Oxford Surveys in Information Technology, Vol 2, 69–104


# User interfaces for office systems

IAN H. WITTEN and
SAUL GREENBERG

Man–Machine Systems Laboratory
Department of Computer Science
The University of Calgary
2500 University Drive NW
Calgary, Canada T2N 1N4

## Contents



## Abstract

This paper surveys recent developments in the 'top-level' interface for interacting with office information systems. This is the level at which users initially make contact with the system, and from which they invoke subsystems for specific tasks such as text manipulation, mail, database access, and so on. Although the style of the top-level interface need not necessarily dictate that of the subsystems, it is generally agreed that they should share a similar nature in order to achieve the effect of an 'integrated' system. Hence the top-level interface design has considerable influence in determining the character of subsystems.

A number of top-level interfaces are described, providing a survey of different interaction styles. Numerous references are made to published accounts of commercial and research systems, with capsule descriptions of typical examples. Illustrations of their use are included. As these man–machine systems must match good design and user compatibility, an introduction of interface design principles is included, although no attempt is made to survey completely the myriad of published guidelines and related human factors research.

## Introduction

One of the biggest problems with office information systems is not the individual subsystems but rather the 'glue' which holds them all

together. Creating subsystems which allow people to do things like construct documents, send mail, use databases, and so on, may not be easy but at least it is something that you can get to grips with. But how do you invoke these subsystems? What does the top-level command interface look like? What sort of general image does the system present to the user? Do you type commands and arguments, select from menus, create windows for different task contexts, fill out forms, use tools and machines, or shuffle paper on a simulated desktop? What does the user have to know about computers to use the system? What about, for example, the concept of *file*: multiple versions, system backups, local storage media like floppy disks, and so on? How does the user discern between conceptually different file types, such as executable application packages, document text and graphical pictures? And how about consistency between applications? Can the same actions produce the same response in different subsystems, and if so, how?

These are very general questions. But they highlight real design choices which will have an enormous effect on the user's perception of the system. This article surveys recent work and systems which illustrate the spectrum of possibilities. Of course, there is no 'best' command interface; many trade-offs must be considered when designing an actual system. We do not attempt to come to general recommendations, for each technique serves its purpose depending on both the task and the audience. However, it is as well to have some knowledge of the possibilities, if only to spark the imagination when designing or comparing systems.

In order to set the scene for descriptions of user interface styles, the paper begins by discussing the notion of 'quality' of a user interface. This can be assessed through design guidelines, models of human performance, or an analysis of compatibility with the individual user. The notion of 'dialogue determination' seems to capture best the differing points of view of a wide user community and forms a common thread throughout the remainder of the paper. The subsequent section presents several different styles of the top-level interface: conventional command-driven interfaces, menus, windows, forms, and natural language. Although some of the illustrations are not specific to office systems, all are applicable in that context. The final section investigates paradigms for integrating system components. The major paradigm, 'direct manipulation', promotes an illusion that the user is manipulating directly some real-world structure or object; the popular examples of desktop simulation and soft machines are described and current implementations reviewed.

## The quality of the interface

What defines a good user interface to an office information system? Before considering this question it is worth getting to 'know the user' (Hansen 1971). Typical office workers have been described as *casual users*, in the sense that they use the office information system only

occasionally, spending most of the day doing something different (Martin 1973). However, as office information systems become more pervasive, the user is likely to spend increasing amounts of his working time interacting with them. Although we retain the term 'casual user', we do not wish to imply that he only makes naïve use of the system—on the contrary, he may have quite a sophisticated working environment and extensive experience with it. What distinguishes him from a professional computer programmer is that the office worker has little or no computer or mathematical background (Zloof 1977).

Consequently we accept Cuff's (1980) extensive portrait of the casual user, which does not rely on frequency of use but rather identifies typical characteristics such as

- poor retention of detail
- propensity for error
- intolerance of formality
- poor handling of 'normal' database constructs.

Even from this brief sketch, it is clear that any interface which requires the office worker to understand computer paradigms and to have strong programming skills is not a good one.

### Interface design guidelines

It is well beyond the scope of this paper to present a complete survey of the myriad of published interface design guidelines. Here we provide some pointers into the literature and convey the flavor of what is known.

A good set of design guidelines can be found in Gaines and Shaw (1983). Shneiderman (1980) provides a comprehensive review of design goals by other authors, while Maguire (1982) summarizes suggestions in the literature and identifies areas of conflict. It is important to realize that almost all published guidelines are based on experience and intuition, rather than on quantitative empirical investigation. Although this gives them something of the status of folklore rather than science, they do have their place for, according to Shneiderman (1979)

> ... it is not possible to offer an algorithm for optimal or even satisfactory design. Interactive designers ... seek a workable compromise between conflicting design goals. Systems should be simple but powerful, easy to learn but appealing to experienced users, and facilitate error handling but allow freedom of expression.

Note in passing that many commercial computer systems—even very successful ones—flagrantly violate user interface guidelines. (For an example, see 'The trouble about Unix', Norman 1981). This is not because the guidelines are wrong but because the marketplace is as yet fairly unsophisticated.

A typical example of a well-respected design principle is the golden

rule *know the user*, first articulated by Hansen (1971). This includes *using the user's model* of the activity being undertaken as a basis for the dialogue (Gaines and Facey 1975; Gaines and Shaw 1983). The goal is to provide the user with a comfortable framework in which to work, with familiar jargon and concepts. Eason and Damodaran (1979) consider it crucial for users to be heavily involved in developing the system if it is to meet their real needs, although the design should be oriented towards the *typical* user (who may differ from those who choose to contribute towards the development process). Personalized self-adaptive interfaces, which adapt themselves to the characteristics of each individual user, offer an attractive alternative to the labor-intensive iterative design and tuning process (Rich 1983; Innocent 1982; Taylor 1981; Greenberg 1984). In this case it is the system which 'knows' the user: the designer is responsible for ensuring that it incorporates sufficient mechanism for getting to know him.

Pertinent concerns voiced extensively in the literature which guidelines seek to address are, for example,

- how the casual user learns the system
- on-line introduction to the system and continual training in its use
- on-line help facilities
- situations where the task is beyond the expertise of the user (or the application system)
- robustness of the interface, and treatment of errors by users
- the tone of error messages
- feedback on how the dialogue is proceeding, at both the command level and the application level
- consistency of feedback with the user's internal model of the system.

Individual references are not given in the above list, for most authors in this area cover much common ground in their design guidelines. Shneiderman (1980) and Maguire (1982) provide excellent reviews.

There is a significant and rapidly growing body of human factors research which is much more specific than the formulation of general interface guidelines. This aims to construct

> ... a model of user-computer interaction that can predict the performance of both new and skilled users of various interaction techniques. While this goal may never be completely achievable in practice, it can nevertheless act as the motivator for research.
>
> (Foley *et al.* 1984)

A good example of this kind of work is the 'keystroke-level model' of keyboard communication with interactive computer systems (Card *et al.* 1980). A recent paper by Foley *et al.* (1984) combines a taxonomy of interaction tasks and techniques with a very useful survey of known experimental findings results pertaining to them. However, this addresses issues of human–computer interaction at a much finer

granularity than the present paper, which is concerned with a broad-brush characterization of different styles of interaction.

### The user's point of view

A user is not of course concerned with interface design guidelines, nor with the human factors of interface design, but reacts in an individual way to the interface as it appears to him. Because the 'feel' of an interface is highly subjective, it is hard to uncover meaningful dimensions along which to measure user response. The principal dimension used throughout this paper is that of *dialogue determination*: the extent to which a user controls the system *vis-à-vis* the extent to which it appears to control him (Thimbleby 1980).

Dialogue determination describes the subjective quality of the user interface by considering the user's feeling of control of the dialogue as a fundamental measure (Thimbleby 1980). It is composed of many aspects, some being a direct consequence of the guidelines suggested above, such as:

- *flexibility* in allowing users to determine their own level of use,
- *commonality* between the interface and a user's previous experience,
- *immediacy* of a response relative to the task on hand,
- *intromission*, which allows parallel events to occur within a dialogue,
- *variability* of language within a task—specific modes may constrain the user's choice of actions,
- *feedback* of the dialogue state.

Thimbleby suggests that a system is overdetermined if it restricts and unnecessarily controls the user, and underdetermined if the user is left at a loss as to what to do or how to do it. For example, a terse dialogue may be underdetermined for a novice needing considerable direction, while a verbose, computer-directed dialogue is probably overdetermined for the trained expert. A dialogue is considered ill-determined when it has attributes of both, but well-determined when it matches the user's expectations, pace and ability. But one cannot label a determination level for a dialogue without considering the experience and expectations of the user.

As an analogy, consider the family car and the sports car. Although each offers the same basic function of transportation, individuals may prefer driving one over the other. The family car driver choses a car that 'drives itself', and considers the sports car as under-determined; the highly-visible instrumentation, manual transmission and choke, and steering sensitivity only serve to confuse. But the sports car enthusiast finds the family car overdetermined; the lack of control and general insensitivity is viewed with disdain to one who considers responsiveness important. Yet both cars are ergonomically well-designed, it is the driver who selects a car type as appropriate, a

point well understood by an automotive industry which markets radically different types of vehicles.

Just as an automobile manufacturer cannot build a single model of car to satisfy all customers, neither should we expect to find a single 'best' interactive interface. The next section will examine several examples of command interfaces and the trade-off of power versus simplicity offered by each. Judge them cautiously, for it is too easy to label any one system as ideal or unsatisfactory. Remember that it is the *user's* highly subjective opinion that counts. Each dialogue technique should be contrasted by approximating the probable dialogue determination level with the skills, knowledge, needs, and background of our casual user—the office worker.

## Examples of command interfaces

Several quite different interface designs have been introduced recently which give examples of top-level system interfaces and the degree of control they offer the user. Be warned, however, that they are by no means mutually exclusive; neither do they all address exactly the same problem or user population nor provide the same facilities. The approaches represent different tools in a kit—several may be used together to do a job. Each one represents a compromise between power and simplicity.

We begin with conventional *command-driven* interfaces, a dialogue normally used by 'experts' interacting with general-purpose computer systems. This is followed by *menu-based* interfaces, which are usually considered more suitable for inexperienced or casual users. Subsequent examples are somewhat more exotic and perhaps more specific to the office domain. *Windows* provide a tool which is suitable for handling the usual multiple-context operations of an office worker, while *forms* take advantage of already familiar office models. *Natural language* has the potential to capitalize on a user's existing dialogue skills.

### Conventional command-driven interactive computer interface

The conventional interface to interactive systems employs commands issued by the user. The system is a passive slave awaiting orders; no attempt is made to guide or help the user explicitly. Once received, it carries out the order and then awaits the next command. We are all very familiar with this kind of interface. Originally designed for teletype terminals, it hasn't changed much since. We will use Unix as an example since it is a fairly sophisticated system within the genre (Ritchie and Thompson 1974).

Despite the existence of some screen-based programs (typically editors), the basic command interface, called 'shell' in Unix, is teletype-like. No use is made of the cursor control features provided on most VDUs. With the single exception of the character-erase and line-erase characters, the screen is treated as a long roll of paper.

Figure 1 represents a typical command screen after log-in. In this sequential dialogue, the user asks for the current date, requests a listing of files, and checks for mail. Further commands would scroll this information off the screen irretrievably.

This interface has particular problems in its handling of full-duplex interfaces and multiple asynchronous processes. For example, if the user types ahead, it would make sense to suppress any prompts for input he has already typed. This is hardly ever done (it raises difficult questions of division of responsibility between applications programs and the operating system). Moreover, it is hard to know how to handle asynchronous output. Unix *write* (*1*) may interrupt the user while he is part way through typing a line; whereas the *csh* Unix command interpreter (Joy 1979) waits for the user to type *return* before informing him that background processes have terminated normally. Neither of these seem particularly satisfactory. If the user had typed the *ls* command in Figure 1 at the same time as a mail notification message was sent, the output of both would be jumbled together as a consequence. More sophisticated interfaces partition a screen so that user commands, normal output, and asynchronous messages can be placed in separate areas (see 'Window systems'). However, this reduces the available screen space for each.

There have been some, but not many, improvements in command interpreters over the years. For example, one breakthrough in the design of operating system interfaces was to eliminate all differences between invoking a system program and a user program. This seems so obvious and natural now that it usually goes unnoticed. It is significant because it allows you to tailor a system to individual needs simply by writing utility programs and putting them in the right place, without having to alter the innards of the system in any way. Earlier operating systems had separate commands for running user

```
Unix: date
Sun Jul 22 20:40:01 MDT 1984
Unix: ls
Figures                abstract                introduction
main-body
Unix: mail
No mail
Unix:
```

Fig. 1. A Unix command screen, showing user-typed commands in italics

and system programs—after all, they were kept on separate areas of disk! However, this flexibility has drawbacks. It encourages users to build and share extensive libraries of commands, causing difficulty with the naming of different programs and multiple versions of programs. Others may have come to rely on programs in a personal library without the owner's knowledge, in the erroneous belief that they were 'standard' utilities.

In spite of these problems, Unix is extremely popular with computer experts, precisely because its terseness and power offer them a well-determined dialogue. Norman (1981) considers Unix to be a disaster for the casual user, for the same reasons. Is the command-driven interface a suitable one for the office worker? Its richness of commands and terse dialogue is probably underdetermined for the very occasional user, especially if his needs are simple. But it may be well-determined for the trained office person who must do many diverse and unintegrated tasks quickly and efficiently. Table 1 summarizes the advantages and disadvantages of the conventional 'glass teletype' interactive command interface.

### Menu systems

Menus attempt to solve the dour and unforthcoming nature of command-driven interfaces by explicitly revealing all possible options to the user, by analogy to a restaurant which presents the diner with a list of choices. Command-driven interfaces, in contrast, are like restaurants with no menus; the casual customer can order standard dishes, while exotic offerings are known only to local 'experts'. The menu paradigm is an old one in graphical interaction, and has served well in a variety of interfaces to drawing, painting, and drafting systems. It has also been used as the interface to many other interactive systems (such as database retrieval systems).

Menus exemplify the oft-cited 'what you see is what you get' touchstone. Smith (1975) regards a display as a working area for our short-term memory, to be operated on by conscious thought. A well-designed menu system fits this conception, for it shows all currently possible choices, and thereby constrains the user. There is no need to 'remember' across displays—only the presented items can be chosen.

There are many different ways of arranging menus. Two of the most popular are *fixed* menus, which present all possible choices on a single display, and *taxonomic* menus, which classify the domain hierarchically and allow the user to navigate through it.

The automated teller in Figure 2 illustrates a fixed menu. The one-line message area provides concise directions and feedback to the user. Only sensible menu choices are enabled. For example, when the machine requests an account type (the black keys in the Figure) it ignores all keys not relevant to that context. The success of this particular menu is evident from the large number of automated tellers appearing throughout the western world. With it, the user can inter-

Table 1. Advantages and disadvantages of command-driven interfaces

| Advantages |
|---|
| • simple<br>• terse<br>• does not constrain applications programs in any way<br>• easy to add commands which appear like new system commands |
| **Disadvantages** |
| • minimal feedback of the system state<br>• encourages a jumble of applications programs with different interfaces<br>• cannot deal elegantly with asynchronous events<br>• impossible to refer to information displayed on previous screens, even though it may be related to the current context<br>• is overly secretive (underdetermined) for most naïve and casual users and to 'experts' stepping out of the bounds of their knowledge |

act easily with a complex database, although his interactions are highly restricted. The dialogue is well-determined for almost anyone, for 'experts' can ignore the message area and rapidly enter their transactions. The limitations of the automated teller are clear: only the choices on the fixed menu are possible. But for clients it remains well-determined, for they expect access to a human intermediary (such as the bank teller) for non-routine tasks.

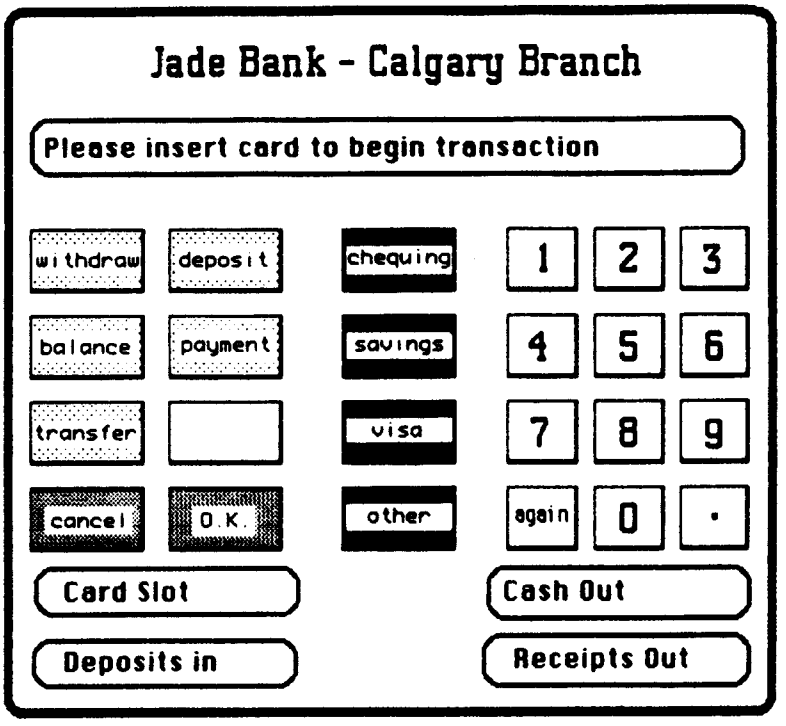


Fig. 2. The Automated Teller.

Most information systems are too complex for a single-page display. Videotex systems are intended to allow casual users access to a large volume of information. A taxonomic menu is used, and menu items are selected on a restricted keyboard. The database may be extensive, possibly including up to 1 million pages of text and graphics (Wilkinson 1980). The user navigates through the hierarchy, attempting to focus in on the desired information by refining the category which is currently displayed. Figure 3 illustrates a user navigating the category of entertainment in a quest for current plays.

A command interface to an office or operating system can be built using the same kind of taxonomic structure. A recent project at Bell Laboratories, called *Menunix*, shows how an extensive and flexible operating system interface can be implemented with menus (Perlman 1984). It allows access to the Unix system by displaying two menus from which users can make selections: the *file menu* which lists the current working directory, and the *program menu* which lists the programs currently available. Figure 4 shows an example display. (The boxes around each *Menunix* component are for illustrative purposes only; they do not actually appear on the display.) One conse-

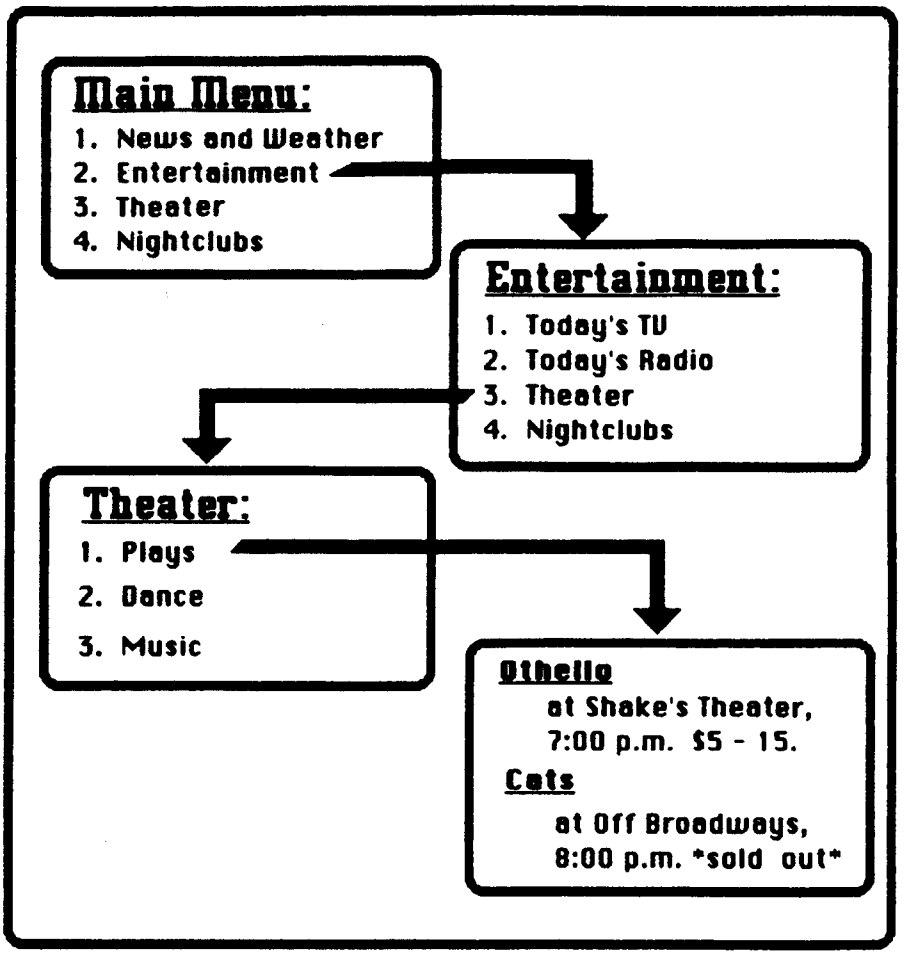


Fig. 3. Retrieving information through a hierarchical menu.

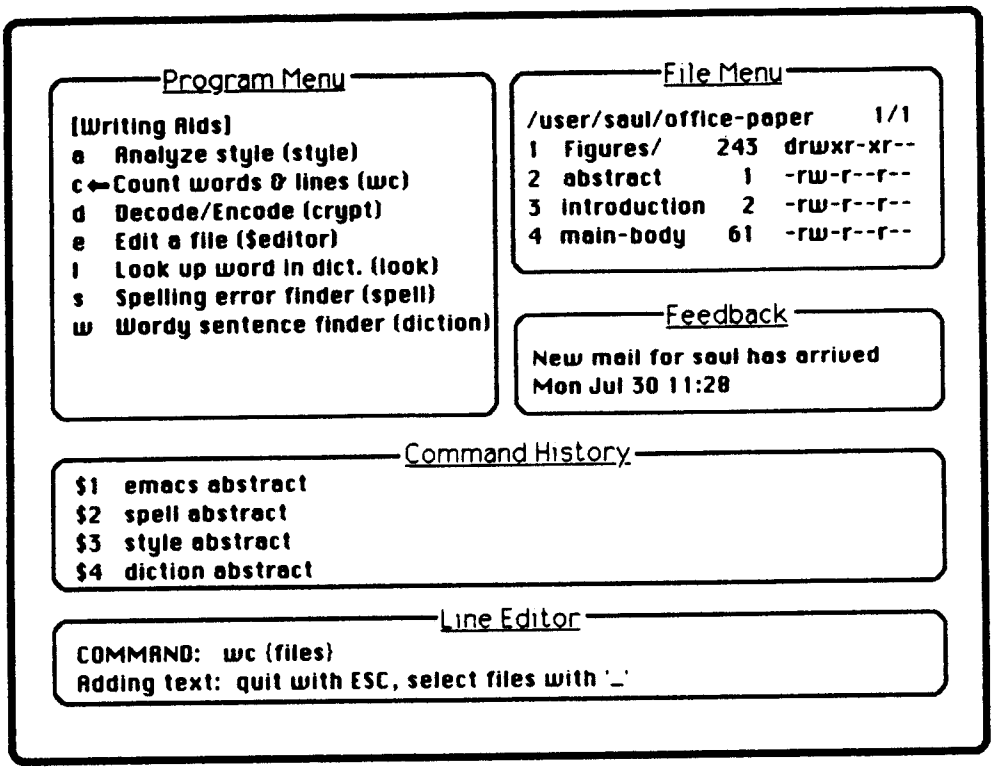


Fig. 4. A Menunix workbench.

quence of menu access to Unix programs is that the vast selection of utilities must be structured somehow into reasonably small subsets; otherwise the menu would become unmanageable. These 'workbenches' provide a useful way of viewing the activity of working with such a system.

*Menunix* uses a single keystroke to select an item from either of these menus. The file menu displays at most 9 filenames, accessed with keys 1–9 (you can leaf through larger directories using the '+' and '−' keys). When a file menu entry is selected, *Menunix* tries to do something sensible with the selected file. If it is a directory file, the current working directory is changed. If it is an executable file, it is run (after arguments are requested). If it is a text file, the user's preferred editor is called on it. Thus users are able to edit files and change directories with just the file menu commands.

Programs are structured into workbenches, and the program menu displays brief (half-line) descriptions of the programs in the current workbench (Figure 4). The descriptions are identified by their initial letter (when a workbench is constructed none of its descriptors may begin with the same letter). Workbenches are organized hierarchically. For example, there is a programming workbench that contains sub-workbenches for general programming and specific programming languages. Figure 4 illustrates one workbench which gathers together writing tools; other workbenches may deal with mail, specific applications, and so on. When a program menu entry is selected, arguments are requested and the program is executed. In

order to implement the hierarchy, an entry in a workbench may point to another workbench (in the same way that an entry in a directory may point to another directory in the file hierarchy). Selecting one of these entries will replace the current program menu accordingly. Unfortunately, not all programs fall neatly into the workbench paradigm; some tools may not be in the location in which the user expects them. For example, the *crypt* program in Figure 4 may be a useful tool for one author who requires secure text files, while to another it unnecessarily clutters his workspace.

In some circumstances it is not necessary for a menu to remain visible on the display screen for long. One important way of displaying a menu is by 'popping it up' on the screen for just as long as is needed. Frequently employed in window systems, a *pop-up menu* (or the very similar *pull-down* menu) is painted on the screen near the cursor or other focus of attention. Typically a mouse button is depressed to call the pop-up menu, and when the button is released the menu disappears and the hole left by the menu is repaired. Menu selection is achieved by pointing at the desired item with the mouse, and indicated visually by shading that menu item. (A pull-down menu is illustrated in the screen in Figure 5 in the next section.) The pop-up menu is a convenient way to keep frequently used commands close by without occupying space on the screen. Different menu contents can be generated depending on the state of the user's interaction, in such a way that it is almost impossible for him to make a syntactic error in the interaction. In addition, pop-up menus do not normally consume screen space. To their disadvantage, their implementation is complex because the damaged area must be saved and repainted.

Menus are a useful medium for users, particularly naïve and casual ones, to communicate commands to computers (Cuff 1980). An assortment of other examples of menu-based systems can be found in Martin (1973). General advantages and disadvantages of menu-based interaction are detailed in Table 2.

The popularity of hierarchical taxonomic menus and the market potential of videotex has stimulated research into their suitability for naïve or casual users. Several serious drawbacks to such schemes have been identified:

- Users may be uncertain about the content of a menu category (Latremouille and Lee 1981) and will have to probe the hierarchy repeatedly in order to retrieve the desired information.
- Retrieval will fail if the sought-after information does not exist in the database.
- Even if the information does exist, retrieval may fail because the user is unable to select the proper categories.
- As databases grow in size, it becomes harder to locate specific items (Martin 1980). This is particularly disconcerting when one considers that Prestel, a British videotex system, is aiming

Table 2. Advantages and disadvantages of menus

| Advantages |
|---|
| • requires only a standard VDU<br>• a restricted keyboard or positional pointer can be used<br>• actions allowable in the current context need not be remembered since they are continually available for review<br>• no typing ability is required<br>• requires little user training |
| **Disadvantages** |
| • extremely inflexible<br>• only a small number of items can be displayed on a menu<br>• experienced users may undergo considerable frustration<br>• discourages typing ahead<br>• users have problems navigating large hierarchies |

for one million pages of information by the mid-1980s (Wilkinson 1980).

- User confusion and spatial disorientation in a large menu hierarchy is likely to occur as a result of the problems just noted (Engel *et al.* 1983; Mantei 1982).
- Difficulty in remembering the choice path used to arrive at the current location causes erratic searches on successive probes (Engel *et al.* 1983).

Dumais and Landauer (1982) note that most schemes provide only a single access route to a given item, and suggest that some of the above faults probably arise from this unnecessary inflexibility. System designers invent word meanings and categorization schemes which may not match those of the end user. In an experimental study of design defects in a menu-driven database, Whalen and Mason (1981) found that miscategorization of information was the most serious defect. Whalen and Latremouille (1981) examined retrieval failure in Telidon through experimentation, and concluded that people are very likely to stop looking for information rather than undertake extensive searches. Fitter (1979) remarks that for a user to feel in control, it is vital that he recollects where he has been in the database. These investigations support the drawbacks which are identified above.

Although studies have indicated that menus are initially well determined for naive users, continued practice quickly renders the dialogue overdetermined. Geller and Lesk (1981) have run experiments comparing menus and command retrieval systems, and their results seriously undermine the premise that novices prefer menu-oriented systems with keyboard selection. Choice by menu selection is by nature slow and tedious. It is particularly likely to be overdetermined in more complex environments where diverse tasks are under-

taken, and in activities which involve context switching, which imposes a high overhead in navigating menu hierarchies.

### Window systems

The primary purpose of a window system is to provide the user with a variable number of virtual views into an information structure, all mapped to a single physical screen. If the views are regarded as virtual output devices, this allows one to communicate with different, asynchronous, processes in different windows—solving one of the problems of conventional terminal interfaces. A standard use of windows in editors and file systems is to allow multiple viewports into one or more documents. They can also be used in general computer systems by having command, input, output, and system state (feedback) windows. In general, windows can be opened, moved around, and closed, to reflect the demands of the task. Like a conventional VDU, a window system constitutes an outer shell within which other user interfaces can operate.

Existing window systems implement only rectangular windows aligned parallel with screen boundaries; there seems to be no good reason to extend this to arbitrary shapes and orientations. Many systems (e.g. Apple's *Macintosh*) allow windows to overlap (Figure 5); some (e.g. *emacs;* Stallman 1981) do not (Figure 6). This is a crucial design decision. Overlaps create considerable complications because they require the overlap area to be saved somewhere. In general there may be multiple overlaps, so that a tree structure of saved portions is needed. At some extra cost in storage, it may be easier to save

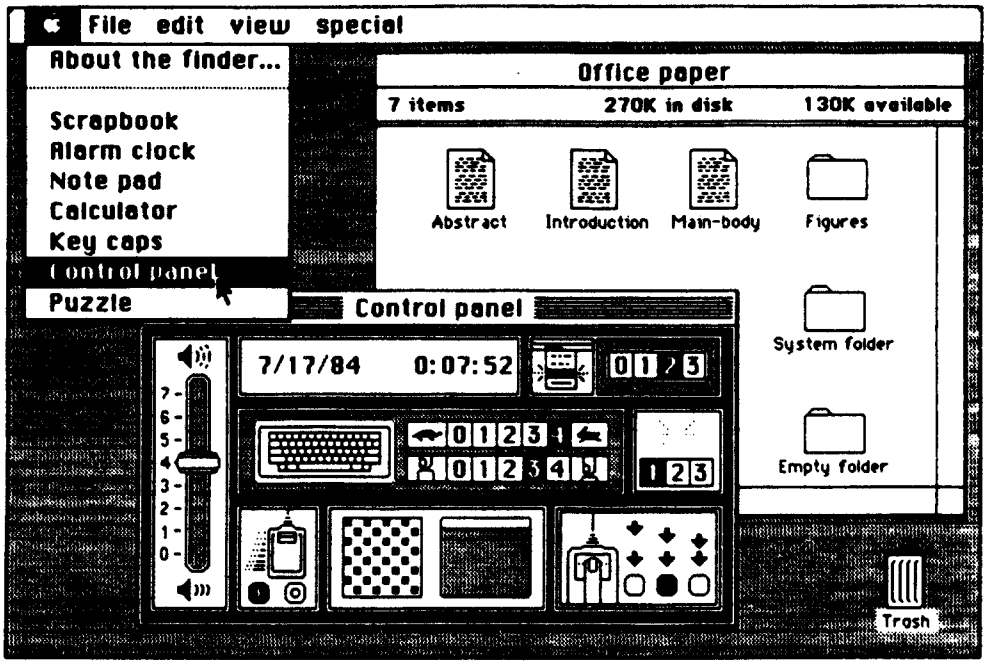


Fig. 5. A screen with two overlapping windows and a pull-down menu.

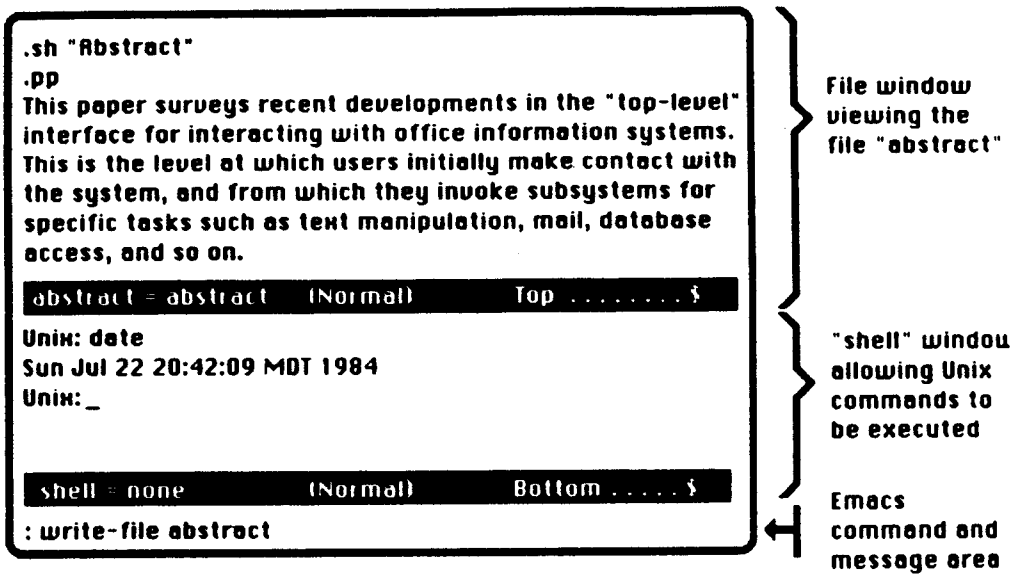


Fig. 6. A screen with tiled window: no overlapping allowed.

complete windows instead of overlapped fragments. However, then it will take longer to redraw when the window pile is altered, and redrawing time is usually a critical resource. We know of no published work which assesses the costs and benefits of overlapped windows in real applications; overlaps seem to be fashionable rather than provably beneficial. It has been observed that some users expend considerable energy in making their overlap displays resemble the tiled screen illustrated in Figure 6.

Windows can also be used to provide direct manipulation of objects on the screen. For example, it seems more 'natural' to transfer a chunk of text from one place to another by literally moving it between windows on the screen than by editing it out of the source document, saving it somewhere, reinvoking the editor, retrieving it, and placing it in the target. Imagine, for instance, moving an address from an address list to a letterhead. Unfortunately, as this example illustrates, what is often required is a copy rather than a move: this introduces some inconsistency with the cut and paste illusion.

Operations on windows are generally performed using a pointing device (such as a mouse), both to indicate positions to place windows on the physical screen and to denote a particular window to be manipulated. Typical operations include creating a window, binding it to a particular process, moving and resizing it, activating and destroying it, and shuffling the order of windows in a pile. The last is relevant only to systems which permit windows to overlap, and is usually implemented as an operation which brings a particular window to the top. If windows are identified by pointing at them, it is also necessary to be able to bury a window in case it completely obscures one below.

A primary concern of window systems is to allow convenient and

dynamic management of screen 'real estate'—a most valuable resource. Although normal windows remain on the screen until explicitly removed (by user or program), temporary pop-up or pull-down windows are useful for displaying transient information (Figure 5). These are typically invoked by pressing a button on the mouse, and persist until the button is released. They are particularly appropriate for menus, to clarify options, or to show help information. The pull-down menu in Figure 5 is one such example; access to normally hidden system functions are consistently available at any time.

A very important user interface design issue with window systems is who has the ability to create and destroy windows. At one extreme, it may be felt desirable to give the user complete control of his screen, so that he—and he alone—has the power to create and manipulate windows. This means that a program which needs a window must request that one be created and assigned to a suitable process. This runs the risk of causing the user considerable bother in doing simple things—another overdetermined scenario. At the other extreme, the user may have no direct control over the creation, placement, and destruction of windows. If a program wants a window, it creates it (and presumably the window system makes an attempt to place it sensibly on the screen). This will work best with non-overlapping windows. Of course, the user still has indirect control because it is he who invoked the program.

Many compromises between these two extremes are possible. One which seems attactive is to allow programs to call the window system to create subwindows within any windows that are allocated to it. At its simplest, this could involve dividing windows into non-overlapping 'panes'. However, for the sake of uniformity it may be preferable if the same window format and overlap possibilities are available to a program within its window as the user has within his screen. This raises the issue of whether a user is allowed to move a program-created window outside its parent window, and whether so doing changes its status as being under control of the program that created it (for the purposes of enlargement, deletion, and so on).

An important difference from conventional VDU systems is that programs executed within a window system do not know the size of their output device until run time (and even then it may change!). This causes difficulty because the best output format may vary substantially with radically different page sizes. For example, the new generation of hand-held computers use their very small displays in quite a different fashion to normal-screen personal computers. Some window systems allow the user to pan across a normal-width text buffer. Zoom would be nice but is somewhat impractical with current display technology (at least for text). However, viewing your text through a peephole makes it hard to scan easily. An alternative is to wrap lines when they reach the window boundary; but breaking lines at arbitrary points within words compromises readability. A

better (but computation-intensive) plan is to reformat all text dynamically into the current window size. This implies that programs should output structured 'document descriptions' instead of plain text.

The main advantages and disadvantages of window systems are outlined in Table 3. User reaction to such systems is extremely favourable, and a large number have become available recently. The Xerox Star (Smith *et al.* 1982), and Apple's Lisa and Macintosh (Williams 1983, 1984) are widely used systems which are aimed at the office and home market. Leading home computer software houses are now marketing window systems (e.g. Microsoft's *Windows* and VisiCorp's *VisiON*). Several window systems have been developed in research laboratories for use in a general programming environment; examples which are on the market are the Sun workstation; the Xerox Dandelion, Dolphin, and Dorado; and the Lisp Machines from LMI and Symbolics. Packages such as the Maryland (Wood 1982) and NUnix (Test 1982) window systems offer the programmer high-level subroutines for window manipulation within the Unix operating system. All of the above systems implement overlapping windows. Several tiling systems are also available, including the CMU Network Window Manager (Gosling and Rosenthal 1983) and the Waterloo Port user interface (Malcolm and Dyment 1983). These are particularly appropriate for use on ASCII terminals, especially in editors (e.g. *emacs;* Stallman 1981).

In view of the growing popularity of windows, there has been surprisingly little human-factors research on their use. There seems to be no experimental data on which to base fundamental design decisions such as whether windows should overlap, or whether only the user should be capable of creating them. Such research as has been

Table 3. Advantages and disadvantages of windows

| Advantages |
|---|
| • have a nice feel<br>• effective use can be made of pop-up windows to provide selective help and advice<br>• ideal for controlling and monitoring asynchronous events<br>• can realistically simulate real-life cut-and-paste operations |
| **Disadvantages** |
| • complexity<br>• need a graphical display<br>• difficult to format program output into arbitrary-sized windows<br>• the user has to cope with a new meta-level of control decision-making<br>• no obvious way of allocating control over windows between user and computer<br>• program creation of windows may be underdetermined for some, while explicit window creation may be overdetermined for others |

done addresses technical questions of implementation (e.g. Pike 1983, Rosenthal 1982) rather than usage.

In the office setting, the success of window systems hinges on the need for viewing or interacting with many different contexts simultaneously, and the need to suspend one context temporarily while dealing with a more urgent interruption. The extra degree of meta-control required for window manipulation may create an under-determined interface for the untrained or occasional user.

### Forms

Paper forms are widely used tools for structuring information in conventional offices. It is natural to consider the use of forms as an integrating medium for office computer systems. A paper business form is a template which, when filled in, becomes a text document. Either it can be viewed as a document in itself, or the filled slots can be viewed as a collection of entries in a database. This dual nature gives the form an important advantage over other ways of recording the information.

A conventional paper form can play any or all of the following four roles:

- display of information, as a structured and stereotyped document
- collection of information (and its modification)
- storage and retrieval of information as records in a database
- transfer of information as messages.

Each of these can be expanded somewhat within a computer-based office forms systems. Firstly, the medium in which a form is presented is not necessarily restricted to paper. Viewed as a document, a form can be reproduced on a VDU screen or on paper, typed or typeset, or (perhaps) spoken over the telephone. Different text templates can be used to present different views of the same information, or subsets of it. In addition, only the necessary portion of a form is disclosed to the viewer. An office worker requisitioning supplies need not concern himself with blank (or restricted) fields filled in later by the purchasing department. Secondly, a similar variety of media is possible for collection of information: hand-printing on a tablet, VDU interaction, speech input, off-line keyboarding. Thirdly, the relational model of databases is the most natural for storage of the information contained in forms. Each instance of the form expresses a single tuple of a relation. Viewed in this way, implicit references may be made to other database entries so that fields may be filled in automatically (e.g. to fill the address field, look up the address associated with the name field). Or fields may be defined to contain the result of operations on other fields (e.g. a 'total' line)—such entries must be manually calculated for paper forms (recall your income tax form). Finally, mailing of forms may be expedited by transmitting

only the form identification and the contents of the fields; the text template can be regenerated by the recipient from his master copy. This is not feasible with paper forms.

In essence, a 'form' comprises both a structured data type, which specifies what lexical objects may occupy each field; and mappings (typically involving text templates) which make an instance of that data type into a message. The first part may be more than just a type definition. It may include specifications of fields which must be filled in by the system (e.g. today's date), optional fields, and default values for fields. If security is an issue, it may include classification levels for who is allowed to fill certain fields (e.g. signatures). Certain action may need to be taken when fields are filled in a certain way. In general, the form may comprise a data type encapsulated with specified procedures for syntax and security checking of each field—a Simula class, a Smalltalk object, or an Ada package. As for mapping from data-type instances into messages, there may be several mappings associated with a single form data type, to present (subsets of) the same information in different ways. The mappings may involve more than simple templates because calculations may be necessary to fill in totals and so on. These also would fit naturally into a package- or object-based programming structure.

As an example, consider the filled-out form shown in Figure 7, being used to add new information into a database. An office worker may complete the blank form using a keyboard, moving from field to field with (say) the tab key or a pointing device. Perhaps default values are given to some fields, such as *City*. If this is a new form, the *File Number* and *Date* may be added automatically. A database may be consulted to simplify the form-filling task; for example, the *Address* may be added automatically once the *Name* is known. After completion, the form becomes part of a database. Its contents may be accessed through a query language, perhaps also in the shape of a form. For instance the same form can retrieve suppliers of pencils by selecting the *query database* option and entering that item under *Supplies*.

The act of entering a form may cause other, more indirect, side effects. Completion of a 'mailing' form may cause mail to be sent. Completion of a 'reminder' form may set a trigger which will be fired at some future time. The form in Figure 7 may be sent automatically to the purchasing department for action and the inventory updated when the appropriate triggers are chosen on the 'trigger menu'. It is through generalizations such as these that forms can play a potent role as office system interfaces.

There are presently few widely known general forms systems. However, some special cases illustrate the possibilities. Spreadsheet calculators (for an example see Williams 1982) provide an excellent example of the utility of the simple idea of computational dependencies between items in a single form. However, they are restricted to matrix-like forms with no text template. QBE (Zloof 1977) shows

'text-less' forms working towards another purpose; namely providing a casual-user interface to a relational database. The experimental Officetalk-D (Ellis and Bernal 1982) is one of the first to try to integrate completely an office information system through the forms paradigm. Through it users view all system activity, manipulate transactions, and browse through databases. Database inference languages such as Prolog (Clocksin and Mellish 1981) offer attractive features for implementing forms systems.

Consider the potential of using a form to represent *directly* some aspect of the internal operation of a system. Altering fields of the form has an immediate and easily understood effect on the system. Editing is such a pervasive operation in all interactive computer systems that it is attractive to combine it with the structuring imposed by a form and extend it to manipulate more general objects within an interactive system. In fact, editing means examining and modifying data, and there is no reason why the concept should not be extended to types of data other than plain text. For example, utilities that delete and rename files edit directories; those that send mail edit mailboxes. Duplicating command languages for these conceptually similar operations is unnecessary and frustrates casual users.

As an example, Fraser (1980) discusses the possibility of editing directories. Having invoked on Unix *edit* ⟨*directory-name*⟩, the user may see a display set as a form similar to that of Figure 8. At this point he could delete the first line (effectively removing a file), create files and more importantly directories, copy files, change the file permissions in the left column, or change the owner's name. These five operations account for 16 per cent of all command accesses on Unix (Fraser 1980).

As this example illustrates, computer-based forms can differ considerably from conventional paper forms—some refer to them as

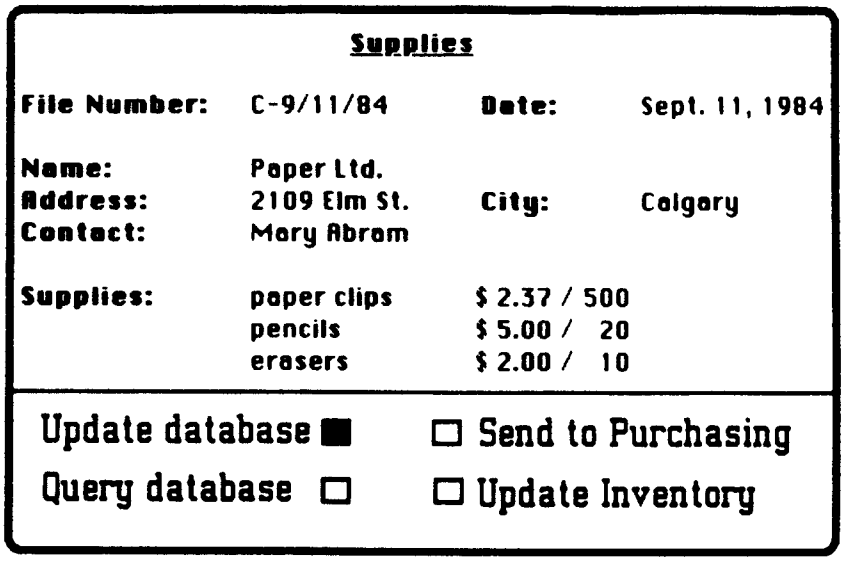


Fig. 7. A filled-out form.

'smart' forms rather than 'dumb' ones. Their strong and weak points as an office system interface are summarized in Table 4. A well-designed forms system can provide a well-determined interface for naïve users, for it is simple and based upon a familiar model. It capitalizes both upon existing skills in text editing or word processing, and upon familiarity with the use of forms as tools for structuring data. The power available in extended and active forms suggests that even experts may find them attractive.

### Natural language interfaces

It seems ideal to be able to express your wishes and ideas in natural language, as you would to a colleague or secretary. However, despite the existence of some sophisticated example systems, natural language has not yet achieved the maturity of the other techniques discussed here.

Some of this is due to the difficulty of handling the knowledge which is needed to decode the input text and understand what is being said. Barrow (1979) puts it nicely:

> In current attempts to handle natural language, the need to use knowledge about the subject matter of the conversation, and not just grammatical niceties, is recognized—it is now believed that reliable translation is not possible without such knowledge. It is essential to find the best interpretation of what is uttered that is consistent with all sources of knowledge—lexical, grammatical, semantic (meaning), topical, and contextual.

While it is perfectly possible to build systems to 'understand' (i.e. react appropriately to) natural language for certain tasks domains, all current systems are limited to rather specific and narrow domains. There are natural language interfaces for moving objects in a toy world (Winograd 1972), investigating the properties of electrical circuits (Brown and Burton 1975), database retrieval for a variety of databases (Woods 1973; Waltz 1975; Harris 1977; Hendrix *et al.*

| PERMISSION | OWNER | FILENAME |
|---|---|---|
| -rw-r--r-- | ian | abstract |
| drwxr-xr-x | ian | figures |
| -rw-r--r-- | saul | introduction |
| -rw-r--r-- | ian | main-body |
| -rw------- | ian | paper.latest |
| -rwx--x--x | ian | print-paper |

Fig. 8. A display ready for generalized editing.

Table 4. Advantages and disadvantages of forms

| Advantages |
| --- |
| • familiarity<br>• provide a 'natural' interface to databases<br>• encourages strong type-checking on data entry<br>• can be extended in several ways over paper forms |
| **Disadvantages** |
| • emotional overtones (everyone hates forms)<br>• extensions must be carefully thought out because they conflict with the basic metaphor |

1978), giving advice to medical diagnosis programs on highly constrained topics (Davis and Lenat 1982) ... the list goes on and on (see Gevarter 1983 for an excellent survey). Each system works in a tightly constrained environment. Although you could build a natural language command interface for a highly circumscribed office-system interface, current technology cannot cope with a rich, varied array of topics and interactions.

Understandable communication in English is highly dependent on many factors, such as inaccuracies in simple logical relationships within a sentence and a need for metacomments for clarification. Although relationships of various kinds are normally required in programming languages, people are imprecise in using logical connectives and quantifiers in natural language (Thomas and Carroll 1981). The first three examples in Figure 9 illustrate this ambiguity—each sentence can have multiple meanings. Metacomments—messages about the communication itself—provide a higher-level communication structure (Thomas and Carroll 1981). They may direct the conversation, adjust its speed, or reflect the internal state of the communicator (Thomas 1978). A metacomment would clarify the dialogue context in the final example in the Figure ('I'm talking about office supplies'). In all these cases, the role of a clarification dialogue between user and computer is important to ensure that communication is effective. For example, the ROBOT system responds to ambiguous queries by presenting the possible interpretations to the user (in a formal database language) and asking him to choose the correct one (Harris 1977). Artificial languages are not normally associated with this ambiguity. One advantage of not using natural language is that the 'special notation of a precise concise artificial language can be a helpful tool in guiding thought processes' (Shneiderman 1980), thus avoiding some of the pitfalls mentioned above.

Another part of the problem is the complexity of a natural language interface. It is hard to build, unwieldy, slow, difficult to maintain, and consumes a great deal of resources.

| Type | Sentence | Meanings? |
|---|---|---|
| Simple logical connectives | Get the suppliers of the red and green binders. | •a single binder is red and green<br>•red or green binders are wanted |
| Quantifier | Find files for Smith and Jones for 1983 and 1984. | •Get both their files for the years 1983 and 1984<br>•Get Smith's file for 1983 and Jones' file for 1984 |
| Relations within a sentence | Supply-all Inc. refuses to supply Nopay Ltd because they have financial difficulty. | •Supply-all has financial difficulty<br>•Nopay Ltd has financial difficulty |
| Context dependency | Tell me about rulers. | •A measuring device in an office supply context<br>•A leader in a political database context |

Fig. 9. Examples of ambiguous natural language sentences.

But even if the technical problems were solved, would you want to use natural language anyway? A lot of typing is needed to express complex thoughts precisely on paper. Although speaking natural language is a much more attractive possibility for the user than typing, existing speech recognition systems are still highly limited research projects. And to what extent should the system forgive lapses of grammar, misspelling, use of slang, ellipsis?—and what effect will forgiveness have on the precision of the communication? Inevitably, only a subset of natural language will be understood, and the user will have to learn this subset by trial and error. It's very hard to constrain your word and syntax usage according to pre-specified rules (Shneiderman, 1980).

Even then, there will be a temptation to ascribe unreasonable powers to the machine—it's easy to forget its limitations if it appears to 'understand' (Weizenbaum 1976). Some argue that forcing machines into the mold of natural language discourse will not be useful, regardless of improvements in the technology.

One possible role of natural language is to use short phrases in conjunction with another scheme like menu selection. For example, in a personnel database you could type 'less than $30 000', or 'at least as much as the average of employees in his Department' against the 'salary' item in a menu. Using fragments like these overcomes some of the human problems associated with typed natural language (Cuff 1982).

Table 5 summarizes some of the advantages and disadvantages of natural language interfaces. In the distant future, it may be that natural language speech interfaces will provide an alternative to more artificial present-day techniques. The role of the *dialogue* will be paramount in clarifying for the system what it is the user wants to do. However, the social consequences of anthropomorphizing machines by mimicking human conversation are unknown and may have far-reaching effects on the viability of such systems.

## Integration—the final package

Manufacturer's descriptions of office automation systems frequently make much of their 'integrated' nature. Along with 'user friendly' and 'intelligent', this has become a buzzword which often means little. The standard dictionary definition is 'to form into a whole: unite'. This could be fleshed out in the context of an office information system as one which is united by a common interface used both between and within applications. Let us see how it could apply to the interfaces introduced in the last section.

The command-driven interface, exemplified by Unix, is somewhat integrated at a shallow level. System utilities are reached through commands. Options are signalled by flags. Pipelines and input/output redirection are normally available. Descriptive manual entries are in a standard format. (See Kernighan and Mashey, 1981, for a description of integration within the Unix programming environment.) However, Unix is far from integrated at the application level, for each application program is created independently. For example, one would expect that a document preparation language and a text editor would be closely associated. This is not the case in Unix. The editor knows nothing about the document-language constructs and shares no common commands. As an alternative command interface,

Table 5. Advantages and disadvantages of natural languages

| Advantages |
|---|
| • 'natural' for the user<br>• requires no learning (except the trial and error necessary to determine what subset is actually implemented) |
| **Disadvantages** |
| • complexity; excessive use of resources<br>• typing is a nuisance<br>• it is hard to make the system forgiving of errors and yet retain precision<br>• may present false image of capabilities to the user<br>• context and semantics of the natural language may not be known to the user<br>• system will never be complete<br>• clarification dialogues are required to ensure accurate communication. |

the *Menunix* system provides greater integration by using the workbench metaphor to unify the user's view of the system. Unfortunately, this is just a patch job; the software subsystems remain unchanged.

Windows constitute an outer shell of the interface, giving a set of 'virtual' views into the system. A window system provides integration only at that level. Together with a universal editing interface, both forms and natural language offer some potential as a common interface between and within applications.

Menus, forms, and window systems are by no means mutually exclusive, and successful systems will doubtless use them together. There are now some highly interactive programming environments which combine many interface styles to allow power in program creation, testing, debugging and documentation. One example is the display-oriented *programmer's assistant* for Interlisp, which successfully uses windows and pop-up menus to provide easy switching of tasks and contexts (Teitelman 1979). An extension of the programmer's assistant is the Smalltalk-80 *system browser* which helps in the structuring, navigating and editing of Smalltalk 'objects' (Goldberg 1984). This browser binds taxonomic menus to four 'subviews' drawn in window panes—each representing a different contextual level of the language—which determine what text is visible in a main subview. Pop-up menus normally provide context-specific commands for each pane. Outside the programming domain, the Smalltalk-80 browser shows promise in viewing structured documents (Weyer 1982).

The above systems are integrated by the programming environment they are meant to support, a model ill-suited for our office worker. Thus different higher-level concepts are needed to support integration. One cornerstone of system integration in modern user interfaces is *direct manipulation* through *metaphors*. This section describes these terms and reviews two examples: desktop simulations and soft machines.

### The direct manipulation paradigm and metaphors

> Piaget [the child psychologist] has hypothesized that infants first learn about causation by realizing that they can directly manipulate objects around them—pull off their blankets, throw their bottles, drop toys ... Such direct manipulations, even on the part of infants, involve certain shared features that characterize the notion of direct causation that is so integral a part of our constant everyday functioning in our environment—as when we flip light switches, button our shirts, open doors, etc.
>
> Lakoff and Johnson 1980

Shneiderman (1983) has identified an attribute which is common to most systems which generates enthusiasm amongst their users. He defines a 'direct manipulation' interface as one which behaves as though the interaction was with a real-world object rather than with an abstract system. Most good video games are certainly direct

manipulation interfaces, and—with increasing stretching of the imagination—so are forms and window manipulation. Even menu selection can be viewed in this light. The simulated realization of familiar concrete systems is an important general idea which deserves consideration in its own right.

The central ideas in direct manipulation 'seem to be visibility of the object of interest; rapid, reversible, incremental actions; and replacement of complex language syntax by direct manipulation of the object of interest' (Shneiderman 1983). They generate 'glowing enthusiasm amongst users—in marked contrast with the more common reaction of grudging acceptance or outright hostility'. Shneiderman gives several examples, such as

- display editors
- spreadsheet calculators
- spatial data management (e.g. the *put-that-there* system: see Bolt, 1980)
- video games
- interactive CAD/CAM systems;

as well as several less familiar or suggested ones:

- Plato CAI lessons which allow you to manipulate simulated chemical apparatus
- a simulated Rolodex card file index for addresses
- a simulated check book for financial records
- bibliographic searching based on a catalog in simulated index drawers
- airline reservation using map, calendar, clock, and seating plan.

The most striking semiotic difference between conventional command-driven computer interfaces and direct manipulation is that the former emphasize *verbs* (or actions), while the latter emphasize *nouns* (actors, or objects). Conventional computer commands begin with an action like 'copy', 'move', 'edit'; and continue with specifications of the actors. Since it is difficult to represent the verb in any way other than as a word, the interaction is composed of textual commands. By turning the emphasis around and concentrating on *objects* as the primary entities, communication can be made more concrete because the objects can be represented pictorially rather than linguistically. Then the action can be specified by indicating movement directly (e.g. with a mouse), or for more complicated actions by selecting a verb from a (perhaps pop-up) menu. By altering the menu so that only appropriate actions are included, errors of syntax can be avoided altogether. Although not all applications are amenable to this object-oriented approach, the clerical office, with its concrete tasks, is well suited to direct manipulation paradigms.

Direct manipulation systems map the interface structure on to some facet of the real world, and then proceed to simulate this. Not all aspects of the 'real world' model need be simulated, of course, but

only those relevant to the operation of the interface. The mapping is a 'metaphor' in a sense close to the dictionary definition:

**metaphor,** n. Application of name or descriptive term to an object to which it is not literally applicable.

Concise Oxford Dictionary

Carroll and Thomas (1982) approach this metaphor-based method of designing interfaces from the viewpoint of the naive user learning how to interact with the system. They make several recommendations for the interface designer, which are worth quoting in full.

1. Find and use appropriate metaphors in teaching the naive user a computer system.
2. Given a choice between two metaphors, choose the one which is most congruent with the way the system really works. The more aspects of the system that can be 'covered' by a single metaphor, the better.
3. Take care to ensure that the emotional tone of the metaphor is conducive to the desired emotional attitude of the user.
4. When it is necessary to use more than one metaphor for a system, choose metaphors drawn from a single real-world task domain (i.e. similar enough) but do not choose objects or procedures which are exclusive alternatives from within that domain (i.e. not too similar).
5. Consider the probable *consequences* to users and system designers of the metaphor.
6. When introducing a metaphor, explicitly point out to the user that it is not a perfect representation of the underlying system and point toward the limits of the metaphor.
7. Keep in mind from the beginning that any metaphors presented to the user are to give an overview of the system and that there may be a time, at least for the continual user, that the metaphor is no longer useful.
8. Provide the user with exciting metaphors for routine work and eventually present the user with a variety of scenarios which represent different views and different actions but whose underlying structure is identical.

Carroll and Thomas 1982

As noted, many of the interface paradigms sketched earlier can be viewed as examples of direct manipulation systems. These recommendations draw attention to the fact that the designer must make an analogy with some real-world structures. This requires a good deal of imagination and creativity, and considerable sensitivity in selecting the metaphor (not, for example, laboring the analogy too far). It is vital to recognize that the user will inevitably import other, unwanted, aspects of the metaphor which may interfere with his conceptualization of the interface.

### Desktop simulations

What is an appropriate metaphor for the office worker? When the 'paper' office is examined, the desktop is found to be the main working area. Information is contained in documents (which includes memos and letters), while management of these documents is through in/out trays, folders and filing cabinets. This paper office can be realized by simulating a desktop on a VDU screen.

The desktop metaphor was pioneered commercially in the Xerox Star (Smith *et al.* 1982) and popularized by the Apple Lisa and Macintosh (Williams 1983, 1984). The Star is a Cadillac personal office computer, with a fast processor, high-resolution bit-mapped display with mouse, substantial fast main and local disk memory, and a network connection. Although the mouse had been used previously for several years as a pointing device in Xerox research computers, the Star was the first commercial end-user system to incorporate it. Lisa is less powerful, with a standard 16-bit microprocessor (M68000), medium bit-mapped display with mouse, a lesser amount of main memory, and a hard disk with floppies for archiving and external communication; Macintosh is less powerful yet (but cheaper!). The most significant difference is that the Star display has much higher resolution, with three times the number of pixels. The general design of Lisa's top-level interface is similar to that of the Star, although it lacks some advanced features.

The Star screen simulates a desktop with *icons* (or pictograms) that represent familiar office objects:

- in baskets
- out baskets
- file folders
- documents
- calculators
- printers
- blank forms for letters and memos.

There is a small number of universal commands that can be used throughout the system, and on any object. They are *move*, *copy*, *delete*, *show properties*, *again*, *undo*, and *help*; all available on the keyboard. Each performs the same way regardless of the type of object selected. For example, you can *move* text in a document or *move* a document in a folder. As is common in window-based interaction, the shape of the cursor changes to indicate to the user what is happening. This provides very natural feedback since attention is generally focused on the cursor. For example, the Star cursor distinguishes *move mode*, *copy mode*, *graphics mode*, *menu-selecting mode*, and so on.

Data structures called *property sheets* are readily available at all times when interacting with the Star. The property sheet is a pop-up form, alterable by the user, which describes the physical qualities of the current context. Within a document, for example, a property sheet indicates font, line spacing, and margins, as shown in Figure 10. In a graphical context, the property sheet may indicate fill type and line style. Even though the attributes of the forms differ, the property sheet presents a consistent and integrated interface to any context.

The Star is an excellent example of integration across all applications. Commands are universal. Consistent feedback on the cur-

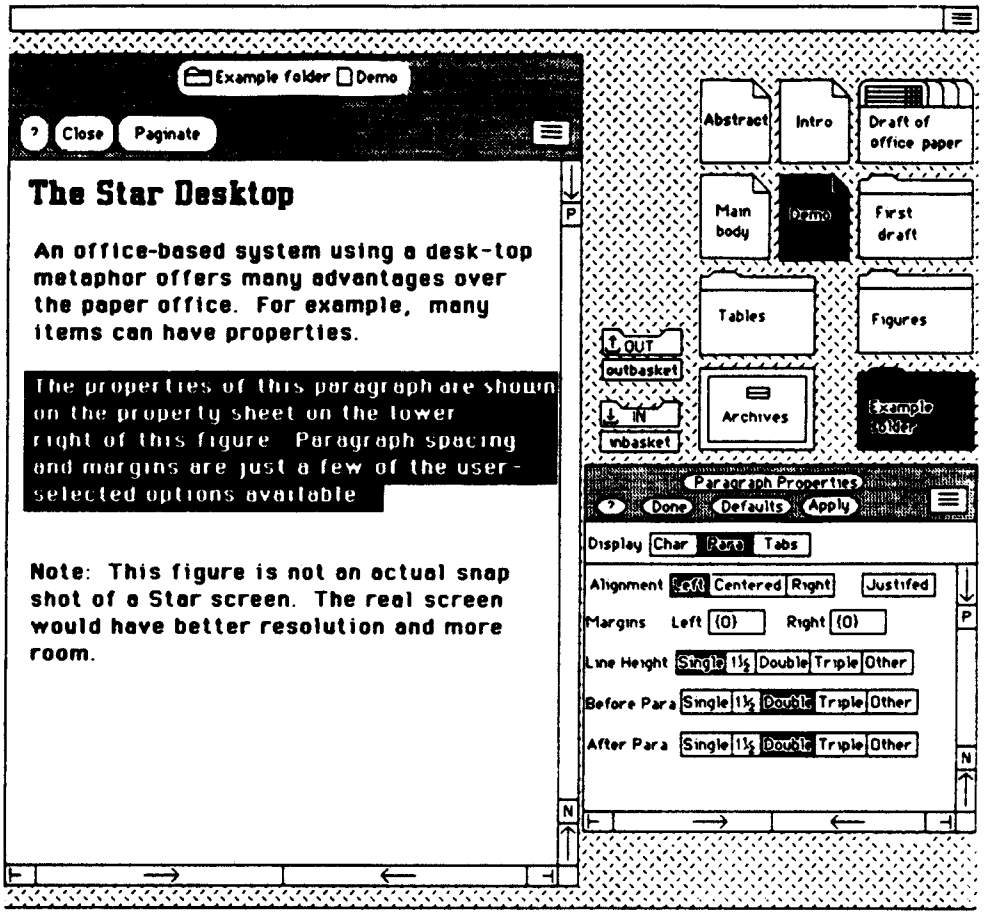


Fig. 10. The Star desktop and keyboard.

rent operation is provided by the shape of the cursor. Property sheets are always available, and have a consistent format.

In the Star user interface, every action has a visible effect on the screen. The designers note that 'a subtle thing happens when everything is visible: *the display becomes reality*. The user model becomes identical with what is on the screen. Objects can be understood purely in terms of their visual characteristics. Actions can be understood in terms of their effects on the screen.' (Smith *et al.* 1982.) Each action has an obvious effect even if you have not tried it on that kind of object before.

It is most important to note that the success of this strategy depends on the fact that intuitively reasonable actions can be performed on objects *at any time*. Once the system has gained his trust, a user should feel secure in this environment. He *knows* that moving a page of a document is like moving a memo to a file or an address to a letterhead. He *knows* that paper files don't just disappear or become garbled by themselves, so by analogy it's 'impossible' for this to happen on his screen. But woe betide the applications program designer who ever betrays this trust!

The word-processor touchstone 'what you see is what you get' is virtually attained by the Star. Although the screen resolution of 72

dots per inch does not approach that of a typesetter (typically 400–1000 dots per inch), it is still enough to make clear distinctions between different font styles and sizes. Star makes $2^{16}$ characters available (enough for 512 different 128-character alphabets). Input of obscure symbols is handled by keyboard interpretation maps generated on the screen.

Another consequence of the desktop object-based dialogue is the altering of the user's view of the computer. Traditional computer concepts include text and graphics files, binary application programs, directories for organizing files, local and centralized disk storage, and physical visual display units to work on. Within the Macintosh desktop, these are transformed. A file is a generic name for a container of information, which can be a document (user-generated information), or a tool (which manipulates documents). Folders contain files or other folders. A desktop is the working environment that uses windows to present and interpret all information (Espinosa and Hoffman, 1983). The Xerox Star extends this view: folders are local repositories of files, whereas filing cabinets are centralized storage repositories. In this new manner of visualizing the computer, it is the office application domain which is dominant in thought, not the machine itself.

The strength of the metaphor in an office context is that the user interacts with familiar objects just as he would in the physical world. Unfortunately, this strength is also its weakness. We have moved from commanding the general-purpose computer to do office tasks to a specific office machine, which disallows 'normal' computer functions. The greatest loss is that of programming. The usual office metaphor is excellent at manipulating normal and routine office procedures, but is very poor at dealing with procedural specification. How, for example, does an office worker tell the system to notify him of mail from his superior and no one else? And how about more complex procedures, such as asking the system to process all audit reports on the second week of the month by mailing a copy to accounting and the archives and then deleting the report from the desktop? Although these procedures could easily be programmed on a general-purpose machine, the specific office machine would not allow it, unless it was predicted beforehand. Some user-oriented programming languages do exist (Martin 1984), but they depart sharply from the desktop metaphor. Programming by example, in which the system infers a procedure after observing examples of its execution by the user, is a topic of current research. Some success has been achieved, but only in very limited applications. Although more complex abstractions such as generalizations, conditionals and iterations are possible (Halbert, 1981; Gaines, 1976; Witten, 1981), it will be some time before full procedural specification arrives in the office marketplace.

To summarize, the advantages and disadvantages of the desktop simulation are presented in Table 6. Considering their infancy, desk-

Table 6. Advantages and disadvantages of desktop metaphors

| Advantages |
|---|
| • has a nice feel, creates enthusiasm, etc.<br>• reduces the learning effort<br>• is predictable<br>• concrete rather than abstract |
| **Disadvantages** |
| • complexity<br>• needs a graphical display<br>• difficult to design<br>• user imports unforeseen expectations<br>• concrete metaphors may not be applicable to all office tasks<br>• difficult for user to 'program' routine tasks<br>• extension of metaphor may conflict with the user's model |

top systems have been remarkably successful. Although certain interface issues still need resolving (e.g. continual switching between mouse and keyboard), the fervor of most users—including computer novices and experts—indicates a dialogue that is well-determined for a wide spread of people. However, this success is predicated upon a close fit between the desktop model and the needs and expectations of the user. Such a system quickly becomes overdetermined when users try to break out of the confines of its metaphor. Conversely, users may lose control if the task has been distorted unnaturally to fit the desktop model.

### Soft machines

Soft machines take the combination of direct manipulation and metaphors to its logical conclusion. A soft machine is a simulation of an existing physical machine that it is designed to supplant. One of the criteria stated previously for choosing a metaphor is that it must be appropriate. To the extent that it is a metaphor, there is potential for the illusion to fall. A soft machine removes this possibility by having the interface simulate directly all features of a physical machine.

Machines are normally special-purpose. Shape and form suggest function, and they are operated with controls which have an obvious one-to-one correspondence with their effects. These attributes make machines easy to learn and efficient to use, as opposed to the computer (Figure 11). The new user generally benefits from positive transfer of his experience with other machines. Computers are the opposite. They are general-purpose, with the controls (normally a keyboard) giving little clue as to the function of the application program. Nakatani and Rohrlich (1983) propose the soft machine as a way of demystifying the inscrutable computer. They define it as 'a

| | Machines | Computers |
|---|---|---|
| Learning | •play | •work |
| Efficiency | •specialized controls<br>•direct | •general purpose controls<br>•circuitous |
| Transfer | •easy | •hard |

Fig. 11. Machines versus computers (after Nakatani & Rohrlich, 1983).

machine realized through computer generated images of controls ... with a touch-sensitive screen for actuating them'. The prime example is a simulated calculator used in exactly the same manner as a physical one (Figure 12). Another instance is the automated teller of Figure 2 realized on a graphical screen. Aside from the card, cash and receipt dispensers, a touch screen could easily take the place of the physical menu buttons. The *control panel* in Figure 5 is yet another soft machine using a variety of controls: the sliding bar controls bell volume, while push-buttons adjust reactions of both mouse and the keyboard. A fourth example is a graphical Rolodex card catalogue for accessing telephone directories (Shneiderman 1983).

Nakatani and Rohrlich (1983) propose a method of integrating links between *soft* machines by analogy with tools in a workshop. The hierarchy used is a *tool bin*, which is the entire set of tools, a *workshop*, which collects similar tools (as in *Menunix*), and a *workbench* on which the actual work is done. Obviously, this metaphor is similar to the desktop one in the Star.

Advantages and disadvantages of soft machines are given in Table 7. Two limitations of hard (physical) machines overcome by soft

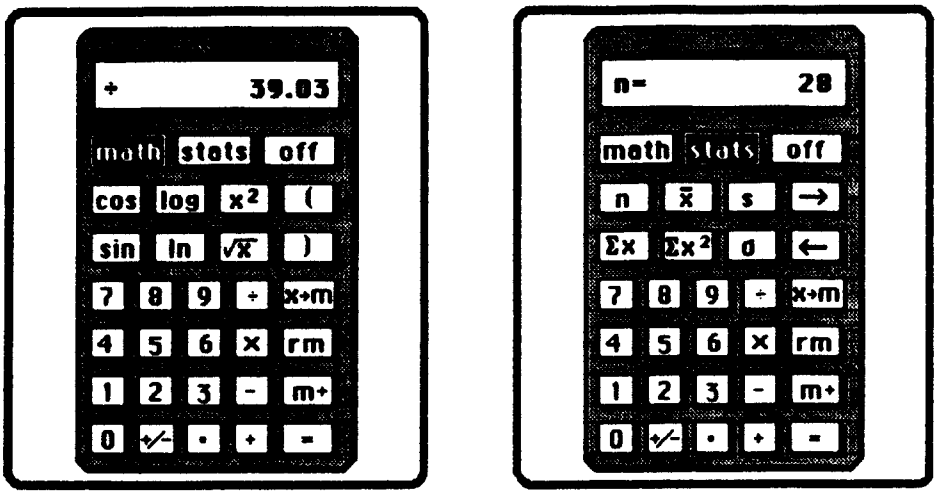


Fig. 12. The calculator as a soft machine.

Table 7. Advantages and disadvantages of soft machines

| Advantages |
|---|
| • metaphor extremely close to reality<br>• allows progressive disclosure and soft labeling of controls<br>• high positive transfer from the physical world<br>• application recognizable by its controls<br>• a body of knowledge already exists on human performance on machines (ergonomics) |
| **Disadvantages** |
| • highly physical representation may not be feasible for many systems<br>• needs graphical display<br>• input devices inadequate for our normal use of controls (e.g. pressure, tactile feedback) |

(simulated) versions are inflexibility and management of complexity. Hard machine controls cannot be changed or hidden, whereas soft machines can easily be relabeled, and controls disclosed progressively only when appropriate. For example, many calculators now have two or three labels attached to each function button. The soft calculator need only show the labels of the current context. In statistics mode it would show statistics functions, and not, for example, mathematical labels (Figure 12). Of course, not all applications fit the soft machine paradigm. But one can take existing office machines which are already well-determined for the worker and simulate them on the computer, thereby decreasing the risk of interface failure.

## Summary

We have looked at many types of user interfaces: command-driven dialogues, menus, windows, forms and natural language. In all of them, integration is a concern. Some are amenable to integration via direct manipulation: through desktop simulation or soft machine representation. Others do not fit these paradigms easily. The styles and flavors of the interface also reflect their end use. Command interfaces are normally used by computer wizards for their power and flexibility. Desktop metaphors and soft machines, on the other hand, reflect a tightly constrained office environment.

But the choice of the 'best' interface is not an obvious one. Office workers will find the very polished desktop window systems inadequate for non-routine tasks, for they lack easy-to-use facilities for specifying procedures. This, of course, is no problem for the computer-oriented dialogue which offers full programming capabilities along with its unfamiliar interface paradigm. Between these two extremes are partial solutions. Menus provide a proven interface for naïve users who may need only very casual and predictable access to applications and/or information. The form metaphor offers a direct

realization of its paper counterpart, with promise of powerful extensions. Natural language, a seemingly desirable interface, is likely to be of little use outside highly constrained domains; due to uncertainty of language meaning, the limited understanding of language by computers, and the primitive state of continuous speech recognition technology. Windows provide a good means of organizing dialogue, but they are really a foundation to interfaces rather than an interface in themselves.

The final solution is still to come. Perhaps it will be a conglomerate of the techniques discussed so far. Or it could be a radically new way of viewing the office. Undoubtedly it will be integrated in a manner both familiar and understandable to the office worker. Whatever interface style is chosen, it is not enough for the system architect to have adhered to all the proper interface guidelines during its design. The manager must strive to match the dialogue determination level with the needs, training, conceptions, and preconceptions of the users.

## References


Barrow, H. G. (1979). *Artificial intelligence: state of the art.* Technical Note 198, SRI International, Menlo Park, CA.

Bolt, R. A. (1980). Put-That-There: voice and gesture at the graphics interface. *Proc Siggraph.* **80**, 262–70, Association for Computing Machinery.

Brown, J. S. and Bruton, R. R. (1975). Multiple representations of knowledge for tutorial reasoning. in *Representation of Learning* eds D. G. Bobrow and A. Collins). Academic Press, New York.

Card, S. K., Moran, T. P. and Newell, A (1980). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM.* **23** (7), 396–410.

Carroll, J. M., and Thomas, J. C. (1982). Metaphor and the cognitive representation of computing systems. *IEEE Trans. Systems, Man, and Cybernetics. SMC-12* (2), 107.

Clocksin, W. F., and Mellish, C. S. (1981). *Programming in Prolog.* Springer-Verlag, Berlin.

Cuff, R. N. (1980). On casual users. *Int. J. Man–Machine Studies.* **12**, 163–87.

Cuff, R. N. (1982). *Database query using menus and natural language fragments.* Ph.D. Thesis, Man–Machine Systems Laboratory, Department of Electrical Engineering Science, University of Essex, Colchester, Essex, UK.

Davis, R., and Lenat, D. B. (1982). *Knowledge-based systems in artificial intelligence.* McGraw-Hill, New York.

Dumais, S., and Landauer, T. (1982). Psychological investigations of natural terminology for command & query languages. In *Directions in human/computer interactions* (eds Badre, A., and Shneiderman, B.) pp. 95–110. Albex Publishing Co., Norwood, New Jersey.

Eason, K. D., and Damodaran, L. (1979). Design procedures for user involvement and user support. *Infotech—Man Computer Communications.* London.

Ellis, C. A., and Bernal, M. (1982). Officetalk-D: an experimental office information system. *Proceedings of the 1st ACM SIGOA Conference.* 131–40.

Engel F. L., Andriessen J. J., and Schmitz, H. J. R. (1983). What, where and whence: means for improving electronic data access. *Int. J. Man–Machine Studies.* **18**, 145–160.

Espinosa and Hoffman (1983). Macintosh user interface guidelines (2nd edition). In *Inside Macintosh.* Apple Computer Inc.

Fitter, M. (1979). 'Toward more natural interactive systems. *Int. J. Man–Machine Studies.* **11**, 339–50.

Foley, J. D., Wallace, V. L., and Chan, P. (1984). The human factors of computer graphics interaction techniques. *IEEE Computer Graphics and Applications.* **4** (11), 13–48.

Fraser, C. W. (1980). A generalized text editor. *Communications of the Association for Computing Machinery.* **23** (1), 27–60.

Gaines, B. R. (1976). Behavior/structure transformations under uncertainty. *Int. J. Man–Machine Studies.* **8**, 337–65.

Gaines, B. R., and Facey, P. V. (1975). Some experience in interactive system development and application. *Proc. Institute of Electrical and Electronic Engineers.* **63** (6), 894–911.

Gaines, B., and Shaw, M. L. (1983). Dialog Engineer-

ing. In *Designing for human–computer communication* (eds M. E. Sime and M. J. Coombs), pp. 23–53. Academic Press, London.

Geller, V. J., and Lesk, M. E. (1981). *How users search: a comparison of menu and attribute retrieval systems on a library catalog.* Internal report, Bell Laboratories.

Gevarter, W. B. (1983). *An overview of computer-based natural language processing.* NASA Technical Memorandum 85635, Washington, DC.

Goldberg, A. (1984). The influence of an object-oriented language on the programming environment. In *Interactive programming environments* (eds Barstow, D. R., Shrobe, H. E., and Sandewall, E.) pp. 141–74. McGraw-Hill, New York.

Gosling, J. A., and Rosenthal, D. S. H. (1983). *A network window-manager.* Report, Information Technology Center, Carnegie-Mellon University, Pittsburgh, PA 15213.

Greenberg, S. (1984). *User modeling in interactive computer systems.* MSc Thesis, Department of Computer Science, University of Calgary.

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

Hansen, W. J. (1971). User engineering principles for interactive systems. *Proceedings of the Fall Joint Computer Conference.* AFIPS Press, New Jersey.

Harris, L. R. (1977). User oriented data base query with the ROBOT natural language query system. *Int. J. Man–Machine Studies.* **9**, 697–713.

Hendrix, G. G., Sacerdoti, E. D., Sagalowicz, D., and Slocum, J. (1978). Developing a natural language interface to complex data. *ACM Trans. on Database Systems.* **3** (2).

Innocent, P. R. (1982). Towards self-adaptive interface systems. *Int. J. Man–Machine Studies.* **16** (3), 287–99.

Joy, W. (1979). *An introduction to the C shell.* Computer Science Division Report, University of California, Berkeley, California.

Kernighan, B. W., and Mashey, J. R. (1981). The UNIX programming environment. *Computer*, **14** (4). 25–34.

Lakoff, G., and Johnson, M. (1980). *Metaphors we live by.* University of Chicago Press, Chicago.

Latremouille, S, and Lee, E. (1981). The design of videotex tree indexes: the use of descriptors and the enhancement of single index pages. *Telidon Behavioural Research.* **2**, Department of Communications, May.

Maguire, M. (1982). An evaluation of published recommendations on the design of man-computer dialogues. *Int. J. Man–Machine Studies.* **16** (3), 237–61.

Malcolm, M., and Dyment, D. (1983). Experience designing the Waterloo Port user interface. *Proc. ACM Conference on Personal and Small Computers.* 168–75. San Diego, California, December.

Mantei, M. (1982). *Disorientation behavior in person-computer interactions.* PhD thesis, University of Southern California, Los Angeles.

Martin, J. (1973). *Design of man–computer dialogues.* Prentice-Hall, Englewood Cliffs, New Jersey.

Martin, J. (1984). *Application development without programmers.* Prentice-Hall, Englewood Cliffs, New Jersey.

Martin, T. (1980). *Information retrieval.* In *Human interaction with computers* (eds Smith, H. T., and Green, T. R. G.) pp. 161–75. Academic Press, London.

Nakatani, L. H., and Rohrlich, J. A. (1983). Soft machines: A philosophy of user-computer interface design. *Proccedings of Human Factors in Computer Systems.* Boston, Mass., December 12–15.

Norman, D. A. (1981). The trouble about Unix. *Datamation.* **27** (12), 139–50.

Perlman, G. (1984). Natural artificial languages: low-level processes. *Int. J. Man–Machine Studies.* **20** (4), 373–419.

Pike, R. (1983). Graphics in overlapping bitmap layers. *ACM Trans. Graphics.* **2** (2), 135–160.

Rich, E. (1983). Users are individuals: individualizing user models. *Int. J. Man–Machine Studies.* **18** (3), 199–214.

Ritchie, D. M., and Thompson, K. (1974). The UNIX time-sharing system. *Communications of the Association for Computing Machinery.* **17** (7), 365–75.

Rosenthal, D. S. H. (1982). Managing graphical resources. *Computer Graphics.* **16** (4), 38–45.

Shneiderman, B. (1979). Human factors experiments in designing interactive systems. *Computer.* **12**(12), 9–19. December.

Shneiderman, B. (1980). *Software psychology.* Winthrop, Massachusetts.

Shneiderman, B. (1983). Direct manipulation: a step beyond programming languages. *Computer.* **16** (8), 57–69.

Smith, D. C. (1975). *Pygmalion: A computer program to model and stimulate creative thought.* PhD thesis, Stanford University.

Smith, D. C., Irby, C., Kimball, R., Verplank, B., and Harslem, E. (1982). Designing the Star user interface. *Byte*, **7** (4) 242–82.

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** (6), 147–55. Portland, Oregon, June 8–10.

Taylor, D. W. (1981). *Should a software interface adapt its behavior to the developing expertise of its users?* PhD thesis, University of Houston.

Teitelman, W. (1979). A display oriented programmer's assistant. *Int. J. Man–Machine Studies.* **1** (2), 157–87.

Test, J. A. (1982). *The NUnix window system.* Internal report, Laboratory for Computer Science, MIT, Cambridge, Massachusetts.

Thimbleby, H. (1980). Dialogue determination. *Int. J. Man–Machine Studies.* **13** (3), 295–304.

Thomas, J. C. (1978). A design-interpretation analysis of natural English with applications to man–computer interaction. *Int. J. Man–Machine Studies.* **10**, 651–68.

Thomas, J. C., and Carroll, J. M. (1981). Human factors in communication. *IBM System Journal.* **20** (2).

Waltz, D. L. (1975). Natural language access to a large data base. *Advance Papers of the International Joint Conference on Artificial Intelligence.* MIT, Cambridge, Mass.

Weizenbaum, J. (1976). *Computer power and human reason.* Freeman, San Francisco.

Weyer, S. A. (1982). *Searching for information in a dynamic book.* PhD Thesis, School of Education, Stanford University (also Report SCG-82-1, Xerox Parc).

Whalen, T., and Latremouille, S. (1981). The effectiveness of a tree-structured index when the existence of information is uncertain. *Telidon Behavioural Research.* **2**, Department of Communications, May.

Whalen, T., and Mason, C. (1981). The use of tree-structured index which contains three types of design defects. *Telidon Behavioural Research.* **2**, Department of Communications, May.

Wilkinson, W. (1980). Viewdata: The Prestel System. In *Videotext: the coming revolution in home/office information retrieval* (ed. E. Sigal), pp. 57–86. Harmony Books, New York.

Williams, G. (1982). Lotus Development Corporation's 1-2-3. *Byte.* **7** (12), 182–97.

Williams, G. (1983). The Lisa computer system. *Byte*, **8** (2).

Williams, G. (1984). The Apple Macintosh computer. *Byte*, **9** (2), 30–54.

Winograd, T. (1972). *Understanding natural language.* Academic Press, New York.

Witten, I. H. (1981). Programming by example for the casual user: a case study. *Proc. Canadian Man–Computer Communication Conference.* 105–113. Waterloo, Ontario, June.

Wood, R. J. (1982). *A window based display amangement system.* Internal Report, University of Maryland.

Woods, W. A. (1973). Progress in natural language understanding—an application to lunar geology. *Proc. National Computer Conference.* AFIPS Press, Montvale, N.J.

Zloof, M. M. (1977). Query-by-example: a data base language. *IBM Systems J.* **4**, 324–43.