Exhibit 1014 – Part 5

describe the necessary action (`Type the address` or simply `address:`) and avoid pronouns (`You should type the address`) or references to the user (`The user of the form should type the address`). Another useful rule is to use the word `type` for entering information and `press` for special keys such as the TAB, ENTER, cursor movement, or programmed function (PFK, PF, or F) keys. Since "ENTER" often refers to the special key, avoid using it in the instructions (for example, do not use `Enter the address`, instead stick to `Type the address`.) Once a grammatical style for instructions is developed, be careful to apply that style consistently.

- *Logical grouping and sequencing of fields:* Related fields should be adjacent, and should be aligned with blank space for separation between groups. The sequencing should reflect common patterns—for example, city followed by state followed by zip code
- *Visually appealing layout of the form*: Using a uniform distribution of fields is preferable to crowding one part of the screen and leaving other parts blank. Alignment creates a feeling of order and comprehensibility. For example, the field labels `Name`, `Address`, and `City` were right justified so that the data-entry fields would be vertically aligned. This layout allows the frequent user to concentrate on the entry fields and to ignore the labels. If users are working from hard copy, the screen should match the paper form.
- *Familiar field labels*: Common terms should be used. If `Address` were replaced by `Domicile`, many users would be uncertain or anxious about what to do.
- *Consistent terminology and abbreviations*: Prepare a list of terms and acceptable abbreviations and use the list diligently, making additions only after careful consideration. Instead of varying such terms as `Address`, `Employee Address`, `ADDR.`, and `Addr.`, stick to one term, such as `Address`.
- *Visible space and boundaries for data-entry fields*: Underscores or other markers indicate the number of characters available, so users will know when abbreviations or other trimming strategies are needed.
- *Convenient cursor movement*: A simple and visible mechanism is needed for moving the cursor, such as a TAB key or cursor-movement arrows.
- *Error correction for individual characters and entire fields*: A backspace key and overtyping should be allowed to enable easy repairs or changes to entire fields.
- *Error messages for unacceptable values*: If users enter an unacceptable value, the error message should appear on completion of the field. The message should indicate permissible values of the field; for example, if

SKYHAWKE Ex. 1014, page 89

the zip code is entered as `28K21` or `2380`, the message might indicate that `Zip codes should have 5 digits`.

- *Optional fields clearly marked*: The word `Optional` or other indicators should be visible. Optional fields should follow required fields, whenever possible.
- *Explanatory messages for fields*: If possible, explanatory information about a field or its values should appear in a standard position, such as in a window on the bottom, whenever the cursor is in the field.
- *Completion signal*: It should be clear to the users what they must do when they are finished filling in the fields. Generally, designers should avoid automatic completion when the last field is filled, because users may wish to review or alter field entries.

These considerations may seem obvious, but often forms designers omit the title, or include unnecessary computer file names, strange codes, unintelligible instructions, unintuitive groupings of fields, cluttered layouts, obscure field labels, inconsistent abbreviations or field formats, awkward cursor movement, confusing error-correction procedures, hostile error messages, and no obvious way to signal completion.

Detailed design rules should reflect local terminology and abbreviations. They should specify field sequences familiar to the users; the width and height of the display device; highlighting features such as reverse video, underscoring, intensity levels, color, and fonts; the cursor movement keys; and coding of fields.

3.10.2 Coded fields

Columns of information require special treatment for data entry and for display. Alphabetic fields are customarily left justified on entry and on display. Numeric fields may be left justified on entry, but then become right justified on display. When possible, avoid entry and display of leftmost zeros in numeric fields. Numeric fields with decimal points should line up on the decimal points.

Special attention should be paid to such common fields as these:

- *Telephone numbers*: Offer a form to indicate the subfields:

  ```
  Telephone: (_ _ _) _ _ _-_ _ _ _
  ```

  Be alert to such special cases, such as addition of extensions or the need for nonstandard formats for foreign numbers.
- *Social-security numbers*: The pattern for Social-security numbers should appear on the screen as

  ```
  Social-security number: _ _ _ - _ _ - _ _ _ _
  ```

SKYHAWKE Ex. 1014, page 90

When the user has typed the first three digits, the cursor should jump to the leftmost position of the two-digit field.

- *Times*: Even though the 24 hour clock is convenient, many people find it confusing and prefer A.M. or P.M. designations. The form might appear as

  ```
  _ _:_ _ _ _ (9:45 AM or PM)
  ```

  Seconds may or may not be included, adding to the variety of necessary formats.

- *Dates*: How to specify dates is one of the nastiest problems; no good solution exists. Different formats for dates are appropriate for different tasks, and European rules differ from American rules. It may take years before an acceptable standard emerges.

  When the display presents coded fields, the instructions might show an example of correct entry; for example,

  ```
  Date: _ _/_ _/_ _ (04/22/93 indicates April 22, 1993)
  ```

  For many people, examples are more comprehensible than is an abstract description, such as `MM/DD/YY`.

- *Dollar amounts (or other currency)*: The dollar sign should appear on the screen, so users then type only the amount. If a large number of whole-dollar amounts is to be entered, users might be presented with a field such as

  ```
  Deposit amount: $_ _ _ _ _.00
  ```

  with the cursor to the left of the decimal point. As the user types numbers, they shift left. To enter an occasional cents amount, the user must type the decimal point to reach the `00` field for overtyping.

Other considerations in form-fillin design include multiscreen forms, mixed menus and forms, use of graphics, relationship to paper forms, use of pointing devices, use of color, handling of special cases, and integration of a word processor to allow remarks.

3.11 Practitioner's Summary

Begin by understanding the semantic structure of your application within the vast range of menu-selection situations. Concentrate on organizing the sequence of menus to match the users' tasks, ensure that each menu is a meaningful semantic unit, and create items that are distinctive and comprehensible. If some users make frequent use of the system, then typeahead, shortcut, or macro strategies should be allowed. Permit simple traversals to the previously displayed menu and to the main menu. Finally, be sure to conduct human-factors tests and to involve human-factors specialists in the

SKYHAWKE Ex. 1014, page 91

design process (Savage et al., 1982). When the system is implemented, collect usage data, error statistics, and subjective reactions to guide refinement.

Whenever possible, use a menu-builder or menu-driver system to produce and display the menus. Commercial menu-creation systems are available and should be used to reduce implementation time, to ensure consistent layout and instructions, and to simplify maintenance.

3.12 Researcher's Agenda

Experimental research could help us to refine the design guidelines concerning semantic organization and sequencing in single and linear sequences of menus. How can differing communities of users be satisfied with a common semantic organization when their information needs are very different? Should users be allowed to tailor the structure of the menus, or is there greater advantage in compelling everyone to use the same structure and terminology? Should a tree structure be preserved even if some redundancy is introduced? How can networks be made safe?

Research opportunities abound. Depth versus breadth tradeoffs under differing conditions need to be studied to provide guidance for designers. Layout strategies, wording of instructions, phrasing of menu items, use of color, response time, and display rate are all excellent candidates for experimentation. Exciting possibilities are becoming available with larger screens, graphic user interfaces, and novel selection devices.

Implementers would benefit from the development of software tools to support menu-system creation, management, usage-statistics gathering, and evolutionary refinement. Portability of *menuware* could be enhanced to facilitate transfer across systems.

References

Billingsley, P. A., Navigation through hierarchical menu structures: Does it help to have a map? *Proc. Human Factors Society, Twenty-Sixth Annual Meeting* (1982), 103–107.

Brown, C. Marlin, *Human-Computer Interface Design Guidelines*, Ablex, Norwood, NJ (1988).

Brown, James W., Controlling the complexity of menu networks, *Communications of the ACM 25*, 7 (July 1982), 412–418

Callahan, D., Hopkins, M., Weiser, M., and Shneiderman, B., An empirical comparison of pie versus linear menus, *Proc. CHI '88 Human Factors in Computer Systems*, ACM, New York (1988), 95–100.

Card, Stuart K., User perceptual mechanisms in the search of computer command menus, *Proc. Human Factors in Computer Systems* (March 1982), 190–196.

SKYHAWKE Ex. 1014, page 92

Clauer, Calvin Kingsley, An experimental evaluation of hierarchical decision-making for information retrieval, IBM Research Report RJ 1093, San Jose, CA (September 15, 1972).

Galitz, W. O., *Human Factors in Office Automation*, Life Office Managment Assn., Atlanta, GA (1980).

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

Herot, Christopher F., Graphical user interfaces. In Vassiliou, Y. (Editor), *Human Factors and Interactive Computer Systems*, Ablex, Norwood, NJ (1984), 83–103.

Kiger, John I., The depth/breadth trade-off in the design of menu-driven user interfaces, *International Journal of Man–Machine Studies 20* (1984), 201–213.

Koved, Lawrence, and Shneiderman, Ben, Embedded menus: Menu selection in context, *Communications of the ACM 29* (1986), 312–318.

Landauer, T. K., and Nachbar, D. W., Selection from alphabetic and numeric menu trees using a touch screen: Breadth, depth, and width, *Proc. Human Factors in Computing Systems*, ACM SIGCHI, New York (April 1985), 73–78.

Laverson, Alan, Norman, Kent, and Shneiderman, Ben, An evaluation of jump-ahead techniques for frequent menu users, *Behaviour and Information Technology 6* (1987), 97–108.

Lee, E., and Latremouille, S., Evaluation of tree structured organization of information on Telidon, *Telidon Behavioral Research I*, Department of Communications, Ottawa, Canada (1980).

Liebelt, Linda S., McDonald, James E., Stone, Jim D., and Karat, John, The effect of organization on learning menu access, *Proc. Human Factors Society, Twenty-Sixth Annual Meeting* (1982), 546–550.

McDonald, James E., Stone, Jim D., and Liebelt, Linda S., Searching for items in menus: The effects of organization and type of target, *Proc. Human Factors Society, Twenty-Seventh Annual Meeting* (1983), 834–837.

McEwen, S. A., An investigation of user search performance on a Telidon information retrieval system, *Telidon Behavioral Research 2*, Ottawa, Canada (May 1981).

Mantei, Marilyn, Disorientation behavior in person–computer interaction, Ph. D. Dissertation, Department of Communications, University of Southern California, Pasadena, CA (August 1982).

Martin, James, *Viewdata and the Information Society*, Prentice-Hall, Englewood Cliffs, NJ (1982).

Mitchell, Jeffrey and Shneiderman, Ben, Dynamic versus static menus: An experimental comparison, *ACM SIGCHI Bulletin 20*, 4 (1989), 33–36.

Murray, Robert P., and Abrahamson, David S., The effect of system response delay and delay variability on inexperienced videotext users, *Behavior and Information Technology 2*, 3 (1983), 237–251.

Norman, Kent, *The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface*, Ablex, Norwood, NJ (1991).

Norman, Kent L. and Chin, John P., The effect of tree structure on search in a hierarchical menu selection system, *Behaviour and Information Technology 7* (1988), 51–65.

Ogden, William C. and Boyle, James M., Evaluating human–computer dialog styles: Command versus form/fill-in for report modification, *Proc. Human Factors Society, Twenty-Sixth Annual Meeting*, Human Factors Society, Santa Monica, CA (1982), 542–545.

Pakin, Sherwin E. and Wray, Paul, Designing screens for people to use easily, *Data Management* (July 1982), 36–41.

Parton, Diana, Huffman, Keith, Pridgen, Patty, Norman, Kent, and Shneiderman, Ben, Learning a menu selection tree: Training methods compared, *Behaviour and Information Technology*, (1985), 81–91.

Perlman, Gary, Making the right choices with menus, *INTERACT '84*, First IFIP International Conference on Human–Computer Interaction, North-Holland, Amsterdam, The Netherlands (1984), 291–295.

Phillips, Chris H. E., & Apperley, Mark D., Direct Manipulation Interaction Tasks: A Macintosh-based Analysis, *Interacting with Computers 3*, 1 (1991), 9–26.

Robertson, G., McCracken, D., and Newell, A., The ZOG approach to man–machine communication, *International Journal of Man–Machine Studies 14* (1981), 461–488.

Savage, Ricky E., Habinek, James K., and Barnhart, Thomas W., The design, simulation, and evaluation of a menu driven user interface, *Proc. Human Factors in Computer Systems* (1982), 36–40.

Seabrook, R., and Shneiderman, B., The user interface in a hypertext, multi-window browser, *Interacting with Computers 1* (1989), 299–337.

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

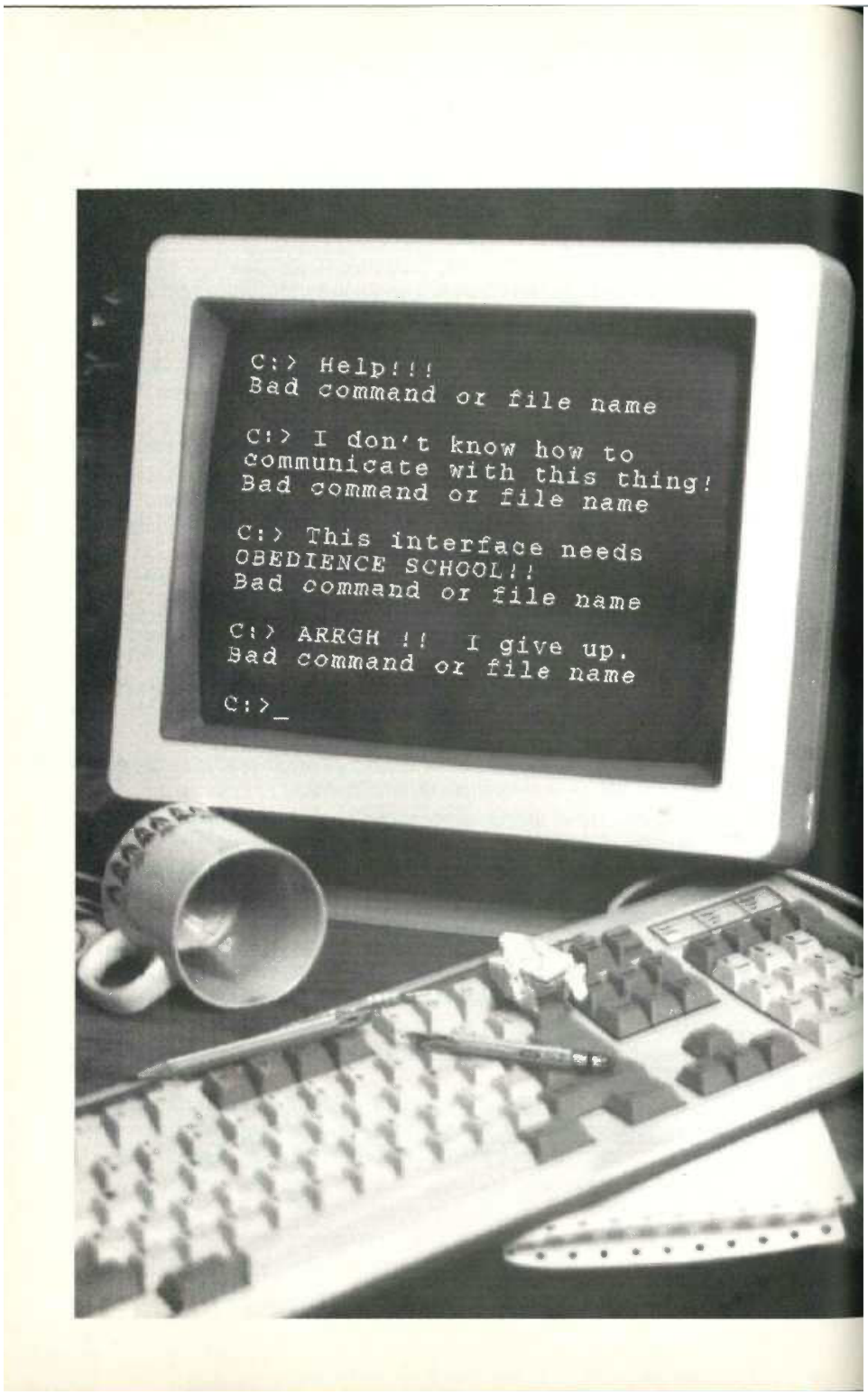
Somberg, Benjamin, and Picardi, Maria C., Locus of information familiarity effect in the search of computer menus, *Proc. Human Factors Society, Twenty-Seventh Annual Meeting* (1983), 826–830.

Teitelbaum, Richard C., and Granda, Richard, The effects of positional constancy on searching menus for information, *Proc. CHI '83, Human Factors in Computing Systems*, Available from ACM, Baltimore, MD (1983), 150–153.

Wallace, Daniel F., Anderson, Nancy S., and Shneiderman, Ben, Time stress effects on two menu selection systems, *Proc. Human Factors Society, Thirty-First Annual Meeting* (1987), 727–731.

Young, R. M., and Hull, A., Cognitive aspects of the selection of Viewdata options by casual users, *Pathways to the Information Society, Proc. Sixth International Conference on Computer Communication*, London, U.K. (September 1982), 571–576.

SKYHAWKE Ex. 1014, page 94

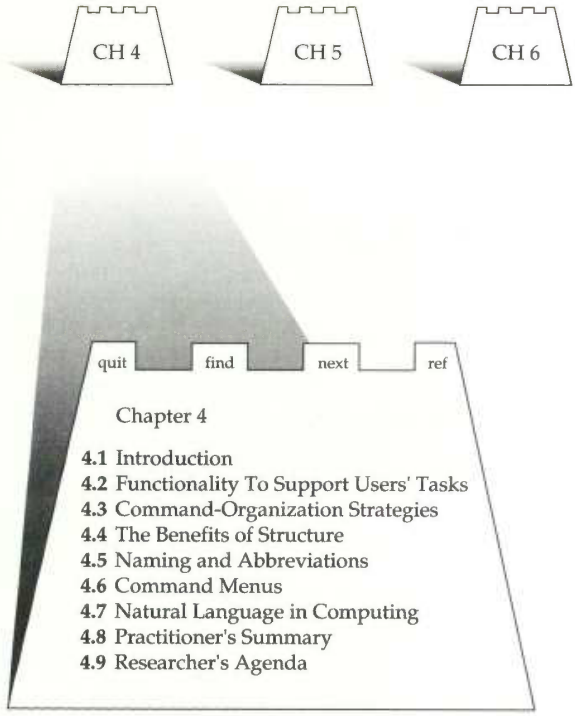

C:> Help!!!
Bad command or file name
C:> I don't know how to
communicate with this thing!
Bad command or file name
C:> This interface needs
OBEDIENCE SCHOOL!!
Bad command or file name
C:> ARRGH !! I give up.
Bad command or file name
C:>_

CHAPTER 4

Command Languages

I soon felt that the forms of ordinary language were far too diffuse . . . . I was not long in deciding that the most favorable path to pursue was to have recourse to the language of signs. It then became necessary to contrive a notation which ought, if possible, to be at once simple and expressive, easily understood at the commencement, and capable of being readily retained in the memory.

Charles Babbage, "On a method of expressing by signs the action of machinery," 1826



4.1 Introduction

The history of written language is rich and varied. Early tally marks and pictographs on cave walls existed for millennia before precise notations for numbers or other concepts appeared. The Egyptian hieroglyphs of 5000 years ago were a tremendous advance because standard notations facilitated communication across space and time. Eventually, languages with a small alphabet and rules of word and sentence formation dominated because of the relative ease of learning, writing, and reading. In addition to these natural languages, special languages for mathematics, music, and chemistry emerged because they facilitated communication and problem solving. In the twentieth century, novel notations were created for such diverse domains as dance, knitting, higher forms of mathematics, logic, and DNA molecules.

The basic goals of language design are

- Precision
- Compactness
- Ease in writing and reading
- Speed in learning
- Simplicity to reduce errors
- Ease of retention over time

Higher-level goals include

- Close correspondence between reality and the notation
- Convenience in carrying out manipulations relevant to users' tasks
- Compatibility with existing notations
- Flexibility to accommodate novice and expert users
- Expressiveness to encourage creativity
- Visual appeal

Constraints on a language include

- The capacity for human beings to record the notation
- The match between the recording and the display media (for example, clay tablets, paper, printing presses)
- The convenience in speaking (vocalizing)

Successful languages evolve to serve the goals within the constraints.

The printing press was a remarkable stimulus to language development because it made widespread dissemination of written work possible. The computer is another remarkable stimulus to language development, not only because widespread dissemination through networks is possible, but also because computers are a tool to manipulate languages and because languages are a tool for manipulating computers.

The computer has had only a modest influence on spoken natural languages, compared to its enormous impact as a stimulus to the development of numerous new formal written languages. Early computers were built to perform mathematical computations, so the first programming languages had a strong mathematical flavor. But computers were quickly found to be effective manipulators of logical expressions, business data, graphics, sound, and text. Increasingly, computers are used to operate on the real world: directing robots, issuing dollar bills at bank machines, controlling manufacturing, and guiding spacecraft. These newer applications encourage language designers to find convenient notations to direct the computer while preserving the needs of people to use the language for communication and problem solving.

SKYHAWKE Ex. 1014, page 98

Therefore, effective computer languages must not only represent the users' tasks and satisfy the human needs for communication, but also be in harmony with mechanisms for recording, manipulating, and displaying these languages in a computer.

Computer programming languages that were developed in the 1960s and early 1970s, such as FORTRAN, COBOL, ALGOL, PL/I, and Pascal, were designed for use in a noninteractive computer environment. Programmers would compose hundreds or thousands of lines of code, carefully check them over, and then *compile* or interpret by computer to produce a desired result. Incremental programming was one of the design considerations in BASIC and in advanced languages such as LISP, APL, and PROLOG. Programmers in these languages were expected to build smaller pieces online and interactively to execute and test the pieces. Still, the common goal was to create a large program that was preserved, studied, extended, and modified. The attraction of rapid compilation and execution led to the widespread success of the compact, but sometimes obscure, notation used in C. The pressures for team programming, organizational standards for sharing, and the increased demands for reusability promoted encapsulation and the development of object-oriented programming concepts in languages such as ADA and C++.

Scripting languages emphasizing screen presentation and mouse control became popular in the late 1980s, with the appearance of HyperCard, SuperCard, ToolBook, etc. These languages included novel operators, such as `ON MOUSE DOWN`, `BLINK`, or `IF FIRST CHARACTER OF THE MESSAGE BOX IS 'A'` (see Section 14.3.3).

Database query languages developed in the middle to late 1970s, such as SQL and QUEL, emphasized shorter segments of code (three to 20 lines) that could be written at a terminal and executed immediately. The goal of the user was more to create a result than a program.

Command languages, which originated with operating-systems commands, are distinguished by their immediacy and by their impact on devices or information. Users issue a command and watch what happens. If the result is correct, the next command is issued; if not, some other strategy is adopted. The commands are brief and their existence is transitory. Of course, command histories are sometimes kept and macros are created in some command languages, but the essence of command languages is that they have an ephemeral nature and that they produce an immediate result on some object of interest.

Command languages are distinguished from menu-selection systems in that their users must recall notation and initiate action. Menu selection users receive instructions and must recognize and choose among only a limited set of visible alternatives; they respond more than initiate. Command-language users are often called on to accomplish remarkable feats of memorization and typing. For example, this UNIX command, used to delete blank lines

SKYHAWKE Ex. 1014, page 99

from a file, is not obvious:

```
GREP -V ^$ FILEA > FILEB
```

Similarly, to get printout on unlined paper with the IBM 3800 laser printer, a user at one installation was instructed to type

```
CP TAG DEV E VTSO LOCAL 2 OPTCD=J F=3871 X=GB12
```

The puzzled user was greeted with a shrug of the shoulders and the equally cryptic comment that "Sometimes, logic doesn't come into play; it's just getting the job done." This style of work may have been acceptable in the past, but user communities and their expectations are changing. The empirical studies described in this chapter are beginning to clarify guidelines for many command-language design issues.

Command languages may consist of single commands or have complex syntax (Section 4.2). The language may have only a few operations, or may have thousands. Commands may have a hierarchical structure or permit concatenation to form variations (Section 4.3). A typical form is a verb followed by a noun object with qualifiers or arguments for the verb or noun. Abbreviations may be permitted (Section 4.5). Feedback may be generated for acceptable commands, and error messages (Section 8.2) may result from unacceptable forms or typos. Command-language systems may offer the user brief prompts, or may be close to menu-selection systems (Section 4.6). Finally, natural-language interaction can be considered as a complex form of command language (Section 4.7).

4.2 Functionality to Support Users' Tasks

People use computers and command-language systems to accomplish a wide range of tasks, such as text editing, operating-system control, bibliographic retrieval, database manipulation, electronic mail, financial management, airline or hotel reservations, inventory, manufacturing process control, and adventure games.

The critical determinant of success is the *functionality* of the system. People will use a computer system if it gives them powers not otherwise available. If the power is attractive enough, people will use a system despite a poor user interface. Therefore, the first step for the designer is to determine the functionality of the system by assessing the users' task domain.

A common design error is excess functionality. In a misguided effort to add features, options, and commands, the designer can overwhelm the user.

Excess functionality means more code to maintain, potentially more bugs, possibly slower execution, and more help screens, error messages, and user manuals (see Chapters 8 and 12). For the user, excess functionality slows learning, increases the chance of error, and adds the confusion of longer manuals, more help screens, and less specific error messages. On the other hand, insufficient functionality leaves the user frustrated because an apparent function is not supported. For instance, the system might require the user to copy the contents of the screen by hand because there is no simple print command, or to reorder the output because there is no sort command.

Evidence of excessive functionality comes from a study of 17 secretaries at a scientific research center who used IBM's XEDIT editor for a median of 18 months for 50 to 360 minutes per day (Rosson, 1983). Their usage of XEDIT commands was monitored for 5 days. The average number of commands used was 26 per user, with a maximum of 34; the number of commands was correlated with experience ($r = 0.49$). XEDIT has 141 commands, so even the most experienced user dealt with less than a quarter of the commands. Users did not appear to employ idiosyncratic subsets of the language, but instead added commands to their repertoire in an orderly and similar pattern.

Careful task analysis might result in a table of user communities and tasks with each entry indicating expected frequency of use. The high-volume tasks should be made easy to carry out, and then the designer must decide which communities of users are the prime audience for the system. Users may differ in their position in an organization, their knowledge of computers, or their frequency of system usage. One difficulty in carrying out such a task analysis is predicting who the users might be and what tasks they might need to accomplish.

Inventing and supplying new functions are the major goals of many designers. They know that marketplace acceptance is often determined by the availability of functions that the competition does not provide. Word-processor designers continue to add such functions as boldface, footnotes, dual windows, mail merge, table editing, graphics, or spelling checks to entice customers. A feature-analysis list (Figure 4.1) can be helpful in comparing designs and in discovering novel functions (Roberts, 1980).

At an early stage, the destructive operations—such as deleting objects or changing formats—should be evaluated carefully to ensure that they are reversible, or at least are protected from accidental invocation. Designers should also identify error conditions and prepare error messages. A transition diagram showing how each command takes the user to another state is a highly beneficial aid to design, as well as to eventual training of users (Figure 4.2). If the diagram grows too complicated, it may signal the need for system redesign.

Major considerations for expert users are the possibilities of tailoring the language to suit personal work styles, and of creating named macros to

SKYHAWKE Ex. 1014, page 101

Text Editor Feature List

Estimated time to install (15 minutes to 2 hours)

Number of diskettes provided (1 to 7)
Right to make copies

On-screen tutorial
Textbook tutorial
Textbook reference guide
Online help
Meaningful error messages

Spelling checker built in to word processor
Thesaurus built in to word processor
Mail merge
Automatic table of contents generation
Automatic index generation

Menu, command, or function-key driven
Save block
Block defined by highlight or markers

Maximum size for block operation
Document size limit
Capacity to edit more than one file
Rename disk files
Copy disk files
Show disk directory

What you see is what you get
Preview print format
Editing allowed during printing
Print part of file
Chain documents for printing
Queue documents for printing
Automatic page numbering
Print multiple copies
Automatic file save
Save file without exiting
Automatic backup file
Create file without embedded codes

Subscript/superscript
Italics
Underscoring
Boldface
Multiple fonts

Multiple font sizes
Left and right justification
Centering
Tabbing
Proportional spacing
Multiple column output
Footnotes
Endnotes
Line-spacing options
Number of printers supported

Characters per line range (78-455)
Lines per screen
Change screen colors
Redefine key functions
Specify macros
Automatic hyphenization
Switch from insert to overwrite modes
Automatic indentation
Multiple indents/outdents
Change case command
Display ruler line to show tabs
Display column, line, and page number
Headers and footers
Math functions
Sort functions

Move cursor by character, word, sentence, paragraph
Move cursor by screen
Move cursor to left or right end of line
Move cursor to top or bottom of screen
Move cursor to top or bottom of document

Delete by character, word, line, sentence, or paragraph
Delete to end of document
Undelete

Search forward and backward
Search by patterns
Ignore case in searching
Leave and locate markers

Copy/move
Copy/move by columns

Figure 4.1

A feature-analysis list can be helpful in comparing designs and in discovering novel functions. This list was distilled from Roberts (1980) and Wiswell, Phil, Word processing: The latest word, *PC Magazine* (August 20, 1985), 110–134.

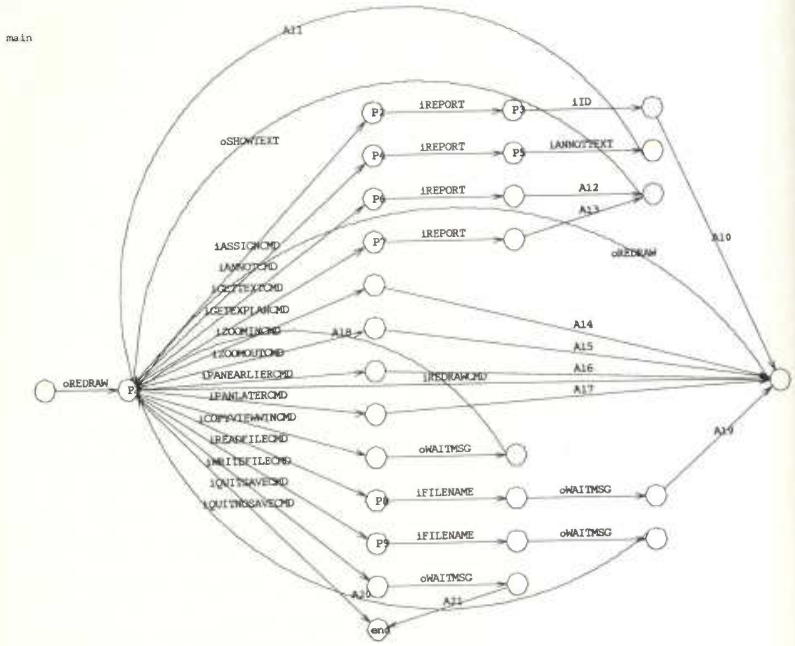


Figure 4.2

This transition diagram indicates user inputs with an "i" and computer outputs with an "o". This is a relatively simple diagram showing only a portion of the system. Complete transition diagrams may be many pages long. (Courtesy of Robert J. K. Jacob, Naval Research Laboratory, Washington, DC.)

permit several operations to be carried out with a single command. Macro facilities allow extensions that the designers could not foresee or that are beneficial to only a small fragment of the user community. A macro facility can be a full programming language that might include specification of arguments, conditionals, iteration, integers, strings, and screen-manipulation primitives, plus library and editing tools. Well-developed macro facilities are one of the strong attractions of command languages.

4.3 Command-Organization Strategies

Several strategies for command organization have emerged, but guidelines for choosing among these are only beginning to be discussed. A unifying concept, model, or metaphor is an aid to learning, problem solving, and retention

(Carroll and Thomas, 1982). Electronic-mail enthusiasts conduct lively discussions about the metaphoric merits of such task-related objects as file drawers, folders, documents, memos, notes, letters, or messages. They debate the appropriate task domain actions (CREATE, EDIT, COPY, MOVE, DELETE) and the choice of an action pair LOAD/SAVE (too much in the computer domain), READ/WRITE (acceptable for letters, but awkward for file drawers), or OPEN/CLOSE (acceptable for folders, but awkward for notes).

Similarly, debate continues over whether the commands should manipulate lines, as in program editors and older line-oriented editors, or words, sentences, and paragraphs, as in new word processors. Choosing one strategy over another is helpful. Designers who fail to choose, and instead attempt to support every possibility, risk overwhelming the users while missing the opportunity to optimize for one strategy. Designers often err by choosing a metaphor closer to the computer domain than to the user's task domain. Of course, metaphors can mislead the user, but careful design can reap the benefits while reducing the detriments.

Having adopted a concept, model, or metaphor for operations, the designer must now choose a strategy for the command structure. Mixed strategies are possible, but learning, problem solving, and retention may be aided by limitation of complexity.

4.3.1 Simple command list

Each command is chosen to carry out a single task, and the number of commands matches the number of tasks. With a small number of tasks, this approach can produce a system that is simple to learn and use. With a large number of commands, there is danger of confusion. The vi editor on UNIX systems offers many commands while attempting to keep the number of keystrokes low. The result is complex strategies employing single letters, shifted single letters, and CTRL key plus single letters (Figure 4.3). Furthermore, some commands stand alone, whereas others must be combined, often in irregular patterns.

4.3.2 Command plus arguments

Each command (COPY, DELETE, PRINT) is followed by one or more arguments (FILEA, FILEB, FILEC) that indicate objects to be manipulated:

```
COPY FILEA,FILEB
DELETE FILEA
PRINT FILEA,FILEB,FILEC
```

Commands may be separated from the arguments by a blank or other delimiter, and the arguments may have blanks or delimiters between them

vi Commands to Move the Cursor

Moving within a window

H	home position (upper left)
L	last line
M	middle line
(CR)	next line (carriage return)
+	next line
-	previous line
CTRL-P	previous line in same column
CTRL-N	next line in same column
(LF)	next line in same column (line feed)

Moving within a line

0	start of line
$	end of line
(space)	right one space
CRTL-H	left one space
h	left one space
w	forward one word
b	backward one word
e	end (rightmost) character of a word
)	forward one sentence
(	backward one sentence
}	forward one paragraph
{	backward one paragraph
W	blank out a delimited word
B	backwards blank out a delimited word
E	go to the end of a delimited word

Finding a character

fx	find the character x going forward
Fx	find the character x going backward
tx	go up to x going forward
Tx	go up to x going backward

Scrolling the window

CTRL-F	forward one screen
CTRL-B	backward one screen
CTRL-D	forward one half screen
CTRL-U	backward one half screen
G	go to line
/pat	go to line with pattern forward
pat	go to line with pattern backward

Figure 4.3

Commands to move the cursor. The profusion of commands in `vi` may enable expert users to get tasks done with just a few actions, but the number of commands can be overwhelming to novice and intermittent users.

(Schneider et al., 1984). Keyword labels for arguments may be helpful to some users; for example,

```
COPY FROM=FILEA TO=FILEB
```

The labels require extra typing and increase chances of a typo, but readability is improved and order dependence is eliminated.

4.3.3 Command plus options and arguments

Commands may have options (3, HQ, and so on) to indicate special cases. For example,

```
PRINT/3,HQ FILEA
PRINT (3,HQ) FILEA
PRINT FILEA -3,HQ
```

may produce three copies of FILEA at the printer in the headquarters building. As the number of options grows, the complexity can become overwhelming and the error messages less specific. The arguments may also have options, such as version numbers, privacy keys, or disk addresses.

The number of arguments, of options, and of permissible syntactic forms can grow rapidly. One airline-reservations system uses the following command to check the seat availability on a flight on August 21, from Washington's National Airport (DCA) to New York's La Guardia Airport (LGA) at about 3:00 P.M.:

```
A0821DCALGA0300P
```

Even with substantial training, error rates can be high with this approach, but frequent users seem to manage and even appreciate the compact form of this type of command.

The UNIX command-language system is widely used, in spite of the complexity of its command formats (Figure 4.4), which have been criticized severely (Norman, 1981). Here again, users will master complexity to benefit from the rich functionality in a system. Observed error rates with actual use of UNIX commands have ranged from 3 to 53 percent (Kraut et al., 1983; Hanson et al., 1984). Even common commands have generated high syntactic error rates: mv (18 percent), cp (30 percent), and awk (34 percent). Still, the complexity has a certain attraction for a portion of the potential user community. Users gain satisfaction in overcoming the difficulties and becoming one of the inner circle ("gurus" and "wizards") who are knowledgeable about system features—command-language macho.

SKYHAWKE Ex. 1014, page 106

```
at 2A Friday timehog
    at 2:00 a.m. Friday, run program
awk '{print $1 + $2}' file1
    print sum of first two fields of each line
cat -n file1
    print specified file to terminal, number output lines
cat file1 >> file2
    append file1 to end of file2
cc file.c
    compile C program, executable in a.out
cd /usr/lib
    change working directory to specified one
chmod g+rw file1 file2
    change mode of files, adding group read and write access
chmod 600 file
    change mode of file, allow only read and write by owner
cp file1 file2
    make a copy of file1 named file2
cp -r dir /tmp
    recursively copy dir and its subdirectories to /tmp
diff -l dir1 dir2
    summarize differences between files in dir1 and dir2
f77 file.f
    compile Fortran program, executable in a.out
f77 -o file file.f file.o
    compile Fortran program, link with file.o, executable in file
find $HOME -name '#*' -exec rm {} \;
    remove files with names beginning with a pound sign
finger name
    look up information on user's login or real name
grep '[Pp]hone' file
    print all lines in file containing Phone or phone
grep -l main *
    print names of files in current directory containing main
head -6 file
    print first six lines of file
kill -9 0
    send a KILL signal to processes started since login
ln -s file1 file2 /tmp
    make symbolic links to files in specified directory
lpq job user
    report print spooler status of user's job
lpr -p file
    paginate file and spool it to the line printer
ls
    print a list of the files in the current directory
ls -R /bin
    list files in specified directory and its subdirectories
mail molly tracey < file
    send a file to specified users as mail
man spell
    print Unix user's manual page for a command
mkdir /tmp/myjunk
    make a new directory
more +50 file
    view file by screenful, starting at line 50
mv file1 file2 /tmp
    move files to specified directory
nroff file | more
    preview formatted file on terminal
pc file.p
    compile Pascal program, executable in a.out
pr file | lpr
    paginate a file with default header, spool output
ps l
    print long listing of current processes, PID's and status
pwd
    print current working directory
rlogin puter2
    login on remote computer
rm file
    remove (delete) a file
rm -i junk[0-9]
    remove files junk0 ... junk9, confirming first
sort +3 -4 file
    print file sorted only on fourth field
stty everything
    print all stty option settings
stty raw; prog; stty -raw
    set terminal to raw mode, run a program, and restore mode
style -p file
    print sentences in file containing a passive verb
vi file
    edit file using full screen editor
w
    list who's logged in, and what they're doing
```

Figure 4.4

Examples of common UNIX commands with brief explanations. (Courtesy of Specialized Systems Consultants, Inc., Seattle WA.)

4.3.4 Hierarchical command structure

The full set of commands is organized into a tree structure, like a menu tree. The first level might be the command action, the second might be an object argument, and the third might be a destination argument:

Action	Object	Destination
CREATE	File	File
DISPLAY	Process	Local printer
REMOVE	Directory	Screen
COPY		Laser printer
MOVE		

If a hierarchical structure can be found for a set of tasks, it offers a meaningful structure to a large number of commands. In this case, 5 x 3 x 4 = 60 tasks can be carried out with only five command names and one rule of formation. Another advantage is that a command-menu approach can be developed to aid the novice or intermittent user, as was done in VisiCalc and later Lotus 1-2-3 and Excel.

Several help systems allow a hierarchical command to retrieve text about subsystems and the letter's commands. For example, to get help on the editor command for deleting lines in a document, the user might type

```
HELP EDIT DELETE LINES
```

Of course, the difficulty comes in knowing what keywords are available. Users can type the first few elements of the command, and then receive a menu of items.

Many word processors, spreadsheets, and operating systems use a hierarchical command structure for the numerous commands that they support. For example, Figure 4.5 shows the command structure for MS-DOS 5.0.

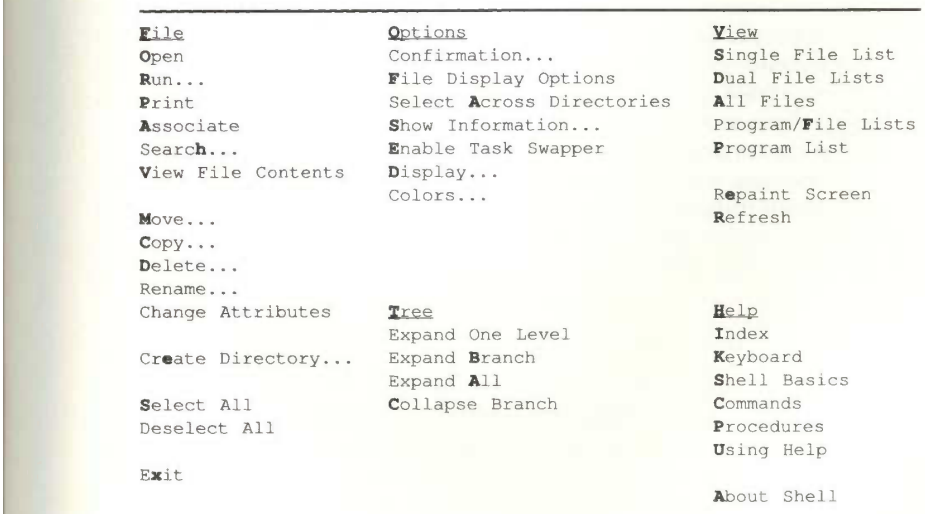


Figure 4.5

The tree structure of menus in Microsoft MS-DOS 5.0. (Screen shot ©1981–1991 Microsoft Corporation. Reprinted with permission from Microsoft Corporation, Redmond, WA.)

4.4 The Benefits of Structure

Human learning, problem solving, and memory are greatly facilitated by meaningful structure. If command languages are well designed, users can recognize the structure and can easily encode it in their semantic knowledge storage. For example, if users can uniformly edit such objects as characters, words, sentences, paragraphs, chapters, and documents, this meaningful pattern is easy to learn, apply, and recall. On the other hand, if they must overtype a character, change a word, revise a sentence, replace a paragraph, substitute a chapter, and alter a document, then the challenge grows substantially, no matter how elegant the syntax (Scapin, 1982).

Meaningful structure is beneficial for task concepts, computer concepts, and syntactic details of command languages. Yet, many systems fail to provide a meaningful structure. One widely used operating system displays various information as a result of forms of the `LIST`, `QUERY`, `HELP`, and `TYPE` commands, and moves objects as a result of the `PRINT`, `TYPE`, `SPOOL`, `PUNCH`, `SEND`, `COPY`, or `MOVE` commands. Defaults are inconsistent for different features, four different abbreviations for `PRINT` and `LINECOUNT` are required, binary choices vary between `YES/NO` and `ON/OFF`, and function-key usage is inconsistent. These flaws emerge from multiple uncoordinated design groups and insufficient attention by the managers, especially as features are added over time.

An explicit list of design conventions in a Guidelines Document can be an aid to designers and managers. Exceptions may be permitted, but only after thoughtful discussions. Users can learn systems with inconsistencies, but they do so more slowly and with greater chance of making mistakes. One difficulty is that there may be conflicting design conventions.

4.4.1 Consistent argument ordering

Choices among conventions can sometimes be resolved by experimentation with alternatives. A command language with six functions, each requiring two arguments, was developed for decoding messages (Barnard et al., 1981). One argument was always a message-identification number, and the other argument was a file number, code number, digit, and so on. In normal English usage, the message identification sometimes would be the direct object of an explanatory sentence, such as `SAVE` the `MESSAGE ID` with this `REFERENCE NUMBER`. So, one rule of consistent command formation was to follow English usage. The second consistency rule was to have the message identification always as the first or always as the second argument. The rules

resulted in four possible command groups:

Direct object, argument first		Direct object, argument second	
SEARCH	file no,message id	SEARCH	message id,file no
TRIM	message id,segment size	TRIM	segment size,message id
REPLACE	message id,code no	REPLACE	code no,message id
INVERT	group size,message id	INVERT	message id,group size
DELETE	digit,message id	DELETE	message id,digit
SAVE	message id,reference no	SAVE	reference no,message id

Consistent, argument first		Consistent, argument second	
SEARCH	message id,file no	SEARCH	file no,message id
TRIM	message id,segment size	TRIM	segment size,message id
REPLACE	message id,code no	REPLACE	code no,message id
INVERT	message id,group size	INVERT	group size,message id
DELETE	message id,digit	DELETE	digit,message id
SAVE	message id,reference no	SAVE	reference no,message id

Forty-eight female subjects used one of these systems for 1 hour to decode messages. (Actually, one-half of the subjects had variant command names, such as SELECT instead of SEARCH, but this manipulation was a minor effect.) Time to perform tasks decreased during the 10 1-hour trials, but the speedup was consistent across command styles. The results strongly favored using consistent argument positions rather than the consistent direct-object positions, suggesting that English language rules of formation were not as effective as is the simpler positional rule. The shortest task times, fewest help requests, and fewest errors occurred with the consistent argument first. These results lead to the conjecture that command languages should allow users to express the simple, familiar, or well-understood features first, and then to consider the more varying aspects.

Follow-up studies by the same group (Barnard et al., 1981; Barnard et al., 1982) replicated the results about positional consistency and pursued several related issues. One frequent design consideration is whether the command verb or the object of interest should come first. Command-first form would be DISPLAY FILE or INSERT LIST; the object first form would be FILE DISPLAY or LIST INSERT. The evidence supports the command-first strategy used in most languages and the principle that there is a fixed order. Allowing users the freedom to put the command and object in either order generated more requests for help than did fixing the order. Subjects pressed function keys to initiate commands and to select objects, so a further