Int. J. Man–Machine Studies (1993) **39**, 391–425

Supporting command reuse: mechanisms for reuse

SAUL GREENBERG
Department of Computer Science, The University of Calgary, Calgary, Alberta T2N 1N4, Canada

IAN H. WITTEN
Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand

(Received 18 December 1989 and accepted in revised form 1 February 1993)

Reuse facilities help people to recall and modify their earlier activities and re-submit them to the computer. This paper examines such mechanisms for reuse. First, guidelines for building reuse facilities are summarized. Second, existing reuse facilities are surveyed under four main headings: history mechanisms, adaptive systems, programming by example, and explicit customization. The first kind relies on temporally ordered lists of interactions, the second builds statistical dynamic models of past activities and uses them to expedite future interactions, the third collects and generalizes more extensive sequences of activities for future reuse, while in the fourth the user explicitly collects items of interest. Third, the paper presents WORKBENCH, a reuse facility that uses an empirically-derived history system as a way of capturing and organizing one's situated activities. An appendix reports a study of a widely-available history system, the UNIX *csh*, and explains why it is poorly used in practice.

1. Introduction

Users often repeat activities that they have previously submitted to the computer. In command-driven systems, these activities comprise both individual commands and command lines complete with filenames and options. Likewise, people repeat the ways they traverse paths in menu hierarchies, select icons in graphical interfaces, navigate through file hierarchies, and choose document nodes in hypertext systems. Yet recalling the original activity can be both difficult and tedious. For example, mental contexts must be re-created for complex activities, command syntax or search paths must be remembered, input lines retyped, icons found, directories and files opened, and so on.

There is potential for a well-designed *reuse facility* to alleviate the problems of activity reformulation by keeping previous activities that are likely to be repeated ready to hand. Interactive reuse facilities allow users to recall, modify, and re-submit previous entries to computers. Clearly, this is only effective if recalling old entries is easier for the user (cognitively and physically) than constructing new ones. Salient differences between reuse facilities arise from their choice of pertinent offerings and from the operations they provide to manipulate them.

Reuse facilities abound in everyday life. A cook can flag preferred recipes in a cookbook with bookmarks. "Adaptive" marking also occurs because the pages fall open to popular locations through wear of the binding. An audiophile returns

0020-7373/93/090391 + 35$08.00/0

records to the top of the pile so that popular ones tend to remain near the top. A carpenter's workbench has a work surface large enough to keep recently used tools close by.

Reuse facilities abound for computers as well. They fall into four classes. The first comprises *history mechanisms* that let users manipulate a time-ordered list of their previous interactions. Items are retrieved and selected through textual syntactic constructs, by pointing to menu items and buttons, or by editing dialogue transcripts. We include in this category history to support the way people navigate to items in networks, as in hypertext, file hierarchies, and menu hierarchies. The second class, *adaptive systems,* uses statistical models of previous inputs to predict upcoming ones. Examples are hierarchical menus that are dynamically configured to give preference to high frequency items, and text predictors that maintain statistical models of the text entered so far to predict future submissions. The third class, *programming by example,* addresses the reuse and generalization of long input sequences. The final class, *explicit customization,* provides tools for users to tailor their working environment to their own liking.

This paper examines existing reuse facilities, beginning with a summary of important design guidelines culled from the literature (Section 2). Sections 3–6 survey schemes for reuse under the four class headings mentioned in the previous paragraph, and illustrate how the diversity of techniques fit the design guidelines. Section 7 describes WORKBENCH, a novel reuse facility that follows the metaphor of a handyman's workbench. It incorporates many of the reuse design guidelines. Through direct manipulation editing, WORKBENCH allows the user to pick items off a history list and stash them temporarily on a visible tool shelf or place them semi-permanently within a drawer of a tool cabinet. Finally, an appendix shows the results of a study of 168 users using one particular history system, the UNIX *csh.* By comparing actual usage to the theoretical potential of reuse, we see how poorly *csh* performs and suggest how such history systems can be improved.

2. Designing reuse facilities: a summary of guidelines

This section summarizes guidelines applicable to the design of reuse facilities, listed in Figure 1. There are three fundamental design requirements: a user's previously submitted activities should be available for recall; activities should be grouped into high-level task sets and switching between these sets should be supported; and end-user customization of activities and task sets should be permitted. These guidelines are derived from our own empirical studies (Greenberg, 1993; Greenberg & Witten, 1988, 1993), from usage observations of other researchers (Bannon *et al.*, 1983; Cypher, 1986; Card & Henderson Jr, 1987) and from design insights provided by existing systems (as surveyed in this paper). This section provides only brief descriptions of the guidelines; readers are referred to the original papers for full detail and argumentation.

2.1. A USER'S PREVIOUSLY SUBMITTED ACTIVITIES SHOULD BE AVAILABLE FOR RECALL

A major theme of most reuse facilities is to allow users to repeat a single activity identical or similar to one invoked previously. The minimum requirement for a

1. **A user's previously submitted activities should be available for recall** (see Greenberg, 1993; Greenberg & Witten, 1993).
   a) Maintain records of activities.
   b) Users should be able to recall previous activities from these records.
   c) It should be cheaper, in terms of mechanical and cognitive activity, for users to recall items than to re-enter them.
   d) A selectable history list of the previous 5–10 submissions provides a reasonable working set of candidates for reuse.
   e) Other strategies for presenting the history list, particularly pruning duplicates and further hierarchical structuring, increase the probability of it containing the next submission.
   f) History is not effective for all possible recalls because it lists only a few previous events. Alternative strategies must be supported.
   g) Events already recalled through history by the user should be easily reselectable (see Appendix 1).
   h) Allow activities to be represented by meaningful symbols (such as labels or icons).
   i) Allow activities to be annotated with extra information (e.g. help descriptions).
2. **Support grouping of activities into high-level tasks, and switching between tasks** (see Bannon *et al.*, 1983; Cypher, 1986; Card & Henderson Jr, 1987; Greenberg, 1993).
   a) Allow linear sequences of activities to be grouped into a procedure.
   b) Allow functional groupings of related activities.
   c) Support suspension and resumption of activities.
   d) Reduce a user's mental load when switching tasks:
      • it should be easy to reacquire one's mental task context;
      • task switching should be fast;
      • task resumption should be fast.
   e) Allow interdependencies between task sets and between items in different task sets.
   f) Provide multiple perspectives on the work environment.
3. **Allow end-user customization of workspaces** (see MacLean *et al.*, 1990; Greenberg, 1988, 1993).
   a) Allow new workspaces to be created and old ones deleted.
   b) Allow existing workspaces to be modified.
   c) Allow elements within a workspace to be added, modified, or deleted.
   d) Keep the mechanical and cognitive overhead of managing workspaces to a minimum.
   e) Allow previously executed activities to be moved from a reuse facility to the task sets contained by the workspace.
   f) Workspaces, task sets and activities should be shareable by the community at large.

FIGURE 1. Summary of guidelines for reuse facilities.

useful system is that the interface must supply users with good candidates for reuse, and that recall should be less work than re-entering activities (Figure 1, guidelines 1a–c) (Greenberg & Witten, 1993).

There are several reasonable strategies for allowing activities to be reused, as outlined in Sections 3–6. Items can be recorded into a time-ordered history list and presented to the user through various presentation methods [as evaluated in the companion paper to this one, Greenberg and Witten (1993); see Figure 1, guidelines 1d,e]. Alternatively, the system can use schemes other than recency to favour certain activities over others (e.g. adaptive methods, end-user customization; guideline 1f).

We have noticed that when people use history, the items they recall tend to be reused again and again (Appendix 1). These items should somehow get preferential treatment (guideline 1g).

Most of the approaches for reuse require that activities must be recorded in a workspace if they are to be reused. However, there is no real need to show these activities in their raw (and often ugly) syntactic form. Instead, meaningful symbols that remind users of the meaning behind an action can label an activity (guideline 1h). For example, the UNIX command

```
lpr -Palw3 -d galley.dvi
```

could be shown instead as the more meaningful (to the user, at least) "Print text galley". Similarly, symbols can represent other attributes associated with an activity. Each entry can be annotated with extra information, such as help text or a property sheet specifying new input parameters (guideline 1i). For example, the help string "Print the paper (a LaTeX document) onto the printer in the main office" may be attached to the above command line. Depending on how one selects the symbol, the activity may be executed, help text displayed, or even a property sheet raised for further clarification.

2.2. SUPPORT GROUPING OF ACTIVITIES INTO HIGH-LEVEL TASKS, AND SWITCHING BETWEEN TASKS

Activities are not necessarily independent of each other, but may be related in many ways. First, linear sequences of activities may represent the steps of a repeatable procedure. If the procedure can be captured, the user may recall the entire procedure in a single step (Figure 1, guideline 2a) (Witten, MacDonald & Greenberg, 1987).

Second, users may partition their actions and the objects they manipulate (such as files) into sets of goal-related tasks, each called a *task set*. Bannon *et al.* (1983) studied UNIX users, and noticed that people performing particular tasks would consistently use the same particular command lines. These differ from procedures as command lines did not always follow in the same order or frequency, and the activity selected at any moment from the task set seemed to depend on the user's particular circumstances. Also, other activities loosely related to the task set may be interposed from time to time.

Third, relations between activities may arise from the function they serve, rather than the particular task they address. Example functions are shaping text, orienting oneself in the environment, process management, printing to various devices, and so on (Hanson, Kraut & Farber, 1984). We believe it self-evident that users organize their activities in many (perhaps vague) ways throughout the computer dialogue. The implication is that the system should somehow group together those activities associated with a task or function, and present them to the user as a set (guideline 2b).

Tasks are not invoked sequentially, but are interleaved because the user switches, suspends and resumes his goals (Cypher, 1986). For example, a user may be "simultaneously" working on a document, reading some mail, chasing references, and so on. Because tasks are frequently interrupted, the system should save and restore the task state between excursions, and allow users to navigate easily between the different sets of activities associated with the tasks (guideline 2c). Since these task sets can act as visible place-holders to reduce one's mental load, task switching and resumption should be fast, and should recreate as much as possible of the user's mental context (Card & Henderson Jr, 1987; guideline 2d).

Of course, task sets are not necessarily independent from one another, and may be related in quite strong ways (perhaps as a goal/sub-goal relationship). Information in one workspace may be related to another, and the display should make such relations obvious. Also, items from one task set can be useful in others. Items could be shared among several tasks, and their visual presentation should be task-specific (Card & Henderson Jr, 1987). Interdependencies should be allowed between task sets and the items they contain (Bannon *et al.*, 1983; guideline 2e,f).

2.3. SUPPORT END-USER CUSTOMIZATION OF WORKSPACES

A *workspace* is a software tool that not only keeps activities available for reuse, but allows them to be organized into related sets. Yet who actually builds and maintains workspaces—the overall structure, the activities included, and the symbols chosen? From a population perspective, designers can create only a few default workspaces, as there is little overlap between what individuals actually do (Greenberg & Witten, 1993). Particular users have their own unique task sets, and no universal scheme can cater to individual idiosyncrasies. Ideally, when a need arises that is not addressed well by predefined workspaces, the system should try to create one for the user. However, this will be at best a close approximation to what the user really requires. Alternatively, each user should be able to create explicitly the workspace organization that best suits their needs (Figure 1, guideline 3a). As user needs, tasks and preferences change over time, the workbenches and their contents should be easily modifiable (guideline 3b,c).

Users, however, will not take advantage of a customizable workspace facility if it involves a significant overhead. The interface must therefore minimize the mechanical and cognitive overhead of manipulating workspaces (guideline 3d). But how can this be accomplished? Even with the best interface, consider the cognitive overhead of forming activities collected by a workspace. If users must anticipate what they are going to do, then the burden of collecting the appropriate activities into the workspace will be high. People may not know precisely what activities are required for their task. Those that are known must be composed, debugged, and tested to ensure that they perform correctly.

A better approach would have users create candidates for a workspace by recalling previous activities (MacLean *et al.*, 1990; Greenberg, 1988, 1993). By merging a reuse facility with a customizable workspace, and by allowing old activities to be changed into workspace items, considerable power can be gained. Users would not only be able to redo old actions but could also use the history list as the primary source of tried and tested candidates for their task sets. They could select, copy, and add them directly into their workspace (guideline 3e). The potential benefits are important. First, workspace items do not have to be anticipated. Instead, users can perform their tasks as normal and decide at any time to assemble the relevant previous activities that make up the task sets. Second, because these items are directly available, they are recalled rather than composed. Third, they have already been debugged and tested to some extent. Finally, interaction tedium is minimized, because modern techniques for selecting and transferring activities (the cut/copy/paste and drag/drop metaphors) take only a moment.

Finally, MacLean *et al.* (1990) argue that end-user customization will occur best within a community that has a culture of changing the workstation environment. Users not only create innovations useful to themselves, but may recognize when things are of value to the community at large. Specialists can reduce a non-programmer's burden by bundling up activities into a simple interface, and then passing it on. Even if the fit is not perfect, non-programmers may be able to modify the workspace to match their particular task by simple changes (guideline 3f).

The remainder of this paper examines particular systems built to support reuse. Generic design ideas of these facilities are abstracted, and (when necessary) related to the guidelines in Figure 1.

3. Reuse through history mechanisms

History mechanisms assume that the last few user submissions are good candidates to make available for reuse (see guidelines 1a–d). This notion of "temporal recency" is cognitively attractive because users generally remember what they have just entered and can predict the offerings the system will make available to them. Little time is wasted searching in vain for missing items.

By far the most common reuse facility available, history mechanisms are implemented across diverse systems in a variety of flavours. Four fundamentally different interaction styles can be identified: glass teletypes, graphical selection, editing transcripts, and navigational traces. The first three pertain to command-line interfaces, while the last applies to systems in which users traverse some information structure.

3.1. HISTORY IN GLASS TELETYPES

Traditional command-line dialogues were created for the teletype, and as a result many of today's VDUs are still a fixed viewport into a virtual roll of paper. Two history systems designed for these "glass teletypes" are the UNIX *csh* and the INTERLISP PROGRAMMER'S ASSISTANT. Both systems have users retrieve old commands by "history directives", themselves commands interpreted in a special way.

UNIX *csh* maintains an invisible record of user inputs, where every command is recorded in a numbered event list (Joy, 1980). Special syntactic constructs allow previous events to be partially or completed recalled, either by position on the list (relative or absolute) or by pattern-matching. The events can be viewed, edited, or re-executed. Although the set of predictions is in principle unbounded, in practice it is small, because users forget all but the last few items they have entered. While users may request a snapshot of the event list, they rarely do so due to the extra work and time involved.

Example Event List	
9	mail ian
10	emacs fig1 fig2 fig3
11	cat fig1
12	diff fig*

Examples and Results of History Uses		
User Input	Action	Description
!!	diff fig*	Redo the last event
!11	cat fig1	Redo event 11
!-2	cat fig1	Redo the second event from last
!mai	mail ian	Redo most recent event containing the prefix "mail"
!?ian?	mail ian	Redo most recent event containing the string "ian"
!! fig3	diff fig* fig3	Append "fig3" to the last event and redo
^diff^page	page fig*	Substitute "page" for "diff" in the last event and redo
!!:p	diff fig*	Print without executing the last event
page !10:1-2	page fig1 fig2	Include the 1st and 2nd argument of event 10 in expression

FIGURE 2. Examples of the UNIX *csh* history mechanism in use.

Figure 2 illustrates an event list (top box) and a few possibilities of *csh* history in use on the next event (bottom box). Inputs in the bottom left column are translated by *csh* to the actions shown in the middle, and the right column describes the semantics of the history directives. The syntax is quite arcane, and deters use of the more powerful features (Greenberg, 1988; Lee, 1988). Since the event list is generally invisible, it is difficult for the *csh* user to refer to any but the last few events (see Appendix 1).

Another functionally powerful history mechanism is the PROGRAMMER'S ASSISTANT, designed for the INTERLISP programming environment (Teitelman & Masinter, 1981; Xerox, 1985). Although INTERLISP is window-based, the top-level "Interlisp-D Executive" occupies a plain scrolling window where the user types list expressions (Figure 3). Historical events may be selected and processed by special command directives entered in this window. For example, the request

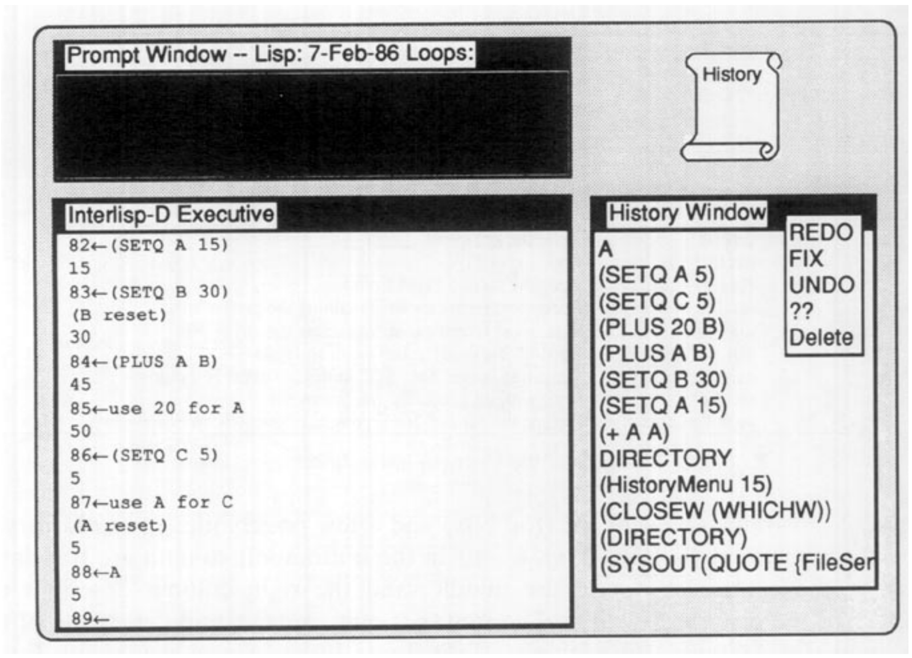
```
USE cons FOR setq IN -1
```

will replace "setq" by "cons" in the previous command. Through the history mechanism, users can retrieve and manipulate several events at a time, specify iteration and conditionals, edit items, undo effects of previous entries, and so on. Figure 3 shows a sample dialogue in the window labelled "Interlisp-D Executive", where events 85 and 87 make use of the PROGRAMMER'S ASSISTANT (the other statements are normal lisp expressions). As with *csh*, the system maintains a true time-ordered event list—neither duplicates nor erroneous statements are pruned.

In general, history in glass teletypes follows only the smallest subset of the guidelines listed in Figure 1: records of activities are kept, and users can recall the last few submissions through memory. However, the mechanical overhead is great (complex recalls demand much typing) and cognitive overhead is obviously high. This violates guideline 1c. We believe this is the reason why UNIX *csh* history is so poorly used, as documented in Appendix 1.

3.2. HISTORY THROUGH GRAPHICAL SELECTION

Present-day terminals allow text to be placed anywhere on the screen, and high-resolution bitmapped workstations with a pointing device (usually a mouse) are



```
82←(SETQ A 15)
15
83←(SETQ B 30)
(B reset)
30
84←(PLUS A B)
45
85←use 20 for A
50
86←(SETQ C 5)
5
87←use A for C
(A reset)
5
88←A
5
89←
```

```
A
(SETQ A 5)
(SETQ C 5)
(PLUS 20 B)
(PLUS A B)
(SETQ B 30)
(SETQ A 15)
(+ A A)
DIRECTORY
(HistoryMenu 15)
(CLOSEW (WHICHW))
(DIRECTORY)
(SYSOUT(QUOTE {FileSer
```

FIGURE 3. A portion of the INTERLISP-D environment, showing HISTMENU in use.

common. Interaction styles have progressed accordingly, from text-oriented menus and forms to mouse-oriented graphical systems running within windows (Witten & Greenberg, 1985). History mechanisms have been extended to present a (possibly transient) menu of previous events, where items are selected and manipulated with the pointing device. In contrast to previous history mechanisms that relied heavily on a user's memory of submissions and their relative ordering, predictions are now offered by presenting them explicitly on the screen. Because selection is usually just a matter of pointing to the desired item, the syntactic knowledge required by the user is kept to a minimum. The mechanical and cognitive effort of recall is fairly low (guideline 1c).

One example is HISTMENU, which provides a limited yet simple way of accessing and modifying the PROGRAMMER'S ASSISTANT history list (Bobrow, 1986). Figure 3 illustrates its use. Commands entered to the "Interlisp-D Executive" window are recorded on the history list, part of which is displayed in the "History Window" (by default, the last 50 items are shown). Although the internal list is updated on every command, the window is only redrawn when the user explicitly requests it. When pointed at with a mouse, items (which may not fit completely in the narrow History Window) are printed in the "Prompt Window". Any entry can be re-executed by selecting it. Moreover, a pop-up menu allows limited further action: items can be "fixed" (i.e. edited), undone, printed in full including additional detail (the "??"), or deleted from view. The History Window also has a shrunken form, as shown by the icon in the Figure.

MINIT is another graphical package that combines command processing and the history list into a single *window management window* (Barnes & Bovey, 1986). It

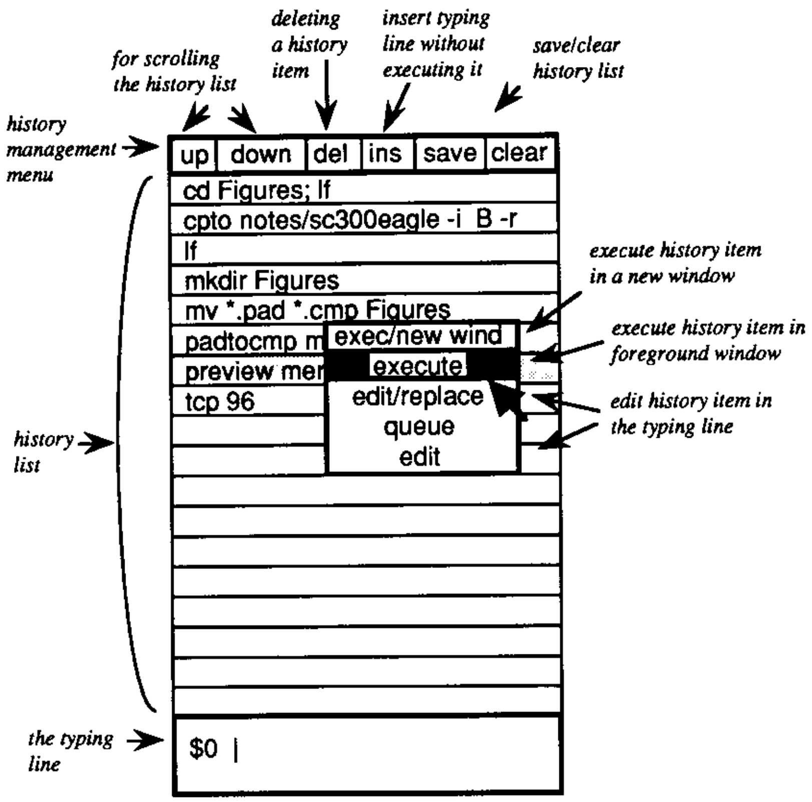


FIGURE 4. MINIT's *window management window*, redrawn from Barnes and Bovey (1986).

differs from other systems in that only through this window can users send commands to the other ones. The *window management window* is divided into three regions (Figure 4). The bottom region is an editable typing line in which commands are typed. Once entered, they are automatically added to the second region, that contains a scrollable history list. As with HISTMENU, the user may select items using a pointing device and control further action—for example, executing the item in various windows or inserting it into the typing line for further editing—with a pop-up menu. The region at the top of Figure 4 contains a history management menu. Options are available to:

- scroll the history list, clear it, or save it for future use;
- search for specific items (available through the typing line);
- delete specific items;
- insert text from the typing line into the history list without executing it.

Two more mechanisms complete MINIT's history management capabilities. First, the user can customize it to prevent short commands which are easily retyped from being added to the list. Second, history is viewable in either alphabetic or execution order. Duplicate lines are eliminated in both methods. In execution order, the user controls whether the original of a repeated command entry remains in its original position or is moved to the end of the history list.

It is less easy to provide a history facility for a graphical interface such as a painting or drawing program, and we are aware of only one system that comes close to offering such capabilities. CHIMERA adopts the metaphor of a "comic strip", a graphical record of the user's past activities that consists of a sequence of panels, each of which illustrates an important moment in a story (Kurlander & Feiner, 1990). Instead of showing miniatures, panels record just the objects being manipulated and the actions performed on them without unnecessary detail. This graphical history provides more power than just reuse, and it is far closer in spirit to a full undo, skip and redo facility (Vitter, 1984). The user can then expand a particular panel as necessary; delete, modify, undo and redo the actions it expresses; and even add new actions to the sequence.

How do these graphical systems fare against the guidelines in Figure 1? They definitely improve upon history through glass teletypes, for guideline 1c is now supported. It is far easier to recognize and select items in a menu than it is to recall them through memory and retype them. Also, all systems usually display at least 10 time-ordered submissions (guideline 1d). However, most waste screen real estate by displaying more than 10 items, for these add little to the overall probability that the next submission is on the menu (Greenberg & Witten, 1993). MINIT allows duplicates to be pruned from the history list, increasing the usefulness of the menu (guideline 1e). However, it also allows alphabetical ordering of items, which has been shown to perform very poorly in practice (Greenberg & Witten, 1993).

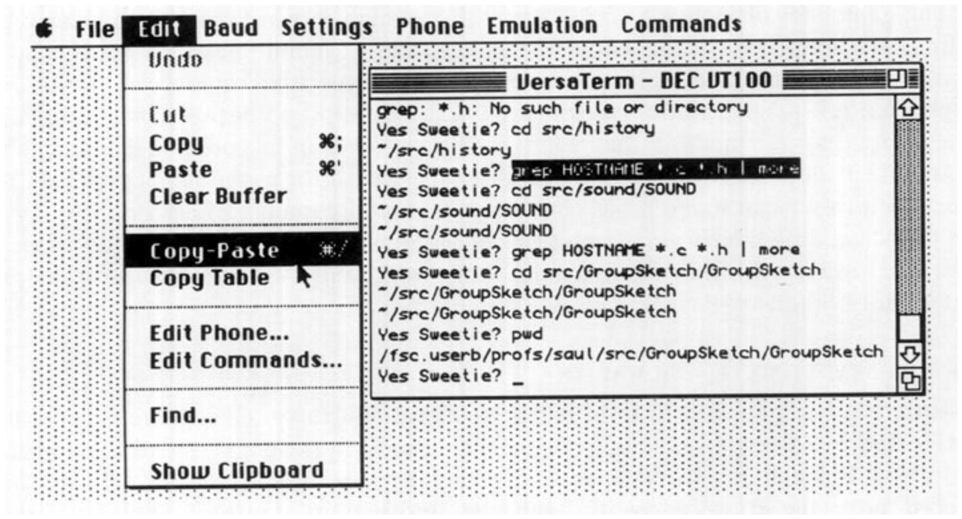
3.3. HISTORY BY EDITING TRANSCRIPTS

Some systems do not have a command history mechanism *per se*, but provide similar capabilities through editing a transcript of the dialogue. Instead of having the sequential text dialogue scroll off the screen (as with a glass teletype), it can be maintained as a scrollable transcript. When text appearing previously can be selected and used as input to the system, the transcript becomes a rudimentary history mechanism.†

Copy and paste capabilities are available in most modern-day window-based environments, where any text can be copied and pasted anywhere else. As a typical example, VERSATERM‡ is a terminal emulator for the APPLE MACINTOSH that maintains the transcript in a scrollable window (Figure 5). As shown in the Figure, text appearing within the transcript can be selected, then copied and pasted by choosing the pull-down menu option. This will insert the text into the command input area after the text cursor at the window's bottom, where it may be edited as needed. Explicit history lists are not maintained except as part of the scrollable dialogue transcript. Although there are some slight interface differences, many other popular window-based terminal emulators allow one to select a text region

† The ability to scroll over a session's transcript and select text for re-execution goes by a variety of names: spatial browsing (Kurlander & Feiner, 1990), history through command typescripts by direct manipulation (Lee, 1990) and history by editing transcripts (this paper).

‡ Versaterm 4.0 software is produced by Abelbeck Software.



```
grep: *.h: No such file or directory
Yes Sweetie? cd src/history
~/src/history
Yes Sweetie? grep HOSTNAME *.c *.h | more
Yes Sweetie? cd src/sound/SOUND
~/src/sound/SOUND
~/src/sound/SOUND
Yes Sweetie? grep HOSTNAME *.c *.h | more
Yes Sweetie? cd src/GroupSketch/GroupSketch
~/src/GroupSketch/GroupSketch
~/src/GroupSketch/GroupSketch
Yes Sweetie? pwd
/fsc.userb/profs/saul/src/GroupSketch/GroupSketch
Yes Sweetie? _
```

FIGURE 5. The VERSATERM terminal emulator for the APPLE MACINTOSH.

anywhere on the display and paste it to the command input area. Examples include *xterm* within the standard X-WINDOW SYSTEM (Quercia & O'Reilly, 1990), *pads* within Apollo's DOMAIN window system (Apollo, 1986), and the *CommandTool* within the OPEN LOOK DESKSET environment (Sun Microsystems, 1990). Although *any* text in a transcript is potentially executable in all these systems, the tradeoff is that mixing previous input commands with output makes useful candidates more difficult to find.

Another example is *emacs*, an editing environment that provides multiple views of buffers through tiled windows (Stallman, 1981). Although buffers typically allow users to view and edit files, it is also possible to run interactive processes (or programs) within them. In most UNIX-based implementations of *emacs*, it is a simple matter to call up a window running UNIX *csh* (e.g. Stallman, 1987; Unipress, 1986). All capabilities of *emacs* are then available—commands may be edited, sessions scrolled, pieces of text picked up from any buffer and used as input, and so on.

A further variant of transcript editing is the *zmacs* editor running within the SYMBOLICS GENERA LISP environment. This editor contains features of all history systems discussed so far (Symbolics, 1985*a*). Within the top-level Lisp Listener, *zmacs* extends the functionality of *emacs*. Although used primarily for entering and editing command lines, previous inputs appearing within the transcript become mouse-sensitive. A box appears around them as the mouse passes over them, and pressing one mouse button copies the old command line into the input area and makes it available for editing. Other button combinations immediately re-execute previous commands, copy arbitrary command words, show context-sensitive documentation, and so on. Alternatively, part or all of the mouse-sensitive event list can be displayed within the Lisp Listener window. Keyboard-based retrieval is also available within *zmacs*. Using the standard editing commands within the one-line input area, a user can search, cycle through and recall previous events, similarly to the command-line capabilities of the VMS operating system (DEC, 1985), UNIX *tcsh* (Ellis *et al.*, 1987), and GNU *emacs* (Stallman, 1987).

The guidelines of Figure 1 do not apply in a straightforward manner to history by editing transcripts. First, the whole screen becomes a "menu" in which items can be chosen by cutting and pasting, so one could argue that guidelines 1a–d are supported (although 1d only holds if little text is interposed between the command lines). An extra benefit is that each activity is seen in its original context; the downside is the added overhead in searching the screen for useful items. Second, the resourceful user can make a file containing favoured command lines and display it on the screen, effectively creating a task set of items available for reuse (guidelines 2 and 3). However, this may involve much extra work, and it is likely that most people would not bother.

3.4. HISTORY BY NAVIGATIONAL TRACES

History has been applied to information retrieval and to systems where items must be located by some navigational process. These include traversing menu hierarchies, searching through file directories, navigating hypertext, and so on. History can record both the route taken through the information structure and the actual information finally selected, and then allow users to quickly re-travel previous paths and re-choose old items.

In many information systems, users tend to retrieve items that have been accessed previously (Greenberg & Witten, 1985*a*). The assumption that previously-read documents are referred to many times has been supported by a study of *man*, the UNIX on-line manual (Greenberg, 1984). Each user frequently retrieved the same small set of pages from the large set available, where sets differed substantially between users. By keeping a history list of the documentation retrieved, users can avoid re-navigating the path to a previously viewed topic. Since items on the list can be regarded as place-holders in a large document, they are sometimes known as "bookmarks".

MACINTOSH HYPERCARD is a hypertext facility that allows authoring and browsing of stacks of information comprised of cards. Navigating cross links between stacks and cards is usually accomplished by simple button or menu selections. RECENT is a history facility within HYPERCARD that maintains a pictorial list of up to the last 42 unique cards visited (Figure 6). Each picture is a miniature view of the card, placed on the list in order of first appearance.† Note that ordering items by their first rather than last appearance is a particularly poor method for offering candidates for reuse, and violates guideline 1e (Greenberg & Witten, 1993). The last card visited is distinguished by a larger border, as illustrated by the second miniature in the first row of Figure 6. A pull-down menu option pops up the RECENT display, and old cards are revisited by selecting their miniature from the list (Goodman, 1987). When more than 42 unique items have been selected, the first row of seven items is cleared and made available for new ones (even though a card in the first row may have recently been selected; this violates guideline 1g).

Feiner, Nagy and van Dam (1982) push the notion of miniatures even further in their experimental hypertext system. A document page includes an image plus

† Figure 6 is a fairly accurate rendition of the screen. As these miniature pictures are of surprisingly poor quality, the value of the current RECENT implementation is questionable. However, this problem could be overcome by a higher-resolution display or by including a "magnifying glass".

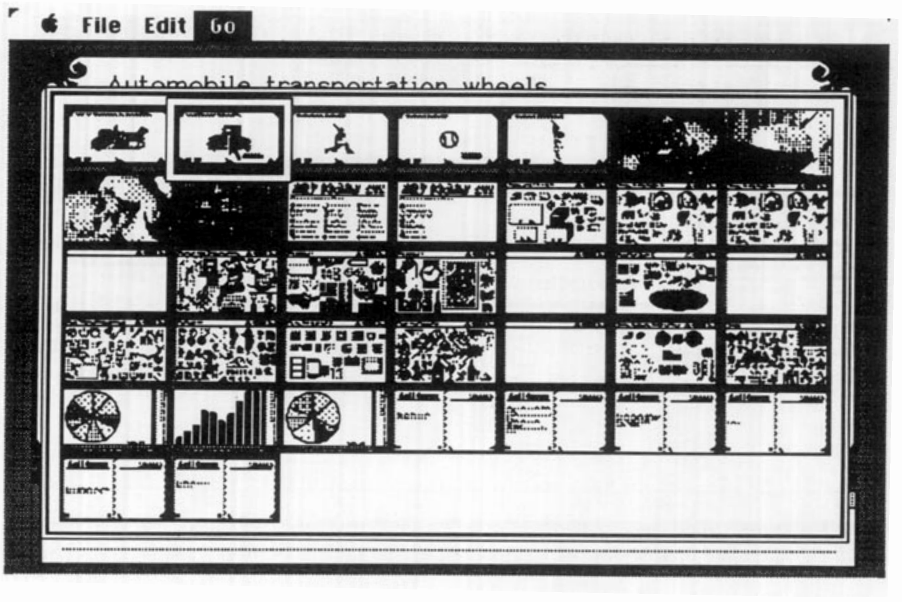


FIGURE 6. The HYPERCARD RECENT screen.

controls for moving between pages. One control is the "back page" button, a miniaturized image of the last page visited with the word "back" overlaid on top of it. Selecting this control will replace the current page with the last one visited. More complex is the special "timeline" page, a time-stamped event list of the pages visited. What makes this interesting is that miniatures are presented in a scrollable two-dimensional grid. The horizontal axis represents chronological order, and the vertical axis the chapters in the document.

Miniatures of the visited pages are positioned on the grid by their parent chapter and chronological order of selection. As with the "back page" button, selecting a miniature will transport the reader back to that page. In one sense, the two-dimensional grid partitions pages into functional groups, each group being a chapter (guideline 2b).

The SYMBOLICS Lisp environment includes a very large on-line manual viewable with the DOCUMENT EXAMINER, a window-based hypertext system (Symbolics, 1985*b*). The main window is divided into functionally different panes: a documentation display area, a menu of topics, a bookmarks area, and a command line. The bookmarks area displays a history list of titles of previously-viewed topics, where each title is a bookmark. Further bookmarks may be added explicitly by the user (these are distinguished visually from historical bookmarks). Selecting a bookmark displays either full documentation or a brief summary of the topic in the documentation area. A similar bookmark strategy has been proposed for videotext systems (Engel, Andriessen & Schmitz, 1983).

These techniques are not limited to hypertext. Navigation occurs in many human–computer interfaces, from hierarchical menu and folder systems to structured browsers for programming systems. Many modern window environments now supply graphical file browsers to let users navigate visually through their (usually

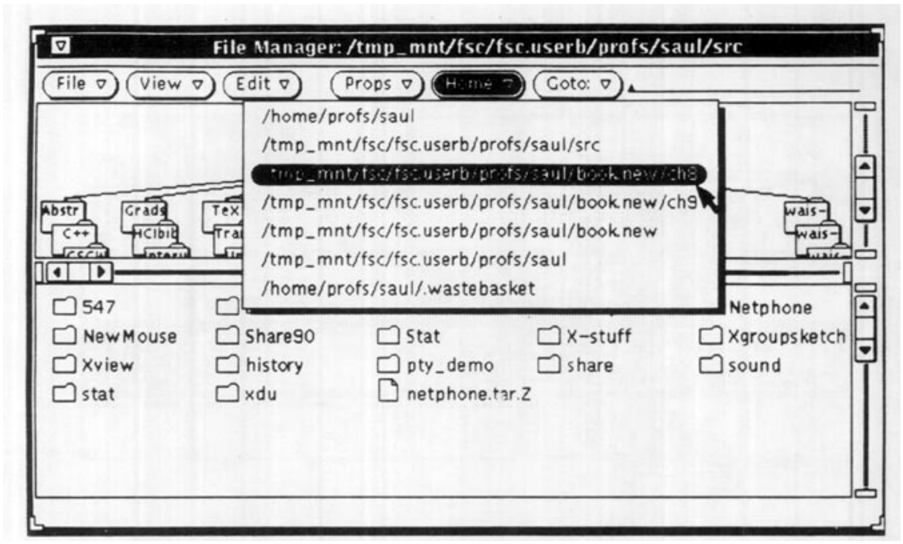

File Manager: /tmp_mnt/fsc/fsc.userb/profs/saul/src
File
View
Edit
Props
Home
Goto:
/home/profs/saul
/tmp_mnt/fsc/fsc.userb/profs/saul/src
/tmp_mnt/fsc/fsc.userb/profs/saul/book.new/ch9
/tmp_mnt/fsc/fsc.userb/profs/saul/book.new
/tmp_mnt/fsc/fsc.userb/profs/saul
/home/profs/saul/.wastebasket
Abstr
C++
Grads
HCIbib
TeX
Tra
wais
547
Netphone
NewMouse
Share90
Stat
X-stuff
Xgroupsketch
Xview
history
pty_demo
share
sound
stat
xdu
netphone.tar.Z

FIGURE 7. A sample FILE MANAGER window, showing a history list of the last few files visited.

hierarchical) file stores. Some include history facilities. FILE MANAGER, the file browser provided by SUN's OPEN LOOK environment, retains a history list of all directories the user has navigated through (Sun Microsystems, 1990). Figure 7 illustrates its use. The list can be popped up as a menu by selecting the Home button, and selecting any of the directory paths presented will immediately bring the user there. As with RECENT, the FILE MANAGER has a poor strategy for displaying past activities, for items are presented in their order of first appearance.

A more elaborate scheme is available on the APPLE MACINTOSH. Within an application, a file is usually opened through an "open dialogue box", a simple mechanism that lets users navigate up and down a folder hierarchy, with files shown in a scrollable and selectable list (left side of Figure 8). While the basic system has minimal support for history—the previously opened folder is presented by default to the user—a third-party system called BOOMERANG adds full history support onto the open dialogue box.† The menu on the right of Figure 8 shows a user employing BOOMERANG's top-level menu to access a history-ordered list of previously opened files (files that cannot be opened by the current application are greyed out and are not selectable). Similar functionality is also available for folders and for disks.

4. Reuse through adaptive systems

History mechanisms model the user's previous inputs by recording them in a time-ordered list. *Adaptive systems* build more elaborate statistical models dynamically and use them to predict subsequent inputs, which are presented to the user for selection or approval. The advantage is that events may be predicted that are not tied to recency (guideline 1f). Moreover, the adaptive system may be integrated directly into the application, with no need for a separate subsystem (cf. history

† Boomerang 2.0 developed by Zeta Soft (H. Yamamoto) and distributed by Now Software, 2425B Channing Wy, Suite 492, Berkeley, California.

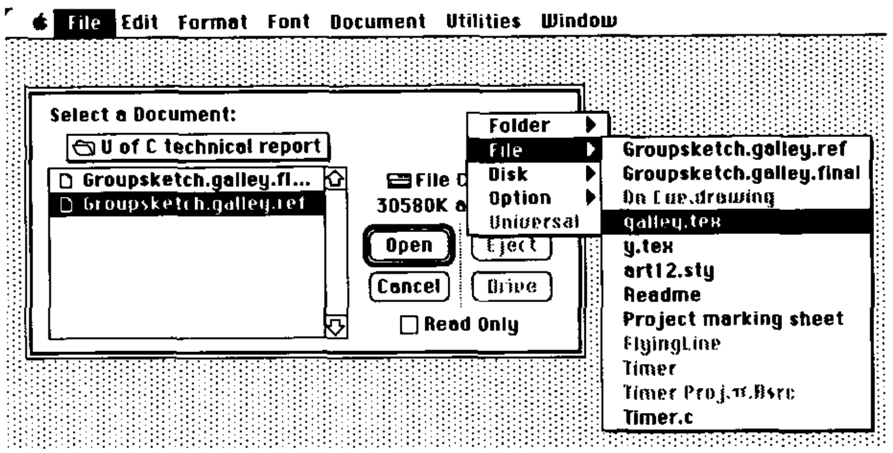


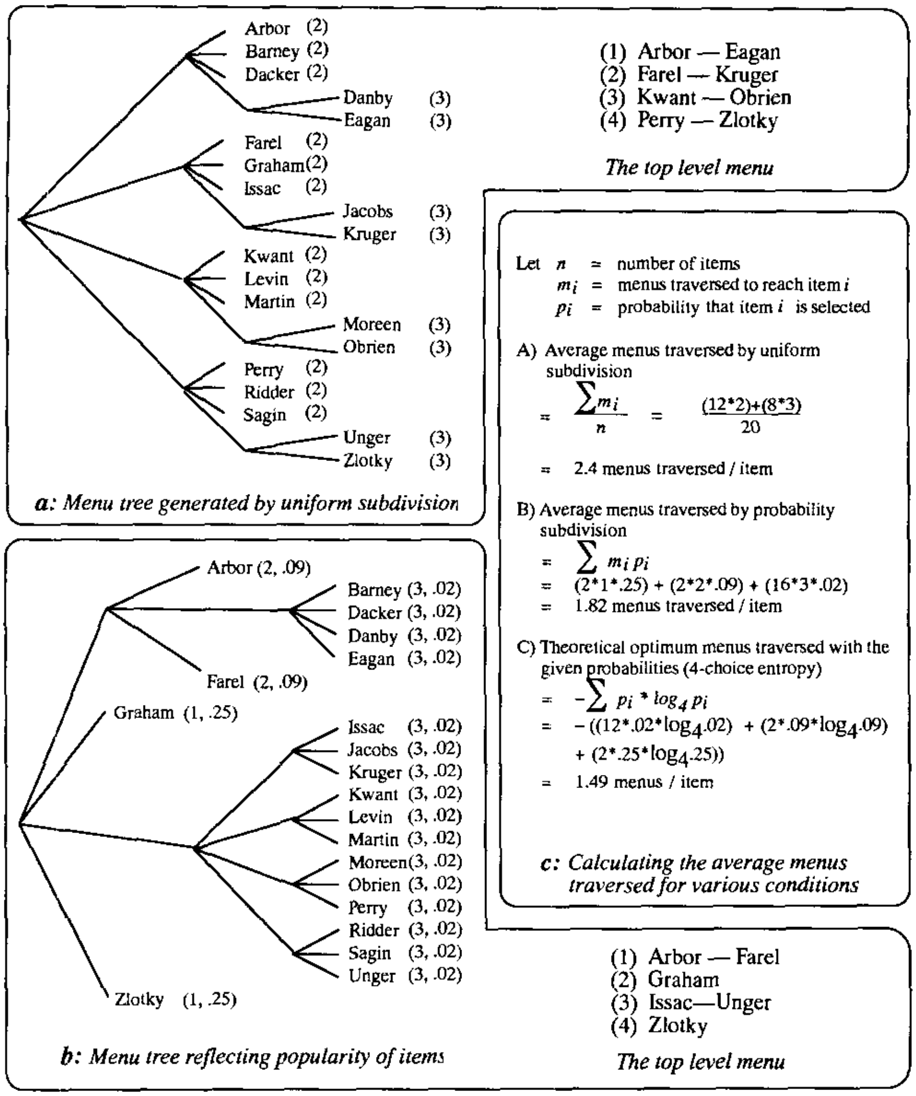
FIGURE 8. The APPLE MACINTOSH Open Dialogue Box, showing the BOOMERANG history menu.

mechanisms). Here we describe two types of adaptive system, one for accelerating selection in a hierarchical menu system and the other for the entry of free text. Both employ predominantly frequency-based, rather than recency-based, models.

4.1. ADAPTIVE MENU HIERARCHIES

It is possible to devise interactive menu-based interfaces that dynamically reconfigure a menu hierarchy so that high-frequency items are treated preferentially, at the expense of low-frequency ones. ADAPTIVE MENUS provide an attractive way of reducing the average number of choices that a user must make to select an item without adding further paraphernalia to the interface (Greenberg, 1984; Witten, Greenberg & Cleary, 1983; Witten, Cleary & Greenberg, 1984). Consider a telephone directory where the access frequencies of names combined with their recency of selection define a probability distribution on the set of entries, which reflects the "popularity" of the names selected (Greenberg & Witten, 1985*a*). Instead of selecting regions at each stage to cover approximately equal ranges of names, it is possible to divide the probability distribution into approximately equal portions. During use, the act of selection will alter the distribution and thereby increase the probability of the names selected. Thus the user will be directed more quickly to entries which have already been selected—especially if they have been selected often and recently—than to those which have not.

Figures 9(*a*) and 9(*b*) depict two menu hierarchies for a very small dictionary with 20 name entries, and their corresponding top-level menu. Figure 9(*c*) calculates the average number of menus traversed per selection. In Figure 9(*a*), the hierarchy was obtained by subdividing the name space as equally as possible at each stage, with a menu size of four. The number following each name shows how many menu pages have to be scanned before that name can be found. Figure 9(*b*) shows a similar hierarchy that now reflects a particular frequency distribution (the second number following the name shows the item's probability of selection). Popular names appear immediately on the first-level menu. Less popular ones are accessed on the second-level menu, while the remainder are relegated to the third level. For this



Let n = number of items
m_i = menus traversed to reach item i
p_i = probability that item i is selected

A) Average menus traversed by uniform subdivision

$$= \frac{\sum m_i}{n} = \frac{(12*2)+(8*3)}{20}$$

= 2.4 menus traversed / item

B) Average menus traversed by probability subdivision

$$= \sum m_i p_i$$
$$= (2*1*.25) + (2*2*.09) + (16*3*.02)$$

= 1.82 menus traversed / item

C) Theoretical optimum menus traversed with the given probabilities (4-choice entropy)

$$= -\sum p_i * \log_4 p_i$$
$$= -((12*.02*\log_4 .02) + (2*.09*\log_4 .09) + (2*.25*\log_4 .25))$$

= 1.49 menus / item

c: Calculating the average menus traversed for various conditions

FIGURE 9. Menu trees generated by uniform and probability subdivision.

particular case, the average number of menus traversed is smaller with probability subdivision than it is with uniform subdivision, although this improvement is not as much as theoretically possible [Figure 9(*c*)].

Given a frequency distribution, it is not easy to construct a menu hierarchy that minimizes the average number of selections required to find a name. Exhaustive search over all menu trees is infeasible for all but the smallest problems. However, simple splitting algorithms achieve good performance in practice (Witten, Cleary & Greenberg, 1984).

Because probabilities are made to decay over time, once-popular (or erroneously-chosen) names eventually drop to a low value. This also builds in a way of balancing frequency and recency. While low decay will see frequently-chosen items migrate up the tree, a high decay rate gives more room to recently-chosen items.

With ADAPTIVE MENUS, previous actions—both recent ones and those submitted long ago—are almost always easier to resubmit (guidelines 1a,b,e,f,g). Also, since no extra detail is added to the interface presentation, screen usage and cognitive interruption is minimized. However, users must now scan the menus for their entries all the time, even for those accessed frequently. Since paths change dynamically, memory cannot be used to bypass the search process. Experimental evidence suggests that this is not a problem in practice, so guideline 1c is preserved. As long as the database of entries is large, the benefits will usually outweigh the deficiencies (Greenberg & Witten, 1985*a*). It is also possible to have the system monitor the average depth of the menu selection process over time. If for some reason the average depth increased beyond what would normally be expected, a static menu system could be substituted for the adaptive one (Trevellyan & Browne 1987; Totterdell *et al.*, 1990).

4.2. REUSE THROUGH TEXT PREDICTION

History facilities assume that the last submissions entered are likely candidates for re-execution. They are the ones visible on the screen in graphical and editing systems, the ones most easily remembered by the user in glass teletypes, and the ones weighted into the probability distribution in ADAPTIVE MENUS (although the latter depends on the decay factor).

Two systems provide an alternative strategy for textual input—the REACTIVE KEYBOARD (Witten, Cleary & Darragh, 1983; Darragh, Witten & James, 1990; Darragh & Witten, 1992) and its precursor PREDICT (Witten, 1982). Although implementation details differ, both use a dynamic adaptive model of the text entered so far to predict further submissions. At each point during text entry, the system estimates for each character the probability that it will be the next one typed. This is based upon a Markov model that conditions the probability that a particular symbol is seen on the fixed-length sequence of characters that precede it. The order of the model is the number of characters in the context used for prediction. For example, suppose an order-3 model is selected, and the user's last two characters are denoted by "*xy*". The next character, denoted by Φ, is predicted based upon occurrences of "$xy\Phi$" in previous text (Witten, Cleary & Darragh, 1983).

PREDICT filters any glass-teletype package, although limited character-graphics capabilities are required for its own interface. It selects a single prediction (or none at all) as the most likely and displays it in reverse video in front of the current cursor position. Users have the option of accepting correct predictions as though they had typed them themselves, or rejecting them by simply continuing to type. Because only a single prediction is displayed, much of the power of the predictive method is lost; at any point the model will have a range of predictions with associated probabilities, and it is hard to choose a single "best" one (Witten, 1982).

The REACTIVE KEYBOARD, on the other hand, has two versions of a more sophisticated interface that allows one to choose from multiple predictions. The first, called RK-BUTTON, has a similar interface to PREDICT except that users now have the ability to cycle through a probability-ordered list of predictions. An interaction with a:

```
$ mail^J                                                    ^N
  cd news^J                                                 ^W
  cd news^J                                                 ^N
  cd rk/papers/ieee.computer^J                              ^L
  cd rk/papers/ieee.computer

$ emacs paper.tex^J                                         ^L
  emacs paper.tex

$ rm paper.tex.CKP paper.tex.BAK^J                          ^L
  rm paper.tex.CKP paper.tex.BAK

$ wc -w paper.tex^J                                         ^L
  wc -w paper.tex

$ readnews -n comp.sources.unix^J                           ^N
  mail^J                                                    ^W
  mail^J                                                    ^N
  mail bdarragh%uncamult.bitnet@ucnet.ucalgary.ca^J         ^L
  mail bdarragh%uncamult.bitnet@ucnet.ucalgary.ca
```

b:

Key	*Description*
^C	Accept the next predicted character
^W	Accept the next predicted word
^L	Accept the whole predicted line
^N	Show the next alternative prediction
^P	Show the previous alternative prediction

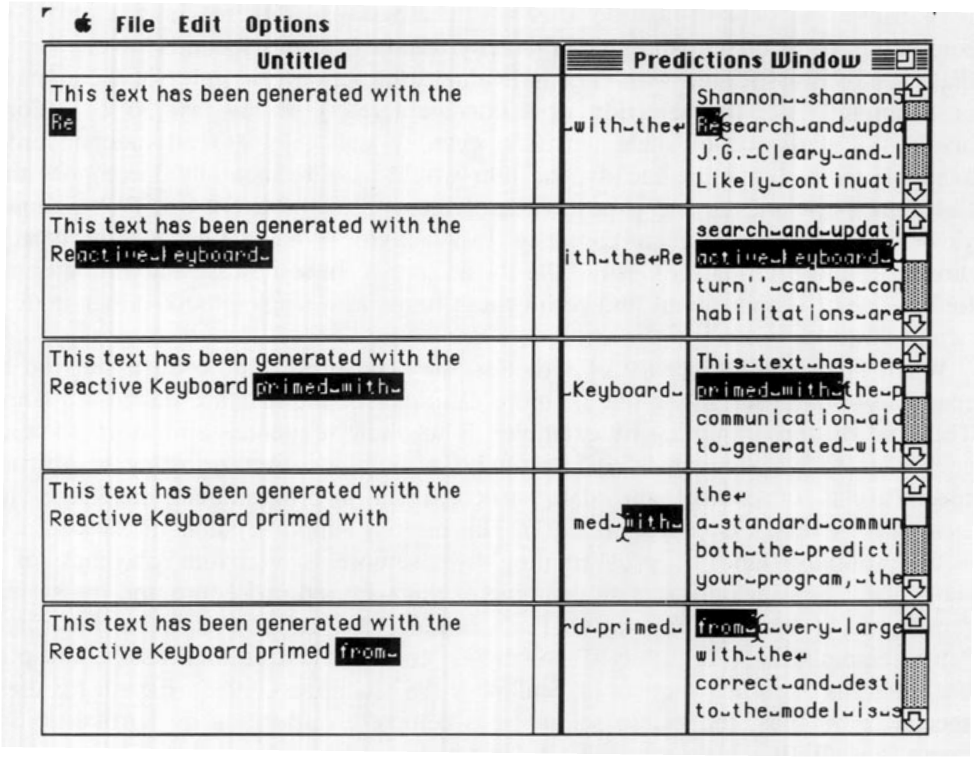
c:

```
$ cd rk/papers/ieee.computer
$ emacs paper.tex
$ rm paper.tex.CKP paper.tex.BAK
$ wc -w paper.tex
$ mail bdarragh%uncamult.bitnet@ucnet.ucalgary.ca
```

FIGURE 10. Using RK-BUTTON, the UNIX version of the REACTIVE KEYBOARD. (*a*) An RK-BUTTON dialogue with UNIX; (*b*) a key to several commands dealing with predictions; (*c*) screen contents at end of the dialogue.

UNIX using RK-BUTTON is shown in Figure 10. The predicted characters are written in reverse video on the screen, and represented in the figure with enclosing rectangles. Control characters are preceded by "^", and "^**J**" is the end-of-line character. The column on the right shows the keys actually struck by the user. Figure 10(*b*) gives the meaning of a few of the control keys; in fact, many more line-editing features are provided. Although not illustrated in the Figure, the system is set up so that typing non-control characters simply overwrites the predictions; thus one may use the keyboard in the ordinary way without even looking at the screen.

The second version of the REACTIVE KEYBOARD, called RK-POINTER, displays a menu containing the best *p* predictions, which changes dynamically with the immediate context of the text being entered (Darragh, Witten & James, 1990).



```
This text has been generated with the        ˽with˽the↵   Shannon˽˽shannon
Re                                                        Research˽and˽upd
                                                          J.G.˽Cleary˽and˽l
                                                          Likely˽continuati

This text has been generated with the        ith˽the↵Re   search˽and˽updati
Reactive˽keyboard˽                                        active˽keyboard˽
                                                          turn''˽can˽be˽con
                                                          habilitations˽are

This text has been generated with the        Keyboard˽    This˽text˽has˽bee
Reactive Keyboard primed˽with˽                            primed˽with˽the˽p
                                                          communication˽aid
                                                          to˽generated˽with

This text has been generated with the        med˽with˽    the↵
Reactive Keyboard primed with                             a˽standard˽commun
                                                          both˽the˽predicti
                                                          your˽program,˽the

This text has been generated with the        d˽primed˽    from˽a˽very˽large
Reactive Keyboard primed from˽                            with˽the↵
                                                          correct˽and˽desti
                                                          to˽the˽model˽is˽s
```

FIGURE 11. Using RK-POINTER, the Macintosh version of the REACTIVE KEYBOARD.

Figure 11 shows RK-POINTER in action by displaying five sequences of the user composing some free text in the window on the left, with a window of predictions on the right. In the "Predictions Window", the left region contains the context string upon which predictions are made (its length is adjustable by the user). On its right are rank-ordered predictions, presented as alternative pieces of text from which the user can choose the next characters. Interaction is through a pointing device, such as a mouse. Selection is two-dimensional, in that the user can point anywhere within a prediction to accept only the previous characters (Figure 11). Less likely predictions are available through scrolling.

In the REACTIVE KEYBOARD, the next prediction is based on the previous context. If the user is following the steps of a procedure they had invoked previously, the system is likely to recreate the entire sequence (guideline 2a).

5. Reuse through programming by example

The schemes discussed so far attempt to facilitate the reuse of individual items of activity, be they commands, command lines, menu selections, or characters predicted in context. This is sufficient if incremental activities have a one-to-one correspondence with tasks the user may wish to repeat later. These are the items supported by the points under guideline 1. Often, however, tasks are accomplished by sequences of several primitive activities.

"Closure" is defined as the user's subjective sense of reaching a goal, of completion or of understanding (Thimbleby, 1980). Previous sections have assumed that closure is associated with each individual user action (the entry of a command or command line, the selection of a document, etc.). If the task to be redone involves a sequence of such activities, even though they are all independently available through a reuse facility, the user would have to mentally decompose this task into its primitives and choose each of them from the event list. For example, viewing a specific file can comprise two activities—navigating to the correct directory; and printing the desired file to the screen. In some cases, it will be easier for the user to think about and recall these items as a single chunk rather than as two separate activities.

When tasks are a sequence of activities, they constitute a procedure that can be specified by the user giving one or more examples of the instance of the sequence. The goal of "programming by example" is to allow sequences and more complex constructs to be communicated concretely, without the user resorting to abstract specifications of control and data structure (in a programming language, for example) (Myers, 1986; Cypher, 1993). This directly supports guideline 2a.

The simplest kind of programming by example is verbatim playback of a sequence. The user performs an example of the required procedure and the system remembers it for later repetition. For example, the use of "start-remembering", "stop-remembering", and "do-it" commands enable a text editor to store and play back macros of editing sequences (Stallman, 1987; Unipress, 1986). Except for these special commands, the macro sequence is completely specified by normal editing operations. With a little more effort, such sequences can be named, filed for later use, and even edited (if presented in a human-readable form). A practical difficulty with having a special mode—remembering mode—for recording a sequence is that one has frequently already started the sequence before deciding to record it, and so must retrace one's steps and begin again.

The ability to generalize these simple macros could extend their power enormously. Some programming-by-example strategies allow inclusion of standard programming concepts—variables, conditionals, iteration, and so on—either by inference from a number of sample sequences, or through explicit elaboration of an example by the user. To illustrate the latter, an experimental system called SmallStar was constructed for the Xerox Star office workstation that operates according to the direct manipulation paradigm (Halbert, 1981, 1984). In the first version of SmallStar, a pop-up menu allowed one to indicate explicitly the generalization required. For example, icons selected at specification time are disambiguated by name, position, or by asking for a similar object. However, because people found it hard to elaborate programming constructs when tracing through an example, a later version had users employ an editor to specify constructs after macro composition (Halbert, 1984).†

Reminiscent of the editing capabilities of SmallStar is CE Software Inc's QuicKeys, a macro facility for the Apple Macintosh. Through a pull-down menu (left side of Figure 12), the user can start, stop and pause recording sequences, choose selected macros for playback (two are shown at the bottom of the menu),

† An excellent survey and discussion of programming by example systems is provided by Cypher (1993).

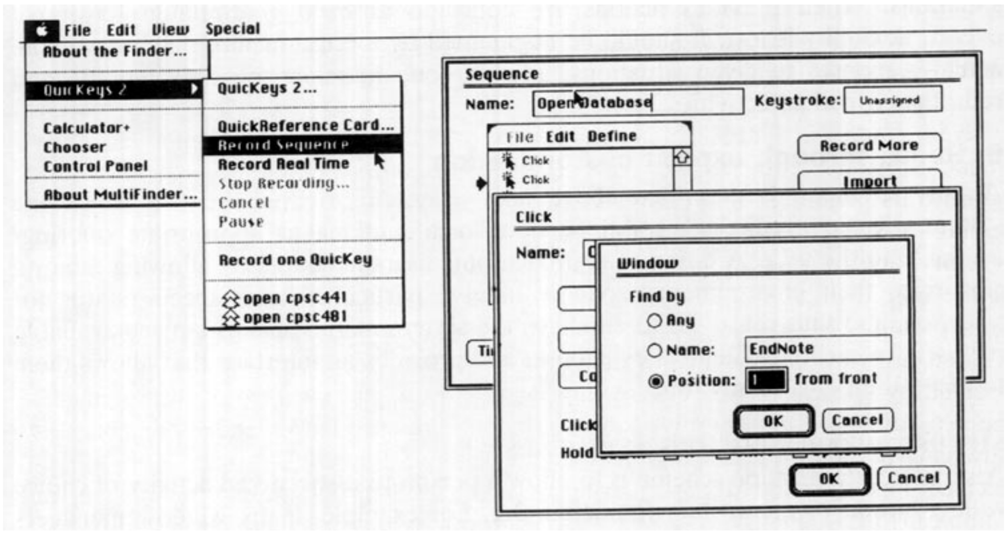


FIGURE 12. The QUICKEYS menu and several sequence-editing windows.

and display a reference card containing all macros that have been recorded previously. Once a macro has been defined, it may be edited. The right side of Figure 12 shows a user editing a macro sequence she has named "Open Database" (background window). A mouse "Click" primitive, which was used to open a window, has been chosen (middle window), and the user now has the option of having QUICKEYS find the window by its name on playback, rather than by its position on the window stack.

Other research on programming by example has concentrated on inferring control constructs from traces of execution given by the user (Witten, MacDonald & Greenberg, 1987) and some systems use domain knowledge, teaching metaphors, and highly interactive interfaces to maximize the speed of transfer of procedures (e.g. Maulsby & Witten, 1989; Maulsby, Witten & Kittlitz, 1989). However, there has been little research on ways of naming, filing and accessing procedures taught by example, and particularly on knowledge- and history-based methods of splitting up a stream of activities into user-oriented "tasks". This limits the practical use of programming by example in reuse systems.

The appeal of programming by example is the belief that a user's activity follows a preconceived plan that can be encapsulated as a procedure. Intentions are realized as plans-for-actions that directly guide behaviour, and plans are actually prescriptions or instructions for actions. These plans reduce to a detailed set of instructions (which may also be sub-plans) that actually serve as the program that controls the action. Suchman disputes this notion by claiming that plans are derived from *situated action*—the necessarily ad hoc responses to the contingencies of particular situations (Suchman, 1987). Initial plans must be inherently vague if they are to accommodate the unforeseeable contingencies of actual situations of action. It is only the post hoc analysis of situated action that makes it appear as if a rational plan were followed. Assuming that user activity on computers does follow situated action, then a programming by example system would not suffice by itself as a complete user support tool, for it would not respond well to the changing circumstances of situations. When previous actions are collected as fixed goal-related scripts of events, flexibility is lost. It should be augmented by a reuse facility that collects the actual responses to given situations, allowing one to select, possibly modify, and redo the individual activities.

6. Reuse through explicit customization

Almost all methods described above take a system-oriented view in which the system tries to predict what will be done next and makes appropriate offerings available to the user. A different approach puts users in charge by allowing them to customize their environment explicitly to give particular items special status for selection, and thus reuse. At its simplest, the system merely makes a place available for users to cache the items they choose, and provides an interface that allows them to recall the items at any time.

6.1. CUSTOMIZABLE BUTTONS AND MENUS

A simple customization scheme is to allow a person to associate an activity of choice with a button or menu item (guideline 3c). For example, many window managers include a user-modifiable file that describes the behaviour of mouse button presses and pop-up menus (as well as other things). Consider this commented entry in a user's MOTIF window manager resource file.

```
Menu ApplicationsMenu
{ ''Applications''  f.title              #The menu title
  no-label          f.separator          #Draw a separating line
  ''Emacs''         !''emacs''           #Invoke the emacs editor
  ''Xdvi''          !''xdvi -expert''    #Invoke the TeX previewer
  ''Xterm''         !''xterm -name       #Invoke an X terminal window
                      ''X-Terminal''
}
Buttons DefaultButtonBindings            #Left mouse button press in the
{ ⟨Btn1Down⟩        root  f.menu         # root screen pops up the menu
                          ApplicationsMenu
  Shift⟨Btn1Down⟩   root  !''xterm''     #Shift left button invokes
}                                        # an X terminal window
```

Whenever the window manager is invoked, it reads the resource file. From these instructions it constructs a menu titled "Applications" which contains items representing three different application programs. When the user presses the left mouse button in the root screen, the menu appears and items may be selected. If the shift key is held down at the same time, the X-Terminal application is invoked directly. Notice that users see the activity as a symbolic label (the menu item), rather than the literal command that will be executed (guideline 1h). While the comments within the script are useful annotations, they are not accessible from the menu and guideline 1i is not supported. Of course, creating, modifying, and debugging these scripts are difficult activities, and guideline 3d is violated.

The VERSATERM terminal emulator (see Section 3.3) allows a user to customize a "Commands" menu (this is invoked by the "Edit Commands" option in Figure 5). Users can invoke a form and fill in the "User Commands" and "Command Label" fields. The latter labels items appearing on the command menu, while the former represents text inserted into the terminal window when an item is selected. The text is then executed by the running process as if the user had typed it directly. This has several advantages over the MOTIF scheme above. First, users can change the Commands menu at any time, and changes take effect immediately. Second, users can fill in the form by several methods. They can create items by typing them in, by cutting and pasting text from the transcript window, or by recording a verbatim macro (guideline 2a; see Section 5).

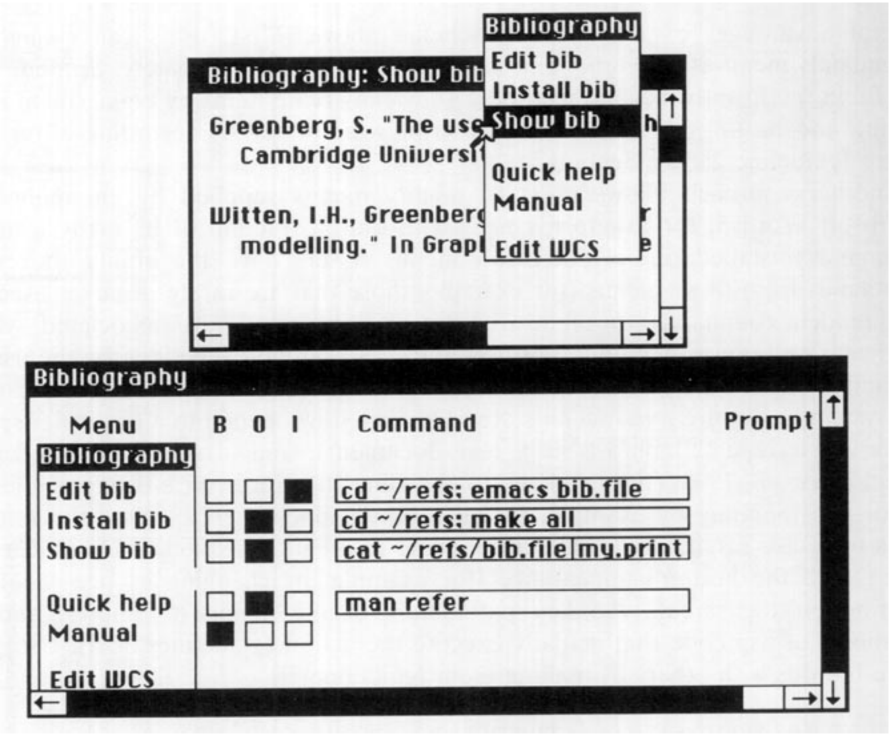
Another approach allows users to modify menus supplied by an application. Microsoft Word 5, for example, comes with only a fraction of its many available commands installed on its pull-down menus. Users have the ability to remove commands from these menus (for example, those that are rarely or never used), or add hidden commands to them. By default, commands are associated with a particular pull-down menu and position, but these can be overridden by the user.

Finally, buttons may be established by "situated creation", where a user can ask the system to capture a particular state (e.g. a window opened to a particular spot in a file, or a repeated phrase in a text document), and attach that to a button (MacLean *et al.*, 1990) (this is related to guideline 3e), EUROPARC's BUTTONS interface is not programming by example, for all a button does is encapsulate the relevant parts of a task instance—no attempt is made to create a sequence. Users can then tinker with the button's appearance (for example, by changing its text label), its location on the screen, the key parameters of the action it denotes, and the fragments of lisp code that actually execute the action (guidelines 3c,d). Users can share buttons with others by mailing them (guideline 3f).

6.2. CUSTOMIZABLE SETS OF ACTIVITIES

As mentioned in Section 2.2, users may partition their work into goal-related tasks. These are usually interleaved with each other as the user switches, suspends and resumes goals. The consequence is that people should have the power to customize their workspace not only by giving special status to favoured activities, but to organize these activities as well.

One system that permits this is the WORKBENCH CREATION SYSTEM (Greenberg & Witten, 1985*b*). This experimental interface allows users to create a hierarchy of pop-up menus (with user-definable items and annotations) and attach them to windows. Each window would generally be associated with a task or function (guidelines 1h,i; 2b–f; 3a–c). Figure 13 illustrates one such configuration. The upper window is a "Bibliography" window that handles a variety of functions associated with one user's personal bibliographic reference library. The pop-up menu shows some of the activities this window handles: editing the bibliography, installing a new version of it, and printing it into the window. The menu also gives access to a help system; either displaying a specific help document related to bibliographies (the "Quick help" item) or calling up a new window (another workbench) that gives general access to the on-line manual. The user can further customize the bibliographic workbench by selecting "Edit WCS", which invokes the workbench editor (lower window in Figure 13). Here, item labels are entered by form filling the menu, and the user-supplied UNIX command (executed when an item is selected) is typed to its right. If an argument to the command line is needed that can only be supplied at run time, the user can specify a prompt message and how the answer should be substituted into the command line (not shown in the Figure). Also out of view is a help field that can be filled in to annotate workbenches, menus, and items. The highlighted boxes specify whether the command is interactive (the "I"

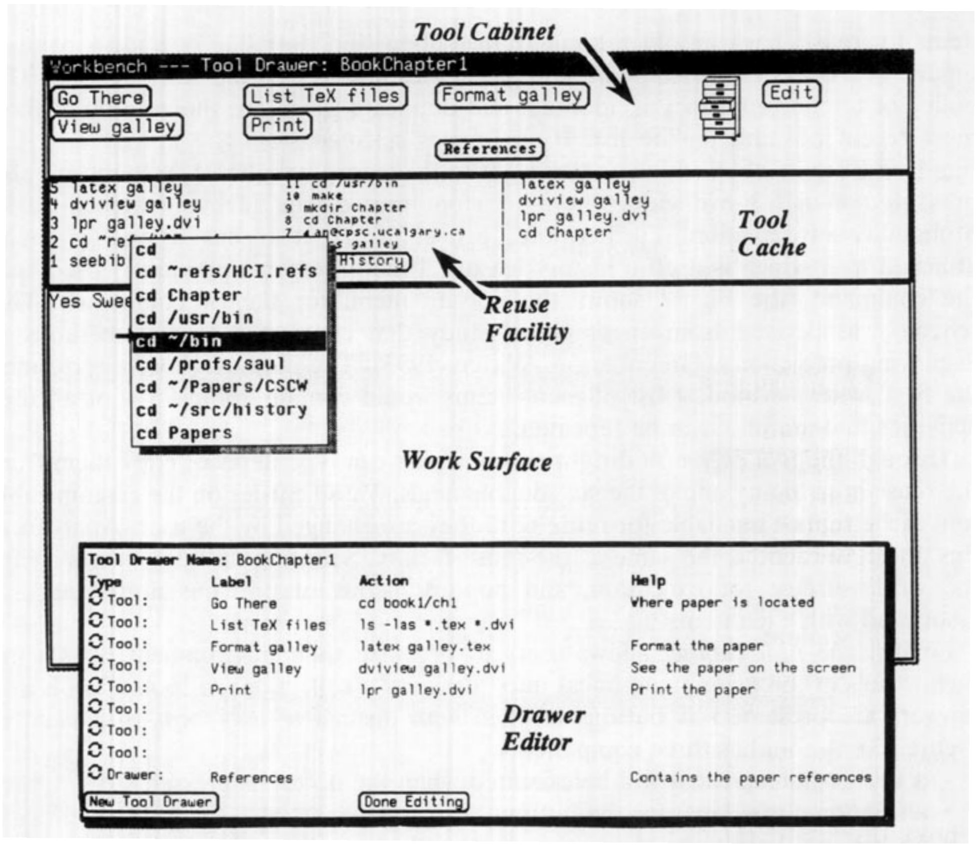


Menu	B	O	I	Command	Prompt
Bibliography					
Edit bib	□	□	■	cd ~/refs: emacs bib.file	
Install bib	□	■	□	cd ~/refs: make all	
Show bib	□	■	□	cat ~/refs/bib.file|my.print	
Quick help	□	■	□	man refer	
Manual	■	□	□		
Edit WCS					

FIGURE 13. The WORKBENCH CREATION SYSTEM. The top window shows a workbench and menu for managing bibliographies, while the bottom one shows the editor associated with that workbench.

box), displays output only (the "O" box), or activates a new workbench (the "B" box). The latter allows a web of related workbenches to be linked together. A system somewhat similar to the WORKBENCH CREATION SYSTEM was later developed by Dzida, Hoffmann and Valder (1987).

While the WORKBENCH CREATION SYSTEM lets users organize low-level commands into linked sets, another approach allows customization at the application level, where applications are grouped into a task-related "virtual workspace". ROOMS, for example, divides groups of window-based applications into collections of windows with transitions between them (Henderson Jr & Card, 1986). Each screenful is a virtual workspace (a screen) containing windows running specific applications. Many virtual workspaces exist, and a user can switch tasks by supplanting the current workspace with the desired one. Although designed mainly to reduce "thrashing" effects that occur when one tries to keep desired windows visible on a small screen, it effectively allows a person to organize collections of applications and move rapidly between them. People customize their rooms by establishing their applications of choice within them, and by linking rooms via "doors". There is even an overview screen showing active icons of all rooms, where a user can manipulate the applications they contain by moving windows and sharing them between rooms. Example rooms may include a mail room, a paper-editing room, a particular programming project room, and so on. ROOMS is perhaps the best system around for supporting all the parts of guideline 2 (except 2a), and the first four points of guideline 3.



```
5 latex galley          11 cd /usr/bin              | latex galley
4 dviview galley        10 make                     | dviview galley
3 lpr galley.dvi        9 mkdir Chapter             | lpr galley.dvi
2 cd ~re                8 cd Chapter                | cd Chapter
1 seebib                7 4 an@cpsc.ucalgary.ca
                          s galley
Yes Swee
```

```
cd
cd ~refs/HCI.refs
cd Chapter
cd /usr/bin
cd ~/bin
cd /profs/saul
cd ~/Papers/CSCW
cd ~/src/history
cd Papers
```

Tool Drawer Name: BookChapter1

Type	Label	Action	Help
Tool:	Go There	cd book1/ch1	Where paper is located
Tool:	List TeX files	ls -las *.tex *.dvi	
Tool:	Format galley	latex galley.tex	Format the paper
Tool:	View galley	dviview galley.dvi	See the paper on the screen
Tool:	Print	lpr galley.dvi	Print the paper
Tool:			
Tool:			
Tool:			
Drawer:	References		Contains the paper references

FIGURE 14. WORKBENCH in use, showing the main tiled WORKBENCH window, the pop-up WORKBENCH editor, and a pop-up menu.

7. Workbench: a reuse facility based on design guidelines

The systems presented above aim either to support reuse of low-level activities (e.g. command line, menu selections), or to allow activities to be grouped into task sets: none allow a user to do both. Here we describe a system called WORKBENCH (which evolved from the earlier WORKBENCH CREATION SYSTEM and was designed around the guidelines listed in Figure 1) that addresses both aims.

WORKBENCH is a graphical window-based front end to the UNIX command line interpreter (Greenberg, 1988, 1989, 1993). Using the metaphor of a handyman's workbench, it provides a history list that predicts future submissions from old ones, as well as an explicit way for users to structure and store submissions for later reuse. Its major visual components are detailed below and illustrated in Figure 14, which shows one paned window and two pop-up windows.

The *work surface* is the familiar glass teletype running the standard UNIX *csh* command line interpreter (the bottom pane in Figure 14). This is the main working area, and users can submit command lines by typing them directly. The standard UNIX history system is available as well.

The *reuse facility* uses history through graphical selection to make available 11 items for reuse, each an old command line submission (left side of middle pane in Figure 14). Users can select, edit, and insert these items into the work surface. The policy of temporal recency is adopted, and duplicate items are shown only in their most recent position on the list. If an item is selected from the history list, it is removed from its old position and brought to the front (guideline 1g). Items are also presented in a fish-eye view where the font size of the text is matched to its probability of selection. Furthermore, every history item has a pop-up menu attached to it which is itself a history list of all the arguments previously used with the command (the Figure shows the pop-up menu for the *cd* command). This scheme was derived from an empirical study that contrasted several methods of displaying predictions (Greenberg & Witten, 1993). The method chosen performed the best, where a modest list of menu items would contain about 75% of all user activities that are about to be repeated.

Through the *tool cache* (mid-right pane), a user can type items or copy them from the reuse area to any one of the six editable fields. These entries on the customizable tool cache remain available for reuse until they are changed by the user. Apart from this, the tool cache is the same as the reuse facility. Selecting an item inserts it into the work surface for execution, and pop-up menus of previous arguments are associated with each item.

Finally, the *tool cabinet* allows users to organize their environment by placing their "tools" (UNIX command lines) into "drawers" (collections of tools). Tools and drawers are presented as buttons labelled with distinctive text fonts (top pane in Figure 13). A tool has three components:

- a UNIX command that will be executed when the button is pressed;
- an optional text label for the button;
- an optional help string that appears when the pop-up menu is raised.

Choosing a drawer opens it and makes its tools available in the tool-cabinet pane. Selecting the cabinet icon on the top right of this pane allows the user to access a history list of drawers visited. What distinguishes the tool cabinet from a conventional menu or panel is that a drawer and its contents are customized by the end-user. By selecting the edit button, an editable representation of the current drawer appears (lowest pop-up window in Figure 14). Users are then free to type in the definition of a tool, or to copy an already well-formed expression from the reuse area or tool cache into a drawer. By drawing on the reuse facility as a primary source of tried and tested candidates for new tools (we call this *situated history*), users can rapidly create, annotate and modify their personal workspaces to reflect the task at hand.† Users are also invited to document their new tools by adding a descriptive label and help message.

WORKBENCH is implemented as a stand-alone process that can inter-operate with any application. It maintains a communication port to which applications are able to send any tokens that should be displayed on the reuse facility. In order to take advantage of this an application must know how to establish communication with WORKBENCH and how to send it tokens, but needs to know nothing of its internal structure. Conversely, WORKBENCH allows any application to connect to it, and does not need any syntactic or semantic information about the tokens it receives.

† In parallel work, MacLean *et al.* (1990) also allowed items to be copied off a history list into their BUTTONS interface.

WORKBENCH follows most of the guidelines in Figure 1. Previously submitted activities are available for recall through the reuse facility, the tool cache and the tool cabinet. By using graphical selection combined with a provably good strategy for offering predictions, the reuse facility is particularly effective (guidelines 1a–e). The tool cache lets the user explicitly favour some activities over others (guidelines 1f,g), and the tool cabinet allows items to be labelled and annotated (guidelines 1h,i). Similarly, users can group related activities into sets by placing them into a drawer (guideline 2b), and can link drawers (and thus tasks) together (guideline 2e). Each drawer can be considered a workspace and it is relatively easy to create, delete and modify drawers, especially since well-formed items may be placed into a drawer from the reuse facility, the tool cache, or other drawers (guidelines 3a–e).

However, WORKBENCH is oriented towards incremental low-level activities. While task-switching between task sets is supported by drawer linking (guidelines 2c–e), it would work best if it were embedded into a system that maintained the contexts created by high-level applications. The best candidate is the ROOMS environment. [MacLean *et al.* (1990) also recommended that their BUTTONS system should be embedded within ROOMS.] The only two guidelines not directly supported in the current implementation are 2a and 3f. Programming-by-example methods (such as a macro facility) could easily be added to encapsulate entire procedures within a tool, instead of just a single UNIX command line. Similarly, workbench descriptions are implemented as text files; there is nothing to prevent packaging the descriptions so that users could mail them to one another.

8. Summary

Most existing reuse facilities do not follow a principled approach to design. As a result, they fall far below their potential for supporting user needs. This paper summarized key design guidelines (Figure 1), and used them as a context to discuss existing systems for reuse.

A reuse facility can arrange for submissions entered to an application to be collected, presented, and made available for reuse. This offers assistance in any dialogue that exhibits recurrence. The approach is a general one because no knowledge of the application domain is needed. Moreover, the mechanism underlying a reuse facility—monitoring the user's interaction and maintaining an internal model of it—has potential for providing more extensive assistance to the user. Possibilities include using the transaction history for undo/redo to allow recovery from errors, providing an external memory aid that allows users to consult or recall information associated with past activities, and building user models (Lee, 1990).

A reuse facility can also help users to group activities into sets. If properly composed, these sets would contain most of the elements necessary to perform a task. Ideally, a set would be attached to a working context, such as a group of applications and the objects being worked on, and provision would be made for users to switch easily between different contexts.

These two aspects of reuse have been combined in a new user support tool, WORKBENCH, which provides a history list that predicts future submissions from old ones, as well as an explicit way for users to structure and store submissions for later reuse. Its use of "situated history" has manifest advantages in enabling users to customize their environment explicitly and accurately, with minimal disruption from their primary task.

References

APOLLO (1986). *DOMAIN system user's guide.* Chelmsford, MA: Apollo Computer Inc.

BANNON, L., CYPHER, A., GREENSPAN, S. & MONTY, M. (1983). Evaluation and analysis of users' activity organization. *Proceedings of the ACM CHI 83 Human Factors in Computing Systems,* pp. 54–57, Boston: ACM Press. 12–15 December.

BARNES, D. J. & BOVEY, J. D. (1986). Managing command submission in a multiple-window environment. *Software Engineering Journal,* **1,** 177–183.

BOBROW, D. (1986). HistMenu. In *Lisp User Library Packages Manual, Koto Release,* Xerox Artificial Intelligence Systems, April.

CARD, S. & HENDERSON Jr, A. (1987). A multiple, virtual-workspace interface to support user task switching. *Proceedings of ACH CHI + GI'87 Conference on Human Factors in Computing Systems and Graphics Interface,* pp. 53–59, Toronto, Canada: ACM Press. 5–9 April.

CYPHER, A. (1986). The structure of user's activities. In D. A. NORMAN & S. W. DRAPER, Eds. *User centered system Design: New perspective on human–computer interaction,* pp. 243–263. Hillsdale, NJ: Lawrence Erlbaum.

CYPHER, A., Ed. (1993). *Watch what I do: Programming by Demonstration.* Cambridge, MA: MIT Press.

DARRAGH, J., WITTEN, I. H. & JAMES, M. (1990). The reactive keyboard: a predictive typing aid. *IEEE Computer,* **23(11).**

DARRAGH, J. J. & WITTEN, I. H. (1992). *The Reactive Keyboard.* Cambridge Series on Human–Computer Interaction, Cambridge: Cambridge University Press.

DEC (1985). *VAX/VMS DCL concepts manual.* Digital Equipment Inc, Maynard, MA, April.

DZIDA, W., HOFFMAN, C. & VALDER, W. (1987). Mastering the complexity of dialogue systems by the aid of work contexts. In H. J. BULLINGER & B. SHACKEL, Eds. *Human–computer interaction—Interact '87,* pp. 29–33, Amsterdam: Elsevier.

ELLIS, M., GREER, K., PLACEWAY, P. & ZOCHARIASSEN, R. (1987). *TCSH: Cshell with filename completions and command line editing.* Department of Computer Science, Toronto, Canada.

ENGEL, F. L., ANDRIESSEN, J. J. & SCHMITZ, H. J. R. (1983). What, where and whence: means for improving electronic data access. *International Journal of Man–Machine Studies,* **18,** 145–150.

FEINER, S., NAGY, S. & VAN DAM, A. (1982). An experimental system for creating and presenting interactive graphical documents. *ACM Transactions on Graphics,* **1,** 59–77.

GOODMAN, D. (1987). *The Complete HyperCard Handbook.* The Macintosh Performance Library. New York: Bantam.

GREENBERG, S. (1984). *User modeling in interactive computer systems.* MSc Thesis, Research Report 85/193/6, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

GREENBERG, S. (1988). *Tool use, reuse and organization in command-driven interfaces.* PhD Thesis, Research Report 88/336/48, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

GREENBERG, S. (1989). *Workbench: a metaphor for using, reusing and organizing personal computer tools.* Videotape, available from the author, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

GREENBERG, S. (1993). *The computer user as toolsmith: The use, reuse, and organization of command-based tools.* Cambridge Series on Human–Computer Interaction. Cambridge: Cambridge University Press.

GREENBERG, S. & WITTEN, I. H. (1985*a*). Adaptive personalized interfaces—a question of viability. *Behaviour and Information Technology,* **4,** 31–45.

GREENBERG, S. & WITTEN, I. H. (1985*b*). Interactive end-user creation of workbench hierarchies within a window interface. *Proceedings of the Canadian Information Processing Society National Conference,* Montreal, Quebec, June.

GREENBERG, S. & WITTEN, I. H. (1988). How users repeat their actions on computers: principles for design of history mechanisms. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems,* pp. 171–178, Washington, ACM Press. 15–19 May.

GREENBERG, S. & WITTEN, I. H. (1993). Supporting command reuse: empirical foundations and principles. *International Journal of Man–Machine Studies.* **39,** 353–390.

HALBERT, D. C. (1981). *An example of programming by example.* MSc Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June.

HALBERT, D. C. (1984). *Programming by example.* PhD Thesis, Department of Electrical Engineering and Computer Science, University of California, Berkeley, June. (Also Report OSD-T8402, Xerox Parc).

HANSON, S. J., KRAUT, R. E. & FARBER, J. M. (1984). Interface design and multivariate analysis of UNIX command use. *ACM Trans Office Information Systems,* **2,** 42–57.

HENDERSON Jr, A. & CARD, S. K. (1986). Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical interface. *ACM Transactions in Graphics,* **5,** 211–243.

JOY, W. (1980). *An Introduction to the C shell.* University of California, Berkeley, California November.

KURLANDER, D. & FEINER, S. (1990). A visual language for browsing, undoing, and redoing graphical interface commands. In S. K. CHANG, Ed. *Visual languages and visual programming,* pp. 257–275, New York: Plenum Press.

LEE, A. (1988). Study of command usage in three UNIX command interpreters. *CHI '88: Proceedings of the Conference on Human Factors in Computing Systems,* Washington, DC, 15–19 May. Interactive Poster.

LEE, A. (1990). A taxonomy of uses of interaction history. *Proceedings of Graphic Interface '90,* Halifax, Nova Scotia, 14–18 May.

LEE, A. (1992). *Investigation into history tools for user support.* PhD Thesis, University of Toronto, Department of Computer Science.

MACLEAN, A., CARTER, K., LOVSTRAND, L. & MORAN, T. (1990). User-tailorable systems: pressing the issues with buttons. *Proceedings of ACM CHI'90 Conference on Human Factors in Computing Systems,* pp. 175–182, Seattle, Washington: ACM Press. 1–5 April.

MAULSBY, M. & WITTEN, I. (1989). Inducing programs in a direct-manipulation environment. *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems,* Austin, Texas: ACM Press. 2–4 May.

MAULSBY, M., WITTEN, I. & KITTLITZ, K. (1989). Metamouse: specifying graphical procedures by example. *Computer Graphics,* **23,** 127–136. Also as Proceedings of the ACM SIGGRAPH Conference.

MYERS, B. A. (1986). Visual programming, programming by example, and program visualization: a taxonomy. *Proceedings of the ACM SIGCHI '86 Human Factors in Computing Systems,* pp. 59–66, Boston, 13–17 April.

QUERCIA, V. & O'REILLY, T. (1990). *X Windows System User's Guide.* X Window System Series, **3,** O'Reilly & Associates.

STALLMAN, R. (1987). *GNU Emacs manual.* Free Software Foundation, Cambridge, MA. March.

STALLMAN, R. M. (1981). EMACS, the extensible, customizable self-documenting display editor. *ACM Sigplan Notices—Proceedings of the ACM Sigplan SIGOA symposium on text manipulation,* **16,** 147–155, 8–10 June.

SUCHMAN, L. A. (1987). *Plans and situated actions: The problem of human–machine communication.* Cambridge: Cambridge University Press.

SUN MICROSYSTEMS (1990). *DeskSet environment reference guide.* Sun Microsystems Inc., CA, June.

SYMBOLICS (1985*a*). *User's guide to symbolics computers.* Symbolics, Inc., Boston, MA, March.

SYMBOLICS (1985*b*). Using the online documentation system. In *User's Guide to Symbolics Computers,* Boston, MA, Symbolics Inc. March.

TEITELMAN, W. & MASINTER, L. (1981). The Interlisp programming environment. *IEEE Computer,* **14,** 25–34.

THIMBLEBY, H. (1980). Dialogue determination. *International Journal of Man–Machine Studies,* **13,** 295–304.

TOTTERDELL, P., RAUTENBACH, P., WILKINSON, A. & ANDERSON, S. O. (1990). Adaptive interface techniques. In D. BROWN, P. TOTTERDELL & M. NORMAN, Eds. *Adaptive User Interfaces.* London: Academic Press.

TREVELLYAN, R. & BROWNE, D. (1987). A self-regulating adaptive system. *Proceedings of the ACM SIGCHI + GI Conference on Human Factors in Computing Systems and Graphics Interface,* pp. 103–107, Toronto: ACM Press. 5–9 April.

UNIPRESS (1986). *UniPress Emacs screen editor User's guide.* Unipress Software Inc, Edison, NJ.

VITTER, J. (1984). US&R: a new framework for redoing. *IEEE Software,* **1,** 39–52.

WITTEN, I. H. (1982). An interactive computer terminal interface which predicts user entries. *Proceedings of the IEE Conference of Man Machine Interaction,* pp. 1–5, Manchester, England. Also as Research Report 82/84/3, Department of Computer Science, University of Calgary.

WITTEN, I. H., CLEARY, J. & GREENBERG, S. (1984). On frequency-based menu-splitting algorithms. *International Journal of Man–Machine Studies,* **21,** 135–148.

WITTEN, I. H., CLEARY, J. G. & DARRAGH, J. J. (1983). The reactive keyboard: a new technology for text entry. *Proceedings of the Canadian Information Processing Conference,* pp. 151–156, Ottawa, Ontario, May.

WITTEN, I. H. & GREENBERG, S. (1985). User interfaces for office systems. In P. ZORKOCZY, Ed. *Oxford Surveys in Information Technology,* **2,** 69–104, Oxford: Oxford University Press.

WITTEN, I. H., GREENBERG, S. & CLEARY, J. (1983). Personalizable directories: a case study in automatic user modelling. *Proceedings of Graphics Interface '83,* pp. 183–190, Edmonton, Alberta, May.

WITTEN, I. H., MACDONALD, B. A. & GREENBERG, S. (1987). Specifying procedures to office systems. *Automating Systems Development Conference,* Leicester, 14–16 April.

XREOX (1985). *The Interlisp-D reference manual, Volume 2.* Xerox Artificial Intelligence Systems. April.

Appendix 1: a case study of the actual use of UNIX history

In Part 1 of "Supporting Command Usage", we studied the ways people submit commands and command lines to *csh,* the UNIX command line interpreter (Greenberg & Witten, 1993). We noted that user dialogues are highly repetitive by measuring the *recurrence rate R*—the probability that any activity is a repeat of a previous one—and finding that an average $R = 75\%$ of all command lines entered had been submitted previously. We observed that the last few command lines have a high chance of recurring—the premise behind most history systems. For example, a standard sequential history list containing the 10 previous command lines would predict the next submission around 45% of the time. Other display strategies could improve the prediction rate to a high of 55% (cf. 75%, the best possible). In short, there are certainly plenty of opportunities for reuse.

It is interesting to consider how well current history mechanisms are used in practice. This Appendix investigates how people use the reuse facilities supplied by the UNIX *csh*. Although *csh* is not a particularly good history system—its syntax is baroque and it relies on human memory—it does give an idea of how history is used.

TABLE A1
Statistics of use of UNIX csh history

Sample name	Users of history: actual	Users of history: (%)	Mean rate of history uses (%)
Novice programmers	11/55	20%	2·03
Experienced programmers	33/36	92%	4·23
Computer scientists	37/52	71%	4·04
Non programmers	9/25	36%	4·35
Total	90/168	54%	3·89

We collected real-life usage data from 168 UNIX users over a four month period [see Greenberg (1993) and Greenberg and Witten (1993) for a complete description of the data collection methodology]. Users were from one of four groups: novice programmers (first year computer science students), experienced programmers (senior computer science undergraduates), computer scientists (computer science faculty, graduates and researchers), and non-programmers (people from another faculty doing mostly non-programming tasks). During command line data collection, all *csh* history uses were noted, although the actual form of use was not. Of course, results must be interpreted carefully, for they may be artefacts arising from idiosyncrasies of the *csh* facilities rather than fundamental characteristics of reuse.

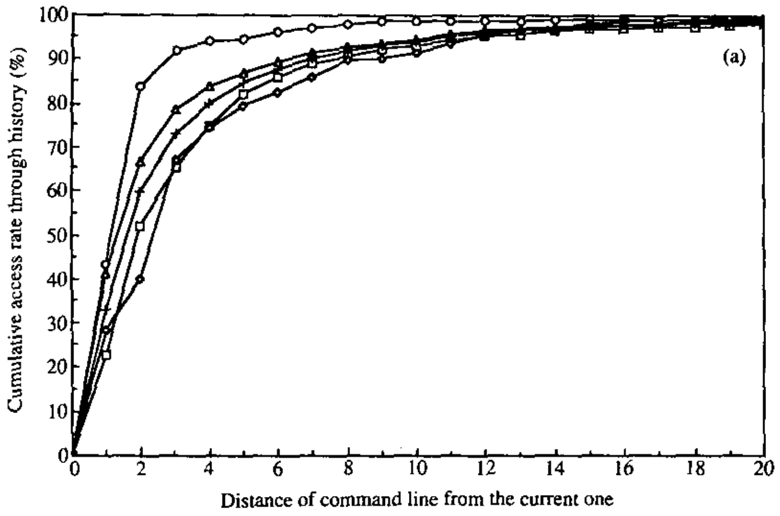
The recurrence rate R and its probability distribution, studied in Greenberg (1993) and Greenberg and Witten (1993), give a theoretical value against which to assess how effectively history mechanisms are used in practice. The average rate of re-selecting items through a true sequential history list (as used by *csh*) cannot exceed the average value of R. By comparing the user's actual re-selection rate with this maximum, the practical effectiveness of a particular history mechanism can be judged.

A.1. RESULTS

Table A1 shows how many users of UNIX *csh* in each sample group actually used history. Although 54% of all users (90 of the 168 users) recalled at least one previous action, this figure is dominated by the computer sophisticates. Only 20% of Novice Programmers and 36% of Non-Programmers used history, compared to 71% for Computer Scientists and 92% for Experienced Programmers.

The 90 people who made use of history did so rarely. On average, only 3·9% of a history user's command lines referred to an item through history, although there was great variation (S.D. = 3·8; range = 0·05–17·5%). This average rate varied slightly across groups, as illustrated in Table 2, but an analysis of variance indicated that differences are not statistically significant ($F(3, 86) = 1{\cdot}02$).

In practice, users did not normally refer very far back into history. With the exception of novices, an average of 79–86% of all history uses referred to the last five command lines. Novice Programmers achieved this range within the last two submissions. Figure A1(a) illustrates the nearsighted view into the past. Each line is the running sum of the percentage of history use accounted for (the vertical axis) when matched against the distance back in the command line sequence (the



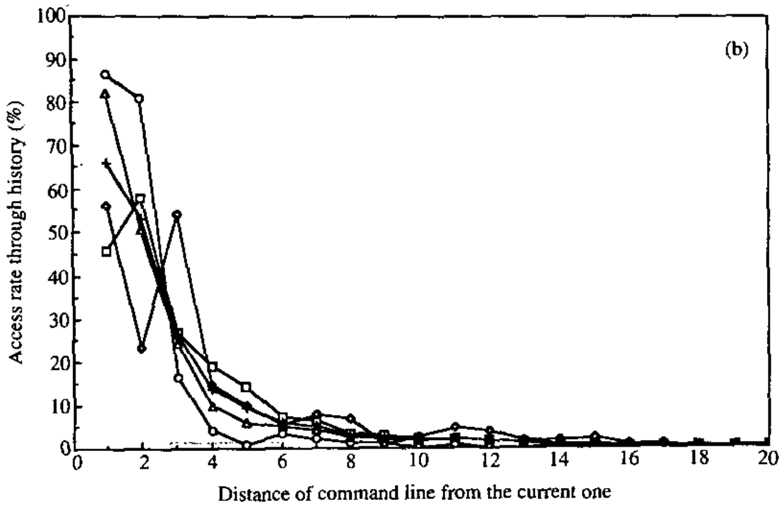


FIGURE A1. (a) Cumulative distribution of history use over distance; and (b) distribution of history use as a measure of distance. ○, novice programmers; □, experienced programmers; △, computer scientists; ◇, non-programmers; +, all subjects.

horizontal axis). The differences between groups for the last few actions (left-hand side of the graph) reflect how far back each prefers to see.†

Since most activities revolve around the last few submissions, the distribution bears closer examination. The data points in Figure A1(*b*) now represent the

† Actual figures are probably slightly higher than those indicated here, due to inaccuracies in distance estimates. As the *csh* data collection only noted that history was used and not how it was used, the actual event retrieved was determined by searching backwards for the first event exactly matching the current submission. If the submission was a modified form of the actual recalled event, the search would terminate on the wrong entry. These are assumed to be a small percentage of the total.

percentage of history use accounted for by each reference back. High variation between groups is evident. Although most uses of history recall the last or second to last entry, it is unclear which is referred to more often.

We also noticed that history was generally used to access or slightly modify the same small set of command lines repeatedly within a login session. If history was used to recall a command line, it was highly probable that subsequent history recalls would be to the same command line.

A few *csh* users were queried about history use. They indicated that they were discouraged from using *csh* history by its difficult syntax and the fact that previous events are not normally kept on display. (The latter point is important, for it reinforces the belief that candidates for reuse should be kept visible.) Users also stated that most of their knowledge of UNIX history was initially learnt from other people—the manual was incomprehensible. Also, the typing overhead necessary to specify all but the simplest retrievals made them feel that history use was not worth the bother.

A.2. CORROBORATION AND EXTENSIONS

Lee (1992) also examined history usage within various command interpreters available to the UNIX environment. Some of her findings corroborate and augment our observations.

1. There were very few uses of *csh* history.
2. Those uses made were of the simpler features, the most popular being `''!''` (retrieve the last event) and `''!!pattern''` (retrieve the most recent event beginning with the given pattern).
3. People rarely retrieved items by absolute or relative event number.
4. Although the history list is available for viewing by special request, users rarely asked to see it.
5. Modifiers for editing were rarely used. When used, they tended to be of the form `^pattern1^pattern2^`, which does simple substring replacement on the previous submission.
6. Other observed ways of modifying events were by using recalled events as prefixes or suffixes. This technique allows one to add more parameters to previous events or to add a new command sequence in a pipeline.
7. Occasional instances were noted of recalling the last word in the previous event (i.e. `!$`) and of printing events without executing them.

Lee also examined *tcsh*, another history mechanism available to UNIX users that uses a simple and familiar *emacs*-like editing paradigm to retrieve, review and edit previous events (see Section 3.3). Although better use of history is expected due to the improved editing power and visualization of the history list, only a marginal increase was noted (although the still-available *csh* history was used less). Even though the usual *csh* directives were available, people preferred the visual scrolling and editing capabilities available in *tcsh* to retrieve events. Unlike *csh*, *tcsh* users were more likely to modify the recalled events, usually by making simple changes to a few characters at the end of a command line or a word within it.

A.3. DISCUSSION

Many people never use UNIX *csh* history. Those who do tend to be sophisticated UNIX users. Yet even they do not use it much. On average, less than 4% of all submissions were retrieved through history out of the $R = 75\%$ possible! The history facility supplied by *csh* is obviously poor.

Some reasons for the failure of *csh* history follow:

1. The complex and arcane syntax discourages its use. Those who did use history indicated that only the simplest features of UNIX history were selected. As one subject noted, "it takes more time to think up the complex syntactic form than it does to simply retype the command."
2. It is hard to find out about *csh*. *Csh* history details are buried in a single on-line manual entry that runs to 31 pages(!). The text is quite technical, and examples are sparse. It is beyond the grasp of many users, particularly novices and non-programmers.
3. The event list is usually invisible. As previous events are not normally kept on display, frailty of human memory usually limits recall to the last few items. This could explain why most recalled events were to the last five command lines. However, Greenberg and Witten (1993) noticed that the last five entries could predict 34% of all submissions. In spite of memory limitations, users should have been capable of retrieving many more entries than they did.
4. It takes at least two characters to recall an event in *csh*, and often several more. Additionally, a cognitive effort is associated with deciding to use history and composing the history request. As most simple UNIX recurrences are short (6 characters on average) (Greenberg & Witten, 1993), users may feel that it is not worth the bother.

Some of these problems are characteristics of *csh* itself; the poor syntax, the terrible documentation, the invisible history list. These deficiencies hit Novice Programmers especially hard. Even though they have the highest recurrence rate of all groups and could benefit the most from history, they are effectively excluded from using it. Other problems could be general to any reuse facility, especially the tradeoff in work between recalling a submission and recomposing it.

It is premature to condemn the ideas implemented by *csh*. As mentioned above, some of the observations are likely artefacts of using a poorly-designed facility, rather than a human difficulty with the idea itself. Still, it is worthwhile commenting on some of the common retrieval methods it provides.

Retrieval through absolute or relative position. It is difficult to associate and remember the number of a previous event, as this involves an indirect reference. Visibly tagging events with numbers offers benefit only for those interfaces without direct selection and only when no better strategy is available. Perhaps its only viable use is as a redundant way of retrieving events when other selection methods are available.

Scrolling and hidden views. If events are not on display, they will not be asked for. Hidden history lists were rarely recalled. Even the preferred use of scrolling through command lines one at a time in *tcsh* only increased usage slightly (Lee, 1992). Events should be constantly visible on the display (e.g. as a menu).

Pattern matching. Simple pattern matching, especially by prefix specification, seems promising as a textual way of retrieving events. Other more complex pattern-matching methods would likely have too much cognitive overhead to make them worthwhile. Also, matching is potentially dangerous as users may accidentally retrieve and execute an interposed but undesired event that fits the specification. Again, if the last few events are visible on the display, most items could be retrieved directly rather than by indirect pattern matching.

Simple methods for recall/selection of very recent events. The syntactically simplest methods are used most to recall very recent events. For example, the ''!!'' directive was heavily used, even though it does not recall the most probable event. This likely reflects limitations of short-term memory—users restrict themselves to ''!!'' because the last item is the only thing they can both remember reliably and retrieve quickly. Overloading a reuse facility with complex functionality would not make it better.

Editing events. Although people do edit command lines as they compose them, they are less willing to modify recalled events. Often the cognitive and physical overhead of recall and editing previous events makes simple re-entry more effective. Still, some simple editing does occur (especially with *tcsh*) and probably has some value. Users should be able to edit prior events.