained in the interface file. Returns an RPC ID to use for future calls or zero if unsuccessful.

load-new-interface *rpc-id interface* &optional *version*
Load the named interface as part of the application specified by the *rpc-id*. If a version number is specified, it must match the one contained in the interface file.

application-goodbye *rpc-id*
Inform the interpreter that the named application is about to exit. Returns t if successful.

eval-string *expression*
Evaluate the expression (a string) and return the expression's output and results as a string.

Appendix B
The Interface Generator

Wgen is the component of Whisper that generates Lisp and C code fragments based on application interface specifications. These code fragments implement the remote procedure calls between the Lisp interpreter and the application. While the implementation of wgen is C-specific, generators could be written for other implementation languages. This section describes the language used in interface specifications.

application *application* &optional *file-name version*
Name the application for which operators are being defined. File-name specifies the corresponding file that implements this application. The version number is used to confirm that the application interface matches the program.

module *module* &optional *version*
Name a part of an interface for which operators are being defined. The module is added to the current interface. Modules are often interfaces to libraries that are included in more than one application.

defop *operator implementation in-arguments out-arguments* &optional
external-return-type internal-return-type
Make available a function called *operator* to the XLisp programmer corresponding to *implementation* within the application. The in and out arguments consist of lists of types and parameters. The standard types are integer, float, character, string, and boolean. Object types declared by the deftype function and automatic parameters declared by the defparam function can also be used here. Either list may be nil. Instead of a type name, an in argument can be a list consisting of a type name followed by the keyword :default followed by a constant value. This value will be supplied to the implementation routine when if the user does not provide one. A single out argument is returned as the function's value and multiple out arguments are returned as a list. If a function returns a value, the function's result is appended to the list of out arguments. The external type is the Lisp type, while the internal type is the implementing function's type.

defavar *variable address-expr base-type variable-type settable* &optional *base-name*
Make available an application variable called *variable* to the XLisp programmer cor-

responding to *address-expr* within the application. The *address-expr* is any legal C expression that can be resolved to an address including variables and array and structure references. *Settable* is a boolean that specifies if the XLisp programmer can change the variable's value. If the variable is indirect (an array or structure reference), *base-name* specifies the name of the underlying variable and *base-type* its type. *Base-name* is optional if it can be determined from the *address-expr* (when the *address-expr* is the *base-name* followed by ".", "[", "->", or "(").

deftype *type*
Declare an object type for use in operator and variable declarations.

defparam *parameter address-expr base-type parameter-type* &optional *base-type*
Specify a parameter that may be automatically passed to routines that are invoked by operators. This permits the application developer to define operators implemented by routines requiring parameters that she wants to remain hidden. *Parameter* is the name given in defop commands. The remaining parameters have the same meaning as they do in defavar.

definset *inset inset-type*
Declare a parameter that is an inset. Only applicable to BE2 applications.

load *lisp-file*
Copy the load command to the generated Lisp file. Useful if the application developer wants to define additional Lisp functions for use with an interface.

Appendix C
Some Whisper Interfaces

This section presents some annotated Whisper interfaces: the calculator, Delta, and BrandX.

C.1 A CALCULATOR

The Whisper interface to the calculator consists of two modules: calculator and calc.

```
; File: calculator.w

(application 'calculator "calc" 1)

(defop quit w_Exit (nil) () boolean)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: calc.w

(module 'calc 2)

(definset current-calc calc)

(defop calc-enter Enter (current-calc float) () float)
(defop calc-digit KeyDigit (current-calc character) () float)
(defop calc-point KeyPoint (current-calc) () float)
(defop calc-function KeyFunction (current-calc character) () float)
(defop calc-clear-entry KeyClearEntry (current-calc) () float)
(defop calc-clear KeyClear (current-calc) () float)
(defop calc-negate KeyNegate (current-calc) () float)
(defop calc-equals KeyEqual (current-calc) () float)

(defavar display "(CurrentInset(calc))->display" "" float nil)
(defavar register "(CurrentInset(calc))->registr" "" float nil)
```

C.2 A DRAWING EDITOR

The Whisper interface to Delta consists of three modules: delta, decmds, and ccmds.

```
; File: delta.w

(application 'delta "wdelta" 1)

(defop quit w_Exit (nil) () boolean)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: decmds.w

(module 'decmds 1)

(definset current-de de)

; Toggles state & returns current setting.
(defop decmds-toggle-palette decmds_TogglePalette (current-de) () boolean)
(defop decmds-toggle-scrollbar decmds_ToggleScrollBar (current-de) () boolean)

; Returns current state of palette and scrollbars.
(defop decmds-get-state decmds_GetState (current-de) (boolean boolean))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: ccmds.w

(module 'ccmds 2)

(load "ccmds-aux")

(deftype inset)
(deftype object "dr_Object *")

(definset current-canvas canvas)

(defparam current-x "CurrentInset(canvas)->thisPt.w.x" "" "WorldCoord")
(defparam current-y "CurrentInset(canvas)->thisPt.w.y" "" "WorldCoord")
(defparam last-x "CurrentInset(canvas)->lastPt.w.x" "" "WorldCoord")
(defparam last-y "CurrentInset(canvas)->lastPt.w.y" "" "WorldCoord")
(defparam current-object "wccmds_GetCurrentObject(CurrentInset(canvas))" "" "dr_Object *")

; (c-co) => object -- get the first selection
(defop ccmds-current-object wccmds_GetCurrentObject (current-canvas) () object)

; (c-scp x y) -- set the current point for create, etc.
(defop ccmds-set-current-point wccmds_SetCurrentPoint (current-canvas float float) ())
; (c-gcp) => x y -- get the current point for create, etc.
(defop ccmds-get-current-point wccmds_GetCurrentPoint (current-canvas) (float float))

; (c-slp x y) -- set the last point for create, etc.
(defop ccmds-set-last-point wccmds_SetLastPoint (current-canvas float float) ())
; (c-glp) => x y -- get the last point for create, etc.
(defop ccmds-get-last-point wccmds_GetLastPoint (current-canvas) (float float))

; an op, not a var, since it must be checked & it changes display
; pass it a bad value (0) to get current value
; (c-st tool) -- set the current tool for create
```

```
; see canvas-aux for tool names
(defop ccmds-set-tool wccmds_SetTool (current-canvas character) () character)

; (c-co [x2] [y2] [x1] [y1] [join]) => created-object
; create an object of the current type using the 2 points as if they were
; generated using the first and second clicks of the mouse.  If join and
; the type of object is joinable and the first point coincides with last
; point of the previously created object, make these points one.  The
; points are given in reverse order to make it easy to create a set of
; connected lines.
(defop ccmds-create-object wccmds_CreateObject
   (current-canvas
      (float :default current-x) (float :default current-y) (float :default last-x) (float :default last-y)
      (boolean :default t))
   () object)
; (defop ccmds-define-object (current-canvas string) character)

; use these for box too by adding last as 2 more optional params
(defop ccmds-select-by-location wccmds_HitDrawing
   (current-canvas nil (float :default current-x) (float :default current-y)) () object)
(defop ccmds-extend-by-location wccmds_HitDrawing
   (current-canvas t (float :default current-x) (float :default current-y)) () object)

; The integer is the handle.
; Passing 0.0 => use current and last points.
(defop ccmds-move-selection wccmds_MoveSelection (current-canvas (float :default 0.0) (float :default 0.0))
   () object)
(defop ccmds-move-handle wccmds_MoveHandle
   (current-canvas (integer :default -1) (float :default current-x) (float :default current-y)) () object)

; Select the objects that match the string and set up so that next will
; cycle through the selections, making the next selection the first
; selection.  If the string is "", just set up first & next.
(defop ccmds-first-object wccmds_FirstObject (current-canvas (string :default "")) () object)
(defop ccmds-next-object wccmds_NextObject (current-canvas (string :default "")) () object)

; Useful to go thru list & reselect or get rid of uninteresting ones.
(defop ccmds-select-object wccmds_SelectObject  (current-canvas object) ())
(defop ccmds-deselect-object wccmds_DeselectObject  (current-canvas (object :default current-object)) ())

(defop ccmds-get-n-vertices wccmds_GetNVertices ((object :default current-object)) () integer)
(defop ccmds-get-vertex wccmds_GetVertex ( integer (object :default current-object)) (float float))

(defop ccmds-get-object-type wccmds_GetObjectType ((object :default current-object)) () string)
(defop ccmds-get-data-type wccmds_GetDataType ((object :default current-object)) () string)

; (defop ccmds-object-type (character) (string))

; Scrolling operations.
(defop ccmds-get-visible-box wccmds_GetVisibleBox (current-canvas) (float float float float))
(defop ccmds-show-point wccmds_ShowPoint (current-canvas float float) ())
(defop ccmds-get-drawing-box wccmds_GetDrawingBox (current-canvas) (float float float float))

; Zooming operations.
(defop ccmds-halve-size ccmds_HalveSize (current-canvas) ())
(defop ccmds-double-size ccmds_DoubleSize (current-canvas) ())
; pass 0 to set-zoom to get current zoom
(defop ccmds-set-zoom ccmds_SetZoom (current-canvas float) () float)

; Other ops.
(defop ccmds-clear ccmds_Clear (current-canvas) ())
```

```
(defop ccmds-duplicate ccmds_Duplicate (current-canvas (float :default 0.0) (float :default 0.0)) ())
(defop ccmds-reshape ccmds_Reshape (current-canvas) ())
(defop ccmds-select-all ccmds_SelectAll (current-canvas) ())

(defop ccmds-read-drawing ccmds_ReadFile (current-canvas) ())
(defop ccmds-write-drawing ccmds_WriteFile (current-canvas) ())
(defop wccmds-read-drawing wccmds_ReadFile (current-canvas (string :default "")) () integer)
(defop wccmds-write-drawing wccmds_WriteFile (current-canvas (string :default "")) () integer)

; more inset commands
(defop ccmds-goto-dinset wccmds_GotoDInset (current-canvas (object :default current-object)) () inset)
(defop ccmds-first-dinset wccmds_FirstDInset (current-canvas (string :default "")) () object)
(defop ccmds-next-dinset wccmds_NextDInset (current-canvas (string :default "")) () object)
(defop ccmds-inset-to-dinset wccmds_InsetToDInset (current-canvas inset) () object)
(defop ccmds-dinset-to-inset wccmds_DInsetToInset ((object :default current-object)) () inset)
```

C.3 A DOCUMENT EDITOR

The Whisper interface to BrandX consists of ten modules.

```
; File:  bx.w

(application 'bx "wbx" 2)

(definset current-window window)

; fails if modified buffers exist
(defop quit wbx_exit (current-window) () boolean)

; really exit -- no matter what
(defop bx-exit w_Exit (nil) () boolean)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: bxinit

(module 'bxinit 1)

(defavar bx-argc bxinit_argc integer nil)

; set to nil if you don't want bx to parse args
(defavar bx-parse-args bxinit_parseArgs boolean t)

; (bx-argv i) => argv[i]
(defop bx-argv bxinit_getargv (integer) () string)

; set keymap for bind-lisp-to-key calls
(defop bx-use-keymap wbxinit_UseMap (string) () boolean)

; (b-bltk expr keys)  -- bind expr to keys
(defop bx-bind-lisp-to-key wbxinit_BindLispToKey (string string) ())

; (b-aft ext data-object)  -- file with extensions is this data object
(defop bx-add-file-type filetype_addentry (string string) ())

; (b-cf fn-name args) => result -- call fn with args & return its result
(defop bx-call-function wbxinit_CallFunc (string (string :default "")) () integer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
; File: bcmds

(module 'bcmds 4)

(definset current-window window)

(defavar bcmds-scratch-buffer "CurrentInset(window)->buffer->ScratchBuffer" "" character t)

(defop bcmds-add-template bcmds_template (current-window) ())
(defop bcmds-cd bcmds_cd (current-window) ())
(defop bcmds-delete-buffer bcmds_deletebuffer (current-window) ())
(defop bcmds-delete-window wbcmds_deletewindow (current-window) ())
(defop bcmds-execute-extended-command bcmds_cmdline (current-window) ())
(defop bcmds-exit bcmds_exit (current-window) ())
(defop bcmds-exit-recursive-edit bcmds_rexit (current-window) ())
(defop bcmds-find-function-pointer bcmds_findit (current-window) ())
(defop bcmds-illegal-operation bcmds_complain (current-window) ())
(defop bcmds-insert-file bcmds_insertfile (current-window) ())
(defop bcmds-list-buffers bcmds_listbufs (current-window) ())
(defop bcmds-new-window bcmds_NewWindow (current-window) ())
(defop bcmds-pwd bcmds_pwd (current-window) ())
(defop bcmds-read-file bcmds_readfile (current-window) ())
(defop bcmds-redraw-display bcmds_fullredraw (current-window) ())
(defop bcmds-save-alternate-form bcmds_savealternateform (current-window) ())
(defop bcmds-save-excursion bcmds_SEEdit (current-window) ())
(defop bcmds-save-excursion-visit bcmds_SEVisit (current-window) ())
(defop bcmds-shell bcmds_shell (current-window) ())
(defop bcmds-switch-to-buffer bcmds_switchtobuffer (current-window) ())
(defop bcmds-use-old-buffer bcmds_oldbuffer (current-window) ())
(defop bcmds-visit-file bcmds_visitfile (current-window) ())
(defop bcmds-write-current-file bcmds_savefile (current-window) ())
(defop bcmds-write-named-file bcmds_writefile (current-window) ())


; (w-at template-name) => success
(defop wbcmds-add-template wbcmds_template (current-window string) () boolean)

; (w-cd new-dir) => canonicalized-working-dir
(defop wbcmds-cd wbcmds_cd (current-window string) () string)

; (w-db buffer-name) => success
(defop wbcmds-delete-buffer wbcmds_deletebuffer (current-window string) () boolean)

; (w-if file-name) => success
(defop wbcmds-insert-file wbcmds_insertfile (current-window string ) () boolean)

; (w-pwd) => working-dir
(defop wbcmds-pwd wbcmds_pwd (current-window) () string)

; (w-rf file-name) => success
(defop wbcmds-read-file wbcmds_readfile (current-window string) () boolean)

; (w-s shell-buffer-name) => success
(defop wbcmds-shell bcmds_shell (current-window string) () boolean)

; (w-stb buffer-name) => success
(defop wbcmds-switch-to-buffer wbcmds_switchtobuffer (current-window string) () boolean)

; (w-uob buffer-name) => success
(defop wbcmds-use-old-buffer wbcmds_oldbuffer (current-window string) () boolean)
```

```
; (w-vf file-name) => success
(defop wbcmds-visit-file bcmds_visitwork (current-window string) () boolean)

; (w-wnf file-name) => success
(defop wbcmds-write-named-file wbcmds_writefile (current-window string) () boolean)


; cycle through buffers or bx windows
; next-buffer returns buffer -- its only use is to know when you've cycled
; thru all the buffers since all commands take a window
(deftype buffer)
(defop bcmds-next-buffer wbcmds_NextBuffer (current-window) () buffer)
(defop bcmds-goto-buffer window_SetBuffer (current-window buffer) ())
(defop bcmds-next-window wbcmds_NextWindow (current-window) () window)
(defop bcmds-goto-window im_GotoWindow (window) () "struct im_window *")

; get name of buffer and file in window
(defop bcmds-get-buffer-name wbcmds_GetBufferName ((window :default current-window)) () string)
(defop bcmds-get-file-name wbcmds_GetFileName ((window :default current-window)) () string)


; set the buffer to be unmodified since last write
(defop bcmds-set-buffer-unmodified wbcmds_SetBufferUnmodified (current-window) ())
; return true if buffer has been modified
(defop bcmds-get-buffer-modified wbcmds_GetBufferModified (current-window) () boolean)

; return a handle to the current window or buffer
(defop bcmds-get-current-window wbcmds_GetCurrentWindow (current-window) () window)
(defop bcmds-get-current-buffer wbcmds_GetCurrentBuffer (current-window) () buffer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: doc

(module 'doc 2)

(deftype doc)
(deftype mark)

; a mark's position and length are updated even as the document changes
; so that it still points to the same characters

(defop doc-create-mark doc_createmark (doc) () mark)
(defop doc-free-mark doc_freemark (mark) ())
(defop doc-set-mark-pos doc_setmarkpos (mark integer) ())
(defop doc-set-mark-length doc_setmarklength (mark integer) ())
(defop doc-mark-pos wdoc_getmarkpos (mark) () integer)
(defop doc-mark-length wdoc_getmarklength (mark) () integer)

; (d-rs doc pos len purge version get-template) -- convert to styles
;   purge -> delete old part of doc
;   version -> "1" or "2" (BE1/2 version)
;   get-template -> read Scribe.template
(defop doc-read-scribe readscr_Begin (doc integer integer boolean string boolean) () boolean "struct doc *")

; (d-is doc pos str size) -- raw insert-string
(defop doc-insert-string doc_insertstring (doc integer string integer) ())

; (d-d doc pos size) -- raw delete
(defop doc-delete doc_delete (doc integer integer) ())
```

```
; (d-rs doc pos size replace-string replace-size) -- replace size charas at
;  pos, preserving style info around original text
(defop doc-replace-string doc_replacestring (doc integer integer string integer) ())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: im

(module 'im 2)

(definset current-im im)

(defop im-start-macro im_startmacro (current-im) ())
(defop im-end-macro im_endmacro (current-im) ())
(defop im-do-macro im_domacro (current-im) ())


(deftype inset)

; unfortunately, current-inset is always the window inset for applications
; that use the window inset (including bx & delta)
(defparam current-inset "im_currentWindow->inputfocus" "struct im_window *" "struct inset *")
(defavar im-current-inset "im_currentWindow->inputfocus" "struct im_window *" inset nil)

; return the type of inset as a string
(defop im-inset-type im_InsetType ((inset :default current-inset)) () string)

; make the input focus the inset & activate its operations
; (if operations for inset x are defined in terms of current-x, current-x
; will only be found when x is the input focus)
(defop im-goto-inset im_GotoInset (inset) ())

; (i-gp inset inset-type) => inset
;  goto the parent of the inset that's the type (name) specified
;  the inset gone to is the first that accepts the input focus
(defop im-goto-parent im_GotoParent ((inset :default current-inset) (string :default "")) () inset)

; (i-wu inset inset) -- request inset to be updated
; 2 params should be same or use default (current)
; move to inset if we add an interface for it
(defop im-want-update inset_WantUpdate ((inset :default current-inset) (inset :default current-inset)) ())

; update everything now
(defop im-force-update im_ForceUpdate () ())

; wait for a character
(defop im-get-tty-char im_getc () () integer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: misc

(module 'misc 1)

(deftype inset)
(definset current-view view)

(defop misc-describe-key misc_DescribeKey (inset) ())
(defop misc-write-modified-files misc_SaveModifiedBuffers (current-view) ())

; (w-dk keymap-name keys) => function bound to keys in keymap
```

```
(defop wmisc-describe-key wmisc_DescribeKey (string string) () string)

; (w-wmf) => number of files NOT written, i.e. errors
(defop wmisc-write-modified-files wmisc_SaveModifiedBuffers (current-view) () integer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: msg

(module 'msg 1)

(definset current-im im)

(defop msg-display-string msg_DisplayString (current-im string) ())

; (m-afs prompt default) => (response success)
(defop msg-ask-for-string wmsg_AskForString (current-im string string) (string) boolean)

; if asking for string, abort the request
(defop msg-cancel-string msg_CancelString (current-im) ())

; if asking for string, return what's been typed so far
(defop msg-get-current-string wmsg_GetCurrentString (current-im) (string) integer)

; (m-is pos str len)
; if asking for string, insert string of given length at pos
(defop msg-insert-string msg_InsertString (current-im integer string integer) ())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: vcmds

(module 'vcmds 3)

(definset current-view view)

(defparam dot-pos "view_getdotpos(CurrentInset(view))" "" int)
(defparam current-doc "view_document(CurrentInset(view))" "" "struct doc *")

; we need to figure out how to handle numeric arguments

;;;
;;; basic commands
;;;

(defop vcmds-add-icon-style vcmds_addiconstyle (current-view) ())
(defop vcmds-add-inset-style vcmds_addinsetstyle (current-view) ())
(defop vcmds-argument-prefix vcmds_ctrlu (current-view) ())
(defop vcmds-backward-character vcmds_backward (current-view) ())
(defop vcmds-backward-paragraph vcmds_backwardpara (current-view) ())
(defop vcmds-backwards-rotate-killbuffer vcmds_backwardsrotatepaste (current-view) ())
(defop vcmds-beginning-of-buffer vcmds_topofbuffer (current-view) ())
(defop vcmds-beginning-of-line vcmds_leftedge (current-view) ())
(defop vcmds-beginning-of-paragraph vcmds_startofpara (current-view) ())
(defop vcmds-copy-to-killbuffer vcmds_copyregion (current-view) ())
(defop vcmds-current-indent vcmds_getspace (current-view (integer :default dot-pos)) ())
(defop vcmds-delete-next-character vcmds_delete (current-view) ())
(defop vcmds-delete-next-word vcmds_deleteword (current-view) ())
(defop vcmds-delete-previous-character vcmds_rubout (current-view) ())
(defop vcmds-delete-previous-word vcmds_ruboutword (current-view) ())
(defop vcmds-delete-to-killbuffer vcmds_zapregion (current-view) ())
```

```
(defop vcmds-digit vcmds_digit (current-view character) ())
(defop vcmds-end-of-buffer vcmds_endofbuffer (current-view) ())
(defop vcmds-end-of-line vcmds_endofline (current-view) ())
(defop vcmds-end-of-paragraph vcmds_endofpara (current-view) ())
(defop vcmds-exchange-dot-and-mark vcmds_exch (current-view) ())
(defop vcmds-fake-kill-command vcmds_fakezapregion (current-view) ())
(defop vcmds-forward-character vcmds_forward (current-view) ())
(defop vcmds-forward-word vcmds_forwardword (current-view) ())
(defop vcmds-goto-paragraph vcmds_gotoparagraph (current-view) ())
(defop vcmds-indent-paragraph vcmds_indentcommand (current-view) ())
(defop vcmds-insert-character vcmds_selfinsert (current-view character) ())
(defop vcmds-kill-line vcmds_killline (current-view) ())
(defop vcmds-kill-white-space vcmds_killwhitespace (current-view) ())
(defop vcmds-line-to-top-of-window vcmds_metabang (current-view) ())
(defop vcmds-make-plainer vcmds_plainer (current-view) ())
(defop vcmds-move-to-column vcmds_gettocol (current-view integer) ())
(defop vcmds-newline vcmds_insertnl (current-view) ())
(defop vcmds-newline-and-indent vcmds_mylf (current-view) ())
(defop vcmds-next-line vcmds_nextline (current-view) ())
(defop vcmds-next-page vcmds_nextscreen (current-view) ())
(defop vcmds-open-line vcmds_ctrlo (current-view) ())
(defop vcmds-preview vcmds_preview (current-view) ())
(defop vcmds-previous-line vcmds_previousline (current-view) ())
(defop vcmds-previous-page vcmds_prevscreen (current-view) ())
(defop vcmds-previous-word vcmds_backword (current-view) ())
(defop vcmds-print vcmds_print (current-view) ())
(defop vcmds-query-replace-string vcmds_qreplace (current-view) ())
(defop vcmds-quote-character vcmds_quote (current-view) ())
(defop vcmds-rotate-killbuffer vcmds_dorotatepaste (current-view (integer :default 1)) ())
(defop vcmds-scroll-one-line-down vcmds_glitchdown (current-view) ())
(defop vcmds-scroll-one-line-up vcmds_glitchup (current-view) ())
(defop vcmds-search-forward vcmds_search (current-view) () integer)
(defop vcmds-search-reverse vcmds_rsearch (current-view) () integer)
(defop vcmds-select-region vcmds_selectregion (current-view) ())
(defop vcmds-set-mark vcmds_ctrlat (current-view) ())
(defop vcmds-set-printer vcmds_setprinter (current-view) ())
(defop vcmds-show-current-paragraph vcmds_whatparagraph (current-view) ())
(defop vcmds-show-styles vcmds_showstyles (current-view) ())
(defop vcmds-transpose-characters vcmds_twiddle (current-view) ())
(defop vcmds-unindent-paragraph vcmds_unindentcommand (current-view) ())
(defop vcmds-yank-from-killbuffer vcmds_yank (current-view) ())


;;;
;;; non-interactive versions of commands that use message line
;;;

; (w-ais inset-name) => t if successful
(defop wvcmds-add-inset-style wvcmds_addinsetstyle (current-view string) () boolean)

; actually goto-line for files with newlines
(defop wvcmds-goto-paragraph wvcmds_gotoparagraph (current-view integer) ())

; (w-rs old new) => pattern-ok (0 is ok, -1 is bad)
(defop wvcmds-replace-string wvcmds_replace (current-view string string) () integer)

; no string => use last one
(defop wvcmds-search-forward wvcmds_search (current-view (string :default "")) () integer)
(defop wvcmds-search-reverse wvcmds_rsearch (current-view (string :default "")) () integer)

; set the printer and return it (different on failure)
```

```
(defop wvcmds-set-printer wvcmds_setprinter (current-view string) () string)

; actually show-line-number for files with newlines
(defop wvcmds-show-current-paragraph wvcmds_whatparagraph (current-view) () integer)

; describe the styles of the selection
(defop wvcmds-show-styles wvcmds_showstyles (current-view) () string)


;;;
;;; commands that have no character equivalent
;;;

(defop vcmds-insert-string wvcmds_insertstring (current-view string) ())

; (w-rts pos) => (string len)
; will return up to 200 characters -- if len is 200, call again with
; updated pos to get more of string
(defop wvcmds-region-to-string wvcmds_regiontostring  (current-view (integer :default 0)) (string) integer)


;;;
;;; inset commands
;;;

; a vinset is an inset within a view while an "inset" is a real inset that
; can be passed to ops defined by im & others.  A vinset is actually a mark
; that provides quick access to the inset in the view.  It would be better
; to keep the marks internal so that language dealt only with ordinary
; insets (easier to pass to other inset routines) and so that each call
; that returned an inset didn't allocate a new mark.
(deftype inset)
(deftype vinset)

; (v-cv data-object-type) => inset-handle  -- yes, it's inconsistent to use
; a data type, but that's what we've got
(defop vcmds-create-vinset wvcmds_CreateVInset (current-view string) () vinset)

; (v-dv vinset) => t if successful
(defop vcmds-delete-vinset wvcmds_DeleteVInset (current-view vinset) () boolean)

; (v-fv vinset) -- free storage associated with vinset
(defop vcmds-free-vinset doc_freemark (vinset) ())

; (v-gv vinset) => t if successful -- sets input focus
(defop vcmds-goto-vinset wvcmds_GotoVInset (current-view vinset) () boolean)

; (v-fv [inset-type] [pos]) => vinset --
;   get first inset of type (or any inset if type == "") starting at
;   beginning of doc or at pos
(defop vcmds-first-vinset wvcmds_GotoPosOfInset
   (current-view (string :default "") (integer :default 0) nil) () vinset)

; (v-nv [inset-type] [pos]) => vinset --
;   get first inset of type (or any inset) after pos (default is dot)
(defop vcmds-next-vinset wvcmds_GotoPosOfInset
   (current-view (string :default "") (integer :default dot-pos) t) () vinset)

; go from view's inset to im's inset & back.  While vti is a struct
; reference, itv must search the doc from the beginning to find the
; corresponding inset.
```

```
(defop vcmds-inset-to-vinset wvcmds_InsetToVInset (current-view inset) () vinset)
(defop vcmds-vinset-to-inset wvcmds_VInsetToInset (vinset) () inset)

; (v-vt vinset) => inset-type -- return the inset's type (string name)
(defop vcmds-vinset-type wvcmds_VInsetType (vinset) () string)

; this isn't in inset or im since in general it can cause trouble
; (v-svs vinset width height) -- set width & height of inset
(defop vcmds-set-vinset-size wvcmds_SetVInsetSize (current-view vinset integer integer) () boolean)

;;;
;;; style commands
;;;

(deftype style "struct stylesheet *")

; (v-al style) => t if successful -- applies style to selection
(defop vcmds-apply-look wvcmds_ApplyLook (current-view style) () boolean)

; (v-gse [pos]) => (start length) -- find smallest enclosing environment
(defop vcmds-get-style-extent wvcmds_GetStyleExtent
   (current-view (integer :default dot-pos)) (integer integer))

; both of these lookups are case-sensitive
; (v-fsbmn menu-name) => style -- lookup by menu item (e.g. Font,Italic)
(defop vcmds-find-style-by-menu-name doc_findstylesheet (current-doc string) () style)
; (v-fsbn name) => style -- lookup by real name (e.g. italic)
(defop vcmds-find-style-by-name doc_identifystylesheet (current-doc string) () style)

; (v-gsn style) => real-name -- inverse of find-style-by-name
(defop vcmds-get-style-name wvcmds_GetStyleName (style) () string)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: view

(module 'view 4)

(definset current-view view)
(deftype doc)

(defparam current-doc "view_document(CurrentInset(view))" "" "struct doc *")
(defparam dot-pos "view_getdotpos(CurrentInset(view))" "" int)
(defparam dot-length "view_getdotlength(CurrentInset(view))" "" int)

(defavar view-dot-pos "view_getdotpos(CurrentInset(view))" "" integer nil)
(defavar view-dot-length "view_getdotlength(CurrentInset(view))" "" integer nil)
(defavar view-doc "view_document(CurrentInset(view))" "" doc nil)
(defavar view-doc-length "view_document(CurrentInset(view))->tsize" "" integer nil)

; set dot to end of selection
(defop view-collapse-dot view_collapsedot (current-view) ())

; (v-gcp newpos click-count mouse-action left-start-pos right-start-pos) =>
;   (new-left-pos new-right-pos)
(defop view-get-click-position view_GetClickPosition
   (current-view integer integer integer integer integer) (integer integer))

; return position of first char displayed
(defop view-get-top-pos view_getframe (current-view) () integer)
```

```
; is position visible?
(defop view-is-visible view_visible (current-view integer) () boolean)

; what char is at screen x,y (pixels)?
(defop view-locate view_locate (current-view integer integer) () integer)

; (v-mb pos units unit-type) => (dist-moved lines-added new-pos)
; unit-type = 0 move-by-lines or 1 move-by-points
; find the position back n units from pos
(defop view-move-back view_movebackn (current-view integer integer integer) (integer integer) integer)

; (v-mf pos lines) => new-pos
; find the position forward n lines past pos
(defop view-move-forward view_moveforwardn (current-view integer integer) () integer)

; set the cursor or beginning of the selection
(defop view-set-dot-pos view_setdotpos (current-view integer) ())

; set the length of the selection
(defop view-set-dot-length view_setdotlength (current-view integer) ())

; set the first char to see (& update)
(defop view-set-top-pos view_setframe (current-view integer) ())

; returns -1 if pos beyond document (doc-char-at takes a doc as param)
(defop view-char-at doc_charat (current-doc (integer :default dot-pos)) () integer)

; create a mark in current doc (doc-create-mark takes a doc as param)
(deftype mark)
(defop view-create-mark doc_createmark (current-doc) () mark)
(defop view-set-mark-pos doc_setmarkpos (mark (integer :default dot-pos)) ())
(defop view-set-mark-length doc_setmarklength (mark (integer :default dot-length)) ())

; (v-slk keymap-name) -- install a local keymap
;   if it doesn't exist, create it
(defop view-set-local-keymap view_SetLocalKeymap (current-view string) ())

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; File: window

(module 'window 2)

(deftype inset)
(definset current-window window)

(defparam top-level "CurrentInset(window)->lhandle" "" "struct inset *")
(defparam current-inset  "CurrentInset(window)->lselection" "" "struct inset *")

(defavar window-top-level "CurrentInset(window)->lhandle" "" inset nil)
(defavar window-current-inset "CurrentInset(window)->lselection" "" inset nil)

; (w-gp inset inset-type) => inset
;   goto the parent of the inset that's the type (name) specified
;   the inset gone to is the first that accepts the input focus
(defop window-goto-parent im_GotoParent ((inset :default current-inset) (string :default "")) () inset)

; (w-gtl) -- go to the top level inset in the window
(defop window-goto-top-level inset_WantInputFocus (top-level top-level) ())
```

Appendix D
The Calculator Program

This appendix includes most of the calculator program as implemented in C.

D.1 WHISPER INTERFACE CODE

This section consists of the code produced by wgen given the specifications for calculator (the main program) and calc (the inset) as shown in Appendix C.1.

The file below, calculator.wh, is included in calmain.c. It exports a single Whisper operation, quit. It also includes a function, calculator_WhisperInit, that when called by the main program initializes the Whisper run-time system.

```
/* This file was automatically generated by wgen. */

#include "wclient.h"


static WVarDescriptor vars[] = {
    0
};


#define _cbool0 ((WVar) (cbools+0))
#define _cbool1 ((WVar) (cbools+1))

static WBoolean cbools[] = { FALSE, TRUE, };


#define __cbool0 ((WParamDesc) (dparams+0))
#define __cbool1 ((WParamDesc) (dparams+1))


static WDParamDescriptor dparams[] = {
    { _cbool0, sizeof (WBoolean) },
    { _cbool1, sizeof (WBoolean) },
};


static WParam _quitins[] = {
    { DIRPARAM, __cbool0 },
      LASTPARAM };
```

```
static WType _quitouts[] = { LASTPARAM, BOOLEAN };
extern WBoolean w_Exit();


static WCallDescriptor ops[] = {
   { "quit", (WProc) w_Exit, _quitins, _quitouts },
   0
};

static WInterface calculatorInterface = { "calculator", 1, ops, vars };


int calculator_whisperInitialized = 0;

int calculator_WhisperInit (serverOnly)
   int serverOnly;                              /* If true, there is no user interface. */
{
   int portal = 0;

   if (calculator_whisperInitialized)  return 0;
   calculator_whisperInitialized = 1;
   IOMGR_Initialize();                          /* Initialize BX and LWP */
   portal = w_GetPortal ("calc");               /* Find RPC address for calc. */
   if (serverOnly && portal == 0)
      return -1;
   if (w_Init (&calculatorInterface, portal) != 0)   /* Initialize Whisper and the calculator interface. */
      return -1;
   return 0;
}
```

This file, calc.wh, is included in calc.c. It defines the interface to the calc inset.

```
/* This file was automatically generated by wgen. */

#include "wclient.h"

#include "wbe.h"
w_DefineModuleObjectName (calc);

static WVar GetIndirectValue();

#define _calc 0

#define _display (-1)
#define _register (-2)


static WVarDescriptor vars[] = {
   { "display", (WVar) _display, DOUBLE, 0, GetIndirectValue },
   { "register", (WVar) _register, DOUBLE, 0, GetIndirectValue },
   0
};



#define _current_calc   ((WParamDesc) (iparams+0))


static WIParamDescriptor iparams[] = {
   { GetIndirectValue, 0, sizeof (struct calc *) },
};
```

```
static WParam _calc_enterins[] = {
   { INDPARAM, _current_calc },
   { STDPARAM, DOUBLE },
     LASTPARAM };
static WType _calc_enterouts[] = { LASTPARAM, DOUBLE };
extern double Enter();

static WParam _calc_digitins[] = {
   { INDPARAM, _current_calc },
   { STDPARAM, CHAR },
     LASTPARAM };
static WType _calc_digitouts[] = { LASTPARAM, DOUBLE };
extern double KeyDigit();

static WParam _calc_pointins[] = {
   { INDPARAM, _current_calc },
     LASTPARAM };
static WType _calc_pointouts[] = { LASTPARAM, DOUBLE };
extern double KeyPoint();

static WParam _calc_functionins[] = {
   { INDPARAM, _current_calc },
   { STDPARAM, CHAR },
     LASTPARAM };
static WType _calc_functionouts[] = { LASTPARAM, DOUBLE };
extern double KeyFunction();

static WParam _calc_clear_entryins[] = {
   { INDPARAM, _current_calc },
     LASTPARAM };
static WType _calc_clear_entryouts[] = { LASTPARAM, DOUBLE };
extern double KeyClearEntry();

static WParam _calc_clearins[] = {
   { INDPARAM, _current_calc },
     LASTPARAM };
static WType _calc_clearouts[] = { LASTPARAM, DOUBLE };
extern double KeyClear();

static WParam _calc_negateins[] = {
   { INDPARAM, _current_calc },
     LASTPARAM };
static WType _calc_negateouts[] = { LASTPARAM, DOUBLE };
extern double KeyNegate();

static WParam _calc_equalsins[] = {
   { INDPARAM, _current_calc },
     LASTPARAM };
static WType _calc_equalsouts[] = { LASTPARAM, DOUBLE };
extern double KeyEqual();


static WCallDescriptor ops[] = {
   { "calc-enter", (WProc) Enter, _calc_enterins, _calc_enterouts },
   { "calc-digit", (WProc) KeyDigit, _calc_digitins, _calc_digitouts },
   { "calc-point", (WProc) KeyPoint, _calc_pointins, _calc_pointouts },
   { "calc-function", (WProc) KeyFunction, _calc_functionins, _calc_functionouts },
   { "calc-clear-entry", (WProc) KeyClearEntry, _calc_clear_entryins, _calc_clear_entryouts },
   { "calc-clear", (WProc) KeyClear, _calc_clearins, _calc_clearouts },
   { "calc-negate", (WProc) KeyNegate, _calc_negateins, _calc_negateouts },
   { "calc-equals", (WProc) KeyEqual, _calc_equalsins, _calc_equalsouts },
```

```
    0
};


static WInterface calcInterface = { "calc", 2, ops, vars };


static WVar GetIndirectValue (param, size, block)
    int param, size;
    int **block;
/* Return value of interface variables. */
{
    int *offset;
    int rc = 0;
    struct inset *r;  /* used by CurrentInset */
    if (block) {
        offset = w_PARAMOFFSET (*block, size);
        *block += w_BytesToWords (size);
    }
    switch (param) {
        case _display:  return (WVar) &(CurrentInset(calc))->display;
        case _register:  return (WVar) &(CurrentInset(calc))->registr;
        case 0:  {  /* current-calc */
            struct calc **o = (struct calc **) offset;
            *o = CurrentInset(calc);
            break;
        }
        default:  rc = -1;
    }
    return (WVar) rc;
}


char *calc_GetName()
{
    return w_ModuleObjectName (calc);
}


int calc_whisperInitialized = 0;

int calc_WhisperInit ()
{
    if (calc_whisperInitialized)  return 0;
    calc_whisperInitialized = 1;
    if (w_AddToInterface (&calcInterface) != 0)          /* Add calc interface to the calculator interface. */
        return -1;
    return 0;
}
```

D.2 THE MAIN PROGAM

The calculator main program is responsible for creating a window and placing a calc inset in the window. It then turns control over to BE2. Note that the calculator was originally implemented without a Whisper interface. All changes needed to incorporate Whisper are bracketed by #ifdef WHISPER and #endif WHISPER.

```
#include "wmclient.h"
#include "basic.h"

#include "CamphorImport.h"
#include "bx/data.h"
#include "bx/inset.h"
#include "bx/im.h"
#include "bx/keymap.h"
#include "bx/lpair.h"
#include "palette.h"
#include "label.h"
#include "calc.h"

#ifdef WHISPER
#include "calculator.wh"
#endif WHISPER

program (Calculator)

main (argc, argv)
   int argc;
   char **argv;
{
   boolean doFork = TRUE;
   int i;
   struct inset *top;
   struct wm_window *w;

   /* Process arguments. */
   for (i = 1; i < argc; i++) {
      if (strcmp(argv[i], "-x") == 0) {
         doFork = FALSE;
#ifdef WHISPER
         w_debug = TRUE;
#endif WHISPER
         continue;
      } /* if */
      printf("Usage: calc [-x]\n");
      exit(-1);
   }

/* must come before any other interfaces */
#ifdef WHISPER
   calculator_WhisperInit(FALSE);                    /* Initialize Whisper and the calculator interface. */
#endif WHISPER

   /* Statically load everything. */
   camphorinit (0, 0, 0, "/usr/andrew/lib/bzlib");
   staticload ("inset", insetCamphorInitializer());
   staticload ("im", imCamphorInitializer());
   staticload ("keymap", keymapCamphorInitializer());
   staticload ("lpair", lpairCamphorInitializer());
```

```
    staticload ("palette", paletteCamphorInitializer());
    staticload ("label", labelCamphorInitializer());
    staticload ("calc", calcCamphorInitializer());

    if (doFork && fork())
        exit (0);

    w = im_CreateWindow(0);                    /* Create a wm window. */
    wm_SetDimensions (140, 140, 175,175);      /* State desired size. */
    top = inset_New ("calc", NULL);            /* Create the top-level inset. */
    im_FillWindow (w, top);                    /* Associate the inset with the window. */
    inset_WantInputFocus (top, top);           /* Send all input to the inset. */
#ifdef WHISPER
    w_ReadyToInteract();                       /* Tell XLisp that we're ready for requests. */
#endif WHISPER
    im_KeyboardProcessor();                    /* Enter BX main loop. */
}
```

D.3 THE CALC INSET

The calc inset provides most of the functionality of the calculator application. Only the functions that call Whisper routines and a few example routines are included here. Note that the calculator was originally implemented without a Whisper interface. All changes needed to incorporate Whisper are bracketed by #ifdef WHISPER and #endif WHISPER.

```
#include <stdio.h>
#include <signal.h>
#include <setjmp.h>
#include "wmclient.h"
#include "basic.h"
typedef char boolean;  /* to be consistent with beglobals.h */

#include "CamphorImport.h"
#include "bx/data.h"
#include "bx/inset.h"
#include "bx/im.h"
#include "bx/keymap.h"
#include "bx/lpair.h"
#include "palette.h"
#include "label.h"

#include "CamphorExport.h"
#include "calc.h"

#ifdef WHISPER
#include "calc.wh"
#endif WHISPER

...

double KeyDigit(), KeyPoint(), KeyFunction(), KeyEqual();
double KeyNegate(), KeyClearEntry(), KeyClear(), KeyNoop();
Quit();

/* Mapping of keys to functions. */
static struct {
    char  key;
```

```
    DProc keyProc;
} buttons[] = {
    '7', KeyDigit,
    '4', KeyDigit,
    '1', KeyDigit,
    '0', KeyDigit,
    '8', KeyDigit,
    '5', KeyDigit,
    '2', KeyDigit,
    '.', KeyPoint,
    '9', KeyDigit,
    '6', KeyDigit,
    '3', KeyDigit,
    '\1', KeyNoop,
    '/', KeyFunction,
    '*', KeyFunction,
    '-', KeyFunction,
    '+', KeyFunction,
    'C', KeyClear,
    'c', KeyClearEntry,
    'n', KeyNegate,
    '=', KeyEqual,
    0, 0
};

struct inset *calc_New()
/* Create and initialize a new calc inset. */
{
    extern char *malloc();
    struct calc *self;

    self = (struct calc *) malloc (sizeof(struct calc));
    calc_Init(self);
    return (struct inset *) self;
}

calc_Init(self)
    struct calc *self;
/* Initialize a calc inset. */
{
    int i;
#ifdef WHISPER
    inset_InitStructure(w_ModuleObjectName(calc), self);          /* Initialize inset using calc atom. */
#else
    inset_InitStructure("calc", self);
#endif WHISPER
    if (! calc_initialized) {
        int CatchFPE();
        calc_initialized = TRUE;
        calcFont = wm_DefineFont (calcFontNAME);
        calcMap = keymap_create();
        for (i = 0; buttons[i].key != 0; i++)
            if (buttons[i].keyProc != NULL)
                keymap_insertproc (calcMap, buttons[i].key,
                  buttons[i].keyProc);
        keymap_insertproc (calcMap, '\003', Quit);
        signal (SIGFPE, CatchFPE);                                /* Handle floating point errors. */
#ifdef WHISPER
        calc_WhisperInit();                                       /* Initialize the calc interface. */
#endif WHISPER
    }
```

```
    self->inDisplay = inset_New ("label", NULL, NULL, NULL);          /* Create display text. */
    self->pal = inset_New ("palette", NULL, self, calcFont, 0);       /* Create array of keys. */
    self->lp = inset_New ("lpair", self, self->pal, self->inDisplay, -90);  /* Put label and palette together. */
    for (i = 0; buttons[i].key != 0; i++)                             /* Install keys in palette. */
        palette_AddIcon (self->pal, buttons[i].key, NULL);
    self->ks = keymap_newstate (calcMap);
    self->ml = im_NewML();                                            /* Create menu. */
    im_AddToML (self->ml, Quit, "Quit", self, 0);
    self->registr = 0.0;
    self->display = 0.0;
    self->fraction = 0.0;
    self->inNumber = FALSE;
    self->op = NOOP;
}

...

calc_FullUpdate(self, how, r)
    struct calc *self;
    int how;
    struct rect r;
/* Redraw calc inset. */
{
    ...
}

calc_Update(self)
    struct calc *self;
/* Update inset—just redraw display text. */
{
    SetDisplayString (self);
    inset_Update (self->inDisplay);
}

Inset calc_Hit(self, action, x, y)
    struct calc *self;
    int x;
    int y;
    int action;
/* Process mouse button (down transition). */
{
    inset_WantInputFocus (self, self);
    return inset_Hit (self->lp, action, x, y);
}

int calc_KeyIn(self, ch)
    struct calc *self;
    int ch;
/* Process input key. */
{
    return keymap_char (self->ks, self, ch);
}

...

/*****************************************************************/

static double KeyDigit (self, ch)
    struct calc *self;
    char ch;
/* Process a digit key (0, ..., 9). */
```

```
{
    Digit (self, ch - '0');
    inset_WantUpdate (self, self);
    return self->display;
}

static double KeyFunction (self, ch)
    struct calc *self;
    char ch;
/* Process an arithmetic function. */
{
    double Sum(), Difference(), Product(), Quotient();
    switch (ch) {
        case '+': Function (self, Sum); break;
        case '-': Function (self, Difference); break;
        case '*': Function (self, Product); break;
        case '/': Function (self, Quotient); break;
    }
    inset_WantUpdate (self, self);
    return self->display;
}

...

/*****************************************************************/

static Digit (self, d)
    struct calc *self;
    int d;
/* Concatenate a digit onto the current display. */
{
    if (! INERROR (self)) {
        if (! self->inNumber) {
            self->inNumber = TRUE;
            self->registr = self->display;
            self->display = self->fraction = 0.0;
        }
        if (self->fraction == 0.0)
            self->display = 10.0 * self->display +
              (double) ((self->display >= 0.0) ? (d) : (-d));
        else {
            self->display += self->fraction *
              (double) ((self->display >= 0.0) ? (d) : (-d));
            self->fraction *= 0.1;
        }
    }
}

static double Sum(self)
    struct calc *self;
/* Do an addition. */
{
    return self->registr + self->display;
}

...

static Function(self, fProc)
    struct calc *self;
    DProc fProc;
/* Process an arithmetic function.  The operation is just stored until =
```

```
  or another operation is invoked. */
{
   if (INERROR (self)) return;
   if (self->inNumber) Eql(self);
   self->op = fProc;
}

static Eql(self)
   struct calc *self;
/* Process = or first half of arithmetic function.  Do the pending operation. */
{
   if (setjmp (fpeJmp))
      SETERROR (self);
   if (INERROR (self)) return;
   self->inNumber = FALSE;
   if (self->op!= NOOP) {
      self->display = self->op(self);
      self->op = NOOP;
   }
}

static Clear (self)
   struct calc *self;
/* Clear the calculator. */
{
   self->registr = self->display = 0.0;
   self->inNumber = FALSE;
   self->op = NOOP;
}

...
```

References

[Adobe 85] Adobe Systems, Inc.
PostScript Language Reference Manual.
Addison-Wesley, 1985.

[Aho 87] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger.
The Awk Programming Language.
Prentice-Hall, 1987.

[Bentley 86a] Jon Bentley.
Programming pearls: Little languages.
Communications of the ACM 29(8):711-721, August 1986.

[Bentley 86b] Jon L. Bentley and Brian W. Kernighan.
Grap—A language for typesetting graphs.
Communications of the ACM 29(8):782-792, August 1986.

[Betts 87] Bill Betts, David Burlingame, Gerhard Fischer, Jim Foley, Mark Green, David Kasik, Stephen T. Kerr, Dan Olsen, and James Thomas.
Goals and objectives for user interface software.
Computer Graphics 21(2):73-78, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Betz 86] David M. Betz.
Xlisp: An experimental object-oriented language.
1986.

[Birrell 84] Andrew D. Birrell and Bruce J. Nelson.
Implementing remote procedure calls.
ACM Transactions on Computer Systems 2(1):39-59, February 1984.

[Borenstein 85] Nathaniel Borenstein.
The BAGS message management system user's manual.
Carnegie Mellon University, Computer Science Department, 1985.

[Borenstein 88a] Nathaniel S. Borenstein and James Gosling.
Emacs: A retrospective (lessons for flexible system design).
October 1988.

[Borenstein 88b] Nathaniel Borenstein, Craig Everhart, Jonathan Rosenberg, and Adam Stoller.
A multimedia message system for Andrew.
In *USENIX Technical Conference Proceedings.* Dallas, TX, February 1988.

[Bourne 78] S. R. Bourne.
The UNIX shell.
Bell System Technical Journal 57(6):1971-1990, July-August 1978.

[Card 83] Stuart Card, Thomas P. Moran, and Allen Newell.
The Psychology of Human-Computer Interaction.
Lawrence Erlbaum Associates, Hillsdale, NJ, 1983.

[Chang 87] Shi-Kuo Chang.
Visual languages: A tutorial and survey.
IEEE Software 4(1):29-39, January 1987.

[Cockton 87] Gilbert Cockton.
A new model for separable interactive systems.
Human-Computer Interaction—INTERACT '87.
Elsevier Science Publishers B. V. (North-Holland), 1987, pages 1033-1038.

[Cohn 87] Richard J. Cohn.
Automated testing of interactive programs.
January 1987.

[Cowlishaw 84] Michael F. Cowlishaw.
The design of the REXX language.
IBM Systems Journal 23(4):326-335, 1984.

[Dance 87] John R. Dance, Tamar E. Granor, Ralph D. Hill, Scott E. Hudson, Jon Meads, Brad A. Myers, and Andrew Schulert.
The run-time structure of UIMS-supported applications.
Computer Graphics 21(2):97-101, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Dolotta 80] T. A. Dolotta and J. R. Mashey.
Using a command language as the primary programming tool.
In *Command Language Directions*, pages 35-49. Edited by David Beech.
North-Holland, 1980.
Proceedings of the IFIP TC 2.7 Working Conference on Command Languages (1979).

[DOS 86] *Disk Operating System version 3.20 reference.*
International Business Machines Corporation, 1986.

[Ellis 80] John R. Ellis.
A Lisp shell.
SIGPLAN Notices 15(5):24-34, May 1980.

[Ellis 87] John R. Ellis.
Private communication.
May 1987.

[Ewing 86] Juanita J. Ewing.
An object-oriented operating system interface.
In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, pages 46-53. September 1986.

[Fitzgerald 86] Robert Fitzgerald and Richard Rashid.
The integration of virtual memory management and interprocess communcation in Accent.
ACM Transactions on Computer Systems 4(2):147-177, May 1986.

[Foley 82] James D. Foley and Andries van Dam.
Fundamentals of Interactive Computer Graphics.
Addison-Wesley, 1982.

[Gates 87] Bill Gates.
Beyond macro processing.
Byte: Applications Software Today 12(7):11-16, Summer 1987.

[GDDM 83] *GDDM general information.*
International Business Machines Corporation, 1983.
GC33-0100.

[Gill 86] Daniel P. Gill.
A proposal for interwindow communication and translation facilities.
In *USENIX Technical Conference Proceedings*, pages 79-88. Denver, CO, January 1986.

[Giuse 85] Dario Giuse.
Programming the Lisp shell.
Carnegie Mellon University, Computer Science Department, 1985.

[Goldberg 83] A. Goldberg and D. Robson.
Smalltalk-80: The Language and its Implementation.
Addison-Wesley, 1983.

[Goldberg 84] Adele Goldberg.
Smalltalk-80: The Interactive Programming Environment.
Addison-Wesley, 1984.

[Goodman 87] Danny Goodman.
The Complete HyperCard Handbook.
Bantam Books, 1987.

[Gosling 82] James Gosling.
UNIX Emacs.
Carnegie Mellon University, Computer Science Department, 1982.

[Gosling 86a] James Gosling and David S. H. Rosenthal.
A window manager for bitmapped displays and UNIX.
Methodology of Window Management.
Springer-Verlag, 1986, pages 115-128.
Proceedings of an Alvey Workshop at Cosener's House, Abingdon, U. K., April 1985.

[Gosling 86b] James Gosling.
SUNDEW: A distributed and extensible window system.
Methodology of Window Management.
Springer-Verlag, 1986.
Proceedings of an Alvey Workshop at Cosener's House, Abingdon, U. K., April 1985.

[Green 85] Mark Green.
The University of Alberta user interface management system.
Computer Graphics 19(3):205-213, July 1985.

[Halbert 84] Daniel C. Halbert.
Programming by Example.
PhD thesis, University of California, Berkeley, December 1984.

[Halbert 85] Daniel C. Halbert.
Private communication.
October 1985.

[Hanson 84] Stephen José Hanson, Robert E. Kraut, and James M. Farber.
Interface design and multivariate analysis of UNIX command use.
ACM Transactions on Office Information Systems 2(1):42-57, January 1984.

[Hergert 86] Douglas Hergert.
Microsoft Excel: The Microsoft Desktop Dictionary and Cross-Reference Guide.
Microsoft Press, Redmond, WA, 1986.

[Hill 87a] Ralph D. Hill.
Supporting Concurrency, Communication and Synchronization in Human-Computer Interaction.
PhD thesis, University of Toronto, 1987.

[Hill 87b] Ralph D. Hill.
Some important features and issues in user interface management systems.
Computer Graphics 21(2):116-120, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Horn 87] Bruce Horn.
Private communication.
October 1987.

[Hudson 87] Scott E. Hudson.
UIMS support for direct manipulation interfaces.
Computer Graphics 21(2):120-124, April 1987.
ACM SIGGRAPH Workshop on Software Tools for User Interface Management.

[Hutchins 86] E. L. Hutchins, J. D. Hollan, and D. A. Norman.
Direct manipulation interfaces.
In *User Centered System Design*, pages 87-124. Edited by D. A. Norman and S. W. Draper. Lawrence Erlbaum Associates, Hillsdale, NJ, 1986.

[Jones 86] Michael B. Jones and Richard F. Rashid.
Mach and Matchmaker: Kernel and language support for object-oriented distributed systems.
In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '86)*, pages 67-77. September 1986.

[Joy 79] William N. Joy.
An introduction to the C shell.
University of California, Berkeley, 1979.

[Kazar 87] Michael Kazar and David Nichols.
The R package.
Carnegie Mellon University, Information Technology Center, 1987.
On-line documentation.

[Keedy 85] J. L. Keedy and J. V. Thomson.
Command interpretation and invocation in an information-hiding system.
In *The Future of Command Languages: Foundations for Human-Computer Communication.* IFIP WG 2.7 Working Conference, Rome, September 1985.

[Kernighan 82] Brian W. Kernighan.
PIC—A language for typesetting graphics.
Software—Practice and Experience 12(1):1-20, January 1982.

[Kernighan 84] Brian W. Kernighan and Rob Pike.
The UNIX Programming Environment.
Prentice-Hall, 1984.

[Kraut 83] Robert E. Kraut, Stephen J. Hanson, and James M. Farber.
Command usage and interface design.
In *Proceedings of the CHI '83 Conference on Human Factors in Computing Systems*, pages 120-124. Edited by Ann Janda. December 1983.

[Laff 85] M. R. Laff and B. Hailpern.
SW2—An object-based programming environment.
In *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 1-11. Seattle, WA, June 1985.

[Lantz 85] Keith A. Lantz, William I. Nowicki, and Marvin M. Theimer.
An empirical study of distributed application performance.
IEEE Transactions on Software Engineering 11(10):1162-1174, October 1985.

[MacLachlan 84] Rob MacLachlan and Skef Wholey.
Hemlock command implementor's manual.
Carnegie Mellon University, Computer Science Department, 1984.
Spice document S177.

[Miller 86] David W. Miller.
The Great American History Machine.
In *Proceedings of the 1986 IBM Academic Information Systems University AEP Conference*, pages 97-107. Milford, CT, 1986.

[Morris 86] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith.
Andrew: A distributed personal computer environment.
Communications of the ACM 29(3):184-201, March 1986.

[Nelson 81] Bruce J. Nelson.
Remote Procedure Call.
PhD thesis, Carnegie Mellon University, May 1981.

[Nelson 85] Greg Nelson.
Juno, a constraint-based graphics system.
Computer Graphics 19(3):235-243, July 1985.

[NeWS 86] *NeWS preliminary technical overview.*
Sun Microsystems, Inc., Mountain View, CA, 1986.

[Nix 85] Robert P. Nix.
Editing by example.
ACM Transactions on Programming Languages and Systems 7(4):600-621, October 1985.

[Notkin 87] David Notkin and William G. Griswold.
Enhancement through extension: The Extension Interpreter.
SIGPLAN Notices 22(7):45-55, July 1987.
Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques.

[Olsen 83] Dan R. Olsen Jr. and Elizabeth P. Dempsey.
SYNGRAPH: A graphical user interface generator.
Computer Graphics 17(3):43-50, July 1983.

[Olsen 84] Dan R. Olsen Jr., William Buxton, Roger Ehrich, David J. Kasik, James R. Rhyne, and John Sibert.
A context for user interface management.
IEEE Computer Graphics and Applications 4(12):33-42, December 1984.

[Olsen 86] Dan R. Olsen Jr.
An editing model for generating graphical user interfaces.
In *Graphics Interface '86 Proceedings*, pages 66-70. 1986.

[Palay 88] Andrew J. Palay, Wilfred J. Hansen, Michael L. Kazar, Mark S. Sherman, Maria G. Wadlow, Thomas P. Neuendorffer, Zalman Stern, Miles Bader, and Thom Peters.
The Andrew toolkit—An overview.
In *USENIX Technical Conference Proceedings*. Dallas, TX, February 1988.

[PCL 82] *Tops-20 Programmable Command Language user's guide and reference manual.*
Carnegie Mellon University Computation Center, 1982.

[Petschenik 85] Nathan H. Petschenik.
Practical priorities in system testing.
IEEE Software 2(5):18-23, September 1985.

[Pfaff 85] Gunther R. Pfaff, editor.
User Interface Management Systems.
Springer-Verlag, 1985.
Proceedings of the IFIP/EG Workshop on User Interface Management Sytems, Seeheim, Federal Republic of Germany, October 1983.

[Pike 84] Rob Pike and Brian W. Kernighan.
Program design in the UNIX system environment.
AT&T Bell Laboratories Technical Journal 63(8):1595-1606, October 1984.

[Purtilo 85] J. Purtilo.
Polylith: An environment to support management of tool interfaces.
In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, pages 12-18. Seattle, WA, June 1985.

[Raker 85] Daniel Raker and Harbert Rice.
Inside AutoCAD: A Teaching Guide to the AutoCAD Microcomputer Design and Drafting Program.
New Riders, Thousand Oaks, CA, 1985.

[Reich 85] Richard Reich.
The macro maker.
MacUser 1(2), December 1985.

[Rexx 83] *VM/SP System Product interpreter reference.*
International Business Machines Corporation, 1983.
SC24-5239.

[Rosenberg 86] Jonathan Rosenberg, Larry Raper, David Nichols, and M. Satyarayanan.
LWP User Manual.
Technical Report CMU-ITC-84-37, Carnegie Mellon University, Information Technology Center, 1986.

[Scheifler 86] Robert W. Scheifler and Jim Gettys.
The X window system.
ACM Transactions on Graphics 5(2):79-109, April 1986.

[Sheil 83] Beau Sheil.
Power tools for programmers.
Datamation 29(2):131-144, February 1983.

[Shneiderman 87] Ben Shneiderman.
Designing the User Interface: Strategies for Effective Human-Computer Interaction.
Addison-Wesley, 1987.

[Smith 82] D. C. Smith, C. Irby, and R. Kimball.
The Star user interface: An overview.
In *Proceedings of the National Computer Conference*, pages 515-528. 1982.

[Snodgrass 83] Richard Snodgrass.
An object-oriented command language.
IEEE Transactions on Software Engineering 9(1):1-8, January 1983.

[Stallman 81] Richard M. Stallman.
EMACS: The extensible, customizable self-documenting display editor.
In *Proceedings of the ACM SIGPLAN SigOA Symposium on Text Manipulation*, pages 147-156. Portland, OR, June 1981.

[Stallman 86] Richard M. Stallman.
GNU Emacs manual.
Free Software Foundation, Cambridge, MA, 1986.

[Stephenson 73] C. J. Stephenson.
On the structure and control of commands.
Operating System Review 7(4), 1973.

[Strom 83] Robert E. Strom and Shaula Yemini.
NIL: An integrated language and system for distributed programming.
In *Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems*, pages 73-82. June 1983.

[Sutherland 63] Ivan Sutherland.
Sketchpad: A man-machine graphical communication system.
Spring Joint Computer Conference , 1963.

[Szekely 87] Pedro A. Szekely.
Modular implementations of presentations.
In *Proceedings of CHI+GI 1987*, pages 235-240. April 1987.

[Szekely 88] Pedro A. Szekely.
Separating the User Interface from the Functionality of Application Programs.
PhD thesis, Carnegie Mellon University, January 1988.
Technical report CMU-CS-88-101.

[Tanner 85] Peter P. Tanner and William A. S. Buxton.
Some issues in future user interface management system (UIMS) development.
User Interface Management Systems.
Springer-Verlag, 1985, pages 67-79.

[Tanner 86] P. P. Tanner, S. A. MacKay, D. A. Stewart, and M. A. Wein.
Multitasking switchboard approach to user interface management.
Computer Graphics 20(4):241-248, August 1986.

[Teitelman 85] Warren Teitelman.
A tour through Cedar.
IEEE Transactions on Software Engineering SE-11(3):285-302, March 1985.

[Tempo 86] *Tempo: Intelligent macros for the Macintosh.*
Affinity Microsystems, Ltd., Boulder, CO, 1986.

[Thomas 83] J. J. Thomas and G. Hamlin.
Graphical input interaction technique (GIIT) workshop summary.
Computer Graphics 17(1):5-30, January 1983.

[Wilkes 68] Maurice V. Wilkes.
Computers then and now.
Journal of the ACM 15(1):1-7, January 1968.
1967 ACM Turing Award lecture.

[XDR 86] *External data representation protocol specification.*
Sun Microsystems, Inc., Mountain View, CA, 1986.

[XEdit 83] *VM/SP System Product editor command and macro reference.*
International Business Machines Corporation, 1983.
SC24-5221.