The WINTERP Widget INTERPreter — An application prototyping and extension environment for OSF/Motif.

Niels P. Mayer (mayer@hplabs.hp.com)

Hewlett-Packard Laboratories
Human-Computer Interaction Department
1501 Page Mill Road
Palo Alto, CA. 94304-1126
USA

ABSTRACT

Winterp is an interactive, language-based user-interface and application-construction environment enabling rapid prototyping of applications with graphical user interfaces based on the OSF/Motif UI Toolkit. *Winterp* also serves as a customization environment for delivered applications by providing a real programming language as an extension language. Many existing user-interface languages only have the expressive power to describe static layout of user interface forms; by using a high-level language for extensions and prototyping, *Winterp* also handles the dynamic aspects of UI presentation, e.g. the use of direct manipulation, browsers, and dialog. *Winterp* makes rapid prototyping possible because its language is based on an interpreter, thereby enabling interactive construction of application functionality and giving immediate feedback on incremental changes.

Winterp's language is based on David Betz's public domain *Xlisp* interpreter which features a subset of Common Lisp's functionality. The language is extensible, permitting new Lisp primitives to be added in the C language and allowing hybrid implementations constructed from interpreted Lisp and compiled C. Hybrid implementation gives *Winterp*-based applications the successful extension and rapid-prototyping capabilities of Lisp-based environments, while delivering the multiprocessing performance of C applications running on personal *Unix* workstations.

October 6, 1991

1. Introduction

Winterp[1] is a **W**idget[2] **Interp**reter, an application development environment enabling rapid prototyping of graphical user-interfaces (UI) through the interactive programmatic manipulation of user interface objects and their attached actions. The interpreter, based on David Betz's *Xlisp* [Betz89], provides an interface to the X11 toolkit Intrinsics (Xt), the OSF/Motif[3] widget set [OSF90] [Young90], primitives for collecting data from *Unix*[4] processes, and facilities for interacting with other *Unix* processes. These features enable *Winterp* to support rapid prototyping of applications using multi-window graphical user-interfaces by allowing the user to interactively change both the appearance and functionality.

In addition to prototyping applications and experimenting with UI layout, *Winterp* may be embedded in applications requiring an extension language for customization or systems integration. Traditional X applications based on the Xtoolkit allow users to alter X resources to tailor application UI parameters such as fonts, colors, window sizes, etc. Motif's User Interface Language (*UIL*) [OSF90] [Bourne90] extends that level of customization by allowing the layout of the application's UI widgets to be tailored. As a language, *UIL* has the expressive power to describe the layout of static UI forms, but has none of the control flow and data handling constructs associated with real programming languages. A programming language is needed to support the full range of requirements needed by User Interface Management Systems (UIMS) [Myers89]; to describe dynamic, data-driven UI forms, and to model user/application dialog. *Winterp* provides such an embedded programming language allowing tailoring of the UI's static and dynamic layout, UI-to-application dialog, and application functionality.

Winterp is thus an interactive ''language based'' user-interface development environment (UIDE). *Winterp* is not a UIMS — it provides UI primitives and a high-level language to support a wide variety of UI-to-application partitionings[5] that are characteristic of the UIMS approach. *Winterp* is designed to allow the programmer to *evolve* a suitable UIMS model that is appropriate for extending and customizing a particular application. *Winterp* is also designed to support direct manipulation UI building. The current version contains a useful primitive for ''direct manipulation programming'' with widget-objects.

An environment similar to *Winterp*'s already exists in the *GNU Emacs* [Stallman87] text editor — in fact, *Winterp* is strongly influenced by *GNU Emacs*' successful design. In *GNU Emacs*, a mini-Lisp interpreter is used to extend the editor to provide text-browser style interfaces to a number of *Unix* applications (e.g. e-mail user agents, directory browsers, debuggers, etc). Whereas *Emacs-Lisp* enables programmers to create new applications by tying together C-implemented primitives that operate on first-class types providing textual interfaces (buffers, windows), *Winterp-Lisp* ties together operations on graphical user-interface objects implemented by the Motif widgets. Both application construction environments achieve the flexibility, expressiveness, and rapid-prototyping capabilities common for systems implemented in Lisp, while still attaining the speed of execution and (relatively) small size associated with C-implemented applications.

Winterp was initially made public on the MIT X Consortium's X11r4 ''contrib'' distribution; up-to-date versions are available via anonymous ftp from a number of Internet sites including export.lcs.mit.edu. *Winterp* is quite robust and bug-free, it is in active use by a number of research projects at HP Labs, and is also being used by companies and universities worldwide. *Winterp* was designed to be portable — it runs on ''standards-oriented'' *Unix* platforms without porting. A number of improvements have already been contributed by *Winterp*'s user group since *Winterp*'s initial public release; submitted improvements will be included in publicly available updates of *Winterp*.

---

1 *Winterp* is pronounced WIN-TERP, not WINTER-P.

2 A Widget is a graphical object that can be manipulated by mouse or keyboard input: examples of OSF/Motif widgets include scrollbars, pushbuttons, menus, text editors, etc.

3 Motif is a trademark of the Open Software Foundation.

4 *Unix* is a trademark of American Telephone and Telegraph, Bell Laboratories.

5 Examples of such UI-to-application partitioning that can be implemented in *Winterp* include Smalltalk's Model-View-Controller paradigm, state transition machines, event grammars, etc.

2. Background

2.1. Design Constraints

Winterp was created as the platform on top of which the ''Collaborative Interaction Tools'' project at Hewlett-Packard Laboratories is developing *Strudel* [Shepherd90], an extensible electronic conversation toolkit enabling computer supported cooperative work. The design of *Strudel* has resulted in strong constraints on *Winterp*: for development, we required an environment supporting rapid prototyping and exploratory programming; for delivery to end-users, *Strudel* had to allow extensive customization in order to fit in to a particular group's computer-based work environment; finally, to allow us to validate our groupware design, the platform must be acceptable to ''early adopters'' wanting to use *Strudel* in real work situations. Unlike traditional software, groupware cannot be evaluated by individual beta-testers — the power of groupware comes from having a number of people using it to work together. In order to gain this wide experimental user base, we found it necessary to place additional constraints on the platform: it needs to be freely distributable, easy to install, and it must have good performance when running alongside other applications on a standard *Unix* workstation. In this paper, we discuss the design of *Winterp* with such constraints in mind, pointing out the advantages of *Winterp* as a general platform for application prototyping and delivery.

2.2. Strudel — A Customizable Application Based on Winterp

Strudel is a generic framework for interlinking group e-mail conversations and group or individual tasks. Strudel consists of *Winterp*'s interpreter and user interface primitives, combined with primitives implementing a distributed hypertext-like system using replicated e-mail messages as nodes, and high-level operators that will allow the nodes to be viewed-by and linked-to a variety of browsers (e.g. conversation presenters, task and to-do lists, calendars). *Strudel*'s flexible architecture makes it akin to a ''*GNU Emacs* for groupware'' — the specialized primitives permit the system to be customized to support special modes of communication for particular workgroup environments much in the same way that *GNU Emacs* Lisp is used to customize the editor to support special editing modes for particular programming environments.

Users will be able to choose from a library of e-mail forms that are designed to track specific types of conversations — scheduling meetings and resources, software defect tracking, group design deliberation, etc. Workgroups can extend the library of forms to help capture and manage recurrent conversations that are not covered by *Strudel*'s standard forms library. The kinds of interface customizations can range from adding new menu entries for often-used functions to designing new e-mail forms and associated browsers for their data. We are working closely with a few HP entities to provide *Winterp*-Lisp ''scripts''[6] implementing specific scenarios involving conversations arising in the domain of team software and hardware production: group design deliberation, software maintenance, defect tracking, etc. See Figure 1 for a snapshot of the *Strudel* prototype; for further information on *Strudel*, see [Shepherd90].

Research related to *Strudel* includes speech act and dialog theory [Searle76] [Reichman85], commitment-based software frameworks [Fikes82] [Winograd86], *The Coordinator*[7] and the language/action perspective [Fikes82] [Winograd87], e-mail message filtering [Rose86] [Borenstein88] semi-structured message systems [Malone86], the Object-Lens [Lai88], conversation management [Comer86] [Dollimore89] [Sulonen90] [Kaplan90], office procedure and coordination models [Holt81], collaborative hypertext systems [Trigg86], and Issue-Based Information Systems [Conk87].

3. Interpreting User Interfaces?

Winterp differs from many other UI languages in that it is interpretive rather than compiled. We believe that interpretation of UI specifications offers a number of practical advantages over the compilation approach taken by *UIL*, traditional C programming with the Xtoolkit, or compiled UIMSs.

Neither traditional C Xtoolkit applications, nor *UIL* applications take advantage of the interpretive nature of the Xtoolkit. When programming in C, one is forced to go through tedious edit/compile/test cycles even

---

[6] In analogy to ''shell scripts'' running under *Unix* shell programs.

[7] *The Coordinator* is a registered trademark of Action Technologies, Inc.

Figure 1. The *Strudel* prototype, showing the e-mail browser and some active e-mail forms used in conversations arising during team software production.

for trivial program changes. The Xtoolkit provides a resource manager (Xrm) that allows one to shorten the edit/compile/test cycle for simple application tailoring such as selecting fonts, colors, label names, or choosing from an enumerated set of application-defined customization choices. This is implemented by having the application load (non-interactive interpretation) the appropriate resource settings from the resource data base at application initialization time. While this mechanism eliminates a compilation stage for a number of simple customizations, the result is still a cycle consisting of repeated edits of the resource database followed by running the application to test the results. *UIL* expands on the type of customizations possible via Xrm by reading a compiled, structured description of the widget hierarchy, along with associated resources and callback names. Rather than ''interpreting'' the data from a resource database, as is done with Xrm, *UIL* uses an additional compilation stage. While *UIL* compilation is quicker than C compilation, it still makes rapid prototyping impractical because of the edit/compile/test cycle.

In contrast to the batch approach provided by *UIL* or C, *Winterp* allows interactive programmatic manipulation of the UI via a message-passing mechanism that takes full advantage of the interpretive, object-oriented nature of the Xtoolkit. The Motif widgets are ''interpretive'' in that one can give programmatic commands to the Motif library to create new widgets, and the Xt intrinsics will create the new widget on-the-fly. One can also send messages to created widget objects via the ''methods'' implemented by widget-class specific functions in Motif or the Xt Intrinsics (such as `XtSetValues()`) — the effects of these messages are interpreted by the toolkit and result in an eventual updating of the graphics and actions associated with the widget.

Winterp provides access to the ''interpretive'' nature of the Motif widgets through its built-in *Xlisp* interpreter. The interpretive approach enables rapid prototyping because one receives immediate feedback on changes — one can incrementally build up a user interface, piece-by-piece, and one can play ''what if'' games by modifying both the layout and functionality of the application. *Winterp* even includes a ''direct manipulation'' primitive that allows changing widget resources, callbacks and event-handlers by designating a widget with the mouse. One need not rerun or recompile the application in order to see the results of a change to a *UIL* or X resource — with *Winterp*, incremental changes to an application can be tested interactively.

4. The Role of a Programming Language in a UIMS.

The UIMS approach requires the power of an embedded programming language to allow abstractions to be derived from complex changes in application state and data. Such abstractions provide the separation between application semantics and the user-interface that UIMSs strive for. This separation is really nothing other than an extension of the notion of encapsulation to the architecture of applications based on graphical user-interfaces. Such encapsulation makes it easier to experiment with the ''look and feel'' of an application without side-effects, enabling designers to more easily refine their applications via rapid prototyping. A well architected system will also provide end-users with a means of customizing the UI without adverse effects on the application's functionality. By embedding full programming language capabilities in a user-interface language, *Winterp* allows designers to develop the appropriate language-based UI abstractions for the particular kind of application and extension architecture.

In contrast to *Winterp*, many of the problems in developing and extending applications based on Motif's User Interface Language (*UIL*) stem from the lack of full programming language support — *UIL* only provides a language supporting a module system, a static widget description language, and simple expression arithmetic. In non-trivial applications, one must write, compile, and link C code which calls hooks into *UIL* in order to control dynamic dialog components that make up the UI of a real application. Such an architecture imposes strong constraints on the amount of customization possible without modifying the C source code. Ultimately, the lack of a programming language in *UIL* limits the usefulness of separating the UI description from the application's functionality:

> For a complex interface, fetching widgets from UID[8] files won't satisfy all of a programmer's needs. Thus, XUI comes with a complete C language library of calls through which the DECwindows[9] widgets can be defined and controlled. These calls are useful, but by using the

[8] A UID file is a User Interface Description file that is produced by compiling a UIL file.

[9] DECwindows is a trademark of Digital Equipment Corporation.

> C language libraries, the correspondence between program structure and program function given by UIL may be lost. [Bourne90 — p. 40][10]

With *Winterp*, programmers can use language constructs to represent and manipulate the state of the application and the UI. *Winterp* makes an effective prototyping environment because one can use *Xlisp* to build the user interface, prototype the ‘‘dialog’’ aspects of the working application, and use *Xlisp*’s object system to evolve language-based abstractions. Depending on the kind of UI style and application architecture, one may use *Winterp*’s features to encapsulate and separate application functionality from the UI representing that functionality; alternately one may find it advantageous to build higher level UI constructs which use class-inheritance to specialize generic UI objects provided by Motif into application-specific objects. Because *Winterp*’s language-base is interpretive, it can be used to describe dynamic, data-driven user-interfaces such as those found in *Strudel*.

Traditional UIMSs are based on an abstract model for separating application functionality from its associated user interface — e.g Smalltalk’s Model View Controller, the Seeheim UIMS model, transition networks, event grammars, etc. [Myers89]. *Winterp* is not a UIMS; it provides little policy for separating the application semantics from the UI. However, *Winterp*’s language base enables the design and development of a variety of UI-to-application modularization policies.

The lack of a particular UIMS policy in *Winterp* is in accordance with the current view that traditional UIMS models are problematic; that it is difficult to separate UI from application semantics in a general way:

> ... in recent years, as UIMSs are built to handle more sophisticated user interfaces with direct manipulation ... there has been concern that the separation between user interface and application raises more problems than it solves. Perhaps the prototypical problem arising from this separation is that of whether to handle the semantics of an interaction in the UIMS or the application program. For example, feedback is a task typically handled by the UIMS, but the semantically rich feedback required by direct manipulation user interfaces (e.g. highlighting while moving the mouse) is difficult to do without involving the underlying application. If the application and the UIMS thus need to frequently interact, their separation becomes a hindrance rather than a help.... [Rosenberg88]

In developing applications with *Winterp* we have found that for certain classes of applications, such as instrumentation controllers, a simple recursive, event-driven state machine[11] is an appropriate abstraction separating application semantics from the UI. In dynamic, data-driven UI’s (browsers, graph editors, CAD drawing tools), separating the UI from the application objects has proven to create messy architectures; a better solution is to use Motif widgets subclassed[12] into application-specific UI objects handling their own state and actions. We believe that it is best to give programmers the full capabilities of the Motif toolkit, augmented by *Winterp*’s interpretive, language-based interface. Application programmers can use these features to evolve the appropriate UIMS to do the job.

Winterp’s policy-free UIMS architecture makes *Winterp*-based applications far more customizable than applications based on UIL or the Widget Creation Library (WCL). Both UIL and WCL enforce an architecture which strongly separates the dynamics of the application from the static presentation of the panels comprising the application’s UI. Such an architecture will only allow trivial customization of applications employing state-based dynamics. A simple example of such a problem: one wants to add a button to an application that does the same thing as a menu-entry (because one selects this entry often). The application designer had the menu entry ‘‘grey out’’ to indicate that it is not a suitable choice given a particular application state. With UIL or WCL, special code must be written at the C-language level to handle such state-based changes; this code is tied to assumptions about the static interface described by the user interface

---

[10] This quote from [Bourne90 — p. 40] comes from an article on programming with DECwindows, focusing on the use of *UIL*. Although the *UIL* in DECwindows is not the same as Motif’s *UIL*, both versions provide essentially the same functionality.

[11] Lisp’s list and symbol manipulation features make it especially easy to implement such state machines.

[12] Note that *Winterp*-Lisp allows subclassing of widgets without resorting to the complexities and tedium of widget subclassing in C with the X toolkit.

language. In the case of the customization just described, the appropriate state-based inactivation of the button may not occur — this can result in the application entering a disallowed state if the user selects it without knowing the choice was invalid. Even if the error can be trapped in the button's callback, the user is not being given the kind of state-based feedback that is the hallmark of good UI design. The problem here is that application state cannot be described in the UI language. The lack of programming language features make it impossible to propagate even the simplest state changes to state-dependent UI components and this creates an inelegant architecture for both application designers and application customizers.

UIL and WCL force application architectures which hard-code application dialog in C for certain expected UI presentations described in the user-interface language. The assumption, that dialog and presentation are separate, limits the kinds of UI customizations that are possible to trivial layout modifications. Applications use dialog to prompt the user for information needed to complete a previously executed command. Dialog is also used to limit the amount of screen real estate used by an application — selection choices and input fields that are not of primary concern are hidden until needed. Personal preference, screen real estate usage and application usage patterns should dictate whether one elects to select a choice directly, or via dialog. Thus, the customization of dialog versus direct-presentation may be a useful feature for certain applications. For example, some electronic mail (e-mail) interfaces support multiple folders for filing/copying information that is received in one's ''in-box''. Some e-mail UI's will use dialog for folder selection when a ''copy'' or ''move'' operation is invoked. People doing a lot of filing may prefer to have direct control over the selected folder[13], rather than having the system prompt for a folder each time. With UIL or WCL, customization of the tradeoffs between presentation and dialog is not possible within the user interface language due to lack of programming language constructs. With *Winterp*, such customizations are possible.

5. Advantages of Lisp as a UIMS Language

Programming language features are present in UIMSs such as *Open Dialogue* [Schulert88], and *Serpent* [SEI89]. *Winterp* differs from such UIMSs because it does not attempt to define a new user interface language into which programming language constructs need to be introduced. Rather, *Winterp* uses a subset of a standard language — Common Lisp — and extends this language to describe UI layout and dialog. The choice of Lisp as the widget layout and prototyping language in *Winterp* provides important advantages:

- Highly expressive:

  Lisp allows new functionality to be expressed quickly using less code than an equivalent C program because it is a high-level language featuring symbol and list manipulation [Creech87b] [Creech87a], first class procedures [Creech87b] [Creech87a], object-oriented programming [Betz89], and automatic memory management via garbage collection [Creech87b] [Creech87a]. Lisp's features for manipulating lists and trees of arbitrary objects are used to represent UI layout structures and *Winterp* introduces new objects (widget-objects) to represent UI components.

- Designed to be interactive, interpretive:

  Lisp makes a good environment to drive an interactive interface to the Motif widgets because the language was designed to be interpreted. Interpretation can also be accomplished in traditionally compiled languages (such as C), but in Lisp, small one-off changes can be made with reduced effort due to dynamic typing. Such changes are typical in rapid prototyping.

  Lisp's interactive error handling and debugging allows for programming errors to be caught, debugged, and fixed interactively. Debugging occurs within the environment that caused the error to arise, and the full power of the language interpreter is available to allow programmers to inspect, alter, or fix the environment. After fixing a bug, one may be able to resume execution of ones code from the point that caused the error without having to restart the program. These debugging features are essential for rapid prototyping.

[13] An example of a e-mail UI that does not utilize dialog for folder selection is the MIT X Consortium client *xmh*. A common complaint about *xmh* is that the folder selection area takes up too much real estate. This is yet another example where dialog versus presentation customization would solve an existing problem.

- Code-Data equivalence:

  Lisp data is represented in the same form as Lisp programs [Creech87b] [Creech87a], which means that programs can perform computations to create/alter data structures representing programs. Such meta-programming allows the creation of high-level data-driven abstractions for user interfaces created programmatically via Motif and Xt Intrinsics calls.

  Winterp-Lisp can thus be used to create dynamic widget layouts through computations that create and mutate data structures representing user-interfaces. For example, in our *Strudel* groupware toolkit, *Winterp*'s interpreter and Motif UI primitives are used as a description language for creating and processing active/graphical forms. Such forms can be sent (as textual programs) through standard e-mail channels. The receiving *Strudel* system interprets the message and displays a form containing a user interface built from arbitrary combinations of widgets and bitmaps. These ‘‘forms’’ can even be whole user-interfaces and associated programs that can be passed around from user-to-user as Lisp ‘‘continuations.’’[14] See Figure 1 for an example of a *Strudel* form.

6. Implementation Issues in Embedding a Lisp Interpreter

6.1. Problems with Traditional Lisp Systems

Despite its advantages, Lisp is traditionally associated with large, slow and expensive systems[15] — Lisp's flexibility has its costs. Attempts at building Lisp-based applications that are good citizens on *Unix* workstations have been problematic because systems such as Common Lisp (CL) create huge, resource-hungry processes that swap out all other applications and cause memory thrashing [Creech87b] [Creech87a]. This results in unacceptable overall system performance if the CL process is but one of many processes competing for resources on a *Unix* box.

While CL continues to be an excellent prototyping platform, few acceptable solutions to the ‘‘delivery problem’’ have been found for applications embedded in a CL environment. One unacceptably drastic solution is to recode the completed CL-based prototype into C — an approach often used to create deliverable versions of expert systems and other complex applications. This approach is time consuming, error prone, and changes the feel and the flexibility of the delivered application: applications prototyped with CL assume and make use of the underlying features of the Lisp system; these assumptions must be removed from the design or be recoded in the delivery language.

In the past, we have also experienced problems in interfacing large Common Lisp systems to other C-implemented libraries and low-level device drivers — the problem stems from the difficulties in importing, exporting and translating arbitrary Lisp data structures to/from the C level; problems also arise from the explicit control requirements of the CL interpreter's evaluator which make it difficult to interface to event-driven programs such as the X toolkit.

6.2. Solution: a Hybrid Implementation

Fortunately, another class of Lisp application has been successful in a general purpose computational environment — a hybrid architecture of Lisp and C giving the flexibility of a Lisp system while allowing delivery of a relatively small and efficient process. Under *Unix*, Richard Stallman has created a highly-customizable editor-based programming environment called *GNU Emacs* [Stallman87] — this is a system that delivers to the *Unix* user a text-editor oriented UI that is the foundation of the Lisp Machine programming environment. Under *MS-DOS*[16], successful programs like *AutoCAD*[17] contain a Lisp customization language embedded in a CAD program.

---

[14] One of our research issues is how to provide security for systems that ‘‘open’’ to programmatic manipulation from outside — they can easily be infected with viruses.

[15] One solution to the ‘‘large’’ and ‘‘slow’’ problems has been to create special operating systems and hardware for Lisp — Lisp machines. Such specialized computers are being priced out of the market by general purpose *Unix* workstations using the X Window System and running software that is portable across a number of vendor platforms.

[16] *MS-DOS* is a trademark of Microsoft Corporation.

[17] *AutoCAD* is a trademark of Autodesk Corporation.

The approach taken by such hybrid applications is that a small mini-Lisp interpreter serves to ‘‘glue’’ together efficient C-implemented primitives that make up an application. User-customization and prototyping under such a hybrid system amounts to using the Lisp interpreter to reconfigure C-implemented building blocks in order to change, modify, or improve the functionality of the system. Such an application architecture follows the ‘‘80/20 heuristics’’ for program execution — low level routines that take up most of the computational resources are coded in C, and are therefore fast and efficient in memory use (no garbage collections caused by low-level code). The Lisp interpreter is relatively slow in comparison to a compiled C program, but it only serves to flexibly glue together components of the ‘‘outer loop’’ of a program. For an illustration of this hybrid architecture, see Figure 2.

Winterp solves the problems traditionally associated with Lisp delivery by using this hybrid approach — a small, fast, lightweight Lisp interpreter based on David Betz’s *Xlisp* serves as an interactive, configurable mechanism tying together high-level C-implemented application-specific primitives. Because *Xlisp* is implemented entirely in C, one can simply use *Xlisp*’s C library to directly recode any Lisp code into a C-implemented primitive.

6.3. Evolving from Prototype to Deliverable with Hybrid Programming

Winterp supports an evolutionary program lifecycle: *Winterp* application writers rapidly prototype new functionality by using the interpreter to interactively refine the layout, looks, and functionality of the application. Once functionality has stabilized, a programmer can improve the application’s efficiency by reimplementing the functionality in C while maintaining the same programmatic interface to the rest of the system. The new primitives will then serve as the building blocks for the next layer of prototyping and customization. The end result, if designed carefully, is a relatively small and fast application that provides the right set of building blocks and hooks to permit end-users to customize the look and feel of the application.

Winterp is also useful for rapid prototyping applications that do not need to be delivered with an embedded customization language. Systems with such delivery goals may still use the aforementioned application lifecycle. As the application matures and Lisp prototype code stabilizes, the program can gradually be recoded entirely into C. Eventually, this process will allow a standard C-implemented Motif program to be delivered.

Contrast this stepwise refinement from prototype to deliverable with the approach of throwing out the entire CL-based prototype and starting from scratch in a language like C or C++. The advantages of creating deliverables incrementally via hybrid programming are: (1) Existing regression tests may be applied to the deliverable under construction, such that the effects of each reimplementation can be tested; (2) Developers and alpha-testers can use the application being recoded for further development and further work, thereby checking the impact of the changes on the usability, customizability, and functionality of the application.

7. An Architecture for Extensible UIs and Applications

The architecture of an application built on top of *Winterp* allows for applications to be delivered with a variety of UI styles. Such customizability is important because it is difficult to please everyone with a single UI style, and because system designers cannot foresee all possible needs of all users. With *Winterp*, UI styles can be specified via *Winterp*-Lisp ‘‘scripts’’ which are loaded into the application at run-time. Inexperienced users can customize the UI or application’s functionality by using ‘‘programming by example’’ based on existing scripts. Application ‘‘gurus’’ may come up with new styles, merge features of existing styles, add shortcuts and accelerators, or come up with new functionality. Such customizations are often distributed to others within the organization. Extensible applications like AutoCAD and Hypercard[18] have even created an ‘‘aftermarket’’ of scripts that create new applications within the environment provided by the extensible application.

Figure 2 shows the architecture of an extensible application in which *Winterp*-Lisp serves as a customization script and ‘‘glue language’’ between C-implemented application and user-interface primitives.

[18] Hypercard is a trademark of Apple Computer.

Figure 2. Architecture of an extensible application based on *Winterp*.

8. Winterp Features

8.1. Xlisp

Winterp uses *Xlisp* [Betz89] because it is reliable, small, fast, and free. *Xlisp* has been around since 1985, and has evolved considerably since it first appeared publicly. Because it has been in widespread use for some time, most bugs have been shaken out of the system. Also, the newsgroup comp.lang.lisp.x has been an effective public channel for exchanging information and patches. *Xlisp* was designed to be run on PCs; because it was designed with a limited environment in mind, it has turned out to be quite fast and memory-efficient while remaining portable across a variety of architectures ranging from 16 bit PC's to workstations. *Xlisp*'s simple object system has enabled an elegant interface to the object oriented structure of the Xtoolkit and the Motif widgets. Finally, *Xlisp* is free, thanks to the generosity of David Betz.

8.2. Motif Widgets are First-Class Objects

Winterp uses *Xlisp*'s Smalltalk-like object system [Betz89] as its interface to the class hierarchy of widgets provided by Motif. Specifically, each Motif widget class is represented by one or more object classes in *Winterp*. A wide range of UI objects are provided. Simple widgets include text or pixmap labels, pushbuttons, toggle buttons as well as scrollbars and scale valuators. More complex widgets include a text editor, a file browser, and a list browser. Motif includes manager widgets that manage the geometries of other widgets via constraints, row/column placement, or menu layout. Shell and dialog widgets provide top-level windows that talk to the window manager. Other widgets that are or will be available in the public domain or through various software suppliers include table layout managers, directed-graph layout managers, graphical gauges, plotting and line graphics widgets, etc.

Xlisp classes describe the type of a particular object by declaring a set of variables held in each object. These *instance variables* may only be accessed by *methods* that respond to *messages* sent to the object. Methods are defined for particular classes, and functionality of other classes may be incorporated into new classes via *inheritance*. From *Xlisp*, Motif widget classes and instances look just like normal *Xlisp* classes and instances, meaning that one can add new methods or override old ones, possibly using polymorphism to exploit similarities between types despite different implementations. Type inheritance and subclassing are available to specialize existing widget classes. The result is that *Winterp* provides a very clean way to interactively rapid-prototype an application, while also providing mechanisms for code structuring and reuse. The latter is necessary in evolving from prototype to a structured, maintainable, and customizable deliverable.

8.3. Open Application Architecture

Winterp promotes an open, extensible architecture for applications because designers cannot foresee all the possible needs of the end-user. In addition to being open to the application customizer, *Winterp* is also open to systems integration in which applications must work together with other applications running on the network. *Winterp* enables such integration because its language interpreter is implemented as a server.

Thus, all *Winterp*-based applications have a built-in, extensible remote procedure call (RPC) mechanism which allows other applications, possibly running non-locally, to send commands to execute application functionality. Such an architecture allows applications to talk to each other, share data, etc. The server configuration can be specified via Xresources, allowing the user of WINTERP or a WINTERP-based application to enable/disable Unix-Domain or Inet-Domain server sockets.

Figure 3 shows a diagram of *Winterp*'s server architecture. Both local and remote applications can invoke remote procedure calls in *Winterp*-based applications. Programmatic changes can be sent to *Winterp* from the *Unix* command shell and shell scripts via a simple client program, `wl`, which is included with the *Winterp* distribution. For example, a user-defined function `start-application` may be called within *Winterp* by executing the following *Unix* command:

```
wl '(start-application)'
```

Figure 3. Inter-application communications via *Winterp*'s *Xlisp* server.

8.4. ''What you program is what you see''

The *GNU Emacs* editor provides an elegant environment for writing and formatting[19] *Winterp* programs. The *Winterp* distribution contains an extension to Emacs' Lisp-mode whereby an Emacs command will send the current Lisp form being edited to *Winterp* for evaluation. This allows truly interactive programming because one need not exit the editor to see the results of evaluating a code fragment; with *Winterp*, one can see the graphical results of interactive changes to a program immediately. Note that *Winterp* is architecturally separate from the *GNU Emacs* editor (see Figure 3) — interfaces to other editors are possible, but are not provided in the current *Winterp* distribution.

Non-Emacs users may find an example program in the *Winterp* distribution useful for interactively editing and evaluating code with the Motif text editor widget. By loading `winterp/examples/w_ctrlpnl.lsp` into *Winterp*, a window will pop up providing a rudimentary editor, file browser, and control panel for the system. The control panel contains buttons that control *Xlisp*'s debugger and error backtrace, as well as controls to load files, edit files, and evaluate the Lisp form being edited.

8.5. Automatic Resource Conversions

In *Winterp*, any Motif resource that can be represented in the X resource manager (i.e. one can set the resource via .Xdefaults) can be specified as a Lisp string, and it will automatically be converted to the appropriate type. This is especially useful for automatically converting strings to XmStrings. Other useful conversions include converting color names to type `Pixel`, and converting bitmap file names to type `Pixmap`.

8.6. Interpreted Callbacks

Xtoolkit callbacks, event-handlers, timeouts, translations, and accelerators are seamlessly integrated with *Winterp* — Xevents can cause arbitrary code (both Lisp and C) to be executed. (See the example code in the next section for details.)

For Xtoolkit translations and accelerators, *Winterp* includes a special action procedure `Lisp()` whose arguments are evaluated as a Lisp function call. For example, the following Xtoolkit translation- or accelerator-table entry[20] will cause the application-defined function `ctrl-A-hit` to be called when the key sequence CTRL-A is entered on the widget:

```
"Ctrl<Key>A: Lisp(ctrl-A-hit ACTION_WIDGET ACTION_XEVENT)"
```

The symbol `ACTION_WIDGET` above is bound to the widget-object that caused the action procedure to fire, and `ACTION_XEVENT` is bound to the XEvent-object that matched the translation or accelerator entry. These values are then passed as arguments to function `ctrl-A-hit`.

8.7. Automatic Memory Management

Xlisp's garbage collector has been extended to reclaim unused X, Xtoolkit, and Motif storage — this allows programmers to concentrate on the UI and application functionality, rather than memory management. Memory management is one of the pitfalls that complicates X programming in C — novices have difficulty in determining the lifetime of objects in X and the Xtoolkit, resulting in hard-to debug program crashes if objects are freed too early, or subtle memory leaks if objects are not freed at all. In *Winterp*, memory need not be managed explicitly, since Lisp's garbage collection automatically frees up any memory that is no longer referenced by the system.

In particular, *Xlisp*'s garbage collector has been extended to reclaim storage associated with destroyed widget objects, along with any toolkit-internal storage associated with these widget objects. Such garbage collection occurs for callbacks, event-handlers, timeouts, XmStrings, and XmStringTables.

---

[19] Emacs' Lisp-mode will automatically indent one's code and help catch unmatched parentheses.

[20] A translation/accelerator entry may be specified in an X resource file, e.g. .Xdefaults, or may be set directly within *Winterp*.

Garbage collection of pixmaps in *Winterp* is especially useful because we can expect pixmaps to take up a reasonable amount of client and server space in typical graphic/iconic Motif applications. *Winterp* extends Motif's reference counting and pixmap caching scheme to work with garbage collection. Server and client-side storage associated with a pixmap will get reclaimed when no references to them exist in any widget-object or other *Winterp* variable.

8.8. Programming by Direct Manipulation

The current release of *Winterp* includes a primitive which allows for ''programming by direct manipulation.'' When one is interactively, programmatically modifying a user interface, one often wants to send a message to a widget-object without knowing it's ''name'' or the symbol to which the widget object is bound. The primitive `get_moused_widget` allows messages to be passed to any widget-object one can point the mouse at. Developers may use this for rapid prototyping — they can immediately see the changes they are making to the UI. Users may customize delivered *Winterp*-based applications by simply ''pointing'' at the widgets they wish to change — colors, fonts, sizes, callbacks, and other such widget parameters may be changed on the fly.

`get_moused_widget` may also be used to implement an interactive, direct manipulation builder and resource editor application on top of *Winterp*. Other direct manipulation primitives are being considered to allow widgets to be placed or moved interactively.

8.9. Extensibility

Winterp is not a closed system. Hybrid programming with *Xlisp* makes it straightforward to add new C-implemented widgets to the system so that they can be accessed via the interpreter. It is also easy to add special Xlib-level primitives to *Winterp* in order to implement functionality not available via the widgets. The same goes for interfaces to special *Unix* device drivers, etc.

Simple widgets (such as a pushbutton) can be interfaced with about 10 lines of C code, which needs to be linked in to the rest of system. More complex widgets (such as a text or graph editor) are added with the same techniques as used by simple widgets. However, if such widgets provide a number of ''convenience functions'' (methods), then each convenience function will need to be interfaced to Lisp. Each convenience function interface usually takes about 5-10 lines of C code. Such C-level interfacing of convenience functions and widgets is simple because most of it amounts to programming via example (''cut and paste'') from existing code.

Widgets requiring new resource representations can be added as well. To achieve this, one needs to code Lisp-to-resource converters, or simply use the String-to-Resource converters required to make such widgets work with the X resource database.

9. Example Use

In the following code, we will first step through some simple *Winterp* code that will allow us to create windows, put widgets inside those windows and attach callbacks to execute code when a particular widget is activated via X input. Afterwards, we will take some of concepts from the example code and apply it to a real application — a bitmap browser.

9.1. Introduction to Xlisp Syntax

First, we shall introduce a few rudimentary aspects of *Winterp* programming and *Xlisp* syntax for those not familiar with the language. The examples below only use a small fraction of the functionality of Lisp — assignment, function calls, object creation, and message passing.

- Symbols:

  Those not familiar with Lisp will find symbols to be similar to variables and function names in compiled languages such as Pascal or C. When a symbol is evaluated, it returns it's value. A value can be assigned to a symbol with the function `setq`.

- Assignment:

The function setq will bind <value> to <symbol>:

```
(setq <symbol> <value>)
```

Upon evaluation of <symbol>, <value> will be returned.

- Function calls:

A function named by the symbol <function-name> can be called with a sequence of arguments <argument-1> <argument-2>...<argument-n>. The syntax for a function call is:

```
(<function-name> <argument-1> <argument-2>...<argument-n>)
```

Where each <argument-i> is a symbol, a function call, or a value. Each argument is evaluated before being passed in to the function associated with <function-name>.

- Sending a message to an object:

The *Xlisp* function send allows one to send the *message* :<message> to the object <object-instance>. If <object-instance>'s *class* contains a *method* for the particular *message*, that method will be called just like a function call with arguments [...arguments...].

```
(send <object-instance> :<message> [...arguments...])
```

- Object creation:

To create a new instance of *class* <object-class>, send the message :NEW to the class. The arguments [...arguments...] are dependent on what kinds of initialization need to be done by the instance initializer method :ISNEW.

```
(send <object-class> :NEW [...arguments...])
```

9.2. Create a Top-Level Window:

Below, we create a top level window[21] by creating an instance of the Xt Intrinsics topLevelShellWidgetClass. The symbol toplevel_w is bound to the widget-object representing the new top level widget instance. The keyword arguments :XMN_HEIGHT and :XMN_WIDTH are set so that the window is of size 500x500[22] , and we also set the titlebar and icon names displayed by the window manager.

In order to actually create the toplevel window, the Xt Intrinsics require that function XtRealizeWidget() be called — that operation is done by sending the message :REALIZE to the widget instance.

Sending the two forms below to *Winterp*'s server will cause a new toplevel window to pop up immediately:

```
(setq toplevel_w
      (send TOP_LEVEL_SHELL_WIDGET_CLASS :new
            :XMN_HEIGHT    500
            :XMN_WIDTH     500
            :XMN_TITLE     "Winterp: example-1"
            :XMN_ICON_NAME "example-1"
            ))
(send toplevel_w :realize)
```

[21] A top level window is conceptually a child of the root window in the X server. These top level windows are managed by the window manager, allowing users to move, resize, and iconize application windows.

[22] Hard-coding the window size in an X application is not a good idea. The code we are presenting here is just the result of our interactive session with *Winterp*, and not representative of how a real application should be coded.

9.3. Create a Widget Layout Manager:

Above, we have created an instance of a `TOP_LEVEL_SHELL_WIDGET_CLASS` which is bound to symbol `toplevel_w`. We now attach a layout manager widget as a child of the top level widget. Layout manager widgets generally do not add to the visual appearance of an application in and of themselves, rather, they manage the layout and geometries of the set of child widgets that appear within them. One of the important manager widgets in Motif is the Row/Column manager `xmRowColumnWidgetClass`. This widget will lay out its child widgets to appear in a configurable number of rows and columns.

In the code below, we create an instance of `XM_ROW_COLUMN_WIDGET_CLASS` bound to symbol `rowcol_w`. The widget is a child of an instance of `XM_SCROLLED_WINDOW_WIDGET_CLASS` which is bound to `scrl_w`. The scrolled window widget will put up scrollbars if the row/column widget within in grows larger than the 500x500 toplevel window associated with `toplevel_w`. The arguments to the resources `:XMN_ORIENTATION` and `:XMN_PACKING` cause the row column widget's children to be laid out vertically, with no space padding between widgets.

Sending the following code to *Winterp*'s server will cause the widgets to be created:

```
(setq scrl_w
      (send XM_SCROLLED_WINDOW_WIDGET_CLASS :new :managed
            toplevel_w                      ;parent is toplevel
            :XMN_SCROLLING_POLICY :automatic
            ))
(setq rowcol_w
      (send XM_ROW_COLUMN_WIDGET_CLASS :new :managed
            scrl_w                          ;parent is scroller
            :XMN_ORIENTATION      :vertical
            :XMN_PACKING          :pack_tight
            :XMN_ENTRY_ALIGNMENT  :alignment_center
            :XMN_FOREGROUND       "Black"
            :XMN_BACKGROUND       "LightGray"
            ))
```

Note that the `:MANAGED` argument for widget creation does the equivalent of `XtManageWidget()`. In simplistic terms, managing a child widget makes it visible within its parent.

9.4. Adding Widgets to the Layout Manager:

Now we can interactively add a variety of widgets to the Row/Column manager by sending each of the forms below to *Winterp*'s server. For every form that gets evaluated below, we will see the corresponding widget appear within the window titled ‘‘Winterp: example-1’’ that we created earlier. See Figure 4 to see how these widgets look.

```
(setq arrow_up_w
      (send XM_ARROW_BUTTON_WIDGET_CLASS :new :managed rowcol_w
            :XMN_ARROW_DIRECTION :arrow_up
            ))
(setq big_arrow_left_w
      (send XM_ARROW_BUTTON_WIDGET_CLASS :new :managed rowcol_w
            :XMN_ARROW_DIRECTION :arrow_left
            :XMN_HEIGHT          50
            :XMN_WIDTH           50
            ))
(setq label_w
      (send XM_LABEL_WIDGET_CLASS :new :managed rowcol_w
            :XMN_LABEL_STRING "hi, I'm a label widget"
            :XMN_FOREGROUND      "White"
            :XMN_BACKGROUND      "Black"
            ))
```

```
(setq string_push_button_w
      (send XM_PUSH_BUTTON_WIDGET_CLASS :new :managed rowcol_w
            :XMN_LABEL_STRING    "hi, I'm a pushbutton widget"
            :XMN_FOREGROUND      "Black"
            :XMN_BACKGROUND      "LightGrey"
            ))
(setq pixmap_push_button_w
      (send XM_PUSH_BUTTON_WIDGET_CLASS :new :managed rowcol_w
              :XMN_LABEL_TYPE    :pixmap
              :XMN_LABEL_PIXMAP  (xm_get_pixmap
                                  "/users/mayer/src/bitmaps/bob0.xbm"
                                  "black" "white")
              ))
(setq text_editor_w
      (send XM_TEXT_WIDGET_CLASS :new :managed :scrolled rowcol_w
            :XMN_EDIT_MODE       :MULTI_LINE_EDIT
            :XMN_FONT_LIST       "hp8.10x20b"
            :XMN_HEIGHT          100
            :XMN_FOREGROUND      "Black"
            :XMN_BACKGROUND      "LightGrey"
            ))
(setq string_toggle_button_w
      (send XM_TOGGLE_BUTTON_WIDGET_CLASS :new :managed rowcol_w
            :XMN_LABEL_STRING    "hi, I'm a togglebutton widget"
            ))
(setq pixmap_toggle_button_w
      (send XM_TOGGLE_BUTTON_WIDGET_CLASS :new :managed rowcol_w
            :XMN_LABEL_TYPE      :pixmap
            :XMN_LABEL_PIXMAP    "/users/mayer/src/bitmaps/bob1.xbm"
            ))
```

9.5. Setting or Querying the State of a Widget

Messages may be sent to widgets in order to set or get widget-specific state information. Widget resources are accessible via the messages `:SET_VALUES` and `:GET_VALUES`, the methods associated with those messages correspond to the Xtoolkit functions `XtSetValues()` and `XtGetValues()`, respectively.

Some widget classes define certain class-specific methods — for example, the Motif `XM_TEXT_WIDGET_CLASS` responds to message `:SET_STRING` by invoking the method `XmTextSetString()` which sets the string within the editor. To set the string in the editor widget instance created above, we do:

```
(send text_editor_w :set_string "Hi, I'm a text editor widget")
```

Note that *Winterp*'s use of a real object system brings added safety and error checking to programming with Motif. In C, a programmer may make the naive assumption that calling `XmTextSetString()` on a label widget would set the text string in the label; in reality, the program would core dump because the function was called on an instance of the wrong class. In *Winterp*, such a misunderstanding would result in a continuable error which the programmer could fix and continue-from interactively.

9.6. Adding Callbacks

Above, we have shown how to create a ‘‘random’’ layout of widgets with *Winterp*. In order to make the interface do anything, we need to add callbacks, event-handlers, or actions to the widgets — these will call arbitrary code in response to user input.

For example, suppose we want to make the push button widget with the label `"hi I'm a pushbutton widget"` call some code that will print something on standard output when it is activated via mouse. To do this, we use the *Winterp* equivalent of `XtAddCallback()`, the method `:add_callback`.

Each widget class's `:ADD_CALLBACK` method takes as argument the name of the callback list, a widget-class specific list of symbols to bind to data from the callback's `call_data` callback structure, and a list containing arbitrary code to be evaluated upon execution of the callback. Note that the symbols bound to the values from `call_data` are only bound within the lexical and dynamic scope of the callback code.

In the example below, we are adding a callback to the push button widget's `XmNactivateCallback` list, which will fire the callback's code whenever the pushbutton is pressed and released. The code simply calls the `format` function, which is akin to the *Unix* `fprintf(3S)` subroutine: it will print out the callback reason and the widget-object that caused the callback on the *Unix* standard output.

```
(send string_push_button_w :add_callback
      :XMN_ACTIVATE_CALLBACK                  ;name of callback list
      '(CALLBACK_REASON CALLBACK_WIDGET)      ;list of callback data
      '(                                      ;code to execute
        (format T "reason = ˜A; widget = ˜A\n"
                CALLBACK_REASON CALLBACK_WIDGET)
        ))
```

Obviously, the callback above doesn't do anything very interesting. We will have to wait till the bitmap browser example to see a more useful callback.

9.7. Direct Manipulation Programming

In *Winterp*, any widget's resources may be changed interactively via method `:SET_VALUES`, which is equivalent to the Xtoolkit's `XtSetValues()`. For example, we can interactively change the direction of the arrow in the arrow button widget by sending *Winterp* the following code. Immediately after *Winterp* evaluates this code, one will see the button display a downward facing arrow:

```
(send arrow_up_w :set_values
      :XMN_ARROW_DIRECTION :arrow_down
      )
```

In the example above, we needed to get a hold of the widget-object value bound to symbol `arrow_up_w` before we could operate on the widget. In some cases, this is impractical, as one would rather just point at the desired widget. The *Winterp* primitive function `get_moused_widget` allows just that.

If we send the form below to *Winterp*'s server, we will see the cursor change to a ''crosshair'' indicating we are to click on a widget. Upon clicking on the widget, we will see it change colors:

```
(send (get_moused_widget) :set_values
      :XMN_FOREGROUND "Green"
      :XMN_BACKGROUND "LightGrey"
      )
```

The primitive `get_moused_widget` can be used in any place that requires a widget-object as a parameter. One can use this function to interactively change callbacks, font sizes, colors, etc. In delivered *Winterp*-based applications, one can use this feature to experiment with resource settings without needing documentation on the internals of the application, nor the widget hierarchy.

9.8. A Simple Application — a Bitmap Browser

We now create a simple application — a bitmap browser — using some of the ideas from the code above. For each bitmap file in a directory of bitmaps, this browser will put up a label widget containing the name of the bitmap file, followed by a push button widget displaying the bitmap itself, followed by a separator widget. All these widgets are held within a scrolled row/column widget, which we've already seen used in the example above. When a push button containing a pixmap is pressed, that pixmap becomes the root window's tile pattern by calling the X11 program `xsetroot(1)` with the appropriate arguments. In Figure 4, the window titled ''Bitmap Browser Example'' shows what this bitmap browser looks like.

The simplicity of implementing the bitmap browser application demonstrates the power of *Winterp* and the expressiveness of its language syntax. Furthermore, this example illustrates one of the interesting features of *Winterp* — the ability to subclass a Motif widget by using *Xlisp*'s object system.

We have seen the following code in the example above — here, we create a top level widget, which contains a scrolled window widget, which in turn contains a Row/Column widget:

```
(setq toplevel_w
      (send TOP_LEVEL_SHELL_WIDGET_CLASS :new
            :XMN_TITLE "Bitmap Browser Example"
            :XMN_ICON_NAME "Bitmap-Br"
            ))
(setq scrl_w
      (send XM_SCROLLED_WINDOW_WIDGET_CLASS :new :managed
            toplevel_w
            :XMN_SCROLLING_POLICY :automatic
            ))
(setq rowcol_w
      (send XM_ROW_COLUMN_WIDGET_CLASS :new :managed
            scrl_w
            :XMN_ORIENTATION     :vertical
            :XMN_PACKING         :pack_tight
            :XMN_ENTRY_ALIGNMENT :alignment_center
            :XMN_FOREGROUND      "Black"
            :XMN_BACKGROUND      "LightGray"))
```

Rather than placing an identical callback structure on each push button widget in the browser (which is memory-inefficient), we exploit a feature of the Motif Row/Column widget which allows us to place a single callback, `XmNentryCallback`, on the Row/Column manager widget. Any activate callback occurring on a widget that is a child of this Row/Column widget will end up firing the callback below. When the callback fires, symbol `CALLBACK_ENTRY_WIDGET` will be bound to the entry in the Row/Column widget that was activated. The code for the callback then sends the message `:XSETROOT` to that widget, which will cause the root window's background tile to be set to the bitmap contained in the button that was activated.

```
(send rowcol_w :set_callback :XMN_ENTRY_CALLBACK
       '(CALLBACK_ENTRY_WIDGET)
       '(
         (send CALLBACK_ENTRY_WIDGET :xsetroot)
         ))
```

Now, we create a specialization of the Motif `XM_PUSH_BUTTON_GADGET_CLASS`[23] by making a trivial subclass called `Niels_Pixmap_Push_Button_Class`.

From the outside, instances of this class work just like a pushbutton. The difference is that at initialization time, an instance of `Niels_Pixmap_Push_Button_Class` requires that one give it a filename containing the full path to a X bitmap file. The pushbutton will then display this bitmap.

Internally, this subclass contains an additional instance variable `pixmap_file`. Each instance of this push button will thus contain the name of the bitmap file it is displaying. This instance variable is later used by the method `:XSETROOT`.

[23] A Motif gadget, is for all intents and purposes the same as a Motif widget. They are more efficient, and take up less client and server memory.

The following code implements the subclassed widget. The second form below overrides the instance initializer associated with the superclass (the *Winterp* code associated with XM_PUSH_BUTTON_GADGET_CLASS that actually creates the widget) in order to initialize the instance variable and pass special arguments on to the superclass's initializer method :ISNEW

```
;;;
;;; make a trivial subclass of XM_PUSH_BUTTON_GADGET_CLASS
;;;
(setq Niels_Pixmap_Push_Button_Class
      (send Class :new
            '(pixmap_file)  ;a new inst-var for this subclass
            '()             ;no class vars for subclass
            XM_PUSH_BUTTON_GADGET_CLASS))
;;;
;;; override XM_TOGGLE_BUTTON_GADGET_CLASS's instance initializer
;;;
(send Niels_Pixmap_Push_Button_Class :answer :isnew
      '(filename &rest args)
      '(
        (setq pixmap_file filename)
        (apply 'send-super `(:isnew ,@args
                                    :XMN_LABEL_TYPE :pixmap
                                    :XMN_LABEL_PIXMAP ,filename))
        ))
```

Now we define the method :XSETROOT for the new class. When one sends the message :XSETROOT to an instance of class Niels_Pixmap_Push_Button_Class, this method answers by calling the X program xsetroot(1) specifying the bitmap file that was contained in the instance variable pixmap_file. *Winterp* calls the xsetroot(1) program through the primitive system, which corresponds to the *Unix* subroutine system(3S):

```
;;;
;;; a method that calls the xsetroot(1) program to set background
;;;
(send Niels_Pixmap_Push_Button_Class :answer :xsetroot '()
      '(
        (system (format nil "xsetroot -bitmap ~A -fg Black -bg DimGrey"
                        pixmap_file))
        ))
```

The code below returns a stream of bitmap file names in a particular directory on my system.

We use the *Unix* popen(3s) routine to read the results of the *Unix* command ls(1), which returns to stdout a list of matching filenames in the shell created by popen(3s).[24] popen(3s) returns a FILE* that can be read by the *Xlisp* primitive read-line.

```
(setq ls_reader_pipe
      (popen "/bin/ls /usr/local/mayer/src/bitmaps/*.xbm"
             :direction :input))
```

The following code does the brunt of the work in the bitmap browser application. The do* form is essentially a do-loop. It will repeatedly use primitive read-line to retrieve a single file name from ls_reader_pipe that we created above. For each filename retrieved, we create an instance of XM_LABEL_GADGET_CLASS displaying the name of the bitmap file, followed by an instance of

[24] People familiar with popen(3S) will realize that the command /bin/ls could in fact be replaced by /bin/echo.

`Niels_Pixmap_Push_Button_Class` displaying the pixmap itself, and finally, we create an instance of `XM_SEPARATOR_GADGET_CLASS` to separate the buttons from one another inside the Row/Column widget that is managing them.

When `read-line` returns NIL, it means that we have hit the end of file in `ls_reader_pipe`. At this point we have created a browser of all the bitmaps in the directory specified above. To clean up, we call `pclose(3S)` to close the pipe opened by `popen(3S)` and then call `XtRealizeWidget()` to create the windows associated with the browser, which is displayed in Figure 4.

```
(do*
 (;; local do-loop variables with initialize and increment expressions.
  (file-name (read-line ls_reader_pipe) (read-line ls_reader_pipe))
  )
 ((null file-name) ;loop done when (read-line) returns NIL ==> EOF
  )
 (send XM_LABEL_GADGET_CLASS :new :managed
       rowcol_w
       :XMN_LABEL_TYPE :STRING
       :XMN_LABEL_STRING file-name)
 (send Niels_Pixmap_Push_Button_Class :new file-name :managed
       rowcol_w)
 (send XM_SEPARATOR_GADGET_CLASS :new :managed
       rowcol_w
       :XMN_SEPARATOR_TYPE :DOUBLE_LINE)
 )
;;;
;;; Cleanup
;;;
(pclose ls_reader_pipe)        ;close the pipe
(send toplevel_w :realize)     ;create the widgets windows
```

10. A Real Application — A Search Browser Based on the Unix Grep(1) Program.

In this section we create a Motif user interface to the *Unix* `grep(1)` program. This example continues to show off the power and expressivity of object oriented programming in Winterp-Lisp — we make extensive use of the *Xlisp* object system here. This simple but useful application searches a wildcarded set of files for a regular expression, displaying each matching line in a browser. When an item is browsed (via mouse double click) the file associated with the item is displayed, with the matching line displayed at the top of the file. See Figure 5 to see what the `grep(1)` browser application looks like.

As an introductory example of using the object system, we create a new class — `Grep-Item-Class`. Instances of the class correspond to a datum produced by the *Unix* `grep(1)` file searching program. `grep(1)` produces output consisting of file-names, line-numbers, and the instance of a regular expression match. Below, we create a new class containing those items as instance variables. The new class instance is created by sending the instance creation message `:NEW` to the class `Class`:

```
;; (SEND Grep-Item-Class :NEW) returns <grep-item> instance.
(SETQ Grep-Item-Class
      (SEND Class :NEW
            '(file-name line-num match-line)
            ))
```

Below, we define a method corresponding to the message `:display-string` — this method will be used by the search browser as a way to present the objects in the browser. We also define the accessor methods `:file-name` and `:line-num`.

```
;; (SEND <grep-item> :display-string) returns "<filename>: <match-line>"
```

```
(SEND Grep-Item-Class :ANSWER :display-string '()
      '(
        (FORMAT NIL "~A: ~A"
                file-name match-line)
        ))
;; (SEND <grep-item> :file-name) returns file-name (string) of <grep-item>
(SEND Grep-Item-Class :ANSWER :file-name '()
      '(
        file-name
        ))
;; (SEND <grep-item> :file-name) returns line-number (fixnum) of <grep-item>
(SEND Grep-Item-Class :ANSWER :line-num '()
      '(
        line-num
        ))
```

Now we add a new method on the class defined above, which reads a line of output from the *Unix* command `grep(1)` from a pipe, and puts the data into the instance variables of a `Grep-Item-Class` object.

```
;; (SEND <grep-item> :read-grep-info <pipe>) returns <grep-item> or NIL
;; if EOF. Ivars are initialized to data from a line of "grep -n" output,
;; which is of the format <filename>:<linenum>:<matching line>\n
(SEND Grep-Item-Class :ANSWER :read-grep-info '(pipe)
      '(
        (if (AND
             (SETQ file-name  (FSCANF-STRING pipe "%[^:]:"))
             (SETQ line-num   (FSCANF-FIXNUM pipe "%d:"))
             (SETQ match-line (FSCANF-STRING pipe "%[^\n]\n"))
             )
          SELF                          ;return SELF if successful
          NIL                           ;return NIL if hit EOF
          )
        ))
```

The Lisp function `grep`, below, is an example of using `Grep-Item-Class`. This function invokes `"grep -n <regexp> <wildcarded files>"` in a sub-shell, sending the output of the command to a pipe. An instance of `Grep-Item-Class` is created for each regular expression match found, with method `:read-grep-info` initializing the object to contain the data corresponding to one match. The function returns a list of objects corresponding to the matching items found by `grep(1)`. This function plays a key role in the upcoming search browser example application.

```
;; (grep <grep-arg-string>) returns list of <grep-item>.
;; <grep-arg-string> == "[flags] <regexp> <wildcarded files>"
(DEFUN grep (grep-arg-string)
  (DO*
   (;; loop variables, initializers, and increments.
    (fp (POPEN (STRCAT "grep -n " grep-arg-string " /dev/null")
               :DIRECTION :INPUT))
    (line (SEND (SEND Grep-Item-Class :NEW) :read-grep-info fp)
          (SEND (SEND Grep-Item-Class :NEW) :read-grep-info fp))
    (result '())                        ;init to an empty list
    )
   ;; loop test and return
   ((NULL line)                         ;:read-grep-info returns NIL on EOF
    (PCLOSE fp)                         ;close the pipe opened above
```

```
    (REVERSE result)                    ;return list of grep objects.
    )
   ;; loop body
   (SETQ result (CONS line result))     ;prepend grep-obj to list
   ))
```

As an example of using *Xlisp*'s object-oriented features with Motif widgets, we specialize Motif's text editor widget as a file-viewer via (trivial) subclassing. Instances of the new class, `Text-Viewer-Widget-Class`, display the file of a browsed item with the text scrolled to the appropriate line. We subclass the Motif XmText widget so that we can add a method `:find-file` which reads the specified file, displays it in the text widget, and scrolls the text to the appropriate line number. The class has an extra instance variable, `file-path` which stores the name of the file. This allows method `:find-file` to be more intelligent — if the same file is browsed multiple times, it will not be reread each time.

First, we define the new class by sending message `:NEW` to class `Class`, with arguments specifying the instance variables of the class, as well as the superclass. Then, we define the instance initializer method `:ISNEW`, which will automatically be called whenever new instances of the widget are created (by sending `:NEW` to the class). The initializer method sets the instance variable, then calls the superclass' initializer method to create the Motif text editor widget. Since `Text-Viewer-Widget-Class` is a specialized editor, the instance initializer hard-codes optional features of the editor such that all instances end up having scroll bars, display multiple lines of text, etc.

```
(SETQ Text-Viewer-Widget-Class
      (SEND Class :NEW
            '(file-path)                ;new instance vars
            '()                         ;no class vars
            XM_TEXT_WIDGET_CLASS))      ;superclass

;; (SEND Text-Viewer-Widget-Class :NEW <:MANAGED/:UNMANAGED> <parent> [resources...])
(SEND Text-Viewer-Widget-Class :ANSWER :ISNEW '(managed-kwd &rest args)
      '(
        (SETQ file-path "")    ;initialize instance var
        (APPLY 'SEND-SUPER     ;call superclass's init to create widget
               `(:ISNEW ,managed-kwd ;either :MANAGED or :UNMANAGED
                        :SCROLLED    ;force the editor to have scrollbars
                        ,@args       ;parent widget + optional arguments
                        :XMN_EDIT_MODE :MULTI_LINE_EDIT ;multi line for files
                        :XMN_EDITABLE NIL ;do not allow user to edit text.
                        ))
        ))
```

The method `:find-file` is defined next. Note that in *Xlisp* methods, the symbol `SELF` is bound to the object receiving the message. The messages `:SET_INSERTION_POSITION`, `:GET_INSERTION_POSITION`, `:SCROLL`, `:REPLACE`, `:SET_STRING`, `:ENABLE_REDISPLAY` and `:DISABLE_REDISPLAY` correspond to methods defined on the superclass `XM_TEXT_WIDGET_CLASS`. These are implemented in C by the Motif toolkit.

```
;; (SEND <textviewer> :find-file <filename> <linenum>)
(SEND Text-Viewer-Widget-Class :ANSWER :find-file '(filename linenum)
      '(
        (COND
         ((STRING= filename file-path)  ;if file already read into widget
          (SEND SELF :SET_INSERTION_POSITION 0) ;then make <linenum> top
          (SEND SELF :SCROLL (1- linenum))      ;by scrolling to it.
          )
         (T                                     ;else read file into widget.
          (LET* ((fp (OPEN filename :DIRECTION :INPUT))
```

```
                 insert-pos
                 text-line)
            (IF (NULL fp)
                (ERROR "Can’t open file." filename))

            (SEND SELF :SET_STRING "")          ;clear out old text
            (SEND SELF :DISABLE_REDISPLAY NIL)  ;show no changes till done
            (LOOP
             (IF (NULL (SETQ text-line (READ-LINE fp)))
                 (RETURN))
             (SETQ insert-pos (SEND SELF :GET_INSERTION_POSITION))
             (SEND SELF :REPLACE insert-pos insert-pos (STRCAT text-line "\n"))
             )
            (SEND SELF :SCROLL linenum)   ;make <linenum> top of screen
            (SEND SELF :ENABLE_REDISPLAY) ;now show changes...
            (CLOSE fp)
            (SETQ file-path filename)
            )))
        ))
```

The browser user-interface to `grep(1)` is made from the Motif list widget, which simply displays a list of (compound) strings as a series of selectable lines of text. Callbacks can be defined on the list widget such that when a line is selected one can retrieve the compound string that was browsed, or get back the position (index) of the item that was browsed. Both callback mechanisms require the programmer to maintain an external association between the browser objects and their textual representation in the list widget. By subclassing the list widget, we can create an interface that hides the mechanism for associating objects with the strings representing them. As an additional feature, the abstraction enables the browser to display an arbitrary collection of objects, as long as the objects respond to a simple ‘‘protocol’’ — objects receiving the message `:display-string` must return a textual representation of the object.

The subclassed list widget adds a new instance variable `items` which holds an array of objects presented by the browser. The browser will display the list of objects given as argument to the method `:set-browser-items` (the method also initializes `items`). When an item is selected via a callback, the position of the selected item is passed to method `:get-br-item-at-pos`, which returns the browsed object using an efficient lookup implemented by array indexing.

```
(SETQ List-Browser-Widget-Class
      (SEND Class :NEW
            ’(items)                     ;new instance vars
            ’()                          ;no class vars
            XM_LIST_WIDGET_CLASS))       ;superclass

(SEND List-Browser-Widget-Class :ANSWER :set-browser-items ’(items-list)
      ’(
        (LET* (
               (items-end-idx (LENGTH items-list))
               (display-items (MAKE-ARRAY items-end-idx)))

          ;; initialize the ’items’ instance variable so that it
          ;; holds all the BROWSER_OBJECTs passed in <items-list>
          (SETQ items (MAKE-ARRAY items-end-idx)) ;create the array
          (DO (                          ;copy elts from list to array
               (i    0          (1+ i))
               (elts items-list (CDR elts)))
              ;; loop till no more elts
              ((NULL elts))
```

```
                ;; loop body
                (SETF (AREF items i) (CAR elts)) ;set items[i]
                (SETF (AREF display-items i)     ;set display-items[i]
                      (SEND (CAR elts) :display-string))
                )
            ;; initialize the widget, passing in the browser items.
            (SEND SELF :SET_VALUES
                  :XMN_ITEMS      display-items
                  :XMN_ITEM_COUNT items-end-idx
                  )
            )
        ))
(SEND List-Browser-Widget-Class :ANSWER :get-br-item-at-pos '(position)
      '(
        (AREF items (1- position))
        ))
```

The user interface for the search browser application is created by defining a hierarchy of widget-objects. Parent widgets manage the geometries of their children, so the resulting interface is a nesting of windows corresponding to the parent-child relationships created. Refer to Figure 5 to see the graphics produced by the code below.

In the UI code below, we first create an instance of `TOP_LEVEL_SHELL_WIDGET_CLASS` which represents the outermost window of the application — its geometry is managed by the X window manager, allowing the user to move, resize, and iconize the window. Inside that, we create an instance of `XM_PANED_WINDOW_WIDGET_CLASS`: this widget divides up the space of the toplevel window into a series of vertical panes; the sizes of the panes can be adjusted by moving their resize handles with the mouse. The first pane is an instance of `XM_ROW_COLUMN_WIDGET_CLASS`, a manager widget which lays out the controls for the search browser in the following horizontal sequence: (1) a pushbutton, `doit-button-w`; (2) a text label "Search for string:"; (3) a single-line text editor, `search-editor-w`, into which the user types the regular expression for the search; (4) a label "From Files:"; and (5) a single-line text editor, `files-editor-w`, into which the user types the wildcarded filenames to search. The pane below the controlpanel is `browser-w`, an instance of the `List-Browser-Widget-Class`. The pane below the browser is `filename-label-w`, which displays the name of the file being browsed. And finally, the last pane is `viewtext-w`, an instance of `Text-Viewer-Widget-Class` which is used to display the browsed files. The keyword `:SCROLLED` used in creating the text viewer and browser widgets mean that these widgets are created with scrollbars.

```
(LET (top-w paned-w controlpanel-w)
  (SETQ top-w
        (SEND TOP_LEVEL_SHELL_WIDGET_CLASS :NEW
              :XMN_TITLE "Grep Browser"
              :XMN_ICON_NAME "Grep Browser"
              ))
  (SETQ paned-w
        (SEND XM_PANED_WINDOW_WIDGET_CLASS :NEW :MANAGED
              top-w                    ;the only child of the toplevel window
              ))
  (SETQ controlpanel-w
        (SEND XM_ROW_COLUMN_WIDGET_CLASS :NEW :MANAGED
              paned-w                  ;the first child of the paned window
              :XMN_ORIENTATION :HORIZONTAL
              :XMN_PACKING :PACK_TIGHT
              ))
  (SETQ doit-button-w
```

```
      (SEND XM_PUSH_BUTTON_WIDGET_CLASS :NEW :MANAGED
            controlpanel-w            ;the first child of the controlpanel
            :XMN_LABEL_STRING "Do Search"
            ))
(SEND XM_LABEL_WIDGET_CLASS :NEW :MANAGED
      controlpanel-w                  ;the second child of the controlpanel
      :XMN_LABEL_STRING "Search for string:"
      )
(SETQ search-editor-w
      (SEND XM_TEXT_WIDGET_CLASS :NEW :MANAGED
            controlpanel-w            ;the third child of the controlpanel
            :XMN_EDIT_MODE :SINGLE_LINE_EDIT
            ))
(SEND XM_LABEL_WIDGET_CLASS :NEW :MANAGED
      controlpanel-w                  ;the fourth child of the controlpanel
      :XMN_LABEL_STRING "From Files:"
      )
(SETQ files-editor-w
      (SEND XM_TEXT_WIDGET_CLASS :NEW :MANAGED
            controlpanel-w            ;the fifth child of the controlpanel
            :XMN_EDIT_MODE :SINGLE_LINE_EDIT
            ))
(SETQ browser-w
      (SEND List-Browser-Widget-Class :NEW :MANAGED :SCROLLED
            paned-w                   ;the second child of the paned window
            :XMN_VISIBLE_ITEM_COUNT 20
            ))
(SETQ filename-label-w
      (SEND XM_LABEL_WIDGET_CLASS :NEW :MANAGED
            paned-w                   ;the third child of the paned window
            :XMN_LABEL_STRING "None"
            ))
(SETQ viewtext-w
      (SEND Text-Viewer-Widget-Class :NEW :MANAGED
            paned-w                   ;the fourth & final child of paned window
            :XMN_HEIGHT 400
            ))
(SEND top-w :REALIZE)                 ;create the toplevel window & children
;;
;; now that the widgets have been created, we know their sizes; set constraint
;; resources on the controlpanel and the filename label widget so that the paned
;; window widget managing their geometries won’t let them be resized (this
;; removes the "resize handles" for these widgets).
(LET (h)
     (SEND controlpanel-w :GET_VALUES :XMN_HEIGHT ’h)
     (SEND controlpanel-w :SET_VALUES
           :XMN_PANE_MAXIMUM h
           :XMN_PANE_MINIMUM h))
(LET (h)
     (SEND filename-label-w :GET_VALUES :XMN_HEIGHT ’h)
     (SEND filename-label-w :SET_VALUES
           :XMN_PANE_MAXIMUM h
           :XMN_PANE_MINIMUM h))
)
```

Figure 4. The graphics produced by the *Winterp* example code.

Figure 5. The *Winterp* `grep(1)` browser example application.

Calling function do-grep-search runs grep(1), searching for the regular expression in search-editor-w from the list of wildcarded files specified in files-editor-w. The list of objects returned by grep is then displayed in the browser widget:

```
(DEFUN do-grep-search ()
  (SEND browser-w :set-browser-items
        (grep (STRCAT
               "'"                                 ;protect regexps from shell
               (SEND search-editor-w :GET_STRING) ;get regexp for search
               "' "
               (SEND files-editor-w :GET_STRING)) ;wildcarded files to search
              )))
```

We complete the user interface by attaching programmatic actions — callbacks — to the widget-objects created above. First, we want the pushbutton associated with symbol doit-button-w to initiate searches when it is pressed by calling do-grep-search.

```
(SEND doit-button-w :ADD_CALLBACK :XMN_ARM_CALLBACK '()
      '(
        (do-grep-search)
        ))
```

We now attach a callback to the browser widget — when a browser-item is double clicked, symbol CALLBACK_ITEM_POSITION gets bound to the position of the object in the browser. The position is passed to :get-br-item-at-pos which returns the browsed item. The file-name and line-number data from the grep-item is retrieved and used to display the file at the correct line number in the text viewer widget.

```
(SEND browser-w :ADD_CALLBACK :XMN_DEFAULT_ACTION_CALLBACK
      '(CALLBACK_ITEM_POSITION) ;bound to the position of the item selected
      `(
        (LET* ((browsed-object
                (SEND ,browser-w :get-br-item-at-pos CALLBACK_ITEM_POSITION))
               (filename
                (SEND browsed-object :file-name))
               (linenum
                (SEND browsed-object :line-num))
               )
          (SEND ,filename-label-w :SET_VALUES :XMN_LABEL_STRING filename)
          (SEND ,viewtext-w :find-file filename linenum)
          ))
      )
```

Finally, to make the user-interface easier to use, we note that after entering text into one of the single-line text editor widgets, it is more natural to hit the return-key to initiate a new search. Otherwise one has to move from the keyboard to the mouse just to click the "Do Search" button. The following Xtoolkit translation entries provide the binding between hitting the return-key and calling the do-grep-search function that initiates a new search.

```
(SEND search-editor-w :OVERRIDE_TRANSLATIONS
      "<Key>Return: Lisp(do-grep-search)")
(SEND files-editor-w  :OVERRIDE_TRANSLATIONS
      "<Key>Return: Lisp(do-grep-search)")
```

10.1. A Note on Programming Style.

The examples presented in this section do not necessarily represent good Motif or *Winterp* programming style — the style has been relaxed and simplified for the purpose of exposition.

First, we have used global variables throughout. For these examples, such usage makes sense because that is the way one might use Lisp in interactively prototyping an application — try bits of functionality out piece by piece until one has come up with functionality that deserves to be encapsulated. Once the prototype works, one may want to turn the code into a function, or encapsulate it within an *Xlisp* object. In that way, the reliance on global variables will be removed.

Second, the examples above hard-code a number of X resources that should be specifiable in the X resource database — either in .Xdefaults or in the application defaults files. For the purposes of these examples, it would have been confusing to specify these resources separately from the code. Again, since the above code is prototype code, it is perfectly justifiable to hard-code resources until one has settled on which parameters of customization should and should not be accessible via the X resource database.

11. Future Directions

Winterp's features make it a good platform for a number of interesting experiments in UIMS and ‘‘direct manipulation builder’’ technologies. Because *Winterp* is currently a language-based builder, it would make an ideal platform for the development of a ‘‘two view’’ builder that would allow application prototyping via programmatic or visual manipulation of UIs.

Winterp was designed to support the addition of direct manipulation interface building capabilities. We envision that a widget ‘‘palette’’ could be built to allow the interactive selection and placement of widgets within an interface. Another useful feature would be a resource browser which would be used to display and edit the resources associated with any widget in the system. Extending *Winterp*'s widget-object based interfaces so that they are ‘‘self describing’’ would be a very elegant way of dumping out a programmatic user-interface specification after it has been built interactively and would provide the basis for a ‘‘two view’’ approach to interface building.

One of the areas where current direct-manipulation and ‘‘What You See Is What You Get’’ (WYSIWYG) interface builders fail is in allowing end-user customization without having to include the builder in the deliverable. The challenge here is to allow the traditional X resources mechanism to customize sizes, fonts, spacing, line-widths, etc., while still maintaining a facsimile of the static UI layout intended by the application designers. Traditional Xtoolkit-based applications do not suffer from such problems because their interfaces are laid-out via the constraints provided by geometry management widgets. With builders, such constraint management is hard to specify graphically, so widget positioning and sizes must be hard-coded. What is needed is the ability to provide an explicit, user-manipulable interface to the constraint-based language that is implicit in the Xtoolkit's manager widgets. This is an open and active research area.

Robert Leichner of HP Labs has already provided a novel UIMS structure for *Winterp* by building a general-purpose event-driven recursive state machine as a high level means of describing how application state interacts with the UI. This UIMS architecture is being used in an ongoing project to support collaboration among distributed workgroups via the use of multimedia (audio and video). *Winterp* and the state-machine based UIMS are being used as the basis for a media management toolkit which controls experimental multimedia hardware residing on the workstation. The *Strudel* project is also experimenting with a variety of UIMS approaches in order to come up with a high-level description of e-mail based forms.

In order to ease application development and simplify ‘‘hybrid programming’’, we are considering the addition of a dynamic loader to allow new C-implemented primitives to be loaded into a running application. It may also be useful to dynamically load widget code on demand, instead of always having it compiled in to the application.

12. Conclusion

We believe that *Winterp* provides an excellent, practical development and delivery environment for extensible Motif-based applications. If this paper has piqued your interest in *Winterp*, you may obtain the current source, documentation, and examples via anonymous ftp from host `export.lcs.mit.edu`: in directory `contrib/winterp` you will find the `compress(1)'d tar(1)` file `winterp-`

`<latestversion>.tar.Z`[25]. If you do not have Internet access you may request the source code to be mailed to you by sending a message to `winterp-source@hplnpm.hpl.hp.com` or `hplabs!hplnpm!winterp-source`.

There is also a mailing list for *Winterp*-related announcements and discussions. To get added to the list, send mail to `winterp-request@hplnpm.hpl.hp.com` or `hplabs!hplnpm!winterp-request`.

13. Acknowledgements

I would like to thank my team-mates Allan Kuchinsky, Allan Shepherd, and Bob Leichner for being highly supportive early users of *Winterp* and giving feedback on problems and limitations of the design. I would also like to thank Nancy Kendzierski, manager of HP Labs' Human-Computer Interaction Department for providing the support for the development of *Winterp* and *Strudel*. Also, Nancy Kendzierski, Jim Miller, and Allan Kuchinsky provided helpful comments on this paper.

Doug Young deserves special thanks for answering lots of ‘‘stupid questions’’ I had in working with the Motif and HP widgets and for providing early copies of his excellent book on programming with Xt and Motif [Young90]. And finally, many thanks are due to David Betz for making *Xlisp* publicly available.

14. Addendum — Instructions for Obtaining WINTERP 1.12 via anonymous ftp.

The following instructions give the details of how to obtain *Winterp* sources, examples, and documentation via anonymous ftp on a *Unix* workstation.

```
Note: In the output below, your input is denoted by '^^^^^^^^^^')

hplnpm-17-/tmp> ftp export.lcs.mit.edu  (HP-ites, do ftp hplnpm.hpl.hp.com)
                ^^^^^^^^^^^^^^^^^^^^^^               ^^^^^^^^^^^^^^^^^^^^^

        [...]
Name (hplnpm.hpl.hp.com:mayer): anonymous
                                ^^^^^^^^^

Password (hplnpm.hpl.hp.com:anonymous): <anypassword you want here>
                                        ^^^^^^^^^^^^^^^^^^^^^^^^^^^

331 Guest login ok, send ident as password.
230 Guest login ok, access restrictions apply.
ftp> cd contrib/winterp   (HPites should do cd pub)
     ^^^^^^^^^^^^^^^^^^                     ^^^^^^

200 CWD command okay.

ftp> type image
     ^^^^^^^^^^

200 Type set to I.

ftp> get winterp-1.12.tar.Z
     ^^^^^^^^^^^^^^^^^^^^^^

200 PORT command okay.
150 Opening data connection for winterp-1.12.tar.Z (15.255.176.225,3988) (1414493 bytes
226 Transfer complete.
1414493 bytes received in 690.63 seconds (1.96 Kbytes/sec)

ftp> quit
     ^^^^

221 Goodbye.
```

[25] As of this writing, <latestversion> ≥ 1.12 .

```
hplnpm-18-/tmp> zcat winterp-1.12.tar.Z | tar xvf -
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

        [... lengthy list of files from tar ...]

hplnpm-19-/tmp> rm winterp-1.12.tar.Z
                ^^^^^^^^^^^^^^^^^^^^^^
```

```
   * For compilation tips, take a look at winterp-1.12/doc/COMPILING

   * For running hints, take a look at winterp-1.12/doc/RUNNING

   * For known (Motif 1.1 only) bugs, see winterp-1.12/doc/BUGS

   * For information on the examples, look at winterp-1.12/examples/README

   * For general information about WINTERP, see winterp-1.12/doc/winterp.doc
```

15. References

[Betz89] David Michael Betz. *XLISP: An Object-oriented Lisp (version 2.1)*. Unpublished documentation accompanying the public domain *Xlisp* software release. David Michael Betz, P.O. Box 144, Peterborough, NH 03458, April, 1989. Note: this documentation is included in the *Winterp* source distribution.

[Borenstein88] Nathaniel S. Borenstein and Chris A. Thyberg. Cooperative Work in the Andrew Message System. In proceedings *Conference on Computer-Supported Cooperative Work*, 1988.

[Bourne90] Philip E. Bourne and Lawrence S. Shapiro. Developing with DECwindows. *DEC Professional*, vol. 9, no. 2, pp. 36-44, February 1990.

[Comer86] D. Comer and L. Peterson. Conversation-Based Mail. *ACM Transactions on Computer Systems*, vol. 4, no. 4, pp 299-319, November 1986.

[Conk87] Jeff Conklin and Michael Begeman. gIBIS: A Hypertext Tool for Team Design Deliberation. In proceedings *Hypertext '87 Papers*, November 1987.

[Creech87a] Michael Creech. Lisp-C Evaluation Final Report. Internal Technical Report STL-TM-87-18, Hewlett-Packard Laboratories, Software Technology Lab, July 9, 1987.

[Creech87b] Michael Creech, Scott Marovich and Niels Mayer. Lisp-C Evaluation Meeting Notes. Internal Technical Report STL-TM-87-17, Hewlett-Packard Laboratories, Software Technology Lab, July 22, 1987.

[Dollimore89] Jean Dollimore and Sylvia Wilbur. Experiences in building a configurable {CSCW} system. In proceedings *1st European Conference on CSCW*, pp 215-225, September 1989.

[Fikes82] R.E. Fikes. A Commitment-Based Framework for Describing Informal Cooperative Work. *Cognitive Science*, vol. 6, no. 4, pp. 331-347, 1982.

[Heimburger] Olaf Heimburger. Elche Im Winter — Interaktive X-Applicationbuilder unter Lisp — Elk und WINTERP. *iX*, July 1991, pp 64-68.

[Holt81] A.W. Holt and P.M. Cashman. Designing Systems to Support Cooperative Activity: An Example. In *Proceedings of Compsac 81. IEEE Computer Society's Fifth International Computer Software And Applications Conference*, pp 18-91, November 1981.

[Kaplan90] Simon Kaplan. COED: A Conversation-oriented tool for coordinated design work. In *Proceedings of IFIP International Workshop on Human Factors in Information Systems*, in print, June 1990.

[Lai88] Kum-Yew Lai and Tom Malone. Object-Lens: A Spreadsheet for Cooperative Work. In proceedings *Conference on Computer-Supported Cooperative Work*, September 1988.

[Malone86] Tom Malone, K. Grant, K. Lai, R. Rao, and D. Rosenblitt. Semi-structured Messages are Surprisingly Useful for Computer-Supported Coordination. In proceedings *Conference on Computer-Supported Cooperative Work*, pp 102-114, December 1986.

[Mayer90a] Niels P. Mayer, Allan W. Shepherd and Allan J. Kuchinsky. Winterp: An object-oriented, rapid prototyping, development and delivery environment for building extensible applications with the OSF/Motif UI Toolkit. In Proceedings *Xhibition '90, X Window System and Open Systems Technical Conference*, San Jose, CA, May 1990, pp 49-64.

[Mayer90b] Niels P. Mayer, Allan W. Shepherd and Allan J. Kuchinsky. The WINTERP Widget INTERPreter — An Application Prototyping and Extension Environment for OSF/Motif. In Proceedings *X Into The Future, The European X Users Group Autumn Conference 1990*, Surrey, UK, September 1990, pp. 33-55.

[Mayer91] Niels P. Mayer. The WINTERP Widget INTERPreter — A Lisp Prototyping and Extension Environment for OSF/Motif-based Applications and User-Interfaces. *Lisp Pointers, ACM SIGPLAN*, Volume IV, Number 1, pp 45-60.

[Myers89] Brad A. Myers. Tools for Creating User Interfaces: An Introduction and Survey. *IEEE Software*, vol. 6, no. 1, pp. 15-23, January 1989.

[OSF90] Open Software Foundation. *OSF/Motif Series* (5 Volumes): *Motif Style Guide; Programmer's Guide; Programmer's Reference; User's Guide; Application Environment Specification; User Environment Volume*. Prentice-Hall, 1990.

[Reichman85] Rachel Reichman. *Getting Computers to Talk Like You and Me*. The MIT Press, 1985.

[Rosenberg88] Jarrett Rosenberg (moderator), Ralph Hill, Jim Miller, Andrew Schulert, and David Shewmake (panelists). UIMSs: Threat or Menace? In *Human Factors in Computing Systems*, SIGCHI '88, Washington, D.C., May 1988, pp. 197-200.

[Rose86] M.T. Rose and J.L. Romine. *The Rand MH message handling system: User's manual, UCI Version 6.5*. University of California, December 1986.

[Schulert88] Andrew Schulert and Kate Erf. Open Dialogue: Using an extensible retained object workspace to support a UIMS. In proceedings *USENIX C++ Workshop*, Denver, Colorado, 1988.

[SEI89] Software Engineering Institute. *Serpent Overview*. Technical Report CMU/SEI-89-UG-2, Carnegie Mellon University, Software Engineering Institute, August 1989.

[Searle76] John R. Searle. A Taxonomy of Illocutionary Acts. In K. Gunderson, editor, *Language, Mind and Knowledge* University of Minnesota Press, 1976.

[Shepherd90] Allan Shepherd, Niels Mayer, and Allan Kuchinsky. Strudel: An Extensible Electronic Conversation Toolkit. In proceedings *Conference on Computer-Supported Cooperative Work*, Los Angeles, October 1990, pp. 93-104.

[Stallman87] Richard M. Stallman. *GNU Emacs Manual*. Free Software Foundation, 675 Massachusetts Ave., Cambridge, MA 02139, 1987.

[Sulonen90] Reijo Sulonen and Panu Pietikainen. Forget-Me-Not — Controlling intercompany operations by intelligent mail. In *Proceedings 23rd Annual Hawaii International Conference on Systems Sciences*, pp. 428-435, 1990.

[Trigg86] Randall Trigg, Lucy A. Suchman, and Frank G. Halasz. Supporting Collaboration in NoteCards. In *Conference on Computer-Supported Cooperative Work*, December 1986.

[Winograd86] Terry Winograd and Fernando Flores. *Understanding computers and cognition, A new foundation for design*. Ablex, Norwood New Jersey, 1986.

[Winograd87] Terry Winograd. A Language/Action Perspective On The Design Of Cooperative Work. Technical Report CSLI-87-98, Stanford University, 1987.

[Young90] Douglas A. Young. *The X Window System: Programming and Applications With Xt, OSF/Motif Edition*. Prentice Hall, 1990.