User engineering principles for interactive systems*

by WILFRED J. HANSEN

Argonne National Laboratory
Argonne, Illinois

INTRODUCTION

The 'feel' of an interactive system can be compared to the impressions generated by a piece of music. Both can only be experienced over a period of time. With either, the user must abstract the structure of the system from a sequence of details. Each may have a quality of 'naturalness' because successive actions follow a logically self-consistent pattern. Finally, a good composer can write a new pattern which will seem, after a few listenings, to be so natural the observer wonders why it was never done before.

Just as a composer follows a set of harmonic principles when he writes music, the system designer must follow some set of principles when he designs the sequence of give and take between man and machine. This paper reports a set of principles—called user engineering principles—which where employed while designing the Emily text editing system. These principles evolved during the course of the project, but were originally based on the author's experiences with a number of other text editing systems.[2,3,4,5]

In text editing applications, the user sits at a console and creates, views, or modifies a document, be it program, speech, article or a chapter of his next book. Here the computer is a tool for the creative worker and the emphasis must be on capturing his thoughts with minimal interference. More common in commercial environments are interactive systems designed as tools to coordinate the work of many clerical workers. Examples are order entry, point-of-sale, inventory control, defense surveillance, and the like. The principles outlined below, though originally intended for creative work, are equally applicable to clerical work. Sometimes more so, because clerks may not have the commitment of the creative worker.

* The work reported here was supported by the U.S. Atomic Energy Commission. The text is taken from the second and fourth chapters of the author's thesis.[1]

One restriction on a few of the principles below is that they apply to systems with display devices for output. This is essential, because a basic principle is that the system respond to the user as fast as possible. A visual display can present more information in less time than available hardcopy devices. The 'economy' of the terminal device must be weighed against the cost of attention-wander-time as the user interacts with the system. Other than the terminal, cost is not a problem in the application of these user engineering principles. In general, they dictate features that are inexpensive to design into a system. They are, however, often expensive to include after implementation is under way.

Disciplines similar to user engineering have been called human engineering, human factors, and ergonomics, but these terms most often refer to analog systems like airplane cockpits where the pilot guides a process. User engineering applies to digital systems where the goal is to store or retrieve information. D. Engelbart[6] refers to these principles as 'User Feature Design.' His point is that this term emphasizes that the features are being designed for the user rather than the other way around. In fact, though, any interactive system will require retraining of the users and some systems—like Emily—may require the user to alter thinking habits of many years standing. (But let there be no mistake, the author is deeply committed to a policy of modifying the system to fit the user.) Other sets of user engineering principles have been reported by L. B. Smith[7] and J. G. Mitchell.[8] Their suggestions are compatible with those below, but less comprehensive. The reader should also read R. B. Miller's paper[9] in which he attempts to estimate a maximum permissible response time in seventeen interactive contexts.

The user engineering principles in the second section below are illustrated by reference to the Emily text editing system. For this reason, the Emily system is sketched in the first section. More complete descrip-

```
1     <STMT> : DO <ARITHV> = <ARITHX> TO
                   <ARITHX>; <STMT*> END;
2          : <ASGN STMT>
3     <STMT*> : <STMT>
4     <ASGN STMT> : <ARITH> = <ARITHX>;
5     <ARITHX> : <ARITH>
6          : <ARITHV>
7          : <NUMBER>
8          : <ARITHX> + <ARITHX>
9     <ARITHX*> : <ARITHX>
10    <ARITHV> : <ARITH>
11         : <ARITH> (<ARITHX*>)
12    <ARITH> IS AN IDENTIFIER
13    <NUMBER> IS A CONSTANT
```

Figure 1—Portion of syntax for PL/I
Each rule specifies a possible replacement for the non-terminal to the left of the colon. If the left side is omitted, it is the same as the previous line. Rules 12 and 13 specify special classes of terminal symbols

tions are available elsewhere.[10,1,11] Emily has been implemented for an IBM 2250 Graphic Display Unit, model 3. The 2250 displays lines and characters on a 12″ by 12″ screen. The user can give commands to the system with a light pen, a program function keyboard, and an alphameric keyboard.

THE EMILY SYSTEM

Emily is primarily intended for construction and modification of computer programs written in higher level languages. Many such systems exist, but all existing systems require the programmer to enter his text as a sequence of characters. With Emily, the user constructs his text by selecting choices from the menu to replace certain symbols in the text. For example, the symbol ⟨STMT⟩ might be replaced by

```
DO ⟨ARITHV⟩ = ⟨ARITHX⟩ TO ⟨ARITHX⟩;
   ⟨STMT*⟩
END;
```

Replaceable symbols begin with '⟨', end with '⟩', and contain a name that usually has some relation to the meaning of the string generated by the symbol. Such symbols are called *non-terminal symbols*, because of their role in the Backus-Naur Form (BNF) notation for describing programming languages.[12]

In BNF a syntax for a formal language has three parts—a set of *terminal symbols*, a set of *non-terminal symbols*, and a set of *syntactic rules*. The terminal symbols are those characters and strings of characters (punctuation, reserved words, identifiers, constants) that can be part of the completed text. The non-terminal symbols are a specific set of symbols introduced only to help describe the structure of the formal language. Every non-terminal symbol must be replaced by terminal symbols before the entire text is complete,

```
[<STMT>]                                              1

DO [<ARITHV>] = <ARITHX> TO <ARITHX>;                 10
   <STMT*>
END;

DO [<ARITH>] = <ARITHX> TO <ARITHX>;                  12
   <STMT*>
END;

DO I = [<ARITHX>] TO <ARITHX>;                        7
   <STMT*>
END;
. . .                                          13,7,13,3,2
DO I = 1 TO 20;                                       4
   [<ASGN STMT>]
END;
. . .                                  12,8,5,12,6,11,
                                          12,9,5,12
DO I = 1 TO 20;
   S = S + A(I);
END;
```

Figure 2—Steps in the generation of a DO loop
In each step, the non-terminal in the rectangle is replaced according to the rule whose number appears at the right

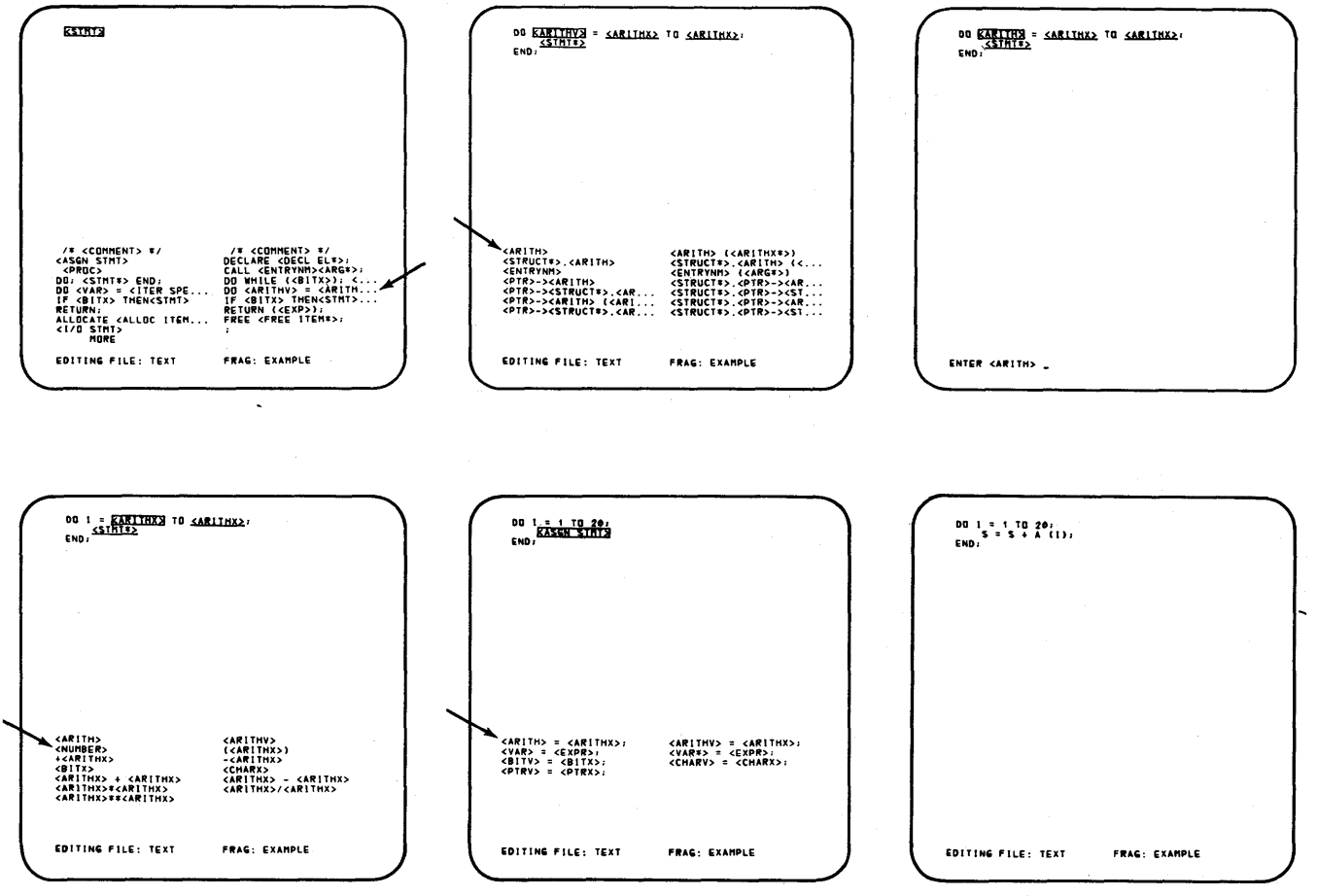


Figure 3—Generation of a DO loop with Emily

These photographs show the same steps as shown in Figure 2. The menu displays all the choices available in the implemented PL/I syntax. An arrow indicates the syntax rule the user will select next. Up to twenty-two lines of text may be shown in the text area, so it appears empty with only 3 lines

but the only allowable replacements for a given non-terminal are specified by the syntactic rules. In each rule, the given non-terminal is on the left followed by a colon followed by the sequence of symbols that may replace the non-terminal. As an example, Figure 1 shows a portion of the syntax for PL/I. Figure 2 shows a DO loop generated using this syntax.

It is important to note that a string generated according to a syntax is not simply a sequence of characters, but can be divided into hierarchies of substrings on the basis of the syntactic rules. Each non-terminal in the sequence of symbols for a rule generates a subsequence. The DO statement in Figure 2 can be one of a sequence of statements in some higher DO loop and can also contain a subordinate sequence of statements (generated by ⟨STMT*⟩). Replacement of a non-terminal by a rule can be thought of as replacing the non-terminal with a pointer to a copy of the rule. The non-terminals in this copy can be further replaced by pointers to copies of other rules. In a diagram each syntactic rule used in the generation of the string is represented by a *node* (a rectangle). The node contains one pointer to a subordinate node for each non-terminal in the syntactic rule. The subordinate node is called a subnode or a descendant, while the pointing node is called the parent.

Emily text structure

Text in the Emily system is stored in a *file*, which may contain any number of *fragments*. Each fragment has a name and contains a piece of text generated by some non-terminal symbol. Generated text is physically stored in a hierarchical structure like that described above. Each node is a section of memory containing (a) the number of the syntax rule for which this node

was generated, and (b) one pointer to each subnode. In a completed text, there is one descendant node for each non-terminal in the syntax rule and the pointer to a descendant is the address of the section of memory where it is stored. If no text has been generated for a non-terminal symbol, there is no subnode and the corresponding pointer is replaced by a code representing the non-terminal symbol. If a subnode of a node is an identifier, the pointer points at a copy of the identifier in a special area. All pointers at a given identifier point to the same copy in this identifier area. Other than identifiers, each node is pointed at exactly once within the text structure. This guarantees that if a node is modified, only one piece of text is affected.

Notice that punctuation and reserved words do not appear in this representation of text. Instead, they can be generated because the syntax rule number identifies the appropriate rule. Two tables in Emily contain coded forms of the syntax rules. One table, called the *abstract syntax*, controls the hierarchical structure of generated text. It specifies which syntax rules can replace a given non-terminal symbol and the sequence of non-terminal symbols on the right-hand-side of each syntax rule. Another table, the *concrete syntax*, tells how to display each rule; it includes punctuation, reserved words, and formatting information like indentation and line termination.

Creating text

The Emily user creates hierarchical text in a series of steps very similar to Figure 2. In each step the right side of a rule is substituted for a non-terminal symbol. Before the user creates any text, the fragment contains a single non-terminal symbol. In the case of Figure 2, that symbol is ⟨STMT⟩. The user sees the result of each step on the 2250 display. Figure 3 shows the steps of Figure 2 as they appear on the screen.

While using the Emily system the 2250 screen appears to be divided into three areas: text, menu, and message. The text area occupies the upper two-thirds of the screen and displays the text the user is creating. The lower third of the screen is the menu where Emily displays the strings the user can substitute in the text. The bottom line of the screen is the message area, where Emily requests operands and displays status and error messages.

Non-terminal symbols** in the text area are underlined to make them stand out. One of the non-terminals is the *current non-terminal* and is surrounded by a rectangle. The menu normally displays all strings that can be substituted for the current non-terminal. These strings are simply the right sides of the syntax rules that have the current non-terminal on the left.

When the user points the light pen at an item in the menu Emily substitutes that item for the current non-terminal. Usually, the substitution string contains more than one non-terminal and the new current non-terminal is the first of these. The user can also change the current non-terminal by pointing the light pen at any non-terminal in the display. Emily moves the rectangle to that non-terminal and changes the menu accordingly. When the current non-terminal is an identifier, the menu displays identifiers previously entered in the required class (some of the classes for PL/I are ⟨ARITH⟩, ⟨CHAR⟩, and ⟨ENTRYNM⟩). The user may select one of these, or he may enter a new identifier from the keyboard. Constants are also entered from the keyboard.

Viewing text

Since text is stored hierarchically within Emily, it can be viewed with operations that take advantage of that structure. The user may wish to descend into the structure and examine the details of some minor substructure. Alternatively, he may wish to view the highest levels of the hierarchy with substructures represented by some appropriate symbol. Both of these viewing operations are possible with Emily.

The symbol displayed to represent a substructure is called a *holophrast*. This symbol begins and ends with an exclamation mark and contains two parts separated by a colon. The first part is the non-terminal symbol that generated the substructure and the second part is the first few characters of the represented string. Figure 4 shows three examples of holophrasts. Note that contraction to a holophrast only changes the view of the file and it does not modify the file itself. Moreover, the user never enters a holophrast from the keyboard; they are displayed only as a result of contracting text.

The user contracts a structural unit in the display by pushing a button on the program function keyboard and then pointing at some character in the text. The selected character is part of the text generated by some node in the hierarchical structure. The display of this node is replaced by a holophrast. If the user points at a holophrast, the father of the indicated node contracts to a holophrast which subsumes the earlier one. To expand a holophrast back to a string, the user returns to normal text construction mode and points the light pen at the holophrast.

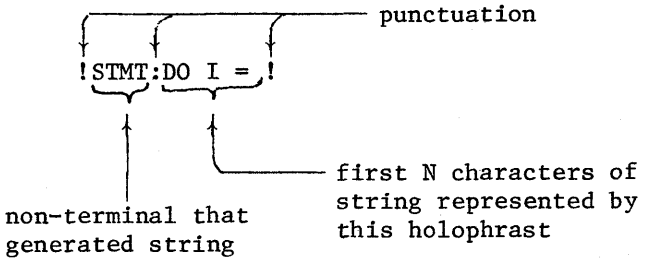
** When it is displayed, a non-terminal is the end (or terminal) of a branch of the hierarchical structure. It is called a non-terminal because it must be replaced with a string of terminals before the text is complete.

The operations to ascend and descend in the text hierarchy are also invoked by program function buttons. To descend in the hierarchy the user pushes the IN button and points at a part of the text. The selected node becomes the new *display generating node*; subsequent displays show only this node and its subnodes. The OUT button lets the user choose among the ancestors of the display generating node and then makes the selected ancestor the new display generator.

System environment

At Argonne National Laboratory, the 2250 is attached to an IBM 360 model 75. The 75 is under control of the MVT version of OS/360. Unit record input/output is controlled by ASP in an attached 360/50. The 360/75 has one million bytes of main core and one million bytes of a Large Capacity Storage Unit.

The Emily system itself requires 60K bytes of main core (the maximum permitted for a 2250 job at Argonne) and about 400K bytes of LCS. Emily is written in PL/I and uses the Graphic Subroutine Package to communicate with the 2250. Files for Emily are stored on a 2314 disk pack. Emily is table driven and can manipulate text in any formal language. To date, tables have been created for four languages: PL/I, GEDANKEN,[13] a simple hierarchy language for writing thesis outlines, and a language for creating syntax definitions.




```
DO I = 1 TO 20;
   !STMT:S = S +!
END;
```

```
DO I = 1 TO 20;
   S = !ARITHX:S + A(I!;
END;
```

Figure 4—Examples of holophrasts
All three examples show the DO loop, but each has been contracted differently. The user may change N, the number of characters of the substring. In the examples, N is seven

USER ENGINEERING PRINCIPLES

The first principle is KNOW THE USER. The system designer should try to build a profile of the intended user: his education, experience, interests, how much time he has, his manual dexterity, the special requirements of his problem, his reaction to the behavior of the system, his patience. One function of such a profile is to help make specific design decisions, but the designer must be wary of assuming too much. Improper automatic actions can be an annoying system feature.

A more important function of the first principle is to remind the designer that the user is a human. He is someone to whom the designer should be considerate and for whom the designer should expend effort to provide conveniences. Furthermore, the designer must remember that human users share two common traits: they forget and they make mistakes. With any interactive system problems will arise—whether the user is a high school girl entering orders or a company president asking for a sales breakdown. The user will forget how to do what he wants, what his files contain, and even—if interrupted—what he wanted to do. Good system design must consider such foibles and try to limit their consequences. The Emily design tried to limit these consequences by explicitly including a fallible memory and a capacity for errors in the intended user profile. Other characteristics assumed are:

curious to learn to use a new tool,
skilled at breaking a problem into sub-problems,
familiar with the concept of syntax and the general features of the syntax for the language he is using,
manually dextrous enough to use the light pen,
not necessarily good at typing.

Throughout the following discussion, reference is made to 'modularity' and 'modular design.' These terms refer to the structure of the program, but have important consequences for user engineering. A modular program is partitioned into subroutines with distinct functions and distinct levels of function. For instance, a high level modular subroutine implements a specific user command but modifies the data structure only by calls on lower level modules. To be useful for the general case, the lower modules must have no functions dependent on specific user commands. In the Emily system, for example, there are user commands to MOVE and COPY text and there are low level routines

```
User Engineering Principles

    First principle:  Know the user

    Minimize Memorization
        Selection not entry
        Names not numbers
        Predictable behavior
        Access to system information

    Optimize Operations
        Rapid execution of common operations
        Display inertia
        Muscle memory
        Reorganize command parameters

    Engineer for Errors
        Good error messages
        Engineer out the common errors
        Reversible actions
        Redundancy
        Data structure integrity
```

Figure 5—User engineering principles

for the same functions. These low level routines always destroy the existing information at the destination, but the user commands are defined to move that existing information to the special fragment *DUMP*. The low level routines must be called twice (destination→ *DUMP*; source→destination) to implement the user commands, but these same routines are used in several other places in the system. Designing adequate modularity into a system requires careful planning at an early stage, but pays off with a system that takes less time to implement, is easier to modify, and can be debugged with fewer problems and more confidence of success.

Specific user engineering principles to help meet the first principle can be categorized into

MINIMIZE MEMORIZATION,
OPTIMIZE OPERATIONS,
ENGINEER FOR ERRORS.

The principles are outlined in Figure 5.

Minimize memorization

Because the user forgets, the computer memory must augment his memory. One important way this can be accomplished is by observing the principle SELECTION NOT ENTRY. Rather than type a character string or operation name, the user should select the appropriate item from a list displayed by the computer. In a sense, the entire Emily system is based on this principle. The user selects syntax rules from the menu and never types text. Even when an identifier is to be entered, Emily displays previously entered identifiers; though the user must type in new identifiers. Because the system is presenting choices, the user need not remember the exact syntax of statements in the language, nor the spelling of identifiers he has declared. Moreover, each selection—a single action by the user—adds many characters to the text. Thus if the system can keep up with the user, he can build his text more quickly than by keyboard entry.

The principle of 'selection not entry' is central to computer graphics and by itself constitutes a revolution in work methods. The author first saw the principle in the work of George[14] and Smith[7] but has since observed it in many systems. The fact is that a graphic display—attached to a high bandwidth channel—can display many characters in the time it would take a user to type very few. If the choices displayed cover the user's needs, he can enter information more quickly by selection. Ridsdale[15] has reported a patient note system used in a British hospital that is based on the principle of selection. In this system, selection is not by light pen but by typing the code that appears next to the desired choice in the menu.

Experience with Emily suggests that keyboard code entry is better than light pen selection because of two user frustrations. First, the menu does not provide a target for the light pen while the display is changing; and second, the delay can vary depending on system load. With keyboard codes, the user can go at full speed in making selections he is familiar with, but when he gets to unfamiliar situations he can slow down and wait for the display. Thus, his behavior can travel the spectrum from typing speed to machine paced selection.

The second principle to avoid memorization is NAMES NOT NUMBERS. When the user is to select from a set of items he should be able to select among them by name. In too many systems, choices are made by entering a number or code which the system uses to index into a set of values. Users can and do memorize the codes for their frequent choices, though this is one more piece of information to obscure the problem at

hand. But when an uncommon choice is needed, a code book must be referenced. Symbol tables are understood well enough that there is no excuse for not designing them into systems so as to replace code numbers with names. In Emily, there are names for files, fragments, display statuses, syntaxes, and non-terminals. Conceivably, the user could even supply a name to be displayed in each holophrast. In practice, though, so many holophrasts are displayed that the user would never be done making up names. For this reason, the holophrast contains the non-terminal and the first few characters of the text—a system generated 'name' with a close relation to the information represented by that name.

It is also possible to forget the meaning of a name, so a system should also provide a dictionary. System names should be predefined and the user should be allowed to annotate any names he creates. The lack of a dictionary in Emily has sometimes been a nuisance while trying to remember what different text fragments contain.

The next principle, PREDICTABLE BEHAVIOR, is not easy to describe. The importance of such behavior is that the user can gain an 'impression' of the system and understand its behavior in terms of that impression. Thus by remembering a few characteristics and a few exceptions, the user can work out for himself the details of any individual operation. In other words, the system ought to have a 'Gestalt' or 'personality' around which the user can organize his perception of the system. In Emily all operations on text appear to make it expand and contract. Text creation expands a non-terminal to a string and the viewing operations expand and contract between strings and holophrasts. This commonality lends the unity of predictable behavior to Emily.

Predictable behavior is also enhanced by system modularity. If the same subroutine is always used for some common interaction, the user can become accustomed to the idiosyncracies of that interaction. For instance, in Emily there is one subroutine for entering names and other text strings so that all keyboard interactions follow the same conventions.

The last memory minimization principle is ACCESS TO SYSTEM INFORMATION. Any system is controlled by various parameters and keeps various statistics. The user should be given access to these and should be able to modify from the console any parameter that he can modify in any other way. With access to the system information, the user need not remember what he said and is not kept in the dark about what is going on. Emily provides means of setting several parameters, but fails to have any mechanism for displaying their values. This oversight is due to a failure to remember that the user might not have written the system. Another such oversight is a failure to provide error messages for many trivial user errors. Even worse, the 'MULTIPLE DECLARATION' error message originally failed to say which identifier was so declared. This has been corrected, but should have been avoided by attention to the 'Access to system information' principle of user engineering.

Optimize operations

The previous section stressed the design—the logical facilities—of the set of commands available to the user. 'Optimize operations' stresses the physical appearance of the system—the modes and speeds of interaction and the sequence of user actions needed to invoke specific facilities. The guiding principle is that the system should be as unobstrusive as possible, a tool that is wielded almost without conscious effort. The user should be encouraged to think not in terms of the light pen and keyboard, but in terms of how he wants to change the displayed information.

The first step in operation optimization is to design for RAPID EXECUTION OF COMMON OPERATIONS. Because Emily text is frequently modified in terms of its syntactic organization, a data structure to represent text was chosen so as to optimize such modification. The text display is regenerated frequently, so considerable effort was expended to optimize that routine. More effort is required, though; it is still slow largely because a subroutine is called to output each symbol. Less frequent operations like file switching do not justify special optimization. Lengthy operations, however, should display occasional messages to indicate that no difficulty has occurred. For instance, while printing a file Emily displays the line number of each tenth line as it is printed.

As the system reacts to a user's request, it should observe the principle of DISPLAY INERTIA. This means the display should change as little as necessary to carry out the request. The Emily DELETE operation replaces a holophrast (and the text it represents) with a non-terminal symbol. The size and layout of the display do not change drastically. Text cannot be deleted without first being contracted to a holophrast, thus deletion—a drastic and possibly confusing operation—does not add the disorientation of a radically changed display. The Emily display also retains inertia in that the top line changes only on explicit command. Some linear text systems always change the display so the line being operated on is in the middle of the display. Because the perspective is constantly shifting, the user

is sometimes not sure where he is. The Emily automatic indentation provides additional assistance to the user. As text is created in the middle of the display, the bottom line moves down the display. Since this line is often not indented as far as the preceding line, its movement makes a readily perceptible change in the display.

One means of reducing the user's interaction effort is to design the system so the user can operate it on 'MUSCLE MEMORY.' Very repetitive operations like driving a car or typing are delegated by the conscious mind to the lower part of the brain (the medulla oblongata). This part of the brain controls the body muscles and can be trained to perform operations without continual control from the conscious mind. One implication of muscle memory is that the meaning of specific interactions should have a simple relation to the state of the system. A button should not have more than a few state dependent meanings and one button should be reserved to always return the system to some basic control state. With such a button, the muscle memory can be trained to escape from any strange or unwanted state so as to transfer to a desired state. In Emily the buttons of the program function keyboard obey these principles. The NORMAL button always returns the entire system to a basic state waiting for commands. Other buttons have very limited meanings and it is almost always possible to abort one command and invoke another simply by pushing the other button (without pushing NORMAL first).

A second implication of muscle memory for system design is that the system must be prepared to accept commands in bursts exceeding ten per second. (Typing 100 words per minute is 10 characters per second. A typing burst can be faster.) It is not essential that the system react to commands at this rate, because interactive computer use is characterized by command bursts followed by pauses for new inspiration. But if command bursts are not accepted at a high rate, the muscle memory portion of the brain cannot be given full responsibility for operations. The conscious brain has to scan the system indicators waiting for GO. Command bursts from muscle memory account for the unsuitability of the light pen for rule selection as discussed under 'selection not entry.'

In addition to optimizing the interaction time, the system designer must be prepared to REORGANIZE COMMAND PARAMETERS. Observation of users in action will show that some commands are not as convenient as their frequency warrants while other commands are seldom used. Inconvenient commands can be simplified while infrequent commands can be relegated to subcommands. Such reorganization is simplified if the original system design has been adequately modularized. High level command routines can be rewritten without rewriting low level routines and the latter can be used without fear that they depend on the higher level.

A good example of command reorganization in Emily has been the evolution of the view expansion commands. In the earliest version, pointing the light pen at a holophrast expanded it one level, so that each of the subnodes of the holophrast became a new holophrast. With this mechanism, many interactions were required to view the entire structure represented by a holophrast. Very soon the system-designer/user added a system parameter called 'expansion depth.' This parameter dictated how many levels of a holophrast were to be expanded. To set the expansion depth, the user pushed a button (on the program function keyboard) and typed in a number (on the alphameric keyboard). It soon became obvious that users almost always set the expansion depth to either one or all. Consequently, two buttons were defined, so that the user could choose either option quickly. Later, the button for typing in the expansion depth was removed and that function placed under a general 'set parameters' command. Further experience may show that only the 'expand one level' button is required. It would take effect only during the next holophrast expansion. At all other times, holophrasts would always be expanded as far as possible.

Engineer for errors

Modern computers can perform billions of operations without errors. Knowing this, system designers tend to forget that neither users nor system implementers achieve perfection. The system design must protect the user from both the system and himself. After he has learned to use a system, a serious user seldom commits a deliberate error. Usually he is forgetful, or pushes the wrong button without looking, or tries to do something entirely reasonable that never occurred to the system designer. The learner, on the other hand, has a powerful, and reasonable, curiosity to find out what happens when he does something wrong. A system must protect itself from all such errors and, as far as possible, protect the user from any serious consequences. The system should be engineered to make catastrophic errors difficult and to permit recovery from as many errors as possible.

The first principle in error engineering is to provide GOOD ERROR MESSAGES. These serve as an invaluable training aid to the learner and as a gentle reminder to the expert. With a graphic display it is possible to pre-

sent error messages rapidly without wasting the user's time. Error messages should be specific, indicating the type of error and the exact location of the error in the text. Emily does not have good messages for user errors. Currently, the system blows the whistle on the 2250 and waits for the next command from the user. Each error is internally identified by a unique number, and it will not be difficult to display the appropriate message for each number.

It is not enough to simply tell the user of his errors. The system designer must also be told so he can apply the principle ENGINEER OUT THE COMMON ERRORS. If an error occurs frequently, it is not the fault of the user, it is a problem in the system design. Perhaps the keyboard layout is poor or commands require too much information. Perhaps consideration must be given to the organization of basic operations into higher level commands.

Emily provides several means of feedback from the user to the system designer. (Though for the most part, they have been one and the same.) A log is kept of all user interactions, user errors, and system errors. There is a command to let the user type a message to be put in the log and this message is followed by a row of asterisks. When the user is frustrated he can push a 'sympathy' button. In response, Emily displays at random one of ten sympathetic messages. More importantly, frustration is noted in the log and the system designer can examine the user's preceding actions to find out where his understanding differed from the system implementation.

'Engineering errors out' does not mean to make them impossible. Rather they should be made sufficiently more difficult that the user must pause and think before he errs. In Emily, time consuming operations like file manipulation always ask the user for additional operands. If he does not want the time consuming operation he can do something else. To delete text, the user must think and contract it to a holophrast. This means that large structures cannot be cavalierly deleted.

A single erroneous deletion can inadvertently remove a very large substructure from the file. To protect the user the system must provide REVERSIBLE ACTIONS. There ought to be one or more well understood means for undoing the effects of any system operation. In Emily, a deleted structure is moved to *DUMP*. If the user has made a mistake, he can reach into this 'trash can' and retrieve the last structure he has deleted. (Deletion does destroy the old contents of *DUMP*.) A more general reversible action mechanism would be a single button that always restored the state existing before the last user interaction. Emily has no such button, but the QED system[16] supplies a file containing all commands issued during the console session. The user can modify this file of commands and then use it as a source of commands to modify the original text file again.

Besides helping the user escape his own mistakes, error engineering must protect the user from bugs in the system and its supporting software. Modular design is important to such protection because it minimizes the dependencies among system routines. The implementer should be able to modify and improve a routine with confidence that his changes will affect only the operation of that routine. Even if the changes introduce bugs, the user will be protected if the designer has observed the principles of redundancy and data structure integrity.

REDUNDANCY simply means that the system provides more than one means to any given end. A powerful operation can be backed up by combinations of simpler operations. Then if the powerful operator fails, the user can still continue with his work. Such redundancy is most helpful while debugging a system, but very few systems are completely debugged and any aids to the debugger can help the user. As an adjunct of redundancy, the system must detect errors and let the user act on them, rather than simply dumping memory and terminating the run. In Emily, the PL/I ON-condition mechanism very satisfactorily catches errors. They are passed to a subroutine in Emily that tells the user that a catastrophe has occurred and names the offending module. Control then returns to the normal state of waiting for a command from the user, who has the option to continue or call for a dump.

A system should provide sufficent DATA STRUCTURE INTEGRITY that regardless of system or hardware trouble some version of the user information will always be available. This principle is especially applicable to Emily where most of the information is encoded by pointers. A small error in one pointer can lose a large chunk of the file. Some effort has been spent ensuring that errors in Emily will not damage the part of the data structure kept in core during execution. But if an error abruptly terminates Emily execution (such errors are generally in the system outside Emily) the file on the disk may be in a confused state. Currently, the only protection is to copy the file before changing it, but there are file safety systems that do not rely on the user to protect himself, and one of these should be implemented for Emily.

Protection and assistance for the user are keywords in user engineering. The principles outlined in this paper are not as important as the general approach of tailoring the system to the user. Only by such an ap-

proach can Computer Science divest the computer of its image as a cold, intractable, and demanding machine. Only by such an approach can the computer be made sufficiently useful and attractive to take its place as a valuable tool for the creative worker.

ACKNOWLEDGMENTS

I am grateful to Dr. John C. Reynolds and Dr. William F. Miller. Any success of the Emily project is due to their persistent advice and encouragement.

REFERENCES

1 W J HANSEN
Creation of hierarchic text with a computer display
Argonne National Laboratory ANL–7818 Argonne Illinois 1971

2 J McCARTHY D BRIAN G FELDMAN J ALLEN
THOR—a display based time sharing system
AFIPS Conf Proc Vol 30 (SJCC) 1967 pp 623–633

3 W WEIHER
Preliminary description of EDIT2
Stanford Artificial Intelligence Laboratory Operating Note 5 Stanford California 1967

4 DEC LIBRARY
PDP–6 time sharing TECO
Stanford Artificial Intelligence Laboratory Operating Note 34 Stanford California 1967

5 STANFORD UNIVERSITY COMPUTATION CENTER
Wylbur reference manual
Campus Facility Users Manual Appendix E Stanford California 1968

6 D C ENGELBART
Private communication
Stanford Research Institute Menlo Park California 1971

7 L B SMITH
The use of man-machine interaction in data-fitting problems
Stanford Linear Accelerator Center Report 96 Stanford California 1969

8 J G MITCHELL
The design and construction of flexible and efficient interactive programming systems
Department of Computer Science Carnegie-Mellon University Pittsburgh Pennsylvania 1970

9 R B MILLER
Response times in man-computer conversational transactions
AFIPS Conf Proc Vol 33 (FJCC) part 1 1968 pp 267–277

10 W J HANSEN
Graphic editing of structured text
in Advanced Computer Graphics R D PARSLOW R E GREEN editors Plenum Press London 1971 pp 681–700

11 W J HANSEN
Emily user's manual
Argonne National Laboratory Argonne Illinois forthcoming

12 J W BACKUS
The syntax and semantics of the proposed international algebraic language of the Zurich ACM–GAMM conference
Proc International Conf on Information Processing UNESCO 1959 pp 125–132

13 J C REYNOLDS
GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept
Comm ACM Vol 13 No 5 1970 pp 308–319

14 J E GEORGE
Calgen—an interactive picture calculus generation system
Computer Science Department Report 114 Stanford University Stanford California 1968

15 B RIDSDALE
The visual display unit for data collection and retrieval
in Computer Graphics in medical research and hospital administration R D PARSLOW R E GREEN editors Plenum Press London 1971 pp 1–8

16 K THOMPSON
QED text editor
Bell Telephone Laboratories Murray Hill New Jersey 1968