Greenberg S., Darragh J. J., Maulsby D., and Witten I. H. (1995). Predictive interfaces: What will they think of next? In A. D. N. Edwards (Ed.), Extra-Ordinary Human Computer Interaction: Interfaces for Users with Disabilities, pp. 103-140. Cambridge University Press, Cambridge. ISBN --521-43413-0.

6

Predictive interfaces: What will they think of next?

Saul Greenberg, John J. Darragh, David Maulsby, and Ian H. Witten

Introduction

Typical dialogues with interactive computer systems contain a great deal of repetition. Frequently used actions (commands, menu items) and objects (file names, mail addresses, icons) are invariably a very small subset of the plethora of options that are available to the user. Furthermore, action sequences such as drawing a picture or using a calculator are inherently repetitive. Even free text is redundant, because of the statistical nature of language. While repeating what was just done is a minor nuisance to the average computer user, it may be challenging for the physically disabled person, who finds every keystroke or mouse selection demanding.

System designers have taken several measures to reduce repetition, including use of terse commands, abbreviation processors, command completion facilities, and macro recorders, but these often meet with resistance from their intended audience. Brief commands are cryptic, hard to remember, and easily mixed up; abbreviation mechanisms are complex to use and take time to learn and set up; command completion is useful only when the possibilities are known in advance; and macros take time to construct and do not easily permit variables or conditionals within a specified sequence. In most cases, users must anticipate their future system usage and divert themselves from the task at hand to create the model explicitly.

Suppose the system could automatically form a model that adapts to what the person is currently doing. An underlying assumption is that what has been done before will most likely be done again. Such a model could be constructed in several ways.

- The system automatically builds it by capturing previous actions. It then predicts the entries the person is about to make, based upon the previous interactions.
- The user teaches the system with explicit instructions. Once the system is taught how to do a task, it will try to do it in the user's stead.

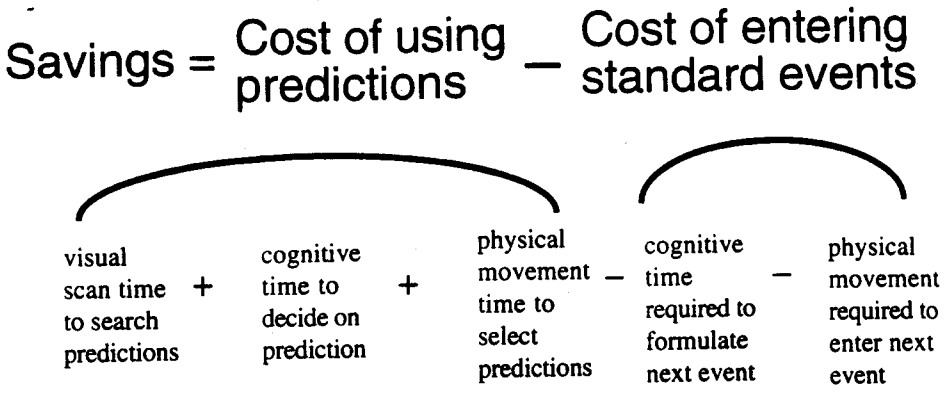


Figure 6.1. The potential savings of a predictive system, showing cognitive–physical trade-offs.

- The user provides the system with examples of the task. As specific problems and their solutions are demonstrated, the system automatically structures them into a model that applies to the more general task.

Obviously, these schemes are unlikely to predict correctly all the time, for then you could walk away and leave the computer to do your work! Given an imperfect predictor, it is essential to ensure that erroneous predictions are easy for the user to ignore while correct ones are easy to accept. This implies a *cognitive/physical trade-off*. The decrease in the number of keystrokes and mouse selections that results from a successful prediction must be weighed against the time a person takes to search visually for the prediction, decide if it is appropriate, and physically select it (Figure 6.1). If the cognitive load of choosing a prediction is high and the mechanical benefits modest, an able-bodied person will find the predictive system of little benefit. The payoff may be much greater for someone with a physical disability. This is essentially a function of the "bandwidth limitation" problem mentioned by Newell in Chapter 1 and Edwards in Chapter 11.

Suppose that in a particular situation the system has generated several (perhaps erroneous) predictions. These can be presented to the user in many different ways; the best method depends on the type of task and the actual predictions made. A conservative approach is to provide users with a list of predictions from which to choose. This reduces the user's job to a recognition and selection process and takes advantage of the fact that the user is the best judge of a prediction's suitability. The disadvantage is that a decision must be made on every prediction. A second approach is for the system to go ahead and perform

the predicted actions while the user monitors correctness, interrupting and guiding the system only when erroneous predictions are performed. A third approach is for the interface to rearrange the presentation to make more likely actions easier to take. Here predictions are presented within the language of the standard interface; no additional interface artifacts are required.

Over the last decade we have investigated a variety of predictive systems for use by a broad spectrum of computer users, ranging from physically disabled to able-bodied to graphically gifted individuals. We have been particularly concerned with several issues.

- What is an appropriate model of user activity?
- How much domain knowledge of the user's task is needed?
- What are practical predictive algorithms, and how good are their predictions?
- How can we minimize both cognitive and physical load when using predictive systems?
- What characteristics do predictive interfaces need in order to integrate smoothly with technical and interface aspects of the system?
- How can the system best be implemented on today's (and tomorrow's!) computers?

We have implemented several interesting predictive systems, ranging from one in actual use around the world to more experimental "proof of concept" systems. The systems differ substantially from each other in their task domain, intended users, algorithm for generating predictions, and user interface. While some systems were designed explicitly for users with special needs, others were not. However, we believe that all computer users are "handicapped" in one way or another (see Chapter 14), and that predictive systems should be viewed as a prosthesis that could fit both general and special needs.

This chapter introduces five quite different predictive systems (Table 6.1) and discusses the general attributes and usefulness of each. The *Reactive Keyboard* is a device that predicts free text and operating system commands based upon the text or commands entered so far. *Adaptive Menus* automatically reconfigures menu hierarchies by moving frequently selected items closer to the top. *Workbench* is a command line processor that allows people to organize and reuse their on-line activities easily. The *Autoprogramming Calculator* looks over your shoulder while you perform a repetitive calculation on a simulated calculator and tries to predict future key presses. Finally *Metamouse* is an example of a system likely to be of utility to able-bodied people as well as those with motor impairments. Metamouse is a drawing program which learns from your inputs and predicts your future actions. Reducing the number of inputs makes drawing tasks less tedious for dexterous users and more efficient for those with limited manual dexterity.

Table 6.1. *Summary of five predictive interfaces.*

System	Goal	Dialogue style supported	Predictive model	Usability
Reactive Keyboard • RK-button in MS-DOS, Unix • RK-pointer in Macintosh	Text entry for disabled people	• Command line interactions in MS-DOS & Unix, free text in Unix • Free text in point-and-click environment	Variable-length predictive models (also known as PPM – prediction by partial matching)	RK-Unix, a front end to the Unix Shell, used by many disabled people (and some able-bodied ones); comments from users encouraging
Adaptive Menus	Early demonstration of viable and testable self-adaptive interface	Ordered hierarchical menu navigation through a very large ordered name space	Menu items assigned probabilities; menu hierarchy recursively split into equal probability ranges	Controlled study showed improved performance by subjects using these
Workbench • Reuse facility • Organization facility	• Treat computer activities as tools that can be easily reused • Allow user to organize tools into a personal workbench	Command line interactions in point-and-click environment	• Design of reuse facility based on empirical study of how people repeat their activities on computers • Organization tool based on concept of situated history	• Reuse subsystem has measurably better predictions offered than current history systems • Benefits seen, but effective use tempered by several usability issues
Auto-programming Calculator	Early demonstration of the power of predictive interfaces	Specialized keypad; emulation of simple calculator	Fixed-length predictive models	Works well, but no user studies done
Metamouse	Automate repetitive editing tasks by teaching an agent	Click-and-drag graphical drawing	Explanation based learning by analyzing actions to infer constraints	Questionnaire indicated system's behavior understandable; usability study now in progress

The Reactive Keyboard

Background

The Reactive Keyboard is a program that accelerates typewritten communication with a computer system by predicting what the user is going to type next (Darragh & Witten, 1991; Darragh & Witten, 1992; Darragh, Witten, & James, 1990). Obviously, predictions are not always correct, but they are correct often enough to support a useful communication aid. Predictions are created adaptively, based on what the user has already typed in this session or in previous ones. Thus the interface conforms to whatever kind of text is being entered: English, French, Pascal, operating system commands, and so on.

The Reactive Keyboard is quite general and can be used for a variety of purposes. In some ways, it is similar to the Pal (Predictive Adaptive Lexicon) system described in Chapter 5. It is unlikely to help a skilled typist – except perhaps when entering documents such as legal contracts that contain a preponderance of standard boilerplate paragraphs. However, moderate typists will find that it assists with the highly structured text commonly found in interactive dialogues and formatted data entry, and novice and reluctant typists (including children) will appreciate the help it gives in free-text entry situations. Within its uniform adaptive mechanism it subsumes numerous features provided by interactive interfaces, such as short versions of command names, command and filename completion, user-definable abbreviations for words and phrases, menus of common operations, and brief command files. Being menu based, it can also be used to replace a physical keyboard in click-and-drag interfaces that also require some text entry. Finally, and most important, it has already found application as a user interface for physically disabled people who find a regular keyboard difficult to use.

Predictions are generated from a model of previously entered text. The modeling technique was developed for the purpose of text compression and in fact forms the core of one of the most effective known compression methods (first described by Cleary & Witten, 1984; see Bell, Cleary, & Witten, 1990, for a recent survey of the field). It builds large tree-structured models of characters in context, and for prediction these models are consulted for matches to the current context. To render it suitable for use in a communication aid, two issues had to be addressed. First, whereas in text compression the adaptive prediction mechanism feeds an encoder that generates a bit-stream representing the message, for interactive use it had to be equipped with a human interface that allows predictions to be displayed and selected. Second, a number of problems concerned with resource consumption – both adaptation/retrieval time and storage space – had to be solved to make the mechanism practical for use in an interactive personal computer environment.

Description

Several versions of the Reactive Keyboard exist. In the standard terminal/keyboard based versions, collectively called *RK-button*, predictions appear in reverse video after the cursor location, and users can accept predictions with a single keystroke or reject them by typing over them. In the window based versions, called *RK-pointer*, an ordered list of predictions appears in a separate window. Users select all or part of a prediction by pointing to the desired text. Thus the ideas in the Reactive Keyboard can be tailored to handle two different types of hardware and interface methods: typing and mouse pointing. Other hardware devices, if required, could probably be incorporated with modest effort (see Chapter 17).

An interaction with Unix using RK-button is shown in Figure 6.2a. 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 6.2b gives the meaning of a few of the control keys; in fact, many more line-editing features are provided.[1] 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.

Figure 6.2a shows the entry of five command lines. Within each of the five groups preceded by the `$` (the system prompt), each line overwrote its predecessor on the screen. Consider the first group. After the first prediction, "`mail^J,`" the user struck `^N` to show the next one. This prediction, "`cd news^J,`" replaced the previous prediction on the screen. The user accepted the first word of it, "`cd,`" using the `^W` command, moved to the next prediction, and accepted it in its entirety. The only thing remaining on the screen at this point was "`cd rk/papers/ieee.computer`." Following this, the next three commands were predicted in their entirety, while the last one required four keystrokes. The screen contents at the end of the dialogue are shown in Figure 6.2c.

In summary, five command lines comprising a total of 138 characters were entered using 11 strokes on just three function keys – an average of 2.2 keystrokes to enter each command line, or 12.5 predicted characters per keystroke. This is fairly typical for command-line dialogues with Unix. A scaled-down version of RK-button designed for MS-DOS computers achieves similar savings.

[1] The choice of control characters used originates from a PCD Maltron one-handed keyboard; while it may seem unsystematic, these functions are invariably bound to function keys on the terminal for convenience.

a

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

$ 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 (= control-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 6.2. Using RK-button, the Unix version of the Reactive Keyboard.
(a) Dialogue with Unix.
(b) Some commands.
(c) Screen contents at the end of the dialogue.

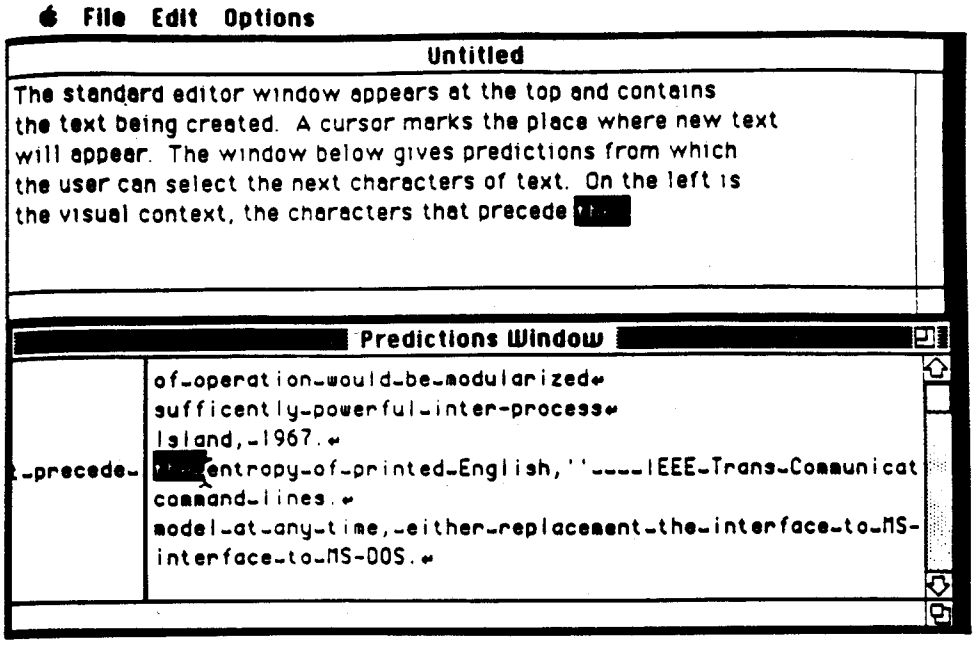
Whereas RK-button has proven most useful in enhancing the command interface to the Unix and MS-DOS operating systems, RK-pointer is designed for entering free text and runs on the Apple Macintosh computer. Full use is made of the mouse/window environment to give users convenient control over both display and acceptance of predictions – no physical keyboard is required. It is embedded within a simple text editor. Figure 6.3a shows a typical view of the

screen. The standard editor window appears at the top and contains the text being created. A cursor marks the place where new text will appear. The window below gives predictions from which the user can select the next characters of text. On the left is the *visual context*, the characters that precede the cursor in the text window. On the right is a menu of predictions which are offered as suggestions of how the context might continue. The user enters text by choosing one of these and clicking at a particular point within it. Characters up to that point are inserted into the upper window, and both context and predictions in the lower one are updated accordingly – the context moves on, and the predictions change completely. At any time the user may enter characters from the keyboard, and both context and predictions are updated as if they were selected with the mouse; thus one can use the keyboard in the normal way until useful predictions appear.

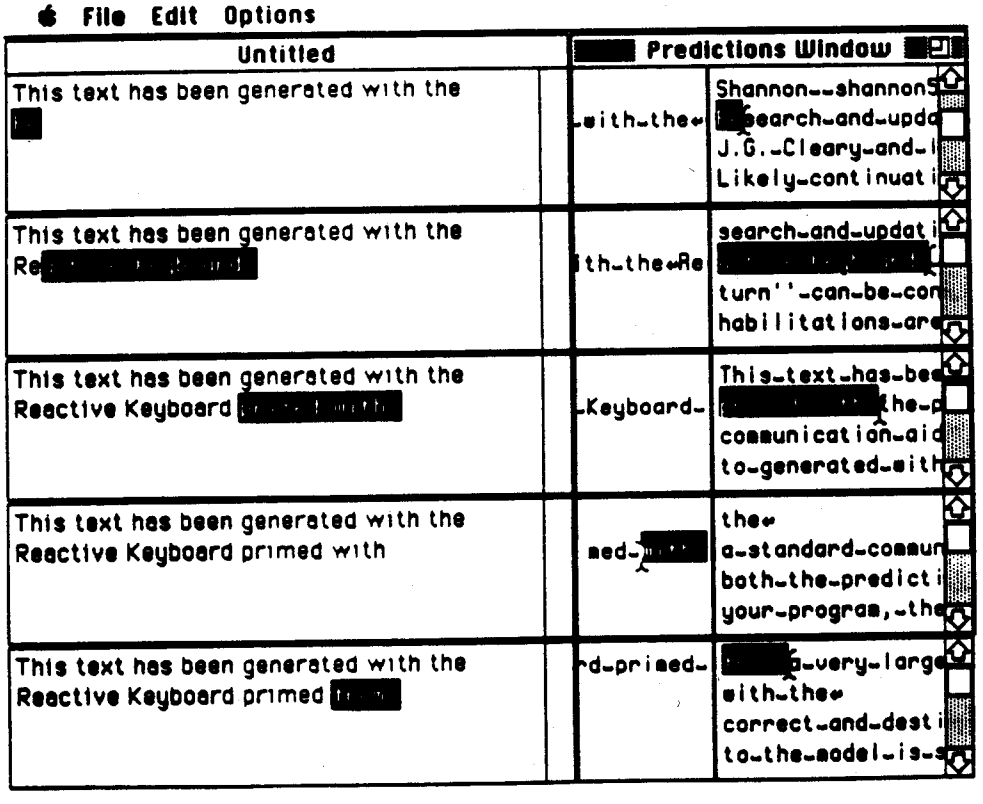
Figure 6.3b shows RK-pointer in use. The entry of several words of text is illustrated as a sequence of five screen images. (For presentation purposes, the windows are rather small and are placed side by side. In practice, all the action takes place in a single pair of windows that are considerably larger than those in Figure 6.3b.) First, the words "Reactive Keyboard " are entered. The initial two letters are taken from "Research," and to the right of the second snapshot can be seen the updated context and new predictions. At this point "active Keyboard " is entered with a single mouse click, and fresh predictions appear. Again two words – "primed with " – are entered together. The fourth image shows the effect of moving the cursor back into the context part of the prediction window: Now the last few characters of context ("with ") are highlighted and, when the mouse is clicked, deleted from the text buffer (and, of course, from the context too). The remaining illustration shows the word "from " being entered. The final result is that four words are entered in just five selections from a four-item menu, including one selection that was needed to delete an erroneously chosen word.

While it is perhaps easiest to envisage the situation where text is being entered at the end of the text window, as in Figure 6.3a, the system works equally well when the cursor is in the middle of the text buffer. Predictions are conditioned on the context preceding the cursor, and accepting a prediction inserts new characters at the cursor position.

The line termination character is shown by the carriage return icon, and in the examples predictions end when this character is encountered – the system makes no guesses about how the next line will begin. For some kinds of text, it is appropriate to predict past the end of the line. Such things are controlled by the "Options" item in the menu bar at the top of the screen, which, when clicked, reveals a preferences dialogue.



a



b

Figure 6.3. Using RK-pointer, the Macintosh version of the Reactive Keyboard.

Mechanism

The ability to guess or predict future text relies on the statistical redundancy of language. Shannon (1951) estimated that English is about 75% redundant and noted that, in general, good prediction does not require knowledge of more than a fairly small number of preceding letters of text. While for a native speaker success in predicting English gradually improves with increasing knowledge of the past, it does not improve substantially beyond knowledge of eight to ten preceding letters (see also Cover & King, 1978; Suen, 1979). The predictions in Figures 6.2 and 6.3 were generated using at most seven letters to predict the next.

The Reactive Keyboard works by suggesting on the basis of preceding inputs what the user might want to select next, exploiting stored information about past selections to predict future ones. Likely continuations are identified by locating a *prediction context* of recent selections in a large memory of previously encountered element sequences. In other words, predictions are made on the basis of the present situation, represented by short-term memory, and past experience, represented by long-term memory (Figure 6.4). The model is continually updated and adjusts itself automatically to the individual user's linguistic idiosyncrasies. The prompting display changes after each selection to present a new menu of predicted elements. A four-phase cycle of user selection, memory update, look up, and display update continues for the duration of a message composition session.

An adaptive model could be based on any of the levels of redundancy present in natural language, ranging from orthographic through syntactic, semantic, and even pragmatic (Pickering & Stevens, 1984). However, models become increasingly hard to generate and utilize at the higher levels, so the Reactive Keyboard uses a simple lexical model that is based on n-grams – consecutive sequences of n characters of text – where n is termed the "order" of the model. These, together with associated occurrence frequencies, are gleaned from representative text samples and from the user's input. The basic idea of prediction is to use the first $n - 1$ letters – the prediction context – to predict the nth). The model's predictive power depends on how accurately its experience matches the user's current communication needs, and on the number and length of stored sequences.

An important innovation in the Reactive Keyboard is that likely continuations for the current context are sought on a longest-match basis. If no elements in memory match the full prediction context, the context is truncated until a match can be found. This proves extremely effective, because it is capable of predicting from more complete lower-order models if less complete higher-order models lack instances of the current context. All lower-order models are implicit in the highest-order model, so no extra storage is required. Such models have been found to be extremely effective for text compression (Bell, Cleary, & Witten, 1990; Cleary & Witten, 1984).

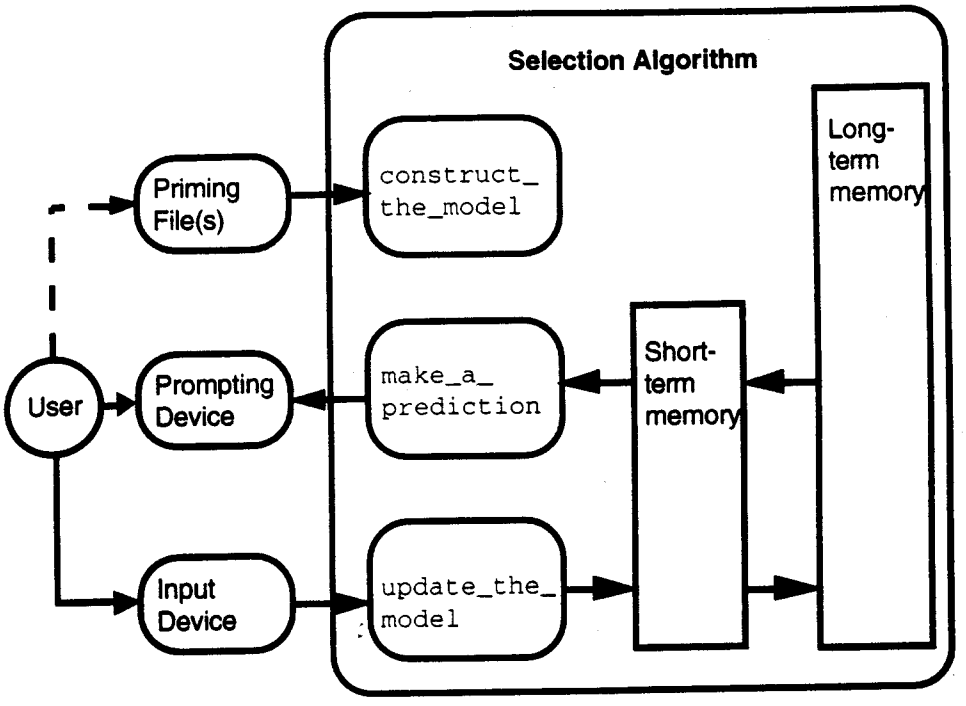


Figure 6.4. Overview of the Reactive Keyboard system.

By far the most important factor affecting the quality of predictions is the text used to prime the model. Finding truly representative text is difficult, and the statistical characteristics of seemingly similar samples can vary greatly (Gibler, 1981; Kucera & Francis, 1967). This implies that the text used for priming must be carefully selected to be as representative as possible of what the user wants to generate. The Reactive Keyboard derives its model in three ways, using a combination of automatic and explicit modeling techniques. As shown in Figure 6.4, priming occurs (a) automatically from a default (or user specified) startup file, (b) automatically from current user inputs, and (c) explicitly from any text file the user chooses to add into the model.

Experience

The predictive terminal emulator of Figure 6.2 has been in use for some time by two physically disabled students at the University of Calgary as their standard command interface to Unix. Their combined daily experience amounts to approximately eight years. One types with a single partially paralyzed hand and uses the system for entering commands and electronic mail. He estimates that

over a two-year period it has provided assistance on over 30,000 host system commands, averaging 10 predicted characters per command, and writes:

> The Reactive Keyboard has dramatically changed the way I use computers. I now use much longer, more descriptive, file names than I otherwise would have without its reliable recall and typing assistance. I also rely completely on RK-button to remember such things as electronic mail addresses and long complex command-line sequences. Life on-line would just not be the same without it.

The other user has a progressive neuromuscular disorder and is extremely slow and otherwise unable to write. He uses RK-button as a general command interface and writing aid.

> I find the Reactive Keyboard to be an extremely beneficial tool for typing. Since I have severe neurological damage in my hands, it seems to cut the time I spend coding manyfold. To illustrate this, I need only inform you how I mailed John this letter. All that was required was my typing "`ma,`" after which time it predicted "`il darragh^J`." So, I was able to type and enter a command normally requiring thirteen keystrokes in but three! This saves much time since it is rather difficult for me to access some keys on the board. It must be remembered that both my hands and fingers move slowly and inaccurately.

The system has recently been released on the Usenet network, and the following unsolicited comment is typical of those received:

> I have cerebral palsy, so my typing is a little bit impaired. I am very impressed with your program. It seems to be ideal for significantly speeding up my typing in the shell while I am programming. I like it a lot, and it saves me a lot of time. I am very glad to have it available to me. The other people that I have showed it to are also very impressed.

The Unix version of RK-button has been tried, tested, and improved with the experiences of people who use it every day. As with any tool that saves people effort, using RK-button has become a matter of course for many users. The occasional experience of using systems that are not equipped with it drives home forcefully how helpful it is.

The IBM PC version of RK-button presents a similar interface to MS-DOS. However, it was completed more recently and is not yet as widely used as the Unix version. RK-pointer on the Macintosh is also relatively new and was designed more as a demonstration program than as a fully integrated application; consequently there is little user experience to report. This version would

probably be more useful if it were reimplemented as a desk accessory and could be used with arbitrary Macintosh applications.

All three versions of the Reactive Keyboard are available free of charge. These systems are provided in source form so that they can be modified to suit different people's requirements. Users are encouraged to make and distribute as many copies as they wish. Recipients are encouraged to report any modifications they make so that others will be able to benefit.[1]

Adaptive menus

Background

Hierarchical menus, where users reach an item of choice by navigating through a large tree, are the standard interface for many interactive systems. Although "hot keys" and other methods can speed access to particular items in the tree, their usefulness is limited to menu hierarchies where a few items are consistently chosen over and over again. When a user's choice of items in a hierarchy varies over time and over situations, alternative strategies must be used for accelerating item selection.

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 a user must make to select an item without adding further paraphernalia to the user interface (Greenberg, 1984; Greenberg & Witten, 1985; Witten, Cleary, & Greenberg, 1984; Witten, Greenberg, & Cleary, 1983).

Description

Consider a computerized telephone directory of names presented through a menu hierarchy. Normally the hierarchy would be constructed by splitting the name space into equal units to provide a tree with a balanced number of leaves at every point. Such menus are "frozen," and the user must always navigate the same path to reach a friend's phone number, regardless of the number of times that same call has been made.

[1] Up-to-date versions are available electronically via "anonymous ftp" from "cpsc.ucalgary.ca" (Internet 136.159.2.1) or by writing to the authors at the University of Calgary (E-mail darragh@cpsc.ucalgary.ca).

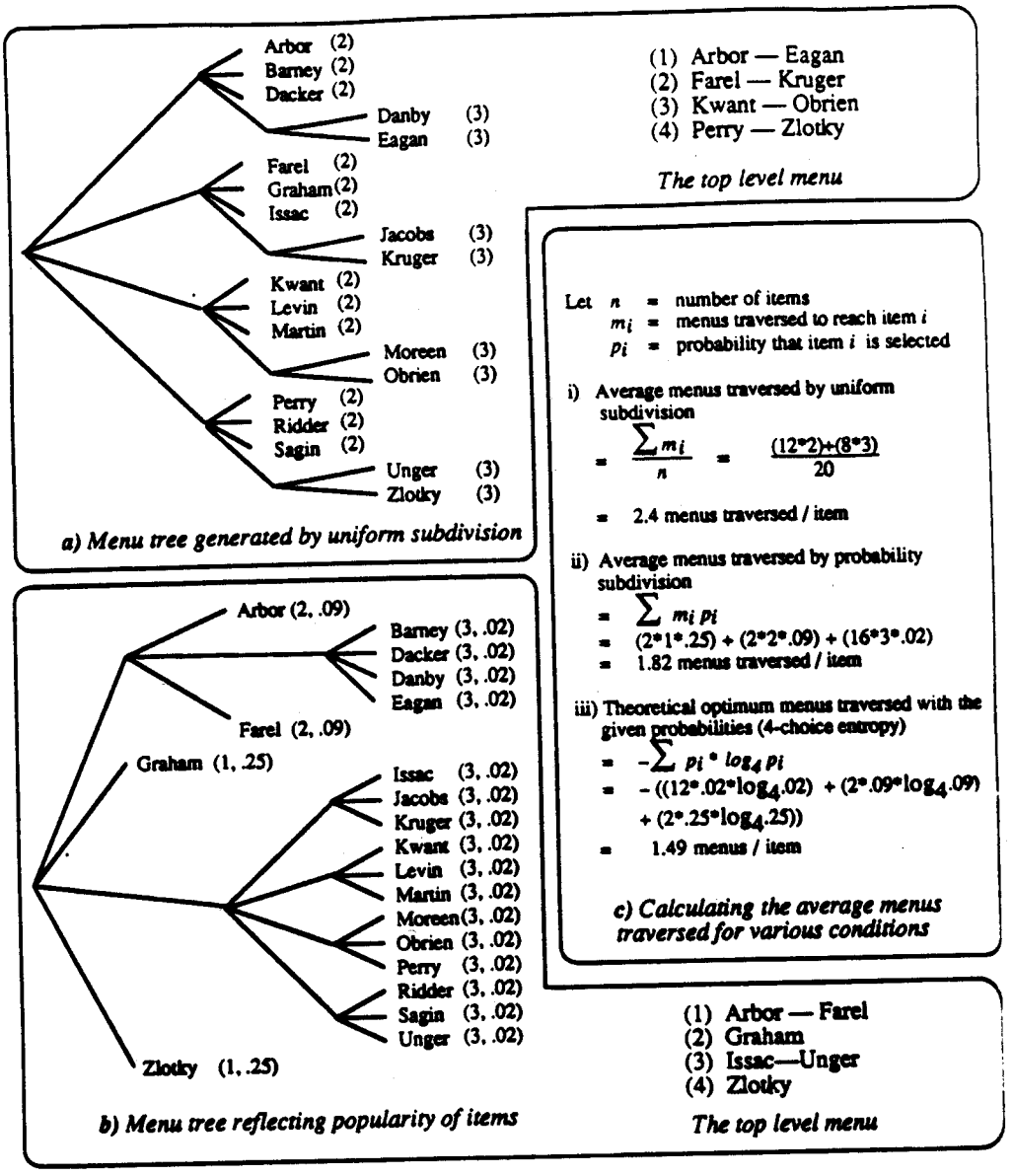


Figure 6.5. Menu trees generated by uniform and probability subdivision.

However, the access frequencies of names, combined with their recency of selection, defines a probability distribution which reflects the "popularity" of the names selected. Instead of selecting regions at each stage to contain approximately equal numbers of names, it is possible to divide the probability

distribution into approximately equal portions. As the program is used, 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 been selected already – especially if they have been selected often and recently – than to those which have not. The key point is that the assigning of probabilities is dynamic; the hierarchy presented adapts to the user's evolving (and ever changing) usage over time.

Figures 6.5a and 6.5b depict two menu hierarchies for a very small dictionary with 20 name entries (left side of box) and their corresponding top-level menu (right side of box). Figure 6.5c calculates the average number of menus traversed per selection. In Figure 6.5a, 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 6.5b 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, and the remainder are relegated to the third level. In this particular case, the average number of menus traversed is less with probability subdivision (Figure 6.5c, point ii) than with uniform subdivision (point i), although this improvement is not as much as is theoretically possible (point iii).

Mechanism

Two factors are critical to the performance of Adaptive Menus: how probabilities are updated over the name space as items are retrieved, and how the probability space is split into a hierarchy that minimizes the average number of transitions to names.

The algorithm for updating probabilities as items are selected is shown in Table 6.2. It is based on several heuristics derived from a study of telephone usage statistics (Greenberg, 1984). First, we ensure that first-time access to entries is not noticeably longer than in a nonpersonalized system. We do this by deciding in advance the maximum number of selections that can be tolerated for worst case access to new items as compared to the uniform distribution. This is implemented as a weight $\mu > 1$ that indicates the maximum multiplicative increase in menus traversed. Second, when a new "friend" is accessed, we immediately increase the probability associated with the friend on the assumption that she or he is now much more likely to be selected again. Third, the probability associated with friends who are not accessed decays slowly (the constant α), thus making room for new or more popular friends. It is the decay factor that builds in a way of balancing frequency and recency. Whereas low decay will see frequently chosen items migrate up the tree, a high decay rate gives more room to recently chosen items. Finally, when a friend is accessed, we

Table 6.2. *Retrieval and update algorithms for assigning probabilities*

Let N be the total number of people on the list and F be the total number of friends. Keep a list of pointers to friends, f, who have already been accessed, together with a weight $\omega(f)$ for each in the range [0,1]. The update algorithm will ensure that the weights sum to 1 over all friends. The parameter α is a measure of how quickly friendships fade away and should be slightly less than 1; the closer α is to 1, the slower friendships will fade. The constant μ, with $\mu > 1$, determines the maximum number of menus traversed that can be tolerated for names that are not treated preferentially (the new or infrequently chosen items). The closer μ is to 1, the closer the number of menus traversed to these names will be to that provided by the uniform distribution.

Retrieval

If F = 0, probability for each name is $\frac{1}{N}$

otherwise,

if friend, probability is $\frac{1}{N\mu} + \omega(f)\left(1 - \frac{N}{N\mu}\right)$

else probability is $\frac{1}{N\mu}$

Update

If a new friend has been made, put on the friend list and update F.

Then

initialize the friend's weight to $\frac{1}{F}$

decrease all other weights by $\frac{1}{F(F-1)}$

If an old friend f has been accessed again,

$\omega(f) \leftarrow \alpha\omega(f) + 1 - \alpha$

$\omega(g) \leftarrow \alpha\omega(g)$ for all other friends $g \neq f$

In either case, check if any $\omega(f) \leq \frac{1}{N}$; if so

delete f from the friend, decrementing F

increase weights of other friends g by $\frac{1}{F}\omega(f)$

increase the probability associated with him or her to reinforce his or her popularity. The full algorithm is explained in Greenberg (1984) and in Witten, Greenberg, and Cleary (1983).

Given a probability distribution, it is surprisingly difficult to construct a menu hierarchy that minimizes the average number of selections required to find a name. Despite the apparent simplicity of this problem, optimal menu construction is possible only through exhaustive search over all menu trees, an approach that is infeasible for all but the smallest problems (Witten, Cleary, & Greenberg, 1984). Fortunately, the simple processes of iteratively splitting the probability distribution at each stage into regions whose probabilities are as similar as possible achieves good (but not optimal) performance in practice. For

example, if seven menu items are allowed per page, we would divide the entire probability distribution into seven equal parts. We would then recursively subdivide each portion of the distribution until only leaves are left.

The implementation of the iterative splitting algorithm introduces the interesting side effect that popular names appear at the splitting boundaries. The advantage is that during presentation of the menus, popular names frequently occur as a range delimiter in menus high in the hierarchy, making the user's search process easier. An example is shown in Figure 6.5b, where a user may be searching for Farel. As Farel appears as a range delimiter on the first-level menu, the user's task of deciding which range (and corresponding subtree) must be navigated is made far simpler.

Experience

With Adaptive Menus, previous actions are almost always resubmitted in fewer keystrokes, as the path through the hierarchy is shorter. Also, since no extra detail is added to the interface presentation, there is no need to learn a new subsystem, and screen use is minimized. The trade-off is that since paths change dynamically, memory cannot be used to bypass the search process. Users must now search the menus for their entries all the time, even for those accessed frequently. However, this is not a problem in practice.

Human factors experiments using able-bodied subjects were performed to compare Adaptive Menus with their nonadaptive counterparts (Greenberg & Witten 1985). Subjects simulated one month of telephone usage in a one-hour session, where names were retrieved from the menu for each simulated call. The average time taken to "dial" a number was reduced by an average of 35% when Adaptive Menus were used. In addition, as subjects did not have to descend deep into the tree (the place where most errors occur), the average number of errors decreased by 60%. When one considers the high penalty of errors in real-life applications – users going down incorrect branches not only lose time but also suffer the risk of getting lost in the hierarchy – then the significance of reducing errors becomes apparent. Also encouraging is that subjects preferred Adaptive Menus to the nonadaptive ones, mostly because of the generally shorter search paths and the frequent appearances of popular names as range delimiters.

Workbench

Background

Top-level interfaces to general purpose computing environments are designed to help people pursue a wide and varying range of tasks by providing them with a rich set of tools and materials. In a command based system, for example, one

invokes an action by typing simple commands and arguments, although some modern systems augment or replace this primitive dialogue style with menus, forms, natural language, graphics, and so on (see Witten & Greenberg, 1985, for a discussion of interface styles).

Users repeat activities that they have previously submitted to their computers surprisingly frequently (Greenberg & Witten, 1988b). Yet reformulating the original activity can be both difficult and tedious, especially for someone with a physical impairment. 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 are two ways to make it easier to reformulate old activities. One is through *reuse*: predicting those activities that are likely to be repeated and placing them ready to hand. The other is through *organization*: allowing people to arrange their command sets in a manner that fits their task. Both facilities have been designed into a system called *Workbench* (Greenberg, 1988).

Description

Workbench is a graphical window-based front end to the Unix command line interpreter. Using the metaphor of a handyman's workbench, it provides both a list that predicts future submissions from old ones and a way for users to structure and store submissions for later use. Its major visual components are detailed below and illustrated in Figure 6.6, which shows one paned window and two pop-up windows.

The *work surface* is a terminal emulator running the standard Unix command line interpreter (the bottom pane in Figure 6.6). This is the main working area of Workbench, and users can submit command lines by typing them directly.

The *reuse facility* is the list from which the user can select, edit, and insert into the work surface an old command line entry (middle pane in Figure 6.6). It implements a policy of temporal recency, where the last few user submissions are considered good candidates for reuse. This is commonly known as a *history list*. 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.

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

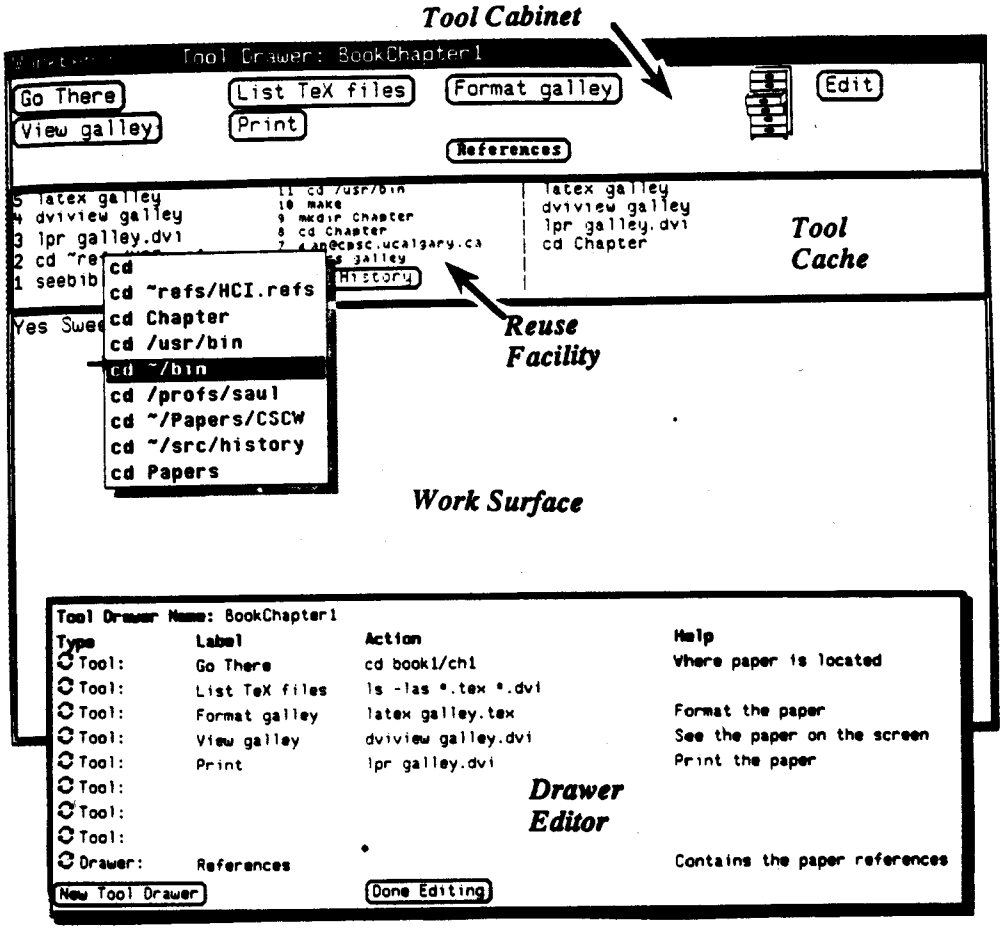


Figure 6.6. Workbench in use, showing the main Workbench window and two pop-up windows.

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 labeled with distinctive text fonts (top pane in Figure 6.6). A tool has three components:

- a Unix command that will be executed when the button is pressed,
- an optional text label for the button, and
- 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 user defined. When the edit button is selected, an editable representation of the current drawer appears (lowest pop-up window in Figure 6.6). 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. Items in the reuse area are not only excellent candidates for becoming more permanent tools, but are easy for a person to gather into a task specific drawer. Users are also invited to document their new tools by adding a descriptive label and help message.

Mechanism

An empirical study motivated the design of Workbench and in particular its reuse facility (Greenberg, 1988; Greenberg & Witten, 1988a, 1988b). An analysis of the interaction records of 168 Unix users revealed that 75% of a user's online activities are repeated submissions. However, finding and selecting items for reuse can sometimes be more work than entering them afresh, particularly if it is necessary to search a complete list of previous entries. To be practical, a reuse facility must choose a small set of previous submissions as predictions and offer only these to the user. The difficulty, of course, lies in choosing good predictions.

A further study contrasted the effectiveness of various predictive strategies: recency, frequency, partial pattern matching, treatment of duplicate items, and using commands as hierarchical entry points to past arguments. The results indicated that the temporal recency of previous submissions is a reasonable predictor of the next one – the user's recently submitted activities are the likeliest to be repeated, particularly if duplicate items are shown only in their most recent positions on the list.[1] The recency policy is cognitively attractive, because users generally remember what they have just entered and can predict what the system will offer them. Predictive power is further increased when items are used as hierarchical entry points to past arguments (implemented as the pop-up menu in Figure 6.6).

Using the strategies above, a list of ten entries successfully predicts 55% of all user activity, with a potential keystroke saving of around 6 characters per selected prediction, or 3.3 characters over all submissions (including new ones). An optimal reuse facility could predict at most 75% of all activities (the percentage of repeated submissions), which results in savings of 4.4 characters over every reused and new submission. Thus the predictions offered by our

[1] In contrast, frequency ordering and presenting duplicates in their order of first appearance are quite poor predictive strategies.

method (55%, 3.3 characters) make up roughly 75% of the theoretical maximum saving (75%, 4.4 characters).

The design of the tool cache and tool cabinet is supported by the idea that plans are derived from *situated action* – the necessarily ad hoc responses to the contingencies of particular situations (Suchman, 1987). In Workbench, the elements of a plan are the tools contained in a drawer. By drawing on the reuse facility as a primary source of tried and tested candidates for new tools (something we call *situated history*), a person can rapidly create, annotate, and modify his or her personal workspace so that it accurately reflects the task at hand. Placing a tool in a drawer is just a matter of dragging it in from the reuse facility – there is little disruption to the user's task.

Workbench is implemented as a stand-alone process that can interoperate with any application. It maintains a communication port, and applications are invited to send any tokens that should be displayed on the reuse facility. While this requires each application to have limited knowledge about Workbench (how to establish communications and the protocol of sending tokens), the applications do not control or need to know anything about its internal behavior. At the same time, Workbench allows any application to connect to it and does not need any syntactic or semantic information about the tokens it receives.

Experience

Workbench was constructed in Suntools, a window system that has now been eclipsed by the newer X Window standard. As a result, we have limited user experience with it. However, several colleagues who used Suntools enjoyed having Workbench available, although their enthusiasm was tempered by the usability issues outlined below.

First, difficulties arose because Workbench is a bolt-on front end to a single application. Ideally, all applications in a user's environment would cooperate with Workbench, each being responsible for collecting, massaging, and passing on relevant user input for display. However, this is difficult to achieve in a Unix environment, because the source code of every application would have to be altered. Even if source code were available, the task of modifying just a few key applications would be daunting. The result was that Workbench worked only with the Unix command line interpreter. It was particularly frustrating to have the reuse facility turn off just because one had entered a different application (a text editor, for example). An integrated system incorporating reuse primitives in every application would have to be designed at the operating system level.

Second, some cognitive and physical trade-offs come into play. On the cognitive side, the user must still decide what items to select from the display and undergo the process of constructing and maintaining the Workbench organization. In practice, users gain considerable benefit from having situated and well-formed actions available in the reuse facility. History items are selected

for reuse and collected in drawers. Still, some modification is usually required to verify and generalize the action, and the entire process could be simplified (see Greenberg, 1988, for further discussion of the design issues). We also found that it takes a conscious effort to switch from performing a task to using predictions, and users must do the extra work of moving from the keyboard to the mouse, selecting items, and, if necessary, navigating drawers. But once the switch was done, users expressed pleasure in being able to perform their actions simply and economically by making Workbench selections.

The Autoprogramming Calculator

Background

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 sequence. The goal of programming by example (Myers, 1986) is to allow sequences and more complex constructs to be communicated concretely, without the user's resorting to abstract specifications of control and data structure (in a programming language, for example).

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 frequently one has 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. Ideally, programming-by-example strategies should 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. While completely automating the general programming process is impossible, it can be done in simple enough situations. Using a calculator (rather than a programming language) sidesteps some of the more difficult problems of automatic programming, for the calculator is suited to simple jobs.

Many programming actions, such as those performed on a simple calculator, are quite repetitive. A long function must be continually reentered even though

only a few arguments (the variables) will differ. Whereas advanced calculators (and computers) could be programmed to do a mathematically repetitive task, this may be beyond the skill of particular calculator users.

Instead, the Autoprogramming Calculator looks over your shoulder while you perform a repetitive calculation where the calculator would try to predict the next key pressed (Bell, Cleary, & Witten, 1990; Darragh & Witten, 1992; Witten, 1981). The idea is that if a computer is shown the steps to perform a task a number of times, it can automate the process by building a model – a program – of the sequence of steps. Entries and operations that are the same each time are considered constants and will be predicted. Any that cannot be predicted correspond to "input" that the system must get from the user. The different input data provided each time will often cause the task to be executed in different ways, adding new paths to the model. Once the model is good enough, it can predict the next step at all times and be left to run its "program" all by itself. Thus the calculator will behave exactly as though it had been explicitly programmed for the task at hand, waiting for the user to enter a number and simulating the appropriate sequence of keypresses to come up with the answer.

Description

The Autoprogramming Calculator, modeled after a simple Casio calculator, was built in the early 1980s to demonstrate the power of predictive interfaces. It constructs an adaptive model of the sequence of keys the user presses. If the task is repetitive (like computing a simple function for various argument values), the modeler will soon catch on to the sequence and begin to activate the keys itself. Inevitably the prediction will sometimes be wrong, and an "undo" key allows the user to correct errors.

Figure 6.7 gives some examples of this "self-programming" calculator. The first sequence in Figure 6.7a shows the evaluation of xe^{1-x} for a range of values of x = 0.1, 0.2, 0.3, and 0.4. The keys pressed by the operator are in normal type; those predicted by the system are shaded. From halfway through the second iteration onwards, the device behaves as though it had been explicitly programmed for the job. It waits for the user to enter a number and displays the answer. It takes slightly longer for the constant 1 to be predicted than the preceding operators because numbers in a sequence are more likely to change than operators. Therefore the system requires an additional confirmation before venturing to predict a number.

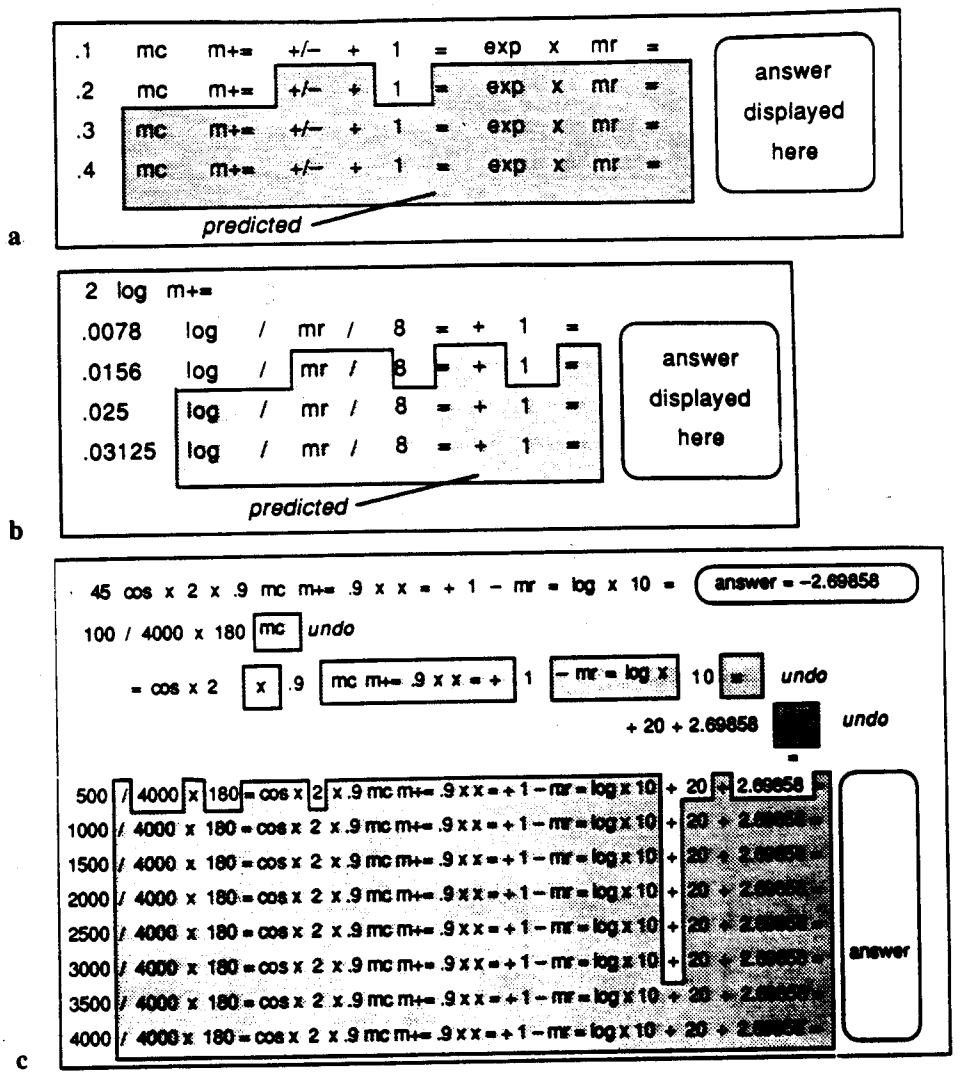


Figure 6.7. Example of operating the Autoprogramming Calculator for various values of x.

Figure 6.7b shows the evaluation of

$$1 + \frac{\log x}{8 \log 2}$$

for various values of x. The first line stores the constant "log 2" in memory. More complicated is the evaluation of

$$20 + 10 \log \left[1 + a^2 - 2a \cos \frac{180x}{4000} \right] - 10 \log \left[1 + a^2 - 2a \cos 45 \right]$$

for $a = 0.9$ (Figure 6.7c). Since the calculator possesses only one memory location, it is expedient to compute the last subexpression first and jot down the result. The result of this calculation (2.69858) has to be keyed only twice before the system picks it up as predictable. Some interference occurs between this initial task and the main repeated calculation, for three suggestions had to be "undone" by the user. The negative effect of one of these undo's continues right up till near the end of the interaction. This means that the penultimate plus sign on each line has to be keyed by the user several times to counter the system's reluctance to predict it.

Mechanism

The Autoprogramming Calculator uses a fixed length (length-k) predictive model, which is a limited-context state machine created from the set of k-tuples that occur in the input string (the input sequence) being modeled. Context models effectively split an input sequence into overlapping substrings of length k, where $k - 1$ is the context length. The first $k - 1$ characters of each substring predict its last character. The sequence of k-tuples that occurs in a particular string fully characterizes the string.[1]

When a new input symbol appears, a k-tuple is created with the new symbol at its end. Given an existing state model formed from previous input symbols, the tuple must now be incorporated into it. If that k-tuple has never occurred before, a new state is added to the model and labeled with the new symbol, with an appropriate transition leading to it. (A transition out of the state will be generated when the next symbol is processed.) If the k-tuple has occurred before, it uniquely determines a state of the model. However, it may call for a new transition to be created into that state. If so, it will be necessary to reevaluate the model by expanding it around the state and by adding the new transition. The model must then be reduced to ensure that extraneous sequences are excluded. A complete description of this process is found in Bell, Cleary, and Witten (1990, chap. 7).

1 Length-k modeling is a more general but fixed length form of the variable-length modelling technique used in the Reactive Keyboard. Another difference is that the calculator uses a confidence parameter to forestall prediction of numbers, where numbers require more evidence than operators (Darragh, in press).

Experience

The Autoprogramming Calculator was built as a research tool and has not been released for general use. Still, the experience of designing and building the calculator laid the foundations for the Reactive Keyboard, and several lessons have been learned that can be applied to programming by example in general.

First, there is a fine line between "teaching" the system and "providing examples" to it. As Figure 6.7c illustrates, a user must be conscious of how a problem should be broken down in order for the system to learn it, and must minimize variation between examples by presenting sequences consistently. The user must also show all steps to the system; performing some calculations in one's head may represent a mental leap that the system cannot bridge (Witten, MacDonald, Maulsby, & Heise, 1992).

Second, the interface presented to the user is critical for a system's success. It is important for the user to understand what predictions the system has made and for the system to present the predictions in the language of the interface. For example, the calculator would ideally show the formula learnt so far and indicate where the input token accepted from the user would be applied. Also important is the need to minimize the work required of the user. The Autoprogramming Calculator, for example, aggressively predicts an input sequence. When predictions are correct, the user just enters whatever arguments are required. When incorrect, the user undoes the prediction and corrects it.

If programming by example proves successful, all users will benefit. The dreary reentry of repetitive program sequences will be reduced to having the user supply only the critical arguments. Also, the cognitive effort of a user's mentally forming and reforming each example task will decrease as the system learns and takes over the task.

Metamouse

Background

Metamouse is an "instructible" interface that predicts repeated actions in graphical editing tasks (Maulsby, Witten, & Kittlitz, 1989). Creating and reformatting structured drawings involves not only repetition but input precision at the pixel level. Such tasks are tiresome and error prone because they require many repeated and highly accurate movements. For users who have some neuromuscular or visual difficulty, editing a drawing by direct manipulation is exhausting toil, if it can be done at all. Metamouse helps the user by learning iterative procedures from single demonstrations and by inferring precise constraints from approximate actions. The cost of automation is a little extra intellectual effort, since the user must focus Metamouse on relevant spatial relations while editing. Although teaching the computer requires more thought,

the user – particularly the physically disabled person – is rewarded by being relieved of tedious work: A tiresome task becomes easy, and the next-to-impossible becomes feasible.

Most commercial drawing programs support very limited kinds of graphical constraint, such as a grid for placing points and commands for aligning objects. Systems like *Gargoyle* (Bier & Stone, 1986) have general capabilities but complex interfaces. Metamouse was designed to do three things with a very simple interface: infer primitive (system defined) constraints from single actions; enable the user to invent more complex constraints as construction procedures; and match action sequences in order to predict loops and branches. Owing to its use of domain-specific knowledge in generalizing a situation, the predictive interfaces Metamouse most resembles are Witten's Autoprogramming Calculator, *Tels* (Witten & Mo, 1993), and *Eager* (Cypher, 1991).

Description

Metamouse was designed for programming by demonstration (Myers, 1988). The user teaches an agent a task by performing it, now and then issuing simple instructions to focus attention and correct mistaken inferences. The agent's persona is Basil, a nearsighted turtle who follows the user step by step and highlights graphical constraints it judges important. "Constraint" in this system means a point of contact between two objects. Thus, to show Basil some spatial relation, the user might have to construct a sequence of touch constraints by drawing one or more objects connecting the related ones.

Figure 6.8 illustrates a session with Basil in which the user's task is to align a set of boxes to an input guideline, while keeping each box at the same vertical coordinate; the final result is shown in Figure 6.8i. The user creates a guideline (step a), then selects and drags the first box to it (b, c); the black tack indicates a touch constraint Basil has inferred. When the user picks the second box (d), Basil matches this with step b and predicts a repetition. She drags the second and third boxes to achieve the same touch constraint (f, g). The user's only input for these actions is to click "OK" to approve them. When Basil fails to find another box, he terminates the loop and asks the user to take over (h). The user removes the guideline and tells Basil to save the procedure under the name "align to guide."

Basil uses three learning strategies. The first employs domain knowledge to "explain" individual actions. Since Basil knows about touch constraints but not other spatial relations, the user has to draw a line to express alignment in terms of touch (step a). An action not resulting in touch constraints, such as drawing the guideline itself, is assumed to be governed by a potential input or constant. The second strategy uses multiple examples to select hypotheses consistent with them. For instance, when predicting an action whose location may be constant or input, Basil performs it using the same location user demonstrated (proposing a constant) but invites the user to alter it (checking for input). The third strategy is

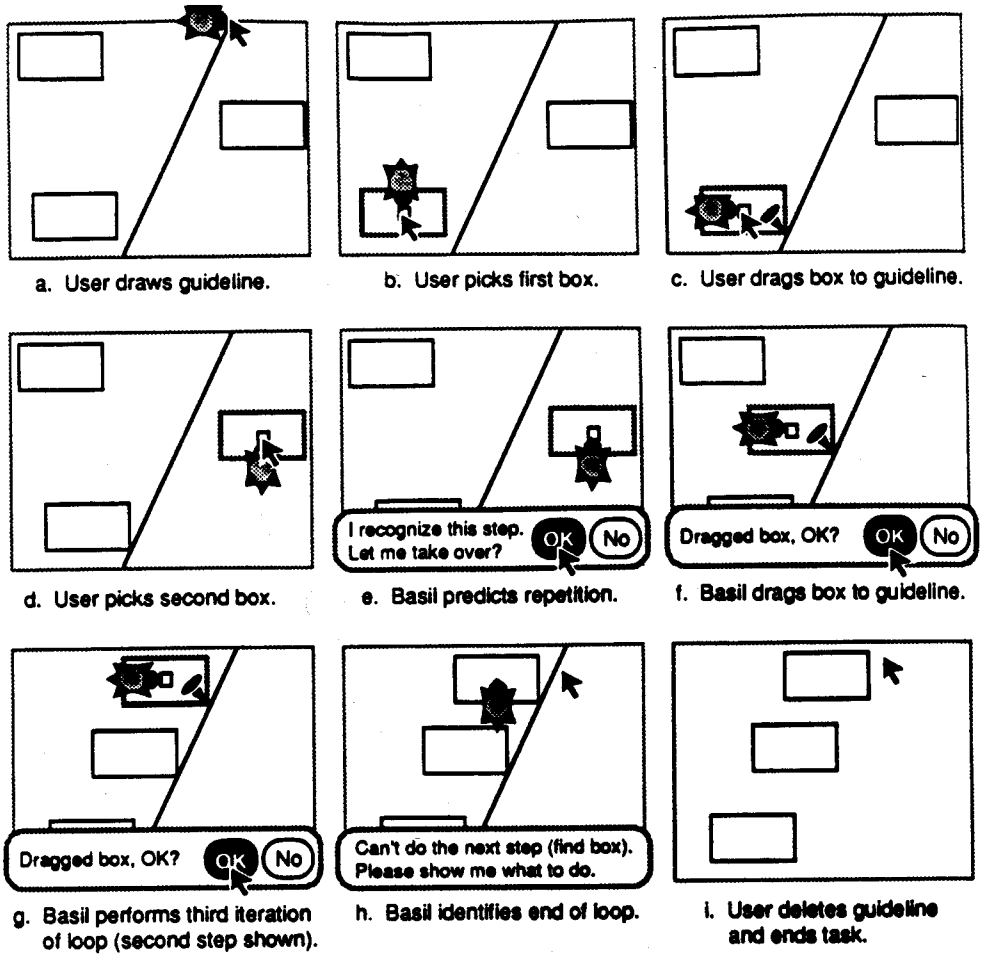


Figure 6.8. Using Basil to align a set of boxes.

to get direct instructions from the user. The system highlights inferred constraints and predicts actions step by step: the user can reject these inferences, causing Basil to try an alternative. The current implementation does not afford all three strategies for all aspects of a task.

Mechanism

Focusing inference on graphical constructions rather than general spatial relations was believed to be a viable approach because people find it natural to express constraints procedurally – they envisage spatial relations with imaginary lines and motions and convey them by gestures. Van Sommers (1984) found in

studies of drawing on paper that people typically orient new components of a drawing to old ones according to touch.

The problem tackled by Metamouse is to predict, from examples, the constraints on selecting and positioning primitive two-dimensional geometrical objects. In both cases, the only example features considered are:

- the object type (line, box) and pre-defined parts (edges, vertices, centers);
- whether the target object is a specific individual or a member of a set;
- the general direction of search over a set of objects (up, down, left, right);
- touch relationships between parts of objects.

For each new action, the system substitutes variables for objects and identifies constraints on position. Constraints are chosen by explanation-based generalization (deJong and Mooney, 1986), using a weak theory of 15 rules. If the action's constraints match those of some previous step, Basil predicts that step's successor. A task is represented as a set of production rules.

The following subsections describe the kinds of inference the system makes and the instructions by which a user can modify them. The examples are taken from Figure 6.8.

Object selectors

When the user selects the first box (step b), Basil infers a variable to stand for it and a selector function to set the variable:

> *BoxVar1 := find-novel-object(prevBindings = nil, typeBox, pathDownward).*

When predicting, this selector will choose the next box farther down that has not been assigned to *BoxVar1* already. The *find-novel-object* selector permits iterating once over a set of objects, processing them in the order of their occurrence along an axis. The other selector, *use-value-of(X)*, picks the object currently assigned to some variable *X*. The current implementation does not let the user modify Basil's choice of selector.

Directions

One parameter of *find-novel-object* is the direction of search, *pathDownward*, inferred from the dominant axis in Basil's path of motion. The system provides multiple-example and choice strategies for modifying this inference. In the alignment task, the second example given by the user (step e) does not match the

predicted path, hence it is generalized to "any direction." If the user had instead disagreed with the inference that Basil should move downward, he or she could have clicked on his shell to rotate him (for instance, to upward).

Constraints

Basil applies a set of rules to rate the significance of touches observed after each action; the top ranked are selected as constraints. The significance of a touch depends on two factors: the extent to which it was – or could have been – affected by the action and the degrees of freedom that remain when positioning Basil after achieving the touch with the given graphical operator. The most significant touches are those caused by the action and leaving the fewest degrees of freedom. For instance, when the user drags the box to the guideline (step c), Basil notes that the box's lower right corner touches the line. This touch is caused by the move and is therefore highly significant in that respect. It involves two objects already known to Basil, so there is no freedom in the choice of object. It involves a vertex and a line segment, so there is one degree of freedom in the actual position. Since no other touches are observed, Basil selects this one as a constraint and marks it with a black "tack" (white tacks are used to indicate irrelevant touches that would not be enforced when predicting). If the user disagrees with Basil's inference, he or she can click on the tack to toggle it.

When the user dragged the box, he or she kept it at nearly the same vertical coordinate; Basil infers that this value should be held constant when selecting the destination point on the guideline. Two alternative inferences are that the destination should be the nearest point or a point selected by the user. The current implementation does not let the user give explicit instructions about selecting a particular solution, but the user could construct a horizontal or nearest point (perpendicular) constraint.

Flow control

Predictions are made by consulting a set of production rules of the form

$$\textit{if context} = [\textit{action}]^* \rightarrow \textit{prediction} = \textit{action}.$$

The difference between Metamouse's context model and those used in the Reactive Keyboard and the Autoprogramming Calculator is that its rules have variable context lengths and are tested in length order. Thus, each context is as general (i.e. short) as possible, yet the model as a whole is deterministic (i.e. no two rules fire in the same context). This approach is intended to maximize the number of predictions made yet distinguish states as determined by events arbitrarily far back in history.

Rules are checked in the order specific to general, that is, from longest to shortest context. A rule fires if it meets two conditions: (a) its context of k previous consecutive actions matches the k most recent actions in history; (b) its prediction is performable, in the sense that its constraints can be satisfied. After a rule has fired, the predicted action is appended to history, and the rule set is consulted again. If no rule fires, the user performs the next action, N, which Basil analyzes and matches with generalized steps already stored. If no match is found, Basil stores N as a new type of action. Whether matched or not, a new rule that predicts N is created. If N was performed because some rule R predicted a nonperformable action, then the new rule is *[R-fails]* $\rightarrow N$. This happened, for instance, at (h) in Figure 6.8. Otherwise, the new rule is initialized as *[M]* $\rightarrow N$, where M is the immediately preceding action in history. If N was performed because the user had rejected the prediction made by rule R, the system tries to specialize R by extending its context backwards through history, such that it would not have fired on this occasion. If no specialization covers all previous successful firings of R, then the new rule for N is specialized instead, such that it covers no previous instance of R.

Experience

We did a small, quasi-controlled user study to learn whether people could put Basil to effective use in practice and to identify shortcomings in its design and implementation (Maulsby, Witten, Kittlitz, & Franceschin, 1992). We tested the system on three computer science students and four nonprogrammers. All subjects were given the same single page instruction sheet and performed the same six tasks several times, with and without Basil's assistance. Four of the tasks required graphical constructions to express alignment or distance, and one (sorting boxes by height) required either an explicit setting of Basil's direction or the use of a movable construction (such as a sweeping line) to select boxes in order.

All subjects, programmers and nonprogrammers alike, benefited from Basil's predictive ability in the two tasks where no constructions were required. The three programmers and three of the nonprogrammers successfully used simple constructions to teach Basil two different ways of aligning boxes. The one user who did not benefit from prediction in these cases did not use constructions during any of the tasks. The study showed that when users employ appropriate constructions, Basil is able to help them by making appropriate predictions. The users, especially the nonprogrammers, were able to explain most of Basil's behavior to themselves (including bad predictions or failures to predict) after just a few minutes' experience.

Of greater research interest, however, are the problems users encountered. The following paragraphs outline the major difficulties and their causes.

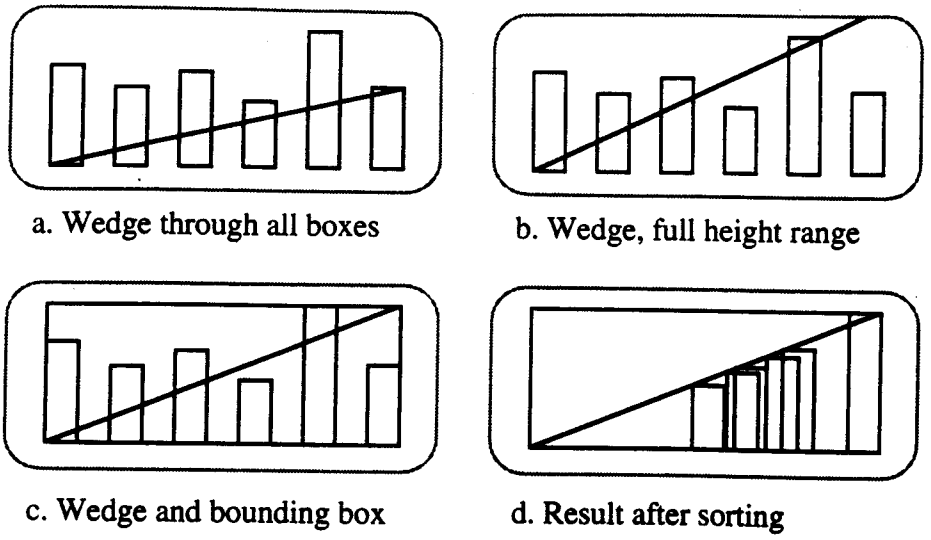


Figure 6.9. Tools attempted in sorting by height.

Constructions

We found that users readily invented "static," declarative constructions for spatial relations but had no insight into using "dynamic," procedural ones like a sweeping line. Figure 6.9 illustrates one nonprogrammer's experiments with a static tool for ranking boxes by height. His tool is based on a "wedge," an oblique line that maps height order onto horizontal order. The first attempt (Figure 6.9a) is a mistake, because the line does not map the entire range of heights. The second version (Figure 6.9b) would work but for the inadequacy of Basil's selector functions (see below). Blaming himself for the failure, the user adds a bounding box (Figure 6.9c), which he believes will show Basil that all the rectangles are to be processed, regardless of whether they touch the wedge initially, and also ensures that boxes are moved strictly horizontally, since they must touch the bottom of the bounding box. Figure 6.9d shows the result he obtained (without predictions from Basil).

Touch metaphor

Users immediately understood Basil's bias towards touch and focused their attention on that characteristic. They rightly did not expect Basil to infer intrinsic properties like object height. All were puzzled when asked to teach him how to sort by height, because they believed they were supposed to find some way of

expressing height in terms of touch. Several said they would prefer a more direct way of instructing Basil with commands.

Selectors

Users who did not initially decompose tasks in a way compatible with Basil's object selector functions did not learn to do so. In particular, if a task could be done either as a single loop or multiple loops with simpler bodies, most users chose the latter; but Basil cannot predict multiple loops over the same objects. Only one user, a nonprogrammer, tried using a single loop after observing that Basil predicted only the first of multiple loops.

The limited choice of selector functions also prevented some constructions from working as users expected. For instance, the wedge tool in Figure 6.9 initially touches several of the boxes, generating a distinct variable for each one; Basil cannot form a loop when observing the user select these, because the variables cannot be coalesced.

Further work

Based on this user study, we conclude that Metamouse needs two additional selector functions and the ability to alter the choice of selector based on multiple examples. One selector iterates over a set of objects that touch a given object; the other iterates over the binding history of a variable. The multiple examples strategy involves comparing rejected or nonperformable predictions with the user's action: If substituting other selector functions for those in the prediction results in a match with the user's action, that substitution is made (if it covers previous instances of the predicted step), or a new rule with that substitution is stored (if not).

The rules for inferring constraints and directions worked well enough (with respect to the drawing tasks studied here) so that we have no data regarding the use of direct instruction (such as clicking on tacks or rotating Basil). The production rule learning algorithm proved successful in distinguishing special contexts such as the first and last iterations of a loop and in suppressing erroneous steps (by lengthening their context such that they are never predicted again). Further testing of these aspects of the system will be warranted once the improvements mentioned above have been made.

In conclusion, Metamouse is a predictive interface that uses background knowledge to abstract features for pattern matching; we have found that this improves the quality of Basil's predictions. The system's metaphor, based on teaching a touch-sensitive turtle and using graphical objects to express spatial relations, is (on the whole) easily grasped by users. Its notion of sets of objects is computationally inadequate to model the way people really do repetitive editing.

The combined use of generalization from examples and direct user instruction has yet to be tested in user studies.

What will they think of next?

The five interfaces described in this chapter and summarized in Table 6.1 illustrate how predictive interfaces can help to reduce the input required not only from physically disabled people, for whom data entry may be mechanically taxing, but from able-bodied users as well. Although the systems have a common theme, they are quite different from each other in three important respects:

- their reliance on application-related background knowledge and syntactic analysis;
- their ability to generalize past input for reuse in predictions; and
- their implementation of mixed-initiative interfaces where the system and user share control over the formation and use of the underlying predictive model.

The Autoprogramming Calculator distinguishes numbers (which can be variables) from operators (which cannot). Workbench distinguishes the first token of a command line from the remainder. The Reactive Keyboard and Adaptive Menus are completely general, have no a priori knowledge, and perform no syntactic analysis.

The ability to generalize history depends on syntax analysis and is enhanced by background knowledge. The Autoprogramming Calculator selectively predicts subsequences of a calculation interleaved with user inputs. Workbench, the Reactive Keyboard, and Adaptive Menus do no generalization.

Most of our predictive systems have separate interfaces which provide the user with a means of controlling, accepting, and rejecting predictions. Adaptive Menus is the exception. The Reactive Keyboard lets the user browse among potential predictions and has a statistically optimized presentation scheme. The user can easily augment or modify the model and its predictions by choosing different text files to prime it. Similarly, Workbench users can save (and edit) their interaction records in separate files and use any one to prime predictions. Workbench provides an array of mechanisms for organizing history, including storage metaphors under which the user can build his or her own task model. The Autoprogramming Calculator's only extra interface option is the "undo" command. The Reactive Keyboard requires that the user explicitly accept a prediction or implicitly reject it by performing an alternative action.

We have learnt several lessons from the design and implementation of these systems and from studying how people use them. First, powerful predictors can be built that can be applied to a variety of task domains. The best example is the Reactive Keyboard. Because its predictive power is based on principles of adaptive text compression, it can be applied to any text domain. We have used it

for composing free text, for writing programs, for supplementing the Unix and MS-DOS command based interfaces, and – through the Autoprogramming Calculator – for formulating mathematical expressions. Furthermore, any advances in text compression algorithms can be applied directly to increase the accuracy of the Reactive Keyboard's predictions. The algorithm behind Adaptive Menus can configure any ordered name space. Workbench is capable of becoming a front end to any command based interface, regardless of whether commands are typed or selected through menus.

Second, predictive systems work well in practice and, if well designed, are easy to use. The Reactive Keyboard has proven its usefulness to disabled people – its users find it indispensable. Users of Workbench appreciate the support it provides. The controlled experiment on Adaptive Menus indicates that user performance is enhanced by the assistance that it gives. We believe that these systems have illustrated the viability of predictive interfaces.

Third, predictive systems can be implemented today using existing technology. All the systems described here have been built on conventional computers, and all perform well in real time.

Finally, whether or not users accept a predictive system will depend on their individual needs. Because of the cognitive/physical trade-off, using the predictions may be more work for some people than just continuing without them. Of course, the greater the need, the greater the motivation to learn and use these systems. A predictive interface that is a curiosity to the able-bodied may be a beneficial – indeed, indispensable – tool for a person who is physically disabled.

The adaptive predictive techniques we have described could easily be made widely available to disabled people. The mechanisms of Adaptive Menus and the Autoprogramming Calculator are suitable for integration into existing applications. Moreover, it is feasible to use predictive techniques such as the Reactive Keyboard and Workbench in conjunction with current systems without the necessity to integrate them into particular applications.

What will they think of next? We would like to see a series of modules that allow users to access these techniques, both integrated within individual application programs and in the form of add-on "system extensions" (as in Macintosh System 7). For example, the Reactive Keyboard could be integrated into the command shell in systems like Unix or MS-DOS as a switchable option for all users. A Macintosh desk accessory could combine the adaptive predictive technique of the Reactive Keyboard with a conventional augmentative communication aid for entering single letters – this would be usable with any Macintosh application. We see no reason why pocket calculators and spreadsheet programs should not routinely embody predictive techniques. A personalized phonebook system, with autodial capability and local telephone directories available on CD-rom, could use Adaptive Menus as its basic interface. Perhaps more futuristically, a full implementation of Workbench could provide users with easy opportunities to create individually crafted tool sets for any application.

Acknowledgments

This research has been supported by the Natural Sciences and Engineering Council of Canada, Esso Resources Limited, Apple Corporation, and the Alberta Heritage Foundation for Medical Research. Amongst many people who have helped us with various aspects of this work, we would like to acknowledge in particular the enthusiastic support of David Hill.

References

Bell, T., Cleary, J. G., and Witten, I. H. (1990). *Text compression.* Advanced Reference Series in Computer Science. Englewood Cliffs, NJ: Prentice-Hall.

Bier, E. A. and Stone, M. C. (1986). Snap-dragging. *Computer Graphics 20*(4): 233–240.

Cleary, J. G., and Witten, I. H. (1984). Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications, 32*(4): 396–402.

Cover, T. M., and King, R. C. (1978). A convergent gambling estimate of the entropy of English. *IEEE Transactions on Information Theory, 24*(4): 413–421.

Cypher, A. (1991). EAGER: Programming repetitive tasks by example. In *Reaching through technology: Proceedings of ACM CHI' 91 Conference on Human Factors in Computing Systems*, ed. S. P. Robertson, G. M. Olson, and J. S. Olson. (pp. 33–40). New York: ACM Press.

Darragh, J. J., and Witten, I. H. (1991). Adaptive predictive text generation and the reactive keyboard. *Interacting with Computers*, *3*(1): 27–50.

Darragh, J. J., and Witten, I. H. (1992). *The reactive keyboard*. Cambridge Series on Human–Computer Interaction. Cambridge: Cambridge University Press.

Darragh, J. J., Witten, I. H., and James, M. (1990). The reactive keyboard: A predictive typing aid. *IEEE Computer, 23*(11): 41–49.

deJong, G., and Mooney, R. (1986). Explanation-based learning: An alternative view. *Machine Learning, 1*: 145–176.

Gibler, C. D. (1981). *Linguistic and human performance considerations in the design of an anticipatory communication aid.* Unpublished PhD thesis, Northwestern University.

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

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

Greenberg, S., and Witten, I. H. (1985). Adaptive personalized interfaces – A question of viability. *Behavior and Information Technology*, *4*(1): 31–45.

Greenberg, S., and Witten, I. H. (1988a). Directing the user interface: How people use command-based systems. In *Proceedings of the IFAC 3rd Man–Machine Systems Conference*, Oulou, Finland.

Greenberg, S., and Witten, I. H. (1988b). How users repeat their actions on computers: Principles for design of history mechanisms. In *Human factors in computing systems,* Proceedings of the ACM CHI'88 Conference on Human Factors in Computing Systems, ed. E. Soloway, D. Frye, and S. B. Sheppard (pp. 171–178). New York: ACM Press.

Kucera, H., and Francis, W. N. (1967). *Computational analysis of present-day American English.* Providence RI: Brown University Press.

Maulsby, M., Witten, I., and Kittlitz, K. (1989). Metamouse: Specifying graphical procedures by example. *Computer Graphics, 23*(3):127–136.

Maulsby, D. L., Witten, I. H., Kittlitz, K. A., and Franceschin, V. G. (1992). Inferring graphical procedures: The compleat Metamouse. *Human Computer Interaction, 7*(1): 47–90

Myers, B. A. (1986). Visual programming. programming by example, and program visualization: a taxonomy. *In Human Factors in Computer Systems: Proceedings of the ACM conference Chi'86,* ed. M. Mantei and P. Orbeton (pp. 59–66).

Myers, B. A. (1988). *Creating user interfaces by demonstration.* San Diego, CA: Academic Press.

Pickering, J. A., and Stevens, G. C. (1984). The physically handicapped and work-related computing: Towards interface intelligence. In *Proceedings of the Second International Conference on Rehabilitation Engineering* (pp. 126–127). Ottawa, Ontario, Canada.

Shannon, C. E. (1951). Prediction and the entropy of printed English. *Bell System Technical Journal, 30*(1): 50–64.

Stallman, R. (1987). *GNU Emacs manual.* Cambridge MA: Free Software Foundation.

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

Suen, C. Y. (1979). *N*-gram statistics for natural language understanding and text processing. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *1*(2): 164–172.

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

van Sommers, P. (1984). *Drawing and cognition.* Cambridge: Cambridge University Press.

Witten, I. H. (1981). Programming by example for the casual user: A case study. In *Proceedings of the Canadian Man–Computer Communication Conference* (pp. 105–113). Waterloo, Ontario.

Witten, I. H., Cleary, J., and Greenberg, S. (1984). On frequency-based menu-splitting algorithms. *International Journal of Man–Machine Studies*, *21*(2): 135–148.

Witten, I. H., and Greenberg, S. (1985). User interfaces for office systems. In *Oxford Surveys in Information Technology*, ed. P. Zorkoczy (Vol. 2, pp. 69–104). Oxford: Oxford University Press.

Witten, I. H., Greenberg, S., and Cleary, J. (1983). Personalizable directories: A case study in automatic user modelling. In *Proceedings of Graphics Interface '83* (pp. 183–190). Canadian Man-Computer Communications Society.

Witten, I. H., MacDonald, B. A., Maulsby, D. L., and Heise, R. (1992). Programming by example: The human face of AI. *AI and Society* 6:166–185.

Witten, I. H., and Mo, D. H. (1993). Tels. In *Watch what I do: Programming by demonstration*, ed. A. Cypher. Cambridge, MA: MIT Press.