To appear in:
Intelligent Interfaces: Theory, Research, and Design,
ed. M.H. Chignell, P.A. Hancock, and A. Loewenthal, Amsterdam: North-Holland, 1986.

Direct Manipulation in the Intelligent Interface

Robert J.K. Jacob

Naval Research Laboratory
Washington, D.C.

Direct Manipulation user interfaces were first identified by Ben Shneiderman [2]. The concept of direct manipulation crystallized a collection of principles and techniques shared by a number of innovative user interfaces that were widely regarded as easy -- and enjoyable -- to learn and use.

This chapter examines the nature of direct manipulation user interfaces and some issues associated with them. It distinguishes interfaces that operate directly on concrete visual objects from those that use visual metaphors to operate in more abstract domains. It then examines two direct manipulation systems in more detail. Some benefits and drawbacks characteristic of direct manipulation user interfaces are considered, and some empirical evidence of the importance of the choice of visual representation is provided. The problem of formally describing or specifying a direct manipulation interface is also investigated. It is concluded that, while the prospects for combining direct manipulation with "intelligent" user interfaces are promising, little research has been conducted in this area to date.

WHAT IS A DIRECT MANIPULATION USER INTERFACE?

Shneiderman identified the defining characteristics of direct manipulation user interfaces [2]:

- o Continuous representation of the object of interest.
- o Physical actions (movement and selection by mouse, joystick, touch screen, etc.) or labeled button presses instead of complex syntax.
- o Rapid, incremental, reversible operations whose impact on the object of interest is immediately visible.

- o Layered or spiral approach to learning that permits usage with minimal knowledge.

The essence of such a user interface is that the user seems to operate directly *on* the objects in the computer rather than carrying on a dialogue *about* them. Instead of using a command language to describe operations on objects, the user "manipulates" objects visible on a display. For example, to delete a file named *FOO* in a command language system, the user would ask the computer to *delete the file whose* name *is FOO.* In a direct manipulation system, he or she would find a representation of the file on the screen and "delete" it directly (perhaps with a delete button, a mouse gesture, or the like). The effect would be apparent immediately on the display. Further, the user could apply the same delete command to any other object that he sees on the display (provided the system permits its deletion); he does not have to learn new commands for them.

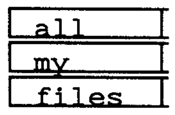
Another important characteristic of direct manipulation user interfaces is the complementarity of input and output. Whenever the user requests output, the objects shown in the resulting display on the screen are acceptable inputs to subsequent commands. Output is thus not a fixed, passive display, but a collection of dynamic, manipulable objects. A typical user command or input is synthesized from objects already on the screen, the outputs of previous commands. For example, if one had displayed a directory with a traditional system:

```
% ls
all
my
files
%
```

he would normally then delete a particular file by re-entering its *name:*

```
% rm my
```

In contrast, the comparable output of a direct manipulation directory listing command, e.g.:



would be "active" or manipulable. To delete the second file, the user would point to it in the output display and then give the delete command.

Hutchins, Hollan, and Norman [15] carefully examine the cognitive factors that underlie direct manipulation user interfaces and divide them into two separate areas. They find two different ways in which such user interfaces are "direct." One is *direct engagement,* the sense of manipulating objects directly on a screen rather than conversing *about* them. "There is a feeling of involvement directly with a world of objects rather than of communicating with an intermediary. The interactions are much like interacting with objects in the physical world. Actions apply to the objects, observations are made directly upon those objects, and the interface and the computer become invisible." [15]

The other form of directness is found in a reduction of *cognitive distance,* the mental effort needed to translate from the input actions and output representations to the operations and objects in the problem domain itself. Using a display screen, the visual images chosen to depict the objects of the problem or application domain should be easy for the user to translate to and from that domain. Conversely, for input, the actions required to effect a command should be closely related to the meaning of the command in the problem domain. The transparency of these representations, or reduction of cognitive distance, is thus a second form of directness in the direct manipulation interface. The problem of choosing an appropriate representation for the objects and actions of the problem domain is crucial to the effective use of the direct manipulation approach.

The most visible characteristic of a direct manipulation user interface is, then, direct engagement -- the ability to manipulate displayed objects. A direct manipulation user interface typically comprises a set of objects presented on a display and a standard repertoire of manipulations that can be performed on them. There is no command language for the user to remember beyond the set of manipulations, and generally any of them can be applied to any visible object. The displayed objects are active in the sense that they are affected by each command issued; they are not the fixed outputs of one execution of a command, frozen in time. They are also usable as inputs to subsequent commands. However, the ultimate success of a direct manipulation interface requires more than this manipulability; it hinges on the choice of a good metaphor for representing the world of the application in terms of screen objects and input actions.

EXAMPLES OF DIRECT MANIPULATION INTERFACES

The problem domains with which direct manipulation user interfaces must deal can be divided into two classes. In the first, the underlying problem is concerned with static, concrete objects; examples include menus, screen layouts, printed forms, engineering drawings, typeset reports, and fonts of type. A direct manipulation user interface for operating in such a domain can simply use a picture of the concrete object in a "what you see is what you get" style of editor. The choice of representation is straightforward, since there is already an agreed-upon concrete visual form for the objects of the problem area. Unfortunately, this approach is only possible where there can be a one-to-one correspondence between the problem domain and the visual representation.

A more difficult problem arises in the second class of direct manipulation user interface. Here, the domain involves abstract objects, which do not have a direct graphical image, such as time sequence, hierarchy, conditional statements, frame-based knowledge, or data in a data base. To provide a direct manipulation interface for such a domain, it is necessary first to devise a suitable graphical representation or visual metaphor for the objects. "What you see is what you get" is still helpful, but not sufficient to solve this problem, since the objects are abstract.

Interfaces for Concrete Objects

Examples of direct manipulation user interfaces for problem domains in the first class, where a concrete representation is available, often involve systems for creating and manipulating graphical images. A screen editor is a good example [31]. A more elaborate example can be found in a typical direct manipulation editor for a typesetting system, which displays a mockup of the final printed page on a high-resolution display [16]. It permits the user to manipulate the displayed page and, as he or she does, it immediately redisplays the portions of the page affected by the change. Since the system is designed for producing printed pages, the visual representation chosen for the direct manipulation interface is simply a high-resolution picture of such a page. Other examples of this class of direct manipulation user interface include:

- o Font editors, where the visual representation is a picture of the character being edited.
- o Computer-aided design or drafting systems based on standard engineering drawings, where the problem domain is that of paper drawings and the visual representation is a screen image of (portions of) the same drawings. (Note however that a modern computer-aided manufacturing system might not have a paper drawing as its final goal. It could, for example, drive a machine tool directly. The problem domain then becomes the operation of the tool or the creation of three-dimensional objects. As with interfaces in the second class, discussed below, the drawing may still serve as a convenient visual representation for the tool operation, but it is no longer the compelling choice it is for a simple drafting system.)
- o Systems for designing printed forms, where, again, the visual representation is simply a screen picture of the form.

The objects represented in these direct manipulation interfaces are concrete physical objects, and most of the manipulations available to the user are based on common physically-understandable actions such as moving, copying, changing size, or removing. These systems therefore generally have small cognitive distances between the problem domain and the objects and operations of the user interface.

Interfaces for Abstract Domains

More difficult problems arise where the problem domain is abstract. It is helpful if a reasonable concrete representation for the abstract problem domain is already in use, perhaps in pencil-and-paper form, and it can be exploited by the user interface designer. For example:

- o A system for manipulating a geographic data base might use choropleth ("patch") maps as its visual representation for the data and allow the user to manipulate the maps to view the data [4].

- A computer-aided manufacturing system that drives machine tools directly can use conventional mechanical drawings as its visual representation, as described above.
- Similarly, consider a system for handling forms-based data in a computer (i.e., a system that stores and retrieves forms-like computer data, without necessarily creating any printed forms). An hypothetical printed form might be used as a convenient and widely-understood visual representation of the internal computer data [32]; but it is not the only possible representation.
- A matrix or spreadsheet calculator. The visual metaphor widely chosen for such systems is the accountant's paper worksheet, with its rows and columns of figures. It is a concrete visual representation of an abstract domain (matrices). The rapid acceptance of direct manipulation spreadsheet systems shows that the paper spreadsheet was a particularly fortuitous choice of visual metaphor for this abstract object.
- STEAMER [14] allows its user to operate a simulated steam engine. The problem domain is the sequence of operations for running a steam engine, and the visual representation is an image of an engine control panel, with some extensions not available on conventional panels.
- The Xerox Star desktop manager [30] and its numerous philosophical descendants, such as Apple Macintosh, handle a domain of computer files and directories. For their visual metaphor they use a stylized picture of a desk surface with papers, file folders, trays, and even a waste basket to represent the computer file domain. A familiar operation, like moving a paper from a file folder to the waste basket, corresponds in an intuitively plausible way to deleting an item from a data file.
- Query-by-example [35] deals with a problem domain of obtaining information from a computer data base, a task typically accomplished through a relatively abstract query language. Instead, it uses an image of a printed report containing specific information from the data base as its metaphor for the data base domain. The user manipulates the image of the report (without real data) until it looks like the report he wants; then the system produces a similar report with actual data.

- Finally the system for designing user interfaces described later in this chapter has as its problem domain the time sequence or syntax of input and output operations performed by a system and its user. The visual metaphor chosen here is the state transition diagram, already used to represent user interface designs on paper. The designer manipulates a picture of a state transition diagram to describe and effect changes in the behavior of the user interface being constructed.

In each of these systems, an abstract problem domain was represented by a concrete visual metaphor. The visual metaphors used in these examples were all existing concrete objects, rather than newly-invented representations. Each representation was chosen to minimize cognitive distance between problem and representation, so that when the user manipulates the objects in the representation, his operations are closely allied to those of the problem domain. The most successful direct manipulation user interfaces thus far have depended on a fortuitous or perspicacious choice of visual metaphor.

It is also possible to design a direct manipulation interface by inventing a new visual object and teaching it to users as the representation of some abstract domain. Inventing good representations is a difficult problem, and there are relatively few examples of this class of direct manipulation system:

- The Spatial Data Management System [13] uses an hierarchical collection of icons arrayed in a map-like layout to depict data in a data base. The user examines the data base by panning across the layout or zooming up or down in the hierarchy. The visual metaphor for the data base data is thus a new pictorial representation. The original icons in the system were individually sketched; methods for generating graphics directly from the data were also studied [7].
- DMDOS is a user interface to the IBM PC-DOS operating system command language developed by Ben Shneiderman and Osamu Iseki at the University of Maryland. It uses a tabular representation of directories, disks, peripherals, and other objects of interest and permits the user to operate directly on the items shown in the display.

- o Hutchins, Hollan, and Norman [15] outline a design for a statistical analysis system in which sequences of mathematical operations on data are represented by a flow graph or circuit diagram, which the user can manipulate.

A Direct Manipulation Military Message System

The Secure Military Message System project at the Naval Research Laboratory is building a family of prototype military message systems [12]. Such a message system is much like a conventional electronic mail system, except that each message (actually, each field of each message), each file, and each user terminal has a security classification. The user can compose and send messages, display them, create files to hold them, move or copy messages among the files, delete or un-delete messages and files, and the like. All these operations must be performed within the constraints of the security rules. For example, a user is not permitted to store a *SECRET* message in an *UNCLASSIFIED* file.

The first set of prototypes built for this project had a user interface based on an abstract *intermediate command language*, combined with extensive use of menus [3]. It was designed principally for novice users. It led the user through a series of menus and prompts that helped him assemble a command and all of its arguments, which was then executed. Figures 1 through 3 show a portion of such a sequence, in which the user is going to display a message from one of his files.

The system has been easy to learn and self-documenting, but sometimes frustrating and indirect. For example, if the user displayed a set of his message summaries on the screen and then decided to display the full text of one of the messages, he would have to go through the full **Display** command sequence again to identify the message he wanted to see, even though he was already staring at a citation to it. It would seem more direct for him to point and say display *that* message than to refer to it by re-typing its full name. (In the actual system, constantly-updated default values for the command arguments mitigate this problem.)

An alternate version of the message system prototype was built to test this idea. It provides a new, direct manipulation user interface for the same underlying application. It was also designed to demonstrate the principle of *dialogue independence*[10] --

that, by proper modularization of the user interface code, application code can remain unchanged in the face of alternative user interfaces, provided they all translate user commands into the same form. Of particular interest here is the opportunity to compare two user interfaces to the identical underlying application system.

The basic visual metaphor chosen for the direct manipulation version of the message system is simple: a paper message. A message is represented by a screen image that is similar to a traditional paper military message. A new object, a file of messages, is also introduced. This is represented by a display of a list of the summaries (called *citations*) of the messages in the file. Some elements of each message citation in such a display can be changed directly by typing over them; they are indicated by borders around their labels. Other elements are fixed because the application requires it (e.g., the user cannot change the date of a message that has already been sent). In addition, each citation contains some *screen buttons*, or small, labeled boxes. Pressing a mouse button while pointing to such a box causes the action indicated in the box to occur. All the commands that the user could apply to a given message are shown on its citation as buttons. If the user sees a message on the screen, he does not need to refer to a manual to find out what commands he could apply to it.

Figure 4 shows a display from this system. The large window in the upper right contains a display of a message, and the other large window contains the list of message citations with their corresponding buttons and modifiable fields. The upper right window displays the message through a text editor, so that its contents can be manipulated with editing commands. If the message has already been sent, all editor commands except scrolling and searching are blocked. The lower window shows the citations of the messages in the user's file called **inbox**. It also contains an upper area with commands and fields that pertain to the file as a whole. Note that the second message in the file ("Subj: Important information") has been tentatively deleted (the user can still get it back by **undelete**-ing it). Most of the command buttons are not shown for that message because those commands are not permitted on tentatively deleted messages. The **Undelete** command is shown for that message but no others, because it can be applied only to deleted messages. Figure 5 shows some of the other windows and commands available in this system.

In comparing the two versions of the message systems, it is interesting to examine the number of arguments needed for each command (that is, the number of operations needed to invoke the command). In the first system, displaying a message required a series of menu picks or entries for: the **Display** command (Figure 1), the type of object to display (**Message-from-file**; Figure 2), the file name (Figure 3), and the number of the message within the file (not shown). The user is led through a sequence of menus and prompts to help him enter these data. For the duration of that sequence, he cannot do anything else (except abort the sequence). Considering the entire set of commands, there is thus a considerable number of distinct states or modes the system can be in, and the user must keep track of where he is at all times. In contrast, in designing the direct manipulation version, it became apparent that most of the explicit arguments to each command are used to identify a single object to which the command is to be applied. They can be avoided if the user is simply permitted to find the object on the screen (perhaps from some previous **Display** command) and point to it. For example, every message citation on the screen now contains a **Display** screen button. To display a message, the user simply chooses its **Display** button. The message is displayed in the message window, and the system immediately returns to the top-level command state, waiting for any new command. The user does not have to keep track of any state changes or remember "where he is" at any point in this command. Of course, to display a message whose citation is not on the screen, he would first have to display the appropriate file of citations and/or scroll to the right point in the citation display. But each of those operations is performed from the top-level command state and returns to that state immediately. The user does not have to go through a long, fixed sequence of states and remember his place in it; instead he performs a sequence of independent "rapid, incremental, reversible operations whose impact on the object of interest is immediately visible," [28] all within the same top-level state. The burden on the user's short-term memory is greatly reduced.

Similarly, to change the classification of a message, the user finds the message on the screen (either in a citation or the full message display) and types over the classification shown. Again, a command sequence involving several arguments and state changes (**Reclassify, Message-from-file,** *file-name, message-number, new-classification)* becomes a single-state command. Changing the classification of other displayed objects, including the terminal screen itself (seen in the top right corner of the display), is done analogously.

Of course, there are some commands that require additional arguments, beyond simply identifying one object. An example is the **Create message** command, which, for security checking purposes, requires that the classification of the new message be entered before the command is executed. (The file in which to place the new message is determined by the location of the selected **Create message** screen button.) When the button for this command is selected, a type-in area appears and the user is prompted to enter the new classification in it. Two new screen buttons, **OK** and **Abort** also appear. The first terminates the typed input and executes the command, while the second aborts the command. Both buttons and the type-in area disappear when the user selects either of the two buttons. This command thus involves one new state. The **Create message** command is selected from the top-level state, but then moves to a new state in which the system awaits the typed input. When the user selects the **OK** or **Abort** button, the system returns to the top-level state. Which of the two states the system is in is apparent from the presence or absence of the two buttons and type-in area.

In examining all the commands of the direct manipulation version of this message system, it was found that most of them involve no state changes or explicit arguments at all, as with the **Display** command described above. A minority of the commands involve one additional state, like **Create message** above; and only two of the commands (**Create text file** and **Create message file**) involve two states. In contrast, a typical command in the previous message system required four states in addition to the top level state from which the command itself was selected (e.g., **Reclassify, Message-from-file,** *file-name, message-number, new-classification).* An additional state was required for all commands not in the main menu (there were four additional sub-menus).

Thus, with the direct manipulation user interface, the system is nearly always in the same (top-level command) state and only occasionally in one of a small set of alternative states (waiting for a command argument). The advent of the alternative states is clearly indicated on the screen, and they are always one keystroke (the **OK** or **Abort** screen button) away from the top-level state. The user interface has a more direct feeling in that the user can perform almost any command at any time, rather than spending much of his time traversing long, fixed sequences from which few alternatives are possible. Of course there is a limit to this approach. If the system had hundreds of different commands, buttons for them would not all fit on the screen, and it would be necessary to introduce some sort of levels, states, or sub-dialogues. But the complexity of the corresponding non-direct manipulation interface would increase comparably, and it would continue to have still more levels or states.

A Direct Manipulation System for Designing User Interfaces

The next example involves an abstract domain: the design of user interfaces (not necessarily direct manipulation ones). That is, the design of the sequence of inputs and outputs, or syntax, of a user-computer dialogue. The visual representation chosen for this abstract domain is one already in use outside the realm of direct manipulation. The state transition diagram has been found to be a good -- pencil and paper -- representation of the user's view of the user interface of a computer system because of several of its properties:

- o In each state, the diagram makes explicit the interpretations of all possible user inputs.
- o It shows clearly what the user can do to change to another state (in which such interpretations would be different).
- o It emphasizes the temporal sequence of user and system actions in the dialogue.

State transition diagrams have been discussed and used for this purpose for several years [5,25] and have been found preferable to other languages for describing user interfaces [9,19]. They are thus an appropriate choice for use in a direct manipulation user interface as the concrete visual representation for the abstract notion of syntax.

The specific language chosen for use here is an extended version of state transition diagrams. It has been used to specify and directly implement user interfaces for several prototype systems [21,22], including the non-direct manipulation prototype message system described above; similar languages have also been used by others for this purpose [6,33]. The state transition diagram language used here is part of a methodology for designing and specifying user interfaces [20,21]; the direct manipulation version of it described below is currently being implemented [23]. The language is based on the conventional graphical diagrams used to describe finite state automata. A diagram in this language consists of a set of nodes (states) and links between them (state transitions).

Each state transition is associated with a token in the user input language. From any state, the next input token received causes the transition labeled with that token to occur. A transition may also be associated with an output token, which provides output to the user, or a processing action, which is performed by the system whenever that transition is taken.

An interactively editable picture of this state transition diagram is used as the visual representation of the user interface syntax. The programmer enters the state diagrams with a graphical editor and affixes the necessary labels and actions. He draws a separate diagram for each nonterminal, token, and lexeme, each in a separate window on the display screen so that the individual graphical objects being edited do not become too complicated. The diagrams are connected to each other by their names; each diagram is given a name, and it may be called by that name from a transition in another diagram. One diagram is designated the top-level diagram, and all the rest are called directly or indirectly by it.

As the diagrams are edited, they can be directly executed, since this state diagram notation is an executable language [21]. Thus the direct manipulation interface provides two types of windows. One shows a demonstration of the sequence being programmed, while the other allows the user to manipulate the visual representation of its syntax. As shown in Figure 6, the simulator window on the left shows the newly-designed user interface (of a simple line-oriented desk calculator program) as it would appear to the user and allows the programmer to interact with it in the role of its user. The programming windows on the right show the state diagrams themselves and allow the programmer to modify them as desired. During execution, the programmer can edit any of the diagrams with a graphical editor and thereby "manipulate" the syntax of the user interface through the state diagram representation while the system is running. He or she can also point to a state in a diagram and thereby cause the interpreter to move directly to that state.

Unix and Direct Manipulation

While not providing a direct manipulation user interface, the original Unix (trademark of ATT Bell Laboratories) operating system is worth noting here, because it took an essential first step in the direction of direct manipulation. It was one of the earliest systems to emphasize the complementarity of input and output. A basic tenet in the design of Unix programs was that the output of a program should be suitable as input to other programs. The system permitted a user to save the output of a program (in a file), edit it, and use it as input to another program or as a command script. It let him or her do this by providing redirection of all command input and output as an operating system primitive, rather than an option that could be provided or not by individual programmers. It encouraged him to do it by making the output of most commands simple and straightforward (sometimes at the risk of being cryptic and terse to human users) and thus usable for input without extensive processing. It further facilitated this by allowing outputs and inputs of command to be combined into pipes, saving the user from having to manipulate the intermediate files. For example:

```
% ls -l | grep Jan | sort
% date | tr '[a-z]' '[A-Z]'
% ls | grep '\.oo*' | sed 's/^/rm /' | sh
```

Unix still provided a command language-based user interface, not a direct manipulation interface, but it took a first step toward making output usable as input, which is now a cornerstone of the direct manipulation approach. The common medium chosen for interchange between program output and input was the unformatted text file or stream of characters, rather than any visual representation.

CHARACTERISTICS OF DIRECT MANIPULATION INTERFACES

Memory Load

The principal advantages of direct manipulation user interfaces are psychological: they decrease the demands on the user's short- and long-term memory. For long-term memory, they require remembering only a few generic commands in the form of general-purpose manipulations, which can be applied to most visible objects. Once the users memorizes this set, most specific operations can be derived from it, in contrast to traditional systems with many specific commands to remember.

Short-term memory load is reduced in two ways. First, most commands that change the values of objects are reflected immediately in changes in the display of those objects. The system thus continuously displays much of its internal data, rather than requiring the user to remember them and ask specific questions about them. A second benefit was exemplified by the message system. The direct manipulation version of the user interface led the user through fewer different states or modes. Long, sequential dialogues were reduced to one or two simple actions; commands with typically four mode changes were reduced to direct commands with no, or occasionally one or two, mode changes. Again, the user has less to keep in short-term memory about "where the system is" and fewer "places" (modes) it could be in. Having different modes permits different interpretations to be applied to the same user inputs. Reducing the number of possible modes the user must distinguish and remember reduces his work load.

Visual Representations

Direct manipulation user interfaces also introduce some difficulties. As seen above, they are easiest to apply to problem domains that have concrete graphical representations. For more abstract domains, inventing an appropriate visual metaphor to represent the objects in the domain is difficult; and using a poor representation hamstrings the resulting user interface. The representations currently in use have typically been designed for specific purposes and are not more widely applicable. Since direct manipulation user interfaces are new, there is not yet a sufficient basis in theory for developing graphical representations of abstract objects. A better understanding of visual perception is needed, which will permit a designer to devise natural graphical representations for a wide variety of objects, in a less ad hoc manner than presently.

Fixed Level of Abstraction

Another more subtle problem is that direct manipulation user interfaces, both for concrete and abstract objects, are more rigidly fixed at a single level of abstraction than command language systems. For example, consider a desktop manager like that of the Xerox Star [30]. It deals with an underlying file system at a particular -- though generally appropriate -- level of abstraction. Low-level details, such as allocating space for new files, reclaiming free space, determining disk layout, and reading and writing directories, are handled conveniently by the system and are invisible to its user. The user operates at the level of whole files and directories, not their component bits, tracks, or sectors. It is, however, difficult for the user to move further up smoothly in abstraction within the same language -- perhaps to operate on aggregates of files from different directories or to operate on commands about files. For example, the system makes it easy to move file *foo* to folder *bar* without worrying about lower-level details. But it would be difficult to move all files whose contents contain the string "Star" from one folder to another. The problem is that, while direct manipulation interfaces abstract away a host of irrelevant details up to a particular level, they remain stuck at that level; it is difficult to move up any further. This makes them convenient for users who happen to want to operate at the level provided but cumbersome at either higher or lower levels. By contrast, command language user interfaces encourage such abstraction because they provide a natural means to express it. Many command languages have good, almost intrinsic facilities for abstraction, control structures, formal parameters, combining individual commands, and the like. For example, with the Unix shell, the command to move a file

```
% mv /x/y/foo /bar
```

is less natural than pointing to and moving the image of the file; but it extends with less discontinuity to a higher level of abstraction to solve the problem posed above:

```
% mv  `grep -l Star /x/y/*`  /bar
```

If a direct manipulation interface is designed for the level of abstraction its user wants to use, he or she will indeed find it convenient. For both direct manipulation and command language user interfaces, that level can be as high or low as desired. The problem arises when the user wants to change levels smoothly. Command languages make it easier to move up or down in level; they encourage abstraction. Direct manipulation interfaces are often stuck at one level of abstraction, be it high or low.

One possible solution is to provide a facility to create, save, and name direct manipulation "programs" or scripts, as described below, and then provide a way to use such programs as objects in further higher-level manipulations [15]. But such a facility is typically absent from current direct manipulation interfaces and difficult to incorporate cleanly into their visual worlds. By contrast, it is usually quite easy and natural (or at least commonplace) to incorporate this facility into a command language.

Another related criticism is that direct manipulation suggests manual operations, the antithesis of automation [24]. A user presumably wants a machine to do work for him, not to do it himself "manually" on the screen. The unifying idea between these two approaches is the level of abstraction. If a direct manipulation system requires the user to perform *low*-level operations manually, it provides little advantage. But if it uses direct manipulation operations to let him select, in a convenient way, which *high*-level operations he wants the machine to perform, it can be helpful.

Command Procedures

Providing the user a method for creating and using scripts or command procedures is a more difficult problem in direct manipulation interfaces than traditional ones. The purpose is to record, save, and re-use a specific sequence of input operations or, more usefully, a generalized or parameterized sequence. With direct manipulation, the user's normal input language is inherently dynamic and ephemeral. The meaning of an action upon the screen depends on the particular object that lies under it at the time. The direct manipulation interface deals with things and not names; but it is easier to capture and operate on names in order to make a script. In recording a direct manipulation command procedure it is thus difficult to distinguish the formal parameters of the procedure from constants. For example, if the user points to object **x** in the top left of the screen, does he mean to record a procedure that always operates on whatever object appears in that position, always operates on object **x**, or to record a general sequence of operations, here performed on **x**, and apply it to other objects? Adequate graphical notation for this problem is lacking, but some intermediate approaches can be considered:

- The system can save and precisely replay a sequence of input actions. This is provided in some versions of the EMACS editor [31].
- It can save a sequence of input actions but provide a special notation to indicate dummy or formal parameter objects within that sequence. Query-by-example [35] illustrates this general idea.
- The system can have an underlying "intermediate" command language, into which all direct manipulation inputs are translated. This language is normally invisible to the user, but he or she could be permitted to write command procedures in it. This is a powerful approach, but it requires that the user switch to a second user interface language to write programs. He

cannot write even a simple command procedure without leaving the direct manipulation interface.

- A further refinement of the intermediate language approach is to provide a facility by which a sequence of direct manipulation input actions can be translated automatically to and saved in the intermediate command language as the user performs them. He can then examine the resulting transcript and edit it to produce the final command procedure. In particular, he can edit it to replace specific objects with appropriate parameter identifiers.

Programming Direct Manipulation Interfaces

Another drawback applies to the user interface programmer rather than the user. While direct manipulation can make a system easy to learn and to use, such a user interface is generally difficult to construct. Most existing examples have required considerable highly machine-specific, low-level programming. Higher-level abstractions for dealing with this new interaction technique are not yet available. Instead, the user interface is typically programmed in an ad-hoc way, making it difficult to modify or re-use. What is needed is a set of appropriate software engineering abstractions for direct manipulation user interfaces and a specification technique to describe them precisely in high-level terms. Such techniques are beginning to become available for traditional user interfaces, to describe the user-visible behavior of an interface without reference to implementation details [19,27,33]. Direct manipulation user interfaces have some important differences, and these must be understood before such techniques can be adapted to them. Work is beginning in this area, and some is discussed further below.

Conclusion

While the disadvantages cited above are more numerous, they are in essence technical problems, which are likely to yield to future research, particularly since direct manipulation interfaces are relatively new. The advantages are more fundamental, rooted in users' psychological characteristics, less likely to change, and thus decisive.

EFFECT OF CHOICE OF REPRESENTATION ON TASK PERFORMANCE

In designing a direct manipulation user interface for an abstract problem domain, choosing an appropriate visual representation for the objects of the domain was seen to be critical to the success of the system. A poor choice makes the resulting direct manipulation interface difficult to learn and use, despite its appealing surface features. To investigate the significance of the choice of graphical representation on performance of a task, an experiment involving a clustering task in an abstract domain was undertaken [17,18].

Subjects were given a set of 50 points in a nine-dimensional space, which were to be organized into 5 groups. The points had originally been generated in 5 clusters, each normally distributed around a center point, named the prototype. The subject's task was to look at the 5 prototypes and then assign each of the 50 deviants to a cluster surrounding one of the prototypes. The correct answers were those that put deviants with the prototypes from which they had been generated and to which they were closest in Euclidean distance. The 55 data points to be presented to the subjects were represented using several alternative visual representations, and subjects performed the same task with each of the different representations. First, each nine-dimensional data point was represented as a matrix of nine digits, representing the (rounded) values of its nine coordinates. Figure 7 shows the prototypes (top row) and some examples of their deviants (succeeding rows), with each point represented as a digit matrix. Second, each point was represented by a polygon, produced by making the lengths of nine equally-spaced radii each proportional to one of the nine data coordinates and then connecting the ends of the radii to form the outline of a nonagon. Only the outline was then retained. Figure 8 shows the same data as Figure 7, but represented by the polygons. Finally, each point was represented by a Chernoff face [2]. This is a cartoon face, in which variation in each of the nine coordinates of the data is represented by variation in one characteristic of some feature of the face image. For example, a component of the data might be represented by the length of the nose or the curvature of the mouth. The overall value of one nine-dimensional point is then represented by a single face. Figure 9 shows the same data represented in this fashion.

Results were computed by tabulating the number of errors each of the 24 subjects made in classifying the 50 points. Chance performance would give 40 errors out of 50. Figure 10 shows the mean number of errors they made and the mean time (in minutes) they took to sort the 50 points. The faces were found clearly to be superior to both the polygons and the digits at $p < 0.001$. No significant difference was found between the polygons and digits. While the polygons were sorted as quickly as the faces, they were not sorted correctly. Although subjects were performing the identical clustering task, their performance differed markedly as they were given different visual representations for the task.

One reason the faces were thought to be superior is that people are highly skilled at the specialized task of recognizing and processing human faces. Another reason is that faces encourage their observer to synthesize the various elements of the graphical display into a single memorable expression, a coherent *gestalt.* Other common types of displays also contain variable elements and can thus be used for graphing multivariate data; but often such displays predispose toward a piecemeal, sequential mode of processing, which obscures the recognition of relationships among elements. Faces induce their observer to integrate the display elements into a meaningful whole. In any event, they were observed to be a particularly good representation for clustering multidimensional data, and the effect of choosing a good representation over a bad one for this task was found to be marked.

MODES IN THE USER INTERFACE

Modes or states refer to the varying interpretation of a user's input. In each different mode, a user interface may give different meanings to the same input operations. Some use of modes is necessary in most user interfaces, since there are generally not enough distinct brief input operations (e.g., single keystrokes) to map into all the commands of a system. A moded user interface requires that the user remember (or system remind him) of which mode it is in at any time, and he must remember the different commands or syntax rules applicable to each mode. Modeless systems do not require this; the system is always in the same mode, and inputs always have the same interpretation. Modern editors [31] and document processing systems [30] have attempted to reduce the number of modes the user must remember. The designers of the Xerox Star argue that this makes them significantly easier to use [30]. Some informal evidence is provided by a consumer testing comparison of "modeless" (Apple

Macintosh) and a moded (IBM-PC) system, which found the former easier to learn [1]. Formal comparison has thus far been less conclusive [26].

Direct manipulation user interfaces appear to be modeless. Many objects are visible on the screen; and at any time the user can apply any of a standard set of commands to any object. The system is thus always in the same "universal" or "top-level" mode. This is approximately true of some screen editors, but for most other direct manipulation systems, where the visual representation contains more than one type of component, this is a misleading view. It ignores the input operation of moving the cursor to the object of interest. A clearer view suggests that such a system has many distinct modes. Moving the cursor to point to a different object is the command to cause a mode change, because once it is moved, the range of acceptable inputs is reduced and the meaning of each of those inputs is determined. This is precisely the definition of a mode change. For example, moving the cursor to the **Display** screen button in the message system example should be viewed as putting the system into a mode where the meaning of the next mouse button click is determined (it displays that message) and the set of permissible inputs is circumscribed (for example, keyboard input could be illegal or ignored). Moving the cursor somewhere else would change that mode.

If direct manipulation user interfaces are not thus really modeless, why do they appear to have the psychological advantages over moded interfaces that are usually ascribed to modeless ones? The reason is that they make the mode so apparent and so easy to change that it ceases to be a stumbling block. The mode is always clearly visible (as the location of a cursor), and it has an obvious representation (simply the echo of the same cursor location just used to enter the mode change command), in contrast to some special flag or prompt. Thus, the input mode is always visible to the user. The direct manipulation approach makes the output display (cursor location to indicate mode) and the related input command (move cursor to change mode) operate through the same visual representation (cursor location). At all times, the user knows exactly how to change modes; he can never get stuck. It appears, then, that direct manipulation user interfaces are highly moded, but they are much easier to use than traditional moded interfaces because of the direct way in which the modes are displayed and manipulated.

FORMAL DESCRIPTION OF DIRECT MANIPULATION USER INTERFACES

It is useful to be able to write a precise specification of the user interface of a computer system before building it, because the interface designer can thereby describe and study a variety of possible user interfaces without actually having to code them. Such a specification should describe precisely the user-visible behavior of the interface, but should not constrain its implementation. A state transition diagram-based notation has proven an effective and powerful medium for formally specifying traditional user interfaces [19,20], but it is necessary to modify this approach to handle direct manipulation interfaces. State diagrams tend to emphasize the modes or states of a system and the sequence of transitions from one state to another. While direct manipulation user interfaces initially appear to be modeless and thus unsuited to this approach, they were seen in fact to have a particular, highly regular kind of moded structure. This structure can be exploited in devising a formal specification technique for direct manipulation interfaces.

The top level of a typical direct manipulation interface can be viewed as a large state diagram with one top-level state and a branch (containing a cursor motion input) leading from it to each mode. Each such branch continues through one or more additional states before returning to the top-level state. There is typically no crossover between these branches. The top-level state diagram is thus a large, regular, and relatively uninteresting diagram with one start state and a self-contained (no crossover) path to each mode and thence back to start state. It is essentially the same for any direct manipulation system and need not be specified for each new system.

The individual paths generally correspond to the manipulable objects on the screen. There may also be some remembered state information within each of these paths, which can be suspended when the cursor leaves that field and resumed when it re-enters. For example, if the user moves the cursor to a type-in field and types a few characters, moves it somewhere else and performs other operations, and then returns to the type-in field, the dialogue within that field would be resumed with the previously-entered characters intact. Similarly, if the user had begun an operation that prompted for and required him to enter some additional arguments, he could move to another screen area and do something else before returning to the first area and resuming entry of the arguments where he had left them. Each of the individual objects on the screen thus has a particular syntax or dialogue associated with it. Each such dialogue can be suspended (typically if the user moves the cursor away) and later resumed at the point from which it was suspended. The relationship between the individual dialogues or branches of the top-level diagram is that of coroutines.

Given this structure, a direct manipulation user interface is best described as a collection of *objects*[8], organized around the manipulable objects and the loci of remembered state in the dialogue. These objects will often coincide with screen regions or windows, but need not. A typical object might be a screen button, individual type-in field, scroll bar, or the like. Each such object will be specified separately, and then a standard executive will be defined for the outer dialogue loop. Thus, to describe a direct manipulation user interface, it will be necessary to:

1. define a collection of *interaction objects*;
2. specify their internal behaviors; and
3. provide a mechanism for combining them into a coordinated user interface.

Note that in devising a specification method the goal is not strictly ease of programming or compactness, but rather capturing the way the user sees the dialogue. The underlying claim is that the user indeed sees the direct manipulation dialogue as a collection of small, individual, suspendable, coroutine-like dialogues, joined by a straightforward executive.

1. How should the user interface be divided into individual objects? An *interaction object* will be the smallest unit with which the user conducts a meaningful, step-by-step dialogue, that is, one that has continuity or syntax. It can be viewed as the smallest unit in the user interface that has a state that is remembered when the dialogue associated with it is interrupted and resumed. In that respect, it is like a window, but in a direct manipulation user interface it is generally smaller -- a screen button, a single type-in field on a form, a command line area. It can also be viewed as the largest unit of the user interface over which disparate input events should be serialized into a single stream, rather than divided up and distributed to separate objects. Thus, an interaction object is a locus both of maintained state and of input serialization.

2. How should an input handler for each interaction object be specified? Observe that, at the level of individual objects, each such object conducts only a single-thread dialogue, with all inputs serialized and with a remembered state whenever the individual dialogue is interrupted by that of another interaction object. Thus a conventional single-thread state diagram is the appropriate representation for the dialogue associated with an individual interaction object. The input handler for each interaction object is specified as a simple state transition diagram.

3. How should the specifications of the individual objects be combined into an "outer loop" or overall direct manipulation user interface? As noted, a direct manipulation interface could be described with a single, large state diagram, but, since the user sees the structure of the user interface as a collection of many semi-independent objects, that is not a particularly perspicuous description. Instead, a built-in executive will be defined that embodies the basic structure of direct manipulation dialogue and includes the ability to make coroutine calls between individual state diagrams. This executive operates by collecting all of the state diagrams of the individual interaction objects and executing them as a collection of coroutines, assigning input events to them and arbitrating among them as they proceed. To do this, a coroutine call mechanism for activating state diagrams must be defined. This means that whenever a diagram is suspended by a coroutine call to another diagram, the state in the suspended diagram is remembered. Whenever a diagram is resumed by coroutine call, it will begin executing at the state from which it was last suspended. The executive causes the state diagram of exactly one of the interaction objects to be active at any one time. As the active diagram proceeds, it reaches each state, examines the next input event, and takes the appropriate transition from that state. It continues in this way until it reaches a state from which no outgoing transition matches the current input. Then, the executive takes over, suspending the current diagram but remembering its state for later resumption. The executive examines the diagrams associated with all the other interaction objects and looks at their current (i.e., last suspended from) states to see which of them can accept the current input. It then resumes (with a coroutine call) whichever diagram has a transition to accept the input. Typically there will be only one such diagram. In fact, since entering and exiting disjoint screen regions will be important input tokens in a typical direct manipulation interface, this is straightforward to arrange when the interaction objects correspond to screen regions. Depending on the overall system design, an input token acceptable to no diagrams could be discarded or treated as a syntax error.

An Example Specification

Figure 11 shows a specification of a single screen button as an individual interaction object using this approach and a simple Ada-based notation. This particular button is highlighted whenever the cursor is inside it. If the user presses the left mouse button while pointing to it, the message file **inbox** is displayed. An interaction object such as the one in Figure 11 is an object in the sense of Smalltalk [8] or Flavors [34]. It comprises a collection of variables, methods, and other impedimenta, most of which are subject to inheritance. Specifically, the specification of an interaction object can contain the following components:

FROM: A list of other interaction objects from which this one inherits elements, with ordering rules similar to those for Flavors.

IVARS: A list of the instance variables for this object and their initial values. These may also include other, lower-level interaction objects that will be used as component parts of this one.

METHODS: Procedure definitions unique to this object.

TOKENS: Definitions of each of the input and output tokens used in the syntax diagram for this interaction object. The tokens are the low-level input and output operations, which can be associated with transitions in the state diagrams. Examples for input are button clicks (both down and, where supported, up), cursor entering or exiting regions, and keyboard characters; for output, they include highlighting or dehighlighting regions, displaying or erasing graphical objects, and "rubber band" or other continuous "dragging" feedback. The internal details of these tokens would be specified here, separately from the state diagram that calls them. In the figure, they are given in English.

SYNTAX: The input handler for this interaction object, expressed as a conventional state transition diagram, which will be called by the executive as a coroutine. In the diagram, each state transition can have an input or output token, the name of another diagram to be called as a subroutine, or an action to be performed. Names of input tokens begin with **i**, and output tokens begin with

o. Further details of this notation are found in [20,21]. An action, such as **DisplayMf(inbox)** in the example, calls a procedure that is defined in the application (semantic) code, which is separate from the user interface. This diagram could also have been entered in a text form, rather than the graphical form shown in the figure.

SUBS: Additional state diagrams, called as subroutines by the syntax diagram above.

STATES: A list of "mixin" [34] or "kernel" [29] states, which are used to define standard sets of behaviors, such as sensitivity to abort or help keys, and which can be applied to states in the above diagrams, so that such descriptions do not have to be repeated for each state.

Discussion of Example

How does the syntax diagram given for this interaction object operate with the executive? When the cursor enters the screen area for this button, the locally-defined input token **iENTER** is generated. Since no other interaction object will have state transitions that accept **iENTER** as defined here (i.e., as the cursor entering this particular **position**), the diagram for this object will be called as a coroutine by the executive. This diagram will take over, accept the input, highlight the button, then wait for more input (in the state marked with a "+"). If the next input is the button press (**iLEFT**), this object performs its action. If the next input is the cursor exiting this region (**iEXIT**), this object dehighlights itself and returns to its start state. There it waits only for another **iENTER** and ignores other inputs. (In particular an **iLEFT** or other button click will no longer be accepted by this object but would probably be accepted by some other object.) Returning to the state marked "+", if the next input received in that state is anything other than **iLEFT** or **iEXIT** (for example, a keyboard key), another diagram that has a transition that can accept that input will be called by the executive. As soon as another input that this diagram can accept occurs, it will be resumed in the same state (the one marked "+").

Why does the state diagram look so complex for an operation that seems intuitively simple to describe? The reason is that there are several possible plausible alternative behaviors for the precise handling of sequences of clicks and mouse motions in a screen button. There are other ways in which the exiting and dehighlighting could be handled. Or, the screen button could be highlighted when the mouse button is depressed and perform the action when it is released. The user interface designer must be able to indicate exactly which of these possibilities he intends. It is not sufficient to have him describe the user interface imprecisely and leave the details up to a coder. Nor is it sufficient to supply one standard version of a screen button and prevent the designer from changing it. Given that the user interface designer must provide this precision, state transition diagrams are a reasonable notation for doing so.

Inheritance

The remaining problem with this notation is that the interaction object descriptions for a non-trivial direct manipulation system are going to become bulky, numerous, and repetitive. The solution is inheritance of the parts of the interaction objects. Specifically, an interaction object inherits all of the **IVARS, METHODS, TOKENS, SUBS,** and **STATES** of its parents and adds them to any that the object itself declares. If the object declares an **IVAR, METHOD, TOKEN, SUB,** or **STATE** of the same name, it overrides the inherited one. In turn, all of an object's own and inherited **IVARS, METHODS, TOKENS, SUBS,** and **STATES** are inherited by its children. The entire **SYNTAX** diagram is also inherited and may be overloaded as a unit; a notation for selectively overloading parts of it is under consideration.

Figure 12 shows the same object as Figure 11, but here the availability of a library of generic objects, from which components can be inherited, is assumed. The library object **GenericItem** defines a set of procedures and tokens that are applicable to a wide range of screen items. In particular, it defines the tokens **iLEFT**, **iENTER**, **iEXIT**, **oHIGHLIGHT**, and **oDEHIGHLIGHT**. The latter four are defined generically, in terms of an unspecified instance variable, **position**, which is to be supplied by the inheriting object. Like a "mixin" flavor, it is not expected that **GenericItem** will be instantiated by itself, but will contribute tokens, methods, and the like to other, more specific objects by inheritance. **GenericButton** defines a generic screen button, again as a mixin, not expected to be instantiated by itself. It defines the **Draw** procedure generically, in terms of an instance variable **legend** instead of a constant, and it provides an inheritable syntax diagram that describes a "standard" screen button. The action in the syntax diagram calls a procedure named **DoAction**, which each inheriting object can define in its own way. Given these primitives, the particular button defined above can now be written more compactly by inheriting the aspects that are common to all items and all screen buttons and defining only those specific to this particular button. The specification in Figure 12 defines the same object as that of Figure 11, taking advantage of the generics. It inherits the components of **GenericItem** and **GenericButton**. It defines only the instance variables **position** (which is used by the tokens in **GenericItem**) and **legend** (used by **Draw** in **GenericButton**) and the procedure **DoAction** (called by the syntax diagram in **GenericButton**). Everything else is inherited from the generics, including the syntax diagram itself from **GenericButton**. If a user interface designer did want this particular screen button to be different from the standard ones, he would simply overload those aspects of the generic objects that he wanted to change.

In the specification language introduced here, then, each locus of dialogue is clearly and appropriately described as a separate object with a single-thread state diagram, which can be suspended and resumed, but always retains state. The overall direct manipulation user interface is defined implicitly by the coroutine-based behavior of a standard executive, rather than inappropriately as a large, highly regular state transition diagram. Given a library of generic interaction objects and an inheritance mechanism, the collection of interaction object specifications need not become cumbersome or repetitive.

THE INTELLIGENT USER INTERFACE

An "intelligent" user interface should be able to describe and reason about what its user knows and conduct a dialogue with the long-term flow and other desirable properties of dialogues between people. It maintains and uses information about the user and his or her current state of attention and knowledge, the task being performed, and the tools available to perform it [11]. It can use this information to help interpret a user's inputs and permit them to be imprecise, vague, slightly incorrect (e.g., typographical errors) or elliptical. It can control the presentation of output based on its model of what the user already knows and is seeking and remove information irrelevant to his current focus. Knowledge-based techniques can also be applied to improve the selection, construction, and layout of graphical outputs.

Most research to date on the processes needed to conduct such "intelligent" dialogues has been applied to natural language, but it is important to remember that such techniques of the intelligent user interface are by no means restricted to natural language. There is no reason why such ideas could not be used in a direct manipulation dialogue. The user's side of such a dialogue can consist almost entirely of pointing and pressing mouse buttons, and the computer's, of animated pictorial analogues. Nevertheless, a dialogue in such a language could exhibit the intelligent user interface properties cited -- following focus, inferring goals, correcting misconceptions. This combination of such intelligent user interface techniques with a direct manipulation type of language holds promise for providing a highly effective form of man-machine communication.

CONCLUSIONS

This chapter has examined direct manipulation interfaces and divided them into two classes: those that operate in a domain of concrete visual objects and those that use such visual objects as proxies to operate in a more abstract problem domain. The importance of choosing a good visual representation for interfaces in the second class was discussed and demonstrated. Some advantages and disadvantages of direct manipulation were also considered. While more numerous, the disadvantages were seen to be largely technical and likely to yield to further research; while the advantages were based on fundamental characteristics of the users. Despite their surface appearance, direct manipulation interfaces were seen to have a peculiar, highly moded structure. The overall organization of a direct manipulation interface was found to be a collection of coroutines, and this observation formed the basis for a formal specification technique for such user interfaces. Finally, the future possibilities offered by combining "intelligence" with direct manipulation were seen to be promising.

ACKNOWLEDGMENTS

I want to thank Ben Shneiderman for introducing me to the idea of direct manipulation and helping me understand it and Don Norman for his ideas and comments on this work, which have helped to clarify the idea. Discussions with my colleagues on the WIS Command Language Task Force -- Phil Hayes, Ken Holmes, Joe Hrycyszyn, Tom Kaczmarek, Jon Meads, and Brad Myers -- helped me to define the specification language for direct manipulation interfaces. Finally, I want to thank Carl Landwehr for facilitating and encouraging this research.

Much of this work was supported by the Space and Naval Warfare Systems Command under the direction of H.O. Lubbes. Other portions were supported by a contract between the Johns Hopkins University and the Engineering Psychology Programs, Office of Naval Research, under the direction of John J. O'Hare.

REFERENCES

1. Consumers Union, "Computers: Apple Macintosh," *Consumer Reports* **50**(1) pp. 28-31 (1985).
2. H. Chernoff, "The Use of Faces to Represent Points in n-Dimensional Space Graphically," *Journal of the American Statistical Association* **68** pp. 361-368 (1973).
3. M.R. Cornwell and R.J.K. Jacob, "Structure of a Rapid Prototype Secure Military Message System," *Seventh DOD/NBS Computer Security Conference* pp. 48-57, Gaithersburg, Md. (1984).
4. J. Dalton, J. Billingsley, J. Quann, and P. Bracken, "Interactive Color Map Displays of Domestic Information," *Computer Graphics* **13**(2) pp. 226-233, Chicago (1979).
5. D.C. Engelbart and W.K. English, "A Research Center for Augmenting Human Intellect," *Proc. 1968 Fall Joint Computer Conference* pp. 395-410, AFIPS (1968).
6. M.B. Feldman and G.T. Rogers, "Toward the Design and Development of Style-independent Interactive Systems," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 111-116 (1982).

7. M. Friedell, "Automatic Synthesis of Graphical Object Descriptions," *Computer Graphics* **18**(3) pp. 53-62, Minneapolis (1984).

8. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Reading, Mass. (1983).

9. S.P. Guest, "The Use of Software Tools for Dialogue Design," *International Journal of Man-Machine Studies* **16** pp. 263-285 (1982).

10. H.R. Hartson and D.H. Johnson, "Dialogue Management: New Concepts in Human-computer Interface Development," *Computing Surveys* (1986). in press

11. P. Hayes, E. Ball, and R. Reddy, "Breaking the Man-Machine Communication Barrier," *IEEE Computer* **14**(3) pp. 19-30 (1981).

12. C.L. Heitmeyer, C.E. Landwehr, and M.R. Cornwell, "The Use of Quick Prototypes in the Military Message Systems Project," *ACM SIGSOFT Software Engineering Notes* **7** pp. 85-87 (1982).

13. C.F. Herot, R. Carling, M. Friedell, and D. Kramlich, "A Prototype Spatial Data Management System," *Computer Graphics* **14**(3) pp. 63-70 (1980).

14. J.D. Hollan, E.L. Hutchins, and L. Weitzman, "STEAMER: An Interactive Inspectable Simulation-Based Training System," *The AI Magazine* **5**(2) pp. 15-27 (1984).

15. E.L. Hutchins, J.D. Hollan, and D.A. Norman, "Direct Manipulation Interfaces," in *User Centered System Design: New Perspectives in Human-computer Interaction*, ed. D.A. Norman and S.W. Draper, Lawrence Erlbaum, Hillsdale, N.J. (1986). in press.

16. Interleaf, Inc., "Workstation Publishing Software," , Cambridge, Mass. (1984).

17. R.J.K. Jacob, H.E. Egeth, and W. Bevan, "The Face as a Data Display," *Human Factors* **18** pp. 189-199 (1976).

18. R.J.K. Jacob, "Facial Representation of Multivariate Data," pp. 143-168 in *Graphical Representation of Multivariate Data*, ed. P.C.C. Wang, Academic Press, New York (1978).

19. R.J.K. Jacob, "Using Formal Specifications in the Design of a Human-Computer Interface," *Comm. ACM* **26** pp. 259-264 (1983).

20. R.J.K. Jacob, "Executable Specifications for a Human-Computer Interface," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 28-34 (1983).

21. R.J.K. Jacob, "An Executable Specification Technique for Describing Human-Computer Interaction," pp. 211-242 in *Advances in Human-Computer Interaction,* ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1985).

22. R.J.K. Jacob, "Designing a Human-Computer Interface with Software Specification Techniques," pp. 139-156 in *Empirical Foundations of Information and Software Science,* ed. J.C. Agrawal and P. Zunde, Plenum Press, New York (1985).

23. R.J.K. Jacob, "A State Transition Diagram Language for Visual Programming," *IEEE Computer* **18**(8) pp. 51-59 (1985).

24. L.H. Nakatani and J.A. Rohrlich, "Soft Machines: A Philosophy of User-computer Interface Design," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 19-23 (1983).

25. D.L. Parnas, "On the Use of Transition Diagrams in the Design of a User Interface for an Interactive Computer System," *Proc. 24th National ACM Conference* pp. 379-385 (1969).

26. M.F. Poller and S.K. Garter, "A Comparative Study of Moded and Modeless Text Editing by Experienced Editor Users," *Proc. ACM SIGCHI Human Factors in Computer Systems Conference* pp. 166-170 (1983).

27. B. Shneiderman, "Multi-party Grammars and Related Features for Defining Interactive Systems," *IEEE Transactions on Systems, Man, and Cybernetics* **SMC-12** pp. 148-154 (1982).

28. B. Shneiderman, "Direct Manipulation: A Step Beyond Programming Languages," *IEEE Computer* **16**(8) pp. 57-69 (1983).

29. J.L. Sibert and W.D. Hurley, "A Prototype for a General User Interface Management System," Technical Report GWU-IIST-84-47, Institute for Information Science and Technology, George Washington University (1984).

30. D.C. Smith and others, "Designing the Star User Interface," *Byte* **7**(4) pp. 242-282 (1982).

31. R.M. Stallman, "EMACS: The Extensible, Customizable, Self-documenting Display Editor," MIT Artificial Intelligence Laboratory, Cambridge, Mass. (1979).

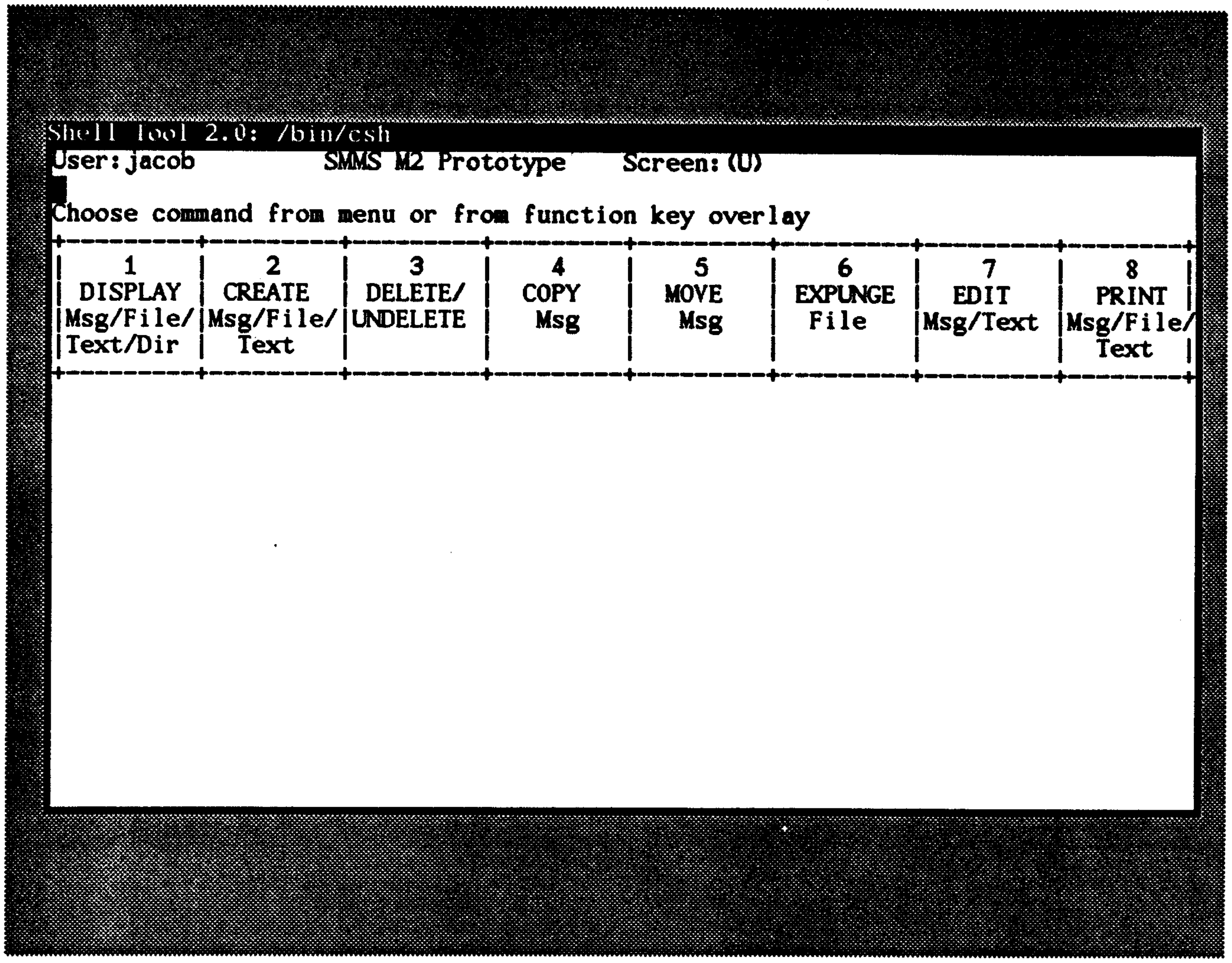
32. D. Tsichritzis, "Form Management," *Comm. ACM* **25** pp. 453-478 (1982).

33. A.I. Wasserman and D.T. Shewmake, "The Role of Prototypes in the User Software Engineering (USE) Methodology," pp. 191-209 in *Advances in Human-Computer Interaction*, ed. H.R. Hartson, Ablex Publishing Co., Norwood, N.J. (1985).

34. D. Weinreb and D. Moon, "Lisp Machine Manual," MIT Artificial Intelligence Laboratory, Cambridge, Mass. (1981).

35. M.M. Zloof, "Query by Example," *Proc. 1975 National Computer Conference* **44** pp. 431-438, AFIPS (1975).

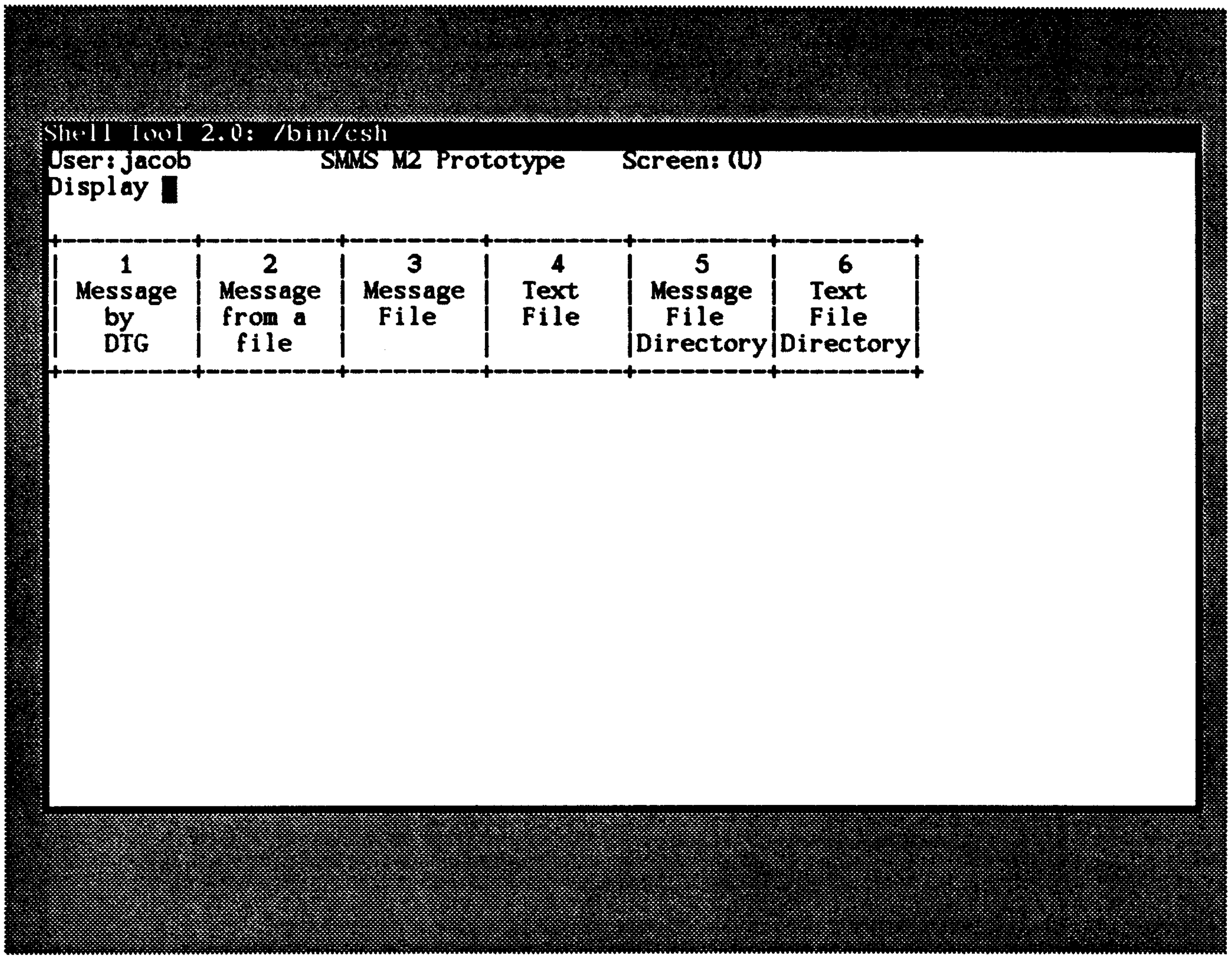
Figure 1. Menu-based military message system prototype -- main menu.



```
Shell Tool 2.0: /bin/csh
User:jacob             SMMS M2 Prototype     Screen:(U)

Choose command from menu or from function key overlay
+---------+---------+---------+---------+---------+---------+---------+---------+
|    1    |    2    |    3    |    4    |    5    |    6    |    7    |    8    |
| DISPLAY | CREATE  | DELETE/ |  COPY   |  MOVE   | EXPUNGE |  EDIT   |  PRINT  |
|Msg/File/|Msg/File/|UNDELETE |   Msg   |   Msg   |  File   |Msg/Text |Msg/File/|
|Text/Dir |  Text   |         |         |         |         |         |  Text   |
+---------+---------+---------+---------+---------+---------+---------+---------+
```

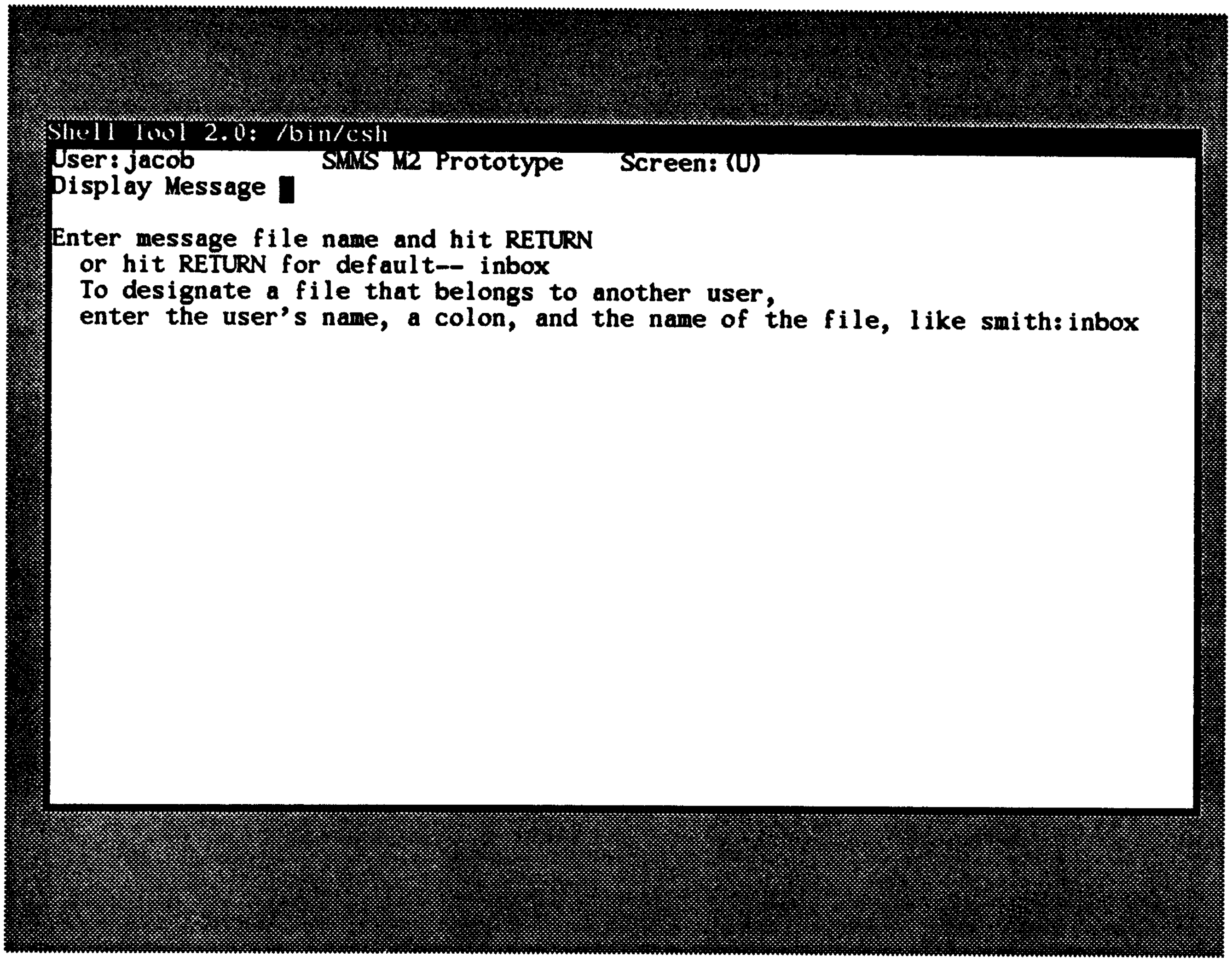
Figure 2. Menu-based military message system prototype -- "Display" command.



```
Shell Tool 2.0: /bin/csh
User:jacob          SMMS M2 Prototype     Screen:(U)
Display █

+---------+---------+---------+---------+---------+---------+
|    1    |    2    |    3    |    4    |    5    |    6    |
| Message | Message | Message |  Text   | Message |  Text   |
|   by    | from a  |  File   |  File   |  File   |  File   |
|   DTG   |  file   |         |         |Directory|Directory|
+---------+---------+---------+---------+---------+---------+
```

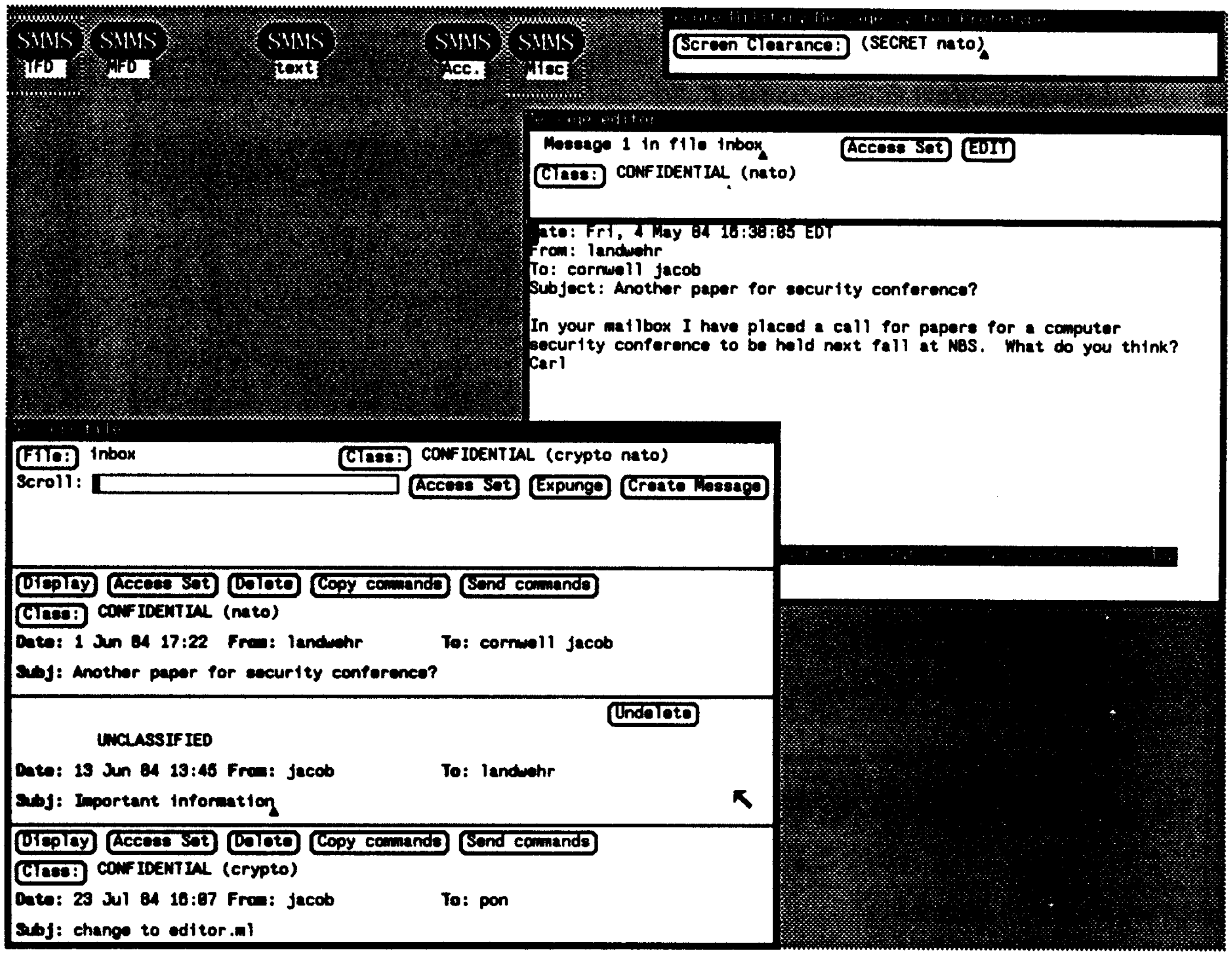
Figure 3. Menu-based military message system prototype -- "Display message from file" command.



```
Shell Tool 2.0: /bin/csh
User:jacob          SMMS M2 Prototype     Screen:(U)
Display Message █

Enter message file name and hit RETURN
  or hit RETURN for default-- inbox
  To designate a file that belongs to another user,
  enter the user's name, a colon, and the name of the file, like smith:inbox
```

Figure 4. Direct manipulation military message system prototype, showing message window (top) and message file window (bottom). The security classifications shown are simulated for demonstration purposes.



Screen Clearance: (SECRET nato)

Message 1 in file inbox Access Set EDIT
Class: CONFIDENTIAL (nato)

```
Date: Fri, 4 May 84 16:38:05 EDT
From: landwehr
To: cornwell jacob
Subject: Another paper for security conference?

In your mailbox I have placed a call for papers for a computer
security conference to be held next fall at NBS.  What do you think?
Carl
```

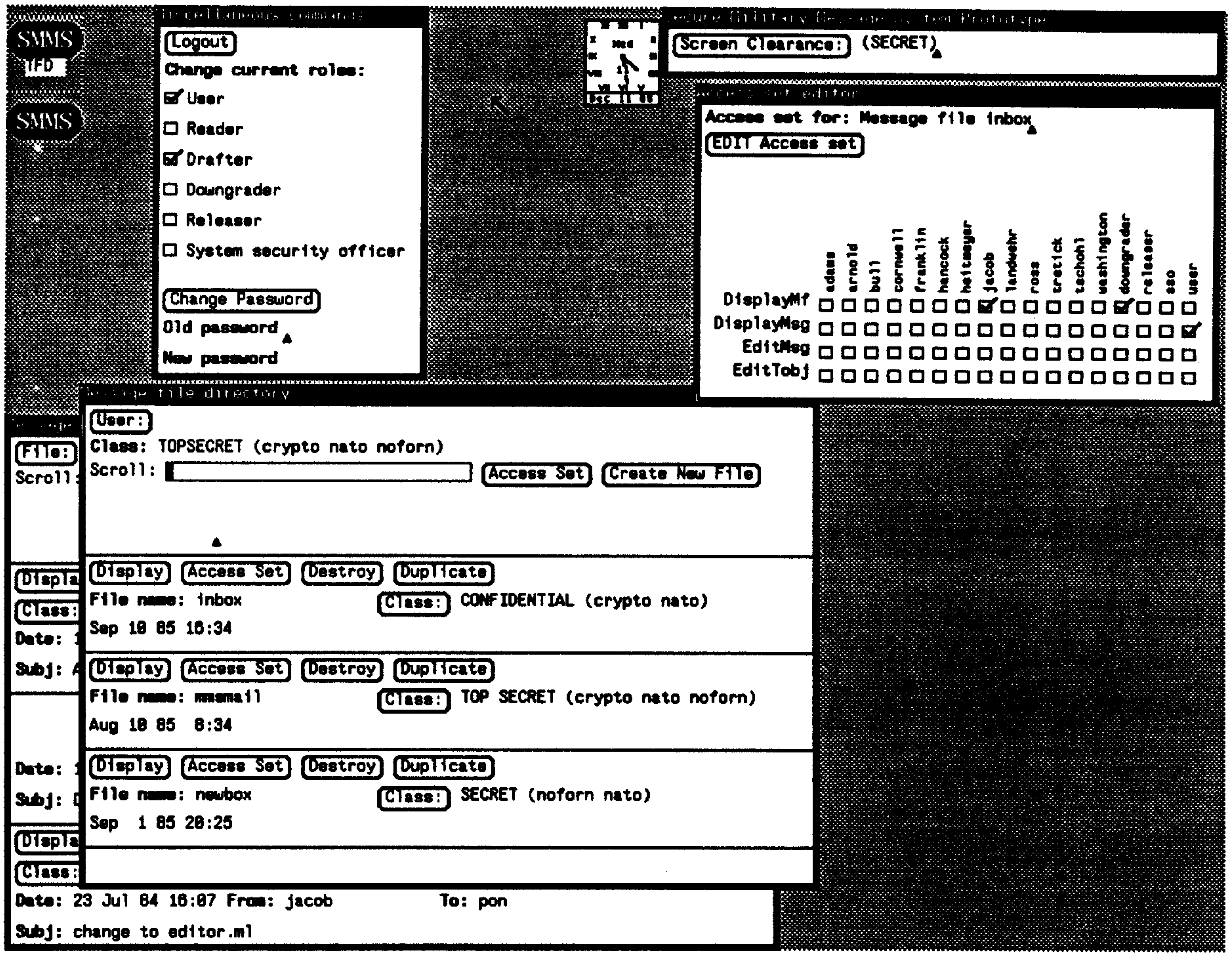
File: inbox Class: CONFIDENTIAL (crypto nato)
Scroll: Access Set Expunge Create Message

Display Access Set Delete Copy commands Send commands
Class: CONFIDENTIAL (nato)
Date: 1 Jun 84 17:22 From: landwehr To: cornwell jacob
Subj: Another paper for security conference?

Undelete
UNCLASSIFIED
Date: 13 Jun 84 13:45 From: jacob To: landwehr
Subj: Important information

Display Access Set Delete Copy commands Send commands
Class: CONFIDENTIAL (crypto)
Date: 23 Jul 84 16:07 From: jacob To: pon
Subj: change to editor.ml

Figure 5. Direct manipulation military message system prototype, showing user roles, access set editor, and message file directory windows.



Logout

Change current roles:

- [✓] User
- [ ] Reader
- [✓] Drafter
- [ ] Downgrader
- [ ] Releaser
- [ ] System security officer

Change Password

Old password

New password

Screen Clearance: (SECRET)

Access set for: Message file inbox

EDIT Access set

	adams	arnold	bull	cornwell	franklin	hancock	heitmeyer	jacob	landwehr	ross	tretick	tschohl	washington	downgrader	releaser	sso	user
DisplayMf	☐	☐	☐	☐	☐	☐	☐	✓	☐	☐	☐	☐	☐	✓	☐	☐	☐
DisplayMsg	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	✓
EditMsg	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐
EditTobj	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐	☐

User:
Class: TOPSECRET (crypto nato noforn)
Scroll: Access Set Create New File

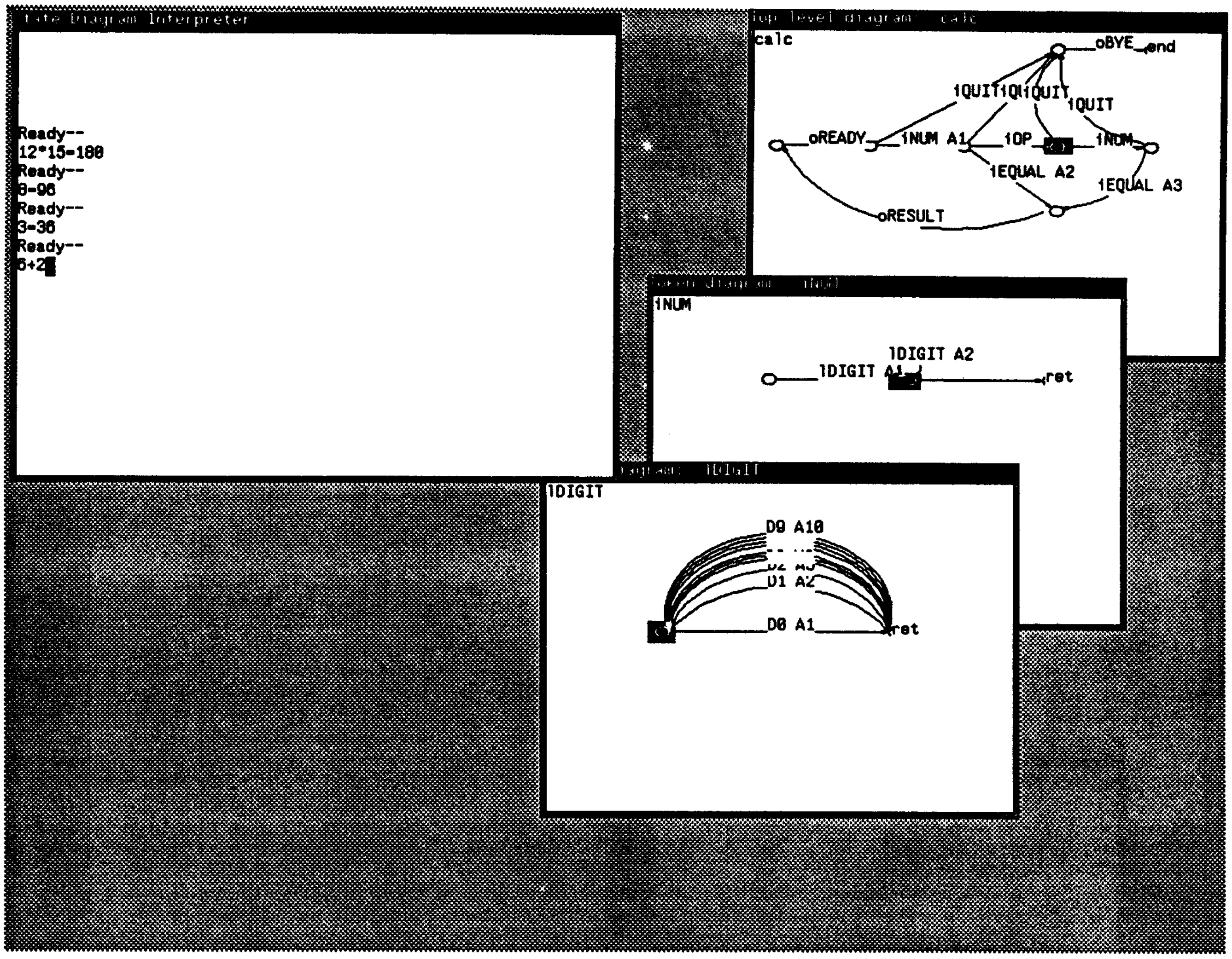
Display Access Set Destroy Duplicate
File name: inbox Class: CONFIDENTIAL (crypto nato)
Sep 10 85 16:34

Display Access Set Destroy Duplicate
File name: mmsmail Class: TOP SECRET (crypto nato noforn)
Aug 10 85 8:34

Display Access Set Destroy Duplicate
File name: newbox Class: SECRET (noforn nato)
Sep 1 85 20:25

File:
Scroll:
Displa
Class:
Date: 1
Subj: A
Date: 1
Subj: D
Displa
Class:
Date: 23 Jul 84 16:07 From: jacob To: pon
Subj: change to editor.ml

Figure 6. Direct manipulation system for designing user interfaces, showing state transition diagrams for the user interface being designed (at right) and the new user interface itself (at left).



```
Ready--
12*15=180
Ready--
8=96
Ready--
3=36
Ready--
6+2
```

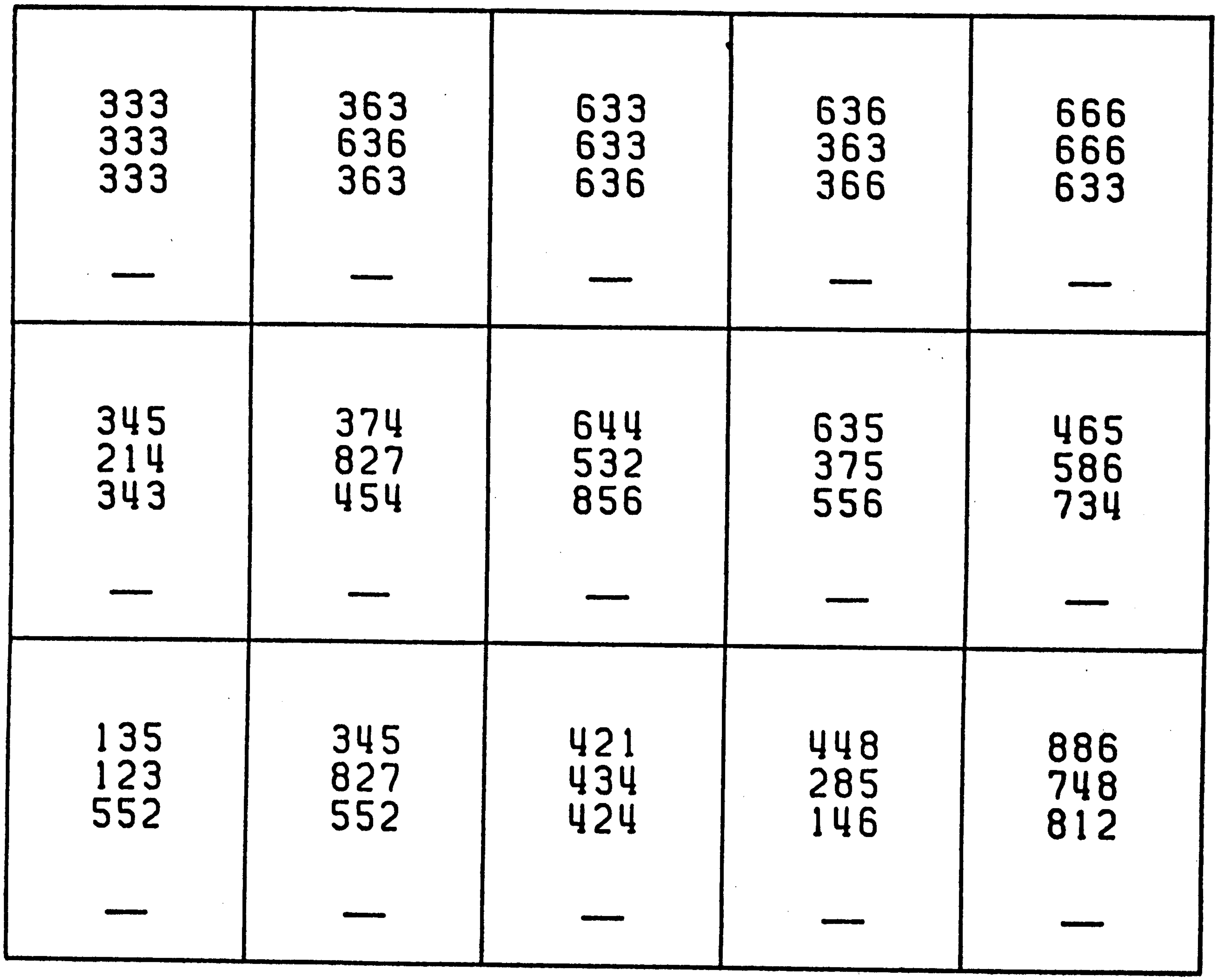


Figure 7. Examples of the nine-dimensional data points represented as digit matrices.

Figure 8. Examples of the nine-dimensional data points represented as polygons.

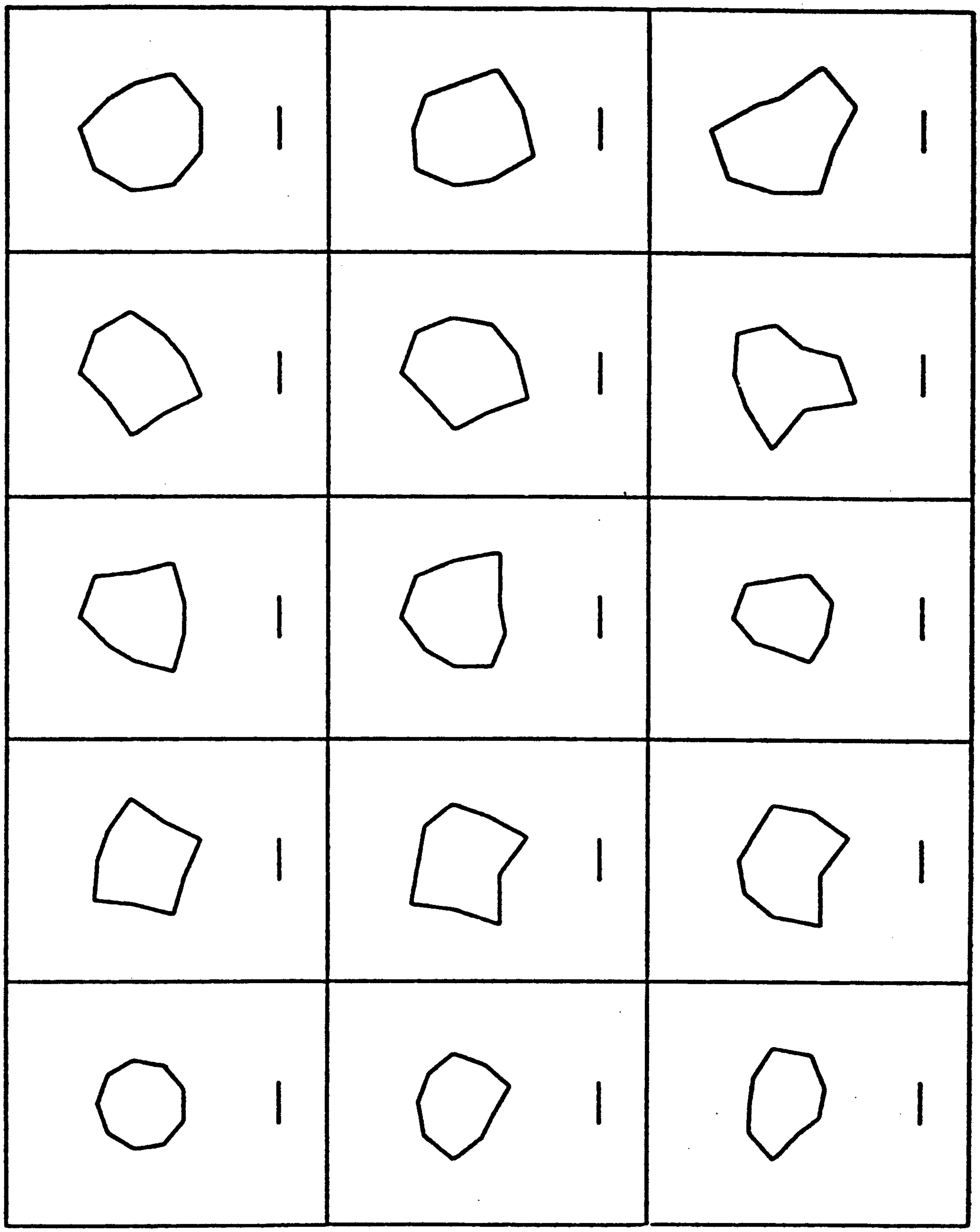


Figure 9. Examples of the nine-dimensional data points represented as Chernoff faces.

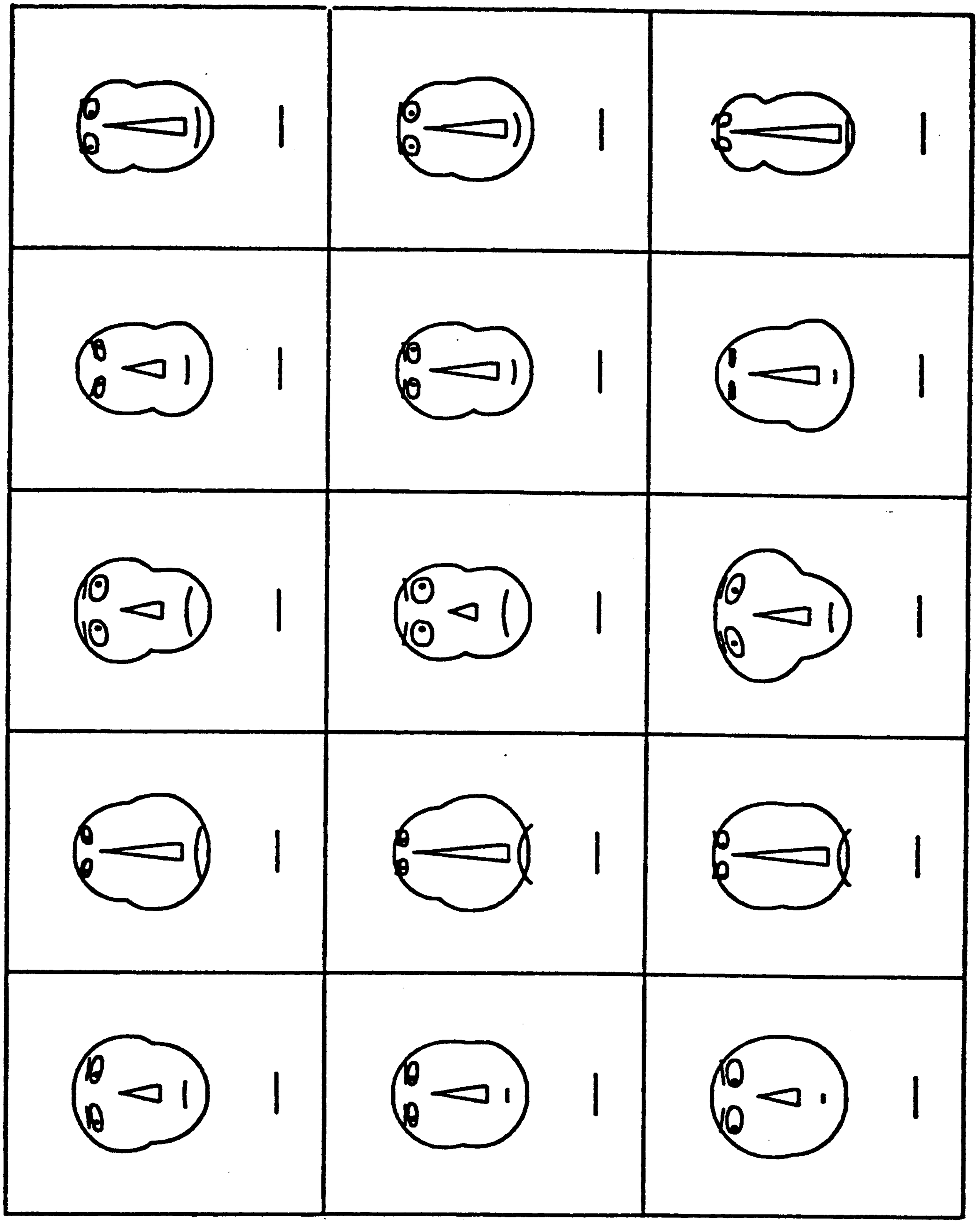


Figure 10. Subjects' performance clustering the nine-dimensional data points.

	Faces	*Polygon*	*Digits*
Mean number wrong	15.33	27.96	31.88
Standard deviation	5.16	4.98	7.30
Mean time (minutes)	4.14	3.69	8.24
Standard deviation	1.63	1.43	3.22

Figure 11. Specification of a direct manipulation screen button.

INTERACTION_OBJECT MessageFileDisplayButton **is**

IVARS:

position	:= { 100, 200, 50, 12 }; --*i.e., coordinates of screen rectangle*

METHODS:

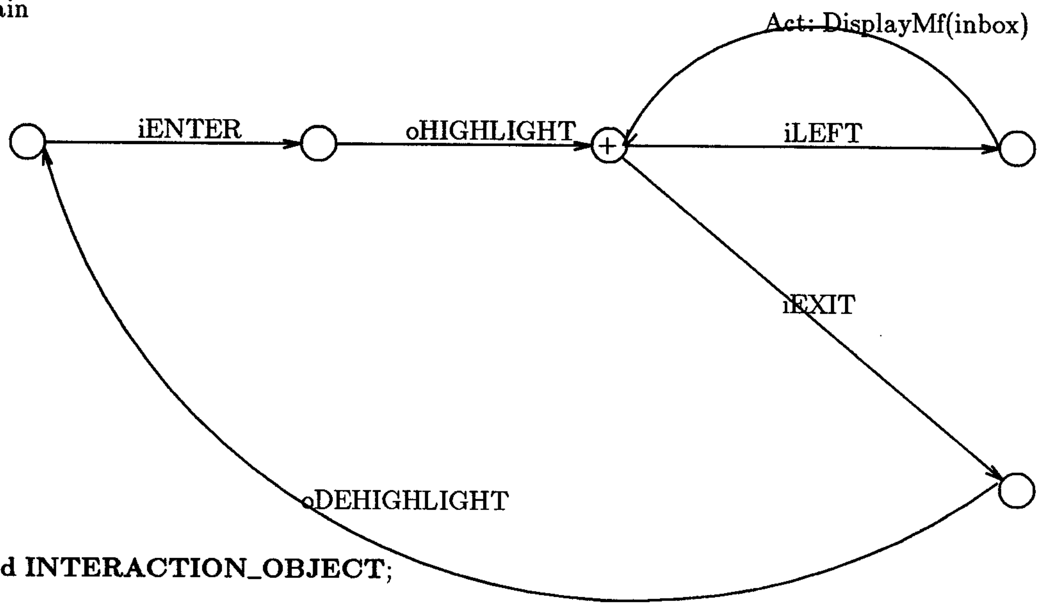
Draw	{ DRAW_TEXT_BUTTON(position, "Display"); }

TOKENS:

iLEFT	{ --*click left mouse button* -- }
iENTER	{ --*locator moves inside rectangle given by position*-- }
iEXIT	{ --*locator moves outside rectangle given by position*-- }
oHIGHLIGHT	{ --*invert video of rectangle given by position*-- }
oDEHIGHLIGHT	{ --*same as oHIGHLIGHT*-- }

SYNTAX:

main



end INTERACTION_OBJECT;

Figure 12. Specification of the screen button of Figure 11, using inheritance.

```
INTERACTION_OBJECT MessageFileDisplayButton2 is

FROM:          GenericButton, GenericItem;

IVARS:

position       := { 100, 200, 50, 12 };

legend         := "Display";

METHODS:

DoAction       { DisplayMf(inbox); }

end INTERACTION_OBJECT;
```