1990

CRT-based dialogs: Theory and design

Jonathan Levine

CRT-Based Dialogs: Theory and Design

by

Jonathan Levine

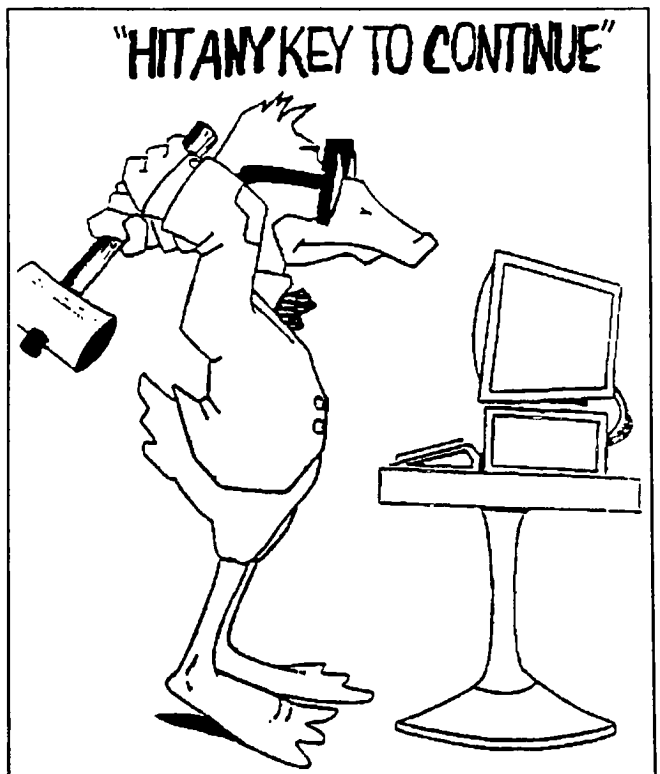


Figure 1: Why isn't this duck smiling?

Rochester Institute of Technology

School of Computer Science and Technology

Computer Based Dialogs: Theory and Design

by

Jonathan Levine

A thesis, submitted to
The Faculty of the School of Computer Science and Technology
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor John A. Biles

Eugene S. Evanitsky

Professor Peter G. Anderson

May 2, 1990

I, Jonathan Levine, hereby grant permission to the Wallace Memorial Library, of RIT, to reproduce my thesis, "CRT-Based Dialogs: Theory and Design," in whole or in part. Any reproduction will not be for commercial use or profit.

Abstract

CRT (cathode ray tube) based, direct selection dialogs for computing machines and systems were apparently a cure for issues like ease of learning and ease of use. But unforeseen -- and probably unforeseeable -- problems arose as increasingly sophisticated systems and dialogs were developed. This paper describes some of the emerging problems in CRT-based dialog design, develops theories about why they occur, and discusses potential solutions for them as a basis for future research. This investigation also provides a survey of the research into what makes programming and programming languages difficult, and what makes them simple.

Table of Contents

Appendices

CHAPTER 1

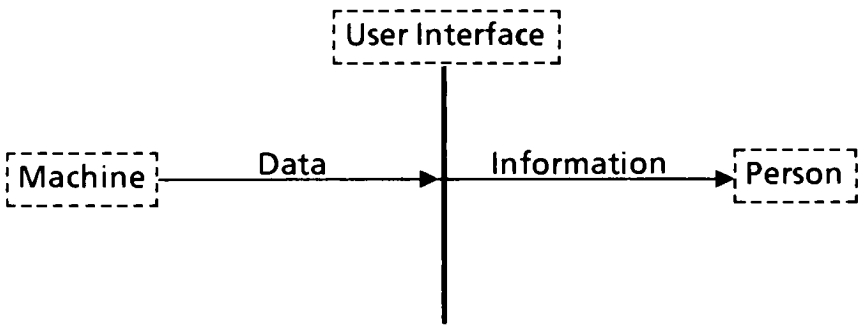
INTRODUCTION

Operational Definitions: User Interface and Dialog

A user interface is a medium that converts data into a useful form and displays it. Figure 2 shows a specific instance of a user interface, a machine-to-human display.

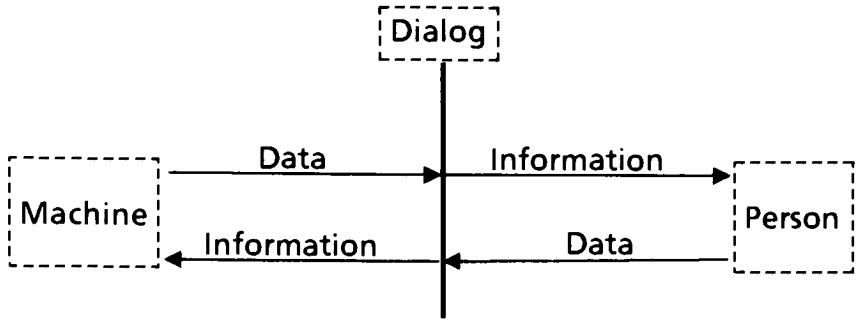
The user interface provides a window into what a machine does, but not necessarily how it does it. Although it's a display, it may actually hide the underlying complexity of the machine -- or even the machine itself -- from the user. For instance, it wouldn't be useful to show someone the gears of a mechanical clock and expect them to easily tell time. Clocks have associated with them a user interface -- the face and hands -- that interprets the gear motion in a useful way and conveys that information visually, and sometimes audibly, to people. When a user interface exists, the human-machine interaction is non-interactive.

Figure 2: A Display



A dialog is a medium for interactive communication, comprising more than one user interface (Figure 3; see Hartson and Hix [p. 8] for a different operational distinction between dialog and interface). When a dialog exists between a person

Figure 3: A Conversation



and a machine, the interaction takes the form of a conversation. For example, when typing into an open computer file, the computer contains a lot of data about the file. One user interface provides feedback about part of the data: that the file is open, where in the file the next typed character will go, a display of some number of lines above and below the current location, the name of the file, etcetera. The user provides data to change the text by adding, deleting, copying, or moving characters, closing the file, renaming the file, and so on. This data, given to the machine via the other user interface in the form of commands, is interpreted and sent to the machine as useful information about what to do. The two user interfaces in this example comprise a dialog. Depending on the computer, the dialog can take many forms. But the meaning of the data is essentially the same.

Operational Definition: Human-to-Human Conversation

A human-to-human conversation is an an asynchronous 6-tuple machine including two concurrent hardware platforms [for a similar argument, see Runciman and Hammond]. The 6-tuple is:

Conversation = (transmitter / receiver, transmitter / receiver, voice, sight, preconceived notions, common generator)

Transmitter / receivers: (the two people having the conversation)

Voice: (hearing, speaking, auditory cues)

Hearing: (speech recognition, auditory cues)
Speaking: (verbal output: words and sounds)
Auditory cues: (verbal inflection (tone), sound recognition)

Sight: (gestures, reading, visual cues)

Gestures: (facial expressions, hand and arm movement, body language)
Reading: (written language)
Visual cues: (color, attire, etc.)

Preconceived notions: (partial knowledge, guesses and implications, context perception, prejudices)

Partial knowledge: (Part of what will be said is known and part unknown [Campbell, p. 68 - 69]; the unknown part is what is learned from the conversation; the known part is what is transferred from previous learning and previous knowledge)

Guesses: (about the other person based on previous knowledge, about the context based on previous knowledge

Context perception: (Interpretations and assumptions about meaning (produced by the generator) of the voice and sight values produced by the generator, assumptions about context of conversation)

Common generator: (a common grammar which generates the structure of the spoken words; also *generates meaning* for the components of sight and voice)

Human-to-machine conversations leave out a large part of this 6-tuple. As a result they leave out a large portion of the data so important to effective communication, and information is lost. But, as we will see, adding more components is not the only

improvement we can make. Human conversation is full of ambiguity, misunderstanding, and intangibles, and adding components doesn't always help.

Operational Definition: Direct Selection Dialog

A direct selection dialog is a cathode-ray-tube (CRT) based dialog that allows machine or system users to control the machine and to get information about the machine's status. The dialog contains a set of "electronic analogs" or "metaphors," pictures of familiar items (see Figure 5, for instance) representing machine features, and "menus," containing lists of other machine functions. All the objects (called icons) and menu items can be selected by touching the desired object (either with a finger or another pointing device, often a mouse). Selection either invokes and reveals the represented function, or makes the icon available to have some operation (eg.: move; delete) performed on it. Many of the objects, which are tools the operator uses to do his job, are on the screen most of the time.

The kind of direct selection dialogs we will examine are asynchronous in that the user has available a variety of tasks, any of which can be chosen by him at any time. "At almost any point in the work on one task, the end user can switch to another task and, later, back to the first . . . . Asynchronous . . . dialogue is sometimes also called e*vent-based dialog* because end-user actions that initiate dialogue sequences (e.g. clicking the mouse button on an icon) are viewed as input events. The system provides responses to each event" [Hartson and Hix, p. 11].

Icons are often coded by color or shade to provide different meaning about their state. An icon that's lit on the screen, when all the other icons are not lit often means that icon has been selected (touched) by the operator. This lit / not lit code is very common but there are many other elements of the graphical code, including the

shape of the icon, the outline of the icon (for example, gray and black outlines have different meanings in some systems), what the icon looks like when it is opened, and so on, that describe the object it represents. The graphical code is, in many dialogs, a source of many interesting observations for it is rich with meaning, and we will look at the meanings of a couple of graphical codes in detail.

Note on a Pragmatic Approach to Dialogs

"The question of whether computers can think is just like the question of whether submarines can swim."

-- (attributed to) Edsger Dijkstra --

"The effort of using machines to mimic the human mind has always struck me as rather silly. I would rather use them to mimic something better."

-- Dijkstra, 1989, p. 1401 --

Can a computer imitate human brain function? Are computers intelligent? Will they ever be intelligent? Researchers intensely argue the following sides of these questions:

Computer / brain identity: At the deepest level, brains and computers are the same. If we can determine the brain's makeup, we can build a thinking machine [eg. Hofstadter, p. 559].

In terms of human-computer interaction, this is the simpler case: Certain components comprise human-to-human conversation. As we include more of these components in machines, we can eventually make true human dialog possible between people and machines. Just keep discovering and adding components until the dialogs are close enough.

Experiential mind:

The human mind and function have their basis in biology, and in cultural, historical, and individual experience [Lakoff]. So machines can never model the human brain. For example, no computer could really comprehend a story about a person laughing unless it could laugh.

From a human-computer interaction perspective, this is the harder case: Since machines will never be able to model human-human dialog, we need to look in new directions for yet-unknown things to facilitate human - computer interaction.

But, regardless of which is correct, *usefulness* always determines the direction computer research takes. Money is spent there because money is made there. And it's not clear that it will be useful to have machines modeling the human brain. For instance, is it useful to have a machine that does math at human speed? Possibly not. But Hofstadter [p. 677-678] implies that a machine with human brain capability might do math like a human, that is, slowly.

So I will use both models pragmatically to investigate what's useful without worrying whether computers can "swim." Can computers really think? The answer, for our purpose, is: it doesn't matter.

The Purpose and Method of Investigation

Lucy Suchman calls it "the problem of human - machine communication." It's a commonplace today (sentiments examined in Figures 1 and 4) that smart machines are not that smart. They don't do what we tell them. They do the opposite of what we tell them. They break at just the wrong time because we pressed the wrong key. We can't figure out how to work them. We can't figure out what they're trying to tell us. And so on. Researchers work hard to find the causes of these kinds of problems and eliminate them, and we've made some progress.

Figure 4: Just another day

But, the latest advances have produced a crop of previously unknown, and probably unforeseeable, questions and problems for dialog designers. These include questions about the relationships between dialogs and

- the nature of work for both individuals and society, including the hows and whys of:
  - what people think about their work,
  - how do people see themselves and their roles,
  - what people think about the machines they use at work,
  - how people learn to do their jobs,
  - how people use their machines to do tasks,

- how people move about in their work space,
- what motivates people to do certain tasks and not others,

  how can using machines be made enjoyable,
- what makes some people move effective than others at their jobs;

the definition and meaning of consistency in dialog-behavior design and dialog-graphic design;

how people come to discover meaning:

what makes dialogs (or particular elements of dialogs) meaningful to users,

exactly what roles do things like context and the interactions between contexts play in providing meaning to users of dialogs;

- what roles artificial intelligence play in making machines easier to learn and use; and,

even such esoteric subjects such as whether emotions are computable functions.

I'll examine some of these in detail -- and some in much less detail -- so that future dialog design efforts that face these questions will at least have a better idea of what to look for than we did. I've tried to suggest possible answers to the questions, or to describe ways in which problems that don't have clear solutions might be managed to the designer's advantage. That this hasn't always been possible is due, in turns, to a lack of currently available technology or of insufficient cleverness on my part.

The overarching theme of this investigation is that, in spite of individual differences in machine operators, and in spite of the different environments they work in, there are psychological similarities, common threads of understanding and meaning that everyone has, and designers can take advantage of these. How future

dialogs can promote ease of learning and ease of use to increase productivity in the face of increasing machine complexity -- that is, **how dialogs can help make machines more useful** -- is the first thrust of this investigation. As the second main theme, I'll describe **techniques for designing these more useful dialogs**.

First, I'll discuss, by example, some of the problems existing in today's dialogs and present assumptions about why these problems exist and why they remain hard problems. The issues I'll address are:

consistency within and between dialogs;

- the meaning of the metaphors and graphic codes in direct selection dialog;

where a user's motivation to learn to use machines comes from; and

- the question of dialog openness and extensibility.

The assumptions are that:

- dialogs and the graphical codes that comprise them are often inherently inconsistent;
- inconsistency is manageable and sometimes very useful; *it is sometimes appropriate to design inconsistencies into a dialog;*

context defines or helps define a graphical code's meaning (by graphical code I mean the various looks and states icons and other dialog elements can take on);

- each graphical code should be minimized (contain the smallest possible number of elements) if they are to be useful in more than one context;
- the number of graphical codes should be minimized; as often as possible, various graphical codes should be combined into a single code, whose various meanings are defined by contextual or other cues;

- in good dialogs, users enforce a consistent meaning onto the dialogs that's more general and useful than the true meaning of the graphics which may be inconsistent; all users attempt to producing consistent meanings, even where there are none; and, since users find different ways of using systems than designers can envision, dialog designs should *enable* users to discover meaning on their own;

- *all* people find certain things easier than other things; discovering what those things are and incorporating them elegantly into a design is a main goal of the dialog designer;

- a good dialog should be invisible (un-apparent) to the user; it should fit seamlessly into his work flow; like a pencil, its use should be so natural that people are virtually unaware of using it;

- machine intelligence is a major aid to making machines easy to learn and use;

- systems should be designed for users rather than having dialogs designed for systems; and, knowledge of the users task and human psychology should be programmed into the functional code of the machine, not only into the dialog; and,

- the entire machine or system, not just the dialog, must help enable ease-of-learning and ease-of-use.

I'll also discuss methods of incorporating these assumptions into future machines. Finally I'll survey the research into what makes programming languages easy to learn and use.

Beyond Direct Selection Dialogs

Direct selection dialogs were a robust and powerful discovery. But no one has taken the next step and the great complexity and power of new machines and systems threatens to overwhelm the direct selection technology until they are, once

again, difficult to learn and use. This paper will produce increasingly general and comprehensive definitions of both "user interface" and "dialog." "User interface" will be redefined as a window not only into machine function but into human function as well, that is, a useful display of how people think and behave. Dialog's definition will incorporate this broader definition of user interface and so, will itself become broader, to include the threads common to all dialog users: processes to generate meaning and motivation, and answers to questions about how to use the darn machine.

In a study of dialogs, one must remember that much of this is soft rather than hard science, in the same sense that psychology is soft. Music evokes emotion but how is not clear. In any case composers are still able to produce delightful tunes and people are still motivated to spend lots of money to listen. Similarly, though none of my assumptions can be proven mathematically, they are useful guides through the art of dialog design. I will not prove the hypotheses I present though I will provide evidence for them as I generate them. My purpose, instead, is to point the way for future research on the next generation of dialog technology.

CHAPTER 2

BACKGROUND

The First Direct Selection Dialogs

The first breakthrough to modern, direct selection dialogs was the Alto, developed at the Xerox Palo Alto Research Center (PARC) in 1972 - 73 as a personal computer workstation [Wadlow]. The Alto's design was "biased toward interaction with the user and away from significant numerical processing" [Thacker et al., p. 549-550]. Each Alto had a bitmap display, a pointing device called a mouse (the now-familiar mouse was invented at Stanford Research Institute [English et al.]), and could run up to 16 tasks concurrently. Another important element relevant to work methods was its connection to a 3 megabit per second Ethernet enabling remote filing, printing, and electronic mail. Available for the Alto were word processors, graphics, and other programs that took a quantum leap beyond previous programs of these types. They were menu driven and the direct ancestor of the iconic direct selection dialogs. The Alto was not intended for sale, but in 1978 Xerox gave 50 to Stanford, Carnegie-Mellon, and MIT. In light of subsequent events, this may have been the most expensive give-away in history.

In 1981, Xerox announced the first product based on the Alto, the 8010 Star Information System [Smith et al.]. Star took the concepts developed and refined in years of use with the Alto and went a step further. For ease of learning, Star provided a desktop "metaphor" on its screen, the graphical equivalent of a real desktop. On the "desktop" were other graphical metaphors, pictures of objects ("icons") that one would normally find on a desk or in an office: documents, printers, file cabinets, file folders. Each icon retained its normal, well known function. In other words, even though it was just a picture of, say, a folder, to the

operator it was used for the *exact* same purpose and in the exact same way as a real folder. *To the user, the icon actually was a folder.* For ease of use, Star provided a simplified command set that was *never* hidden from the operator. Icons opened, with a single command, into "windows," a region of the screen containing the contents of the icon. Up to six windows -- more on later versions -- could be opened on the desktop at a time. With a single command, text, graphics, and icons could be transferred between windows.

In spite of its revolutionary nature, and though it's still being produced under the name Viewpoint, Star was a marketing failure for Xerox due to mismanagement and high price. Somewhere in its development cycle, Xerox showed the Alto to a young entrepreneur named Steve Jobs. As the story goes, Jobs left the presentation with a lightbulb on in his head, took the idea back to his company, Apple Computer, and proceeded with his engineers, some raided from Xerox, to develop the Lisa, the linear ancestor of the MacIntosh computer.

> " 'Office equipment analysts have started referring to PARC-style systems as 'Lisa-like,' not 'Star-like,' ' noted a reporter. 'Apple's next computer, MacIntosh, scheduled to ripen into a commercial product by the end of this year, could further identify Apple with PARC's ideas. The engineering manager for MacIntosh came from PARC where his last big project was a personal computer' " [Smith and Alexander, p. 241 - 242].

MacIntosh popularized the direct selection, iconic, mouse driven dialog [Hartson and Hix, p. 12]. With a low price, strong marketing, and lots of third-party applications, the Mac eventually took off and influenced a generation of software developers and users.

Star still provides the model for all direct selection CRT based dialogs. Today, 10 years after Star was developed, the entire computer industry is attempting to move towards consistent, Star-like environments for their machines and networks. No significant breakthroughs have occurred in dialog design since Star. But, theorizing about dialogs and dialog design has, on the other hand, exploded, and the literature about dialog design and design methodology is massive.

Early Dialog Theory

"There are also no special entities called 'files' in the system . . ."
-- Alan Kay, 1969, p. 127 --

Early theoretical -- some say visionary, but that's yet to be demonstrated -- accounts of what computer based dialogs might look like in the future appeared with Alan Kay's Dynabook concept [Kay et al., 1976] and Nicholas Negroponte's *The Architecture Machine* [Negroponte].

Negroponte's *The Architecture Machine* was, in Brand's words [p. 148], the "founding image" that kicked off the endeavor that became the MIT Media Lab where, today, one can see so many solutions and potential solutions to user interface and dialog design issues. Negroponte saw the machine and the user as partners, not a master and slave, together achieving what neither could achieve alone. Negroponte wrote:

> "Imagine a machine that can follow your design methodology and at the same time discern and assimilate your conversational idiosyncrasies. This same machine, after observing your behavior, could build a predictive model of your conversational performance . . . . The dialog would be so personal that you would not be able to use someone else's machine and he would

> not understand yours. In fact, neither machine would be able to talk directly to the other. The dialog would be so intimate -- even exclusive -- that only mutual persuasion and compromise would bring about ideas, ideas unrealizable by either conversant alone" [p. 12 - 13].

"To accomplish all this, Negroponte posited a high degree of intelligence in the machine, a high degree of intimacy between machine and user, and a rich dialog between them" [Brand, p. 149].

In 1968 [Rose, p. 60], Alan Kay, building on his own doctoral dissertation [Kay, 1969], and Papert's [1970, 1972] and Feurzeig's [1971] important work with children and the Logo computer language [Kay, 1976, p. 8; Rose, p. 61], conceived of the Dynamic Notebook ("Dynabook"), a notebook sized computer that everyone from young children to professional businessmen and engineers would carry with them. It would be simple enough for anyone to use, powerful enough for the most sophisticated applications, and apparently embody a style similar to the one Negroponte envisioned: an intelligent and useful helper / partner [Kay, 1976; Rose, 1987]. His optimistic prediction [Kay, 1977] that this technology would be available and useful in the 1980's has not proven true. Kay still holds onto his influential vision [Kay, 1984; Rose, 1987] but recently he has more cautiously suggested that we still need leaps in both artificial intelligence and dialog technology to achieve a Dynabook situation [Fisher, 1988].

A number of researchers were influenced by *Metaphors We Live By*, Lakoff and Johnson's [1980] thorough examination of the pervasiveness, usefulness, and necessity of metaphors in everyday conversation. Graphical metaphors like icons, desktops, and file folders are standard and useful elements of direct select, CRT

based dialogs. Lakoff and Johnson's work gave an important theoretical underpinning to work that had already begun.

Dialog Design Principles and Methodology

". . . The principles must yield sufficiently precise answers that they can actually be of use: Statements that proclaim 'Consider the user' are valid but worthless."
-- Donald A. Norman, 1983, p. 1 --

In 1983, Halasz and Moran put something of a damper on the theory, if not the implementation, of the usefulness of graphical metaphors in dialogs by publishing a number of examples of cases where metaphors hindered rather than helped users. Their solution was conceptual models -- displaying to the user the internal workings of the system rather than hiding it behind a metaphor. But that required a new method of developing products and a special simplicity and coherence in system design. No one has actually taken that step, but the theoretical foundation exists.

In 1983, three PARC researchers, Card, Moran (the same Moran), and Newell, attempted to take the rapidly developing knowledge in cognitive psychology and create an "applied information-processing psychology" [p. 3]. Their goal was to help move what was known about human cognition systematically into useful applications in human-computer interaction. They attempted to define *how* products should be developed for the user within the technological constraints. In their conception

> ". . . the primary professional -- the computer system designers -- [should] be the main agents of applied psychology. Much as a civil engineer learns to apply for himself the relevant physics of bridges, the system designer should become the possessor of the relevant applied psychology of human-computer interfaces.

> Then and only then will it become possible for him to trade human behavioral considerations against the many other technical design considerations of system configuration and implementations. For this to be possible, it is necessary that a psychology of interface design be cast in terms homogeneous with those commonly used in other parts of computer science and that it be packaged in handbooks that make its application easy" [p. 12 -13].

To help with the definition, they set up experiments to explain *why* what worked worked, and *why* what didn't work didn't. They show, for example, that the mouse is the best pointing device for text editing, and *why* that's true. It was a significant work that helped focus research in human-computer interaction.

Unfortunately, as I've mentioned, no one, except for the Star project designers, has actually designed products by developing the entire system from the users point of view and, perhaps because of this, dialog designs have not significantly changed in many years. Simon [1987] discussed one view of why this happens in his blistering criticism of the human factors profession, "Will Egg Sucking Ever Become a Science?" Simon believes that human factors researchers employ

> "a hodgepodge of quick fixes that evolved over the years into a paradigm that is taught and employed as sacrosanct, when in fact it is woefully inadequate and frequently incompetent when used to obtain the information needed to understand, design, and control complex, real-world man-machine systems. [As a result, engineers,] who used to seek out human factors data, have learned to ignore it. They have learned that our quantitative answers are shams of quasi-precision and require so much qualification that in many cases the engineer's judgment would probably be equally as good without it" [p. 2].

Parenthetically, the article's name came from, the late Admiral Hyman Rickover's response to a proposal for a major human factors program in the research and development of navy ships. Said the Admiral, "[the proposal] is about as useful as teaching your grandmother to suck an egg."

Dialog Design Criteria

In the early 1980's, the Air Force, Army, Navy, and NASA began work on a *Handbook of Perception and Human Performance*. It was an analysis of all the literature on user interface design up to that point. The result was a 3000 page document, divided into 65 subareas and designed as a usable presentation of behavior data to design engineers. The work was summarized in Shaw and McCauley [1984], which presents a good discussion of the major guidelines for (and demonstrated the vastness of the published data on) dialog design. One of the main results was:

> " 'Know thy user' must go beyond mere identification and stereotyping of the user population, especially when these data are obtained through indirect means or logical conjecture. Without direct contact between designer and the user groups, underestimation of the diversity and capabilities of the user groups can occur. Interaction between designers and the users can provide invaluable insights into the differences between designers of systems and users of systems" [p. 51].

In virtually every book I've read on design criteria, there is explicit criticism of the methodology computer programmers use to design dialogs for their programs, and the user models on which their dialogs are based. Heckel's [1984] easy-to-read, entertaining, but deep book develops criteria for designing dialogs based on what

people enjoy, even love, in a variety of other media including film, literature, animation, magic, even the culinary arts. He says: "The general reader need not be afraid that this book is too technical. Indeed, such a reader has a distinct advantage over the computer experts in that he has less to unlearn" [p. xii].

Another example among the vast literature on dialog design criteria, and perhaps more typical of the bulk of the work in this area, is Lin Brown's [1988] straightforward series of recommendations. They include specific examples of what to do and not do in areas of design display formats, nomenclature, color usage, graphics, information systems, status and error messages, system response time, data entry, input and output devices, etcetera. The book contains good recommendations. But, as Card, Moran, and Newell warn, and as we will see, "there is more to human-computer interaction than can be caught with checklists" [p. 8].

I'll also mention here two prominent researchers (though there are many others) who've each published a large body of work: Donald Norman [eg.: 1983, 1988] and Ben Shneiderman [eg.: 1987, 1988]. Both these men have been influential and active in developing, testing, and encouraging what have become standard dialog design principles.

Recent Dialogs

There have been a number of attempts at providing programmers with graphical programming languages. One of these, the Alternate Reality Kit (ARK), was developed at PARC and provides users with the ability to change the laws of physics, gravity, the speed of light, electrical charge in objects, etcetera, and move objects around in the altered world (". . the user . can literally throw the moon into orbit

around the planet" [Smith, 1988, p. 6]). Randall Smith, ARK's creator, has found this useful for doing simulations and for researching how accurate people's perceptions of reality actually are. One interesting element of ARK is that it provides an example of the potentially exclusive nature of ease-of-learning on the one hand, and increased usefulness on the other. I will have an opportunity to go into this question later.

Another graphical programming language is Trillium [Henderson, 1986]. Trillium, written in Interlisp-D (Xerox's version of LISP) and developed in the early 1980's, is an environment in which designers with no programming background can design and test dialogs. It is a dialog for developing dialogs. It was influential and the principles developed by the Trillium designers and users have been incorporated into a number of environments for producing dialogs, including, perhaps, Apple's Hypercard software, and the dialog development language on the NeXT machine.

There are problems with graphical programming. For example, in a videotaped presentation of ARK, Smith points out a large number of graphical objects with complex connections that together look like a morass but that could be written in only a couple of lines of code. And, on the other hand, he showed large amounts of complex code that would be very difficult to develop in graphical form. In both cases, standard programming is considerably simpler. These problems inhibit the acceptance of graphical programming languages and language designers have not yet solved them. In Trillium, it turned out that for dialogs of any significant complexity, users still had to learn to write LISP code. In this case, Apple's Hypercard software has solved some of this problem, making dialogs easier to produce. But, even in Hypercard, complex interactions and behaviors must be programmed by someone with knowledge of programming logic.

In the mid 1980's Henderson and Card [1986] and others [Henderson and Card, p. 213 - 217] found that users of word processors expend significant effort (they "suffer" [p. 217]) continually closing or moving one document window to get information in another, due to small screen size. On the other hand, people doing the same task with pencil and paper tend to spread their documents out on large tables so none are hidden. Using data gleaned from studies of the analogous "thrashing" phenomenon in virtual memory systems, they devised a solution called "Rooms." Rooms is simply a multiple desktop environment where each room is a new desktop. The user can create and set up each room separately for each task. He might create a separate "mail room," for instance, where he only does tasks relating to electronic mail. Each room is significantly less crowded than a single desktop. That means there is more screen space to display the data required to do each task since documents not relevant to that task are in a different room. In a videotaped presentation of Rooms, Henderson claimed that, after using Rooms, going back to a single desktop apparently feels as constraining as returning from a desktop dialog to a line editor.

Future of Dialog Design

For a look at the future of dialogs see Brand's book, *Inventing the Future at MIT*. In it he discusses (with some awe) such obscure and remarkable topics as "sincere eye contact with a computer" [p. 145]. Artificial intelligence will be increasingly important to dialog design: the machines will have to pick up some of the slack as the feature sets explode. And there are many exciting ideas floating around. But, in a way, the yet-unrealized works of the early visionaries, Kay, Negroponte, and a few others (including some works of science fiction), are still the best places to discover

how people will, in the future, interact with computers and computer-based machines.

CHAPTER 3

DIALOG THEORY: PURPOSE AND NATURE, CONSISTENCY AND MOTIVATION

The Purpose of the User Interface

As a teacher of BASIC, I always taught the Print statement first, not because it's easy but because unless one can see a result, the most sophisticated calculations are useless (and, in this sense, every programmer is a user interface designer: Print (along with Input) statements define the dialog for a BASIC program).

At the same time, Print has a natural meaning, and so is an intuitive, simplifying user interface to an underlying, hidden, and far more complex series of commands, the language of the hardware, the machine code.

With this example we can refine our understanding of the purpose of the user interface and the methods user interfaces use to achieve their purpose. User interfaces:

- *hide underlying complexity*, perhaps even entire machines, from the users to decrease learning time and promote ease of use. And,
- they make the *useful and intended results of computations obvious* to decrease learning time and promote ease of use.
- These boil down to increased productivity for the user.

For example, machines would be difficult (high level programming language), impossible (clocks), or even dangerous (light switch) to use without a user interface. In each case, productivity decreases without a user interface. (Of course, it's not humanistic to consider accidental death a loss of productivity. But that's exactly one of the things that companies worry about: it costs less to keep an employee alive.)

The Purpose of Dialogs

Dialogs retain the general user interface goals, ease of learning, ease of use, and increased productivity (speed of accurate use) for the user. However, a dialog is more helpful than the user interface alone in achieving these goals because, as a special case with more defining elements, it is more specific in all aspects. And one of the primary human factors principles in designing for usability is to always be specific instead of general [Heckel, p. 74; Smith et al, p. 248].

Examples: User Interface and Dialog Interaction

The home telephone displays a simple user interface -- an address-specifier (the "dial") and a talk/listen device (usually a handset) -- that hides vast complexity. The network software, the hardware, the path the voice takes, and the math behind it all are invisible to the user. The dialog consists of dialing and listening for tones, ringing, busy, or disconnected-destination messages, and speaking with the operator. Without its address specifier user interface, phones would not be a product. It's the "dial"og that makes it convenient. Notice that over the last four decades the user interface hasn't changed (very much) while the dialogs become increasingly sophisticated[1] and, not coincidentally, complicated.

Programming languages are user interfaces for machine code. Given that there is a minimal set of required elements in programming logic (if-then-else, loop, sequence), it becomes a dialog problem to represent these structures in simpler ways

1. For example, one might need to enter 15 or 20 digits to make certain connections like overseas with a different phone company. Also, parts of todays telephone dialogs are menu based: if you want this function enter a 1, if you want this function enter a 2, now enter your 12 digit ID number, etcetera.

for easier interactions during design, coding, debugging, and testing. Assembly code, BASIC, FORTRAN, Pascal, C++ and graphical environments are all dialog based attempts to enable programmers to more simply use the same old logic. Previously, machine code made programming an obscure art available to a few experts, and even they could not write very complex or large programs without being overwhelmed by complexity. Advances in programming languages made programming a *useful* activity, and money was spent on these efforts for exactly this reason. Today, the user interface is a CRT or flat panel display. The dialog is the interaction between the user and machine on the user interface as code is written.

The standard analog clock is a user interface to time-of-day. Although it's easy to use it's not easy to learn for it is surprisingly complex. For example, the "2" on the clock face is sometimes a two and sometimes a ten, depending on which hand points to it. Sometimes it's even a fourteen, if one is in the right country. This flies in the face of all modern assumptions about user interface design: it is inconsistent and potentially confusing. But we spent years as children learning how to use clocks and we "error handle" the confusing parts unconsciously. So, the designers get away with it.

Note that the time-telling mechanism is pure user interface without dialog for there is no interaction. The setting mechanism, on the other hand, requires a dialog (for more on the meaning of clocks see Smith [1987]).

Now, I also want to extend our definition of user interface beyond machine function and use it in the sense of a window or display of any function. Natural language is a user interface to thought. Each language represents concepts perhaps unknown in cultures using another language. For example, Inuit has, as I recall, 15 or 20 words for snow. Each word describes a different kind of snow. In our culture we

don't usually distinguish between these different forms of snow except maybe between wet and dry snow, and it's likely that most of us don't even know about the other distinctions.

We can take this further and say that mathematics notation is a user interface. For it too is a language and a convenient window to thoughts of a particular kind.[2]

Yet another conceptual user interface is art, a user interface to human emotions. I don't want to be accused of taking all this too far but I want to use this thinking below and later when I examine Walt Disney as dialog designer.

History

Speculatively: the first dialogs were conceptual. Languages were windows into thoughts. Another early conceptual user interface was mathematics, beginning with counting[3] and progressing toward formalism. A third was art, a user interface window -- and perhaps a dialog -- to the emotions.

As language, math, and art seem to be among the defining elements of the human being, we could call these the "natural" user interfaces, parts of the "natural" mind.

Direct selection dialogs are the oldest kind of machine dialogs and were much more direct than they are today. In early industrial revolution machines, water wheels for instance, users dealt with obvious relationships between the controls and the machine: stop pushing and the machine stops; or, pull the lever, watch the gears

2. Perhaps the existence of language is the user interface and the use of the language in conversation is dialog. This sort of broad definition suits language, which itself is so powerful, clear, and also ambiguous.
3. A cardinal number, like a user interface, is an abstraction. It is what remains of a set when you remove the nature of the elements of that set and their order [Cantor, p.86].

disengage, hear the slowing machinery. More recently these relationships were increasingly hidden from the worker. Gears disappeared into "black boxes" or became too complex to understand by looking at them. The machines seemed more mysterious and understanding how they worked became more difficult. Early programmable machines, moving dolls, the Jacquard loom, and various arithmetic engines with their complex mechanical arrangements of gears and rods, are good examples.

Even with the development of electronic computers, direct selection remained dominant for a while. Machine language limited programming to an elite group who directly manipulated the hardware. But it was mind breaking work to code more than small functions. What's more, for the layman, the level of abstraction from the functioning hardware was even greater than merely hiding gears in a box. Now there were no gears, no obvious physical representation of the work-in-progress to conceptualize. The answers to some of these problems were the user-interfaces-to-the-hardware: assembly code, operating systems, compiled languages were all attempts to control the hardware through simplifying, interactive filters, that is, dialogs.

Interestingly, though the designers of the early languages were interested in creating something that was easy to use, the users of the languages, the programmers, were not, and the simpler programming languages enabled a torrent of software that was a boon to industry and replaced billions of secretaries, but was hard to learn and use. So, while the new programs helped to make money, they also wasted money. Difficulty-of-use was not recognized as a problem for quite a while. In 1978, Grace Murray Hopper put it this way:

> "There was also the fact that there were beginning to be more and more people who wanted to solve problems, but who

were unwilling to learn octal code and manipulate bits. They wanted an easier way of getting answers out of the computer. So the primary purposes were not to develop a programming language, and we didn't give a hoot about commas and colons. We were after getting correct programs written faster, and getting answers for people faster. I am sorry that to some extent I feel the programming language community has somewhat lost track of those two purposes. We were trying to solve problems and get answers. And I think we should go back somewhat to that stage.

We were also endeavoring to provide a means that people could use, not that programmers or programming language designers could use. But rather, plain, ordinary people, who had problems they wanted to solve. Some were engineers; some were business people. And we tried to meet the needs of the various communities. I think we somewhat lost track of that too."

Maybe we can attribute this to the fact that these programs made more money than they wasted. Anyway, programmers heard complaints. Eventually they began to listen.

In the early 1970's, Xerox researchers devised the direct manipulation dialog for the Alto computer [Thacker et al; Wadlow] consisting of menu selections on a CRT and a pointing device, the mouse. In the late 1970's, graphical representations (called icons) of the objects the user was manipulating replaced words as the main access element of the dialog display in a product called Star [Smith et al]. Icons are graphical metaphors for familiar objects (Figure 5). They're used to map properties of

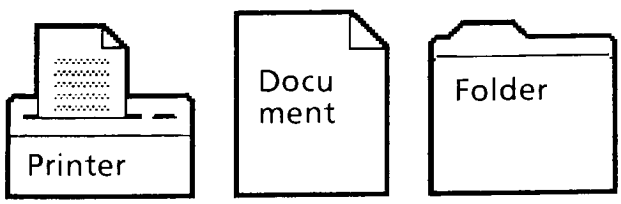


Figure 5: STAR icons

familiar objects (eg. folders inside other folders) into analogous properties of the less familiar objects the computer user deals with (eg. tree structure filing), and to thereby hide unfamiliar or complex elements of the less familiar objects. With Star, direct selection returned as the main method of manipulating complex machines.

The Nature and Function of Machine-Based Dialogs

"There's no sense in being precise when you don't even know what you're talking about."

-- (attributed to) John von Neumann --

We can see this trend as a synthesis of the "natural" (conceptual) user interfaces and the older direct selection machines of the early industrial revolution. Instead of visual and auditory cues directly to hardware, there are layers of abstraction from the hardware. But instead of confusing, abstractions are pressed into service as helpers. They are given physical forms that represent in "natural" terms the work that the user is attempting to carry out as if they were levers directly controlling hardware on an ancient machine. In this sense the abstractions incorporate, rather than hide, the natural human thinking (that is, the things everyone already knows) about the task Like the BASIC Print statement, the iconic abstractions produce a simpler interactive task (a dialog) due to their natural meaning, and each is a simplifying user interface to the hardware.

We're now prepared to produce a more complete definition of "dialog:" **When a machine contains a user interface to the "natural" mind (the things everyone already knows) and a user interface to the system (hardware and software), and when the two interact to achieve the same goals as a single user interface, a dialog exists** (see Figure 3).

A human-to-machine dialog is a manifestation of user requirements, an attempt to model human psychology and behavior in a particular user population, and to link that model to the hardware's function to produce a -- one hopes -- simple interaction. Psychology has advanced a lot but most of it is still mysterious, so I'll heed von Neumann's warning as I proceed.

Macro 5090 Dialog Coding

Figure 6 pictures the Xerox 5090 duplicator dialog. The user touches icons on the CRT to program the machine. File cabinets contain file folders. Each file folder a job type and contains scorecards. The scorecard contains machine features available in its file folder. By selecting a scorecard icon, a work area, containing the programming options for that feature, appears to the right. Each scorecard icon displays its current programming so that the user can always see the programming for the entire job. So, scorecards are menus of machine features, and a feedback mechanism describing what the user will get when he presses Start.

In Current Program, Standard the user can program a standard copying job with features like Reduce-Enlarge, Sides Imaged, Copy Quality, and so on. In the scorecard, the user selects a feature he wants to program and then selects the desired settings in the work area. He repeats those steps for each desired feature. Actually, every machine feature is programmed in this way, so this isn't too interesting. More interesting is what happens to the icons when the user selects them.

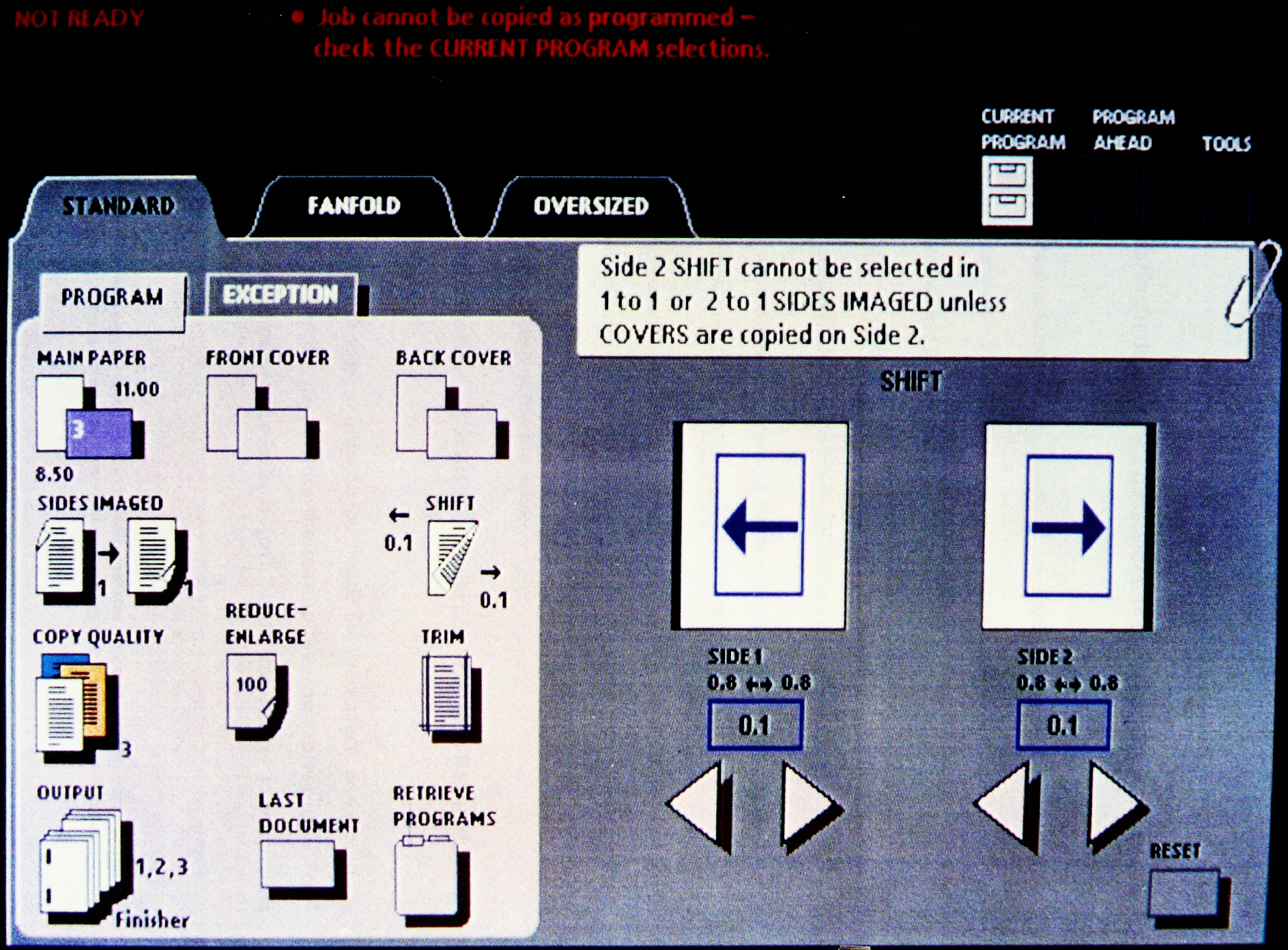


Figure 6: 5090 Screen, Shift work area

Micro 5090 Dialog Coding

Each icon's meaning is defined by 5 dimensions of coding on the 2 dimensional CRT: The icon's

- look (the graphic form used to represent a machine or dialog feature),
- location on the CRT in the macro metaphor (file folder) structure,
- shadow state (visible or invisible),
- color (full color or ghosted), and
- an implicit historical dimension (how it achieved its current state).

The look of an icon depends either on its relationship to the filing metaphor (a file folder tab, for instance) or the machine feature it represents.

We also defined an icon's meaning in terms of its location. As we'll see, an icon in a scorecard has a completely different meaning from the same icon in a work area. The location also helps define the icon's context. For instance, only file folder tabs can appear in the area on top of the file folder.

When an icon has a shadow, it's available for selection. If an icon has no shadow, selecting it has no effect.[4]

An icon can also be full color or ghosted (outlined). On the surface, one would say the full color code means that the icon (or the feature the icon represents) is "on" and the ghosted code means the icon is "off." But the reality is much less straightforward.

4. Actually, selection of two unshadowed icons in a row generates a message telling the user that only shadowed icons can be selected. The message says nothing about why the unshadowed icons have no shadow.

Inconsistency 1: Internal Variation in Meaning

The icon states along the color dimension (full color/ghost) have different meanings in different contexts. A full color

file cabinet means:	"You are in this pathway."
file folder tab means:	"You've programmed the machine for this [the name on the tab] particular class of input and/or output paper."
scorecard tab means:	"You've selected a set of features of this [the name on the tab] category which you can now inspect or program."
scorecard icon means:	"This machine feature is on. You'll get the effect this feature provides if you press Start."
work area icon means:	"You've programmed this value for the currently selected feature."

There are other interesting variations. For example, a ghosted scorecard icon means the feature is off. But often, in order to turn that feature off, a work area icon (usually called "Off") must be full color ("on").

Inconsistency 2: Internal Variation in Effects

Deeper and more interesting variations in meaning occur when we look at the fifth dimension, how an icon gets to a state, instead of just the meaning of a state. For example, different icons reach different states by the same user action. Selecting a ghosted scorecard icon will not make it full color. That's because the user has not programmed the feature by that action and it will not be applied to the job if he presses Start. Touch a ghosted work area icon, on the other hand, and it will go full color since that is the current setting programmed for that feature. Furthermore,

with that work area selection, the related, ghosted scorecard icon goes full color. How an icon achieved its current state can affect its meaning enough to alter user behavior.

Now, the *result* or effect of a user action may be an end in itself (eg. pressing Start, or programming Reduce-Enlarge in a program saved for future use (called Program Ahead)) or a means to some other end (eg. programming Reduce-Enlarge for the Current Program). The latter case incorporates an abstraction, a separate step in achieving the desired result, so its effect on the result isn't necessarily completely clear. Also, that particular medial action may affect or be affected -- perhaps in unpredictable ways -- by other medial steps on the way to the desired result.[5] On the other hand, the former achieves a direct and (usually) complete result.

The distinction between these is important for each imposes different meanings on the same dialog coding of the results of the identical action. This changes the meaning of the feature selection and how it relates to the application of that feature, contributing to a major barrier for novices, the feature application problem. The feature application problem can be characterized by the following example:

> It's one thing for novice programmers to correctly state what a discrete structure like a loop or a conditional is. It is another thing entirely to apply the appropriate structures to algorithms and in the correct order so that the program will solve the problem. Experienced programmers can apply these structures efficiently because they have a large body of knowledge, have it efficiently organized, and can use the knowledge to generalize about solutions and constraints. Students who did poorly in my

5. Take, for instance, the medial step of programming Tabs as the copy stock in Current Program. The 5090 then automatically programs a half inch image Shift without user intervention, so that printing can be done on the tab portion of the Tab stock. Users have told me that they sometimes program Shift by programming Tab stock instead of programming the Shift feature, even if they're not using Tab stock in that job. They do it because in some instances it's apparently easier for them to program in this way.

introduction to programming courses could explain what a loop or a conditional was. But, they had trouble applying those structures to create correct programs.

See the Appendix A for further discussion of the feature application problem.

But it's OK: No one Notices, and Graphical Elegance

This is all a bit disconcerting to the designer because, after agonizing over the coding inconsistencies, it turns out that no one notices them. Like the analog clock designers, we got away with it. And there's a big advantage: "Graphical elegance is often found in simplicity of design and complexity of data" [Tufte, p. 177]. And we did exactly that: by overloading the meaning of some of the coding dimensions without *perceived* inconsistency we minimized the required number of graphical codes.[6] This led to a simple design with a complex message.

We've all been exposed to designs we perceive as inconsistent. That means that the kind of economy of code we achieved on 5090 does not happen automatically. The important point is: there exists a "line of inconsistent meaning." Below the line, inconsistent meanings are not perceived and economy of coding and its corresponding elegance is available to the designer. Above the line, the inconsistencies will confuse and inhibit operator performance.

6. We perceive this kind of economy as elegant because of the surprising psychological connections it makes and because it saves on "psychical expenditure." For a remarkable discussion on pleasure due to psychical economy, see Freud, 1960, especially p 41 - 45 and Chapter IV. Also, see the section on The Nature of Fun, below.

The Location of the Line of Inconsistent Meaning

The 5090 user has to remember only two codes on the color dimension. The meanings in the different contexts are analogous: yes I'm here, no I'm not here; yes I'm on, no I'm not on; yes I'm going to use this input, no I'm not going to use this input, and so on. Now, what we've identified is a more general, more abstract *meta-meaning* of the coding which is easy to remember and, in some natural way, maps to each context: full color = yes. Ghosted = no.

One abstracts meaning from the code and context. Maybe users abstract the meta-meaning from the code and context, completely bypassing the true meaning. This meta-meaning is completely valid and though it doesn't tell the whole story, it reduces the user's memory load because now there are only two meanings to remember instead of two for each context. This makes the code more *useful* because reduced memory load translates into improved performance [see Miller].

The designer's job is, as Heckel points out, to provide the specific meaning, but note that the *user* provides the meta-meaning. In some way the designer has enabled this abstraction to take place. What is the enabler? When the users are given a consistent code like full color, they look for a single meaning to cut across all contexts. In this case the single meta-meaning "yes" was perhaps so natural that novice users provided it unconsciously.

What if the color dimension had three or more code elements for each context rather than two? The user would need to remember a lot more meanings for each context and performance would drop [Broadbent]. Also, it would be harder and less natural than drawing a simple binary (yes / no) meaning from the code.

So, we can say that the "line of inconsistent meaning" is drawn where a minimized (binary) code can represent a general and consistent meta-meaning used in place of the various, specific, and inconsistent meanings found in the different contexts. The inconsistent meaning of the code in each context is made consistent when *the user naturally and unconsciously* discovers the meta-meaning, an act of abstraction from the code and the context. Notice that the line is drawn at the same place that graphics become elegant. Where complex data is coded simply, the user perceives elegance.

Context and Memory Load

Notice the users still have to learn and remember the various contexts. They can't derive meaning from their simple generalization unless they know what they're saying yes or no to. Yet, even with this all the different contexts to remember, users don't seem to have trouble understanding the meaning of the color code in each context. Why?

The answer may be that, on the 5090, the context is not a contributor to memory load because all the contexts are always visible to the users. Graphics and labels provide the meaning of each contextual category. So, there's very little remembering required because the text says explicitly what the context is. And the graphics and metaphor provide a simple way to learn and recall what each context looks like, where it is, and what it is.

Inconsistency 3: External Variation in Meaning (Interpretation)

"Why do people say their alarm went off when it goes on?"
-- Arsenio Hall --

Even when the 5090 dialog provides internally consistent meanings, as with the the shadow dimension (visible/invisible), the interpretation a user makes can muddle the meaning. For example, 5090 users often didn't see the shadow as they learned to use the machine. For these novices, the shadow had no meaning. Some untrained operators interpreted "no shadow" to mean that they had already selected the icon and shadow to mean they hadn't selected the icon.

None of the incorrect interpretations changed the shadow's real meaning: when an icon has a shadow, it's available for selection. If an icon has no shadow, it's not available for selection. But, as people give their own, sometimes incorrect meanings to codes, they provide opportunities for inconsistent meaning. Once again, designers have the opportunity to minimize the chance for misinterpretation. But no one can code data to take all, or even most interpretations into account. Cultural, social, and individual differences combine to produce many various interpretations of a code.

The Consistency Problem: Managing Inconsistency

It's been almost impossible to achieve a definition of consistency. Designing for consistency has been even harder. Now we have some clues about why this is so. Even without external variation due to individual and cultural differences, both micro and macro dialog coding *almost always* provide inconsistent meanings. On a single product it takes a particular kind of understanding to find the "line of inconsistent meaning." Between products the location of the "line of inconsistent

meaning" is even more difficult to find. Recognizing the nature of the problem is the first step to achieving consistent dialogs, but there's no doubt that finding and maintaining the line is today an artistic activity at best. We are dealing with human psychology in areas not well understood.

More importantly, we must recognize that *it is the users themselves who unconsciously provide the meta-meaning* that makes the code's true meaning seem consistent across many contexts. On 5090, the designers provided the code, context, and meaning. The users discarded or ignored the inconsistent, specific meanings and produced a more general (less correct but more useful!), consistent one. So, *the key to consistency resides with the user,* and to design consistent dialogs we must understand what generalizations *everyone* will use, concepts that don't have to be taught.

But more thought yields a deeper conclusion: that inconsistency can be helpful. We've seen that, below the line of inconsistent meaning, inconsistency is not perceived and economy of coding is available to the designer. We've seen the full-color/ghosted code producing inconsistent meanings but enabling an elegance via economic design. The important implication is that, *in some cases inconsistency is useful and can and should be deliberately designed into the dialog.* For it can provide economies that contribute to graphical excellence and, as we will see, contribute to user motivation. The amount of inconsistency and how it should appear is for the designer to decide; if it's carefully done, the users will never notice because they'll create their own consistent meta-meaning, and perhaps enjoy doing it.

Usefulness: Interpretation and Meaning

Machines are made of metal, plastic, mineral, even stone. They move voltage and current from one place to another. It is only our interpretation of these components that make them useful. Designers interpret voltages as zeros and ones. They add many layers of abstractions -- all geared towards usefulness -- one on another, until some apparently consistent and useful abstraction emerges.

As we've seen, consistency is a fuzzy concept and interpretation of the coding rises in importance in determining meaning because people have a need to make sense of unclear things. Yet, in operable dialogs, these interpretations apparently vary little enough so that the abstractions displayed in the dialog remain useful. For example, a word processor that represents a document as a frog will probably be more difficult to use (or, at least, difficult to learn) among a bunch of other animal icons representing machine features, than one using a picture of a piece of paper to represent the same concept.

Of course, a frog icon will produce the same output as the sheet-of-paper icon. I recall my first experience with a word processor: A professor at Brooklyn College was lecturing on the joys of word processing (it was a line editor with embedded commands). It was great, he gushed, and he could perform all these neat functions. It took two years to learn, he said, but now that he's got the hang of it, it's great. When I arrived at Xerox and got my Star machine, I was not enthusiastic. Two years is a long learning curve. But within an hour with no training I was producing at the level the Brooklyn College professor had described as great. And I was not alone. The Star revolutionized the industry because so many people had common rather than competing interpretations of the graphical dialog. This points out that the dialog designer can make a task easy -- if he is sufficiently creative -- or very hard for the

user (for instance, writing in assembly language or Pascal can produce functionally identical programs; but writing assembly code is hard while writing Pascal is easier).

It's also significant that the designer has quite a different interpretation of the operable dialog than the user. The designer understands the system's internal inconsistencies and it bothers him. The novice doesn't notice the inconsistencies. The experienced user understands that the inconsistencies exist but it doesn't interfere at all with his work.

Can Dialogs Solve *Every* Operability Problem?

A dialog can never make a system simpler. And it's here we come to one of the core problems in dialog design. Dialogs are supposed to make systems easier to use. But, various useful elements of dialogs break down as the system they hide becomes more complex. System designers need to build systems with the user in mind rather than the system function. Perhaps this will make systems *inherently* easy to use, allowing dialogs to expose rather than hide the system, thereby extending the dialog's limits in attempting to make the most of the user's capabilities. Given an inherently simple system, the role of the dialog may be to provide applications, the connections between the user's goal and the machine's feature set. I'll return to this point in Chapters 4, 5, and 6, to discuss some outcomes and implications of this statement. I will move in the theoretical direction that says, as much as possible, *it is not the dialog but the machine or system that must enable operability solutions*. For that will help make dialogs simpler in the face of increasing functional complexity.

Easy to Learn and Easy to Use

We grow up learning a variety of user interfaces and dialogs through long-term and sometimes unconscious exposure. This kind of learning is sociological in that most people in our culture learn approximately the same things and can communicate them without explanation.[7] People learn how to use light switches and doorknobs relatively rapidly. The extraordinarily complex analog clock takes many years to understand. And no wonder. Not only is the concept of time very abstract, but also, the clock's user interface is inconsistent: the 1 sometimes represents a 1 and sometimes represents a 5 (depending on which hand is pointing to it). And the 5 sometimes represents 5 minutes and sometimes 5 seconds. This conflicts directly with human factors principles. And no dialog designer would design a clock face in this way today. But the designers get away with it for two reasons. First, the meaning of the display (what is the time now) is so heavily overlearned in childhood that we don't see the inconsistencies. Second, the analog face elegantly and simply represents such useful information that it's a virtually indispensable tool.

Due to the sociological osmosis, we find analog clocks simple to use. And we don't think at all about whether they're easy to learn because all the learning occurred in early childhood. However, the long learning period indicates that they are not *inherently* easy to use -- compared to, say, a light switch -- and that they were quite difficult to learn. We practiced a lot until we got it right.

A light switch, on the other hand, is an example of something easy to learn and use *in the right context.* There's a direct connection between the action and the result, provided the user can see the light change state. But with the light bulb in a

7. For example, try asking someone from China about "Three Blind Mice."

remote location and a 3 way switch, even an experienced user would not know what effect flipping the switch had without extra coding. Let's assume the code was an LED light on the switch itself. Then, an experienced user still could have a hard time, depending on which of his interpretations of the code was correct. Either:

- An "on" LED means the light is off; the designers made it so the switch could be found in the dark.
- An "on" LED means the light is on; the designers want the LED to reflect the state of the light.

Of course, once they learned the code, it should be easy to use.

From this discussion, one might be tempted to conclude that if something is easy to learn, then it's easy to use or that if something is easy to use, then it'll be easy to learn. As usual, it's not that simple.

Easy to Learn *versus* Easy to Use

For example, Apple's MacIntosh runs software that's touted as easy to learn. This is due to the metaphoric nature of both the micro and macro elements dialog and its direct selection characteristics. As with Star, menus and icons reduce memory load and help focus on achieving results rather than on the mechanics of the task. However, once you learn to use the program, accessing commands through menus appears slow and downright frustrating (that is, hard to use). The MacIntosh provides keyboard accelerators to compensate for this problem. The keyboard accelerators are sequences of (usually) 2 keystrokes on the keyboard that will select its corresponding menu function without selecting the menu, displaying the menu,

and then selecting the desired menu item. In this case the menu dialog is hard to use precisely because it's easy to learn.

Another component of ease-of-use is that it should be hard to forget. If someone uses a machine once every 2 months, he may not recall exactly how it operates. In this case it's hard to use because he has to relearn it each time, another frustrating experience. Here again, the machine is easy to learn but difficult to use.

Fun: Dialogs to the Emotions

"Products must stimulate emotion."

-- A. Alessi Anghini --

I've described conventional dialogs and some of their theoretical components. I want to move into an area that might be considered unprofessional were its current incarnations included on machines like 5090. The concepts I'll discuss may be nebulous and rarely noted except jokingly, but its success is so widespread that it's difficult to ignore the message. I'm thinking of *play*, and specifically, play's relationship to motivation.

Play may be the *most* important element for correct dialog design. In environments where complexity and feature set size can always grow but feedback mechanisms cannot, internal user motivation to understand and master the use of a machine may be the ultimate solution to push at the limits inherent in machine feedback and human comprehension. This motivation may arise from play's major useful (as I define it) outcome, *enjoyment*.

Video games[8] are the most dramatic and widespread success of dialog design, and they provide a powerful (some say sinister) clue to how far people will go to

overcome their own limitations of learning and usage when confronted with a dialog of specific purpose. I recall my own initiation into the first video game, Pong, and its descendants, games like Biplane, Space Invaders, and Pac Man. The games were addicting. People spend hours playing and improving, competing against each other and the machine. One very interesting aspect of many video games is that you cannot beat the machine. You can always get a higher score, but, as you improve, the game gets harder. One can *never* win. That fact seems to attract rather than deter people, though our culture's main message is that the individual should win. And though it's often very frustrating, people pay to play, relishing small victories like getting one more point than last time before being blown away, getting onto the vanity board, or getting a better score than a friend. Why does this happen?

Another example of a window to the emotions is Walt Disney's animations [Heckel, p. 173 -184]. These are not, of course, dialogs to machines. But they are revealing for, over the decades, people have found so much enjoyment in Disney's two dimensional creatures that they've spent enough for Walt to build Disneyland and Disneyworld. Like video games, the appeal cuts across a broad spectrum of enjoyers. What is the unconscious mechanism that triggers so much enjoyment and how is it useful for dialog design?

The Nature of Fun

Earlier we found that users, unconsciously and on their own, generate meta-meaning to force consistency across dialog contexts and that one key to consistency in dialog design may be to understand what generalizations *everyone* uses. With

8. I'm using video games as an *example* that dialogs can help generate internal motivation. Many people don't play video games and, men apparently play video games more than women. The point is that this kind of access to motivation exists.

video games we have another example of that kind of key. A lot of people think that challenging computer games are fun. This is widespread in the culture but the things that make it fun are hidden from the player. The games produce self motivation (in their limited domain) to learn to be better by providing challenge and enjoyment through challenge. *What are the common sociological / psychological underpinnings of this kind of enjoyment?* The answer lies at the heart of the question: What is "fun?"

In 1898, Freud (1960) theorized about why people find jokes funny and about why people laugh at them. He hypothesized that a joke establishes a special, previously unknown connection between two ideas. The connection is special by the fact that it is an "economical" connection. That is, the joke connects the ideas with the minimum number of intervening steps. The smaller the original (pre-joke) connection between the ideas, and the greater the economy in making the connection, the greater the surprise and enjoyment when the connection is made (and, as I've mentioned, graphical elegance employs the same kind of economy).

In 1979, Douglas Hofstadter defined an isomorphism as

> "an information preserving transformation . . . . The word isomorphism applies when two complex structures can be mapped onto each other, in such a way that to each part of one structure there is a corresponding part in the other structure, where "corresponding" means that the two parts play similar roles in their respective structures. . . . It is a cause for joy when a mathematician discovers an isomorphism between two structures which he knows. It is often a 'bolt from the blue,' and a source of wonderment" [p. 49 - 50].

The idea that a previously unknown connection between two previously known "structures" may be the source of the perception of beauty in mathematics can be

taken further. In Hofstadter's opinion, the perception of an isomorphism "creates meaning in the minds of people" [p. 50]. Video game developers and Walt Disney apparently found isomorphism(s) between their products and some part of the human mind that deals with fun.

Unlike the Chess addicts who get to win sometimes, and unlike carnival arcades where prizes are awarded, the video game player and the cartoon watcher must create his own prizes in the form of small victories. When one defines one's own rewards, it is easy to attain them and self-reinforcing when they actually are achieved, even when the *definition process is unconscious*. In this scenario, the enjoyment is greater than having outside definition of prizes and rewards because the player feels he has control. The fact that this perception is valid only in a limited way is less important here than the fact that this perception is a self-generated reinforcement, a powerful motivator. Perhaps, one derives enjoyment at video games from meeting the challenge because the self-defined goals of the challenge are attainable.

Why are video games and Disney animations successful dialogs? Because people enjoy having fun, and here they can have fun successfully. And it's not just fun: Hofstadter would say due to the isomorphism it's *meaningful* fun. But let's also remember that it's not the fact that the game exists that produces enjoyment. It's the *design* of the game that matters.

Will Anyone Fund Fun?

Play isn't normally the subject of corporate research and development investment. But perhaps Nintendo has taken the first step:

> "Those of us who've seen Nintendo take over whole households know how insidious this video game can be.
>
> The relentless boink, boink of the computer music, the Mario brothers, turtle soldiers, Bloobers and Koopa Troopas -- they turn the mind to mush.
>
> All along we've suspected it was a plot by Japanese video moguls. Last week we learned we were right.
>
> Nintendo really *is* a plot, though not quite the one we suspected.
>
> The moguls gleefully revealed to *The Wall Street Journal* that the video games were merely a Trojan horse for the product Nintendo wanted to sell all along: video information. . . .
>
> It's a brilliant marketing strategy. Since the game is now in 21 percent of U.S. households, the company has a ready audience for its new product.
>
> If Nintendo had tried to sell the computer first, it would have run into a buzz-saw of competition -- and resistance. Marketing the innocent-looking game enabled it to capture 80 percent of the home video market" [Democrat and Chronicle].

Nintendo's got half the idea. If they can make data base searches as much fun as their games, they'll have it all.

The Consistency Problem Again

It may be that the line at which people complain about inconsistency is the point where the isomorphism, the connection between two similar structures, is no longer perceived or the connection is made in an uneconomical way. If this is true then, by

- discovering the correct isomorphisms between mind function and the presentation of machine function, and
- discovering the unconscious meanings common to all users,

designers may have access to effective economies and may, as a result, be able to provide users with the most elegant graphics for the most enjoyable interface where none of the inconsistent dialogs codes are noticeable and none of the external interpretations have a negative effect. But this is a tentative result.

CHAPTER 4

DIALOG THEORY: APPLICATIONS AND EFFECTIVENESS

Art: User Interface to Emotions

Like finding the "line of inconsistent meaning," the rest of dialog design today is an artistic endeavor. The art behind designing video games and animations provide clues to what dialog designers need to display and corporations need to allow in their dialogs, the faces of their products. Exactly what solutions we can glean from that art is one of the main subjects of this section, with our continuing purpose to investigate increasing ease of learning, ease of use, and user productivity.

The First Three Revolutions in Computer-Based Dialog Design

After punch-tape programming on early mechanical computing machines and switches on the earliest electronic computers there were:

1. Keypunch machines with keyboards providing punch cards for batch jobs,
2. Character based, CRT timesharing terminals with keyboards providing interactive file creation through command line communication with the operating system and line (later full screen) editors through typewritten input.
3. High resolution bitmap, windowing CRTs with mouse for menu and object oriented input on stand-alone computers providing communications with the operating system and applications via graphical metaphors containing menus and metaphoric icon selections, full screen direct selection for graphics and text, and visual inspection of multiple processes.

A Direct Selection, Metaphoric, Graphical Dialog Isn't Enough

The third revolution in user interface design happened due to increased understanding of human cognition in general, and how people think about the tasks they do. It was a powerful discovery, but it has limits. We've investigated some of the problems with direct selection: consistency, uncertainty of meaning of codes and context, memory load of code and context, ease of learning versus ease of use, enjoyment. Before we investigate how future dialogs might mitigate some of these, I want to briefly review one more.

Transfer of Learning

One important tool used by direct selection dialogs is transfer of learning.

> "Transfer of learning occurs whenever prior-learned knowledges and skills affect the way in which new knowledges and skills are learned and performed. When later acquisition or performance is facilitated, transfer is positive. When later acquisition or performance is impeded, transfer is negative. Transfer can be general (i.e., content independent), affecting a wide range of new knowledges and skills, or specific (i.e., content dependent), affecting only particular knowledges and skills within a circumscribed subject matter" [Cormier and Hagman, p. 1].

Viewpoint, **Xerox**'s latest version of Star, achieves transfer by having the same commands applying to every icon, text string, and graphic the user can access. The meaning of a command can vary slightly but, as with the 5090, this doesn't seem to bother the users. Also, each mouse button usually provides the same meaning: the left selects, the right extends a selection. And, when it comes to text strings,

Viewpoint provides simple commands for font size and weight, case, underlining, etcetera.

But, transfer of learning between tasks occurs best between highly specific, similar tasks [Gick and Holyoak, p. 11]. And users not only engage in many various and diverse tasks, but they also apply features to tasks in ways the designers never thought of.

For instance, on the 5090 transfer also breaks down once you get to the work areas themselves. You always get to the work areas in the same way but once you get to the work areas they're all different because they represent programming for completely different functions.

And, in another 5090 example, a user normally selects a paper stock on the screen (Figure 7) corresponding to the paper placed in the paper tray. That task is, on the surface and in the dialog, completely independent from the image Shift feature (Figure 6) which the user programs when he wants the image on the original to move right or left on the copy. However, when the user selects Tab stock, the machine provides a shift of 0.5 inches so that, using normal size originals, the user can image onto the tab part of the tab stock. Although Paper Supply and Shift are programmed by the same method, scorecard selections followed by work area selections, in this case the user's programmed Shift in an inconsistent way. Interestingly, experienced users use this fact to their advantage, selecting tabs when there is no Tab stock to achieve a known shift. But the main point is that where transfer of learning occurred for novices through scorecard-work area selection procedure, at deeper levels, this transfer breaks down: you don't go to the Shift work area in this application because it's easier and faster to shift in *another* way.

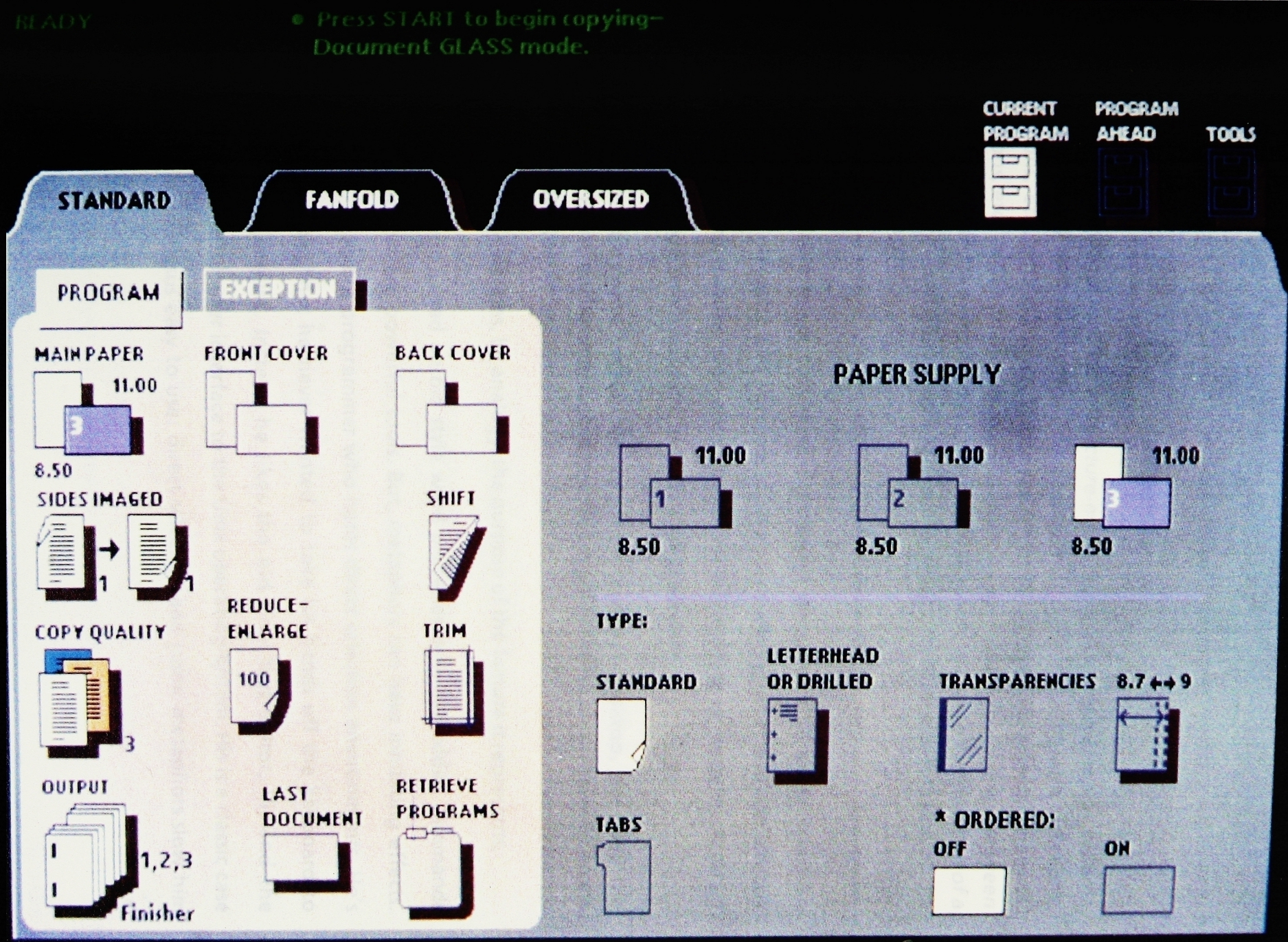


Figure 7: 5090 Screen, Paper Supply work area

Metaphors

"An analogical model involves a structure-mapping between two complex systems, one known and one unknown. The analogical model maps objects, relations, operations, etc., in the known source system to corresponding objects, relations, operations, etc., in the unknown target system. An example is the analogy between a filing cabinet and a computer file system. Here the basic structure and function of a filing cabinet with labeled folders is mapped onto the structure and function of the computer file system"

-- Halasz and Moran [p. 383] --

Metaphors are a major tool enabling transfer of learning in direct selection dialogs. We use metaphors to enhance learning by mapping what the user already knows into an analogous area about which he knows little or nothing. A direct selection dialog generally contains two classes of metaphor.

- The "environment" metaphor (eg., the Viewpoint desktop, the 5090 file folder). The environment gives the user a general idea about how to access and manipulate the machine features.
- Each icon describes, by analogy, the purpose of the feature it represents.

As anyone who's used an old style word processor with embedded commands knows, metaphors are powerful tools. But, metaphors can have inhibiting effects. For example, I know a programmer who hates direct selection environments. He's such a good typist that he never wanted to take his hands off the keyboard to manipulate the mouse. Instead he likes the old Unix text editor, Vi, and the command line editor user interface to the Unix operating system. His is a classic case of easy to learn versus easy to use: direct selection and iconic metaphors slow him down.

During the initial excitement over direct select, metaphoric dialogs, Halasz and Moran (1982) published "Analogy Considered Harmful." In it they argue that analogies actually inhibit learning.

> "Consider a user who wants to file a document under a new name. Using the filing cabinet analogy, the user would reason as follows. In the filing cabinet system, there are two methods to "rename" the document: (1) relabel the folder in which the document resides or (2) make a copy of the document and place it in a new folder labeled with the new name. In the first method there is only a single copy of the document in the cabinet, while in the second method there are two (at this point identical) copies in different places in the cabinet. Since the computer is analogous to the file cabinet, there must be two analogous procedures on the computer: (1) rename the file (resulting in a single copy) or (2) copy the file to a newly-created, newly-named file (resulting in two files with identical contents). Note the order of the user's reasoning. He thinks in terms of the already-well-understood filing cabinet system and then maps the results into the to-be-understood computer file system.
>
> So far the filing cabinet model seems to work well. But consider how you would explain the concept of file-based password protection in terms of the operation of a filing cabinet system. There is not natural corresponding operation or structure in the filing cabinet system. There are two approaches to fixing up the analogical model. For example, you could evoke the notion of a file cabinet in which each individual folder had its own combination lock. Of course, you would have to make similar provisions in the model to explain other computer file system concepts such as directory structures, file-to-file links, and undeleting previously deleted files. At this point, what has become of the filing cabinet model? After all the special addenda have been tacked onto the analogical model, the filing cabinet is no longer a familiar filing cabinet. Further, the addenda are the most important parts of the model. . .

> The second approach to fixing up the analogy is to invoke a whole new analogical model to explain password protection, e.g., a model of a guard whose duty it is to retrieve documents from the filing cabinet for you, but only after you have given the correct password. Notice that this model is only loosely integrated to the filing cabinet model. The user, faced with multiple analogical models of a single device, has the task of deciding which analogy is relevant in the situation at hand. Reasoning about guards when he should be reasoning about file cabinets could leave the user far afield.
>
> But even without the problem of multiple or baroque models, the filing cabinet analogy is problematic. The filing cabinet system contains many aspects which are irrelevant to the analogy [more examples here] . . . .
>
> The point made by this example holds in general for all analogical models of computer systems. An analogical model is, by definition, a partial mapping to the computer system it is supposed to explain. No simple analogical model is sufficient to completely explain the operation a [sic] computer system."

And, they conclude after more discussion, "analogy is dangerous when used for detailed reasoning about computer systems." So, a major enabler to transfer of learning has very similar problems with effectiveness as transfer itself. But how are we to interpret this result when dialogs have been so successful?

The answer again lies in the fact that the down side has not been fully recognized. Though hard-to-use software was making more money than it was wasting, it was still wasting. Similarly here, the full capabilities of dialog design haven't yet been realized. Plus, more and more features arrive on the scene daily, overwhelming the architecture and usefulness of dialogs.

Physical Openness

This brings us to the problem of open dialogs. Before CRT-based dialogs, feature proliferation meant larger control panels. CRT-based dialogs enabled layering, where an unlimited number of features could occupy the same space, just not at the same time. It seemed for a while that the problem of feature control panel growth was solved. But, once again, complexity reared its head. On the 5090, for example, finding space to program each feature was no longer a problem. As the feature set increased, the problem became finding space for *access* buttons to each feature's programming area (that is, we ran out of room in each scorecard and we ran out of room for scorecard tabs to access new scorecards). *And*, we ran out of space to put in more categories in which to place new scorecard tabs (that is, we didn't have room for more file folder tabs). Then, we didn't have room for more file folders (we ran out of space for new file cabinets). 5090 is an example of a physically closed dialog. Its feature set cannot be expanded due to physical space constraints.

The problem of physical space constraints has a number of interesting dimensions. I'll briefly discuss two of them. First, the problem was partly caused by the way we categorized the features. For example, we could have produced a single scorecard icon (machine feature) called Image Motion. It could have associated with it three layered work areas contained the current Trim, Shift, and Reduce-Enlarge features. But, there is a tradeoff in this scenario with number of layers. There are too many and the user would get lost navigating through that many hidden layers. Alternately, Image Motion might reside in a single work area containing Trim, Shift, and Reduce-Enlarge and use *a* mouse input. Each function could be programmed based on the mouse button used: select an animated iconic representation of an

image on copy paper. Left button changes Shift, right button changes Trim, chording changes Reduce-Enlarge. In this way we achieve a useful dialog and reduce the number of scorecard icons by two with a single recategorization of features into a higher level, meta-feature (and a completely different input device).

Another part of this problem, to use the 5090 example again, was the metaphor itself. Note that a file folder has tabs attached to it. This means that, when there's no more room for another tab, there's no more room for another category at that level and another higher level category must be created *if* there's room there. If buttons in the file folder tab category were not constrained to being tabs connected to the file folder, then this problem can be addressed cleanly. For example, if we had buttons pertaining to input / output stock type (in 5090 represented by the file folder tabs) instead of the metaphoric tabs, we could have enabled one of them to say "more," giving another *layer of buttons*. That is, instead of just layering programming regions, we could have layered access to the regions as well. But, as it is, the file cabinets perform the "more" function for the file folder tabs. And there's no room for more.

Conceptual Openness: Modes

Another aspect of the openness question deals with user perception of openness. One form of conceptual openness deals with enabling product extension features to appear consistent with the original feature set. This is not that interesting. Sometimes it involves providing enough memory or physical screen space (by addressing the physical openness question).

A more interesting form of conceptual openness appears with the question of modes. A mode is a state, lasting for a period of time, during which various commands change their meaning.

Modes constrain a user at a given time and therefore close his options. Reducing the number of modes and the number of times they occur reduces memory load. Viewpoint and 5090 are examples of a dialog whose modes have been reduced a lot. If you're editing a document, you don't have to explicitly leave the edit mode (and therefore you don't have to *remember* to leave the edit mode) before doing another task. But no system is completely modeless. For instance, in Viewpoint if you want to move a document you select the document and then press the "Move" button. At that point, until you place the document in its destination, you are in the "Move" mode and you can't do anything else.

Leaving the Move mode is simple. Just click the mouse on any valid destination and the document moves there. Or, clicking on an invalid destination or pressing stop cancels the move command. One almost *has to* leave the mode so memory load is not a big problem here. The important point, though, is that below a certain line, modes in some form are necessary.

Modelessness and its Internal Contradictions

Conceptual openness contains within itself some strange contradictions. For example, producing a truly modeless dialog could provide the user with more complicated operations and increased memory load. Imagine, in the Viewpoint example, that one can perform any operation between the Move key selection (entering the mode) and before the destination selection (exiting the mode). The

user suddenly has a large increase in memory load. Let's say he initiates five move operations in a row, each interrupting the previous. At that point he interrupts the last move operation, opens a new document and begins editing it. When he'd done editing, he begins completing the move operations (the fact that we can correctly use the phrase "begins completing" indicates something a little complicated or unnatural is going on here) by touching five valid destinations. This means he must remember the order in which he selected the documents to select the correct destinations. Or, Viewpoint must provide a list of the documents and the order of the move operations that were interrupted. Neither is elegant.

Another complication arises in a system where a document can be moved onto another document to merge them (which is sometimes legal in Viewpoint). In the above example, selecting the document to edit does not suspend the fifth move operation. It completes it. But what if the user actually wanted to open the document to edit it? The solutions to these problems are not elegant either.

So, though modes can cause problems, they are important and can be included safely in dialogs [Smith et al, p. 278]. The key is that, once the mode is entered, exiting the mode should be the only natural option. The exit should be an implicit outcome of the next logical step in a command sequence. Again we see the effects of economic design with a mode exit compressed into a command. The fact that most user's have no trouble with this and learn it very fast demonstrates again that "naturalness," the things that everyone already knows beforehand, exists and can be employed by designers.

And, here again, we see that it's sometimes advantageous to design inconsistency into the dialog. Having modes in a limited form is useful so bringing them into the

dialog, with all the inconsistency that implies, is better than leaving them out. And, once again, the inconsistency can be managed so the user doesn't notice.

The Process of Extracting "Natural" Meaning

Users might understand much more about a dialog and its operation if they understood the specific meanings of the codes rather than the general meaning they provide. But we've already seen that inconsistencies inherent across a code's specific contextual meanings pressures users to ignore specifics and extract their own general meanings for their own purposes. On the natural mind side of "dialog" we've seen a number of abstractions and generalizations: meta-learning, metaphors, fun. In each case it's the user who overcomes inconsistencies by abstracting general meaning from codes the designers provide. The important point is that *all* the users do it. That's what makes the dialogs useful. And it indicates that the users all have something in common in their heads. I'm not saying all users extract the same meanings from a code. But they all engage in some *process* by which they develop general meanings. Can designers enable users to provide *specific* meanings for their own purposes without inducing confusion?

Abstract Meaning versus Specific Meaning

Apparently, the answer is "no." As I discussed earlier, for the user to provide general meanings, he requires a limited number of elements in each coding dimension. And, the general meaning the user extracts must, in itself be simple enough to be natural -- that is, every user must be able to produce such a meaning. And, the meaning must be general enough to cut across all the contexts where the

particular code dimension resides. These are all limiting factors. To provide specific meanings the user would need more data because they don't draw specific meanings from the data we provide today. But more data -- more coding dimensions, more elements in each dimension, more layers, etcetera -- would increase memory load and dialog complexity and would reduce performance.

Also, remember that not all users extract the same general meaning. Instead they engage in a process which provides *some* useful meaning. But, if each user provided specific meaning for each code in each context, many would operate the dialog incorrectly for their specific meanings would not correctly apply to the specific machine features.

Since this isn't useful, I won't pursue it further. But then what tools can we use to increase performance as machine features and functions increase in number and complexity?

Metacognition

Recall that researchers developed direct selection dialogs as an outcome of an understanding of human cognition. Since designers and users solve problems with abstractions, perhaps abstracting from cognition can generate a new and even more useful class of dialog.

"Cognitive responses, broadly defined, involve manipulation of verbal or symbolic information such as words and concepts. Metacognition refers to understanding of one's own cognitive behavior involved in the planning and monitoring of performance, and in the use of cognitive strategies" [Cormier and Hagman, p. 6]. Metacognition may provide a way for thinking about the process of

abstraction, eg., what does the user actually *do* to generate meta-meaning? How does the user produce higher level categories? How is this knowledge useful to the dialog designer?

Our current definition of "dialog" is inadequate to handle this last question because we are no longer talking about individual elements of a dialog, the user interfaces to hardware and mind. We are talking about the relationship between them, and how the user connects them to complete a task. What is the connection?

Refining the Definition of "Dialog"

A dialog's purpose is to make the user interface to the system work with the user interface to the mind to achieve user productivity. But we've left out the link between the system and the mind for a given task: the application. I'm using "application" in a very specific sense here:

> The application is the relationship between the task the user needs to do and the machine features he will use to do that task. That is, I'm combining two senses of the word application. The first is the task itself and the second is the features the user applies to accomplish the task.

"Application" deals with the user's physical behavior. It links the user's thinking about the task with the machine's feature set and finally gets the job done. "Application" is what makes cognition and machines *use*ful.

So, let's further refine our definition of "dialog:"

When a machine contains a user interface to the "natural" mind and a user interface to the system and a user interface to applications, and when the three interact to achieve the same goals as a single user interface, a dialog exists.

Metabehavior, Metacognition, Efficiency, and Applications

I worked in a large New York City restaurant for two years. The first three months I sliced up my hands for I'd no experience working with knives. Once I got that under control, I could prepare food for lunch service, serve lunch, and then prepare food for the dinner service during my 8am to 5 shift. About 14 months into the job there was a major change. I thought about it a little at the time but now it seems to be of critical importance in my current profession. Suddenly I was able to do both lunch and dinner preparation before lunch service. I say "suddenly" because the change occurred in about a week. At first, I noticed small improvements in my efficiency. These interested me but more important, they excited me. I felt I was becoming the best at my job. And as I discovered each new efficiency, I felt great and I'd tendentiously look for more. Lunch service was from 11:30 to 2:00. My own lunch break was a half hour at 11. So I turned a six hour job into a three hour job. I had become an efficiency expert.

First I learned to apply the tools I had -- knives, mixers, bowls, stoves, sinks, etcetera -- to the task. Then, at some critical mass of sufficient knowledge and motivation, I discovered more effective applications, relationships between the tools and the tasks. The methods I developed for myself were mainly physical: place the juice container here instead of here, move the knife in this way, place the garlic here while cooking the shrimp and the oil there, wash the lettuce and make mayonnaise at the same time, slice all the vegetables one after the other (and many many more;

one can see why it took 14 months to integrate all these jobs). The methods comprised a metabehavior, a way of moving around the kitchen where no motion was wasted. It was a synthesis, a higher level behavior than before.

As I moved to other jobs, I discovered that my experience in the restaurant had transferred. For example, I taught high school math in New York City for three years. By that time I was consciously aware that I was searching for economies. And I found them. Though specific parts of the restaurant job, cleaning squid and so on, did not apply to teaching, what *did* transfer was the knowledge that

1) economies exist and

2) methods of finding the economies exist.

I deliberately searched for compression of tasks and concepts into higher levels. After two years I developed a scheme where I'd reduced Algebra to two concepts that I taught each day for the entire year. Then, without any review classes at the course's end, my students achieved the highest passing percentages in the school on the New York State Algebra Regents exam with a mode of 100% and a mean of 92.

Here, I was using metacognition. I was thinking about how to think about my behavior and thoughts, their meaning, and their applications to the task of conveying algebraic meanings and techniques.

Note that, though the efficiencies of the restaurant were mainly physical, behavioral ones, they transferred to a search for conceptual efficiencies. I engaged in a conscious process to find increasingly general concepts, and meanings. So, though the 5090 users engage in an unconscious process to create general meaning, it is possible for the process to be conscious. But it is much more important that *I was*

able to learn the process because that means it can be taught. And that means we can make it useful.

Creating Novice Efficiency Experts

I want a novice user to be able to walk up to, say, a 5090 and immediately know the most effective and efficient programming to do his job.

Recall that, a dialog is effective when the user generates the meanings. Yet it took me over a year to learn the restaurant efficiencies and 2 years to of deliberately trying to learn the algebraic ones. So, to achieve our goal of having novice efficiency experts (or should I say: minimizing the time a user remains a novice), we need to pursue three questions:

1) How can designers minimize the time it takes to search for the most effective and efficient job programming solutions?

2) How can designers provide users with the knowledge (either conscious or unconscious) that they should look for efficiencies in the relationship between tasks and machine features?

3) How can designers get users to motivate themselves to engage in the search for efficient methods?

The answers that come to mind are:

1) make the applications obvious by presenting them up front in the dialog,

2) enable the machine to, in a certain sense, understand the user's behavior, plans, and methods in completing a task, and

3) provide the user with enablers to motivate himself.

Eliminate Abstractions from the Applications User Interface

Recall that part of a dialog designer's job is to build in layers of abstraction from the system and the natural mind. A user interface to applications requires the opposite: eliminating abstractions and displaying the *exact* application the user needs. That is, the user should see a direct rather than abstract connection between machine features and the task. The reason is because, if the dialog presents an application up front, the user won't have to figure it out. Let's look at three possible methods.

First is for the designer to know all the user applications and have them resident in or loadable into the dialog and present them up front with some categorization scheme. In this scenario, the designer provides the applications. But we've already seen that effective dialogs enable users to supply things themselves.

The second possibility is for the machine to recognize the user's application and present the most efficient one, with the user and machine learning from each other's discoveries. How this can be accomplished is not clear to me. Plan recognition and other artificial intelligence strategies might help if they mature. The main point is that this is a direction that researchers could investigate.

Third, and most effective, may be a hybrid of the two where the novice sees the most effective applications provided by the designer. As the user learns and finds new efficiencies, the system tracks and understands that user's newly discovered applications, displays them, and attempts to provide clues to further efficiencies as it learns the new ones.

Adding more automation to this kind of set-up could help resolve problems of meaning generated by medial programming steps (see Chapter 3, Inconsistency 2: Internal Variation in Effects, above). For instance, perhaps programming the 5090 feature by feature could be replaced by a single button press for a particular application. The machine would then program the features for that application and begin running (that's the same as invoking an agent; see chapter 2, above).

Programming Metacognition

Over the past few years I've seen a number of CRT displays containing objects which looked, when nudged with a mouse, as if they were moving according to the physical laws we all know and love, thus: a ball made of jello and floating in a transparent cube, undulates and bounces just as expected when the cube is moved by the user and its sides touch the ball. In displays like this, the laws of physics are programmed into objects and the environment in which they reside.

Similarly, it may also be possible to program metacognitive principles into dialogs. "All" that's required is a better understanding of human psychology. But if, in the future, this could be done, the structures underlying human thought should be programmed not just into the dialog but into the machine's functional code itself, the code hidden from the user by the user interface to the system. This idea is a direct outcome of our definition of "application." The user's interpretation of this connection between the machine's features and the task at hand affects how he will set up the machine for the task. Some will do this work more effectively than others. When the machine and its functions can understand how and why a person has programmed those functions, then it may be able to adjust its features according to the (novice or inefficient) user's needs, rather than to what the user has done. Then,

via the dialog, it could provide users with what it knows about efficiencies relating to that task, enabling, if not ensuring, learning and growth. I'll return to this point again in Chapters 5 and 6. Here, as I mentioned in Chapter 3, I'm moving toward the idea that the machine, the entire system, and not just the dialog, must enable operability.

Art: User Interface to Motivation and the Definition of a Useful Dialog

In describing my learning experiences in previous professions, I said that I was able to learn efficiencies because I reached a point where I had sufficient knowledge and motivation. We've look at a couple of ways to increase knowledge; but what about increasing motivation?

Motivation can be the product of necessity. For example, there are many people motivated to work at jobs they don't like for economic reasons or because they don't see an alternative [Terkel; Kozol]. Necessity implies a forced circumstance where a person does something due to external pressure and trades off other, possibly happier, circumstances or behaviors. But though force can change behavior, it doesn't necessarily change minds or make the condition more palatable.

On the other hand, motivation can be self-generated, the outcome of fun, happy, or exciting emotions as with video games, cartoons, and challenges like attempting to become the best; or, motivation may be the outcome of quieter emotions like feeling good about trying something or feeling proud about succeeding. In these cases motivation is internally generated because more feels good.

I think incorporating playful or whimsical elements in a dialog is one method designers can use to enable users to motivate themselves. Another method is to

allow the users to be successful. That is, provide the novice different expectations and methods of accomplishing tasks than the expert. Provide simple, achievable goals that demonstrate progress and also lead the user to further understanding, investigation, and uses of the machine.

Note the abstraction here: the designer does not provide motivation. He provides enablers to motivation. Through these the designer opens a window onto the hidden psychological process, motivation, and the operator motivates himself.

Now we can define a *useful* dialog by including a motivation generator, a user interface to motivation:

When a machine contains a user interface to the "natural" mind and a user interface to the system and a user interface to applications and a user interface to motivation, and when the four interact to achieve the same goals as a single user interface, a useful dialog exists.

CHAPTER 5

DIALOG THEORY: MACHINE INTELLIGENCE AND FUTURE DIALOGS

Future Dialogs and Artificial Intelligence

". . . we, the Architecture Machine Group at MIT, are embarking on the construction of a machine that can work with missing information."

-- Negroponte [p. 119] --

"All the human engineering in the world will not turn a 10-character-per-second teletypewriter into a high resolution graphics terminal"

-- Card, Moran, and Newell [p. 8] --

A machine that can calculate correctly without all the facts is part of the ideal machine that can think and communicate like a person. Workers in the field of Artificial Intelligence (AI) are interested in how people think, see, hear, interpret visual and auditory signals, make conjectures, use common sense, and in other ways gather knowledge from and interact with the environment. They claim that, since people already do these things so well, if they can find out how we do it, they'll be able to figure out how to make machines do it too. The problem is hard because there are so many variables in human thought that it will take a long time to understand what they are, understand their interactions, and then produce a machine that uses those variables and interactions. Neural net researchers attempt to achieve the same promise by attempting to model human brains more closely than do the standard von Neumann machines. This is all very important work. Machines will need to understand about human psychology and be powerful enough to do the required computations, and AI holds many potential solutions to user performance and other dialog issues (in the next section I'll describe a couple of examples related to voice).

In this chapter I present my ideas about what will comprise future dialogs. I'll argue that the next revolutions in dialog design will occur with an understanding and application of metacognition, and that artificial intelligence can remove many obstacles dialog designers face today. One concept is that intelligent machines (not dialogs) will teach users -- via the dialog -- how to learn new strategies and tactics for applying the machine's feature set; the user, just by virtue of doing his job, will implicitly *teach the machine* (not the dialog) how he applies features, what he knows and what he needs to learn; and, in the same way, the user may even teach the machine better methods of working. That is, the user and machine will be partners, continually working together, building on each other's knowledge and experience, teaching each other to improve the work process. Dialogs, on the other hand, will provide seamless interactions between the user and the machine. The user won't even think about the machine except in the same way he might think about a pencil: as a tool that seems so natural to use that one doesn't need to think about using it; one just uses it.

Negroponte's team, since evolved into the MIT Media Lab, has yet to fulfill its goal. This chapter is about why they and other researchers have so much trouble reaching it and the kinds of things they may need to investigate.

The Fourth Dialog Revolution: Adding Conversation Elements

In Chapter 1, I described human-to-human conversation as an asynchronous machine running on two independent platforms. Perhaps the simplest way to simplify human-to-machine conversation might be to add more of the components that make up human conversation into human-to-machine dialogs.

Voice is the the element of conversation closest to usefulness today. Speech synthesis has been around for years and speech recognition is improving rapidly. Adding vocal commands, for example, to a word processor program could reduce or eliminate much of the hand motion off the qwerty keys to the mouse and command keys. So, the spoken sentence "Save Chapter 2 In Smith," means:

Save: Keyword command

- Chapter 2: File name
- In: Keyword for destination (often implicit in keyboard commands by command order)

Smith: Destination File Drawer name

In addition to the ergonomic efficiencies, voice commands could be given while other work is being done with the keyboard, mouse, or command keys providing time efficiency. The "Retrieve Mail" could be given while typing a memo, for instance.

Similarly, a 5090 operator could benefit from radio receiver-transmitter voice commands. 5090 users constantly move about their work space and often aren't watching the machine when its' paper supply runs out or stops for other reasons. With voice communications, each machine could status the user: "Machine 1 reporting; Tray 3 empty in two minutes." Or the user could initiate the conversation: "Machine 4: Status Tray 2" (how much time before Tray 2 runs out). Or: "Machine 4, Paper requirement, Tray 3" (how much of the paper stock type in Tray 3 does the machine need to complete the job).

Let's take this a little further. Adding full speech recognition to a word processor could eliminate the qwerty altogether. Just speak and the machine types the words.

This kind of system has a number of inherent problems like: when is the spoken sentence a command and when is it a string to-be-typed? Even a keyword precursor like "command," spoken before a command is a potential problem: when is command a word and when is it a prefix to a command. Some kind of knowledge of context is required. This is really a question of how to initiate a mode. A related question is how to switch modes. Having the machine end a command by implicitly recognizing its end is one possibility, but these and other related problems are not well understood today. Even so, the power and appeal of voice communication will no doubt foster its growth.

A dramatic example of how other components of human conversation might be incorporated into dialogs comes from the MIT Media Lab where the disembodied face of a real person can gesture and hold a conversation:

> "Negroponte: 'We came up with the idea of projecting onto video screens sculpted like people's faces and also having the screens swivel a bit -- so they could nod, shake their head, turn to each other. At each site the order of sitting of the five people would be the same. At my site I'm real and you're plastic and on my right, and at your site you're real and I'm plastic and on your left. If we're talking and looking at each other, and one of the faces across the table interrupts, we would stop and turn toward him.'
>
> The important issue in the exercise, naturally dubbed "Talking Heads," was how well you could transmit nuance. If a senior national official is still dubious about a proposal but eager to come to some agreement soon, that's more likely to be expressed in gesture and facial expression than in words. If someone is joking to relieve tension, you want that to come across clearly, and the amusement of the others has to be registered immediately, or deep misunderstanding could develop -- 'Who did he mean when he said that?'

> By use of head-tracking devices at each site, the motions of each person's head could be easily transmitted, and fixed video cameras could send the images. The television tubes, it turned out, could indeed be molded like life masks to the shape of anyone's face. Negroponte: 'It was uncanny. When we rear-projected talking faces, even though the face-shaped surfaces were solid, you swore you saw the physical lips moving. It was creepy' " [Brand, p. 92].

(Could this kind of conversation be incorporated into human-computer conversation by giving the computer a human face? Of course the machine would need to know *when* to gesture and nod.)

We can see how these kinds of dialogs could help. Both the word processor and 5090 examples could enable the kinds of physical efficiencies I found in restaurant work. Including gestures could disambiguate computer communications as they do in human conversation. Plus, they have the practical advantage of already being underway. For instance, researchers are working on disambiguating natural language communications [eg.: Moore; Roth, et al.]. But there are also other directions to investigate.

Intelligent Machines and Intelligent Dialogs

Before we look at other directions, I want to point out that, in human speech, it's the speech mechanism that enables speech, but it is cognition, emotion, reflex, and other factors, not the speech mechanism, that determine which words are said. Or, for example, the Talking Heads dialog produced gestures but the meaning of the gestures was provided by the observer, not by the gestures themselves. And, the meaning was not provided by the observer's sight mechanism. It was provided by the

thoughts that interpreted the viewed gestures. In these examples we can see that, in both directions of a conversation, it is not the dialog mechanism that provides meaning. It's the underlying intelligence.

This brings us to the notion that machine intelligence may be a key to enabling operability. One example is the 5090 paper stock selection issue. To do some of its complex work, the 5090 must know what copy paper is loaded for each job. Without going into the detail, I'll just say that, without the user knowing, it operates differently for 3-hole, transparent, 8.5 × 11 inch, 9 inch, and tab stocks. How does the 5090 know what stock is loaded? Users tell the machine by making a work area selection (Figure 7). Of course it would have been better for the machine to know the stock type by virtue of the user placing the stock into the paper tray, with no dialog selection at all. But that wasn't an option because the technology to sense the paper type did not exist. So, 5090 operators have an extra responsibility (selecting the stock type at the dialog *in addition* to putting the stock in the paper tray), and an extra memory load (we often find operators forgetting to select the stock type at the dialog). Also, it added a great deal of complexity to the dialog and the dialog design process. Notice, for instance, access to the Paper Supply work area is not on the scorecard; it's in another work area (Figure 8). All in all it was a headache for everyone that could have been completely eliminated if the machine was smarter.

In the remaining sections, the focus will move, for a while, from "What can we do to the dialog?" to What can we do to the machine?" For, it seems to me that, without intelligent machines, intelligent dialogs can only be useful up to a point, that intelligent dialogs alone cannot solve all the emerging issues in dialog design. But, dialogs will reassert their importance near the end of the chapter as I discuss the relationship between language and machine function.

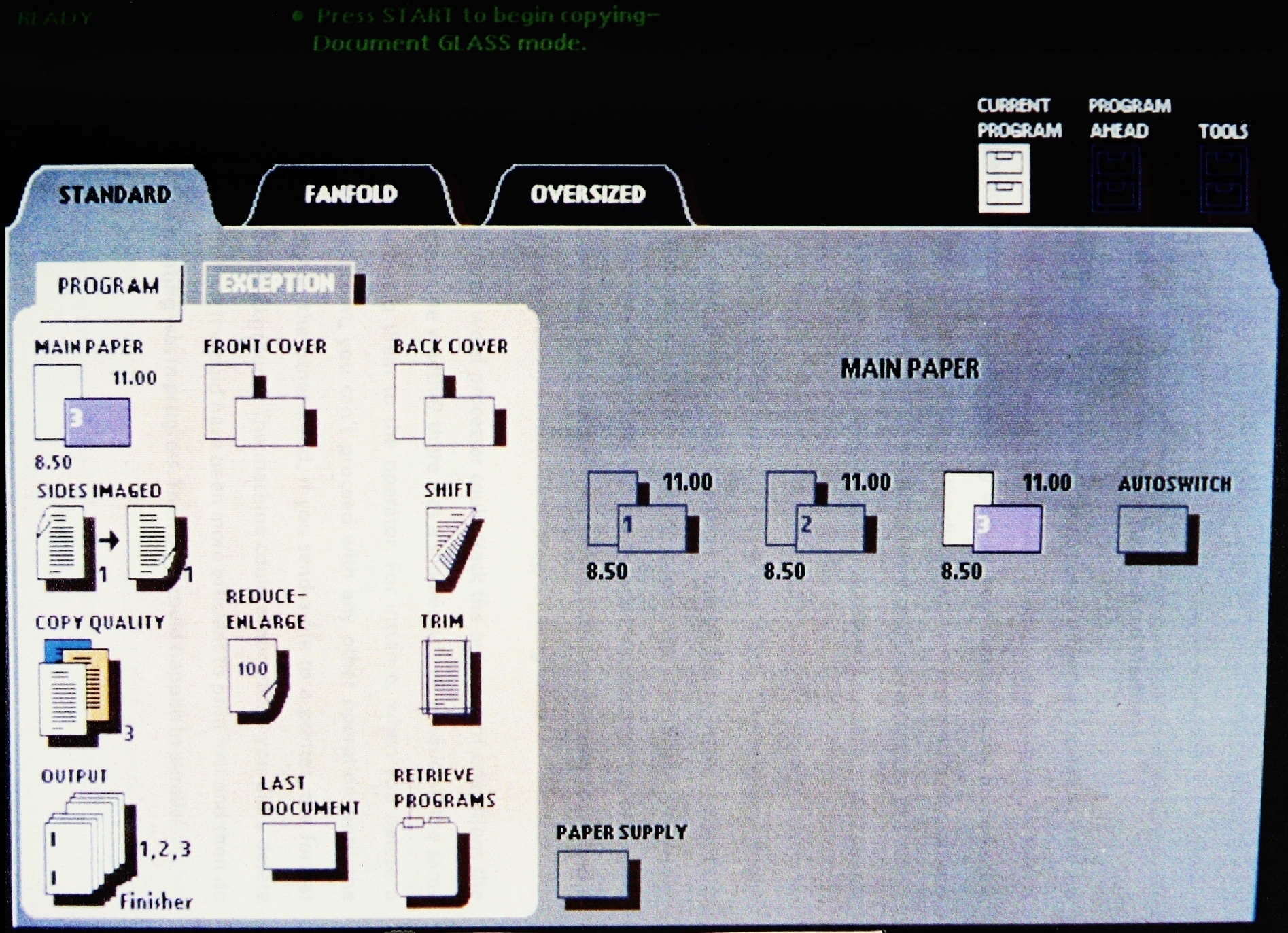


Figure 8: 5090 Screen, Main Paper work area

The Fifth Dialog Revolution: Tactic-Based Machines

Normally tactics follow from a particular strategy. However I think it may be easier for machines to begin to understand a user's tactics -- the features he selects to do a particular task -- than his strategy because user commands are immediately and directly available to the machine via the dialog without any special calculations or intelligence. A user's strategies, on the other hand, must be recognized by finding patterns of user behavior. So, a machine that understands a user's tactics seems closer to realization.

For example, let's say a 5090 operator programs a job. The machine might recognize that it could produce the identical output faster with a different combination of features. In this case it could inform the user about this other possibility and let the user decide if he wants to change his programming, or it could change the programming itself and tell the user why. Or, the machine could find a way of producing the same output with fewer programming steps.

Similarly, a Star-type word processor could track the outcome of operations the user performs, determine whether there was a simpler way of producing the same outcome, and reporting them to the operator. For instance, when you delete a remote file in Viewpoint, you can't proceed with any other operation until the delete is complete. On the other hand, if you send a file to a printer, the format operations go on in background. The machine could report, after you did a delete followed by a print, that it would have been more efficient to print first and then do the delete while printing was in progress. Then both could happen in parallel.

At the same time, such a machine might track the user's operations and report when he's come up with a programming combination better than any previously discovered for a given outcome.

Notice that, in these examples, the machine devises a more effective method of producing the same output. This kind of understanding the machine has here is very straightforward but it could pass for intelligence: "Why is the user making these selections? To produce *this* [the calculated] output." Since 5090's and word processors know what their output will be before it's produced, the data required to draw this conclusion exists today. A lot of engineering and a lot of code may be required to produce knowledge from the data for a tactic based design, but I don't think it's an intractable problem.

Tactic based dialogs are a good example of how machine intelligence can simplify a dialog designer's job while helping the user. The dialog designer doesn't need to worry that the user won't get good information about how best to program the job. Instead he needs to develop a simple and timely presentation of the data the machine provides.

If the machine reports what it's found and *why* it's better, a lot of learning can take place that the user would normally do on his own over a long period of time. Plus, it would let the user know that this kind of search for efficiency is desirable. It might even be fun. Tactic based machines could provide major productivity increases, especially for novice users. In combination with voice and other conversation components, it could be a major help for experts as well.

One interesting aspect of building dialogs for tactic-based machines is that they might provide novices with different tactical suggestions than experts. As we've seen, things that make a dialog easy to learn may make it hard to use. A machine

that could track a user's experience level might suggest a variety of ways to program the machine on that basis. This presents some interesting challenges for the dialog designer, like how enable consistent to keep a changing dialog. But it also would require more machine intelligence than is currently available. We'll face this same kind of problem in the rest of our investigation into future dialog types.

The Sixth Dialog Revolution: Strategy-Based Machines

Showing a user new ways to use a machine's features to do a *particular* job may in the long run enable him to develop a higher order concept of how to do his job. Some people have the ability to survey what they've done, recognize a pattern to it, and then break out of the pattern and form a new, more efficient one. For others, "a little prompting will often suffice" [Hofstadter, p. 37]. But, in my own work experience, we've seen that kind of synthesis take a long time to develop, even when one looks for it. Users of a tactic-based system would learn a better method of doing a task *after* they've completed the task. This leaves opportunities to forget, especially for infrequent job types or casual users, because there's no chance to practice. Furthermore, this pertains to a single job, not to a way of working that will foster improvement in tactics. As the proverb says, it's better to prevent the problem than to fix it. So, instead of having people search, by trial and error over long periods, for an effective strategy, why not *give* them the most effective known work strategies up front in the dialog.

Back to the 5090 for an example. The 5090 program had a test group consisting entirely of people with no experience in operating duplicators. This team members spent their days running jobs on the 5090 in order to identify problems. Each Monday, every person got a set of jobs to finish before the end of the week. A few

people *always* finished their jobs first regardless of the kind of jobs they are assigned. The individuals in this group of efficiency experts also wrote up more problem reports than anyone else. That means they spent more time doing things other than running jobs, yet they still *always* finished first. In discussions with members of the test group and their manager, it appeared that the efficiencies the faster members found related to "breaking out of patterns and forming new, more efficient ones." Aside from being motivated to find efficiencies, becoming more productive involved small, incremental improvements which added up to hours worth of savings over the course of a week. Most of the efficiencies came from saving steps: organizing and running similar jobs consecutively, methods of eliminating walking to get more copy paper, methods of loading the paper trays fewer times, figuring out what programming would get the machine to do a job more quickly, and so on. These may seem like they are all small strategies. On 10 different jobs there may be 10 different required tactics to reduce the time it takes to load new paper stock, but the single strategy was to "find out how to reduce the time it takes to load new paper stock."

That these strategies were discovered by some operators and not others deals in part with motivation which I'll return to below. The questions I want to address here are:

- can the best strategies discovered and employed by the expert users be recognized by a machine and presented to all the operators; and,
- will this enable more operators to create their own, new and better strategies by building on the best know ones?

A machine that could glean user strategies from their behavior, evaluate those strategies, and display the best of them via its dialog would require a different class of machine intelligence than simple tactic-based machine, because it would need to

know about things we don't yet understand, or don't understand how to code or display:

- what is a strategy;
- how does one extrapolate a strategy from a set of user behaviors and tactics;
- how does one represent strategies in the machine;
- how should different strategies be evaluated and compared;
- how do strategic categories not known to the machine get handled (how categories are created and defined)?

AI techniques might play a large role in answering these questions. Uncertainty handling could help with defining categories, which, by their nature have fuzzy or overlapping boundaries. Plan recognition could help define strategic categories. Common sense reasoning could provide data about which strategy is better. Machine learning could help build knowledge about how individual users and users in general interact with the system, and this knowledge could provide the basis for devising improved work strategies. There are stories in the media about computer programs that can, for instance, make conjectures, albeit in a limited domain of graph theory: "Suppose a computer program could give artists their ideas for what to paint. Or what if a computer program churned out possible story ideas for novelists. . . Mathematicians, many of whom consider themselves more artists than scientists, have had to consider exactly this problem" [Kolata]. We would continue, ". . . Or what if a computer turned out possible strategies for using your machine?" However, none of these techniques are viable or robust enough today to handle these missing links. And, in order to accomplish the strategy recognition and evaluation, the various techniques would all need to interact, making the endeavor even more complex. So, it appears that having machines identify and help improve

user strategies by identifying the best known strategies is a long way off. But, even with this technology, would users create new and better strategies?

There is evidence that strategy-based machines would help. Cormier [p. 162], describing transfer of training, says that increased practice almost always leads to increased performance in both quality and speed. Showing tactics and strategies to users, that is, training them in more effective methods of using the machine, would decrease the search time for these and, because they are immediately available, increase the amount of time one gets to practice them. With practice, higher order concepts or new relations may emerge [Cormier, p. 175].

Dialog Solutions as a Way to Increase a System's Value

One of the big problems with computers and computer based machines is that their CPU's are idle a lot of the time. For example, at night most word processors are doing no useful calculations at all. And between keystrokes, they do only management functions which are useful but which don't use up the entire CPU capacity. Tactic and strategy-based dialog solutions could be arranged to increase a machine's value by helping the user find the best procedures while the user isn't using it. The CPU could, during these off hours (or moments), work on calculating the best tactics or strategies for the user. Then it wouldn't interfere with the operators normal tasks but the end result would be useful, productive information.

A network of machines could even work to find the best solution discovered by *any* operator, machine, or operator / machine team on the network and make *all* the users of the system aware of it; and all the communication could be done unobtrusively. Having a system automatically spending computing time, say at night,

calculating which applications are the best and providing all users with that information could provide an increase in CPU utilization while making users more productive.[1]

The Seventh Dialog Revolution: The Science of Motivation

Intelligent machines and their dialog displays may provide users with sufficient knowledge to make the jump to expert. But unless the user is motivated to discover more, all the AI in the world won't help. Users have to *want* to build on the known strategies, they have to *want* to learn new tactics.

Challenges like those in video games or sports motivate some. Pleasant stories with attractive characters and happy endings motivate others. A number of users of Viewpoint and 5090 take pleasure in using and learning about these devices due to their amazement at the technology. I recall some subjects in early 5090 operability tests describing the dialog as "fun."

We have the technology to provide some of this motivation. Ease of learning can be a motivator. Perhaps adding playful elements can motivate. But, at this time, providing enablers for users to motivate themselves is an art and choosing devices to elicit motivating emotions is hit or miss. No one knows what makes people want to do some things and not others: even Michael Jackson doesn't produce a hit record or a dance-able tune every time. Where motivation resides in the brain and an understanding of how it works may, at some future time, provide us the tools to move beyond trial and error, and tendentiously produce dialogs that people really

1. "...as much as 50 percent of the added power of tomorrow's computers will go to make the machines easier to use" [Rogers, p. 53]. If this is the case, it will be in everyone's interest to have machine's do this kind of complex calculation when the users aren't using the machine.

want to use and learn about because they are excited and interested, because they are motivated from within.

I found a recent article describing the discovery, by Dr. Joseph Ledoux, of a physical center of emotions in the brain, the amygdala, that operate largely independent of thought.

> "The new evidence suggests that certain emotional reactions occur before the brain has even had time to fully register what it is that is causing the reaction; the emotion occurs before thought. That view is a direct challenge to the prevailing wisdom in psychology, that emotional reactions follow from thoughts about a situation. . . .
>
> 'Dr. LeDoux's research is the first to work out neural pathways for emotional response that don't go through the cortext,' said Dr. Michael Gazaniga, a psychiatry professor at Dartmouth Medical School. 'It may explain why so much of emotional life is hard to understand with the rational mind' " [Goleman].

This kind of work may provide our first preliminary clues to understanding what motivates people. When a person is chooses to engage in playing video games, watching Disney cartoons, reading, working at a particular job, or any other behavior, he is somehow motivated to do it. That motivation, it seems to me, derives from an emotional response: enjoyment of video games or work; or, working for fear of losing one's lifestyle; attending school for fear of parental sanctions, love of learning, boredom with other options, or fear of being left out. If they can be tapped, understood, and programmed into a machine's functional code, then maybe machines can provide users with reasons to learn and grow. It's all a ways off.

The Eighth Dialog Revolution: Distinctions Between Function and Language

"If we can someday match software encoding schemes to primitives used by the human visual system in its representation of size and shape information, then we should be able to make the entire system (the computer plus the user) more efficient."

-- Francine S. Frome [p. 19] --

"How will we directly connect our nervous system to the global computer?"

-- Roy Donaldson [Brand, p. 1] --

Discovering the "primitives used by the human visual system" and other systems could usher in a new era of machine intelligence and human-computer interaction (perhaps even the advent of cybernetics). The kind of machines to which Frome and Donaldson allude would, in some sense, *understand* the user and operate accordingly, in conversation and function.

One can see, in any AI textbook, that a major outcome of computer science research is the realization that human language is more ambiguous than anyone previously realized. But language is one of our most useful tools. How is it that we can understand despite the ambiguity? Apparently, a lot of ambiguity in conversation is resolved by understanding the context in which the ambiguities appear: first we hear a sentence. We know some of the information contained in the sentence (the grammatical structure, the context) before it was said. So, then we identify the statement's meaning, even if it's ambiguous, by relating it to the context. Then we apply that meaning in the conversation. All this happens instantly; how does it happen so fast? Human communication is based in part on common elements -- things that "everyone" knows (perhaps unconsciously) -- that don't require verbal communication. This "common knowledge" arises from our common experiences (upbringing, culture, education, etc.), biology, and the interactions

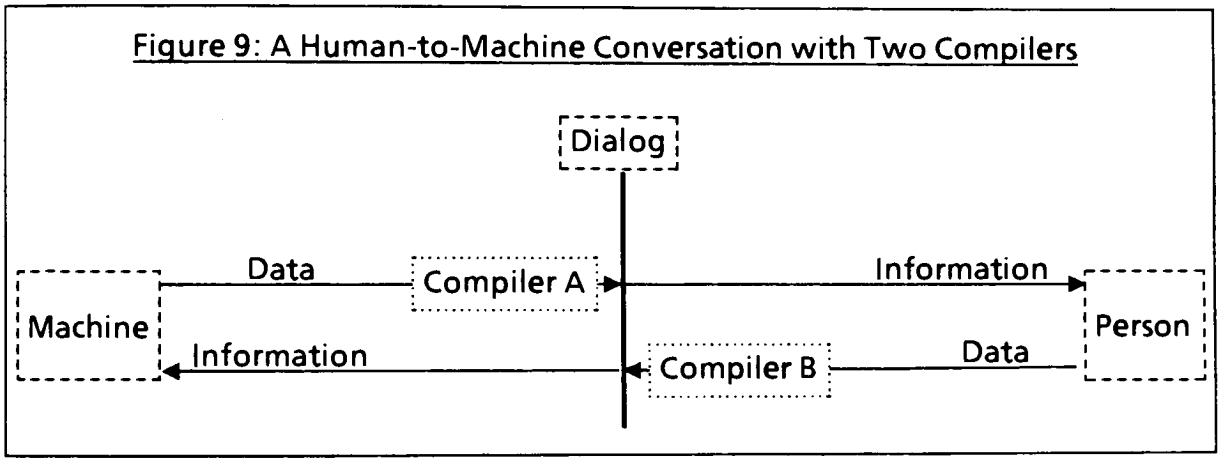
between them. These elements and relationships *are* our context. Since these are things that everyone already knows, we take advantage of them in conversation by reducing the number of words and explanations required to produce correct meanings [Lakoff and Johnson]. Perhaps these "things that everyone knows" are so deeply part of us that we can recognize them in real time.

The idea that there are things that everyone knows is one of the main themes of this investigation. For, if we can discover what it is that everyone knows, then maybe we can teach it to machines and display it in their dialogs, thereby making the machines available to the largest number of operators. Can a machine ever know these things that seem to depend on biology and learning over years of human experience? The debate continues.

But, even if machines can someday understand and make use the human of context, there is a further barrier to human-computer communication (and it is here that the importance of "dialog" reasserts itself). We've discussed how cognitive, behavioral, and metacognitive information about users should be known to the machine. What, then is the dialog's function? I'm thinking that machines understand elements of the operator's psychology *in their own language*, users understand things about the machine *in their own language*, and it is the dialog that will bridge that gap. This is what dialogs attempt today, and they've achieve some success. But today's dialogs are limited in that they are static and unintelligent. For people to communicate with intelligent machines, dialogs will need to *actively translate the machine's language into the user's language and vice versa*. The dialog takes on the role analogous to that of a simultaneous translator for people speaking different languages: both members of the conversation are intelligent and can speak for themselves; the translator facilitates the communication in real time.

Modern computer science employs a language translator called a compiler. A compiler's standard function is to translate a program written in a high level language the programmer understands into a program written in the low level language that the target machine can understand and run. Now, I'm thinking that with more knowledge about human psychology and behavior at tasks, compilers could be written to translate the machine's language into human language and vice versa, in real time. So, the underlying structure of a human-machine dialog might look like Figure 9.

Figure 9: A Human-to-Machine Conversation with Two Compilers



The machine will speak its own language, conveying its data. The compiler knows and *understands* the language and converts it into meaningful codes, displayed at the dialog, that the operator will find useful. In the other direction, the user will issue commands or queries and the second compiler will translate them into terms that the machine will find useful and meaningful. The front end of compiler A and the back end of compiler B will "understand" machine psychology. The front end of compiler B and the back end of compiler A will "understand" human psychology. In this way, the dialog and the machine will work together to help the user get his job done without spending a lot of time explaining and complaining: "what I really

meant was . . . ." But something's missing here and in all our discussion of future human-machine communication.

Machine Psychology: A Barrier to Machine Intelligence

How can "front end of compiler A and the back end of compiler B" understand machine psychology? What is machine psychology? To paraphrase the definition of psychology, it is (or would be if it existed) the study of how machines think and behave. But this seems absurd, or a science fiction at best. How could such a thing even exist? Why would I even address this question?

It may be that one of the great enablers of human intelligence is our ability to be introspective and reason about our own behaviors, thoughts, and feelings. Or it may be that introspection is a result of our great mental abilities. In either case, as I write about potential future directions, I can't help thinking that we'd be asking machines to reason without one of the major components of Reason. Will a machine be capable of intelligence without the ability to search for, clarify, and justify the methods by which it exists, survives, and thinks?

If such an eccentric thing is possible, if machines can explore their own internal processes, find consistent patterns, explain, think about, and communicate what they find, we may have an opportunity to take our communications with machines to where today only science fiction goes. On the other hand, perhaps I've taken this train of thought too far. For who can say what machines will enable next, and with what result?[2]

2. A paraphrase of the last line of Freud, 1961, p. 92.

CHAPTER 6

DIALOG THEORY: DESIGN METHODOLOGY

"Listen, listen, listen to the people who are doing the work."

-- (attributed to) H. Ross Perot --

" 'Know they user' must go beyond mere identification and stereotyping of the user population, especially when these data are obtained through indirect means or logical conjecture. Without direct contact between designer and the user groups, underestimation of the diversity and capabilities of the user groups can occur. Interaction between designers and the users can provide invaluable insights into the differences between designers of systems and users of systems."

-- Shaw and McCauley, p. 51 --

"We have learned from the Star the importance of formulating the fundamental concepts (the user's conceptual model) before software is written, rather than tacking on a user interface afterward. Xerox devoted about thirty work-years to the design of the Star user interface. It was designed before the functionality of the system was fully decided. It was even designed before the computer hardware was built. We worked for two years before we wrote a single line of actual product software. Jonathan Seybold put it this way: 'Most system design efforts start with hardware specifications, follow this with a set of functional specifications for the software, then try to figure out a logical user interface and command structure. The Star project started the other way around: the paramount concern was to define a conceptual model of how the user would relate to the system. Hardware and software followed from this.' "

-- Smith, et al, p. 246 --

"The user interface was once the last part of the system to be designed. Now it is the first. It is recognized as being primary because, to novices and professionals alike, what is presented to one's senses is one's computer."

-- Alan Kay, 1984, p. 54 --

Sorry, Alan

Sorry to disappoint you Alan, but the dialog is normally still the last part of the system to be designed. As late as July, 1988, a Xerox product planner told me he didn't want to develop a system around the user or the dialog. At Xerox, *no* product took the cue from Star's extraordinary success. The current application of Xerox's product delivery process insures that the hardware will always be developed before the dialog. At IBM, systems already in the field were so entrenched that they couldn't just produce a new machine. Office Vision software, designed to bridge various IBM operating systems with a Star-like dialog, was developed years after the products were in the field. Only OS/2 took it into account at all. Similarly, AT&T has a new direct select environment for Unix System V called Open Look, based on Viewpoint by agreement with Xerox. But, Unix has been around for many years and, once again, the dialog was plugged in late in the game.

User Oriented Systems and Conceptual Models

Members of the 5090 dialog design team often expressed the desire to work on a program where the dialog gets designed first, before the machine. But, Smith et al. [1982], Card, Moran, and Newell [1983], and Halasz and Moran [1983] prefer a modified approach: don't design the dialog to hide the system's complexity; instead, *design the system for the user, so that it's simple and allow the user to see the system as it is.* Halasz and Moran would display the system to the user via a dialog vehicle called a conceptual model. Conceptual models "present the underlying conceptual structures [of the system] directly to the user" [p. 384]. The user gets to see the actual workings of the system. With metaphors, to the user the dialog is the entire system.

With conceptual models, so far as the user is concerned, *the entire system is the dialog*.

A dialog using metaphors cannot hope to present an entire system accurately because they attempt to represent the system's structure "with familiar concepts that are fundamentally inappropriate for representing computer systems" [Halasz and Moran, p. 384]. And, since the system is hidden behind the metaphor, the user can only reason accurately in detail about the workings of the dialog, not the system. With conceptual models, on the other hand, the system is *exposed* instead of hidden, and the user is able to reason in detail about how to operate the system.

Conceptual models may be useful for efficiently using a system. But they lose the most important function of metaphors, incorporating a user's prior knowledge to facilitate learning about a system. Halasz and Moran include a role for metaphors in a conceptual model system to facilitate learning. They use the example of literary metaphors:

> "When we say that 'Turks fight like tigers,' we mean only to convey that they are fierce and cunning fighters, not that we should think about Turks in terms of tigers. We mean only to convey a point, *not a whole system of thought* -- the tiger is only a vehicle for expressing the concepts of ferociousness and cunningness. Literary metaphor is simply a communication device meant to make a point in passing. Once the point is made, the metaphor can be discarded" [Halasz and Moran, p. 385].

Though I won't pursue it, I should mention parenthetically that training is another method to promote learning about a system. Though often overlooked, training is a powerful tool that can pay off if it's done well.

It's one thing to say a user centered dialog or system should be designed first, before the machine is built. As a practical matter, as I've already noted, many companies base future products on past ones for important reasons of cost and reliability. The idea of designing a system's human interaction first may appear idealistic. But it is, in my opinion, a necessity in the long run. For, in the future, good dialogs will be the rule, not the exception. The winners will have them and the losers will not.

Cognition and its Place in Systems

Let's return to our operational definition of user interface: "A user interface is a medium that converts data into a useful form and displays it" (p. 1 - 1). In the definition of dialog, conceptual models fall into our "user interface to the system" category. The data's "useful form" is, in this case, the form of the system itself. This brings us back to the idea that cognitive and metacognitive user data should be incorporated into the system's functional code.

By incorporating knowledge of human psychology into the functional code, we'll accomplish three things. First, it's one of the main ways to design a system for the user. Without it, a system may meet the physical, ergonomic user requirements via good industrial design, without meeting the cognitive requirements. Second, by incorporating intelligence about the user into the system instead of just the dialog, the workings of the system itself will be more familiar to the user and so more easily comprehended when the system data is sent to the user interface medium. In this way, useful conceptual models are enabled because the data sent to the user interface medium takes on the structure of the system . Third, the system will be comprised of its features and its understanding of human psychology by design. This

may enable interactions between them *in the system*. These interactions may help the system itself come up with ways to eliminate abstractions from the applications (the connection between machine features and tasks) *before the data is sent to the user interface to the applications* and then display the ultimate application precisely; the user won't need to search for it.

Roles and responsibilities

The first task for a system designer is to decide what model is preferable for users of the system.

-- Smith et al. [p. 249] --

In artificial intelligence the knowledge engineer goes out and discovers what an expert knows about a particular subject, then brings that expertise back and codes it into an expert system. In dialog design, a similar role should exist: the "user engineer." In addition to expert knowledge, the user engineer gathers information on user behavior and cognition: what the users do and why they do it in the target work place. These data are brought back and incorporated into the system (recall that we are looking to include this information in the machine itself, as well as in the dialog). So, dialog design becomes a subset of the systems engineering function. But the systems engineering function has, itself, moved from one of understanding and insuring integrated system function, to one of understanding and insuring operability in light of the system's functions.

In order to do this, the machine goals, features, and possible feature extensions must be discovered first. Then the user's requirements must be identified and their behavior in this domain understood. Lakoff and Johnson, in their study of linguistic metaphors, demonstrate that there are quite a number of metaphors that *everyone*

is aware of -- often through cultural exposure -- and can hear without further explanation. The "things that everyone knows" is another example of the data that might be brought back to the system, if they could be discovered. The machine could then be designed from these criteria, the feature set and the user requirements, with the goal of providing the features in the simplest possible hardware and software configuration for that particular user population.

How the user requirements are manifested in the hardware and at the CRT is the responsibility of a combination of graphics designers, industrial designers, nomenclaturists (including for foreign language), systems engineers, and human factors (cognitive and ergonomic) professionals. These people work together *as a single, interdisciplinary team* to design the system.

In addition, the team also includes hardware and software engineers. During the design phase, this group should be constantly consulted for feasibility and performance issues, and to begin thinking about the directions their own designs will take and what kind of machine intelligence will be required.

It is important that these various skills be incorporated into a single team, for it provides a sense of single purpose via improved communications between a variety of normally distinct functions that sometimes seem to work at cross purposes.

Another, as-of-yet-less-tangible contribution could be made by motivational experts. Now, we don't know, for instance, why colors evokes particular emotions. We can't reliably duplicate motivational events because we don't understand the underlying psychological processes. Evoking motivation is still an art. So, apparently, motivational artists need to be involved. Alan Kay's done this in his most recent project at Apple. On his team are Jim Henson, creator of the Muppets, Frank Thomas, the Disney animator who created Bambi, Stewart Brand, author of the

Whole Earth Catalog, Marvin Minsky, one of the founders of artificial intelligence, and others [Rose, p. 140].

Tools

Before getting into this section, a warning is in order: to develop advanced and useful tools, the tool designer must know as much about the dialog designer and his work process as the dialog designer knows about the system end user because, *for tools to be useful, tools development must include its own dialog design effort.* On 5090, difficult-to-use tools cost us, all the way to the program's end. An early, concerted effort in tool operability would have save us large amounts of dollars and time. But it was not done.

A tool for developing CRT based, direct selection dialogs must incorporate a number of tools. These tools must provide:

- graphics,
- nomenclature,
- behavior prototyping,
- consistency,
- documentation, and
- requirements.

Graphics packages and discussion about the need for dialog prototyping and how to do it are common today. So, I won't pursue those very large topics. I also won't talk about nomenclature (except to mention that it too is remarkably

complicated, and such constraints as consistency and translation requirements must be taken into account). Instead I'll briefly examine a theoretical tool that might, in the future, help designers adhere consistently to user and feature requirements.

During product development, it's important that a user engineer continue to gather data on user requirements on an ongoing basis. This implies that, even during design, as new data arrives, user requirements may change or the designer's understanding of the requirements may change. Also, as marketing learns more about its customers, the purchasers of the machine (who may not be the users), the machine feature set itself may change. The changes may affect the adequacy of the dialog design and it's difficult to know exactly how the design is affected and where to change. But, even if the requirements don't change, designers often don't know whether their design meets user needs until after it's implemented and tested when it's too late to change.

If it's possible to describe user requirements in a structured way, say, via a grammar, then that information could be stored in a data base. The same data base could also contain all the information about the system's features and their interactions. When requirements or features change, those changes could also be stored in the data base. A design tool could be developed that could check the user requirements and the user-requirement interactions with the feature set against the dialog design being produced on the tool. For instance, let's say that 95% of the users of system XYZ are under six feet tall and that 80% of the time they are 15 feet from the CRT that houses the dialog. Also, the CRT is a touch screen that sits on the machine at a height of 50 inches. During the design, the designer chooses an 8 point font for a particular message and places it in the dialog. At this point the design tool would check the data base for violations of the known data interactions. The tool would inform the designer that 99.9% of the users cannot read 8 point font at 15

feet. However, with a screen height of 50 inches and a user population mainly under six feet tall, 8 point font would be readable by 95% of the users working at the machine. The designer could then intelligently decide whether or not to use 8 point font based on whether the users would need to see that particular message when they are away from the machine. There are thousands of decisions like this one that designers need to make.

Another important element of design that can be aided by good tools is documentation. I recall one designer who began documenting her designs and then stopped for one year. When we finally found out, the implementation and the documentation were completely different and no one knew which was correct because she'd left the company. One can envision a tool that produces documentation by the very fact of engaging in design; that is, designs are automatically documented as they are produced. Another, more sophisticated version might be a tool where there is no distinction between a design and its documentation, the design *is* the documentation. Such a tool would eliminate all disconnects between design and documentation. But exactly how it would do that is unknown.

Adaptive Dialog Design

A manual method that attempts to achieve the same goal of meeting user requirements is called "adaptive design" [Edmonds]. It was developed so that dialogs, while being built, can still be adapted to the needs of users. He describes a design method in which the

> "software component of an interface should be treated as a separate module within the computer system as a whole and not

> simply be embedded at a range of points throughout it. One of the reasons for this is that the design of an interface is difficult to complete without letting people experience it. It follows that it needs to be easy to change" [p. 389].

Edmonds' approach is, for now, more reasonable because grammars describing user behavior and requirements in detail are apparently a long way off. There are, however, grammars that *can* describe elements of user behavior and that could incorporate more detail over time as we learn more about user behavior and cognition [Payne, and Green].

Incorporating Machine Intelligence: Adaptive Dialogs

Another use for automatically referencing user grammars during design is to enable the development of automatically adaptive dialogs (tactic and strategy-based dialogs fall into this category). These dialogs change they way they look or behave based on the user's experience or expertise to accommodate each individual's work style and expertise. Some versions of this kind of dialog are manual. For instance, a library search program might ask the user: "Are you a beginner, intermediate, or expert user?" Beginners get instructions along the way. Experts get no instruction; they just enter the desired commands.

More advance versions could do the adaption automatically: the system might log each user's selections and change its display based on conclusions it draws from the input.

To accomplish the automatic adaptive dialog, the designers will need to include more than one style of dialog in his dialog. Each dialog would need to incorporate intelligent guesses about what category the user falls into, according to the

selections he makes. The user-grammar data base could help designers make intelligent decisions with less iteration, about what dialogs are appropriate for each user category.

Incorporating Machine Intelligence: Agents and their Agents

"No doubt, in such a symbiosis it would not be solely the human designer who would decide when the machine is relevant"

-- Negroponte [p. 13] --

As I've already noted, machine intelligence will play a large role in relieving the user of unpleasant responsibilities. Yet another application for the user grammar reference tool might be to help develop "agents," a further application of machine intelligence. An agent is a "helper," a piece of software that knows how to do a job; the user tell the agent what he wants done and the agent goes and does it because he *already knows* how. The concept of agents were developed in the 1950's at MIT [Rose p. 132]. Later Negroponte and Kay both incorporated the concept into their work.[1]

To design this kind of dialog, the designer needs to know how users actually do their jobs. But, it's one thing to know how users do their job. It's another to incorporate that knowledge into a design so that the design supports the user's work. A design tool referencing a user grammar could help enable this kind of dialog by giving the designer clues about how users do specific parts of their job. This data could then be accurately incorporated into each agent which, after all, will do a specific part of the user's job and therefore needs to know how to do it.

1. Finding out the best applications (connections between the task and the machine features) for a task might best be done by agents. Who'd know better the most effective work strategy than a dedicated agent?

Inconsistency 4: Variation in Designers

"Why do the numbers on touch-tone phones start at the top but the numbers on calculators start at the bottom?"

-- (attributed to) Jack Rosenthal --

The same user data often generates different designs and many variables come into play. For example, when designers adopt a "user conceptual model," the system's functionality changes [Smith et al, p. 249 - 250]. So, the resulting designs depend not only on incorporating user data, but on interpreting it, as well. Designers in different companies may be producing dialogs for similar products and for the same user base. Of course, they don't communicate and the resulting designs may be antithetical, even in conception. When designers do communicate, many different designs will arise from the same set of user data. Then, the designers will not always agree about which approach is best. When more than one successful precedent exists all subsequent design efforts must ask: "which should I be consistent with?" It's a toss up, and going either way -- or even choosing a new direction -- will call the designer's judgment into question!

Now, as we have seen above, and as Jonathan Grundin has recently written, inconsistency may sometimes be appropriate. Understanding the users need is the most important thing. But when the user requirements produce a variety of effective designs that each work differently, there may be a problem. For, users don't always want the "best" design: they often want designs consistent with what they're used to, even if those designs promote inefficiencies or are hard to use. This is another user requirement, but it's often overlooked. Should calculator keypads be reverse order from phone keypads? The answer isn't clear today. But what is clear is that the users of these two items are often the same people. And, in the near future, having

consistent calculator and phone keypads may be very important if their functions converge.

A tool that understands and responds to a grammar of user behavior and its interaction with machine features could help. It could provide designers, at least within a single company, with a consistent view and interpretation of the user requirements they design to. Such a tool could also incorporate future product trends and check the designs for them. It could have brought the various keypad designs together.

Artificial Intelligence and The Things Everyone Knows

If it is true that:

- machine intelligence is important for making systems simpler and simplifying their presentation to the user; and,
- systems, not just dialogs, should be designed for the user; and,
- there are things that almost everyone knows about in a given culture,

then, system designers must incorporate design methods that bring these elements into their systems in a coherent way. For, in the near future, someone will begin to do this effectively and consistently. Their systems will win.

Appendix A

PROGRAMMING LANGUAGE ISSUES

Graphical Programming

"... programming is more difficult than is commonly assumed."
-- Dijkstra, 1972 p. 10-11 --

"Far from finding a way to eradicate every single programming error, we seem to have arrived at the opinion that no such way is possible at all. . . .There is no escape from the need for good judgment, nor from the costs of misjudgment."
-- Green, 1980, p. 307 --

Green's attack on the early promise of software engineering focuses the need to give programmers the ability to observe and manage their judgments. Moving from user interface based (that is, essentially static) programming languages to dialog based languages (those housed in environments) can facilitate this goal.

In the late 1970's, some researchers began replacing "linear, symbolic computer languages" with "convenient and natural *visual* programming languages" [Jacob, p. 51] and programming environments. These environments appeared in large number [Brown, et al; Melamed and Morris; London and Duisberg; Moriconi and Hare; Shu; Sannella]. These were attempts to take the types of diagrams or sketches a programmer uses naturally to model general program behavior and data, and make them, rather than the standard languages, the program specification [London and Duisberg]. Some of these attempted to incorporate graphics into "all phases of the software life cycle:

- System requirements diagrams,
- Program function diagrams,
- Program structure diagrams,

- Communication protocol diagrams,
- Composed and typeset program text,
- Program comments and commentaries,
- Diagrams of flow of control,
- Diagrams of structured data,
- Diagrams of persistent data, and
- Diagrams of the program in the host environment" [Brown, et al., p. 27].

These even included visual representations of the running programs [Brown, et al; London and Duisberg]. In terms of popularity, these trends culminated in Apple's Hypercard and its clones in the late 1980's.

From an environment point of view, these were successful. For, example, moving from Interlisp-D (a graphical programming environment in which many graphical programming environments have been written) on Xerox machines to Common LISP on Suns, is a painful experience [Emanuel]. For the tools available in Interlisp-D are much richer and easier to use. But, in both cases one must still write LISP code as "composed and typeset program text." The state of graphical programming language solutions is well summarized by Fitter and Green: "There was no way . . . to lay down principles that would ensure a good fit between objectives, human abilities, and the performance of the system; all that could be done was to eliminate the misfits. While that may be a depressing view for anyone who hopes to discover design principles of the same stature as Newton's Laws, it is a very reasonable one in the present context" [p. 284 - 285]. Items:

- Trillium [Henderson], an attempt at a fully graphical, object-oriented programming language, was written in Interlisp-D to provide for non-programmers a tool to create working prototypes of interactive control panels and CRT based dialogs. Though used extensively at Xerox and, for a short time sold as a product, it too didn't live up to advanced billing.

Programming in Interlisp-D was required for the complexity of product based dialogs.

- The more recent Trillium-type product, BLOX™, developed by Template Graphics Inc., also has the same shortcoming. Once again, the interactions between the graphical objects must be specified with a standard-type language for any other than toy interfaces.

- For programs of any complexity, even Apple's highly touted language for non-programmers, Hypercard, provides the standard kind of linear programming language, in case the user wants to write anything beyond simple programs (there's an entire industry devoted to providing Hypercard programs; these programs are ubiquitous and easy to use, but this does not imply that they were easy to write).

- In the earlier systems, standard code
  - was placed next to its graphic representation so that it could be understood in terms of graphics [Moriconi and Hare],
  - was placed under graphic representation which were used to essentially name it [Moriconi and Hare; Jacob],
  - was required to define the meaning (semantics) of procedures [Jacob], and,
  - as in Hypercard, was required to program anything more complex than simple data processing [Shu].
  - Even animated, graphical representation of output needed to be programmed by standard means, by trial and error, until it looked good [London and Duisberg].

In a moment I'll discuss the problems associated with programming and programming languages, and some of their theoretical solutions. First, a brief overview of the programming task.

Programmers go through three main steps when writing a program: planning, coding, and debugging. Each requires a distinct set of skills and each has its own set

of associated problems. Within each step, a language should also take into account ease-of-use versus ease-of-learning for novices, experts, and those in between. And, the designer must decide if the language is for a specific or general set of tasks.

I've incorporated these considerations into the following discussion. But a completely coherent presentation of these ideas in a language is not going to wind up in any single programming language in the right combination or measure. Individual differences and the variety of applications will prevent that.

Step 1: Planning

Planning deals with developing an algorithm to solve the problem at hand. Generating an algorithm produces an abstraction from the problem: a description of the problem in terms of its solution.

For the novice especially there are a number of barriers to algorithm creation:

- semantics (language independent): novices often can't construct an algorithm to solve a problem because they don't understand the meaning of the problem or the meaning of the algorithm;
- the ways in which the programmer prefers to think; for example operation learners build rules (what a program does) by understanding low level, local information and learn programming more efficiently than comprehension learners who build descriptions (what a program is) by examining global features [Coombs, Gibson, and Alty].
- novices tend to over specify a set of conditions from the problem because they are unfamiliar with the level of generality expected in a computer program.

In addition, for both novices and experts, there are the problems of:

- impatience to move to the coding step without first planning; and

- a poor understanding of how the program will actually be used.

Step 2: Coding

Coding deals with translating the algorithm into a computer language. The code produced in this step produces the second abstraction from the problem at hand: a description of the problem's solution in another language.

For the novice, barriers to coding include:

- syntax (language dependent): a poor understanding of the language they'll be coding in;
- the application problem: correctly applying the structures of the language to the algorithm;
- the computation problem: the inability to express parts of the algorithm mathematically. This is due, perhaps due to poor mathematical background or a lack of understanding of the fact that mathematical expression describe reality.

We've already investigated the application problem. More on both the application problem and the computation problem later.

Step 3: Debugging

"Debugging" is jargon for correcting logical errors in a program -- either in the the algorithm or the code -- that produce incorrect results.[1]

For a novice, debugging becomes a problem when he:

- doesn't know that the results are incorrect; or
- can't decide whether the problem is the algorithm or the code (that is, he doesn't know where to look);

For experienced programmers, problems in debugging occur more as program complexity increases. Then, obscure, incorrect interactions manifest themselves only sporadically, often leaving no trace of how they were generated.

Theoretical Solutions

"This view is echoed by Thomas (1978) who argues that man-computer dialogues should be modeled on human dialogues, a point also argued by Kennedy (1974), though he does not suggest how the consequent programming problems would be solved."

-- du Boulay and O'Shea, p. 166 --

"The programmer can only use [a language] by virtue of . . . its properties; conversely a programmer must be able to state which properties he requires."

-- Dijkstra, 1972, p. 15.--

"Programming languages provide massive control structures with embedded data manipulations. However the natural human tendency seems to be to begin with data manipulations and add control structures as a qualification to the action."

-- Curtis, p. 215 --

"Gould et al. also showed that when subjects started off doing their descriptions in natural English, ambiguities and all, and then gave another description in a restricted syntax, they readily switched to using a more restricted and less

1. Grace Murray Hopper coined the term bug. "During the summer of 1947 the Mark II computer . . . was acting up, giving some erroneous information. The faulty relay was located, and there, inside it, was discovered the cause of the malfunction: a moth, beaten to death by the relay. The moth was taken out of the relay with a pair of tweezers and scotch-taped onto a page of the logbook. An operator wrote in the logbook, 'First actual bug found.' Aiken [the boss] liked to come into the laboratory and ask, 'Are you making any numbers?' When little work was being accomplished, Hopper and the others would say they were 'debugging' the machine -- a handy excuse" [Slater, p. 223].

ambiguous style thereafter, even though the instructions made no such suggestion."

-- Green, 1980, p. 301 --

"Ideally the grammar of a programming language should reflect its usage so that its application becomes transparent in the formation of the problem solution. This implies a grammar of problem solving. There should be a grammar which brings the 2 steps of construction of a solution and translation of the solution into a program together."

-- Boldyreff, p. 330 --

A programming language should address these and other problems involved in the three programming steps, the interactions between them, their interaction with programmer behavior, experience, and cognition, and the interaction with the problem set programmers will be addressing. To facilitate problem solving and understanding for ease of learning and ease of use, the language must enable a number of things for the programmer that leave him free to approach, understand, and attack the problem at hand.

As with dialog design in general, programming language design and the programming process has a mountain of research associated with it. In addition to the previous discussion in this investigation, a number of recommendations specific to programming language design emerge from the literature, as follows.[2] A language should:

Be Simple

2. Note that there is overlap among the categories: a number of items assigned to a particular category could also appropriately fall into one of the other categories.
Also note that there are many apparent contradictions in the literature. Often, but not always, seemingly contradictory conclusions refer to different contexts and aren't actually contradictions. For example, while one author says using English is appropriate for programming and another says it is harmful, the first refers to the planning phase while the second refers to the coding phase.

Employ a High Level Language

High level languages can bring a programmer closer to the problem as opposed to low level languages which bring him closer to the language [du Boulay and O'Shea, p. 184]. Then the programmer can spend more time getting the program right instead of figuring out how the language works.

Whether "high level" means a language more oriented toward specific tasks rather than general, depends on its use. If one will always code for the same specific task, a language designed to do that may be useful and simple. In this scenario, a language designer could meet Sebrecht's requirement that mapping of action to function should be simple for the user to understand [p. 643]. On the other hand, using the specific-purpose language for general programming problems might make the job more difficult than it would normally be using a general purpose language.

Error Avoidance

Typing in names (of, say, variables) incorrectly is an example of type-in error (type-in errors occur during coding). Syntax errors are another kind. Here, the programmer incorrectly types some piece, even a single character, of a predefined structure. In these cases there are a number of strategies for error avoidance:

- use graphics rather than typing for input [Bauer and Eddy, p. 9-10];
- prevent "significant omission" where a slight change in syntax creates a big change in meaning [Green, 1980, p. 279];
- provide no "default options" (i.e. option A is given by the computer if the user does not specify A or B) [Green, 1980, p. 279];

- impose an order on naming (say) variables; (for instance: the programmer must first define a name; then, when required, a menu of available variables appears, insuring no typo will occur for any name).

Preventing type-in errors may also help users focus on the task rather than the mechanics of the language. "Despite studies which show that, in general, syntactic errors are not the critical bottleneck in programming, detailed studies of novices' programs show errors in this class are frequently committed . . ." [du Boulay and O'Shea, p. 190]. So, these kinds of errors should be minimized [du Boulay and O'Shea, p. 169-170].

Another class of errors deals with the application problem: mapping real world tasks into systems tasks. Here again we're faced with question of whether there can be a user interface to applications. One feels that there might be in a language developed for specific tasks for then the applications are fewer. For general purpose languages, there may be an unlimited number of applications. Carroll and Kellogg's solution embodies the claim that "understanding real-world tasks in terms of system tasks is facilitated by filtering inappropriate goals" [p. 8]. For example, a language could encourage a programmer to state goals rather than algorithms, a more natural way of working [du Boulay, O'Shea, and Monk, p. 239].

Automatic Error Correction

The proverb says "an ounce of prevention is worth a pound of cure." But, people apparently prefer and use error correcting, rather than error preventing, strategies [Reed]. If this is the case, then automatic error correction routines to catch common programming errors should be included in the language [Black and Sebrechts, p. 174].

In addition to relieving the problems associated with ignoring the proverb, if an error correction strategy informs the programmer about errors, the problems also become less likely because the programmer is more likely to remember to not make the same mistake again. In this way, error correcting strategies can also be instructional.

Manual Error Correction (Debugging)

"The results reported by Sime et al. (1977) suggest that what does bring down error lifetimes [how long a bug remains in a program] is to improve certain aspects of the programming language."

-- Sime, Arblaster, and Green, p. 126 --

Programmers spend about three times as long eliminating errors (debugging) their programs as they do coding them [Kahn, p. 856; Gould, p. 151]. Unforeseen and unforeseeable errors creep into code from every direction. The causes of some bugs are so obscure that they're *never* discovered. Software engineering sought to eliminate bugs. But, no one has hit that homer yet (and, according to Dijkstra [1989], no one ever will). So any tool that helps debugging is a welcome addition. One of those tools may be the language itself.

Perhaps because it's so expensive, debugging has generated a vast literature. The result is that we know a lot about debugging. But we still don't understand it very well. In general, when debugging a program a person may perform four distinct actions:

- understanding the system or program being worked on;
- testing the system;

- locating the erroneous component of the system; and
- repairing the component [Katz and Anderson, p. 386].

There is evidence that programmers can and do make use of several sources of information in debugging [Gould, p. 168]. Depending upon what a person understands about a program, different debug strategies might be used [Katz and Anderson, p. 386]. Locating the bug is usually the most difficult part of debugging [Katz and Anderson, p. 388; Gould, p. 169: "Sometimes . . . even after they isolate the bug to a few lines, they cannot detect it"]. On the other hand, once an error is found, it's easily fixed [Katz and Anderson, p. 388; Gould, p. 168]. Also:

> ". . . [programmers] adapt their debugging tactics to their environment . . . . [They] seem to employ a general strategy of selecting a particular debugging tactic, finding a clue, and developing a hypothesis about the bug . . . . Subjects generally use an 'ease into it' strategy, initially avoiding relatively difficult sections of code . . . . Subjects focus their attention on a local region of code which is defined by both geographic factors (e.g. neighboring statements) and conceptual factors . . . . Bugs in assignment statements were the most difficult to debug . . ." [Gould, p. 168 - 169].

In this light, the debug tool should:

- run *any* section of code independently;
- shows arguments to any section of code, and their values [Lukey p. 209, 211];
- allow users to specify the argument values before running any section of code;
- include trace of the execution of each function including variables and temporary values; facilitates understanding as well as debugging [Plum, p. 219].

In addition to an explicit debug tool, if a language is to help with the debugging task, it should:

- focus the user's goals [Carroll and Kellogg, p. 8] to enable understanding;
- eliminate syntax errors; eliminate grammatical errors that compilers don't detect; "There tends to be an order in which subjects look for bugs (syntactic; grammatical errors that compilers do not detect; and, finally substantive)" [Gould, p. 168]. Eliminating the first two narrows the search, the most difficult part of debugging; and
- where possible, specific information should be derived as an instance of a more general principle; such an approach enables students to gain more systematic understanding of the programming language [Boldyreff, p. 319].

These points just scratch the surface of the issues associated with bugs.

Keep the Number of Different Concepts Small

One advantage here is there are fewer things to learn and remember. This means each concept can be learned more fully and quickly, enabling earlier usefulness.

Plum "argues for small systematic languages and against languages that give the superficial appearance of naturalness" that encourages the user to type in English [Plum; du Boulay and O'Shea, p. 166; also: Smith, et al., p. 274].[3] But, it's important that the language not be so small that it becomes overly complex or low level. For example, Green [1980, p. 285] cites the following example:

3. A number of other studies suggest that using natural language commands facilitates the use of computers (Black & Sebrechts, p. 153-154]. But these deal mainly with using programs, not writing them. For programming languages, it might be preferable if English was used to describe the purpose of a command rather than what it is.

NAND can replace AND, OR, and NOT, but then, for example:
(p OR q) AND r = ([(p NAND p) NAND (q NAND q)] NAND r) NAND ([(p NAND p) NAND (q NAND q)] NAND r).

The total number of statements in a language should follow Barrow's "Economy of Concept:" a programming language should embody a small number of concepts appropriate for a given class of tasks [du Boulay and O'Shea p. 153; Carroll and Kellog, p. 10].

Make the Command Language Easy to Follow and Understand

Miller and Thomas say of a simple command language: "no other feature is as important in determining an individual's effectiveness in using a computer system" [du Boulay and O'Shea p. 156]. To enable this, Bauer and Eddy suggest: "Graphic [vs. text] representation of command language is easier to learn, remember, and use as a reference. Rules are faster to learn; subjects remembered a rule better when it was presented in a graphic format and extracted needed information faster than with text rules" [p. 9 - 10].

Green notes three other issues in simplifying command languages: discriminability, learning, and understanding.

> "Discriminability is one key issue. Does this test here take part in a conditional or a loop? Easy to tell in a high-level language, hard in a flowchart or an assembly language. Is this loop event-driven or count driven? Easy to tell in modern languages, hard to tell in some older ones. Under what conditions can this action occur? Easy to tell in decision tables, also easy with Sime et al.'s if-P, not-P, end-P notation, hard with conventional languages . . . . The other key issue is learning . . . . Superordinate to both

> discrimination and learning is understanding. This, I maintain, is what it is all about . . . programs should be understandable . . ." [1980, p. 313-314].

Finally, as Green says, "full English is out" [p. 301].

Conditionals Should be the Easy Kinds

Conditionals are the statements that novices have the most trouble with [du Boulay and O'Shea, p. 152]. Items:

- If-then-else conditionals were originally devised in reaction to the unrestricted use of go-to statements. But "the medicine wasn't right for all cases, and sometimes the cure could be as bad as the disease" [Weinberg, Geller, and Plum, p. 34 - 35].

- du Boulay and O'Shea [p. 172] say that conditionals clash with natural language usage, due to internal inconsistencies in the programming language or arbitrary restrictions in the programming language;

- Friend found two predictors of programming problems for students are the number of conditionals and whether loops and subroutines were required [p. 146 - 147];

- Youngs found that novices were highly likely to make an error when coding conditionals even though they did not have many conditionals in their programs.

Some solutions are:

- if conditionals are used, conditionals should be in action-qualification form (principle actions followed by qualifying conditions) rather than the reverse [Black and Sebrechts, p. 159];

- affirmatives and conjunctions should be used rather than negatives and disjunctions [Green, 1980, p. 280; Black and Sebrechts, p. 159];

- Weinberg, Geller, and Plum present a number of "select" (case-style) statements they believe handle conditional events in more understandable ways than if-then-else's by reducing nesting;

- Sime, Arblaster, and Green found that nested conditionals in the form

  IF p: DO a;
  IF NOT p: DO b;
  END p

  allowed novices to correct programming mistakes 10 times faster than conditional constructions of the form IF p THEN DO a ELSE DO b;

- Green [1980, p. 280]: use case statements in which both alternatives are named explicitly, i.e "hot and cold" instead of "hot and not-hot."

- ". . . Miller (1975) found that novices were more at home with 'qualificational' rather than 'conditional' languages. They preferred instructions of the sort 'put all red things in box 1' to the conditional 'if thing is red then put it in box 1.' That is a statement of the goal rather than an algorithm for achieving it. This preference probably derives from the way instructions are usually given in English . . . and underlines the fact that instructing a computer is an unnatural activity and not at all like instructing a person. But caution must be exercised in attempts to make programming languages that look like English, lest the novice be fooled into believing that he is communicating with a machine with human capabilities and knowledge (Plum, 1977)" [du Boulay, O'Shea, and Monk, p. 239].

Nested Loops Should be Easier Than They Are

Double loops are difficult [Green, 1980, p. 291]. But these are useful constructs for programmers so eliminating them is not the answer. As Green points out: ". . . the grammar *in the head* is not likely to be the same as the structure on paper,

derived from the definition of the programming language" [p. 291]. One solution is to understand human psychology better and attack the difficult portions of computer programming via good language design. "Choose a good mental representation and the chances of getting a solution are at once much better" [p. 302].

In this context, one possible method of mitigating this problem is to categorize program constructs like double loops more appropriately. For example, perhaps the language contains only one kind of loop: the Loop. Loops could have a variable property, that is, the number of nested loops. Each nested loop could work exactly like the Loop and be run independently for debugging. Also, if the Loop was graphical, one could see its structure change as the variable property changed, and watch its operation as well. This would provide another cue about how nested loops work. In this way, multiple nested loops could all fall into the same conceptual category as single loops by language design. That way there'd be less to learn about how nested loops operate.

Simple Flow of Control

Although "specifying flow of control is central to programming in algorithmic languages, . . . many novices find it very hard" [du Boulay and O'Shea, p. 157]. (I'd say this is also true for non-algorithmic languages like Prolog.) The problem of specifying flow of control has led directly to important developments like structured programming, and, less decisively, to object oriented programming. In a programming language:

- there should exist discriminable blocks each with one way in and one way out [Dijkstra, 1972, p. 18];

- the user should be allowed to generate all or some of the logical relationships in a program [Mayer, p. 125, 133];

- the problem's solution algorithm should be expressed in the user's own words [Mayer p. 122, 134]; (but recall that using "one's own words" (eg. English) does not apply to the programming language itself [Green, 1980, p. 301])

- Green [1977, p. 105 - 108] reports that GoTo (jumping) is harder than nesting (if-then-if-then-else) for specifying flow of control in terms for coding, debugging, and understanding. This, Black and Sebrechts argue [p. 161], is an advantage in debugging and in understanding the flow of control. Wirth has argued (not decisively) for the complete elimination of GoTo's [Green, 1977, p. 107]. Knuth, on the other hand, argues convincingly (again, not decisively for it's opinion, not experimentation) for "the elimination of GoTo's in certain cases, and for their introduction in others" [Knuth, p. 262]. In the right situations, he feels, GoTo's can actually simplify flow of control. Though they are not often required [p. 294], in the right situations, he feels, GoTo's can be useful.

- there should exist no conditionals, which is one of two predictors of problem difficulty with flow of control [Friend, p. 146]

- there should exist no loops, which is the other predictor of problem difficulty with flow of control [Friend, p. 146 - 147]

No loops or conditionals? What's left except sequence? And without loops and conditionals, the sequence tends to become long and unwieldy. On the other hand a language could compromise between these extremes by presenting loops and conditionals in an obviously sequential form where each is a discrete, testable entity whose purpose is clearly defined by the programmer. This corresponds to the black box theory of structured programming. All the programming structures appear, at the top level, as discrete units appearing in sequence. Of course at the lower level, the programmer must implement the loop. But this kind of compromise could perhaps be a step in the right direction. This is what Green, Sime, and Fitter call

"imposing a *macrostructure*" on the program [p. 222 -223]. It helps by reducing the detail programmers have to concentrate on at any one time.

Keep the Syntax Simple

"The effect of syntactic complexity is to influence the learning of the task. However, once the computer task is learned, the execution of the task in time-sharing fashion can be achieved equally well for long and short syntactic strings" [Chechile, Fleischman, and Sadoski, p. 11-12].

Be Flexible

Modular

In modular languages, pieces of programs can be plugged in or out as required without disturbing the rest of the code. Ease of planning, coding, and debugging, as well as long term maintenance for programs all come into play with modularity.

In 1981, Lukey [p. 208] called for new types of program segmentation, some way to segment programs into "reasonable chunks." Long before then, the structured programming movement was calling for and implementing programming languages containing discriminable blocks each with one way in and one way out [Dijkstra, p. 18]. Since then, object-oriented programming languages have become increasingly common. The "objects" in these languages can be thought of as modules. Calls for modularity include:

- dividing the task into "mind size bites" (idea attributed to Papert);

- forcing the user to modularize the program so that he would only write code for arbitrarily small pieces of the problem. This would hopefully decrease the probability that any given part of the program will be wrong, thus increasing the chance that the entire program will be right [Dijkstra, 1972, p. 5 - 6]. Similarly, Sebrechts, Deck, and Black felt modularity was important because it would help break information into conceptual chunks [p. 203-204];

- aiming to "reduce the number of units of information which are necessary for understanding . . . The size and conceptual units that a programmer has available will be an important determinant of his programming behavior" [Brooks, p. 738];

- data abstraction techniques such as Ada's Packages (fully independent pieces of code used to do specific things) that deal only with interfaces to an object, not the implementation of the object;

- the object-oriented programming movement (for instance see Liskov).

Flexibility

Flexibility refers to the ability to load, save, and move pieces of code independently [Lukey, p. 208], and to easily altering code when required. Modular languages may help enable flexibility in some of these cases.

Reusability

Another element of flexibility is reusability [Lange and Mohan; Neal]. Reusable code, "software components that may be used in many different applications" [Meyer, p. 3], is a big theoretical component of object-oriented programming. Presumably, languages that feature reusable components ease the programmer's task by enabling them to write less software, and having the re-used functions be

correct more often. Meyer says that that reusability is under-used in practice [p. 3]. He's produced a language called Eiffel that encourages software reuse. But there are strident critics of the object oriented programming movement's methods for presenting the reusability function in languages [Guthery, p. 82].

Another example from Neal is a kind of a code reuse editor called Example-Based Programming:

> "Novices programmers often reuse code from books and manuals, as well as reusing their own, previously written, code. Experienced programmers match pieces of a problem with familiar solution segments, therefore reusing designs, and experienced programmers, like novice programmers, use code that others wrote . . . .
>
> We have replicated this natural process by making code accessible within an editing environment. We provide a mechanism that informs the user what is available and allows easy access to it. Code can be viewed, copied, or run" [p. 64].

Tractability

Tractability has two meanings associated with it. The first is close in meaning to modularity or flexibility: "When the sub-component can fairly readily be removed from the whole and a new one inserted, or when the order of executing sub-components can fairly readily be altered, we say the program is 'tractable'" [Green, Sime, and Fitter, p. 246]. This definition of tractability deals more with coding and planning.

In another sense, tractability deals with enabling to-be-performed operations and its relationship to programming notation. This version is easier to illustrate than

define: One notation for expressing values is Roman numerals. Another is Arabic numerals. Multiplication is much more "tractable" with Arabic numerals than with Roman (try it!). To paraphrase Green, programming languages are, at present, much more like Roman numerals than Arabic [1980, p. 274 - 275]. Programming language notation ought to, in its own domain, copy the kinds of virtues Arabic numerals provide in their's.

Note also that this relationship between to-be-performed operations and notation is similar to the application problem discussed previously: how does a user know when to use a particular machine feature to help do a given task? I'll return to the problem of application again later.

Be Structured

Structure

Programming should be structured in a way that will facilitate a match between the user's cognitive model of programming and the language [Sebrechts, p. 645]. "The logic of novices' programs often leaves much to be desired. This can be improved by giving them more experience of programming and by prescribing methods for program-building that encourage sound structure" [du Boulay and O'Shea, p. 191]. Keeping a strong structure also simplifies debugging and helps improve flow of control [Dijkstra, 1972]. In addition to the normal arguments of structured programming, languages could help provide structure in a number of ways:

- avoid using non-unique action-function mappings; the same object should not change functions in different contexts [Sebrechts, p. 643] (but see the discussion of modes, pages 4-8 to 4-11);

- visual displays should be organized to provide as little irrelevant information as possible. For example, screen presentation should be restricted to actual options [Sebrechts, p. 644];

- distinguish subprograms called by the main program from sub-programs called by a sub-program [Lukey, p.210].

Macrostructure (Skeleton)

> "To understand a text of any length, readers must be able to impose a *macrostructure* on it. There is too much detail to comprehend it all, so the reader will remember the gist of each chapter in a book, perhaps the gist of the sections within a chapter, occasionally the paragraphs within a section . . . . the macrostructure of a program (if we knew how to extract it!) would reveal not only the control flow and the data flow, but also the relationships between components. In contrast, there is also a *micro*structure, corresponding to the parsing of individual components" [Green, Sime, and Fitter, p. 222-223].

With a macrostructure, the overall structure of the system is preserved without providing a lot of irrelevant detail thus helping the user to focus attention on the procedural aspects of the task [Sebrechts, Deck, and Black, p. 204].

The macrostructures should be visible and tractable (in the modular sense). Structured programming makes a program more tractable [Green, 1980, p. 276; Green, Sime, and Fitter, p. 248-249]. But . . .

Tractability

Green [1980, p. 276] claims that structure improves tractability. And this may be true for the definition of tractability as it applies to the planning and coding phases. But, when it comes to the tractability of notation -- the things a language enables a user to do more simply -- as the amount of structure in a notation increases, the more difficult it is to modify it [Fitter and Green, p. 283]. So, structure apparently trades off against tractability in one context and improves tractability in another (for more on the importance of context, see the section "Be Understandable," below).

Microstructure

In addition to preserving for the user the system's macrostructure, at the same time there is a need to provide sufficient low level detail in the section of the task currently being worked on [Sebrechts, Deck, and Black p. 204]. With both available, the user should be able to work on details without losing the context in which the details occur. This kind of support helps provide meaning to the current detail work without increasing memory load too much.

Signal Syntax Constructions

"A good language signals its syntactic constructions, using devices that make it perceptually obvious what each construction is and where it stops and starts" [Green 1980, p. 282-284].

- Use markers to help: (eg. indenting) [Green, 1980, p. 296];

- use more informative begin-end structures (for instance:. begin x end x) [Green, 1980, p. 296];

- allow three different types of brackets to categorize syntactic constructs (parentheses, braces, and brackets) [Green, 1980, p. 297]. Or, similarly, graphical objects could take the place of various brackets.

Be Consistent

Consistent

Systems should be consistent in structure and design of commands to minimize memory problems in retrieving operations [Norman, 1982 p. 381]. In programming language design, this implies that each element in a language should behave in a manner consistent with the behavior of the other elements. There should be a universal set of commands (eg.: move, copy, delete, show, open, undo, etc) that apply in the same way to each aspect of the language [Smith, et al., p. 268]. Also, there should be only one way to do each thing [Smith, et al., p. 274].

Unity

By applying a successful way of working in one area to other areas, a system acquires a unity that is both apparent and real. Paradigms that Star used are:

- editing;

- information retrieval; and

- moving and copying [Smith, et al., p. 270].

For example, the move command applies to moving text, rearranging your desktop, filing, printing, mailing documents, and other system elements. *In each*

case, the command is applied in *exactly* the same way: select the object, issue the move command, select the destination. This makes it very easy to understand and remember what the move command does and how to use it.

Be Learnable and Instructional[4]

Some learning strategy should be incorporated into programming languages [Mayer, 1981, p. 133]. Below are some examples:

Advance Organizer

An "advanced organizer" is tool used to facilitate learning that normally takes the form of a short expository introduction prior to the thing to-be-learned and has the following characteristics [Mayer, 1979, p. 383]: It

- is a short set of verbal or visual information;
- is presented prior to learning a large body of to-be-learned information
- contains no specific content from the to-be-learned information
- provides a means of generating the logical relationships among the elements in the to-be-learned situations
- influences the learner's encoding process: "The manner in which organizers influence encoding may serve either of two functions:to provide a new general organization as an assimilative context *that would not have normally been present* or to activate a general organization from the learner's existing knowledge *that would not have normally been used* to assimilate the new material" [Mayer, 1979, p. 383; italics mine].

Advanced organizers have their strongest effect in situations where learners are unlikely to have useful prerequisite concepts [Mayer, 1979 p. 372]; and they tend to

4. Also, see the section below on problems faced specifically by novice programmers.

developing understanding rather than simple retention [Mayer, 1981 p. 124-125]. To accomplish the goal of enabling understanding, advance organizers should:

- allow the user to generate some or all of the logical relations in the to-be-learned information;
- provide means of relating information in the text to existing knowledge [also: Black and Sebrechts p. 150-151];
- be familiar to the user;
- encourage the user to use prerequisite knowledge;
- encourage the learner to actively integrate the new information [Mayer, 1979, p. 376];
- provide a model of the language first: "Model instruction provides the learner with a rich set of prior experiences which are familiar to the learner and by which new information may be understood and organized" [Mayer, 1975, p. 732]. This would also makes for faster comprehension of new information by providing suitable antecedents for the new information [Haviland and Clark, p. 514].

"If the to-be-learned material is a collection of isolated facts that lack any systematic structure then an effective advanced organizer could not be constructed" [Mayer, 1979, p. 375]. But, it's also interesting to note that flowcharts do not serve as useful advance organizers [Mayer, 1975, p. 732]. So the fact that something has "systematic structure" doesn't automatically qualify it as an advanced organizer. We might conclude that the description of a programming language, to be useful as an educational tool, *must be more than just a structured description of the language itself*. The description must also incorporate elements like user's previous knowledge and understanding of the task.

Instructional Components

Throughout this investigation I've alluded to items that help understanding and learning. I'm arguing less for a separate information system than for a language that has incorporated in its' design, elements that are instructional just by the fact of using them. The following should be clearly, simply, and always provided to the user:

- a meaningful representation of the system;
- the state of the system and the meaning of that state;
- the meaning of each command;
- the effect or result of each command and the meaning of the result;
- the context(s) of each of the four previous listed elements, and the meaning of the contexts.

Explicitness

To enable the instructional components, users need explicit, not implicit cues. Explicitness could mean an excess of text and graphics. But we want elegant, uncrowded ways of conveying the information. Methods of enabling elegant explicitness might be:

- *"Perceptual cues beat symbolic cues; symbolic cues beat no cues*

  If you want to convey information, think first about typographical or diagrammatic codes. Even when the same information is available symbolically it is often helpful to provide a 'redundant restatement' using a perceptual code. The standard example is indenting: it clearly displays the program structure, but the actual text contains the same information" [Green, Sime, and Fitter, p. 248].
- different classes of actions should have dissimilar command sequences;

- the system can ask clarifying questions;
- the system can answer clarifying questions;
- *"Short trails are better than long trails*

  When information is not immediately manifest but has instead to be deduced from the text, look at the various points in the text which the reader has to traverse. The fewer stepping stones the better. Not too many sub-routines -- not too many conditionals -- not too many layers of data structure" [Green, Sime, and Fitter, p. 248].
- following a trail should demand few mental operations [p. 248]
- only one mental finger should be necessary; that is, keep memory load low at any given time [p. 249]
- Another method is to make things visible:

> "A well designed system makes everything relevant to a task visible on the screen . . . a subtle thing happens when everything is visible: *the display becomes reality*. The user model becomes identical with what is on the screen. Objects can be understood purely in terms of their visible characteristics. Actions can be understood in terms of their effect on the screen" [Smith, et al., p. 259].

Of course the visible things must be meaningful. Graphic design is a critical skill for conveying meaning to the user via visible objects. But graphical objects are limited in the meaning they can convey. Users should be able to gather meaning from other sources too (one major source is context; but there are others). So, though the user generates his own meaning, the designer must provide enablers.

The Black Box Inside the Glass Box

A black box is an object whose internal workings don't need to be explained. It's sufficient to know what the black box does without worrying how it does it. This

hides complexity from the learner. But how does one find the black boxes and discover their functions? By looking through a glass box.

> "Now languages for novices can be implemented in such a way that some form of *commentary* is available. This commentary is the 'glass box' through which the novice can see the 'black boxes' working. It functions rather like the cut-away models of machines to be found in technical museums, and indicates the more important events going on inside . . . . There's no reason why a high level language such as PROLOG, with its pattern matching and back-tracking facilities, could not be implemented to present the novice with a simple and visible notational machine" [du Boulay, O'Shea, and Monk, p. 238 - 239].

Provide Users with Apprentices, Helpers, and Plan Recognition Data

Maulsey and Witten describe a system called Metamouse that attempts this very thing. And, the recently marketed Hewlet Packard New Wave Office environment provides "agents," helpers that know what to do because the user has "trained" them. Their true usefulness has yet to be tested in the real world.

Be Understandable

"Superordinate to both discrimination and learning is understanding. This, I maintain, is what it's all about."

-- Green, 1980, p. 314 --

Context Sensitivity, Semantic Cues, and Meaning

"If what people are bad at is handling complicated context free structures, what they are good at -- superb, even -- is using semantic cues to help."

-- Green, 1980, p. 297 --

And programming languages are exactly that: complicated context free structures. Semantic cues are tricks we use to glean meaning from ambiguity. For example, we use them to figure out meaning:

> "One of my favourites from his paper is 'Dogs must be carried,' an innocent-looking sign found by most escalators. Only the similar command 'Crash helmets must be worn' makes one wonder if doglessness is an offence" [Green, 1980, p. 300].

In this example, the correct meaning is -- apparently automatically -- recognized by the reader from the context in which it appears. First, from knowledge that the sign is next to an escalator. Second, from previous learning about how escalators function and the dangers associated with riding in them. Third, from the knowledge that, in our culture at least, it is unlikely that one will be prosecuted for not carrying a dog. These and other contextual elements interact as semantic cues to produce an appropriate and immediate meaning. So, the meaning of "Dogs must be carried" is *context sensitive*, it depends upon the context in which it is perceived by the reader. Note that this context is very broad incorporating everything from the context in which it seen, to the society the reader lives in, even to individual differences between readers.

Research into how to incorporate semantic cues into programming languages has been going on for a long time (relative to the history of computers). But none of these schemes has moved into the mainstream. The languages in use today, BASIC, C, FORTRAN, PROLOG, Ada, and PL/1 have no semantic cues or only very primitive ones.

BASIC, for instance, has context sensitive overloading of some symbols, that is, different symbols can mean different things in different contexts. For example, the equal sign (" = ") sometimes means "assign the value on the right of the symbol to the variable on the left." Other times it means "test to see whether the values on the

right and left are equal." Other languages like PL/1 and Ada have more complex versions of symbol overloading and it's sometimes useful. But people are not immediately familiar with the kind of semantic cue required to extract meaning from overloaded symbols because, unlike the overloaded 5 on an analog clock, we haven't spent years learning the context and applying it *rapidly and unconsciously* to discovering meaning in the symbols.

We need to incorporate into computer languages semantic cues derived from this sort of sociological osmosis (either that or change the things the very young see daily). For these are the things that everyone knows. Otherwise computer languages will continue to require learning not just the language (which seems to me to be the easiest part), and its application, but also the *meaning* of its symbols and structures in each context.

The following sections describe possible enablers -- excluding social change -- for incorporating more effective semantic cues.

Principles

People actively build mental models of their tasks [Sebrechts, Deck, and Black, p. 202]. To enable them to build e*ffective* models, the designer should (for example):

- attempt to remove barriers to understanding: eliminate the mundane so the user can concentrate on the important stuff. One method is to, provide and adhere to a set of principles. For Star, these were:
  - familiar user conceptual model;
  - seeing and pointing rather than remembering and typing;
  - what you see is what you get;

- universal commands;
- consistency;
- simplicity;
- modeless interaction;
- user tailorability [Smith, et al p. 248-280].

make things visible (see page A - 28, above):

- Never invoke a command or push a key and have nothing visible happen [Smith, et al., p. 262];
- employ graphics and graphic strategies;
- categorize dialog elements;
- the discriminability of programs will certainly make the macrostructures more visible [Green, Sime, and Fitter, p. 245]
- Employ Diagrammatic Principles
  - coded information be relevant;
  - notations should restrict the user to forms which are comprehensible;
  - notations should redundantly recode important parts of the information;
  - notations should reveal the underlying processes they represent preferably in a responsive interactive system which permits manipulation of the diagrams;
  - notations should be revisable [Fitter and Green, p. 259 - 283].

user goals should be established and distinguished from the goals of the programming language (experts are not generally good test cases here [Sebrechts, p. 645]). Then, "once user goals are established, the domain image presented within that context must be matched as closely as possible to the actual domain . . . . However, information about some of these parameters will in fact constrain other parameters . . . . We explicitly built those constraints into the simulation so that the user cannot develop a model that

violates real world constraints due to lack of constraints on the simulation" [p. 645].

- Training: Learners should be encouraged to "relate problem solving information to his general cognitive structures." This produces a restructuring and an integration of the new information [Mayer and Greeno, p. 362].

Conceptual Models

"Choose a good mental representation and the chances of getting a solution are at once much better."

-- Green, 1980, p. 302 --

From Lukey: "A key step in the understanding of a program is the identification of the inputs and outputs of the program segments. This principle is already well established for sub-programs and it can be usefully applied at other levels of segmentation" [p. 209, 211]. Here there are only three simple concepts: input, output, program segments (the rules that change inputs into outputs). Lukey's example could be used as a conceptual model for program segments smaller than sub-programs: the idea is for the language to show exactly these three elements and their relationship at all times because this is one of the things that really happens at the hardware level. And, it also happens to be a sufficiently simple concept to show to users.

Names and Categories

"In naming commands we want to maximize the ability to convey an implicit model (i.e. a set of relationships) of the system's actions by naming its commands to reflect that model."

-- Rosenberg, p. 12 --

"Whichever code is used, however, these 'structured diagrams' have grasped an important principle: by restricting the user to a few higher order blocks, instead of giving him very small blocks with which he can create whatever structures he likes, it becomes possible to use names. Instead of having to deduce the properties of a larger structure from its component pieces, one can recognize its name and recall its properties"

-- Fitter and Green, p.272 --

Programming languages categorize their structural elements by type while users categorize them, in context, by meaning. So, Green [1980 p. 297] says, categorize programming language objects by both their type and their meaning. That is, each structure's category should be specific to the nature of its operation [Barnard, et al., p. 6-7]. As with conceptual models, this already applies to sub-programs and could also be applied to other levels of program segmentation by using appropriate names.

This means that programming languages should enable the user to name programming structures according to its intended purpose. These names could make the structure easier to identify and more obvious when to use it.

Problems Faced Specifically by Novice Programmers: More on Being Instructional

Barriers

"Ideally the grammar of a programming language should reflect its usage, so that its application becomes transparent in the formation of problem solution. This implies a grammar of problem solving. In programming, analysis of the problem is often followed by two separate steps: construction of a solution and translation of the solution into a program. We should be thinking of grammars which will bring these two steps together"

-- Boldyreff, p. 330 --

Many researchers have devised explanations about why novices don't easily learn to program. For example:

- Novices need to develop problem solving skills. Good problem solving is based on domain specific knowledge [Mayer 1981 p. 138]. If the domain is the planning stage of program development, then we need to determine what specific problems exist and what problem solving skills are useful in planning. It is important to match the language representation to each relevant domain [Sebrechts, Deck, and Black p.205; Sebrechts p. 644-645].

- du Boulay and O'Shea identify two problems in the planning stage: "In general, during the planning stage novices have difficulty in formulating an algorithm for two reasons. First, they are unfamiliar with the level of generality expected in a computer program, and tend to deal with an over-specific set of conditions from the problem -- for example they forget to specify actions when these conditions do *not* hold. Second, they quite reasonably model their algorithms on more familiar sets of instructions such as those found in recipes. This leads to a qualificational rather than to a conditional formulation of the task that will be at odds with most current languages. There seem to be two possible solutions to these difficulties. One is to devote more teaching effort to qualificational/conditional distinction, the other is to employ languages where control mechanisms are closer to human (and natural language) usage" [p. 188-189].

- "Of those attending post-graduate courses, some individuals require little more than a manual and a user number [in order to write a program], while others require careful explanation before they are able to complete even the simplest task. This discrepancy is particularly interesting in a university environment because it suggests that some factors other than general intelligence (motivation, prejudice, etc.) are operating" [Coombs, Gibson, and Alty, p. 292].

  Coombs, Gibson, and Alty focus on "operation learners" vs. "comprehension learners" to explain how understanding comes about (or does not come about) in these cases. They find that operation learners -- those who build rules (what a program does) by understanding low level, local information -- learn programming more efficiently than comprehension learners -- those who build descriptions (what a program is) by examining global features.

- In addition to different methods of approaching a problem, there's also different ways to write programs. "Among programmers of very similar experience levels, differences of as much as 100 to 1 were found across programmers in the time taken to write a given program. Additionally, across problems constructed to be of similar difficulty, an individual programmer often displayed a six-fold difference in writing time" [Brooks, p. 738]. This implies that each person has a different natural style of programming and that some styles are more effective. But how can an instructor tell which style is best -- most natural and most effective -- for each learner? Learning a style of programming that does not fit one's mental model may inhibit learning.

- Shneiderman [1977] divides the problems novice programmers face into the broad categories of semantics and syntactics. Semantic knowledge is concerned with programming patterns and algorithms. It is language independent, is acquired by "meaningful learning," and is therefore resistant to forgetting. "The semantic knowledge is hierarchically organized with concepts ranging from low level details such as the comparison of two values or the assignment operation; to middle level issues such as the pattern for finding the largest element of an array or zeroing out an array; to higher level operations such as the quicksort algorithm or the conversion of infix notation to prefix notation . . ." [p. 194].

Syntactic knowledge deals only with a particular programming language and includes knowledge of syntax for an iteration statement or the arguments for built-in functions. Syntactic knowledge is acquired by rote learning and is subject to forgetting [Shneiderman, 1977, p. 194].

- "Programming languages provide massive control structures with embedded data manipulations. However the natural human tendency seems to be to begin with data manipulations and add control structures as a qualification to the action" [Curtis, p. 215]. So, the "system image" [Norman, 1982, p. 381] of a language should contain, at its top level, some method for abstracting information from a given problem and developing data structures for it. Later, it can be ordered according to the structures of the language.

- Based on my experiences teaching programming to high school students, I've found these specific examples of the problems that novice programmers face (all three are problems of abstracting a solution from a problem):

  - THE SOLUTION APPLICATION PROBLEM (planning phase): Normally, programmers attempt to understand the problem at hand, and then figure out an algorithm that solves it. The algorithm is an abstract description of the solution. I've observed that novice programmers have trouble creating algorithm. I believe (from observation only) that my students understood algorithms and how they worked when they saw them. What they didn't understand was how to think about a solution to the problem *in terms of an algorithm*, how to apply a solution that could be expressed as an algorithm.

  - THE FEATURE APPLICATION PROBLEM (coding phase): It is one thing for novice programmers to understand and explain what a discrete structure like a loop or a conditional is. It's another thing entirely to apply the appropriate structures of a particular language to algorithms and in the correct order so that the program will solve the problem. Experienced programmers can apply these structures efficiently because they have a large body of knowledge, have it efficiently organized [Mayer, 1981 p.138], and can use the knowledge to generalize about solutions and constraints [du Boulay and O'Shea, p. 150]. Students who did poorly in my

introduction to programming courses could explain what a loop or a conditional was. They had trouble applying those structures to create correct programs.

- THE COMPUTATION PROBLEM (coding phase): ". . . the purpose of programs is to evoke computation and the purpose of the computations is to establish the desired effect" [Dijkstra, 1972, p. 16]. It is my experience that very few people understand that a mathematical expression describes something or, even if they do understand that, they don't know how to write such an expression. But the programming task insists that programmers comprehend exactly that relationship between computation and real objects or concepts. For instance, Friend, in a study of how and why students write programs the way they do, found "a rather large proportion (over 20%) of algebraic errors" [p. 62, 64, 144]. Or to put it another way, "our concepts of causation . . . are not well adapted to reasoning about the sort of problem that is typical in human-computer interaction" [Lewis, Hair, Schoenberg, p. 1].

I present "application" and "computation" here as problems. Indeed they seem to be fundamental barriers for the novice programmer. But, when they interact, application and computation form the essence of algorithm implementation and programmers must deal with these complexities by the nature of the programming task itself [see: Dijkstra, 1989] rather than some failure of user interface design.

One might then suggest that novices who cannot learn to deal with these concepts and their interactions cannot be programmers. But I am not necessarily pessimistic.

The Solution Application Problem

One of the problems here is that, while the problem is obvious and visible, the solution must be figured out. And it can't be just any solution (if it could it might be

easier). It must be a solution that can be expressed, very exactly, as an algorithm. It's bad enough that problem solving is required. This very specific kind of problem solving required for programming is even more abstract.

One partial solution might be Bulman's method for making the abstract concrete. "Many software engineers use models of their solutions, but few explicitly model the problem" [Bulman, 1989a, p. 50]. His idea is to build a model of the problem and then to extract the objects in that model and code *them* (his example is a program that models a cruise control in an automobile).[5] One advantage of this approach is that the objects in the problem are concrete, not abstract. So they should (in theory) be easier to understand and code than a solution which is not a "natural" product of the problem but a devised abstraction. This means that the solution is harder to understand and generate than the problem itself. If this is true, then, given that both approaches require a correct understanding of the problem, there seems to me a chance of reducing the abstraction from problem to solution by changing the method or definition of solving problems.

The Feature Application Problem

I think that two things -- examples and instruction -- could help with this particular problem. For instance, the Eiffel programming language provides a library of commonly used data structures like linked lists and trees. Each structure comes packaged with operations normally associated with it like traversal and insertion. Perhaps the language could provide many clear and specific examples of how and when to use each one. In more general terms, a language could provide this kind of

5. Bulman's article is directed towards designing for object-oriented programming languages. But his main point is independent of object-oriented languages. That is, "objects don't replace [good] design" [Bulman, 1989b, p. 151].

example for each language element -- like a loop -- that always remains accessible and somehow comparable to the current portion of the problem being worked on.

Instruction via an intelligent information system might also help. Intelligent help in this context implies a knowledge of the problem in order to to know what language elements to use at any given time. This, in turn, implies that one must tell the programming environment what the problem is. And this brings us again to Bulman's concept of modeling and coding the problem. An information system that understood the particular problem would have a great advantage over any of today's information systems. The question I have is, once you model the problem for the machine, haven't you already finished the task anyway and so no longer require an information system?

The Computation Problem

This particular problem derives from a social, educational, political problem: math illiteracy. Advanced languages like Prolog attempt to get away from using math expressions for everything. But it's often still required to get a program written. The answer may lie in some future high level programming language that altogether eliminates mathematical expressions.

There are ways to attack this problem outside of the realm of programming languages. But that's another story.

BIBLIOGRAPHY

Barnard, P.
Hammond, N.
MacLean, A.
Morton, J. — Learning and Remembering Interactive Commands" *Conference on Human Factors in Computer Systems.* Gaithersburg, MD., 1982.

Bauer, David W.
Eddy, John K. — "The Representation of Command Language Syntax" *Human Factors.* 1986, 28(1), 1-10.

Black, John
Sebrechts, Marc — "An Invited Article - Facilitating human - computer communication" *Applied Psycholinguistics.* 2, 1981, 149-177.

Boldyreff, C. — "Generating a Programming Environment for Learners" in *Computer Skills and the User Interface.* M.J. Coombs & J.L. Alty (eds), Academic Press, New York, 1981.

Brand, Stewart — *The Media Lab - Inventing the Future at MIT.* New York: Viking, 1987.

Broadbent, Donald E. — "The Magic Number Seven After Fifteen Years" in Kennedy and Wilkes (eds), *Studies in Long Term Memory.* New York: Wiley, 1975.

Brooks, Ruven — "Towards a Theory of the Cognitive Processes in Computer Programming" *Int. Journal of Man-Machine Studies.* (1977) 9, 737-751.

Brown, C. Marlin "Lin" — *Human-Computer Interface Design Guidelines.* Norwood, New Jersey: Ablex Publishing Corporation, 1988.

Brown, Gretchen P.
Carling, Richard T.
Herot, Christopher F.
Kramlich, David A.
Souza, Paul — "Program Visualization: Graphical Support for Software Development" *Computer.* August, 1985, pp. 27 - 35.

Bulman, David M. — ""An Object-Based Development Model" *Computer Language.* August 1989a, pp. 49 - 59.

"Objects Don't Replace Design" *Computer Language.* August 1989b, pp. 151 - 152.

Campbell, Jeremy — *Grammatical Man: Information, Entropy, Language, and Life.* New York: Simon and Schuster, 1982.

Cantor, Georg — *Contributions to the Founding of the Theory of Transfinite Numbers.* New York: Dover Publications Inc., 1915.

Card, Stuart K.
Henderson, D. Austin — "A Multiple Virtual-Workspace Interface to Support User

Task Switching" *CHI '87 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1987.

Card, Stuart K.
Moran, Thomas P.
Newell, Allen — *The Psychology of Human - Computer Interaction*. Hillsdale, New Jersey: Lawrence Erlbaum Associates, 1983.

Carroll, John M.
Kellogg, Wendy A. — "Artifact as Theory-Nexus: Hermeneutics Meets Theory-Based Design" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1989.

Chechile, Richard A.
Fleischman, Rebecca N.
Sadowski, Darleen M. — "The Effects of Syntactic Complexity on the Human-Computer Interaction" *Human Factors*. 1986, 28(1), 11-22.

Coombs, M. J.
Gibson, R.
Alty, J. L. — "Acquiring a First Computer Language: A Study of Individual Differences" in *Computer Skills and the User Interface*. M.J. Coombs & J.L. Alty (eds), Academic Press, New York, 1981.

Cormier, Stephen M. — "The Structural Processes Underlying Transfer of Traiing" in Stephen M. Cormier and Joseph D. Hagman (eds), *Transfer of Learning: Contemporary Research and Applications*. New York:Academic Press, 1987.

Cormier, Stephen M.
Hagman, Joseph D. — "Introduction" in Stephen M. Cormier and Joseph D. Hagman (eds), *Transfer of Learning: Contemporary Research and Applications*. New York:Academic Press, 1987.

Curtis, Bill — "A Review of Human Factors Research on Programming Languages and Specification" *CHI '81 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1981, pp. 212 - 218.

Democrat and Chronicle — "A Trojan horse is eating our lunch" *Democrat and Chronicle* (editorial). Rochester, New York: June 12, 1989, p. 6A.

Dijkstra, Edsger. W. — "Notes on Structured Programming" in O. -J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Strcutured Programming*. NewYork: Academic Press, 1972.

"On the Cruelty of Really Teaching Computing Science" *Communications of the ACM*. Vol. 32, No. 12, December, 1989.

du Bouley, Benedict.
O'Shea, Tim. — "Teaching Novices Programming" in *Computer Skills and the User Interface*. M.J. Coombs & J.L. Alty (eds), Academic Press, New York, 1981.

du Bouley, Benedict.
O'Shea, Tim
Monk, John

"The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices" *International Journal of Man-Machine Studies*. 14, 1981, pp. 237 - 249.

Edmonds, E. A.

"Adaptive Man-Computer Interfaces" in M. J. Coombs and J. L. Alty (eds), *Computing Skills and the User Interface*. New York: Academic Press, 1981.

Emanuel, Keith

Personnal Communication (Conversation), March 7, 1989.

English, William
Engelhart, Douglas
Berman, M. L.

"Display Selection Techniques for Text Manipulation" *IEEE Transactions on Human Factors in Electronics*. HFE-8, no. 1, 1967, pp. 21 - 31.

Feurzeig, W., et al.

Programming-Languages as a Conceptual Framework for Teaching Mathematics. Final Report on BBN Logo Project, June 30, 1971.

Fisher, Lawrence M.

"Prospects: High Tech's New Twist" *The New York Times*. Sunday, February 21, 1988, sec 3, p. 1, col 1.

Fitter, M. J.
Green, T. R. G.

"When do Diagrams Make Good Computer Languages?" in M. J. Coombs and J. L. Alty (eds), *Computing Skills and the User Interface*. New York: Academic Press, 1981.

Freud, Sigmund

Jokes and their Relation to the Unconscious. New York: W. W. Norton and Company, 1960.
Originally published 1898.

Civilization and its Discontents. New York: W. W. Norton and Company, 1961.
Originally published with the cited line, 1931.

Friend, Jamesine

"Programs Students Write" *Institute for Mathematical Studies in the Social Sciences, Technical Report Number 257*. July 25, 1975.

Frome, Francine S.

"Improved Color CAD Systems for Users: Some Suggestions from Human Factors Studies" *IEEE Design & Test*. February 1984, pp. 18-27.

Gick, Mary L.
Holyoak, Keith J.

"The Cognitive Basis of Knowledge Transfer" in Stephen M. Cormier and Joseph D. Hagman (eds), *Transfer of Learning: Contemporary Research and Applications*. New York:Academic Press, 1987.

Green, Thomas

"Conditional Program Statements and their Comprehesibility to Professional Programmers" *Journal of Occupational Psychology*. 50, 1977, pp. 93 - 109.

"Programming as a Cognitive Activity" in *Human Interaction with Computers*. H.T. Smith & T.R.G. Green (eds), London:Academic Press, 1980.

Green, T. R. G.
Sime, M. E.
Fitter, M. J.
"The Art of Notation" in M. J. Coombs and J. L. Alty (eds), *Computing Skills and the User Interface*. New York: Academic Press, 1981.

Grundin, Jonathan
"The Case Against User Interface Consistency" *Communications of the ACM*. October, 1989, Volume 32, Number 10, pp. 1164-1173.

Goleman, Daniel
"Brain's Design Emerges As a Key to Emotions" *The New York Times*. Tuesday, August 15, 1989, p. C1.

Gould, John D.
"Some Psychological Evidence on How People Debug Computer Programs" *International Journal of Man-Machine Studies*. 7, 1975, pp. 151 - 182.

Guthery, Scott
"Are the Emperor's New Clothes Object Oriented?" *Dr. Dobb's Journal*. December 1989, pp. 80 - 86.

Halasz, Frank
Moran, Thomas P.
"Analogy Considered Harmful" *Proceedings of the Conference on Human Factors in Computer Systems (ACM)*. Gaithersburg, Maryland, March, 1982.

Hartson, H. Rex
Hix, Deborah
"Human-Computer Interface Development: Concepts and Systems for Its Management" *ACM Computing Surveys*. Vol. 21, No. 1, March 1989.

Haviland, Susan E.
Clark, Herbert H.
"What's New? Acquiring New Information as a Process in Comprehension" *Journal of Verbal Learning and Verbal Behavior*. 13, 1974, pp. 512 - 521.

Heckel, Paul.
The Elements of Friendly Software Design. New York: Warner Books, 1984.

Henderson, D. Austin
"The Trillium User Interface Design Environment" *CHI '86 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1986.

Henderson, D. Austin
Card, Stuart K.
"Rooms: The Use of Multiple Virtual Workspaces to Reduce Space Contention in a Window-Based Graphical User Interface" *ACM Transactions on Graphics*. Vol. 5, No. 3, July, 1986, pp. 211-243.

Hofstadter, Douglas R.
Gödel, Escher, Bach: an Eternal Golden Braid. New York: Vintage Books, 1980.

Hopper, Grace Murray
"Keynote Address to the ACM Conference on the History of Programming Languages" in Richard L. Wexelblat (ed) *History of Programming Languages*. New York: Academic Press, 1981, pp. 7 - 20.

Jacob, Robert J. K.
"A State Transition Diagram Language for Visual Programming" *Computer*. August, 1985, pp. 51 - 59.

Kahn, Arthur — "Problem Solving Training - A Factor in Debugging Computer Programs" *Proceedings of the Human Factors Society -- 29th Annual Meeting*. 1985, pp. 856 - 860.

Katz, Irvin R.
Anderson, John R. — "Debugging: An Analysis of Bug-Location Strategies" *Human-Computer Interaction*. Vol. 3, 1988, pp. 351 - 399.

Kay, Alan C. — *The Reactive Engine*. Ph. D. Dissertation, University of Utah, September 1969 (University Microfilms).

"Microelectronics and the Personal Computer" in *Microelectronics*. New York: Scientific American, 1977.

"Computer Software" *Scientific American*. Volume 251, Number 3, September 1984, pp. 53 - 59.

Kay, Alan C. and the Learning Research Group — *Personal Dynamic Media*. Xerox Palo Alto Research Center Technical Report SSL-76-1, 1976 (a condensed version is in *IEEE Computer*, March 1977, pp. 31-41).

Knuth, Donald E. — "Structured Programming with *go to* Statements" *Computing Surveys*. Vol. 6, No. 4, December, 1974, pp. 261 - 301.

Kolata, Gina — "Mathematicians Meet Computerized Ideas" *The New York Times*. Sunday, June 18, 1989, sec 4, p. 7.

Kozol, Jonathan — *Illiterate America*. Garden City, NY: Anchor Press, 1985.

Lakoff, George — *Women, Fire, and Dangerous Things: What Categories Reveal About the Human Mind*. Chicago: University of Chicago Press, 1987.

Lakoff, George
Johnson, Mark — *Metaphors We Live By*. Chicago: University of Chicago Press, 1980.

Lange, Beth M.
Moher, Thomas G. — "Some strategies of Reuse in an Object-Oriented Programming Environment" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1989, pp. 69 - 73.

Lewis, Clayton
Hair, D. Charles
Schoenberg, Victor — "Generalization, Consistency, and Control" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1989, pp. 1-5.

Liskov, Barbara — "Data Abstraction and Hierarchy" *OOPSLA '87 Addendum to the Proceedings*. October, 1987.

London, Ralph L.
Duisberg, Robert A. — "Animating Programs Using Smalltalk" *Computer*. August, 1985, pp. 61 - 71.

Lukey, F. J. "Comprehending and Debugging Computer Programs" in *Computer Skills and the User Interface*. M.J. Coombs & J.L. Alty (eds), Academic Press, New York, 1981.

Markoff, John "In Computer Behavior, Elements of Chaos" *The New York Times*. Sunday, September 11, 1988, sec 4, p. 6, col 4.

Maulsby, David L.
Witten, Ian H. "Inducing Programs in a Direct-Manipulation Environment" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1989, pp. 57-62.

Mayer, Richard E. "Different Problem Solving Compentancies Established in Learning Computer Programming With and Without Meaningful Models" *Journal of Educational Psychology*. Vol. 67, No. 6, 1975, pp. 725 - 734.

"Can Advanced Organizers Influence Meaningful Learning?" *Review of Educational Research*. Vol. 49, No. 2, Summer, 1979, pp. 371 - 383.

"The Psychology of How Novices Learn Computer Programming" *Computing Surveys*. Vol. 13, No. 1, March, 1981, pp. 121 - 141.

Mayer, Richard E.
Greeno, James G. "Effects of Meaningfulness and Organization on Problem Solving and Computability Judgements" *Memory and Cognition*. 1975, Vol. 3(4), 356-362.

Melamed, B.
Morris, R. J. T. "Visual Simulation: The Performance Analysis Workstation" *Computer*. August, 1985, pp. 87 - 94.

Meyer, Bertrand "Eiffel: An Introduction" *The Eiffel Language Manual*. Interactive Software Engineering, Inc, TR-EI-3/GI, Version 2.1, 1989.

Miller, George A. "The Magical Number Seven, Plus or Minus Two: Some Limits on our capacity for Processing Information" *The Psychologcial Review*. Vol. 63, No. 2, March 1956, pp. 81 - 97.
Miller's famous paper had a strong influence on the entire 5090 dialog design.

Moore, Johanna D. "Responding to 'Huh?': Answering Vaguely Articulated Follow-up Questions" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, 1989, pp. 91 - 96.

Moriconi, Mark
Hare, Dwight F. "Visualizing Program Designs Through PegaSys" *Computer*. August, 1985, pp. 72 - 85.

Neal, Lisa Rubin "A System for Example-Based Programming" *CHI '89 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1989, pp. 63 - 68.

Negroponte, Nicholas — *The Architecture Machine*. Cambridge: MIT, 1970.

Norman, Donald A. — "Steps Toward a Cognitive Engineering: Design Rules Based on Analyses of Human Error" *Conference on Human Factors in Computer Systems*. Gaithersburg, MD., March, 1982.

"Design Principles for Human-Computer Interfaces" *CHI '83 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, 1983.

The Psychology of Everyday Things. New York: Basic Books, 1988.

Papert, Seymour — "Teaching Children Thinking" *IFIP Conference on Computer Education*. 1970. Amsterdam: North-Holland.

"Teaching Children to be Mathematicians Versus Teaching About Mathematics" *International Journal of Math. Educ. Sci. Tech*. Vol. 3, 249-262, 1972.

Payne, S. J.
Green, T. R. G. — "Task Action Grammars: A Model of Mental Representation of Task Languages" *Human-Computer Interaction*. Volume 2, 1986, pp. 93-133.

Plum, Thomas — "Fooling the User of a Programming Language" *Software - Practice and Experience*. Vol. 7, 1977, pp. 215 - 221.

Reed, Adam V. — "Error-Correcting Strategies and Human Interaction with Computer Systems" *Conference on Human Factors in Computer Systems*. Gaithersburg, MD., March, 1982.

Rogers, Michael
Sandza, Richard — "Computers of the '90s: A Brave New World" *Newsweek*. October 24, 1988, pp. 52 - 57.

Rose, Frank — "Pied Piper of the Computer" *The New York Times*. Sunday, November 8, 1987, sec 6, p. 56.

Rosenberg, Jarrett — "Evaluating the Suggestiveness of Command Names" *Association for Computing Machinery*. 1981.

Roth, E. M.
Butterworth, G.
Loftus, M. J. — "The Problem of Explanation: Placing Computer Generated Answers in Context" *Proceedings of the Human Factors Society -- 29th Annual Meeting*. 1985, pp. 861 - 863.

Runciman, Colin
Hammond, Nick — "User Programs: A Way to Match Computer Systems and Human Cognition" in M.D. Harrison and A. F. Monk (eds), *People and Computers: Designing for Usability*. Cambridge: Cambridge University Press, 1986.

Sannella, Michael
et al. — *Interlisp Reference Manual*. Xerox, PARC, October, 1983.

Sebrechts, Marc — "Cognitive Guidelines for Computer-Aided Instruction: A Navigational Case Study" *Proceedings of the Human Factors Society*. 1983.

Sebrechts, Marc
Deck, Joseph
Black, John — "A Diagrammatic Approach to Computer Instruction for the Naive User" *Behavior Research Methods & Instrumentation* vol. 15(2), 1983, 200-207.

Shaw, Brian
McCauley, Michael — "Person-Computer Dialogue: A Human Engineering Data Base Supplement" Prepared for the Air Force Aerospace Medical Research Laboratory by Essex Corporation, November, 1984.

Shu, Nan C. — "FORMAL: A Forms-Oriented, Visual-Directed Application Development System" *Computer*. August, 1985, pp. 38 - 49.

Shneiderman, Ben — "Teaching Programming: A Spiral Approach to Syntax and Semantics" *Computers and Education*. Vol. 1, 1977, pp. 193 - 197.

Designing the User Interface. Reading, Mass.: Addison-Wesley, 1987.

"We can design better user interfaces: A review of Human-Computer Interaction Styles" *Ergonomics*. Vol 31, No. 5, 1988, pp. 699-710.

Sime, M. E.
Arblaster, A. T.
Green, T. R. G. — "Reducing Programming Errors in Nested Conditionals by Prescribing a Writing Procedure" *International Journal of Man-Machine Studies*. 9, 1977a, pp. 119 - 126.

"Structuring the Programmer's Task." *Journal of Occupational Psychology*. 50, 1977b, pp. 205 - 216.

Simon, Charles W. — "Will Egg-Sucking Ever Become a Science?" *Human Factors Society Bulletin*. Volume 30, Number 6, June 1987, pp. 1-4.

Smith, Brian Cantwell — "The Semantics of Clocks" *Center for the Study of Language and Information Report*. Number CSLI-87-75, March, 1987.

Smith, David Canfield
Irby, Charles
Kimball, Ralph
Verplank, Bill
Harslem, Eric — "Designing the STAR User Interface" *BYTE*. April, 1982, pp. 242 - 282.

Smith, Douglas K.
Alexander, Robert C. — *Fumbling the Future*. New York: William Morrow and Company, 1988.

Smith, Randall B. — "Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic" *Xerox Internal Report*. November, 1988. Also, see *CHI '87 Conference on Human Factors in Computing Systems*. ACM/SIGCHI, New York, 1987.

Slater, Robert — *Portraits in Silicon*. Cambridge, Mass.: The MIT Press, 1987.

Thacker, C. P.
McCreight, E. M.
Lampson, B. W.
Sproull, R. F.
Boggs, D. R. — "Alto: A Personal Computer" in D. Siewiorek, C. G. Bell, and A. Newell (eds), *Computer Structures: Principles and Examples*. New York: McGraw Hill, 1982.

Suchman, Lucy A. — *Plans and Situated Actions: The Problem of Human-Machine Communication*. New York: Cambridge University Press, 1987.

Tufte, Edward R. — *The Visual Display of Quantitative Information*. Chesire, Connecticut: Graphics Press, 1983.

Terkel, Louis (Studs) — *Working*. New York: Pantheon Books, 1974.

Wadlow, Thomas A. — "The Xerox Alto Computer" *BYTE*. September, 1981, pp. 58 - 68.

Weinberg, Gerald M.
Geller, Dennis P.
Plum, Thomas W-S — "IF-THEN-ELSE Considered Harmful" *Sigplan Notices (ACM)*. Vol. 10, 1975, pp. 34 - 44.

Youngs, E. A. — "Human Errors in Programming" *International Journal of Man-Machine Studies*. 6, 361 - 376.

SOME RELATED ARTICLES

Aaranson, Jamie
Sharp, Earl
Ancona, Don

"Target Detection as a Function of Color-Coding and Field Density" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 1066-1069.

Allen, James F.
Hayes, Patrick J.

"A Common-Sense Theory of Time" *Technical Report Computer Science Department*. Univ. of Rochester, 1985.

Apt, Charles M.
Sarkar, N. Ronnie

"The Outlook for Large-Area Flat-Panel Displays in the Computer Market" *Spectrum: Information Systems Industry Products and Technologies*. April 1987, pp. 4-3 - 4-10.

Barnsley, Michael F.
Sloan, Alan D.

"A Better Way to Compress Images" *BYTE*. January 1988, pp. 215-223.

Bennet, Charles H.

"Notes on the History of Reversible Computation" *IBM Journal of Research and Development*. Vol 32(1), January 1988, pp. 16-23.

Black, John B.
Moran, Thomas P.

"Learning and Remembering Command Names" *Conference on Human Factors in Computer Systems*. Gaithersburg, MD. 1982.

Blackman, Harold S.

Methodologies for the Evaluation of CRT Displays" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986*. pp. 1078-1081.

Brachman, Ronald J.
Henig, Fran H.

"The Emergence of Artificial Intelligence Technology" *AT&T Technical Journal*. January/February 1988, Volume 67, Issue 1, pp. 3-6.

Cammack W. B.
Rodgers H. J.

"Improving the Programming Process" *IBM Technical Report TR 00.2483*. 10/19/73.

Carroll, John M.
Campbell, Robert L.

"Softening Up hard Science: Reply to Newell and Card" *Human-computer Interaction*. 1986, Volume 2, pp. 227-249.

Charney, Davida H.
Reder, Lynne M.

"Designing Interactive Tutorials for Computer Users" *Human-Computer Interaction*. 1986 Volume 2, pp. 297-317.

Connally, Cynthia E.
Tullis, Thomas S.

"Evaluating the User Interface: Videotaping Without a Camera" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986*.

Coplien, James O.
Dewhurst, Stephen C.
Koenig, Andrew R.
"C + + : Evolving Toward a More Powerful Language" *AT&T Technical Journal.* July/August 1988 pp. 19 - 32.

Deck, Joseph
Sebrechts, Marc
"Variations on Active Learning" *reference unknown.*

Den Beste, William E.
"Capture the Intent, Not Just the Design" *VLSI Systems Design.* March 1988, p. 14.

Douglas, Sarah
Moran, Thomas
"Learning Text Editor Sematics by Analogy" *CHI '83 Proceedings.*

Drexler, Eric K.
The Engines of Creation. Garden City, NY:Anchor Press/Doubleday, 1985.

Dumas, Joseph S.
Redish, Janice
"Using Plain English in Designing the User Interface" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1207-1211.

Eastman, M. C.
Woods, D. D.
Elm, W. C.
"Specifying and Communicating Data *Structure in Computer-Based Graphic Displays" Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1034-1037.

Eike David R.
Fleger, Stephen A.
Phillips, Elizabeth R.
"User Interface Design Guidelines for Expert Troubleshooting Systems" *Proceedings of the Human Factors Society - 30th Annual Meeting -- 1986.* pp. 1025-1028.

Elm, William C.
Woods, David D.
"Getting Lost: A Case Study in Interface Design" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 927-931.

Farrell, E. J.
"Visual Interpretation of Complex Data" *IBM Systems Journal.* Vol. 26, No. 2, 1987.

Finke, Ronald A.
"Mental Imagery and the Visual System" *Scientific American* (unknown date).

Fisk, Arthur D.
Scerbo, Mark W.
Kobylak, Richard F.
"Relative Value of Pictures and Text In Conveying Information: Performance and Memory Evaluations" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1269-1272.

Fraser, Christopher W.
Myers, Eugene W.
"An Editor for Revision Control" *ACM Transactions on Programming Languages and Systems.* Vol. 9, No. 2, April 1987, pp. 277-295.

Gadenne, Francois G.
Braunschvig, David G. "Beyond the AI Winter: Integration as the First Step to Managing Knowledge" *Spectrum: Information Systems Industry Products and Technology.* June 1987, pp. 5-1 - 5-8.

Garner, Wendell R. "The Aquisition and Application of Knowledge: A Symbiotic Relationship" *American Psychologist.* October, 1972, p. 941.

Gordon, Fred
Isenhour, Robert "Simultaneous Engineering" *Engineering Manager.* January 30, 1989, pp. 4-5.

Gould, John D. "Looking at Pictures" in *Eye movements in Psychological Processes.* pp. 323-345.

Gould, John D.
Alfaro, Lizette
Finn, Rich
Haupt, Brian
Minuto, Angela
Salaun, Josiane "Why is Reading Slower from CRT Displays Than From Paper" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 834-836.

Gould, John D.
Grischkowsky, Nancy "Effects of Visual Angle On Reading" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 1106-1109.

Green, Paul
Wei-Haas, Lisa "The Rapid Development of User Interfaces: Experience with the Wizard of Oz Method" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.*

Harel, David "Statecharts: A Visual Formalism for Complex Systems" *Science of Computer Programming.* (8), 1987, 1-29.

Hill, Susan G.
Kroemer, Karl H. E. "Preferred Declination of the Line of Sight" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 625-629.

Hirschberg, Julia
Ballard, Bruce W.
Hindle, Donald "Natural Language Processing" *AT&T Technical Journal.* January/February 1988, Volume 67, Issue 1, pp. 41-57.

Hoffman, Susan "Integration of Principles of Human Factors into an Undergraduate Computer Technology Curriculum: A Case Study" *Proceeding of the Human Factors Society.* 1985.

Howard, James H. Jr. "Spatial Scale in Detection and Recognition: Implications for Image Processing" *Proceedings of the Human Factors Society --29th Annual Meeting -- 1985.* pp. 1114-1117.

Irons, Dennis M. "Cognitive Correlates of Programming Tasks in Novice Programmers" *Conference on Human Factors in Computer Systems.* Gaithersburg, MD., March, 1982.

Johnson, Steven L. "Using Mathematical Models of the Learning Curve in Training System Design" *Proceedings of the Human Factors Society*. 1985.

Keough, Lee "The New Face of Computing" *Computer Decisions*. February, 1989, pp. 36-41.

Knapp, Beverly G. "The Precedence of Global Features in the Perception of Map Symbols" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 287-291.

Koubek, Richard J.
Janardan, Chaya Garg "A Basis for Explaining the Conflicting Results in Performance on CRT and Paper Displays" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 1102-1105.

Laurel, Brenda K. "Interface as Mimesis" in *User Centered System Design, New Perspectives on Human-Computer Interaction*. Donald A. Norman & Stephen W. Draper (eds.), Lawrence Erlbaum Associates:Hillsdale, N.J., 1986.

Lawler, Robert W. "The Progressive Construction of Mind" *Cognitive Science* 5, 1-30 (1981).

Lee, Kwan S.
Oh, Yeon G. "An Alternative to Reduce the Physical Stress of VDT Operators" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 932-936.

Lewis, Clayton
Mack, Robert "Learning to use a Text Processing System: Evidence from 'Thinking Aloud' Protocols" *Association for Computing Machinery*, 1981.

MacGregor, Donald
Slovic, Paul "Graphic Representation of Judgemental Information" *Human-Computer Interaction*. 1986, Volume 2, pp. 179-200.

Marcus, Aaron
Cowan, William B.
Smith, Wanda "Color in User Interface Design: Functionality and Aesthetics" *CHI '89 Proceedings*. pp. 25-27.

Meyer, Orville R. "Assessment of Major Changes in User Computer Interfaces in 1990 to 2000" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986*. pp. 1086-1088.

Milbkn, Michael "Compund Documents Take the Stage" *Patricia Seybold's Office Computing Report*. Vol. 12, No. 2, Feb. 1989, pp. 1-11.

Miller, Lance A. "Programming by Non-Programmers" *International Journal of Man-Machine Studies*. 6, 1974, pp. 237-260.

Mills, H. D.
Linger, R. C.

Hevner, A. R. "Box Structured Information Systems" *IBM Systems Journal*. Vol 26, No. 4, 1987, pp. 395-413.

Morse, Robert S. "Glare Filter Preference: Influence of Subjective and Objective Indices of Glare, Sharpness, Brightness, Contrast, and Color" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 782-786.

Mourelatos, Alexander P. "Events, Processes, and States" *Linguistics and Philosophy* 2 (1978) pp. 415-434.

Myers, Greta
Fisk, Arthur "Toward a Skill Eased Training Technology: Review, Evaluation, and Application of Automatic/Controlled Processing Training Guidelines" *Proceedings of the Human Factors Society*. 1985.

Neilson, Jakob "Classification of Dialog Techniques: A CHI + GI '87 Workshop" *SIGCHI Bulletin*. October 1987, Volume 19, Number 2, pp. 30-35.

Neilsen, Jakob "Coordinating User Interfaces for Consistency" *SIGCHI Bulletin*. January, 1989, pp. 63-65.

Nelson, William R. "Operator Aids and Expert Systems in User Computer Interfaces" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986*. pp. 1082-1085.

Normore, Lorraine F.
Tijerina, Louis "Evaluation of Guidelines for Designing User Interface Software" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1986*. pp. 1363-1365.

Ogden, William
Kaplan, Craig "The use of 'and' and 'or' in a natural language computer interface" Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986, pp. 829-833.

Penchas, Samuel "High-tech transference: overcoming obstacles in the application of knowledge" *Ergonomics*. 1987, vol. 30, No. 2, 185-192.

Pisoni, David B. "A Brief Overview of Speech Synthesis and Recognition Technologies" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986*. pp. 1326-1330.

Pisoni, David B.
Nusbaum, Howard C. "Developing Methods for Assessing the Performance of Speech Synthesis and Recognition Systems" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986,*.pp. 1344-1348.

Reddy, Thota V.
Bennett, Corwin A. "Cultural Differences in Color Preferences" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985*. pp. 590-593.

Richards, Robert E.
Gilmore, Wlater E.
Haney, Lon N.
"Human Factors Guidelines and Methodology in the Design of a Computer Interface: A Case Study" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1073-1077.

Robertson, Scott P.
Black, John B.
"Structure and Development of Plan in Computer Text Editing" *Human-computer Interaction.* 1986, Volume 2, pp. 201-226.

Rosch, Eleanor
Mervis, Carolyn B.
"Family Resmblances:Studies in the Internal Structures of Categories" *Cognitive Psychology.* 7, 573-605 (1975).

Rosen, Brian
Kriz, Stan
"Case Study: Developing a 3000 line Interactive CRT display" *Information Display.* Vol. 4(1), Dec. 1987, pp. 12-15.

Roth, E. M.
Butterworth, G.
Loftus, M. J.
"The Problem of Explanation: Placing Computer Generated Answers in Context" *Proceedings of the Human Factors Society.* 1985.

Sanders, Elizabeth B.-N.
"Toward a Theory of Information Design" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1068-1071.

Sawyer, Dick
Talley, Walter T.
"Display Legibility Guidelines" *Information Display.* Vol. 3(11), Dec. 1987, pp. 13-16.

Schatz, Joseph G.
Bender, Warren G.
"Intelligence in the Public Network of the Future" *Spectrum - Information Systems Industry Products and Technologies.* October, 1987, pp. 2-19 - 2-26.

Schneiderman, Ben
"Teaching Programming: A Spiral Approach to Syntax and Semantics" *Computers and Education.* Vol 1, pp. 193-197, 1977.

Schultz, E. Eugene
Nichols, David A.
Curran, Patrick S.
"Decluttering Methods for High Density Computer-Generated Graphic Displays" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 300-302.

Shannon, Claude
"A Mathematical Theory of Communication" *The Bell System Technical Journal.* July 1948.

Shoham, Yoav
"Chronological Ignorance: Time, Nonmonotonicity, Necessity, and Causal Theories" *Science.* (date unknown), 389-393.

Silver, Laura D. "Displays, Aquisition, and Performance" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 620-624.

Singleton, W. T. "The Design of Design Teams in High-Technology Industries" *Applied Ergonomics*, 1987, 18.2, 111-114.

Smith, Michael J. "VDT Strain: Psychological or Physical Basis?" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 689-693.

Smith, S. L.
Aucella, A. F. "Design Guidelines for the User Interface to Computer-Based Information Systems" *Design Guidelines for the User Interface to Computer-Based Information Systems (MTR - 8857).* Bedford, Mass: MITRE Corp., 1983.

Stoddard, Mary L. "Critical Variables in Start-Up Training for Multi-Functional Information Systems" *Proceedings of the Human Factors Society.* 1985.

Stroud, John M. "The Fine Structure of Psychological Time" *Annals of the New York Acadamy of Sciences.* 1966, pp. 623-631.

Tijerina, Louis "Design Guidelines and the Human Factors of User Interface Design" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 1358-1362.

Tolin, Philip "Maintaining the Three-Dimensional Illusion" *Information Display.* Vol. 3(11), Dec. 1987, pp. 10-12.

Tullis, Thomas S. "A System for Evaluating Screen Formats" *Proceedings of the Human Factors Society -- 30th Annual Meeting -- 1986.* pp. 1216-1220.

Turner, Marylin L.
Engle, Randall W. "Working Memory Capacity" *Proceedings of the Human Factors Society - 30th Annual Meeting -- 1986.* pp. 1273-1277.

Vere, Steven A. "Planning in Time: Windows and Durations for Activities and Goals" *IEEE Transactions on Pattern Analysis and Machine Intelligence.* Vol. PAMI-5, No. 3, May 1983, pp. 246-267.

Weiss, Ray "C Programmers push for Object-Orientation" *Electronic Engineering Times.* April 10, 1989, pp. 61-73.

Weitzman, Donald O. "Color Coding Re-Viewed" *Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985.* pp. 1079-1083.

Weldon, Linda
Koved, Larry
Schneiderman, Ben "The Structure of Information in Online and Paper Technical Manuals" *Proceedings of the Human Factors Society.* 1985.

Wilson, Denise L.
Burns, Kevin R.
Kuperman, Gilbert G. "Feature Positive Coding Effects on Object Localiztion"

Proceedings of the Human Factors Society -- 29th Annual Meeting -- 1985. pp. 292-296.

Woods, David D. "Knowledge Based Development of Graphic Display Systems" *Proceedings of the Human Factors Society -- 29th Annual Meeting - 1985.* pp. 325-329.

Young, Richard M.
Green, T. R. G.
Simon, Tony "Programmable User Models for Predictive Evaluation of Interface Designs" *CHI '89 Proceedings.* pp. 15-19.