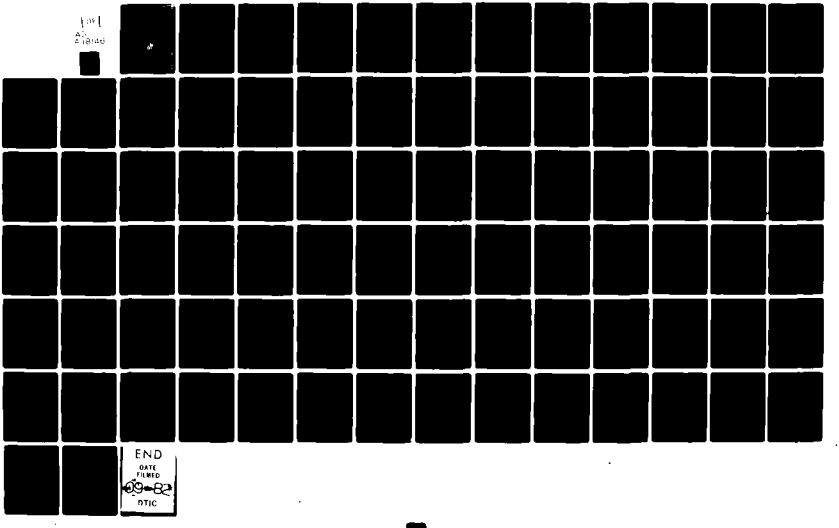
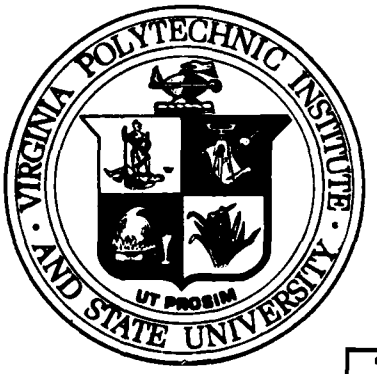
AD-A118 146 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG --ETC F/G 5/8
THE ROLE AND TOOLS OF A DIALOGUE AUTHOR IN CREATING HUMAN-COMPU--ETC(U)
MAY 82 D H JOHNSON, H R HARTSON N00014-81-K-0143
UNCLASSIFIED CSIE-82-8 NL



END
DATE FILMED
DTIC



Virginia Polytechnic Institute
and State University
Computer Science
Industrial Engineering and Operations Research
BLACKSBURG, VIRGINIA 24061

CSIE-82-8 May 1982

(6)

THE ROLE AND TOOLS OF A DIALOGUE AUTHOR IN CREATING HUMAN-COMPUTER INTERFACES

Deborah H. Johnson

H. Rex Hartson

TECHNICAL REPORT

Prepared for
Engineering Psychology Programs, Office of Naval Research
ONR Contract Number N00014-81-K-0143
Work Unit Number NR SRO-101

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CSIE-82-8	2. GOVT ACCESSION NO. AD A118146	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE ROLE AND TOOLS OF A DIALOGUE AUTHOR IN CREATING HUMAN-COMPUTER INTERFACES		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Deborah H. Johnson H. Rex Hartson		8. CONTRACT OR GRANT NUMBER(s) N00014-81-K-0143
9. PERFORMING ORGANIZATION NAME AND ADDRESS Computer Science Virginia Polytechnic Institute & State University Blacksburg, VA 24061		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 61153N42; RR04209; RR0420901; NR SRO-101
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research, Code 422 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE May 1982 13. NUMBER OF PAGES 79
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

human-computer interface, human factors, dialogue author, dialogue independence internal dialogue, external dialogue

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

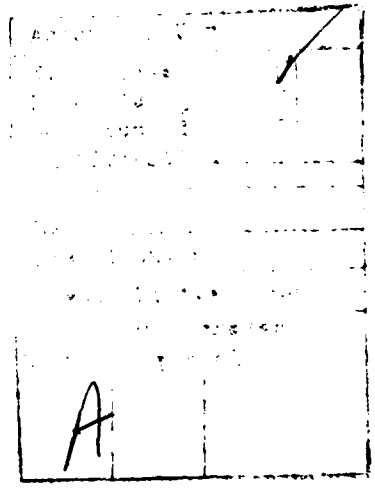
In order to facilitate the development of human-factored human-computer interfaces, a Dialogue Management System (DMS) is being created. Dialogue independence and internal and external dialogue have developed as underlying concepts of DMS, and are manifest in the separation of the dialogue components of a software system from the computational components. In a new system design role, a dialogue author is responsible for creating the dialogue which constitutes the human-computer interface of an application system. DMS con-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (Continued)

sists of a comprehensive structured methodology, a multiprocess execution environment, and a set of automated tools for both a dialogue author and an application programmer to use in creating human-factored human-computer systems. This report presents an overview of DMS and its development. The role of the dialogue author and the tools in the Author's Interactive Dialogue Design Environment (AIDE) are discussed in detail.



ACKNOWLEDGMENT

The authors wish to express appreciation to Roger Ehrich, Rick Johnson, and Tamer Yunten for their help in preparing this report.

This research was supported by the Office of Naval Research under contract number N00014-81-K-0413 and work unit number NR-SRO-101. The effort was supported by the Engineering Psychology Group, Office of Naval Research under the technical direction of Dr. John J. O'Hare. Reproduction in whole or in part is permitted for any purpose of the United States Government.

The Role and Tools of A Dialogue Author In Creating Human-Computer Interfaces

Deborah H. Johnson

H. Rex Hartson

"...After a total of 70 hours with the computers..., I remained totally incapable of utilizing them to suit the needs I would have bought them for."

1. INTRODUCTION

The above quote from the cover article of the February 22, 1982 issue of Newsweek details a naive user's frustration at attempting to learn to use a computer for the first time. Unfortunately, the experience described in that article is not an isolated incident. As computers become an ever-increasing part of twentieth-century life, the need for human interaction with them to be simple and efficient increases. The human aspect of computer systems design has also become more important as the spectrum of user diversity has widened. The user community of any given computer installation is potentially highly varied, ranging from the very specialized systems programmer to the engineer who needs some "numbers crunched" to the nervous

secretary using a text editor for the first time. Thus, the design of systems that will satisfy the needs of, and provide a friendly environment for, a diverse user population becomes increasingly important. As a result, the human factors aspect of computer system design both in hardware and in software has become one of the fastest growing fields in the computer industry. "Software psychology" and "human-computer communication" are only two of the numerous new "buzz words" that have arisen in this area. "Human factors" has long been an industrial engineering term that conjured up questions of where to place the knobs on a stove and how high a chair should be. Today, human factors in computer system development is of utmost importance in producing systems that are effectively and easily usable by humans.

At Virginia Tech, a research project on human interaction with computers, sponsored by the Office of Naval Research, is underway. This project, a joint research effort of the Computer Science Department and the Industrial Engineering/Operations Research Department, has as its goal the development of an environment for interactive design of effective human-computer interfaces, as well as empirical testing and evaluation of this design. One aspect of this interdisciplinary research is focused on the creation of human-factored human-computer dialogues. In current software development, human-computer dialogues are typically written by programmers who generally have little or no formal training or even intuitive feeling for what constitutes an effective human-computer dialogue. Consequently, typical human-computer interfaces may be cryptic, puzzling, frightening, frustrating, and even totally unusable, especially by those inexperienced in computer interaction. Thus, a need for specialists skilled in writing human-computer dialogues arises. A new role for just such a specialist is being developed as

part of our research. Called a dialogue author, this person is responsible for designing and implementing human-factored dialogues. An application programmer still writes all computational portions of the software, but writes no dialogues. In fact, the dialogue and computational software components are separate in the implemented, executing system. Additionally, this design allows both computational and dialogue components to be modified independently, without necessarily requiring revision of the other. This separation of dialogue and computation is called dialogue independence and is the basis for a novel new approach to system development. The dialogue author and the application programmer interact closely during system design and implementation; in fact, part of our research is directed toward communication issues between these two roles. All of these topics will be further addressed later in this paper.

In order to amplify and evaluate these ideas, a Dialogue Management System (DMS) is being developed which integrates all these aspects of our research. DMS is more than an execution environment for a typical operational software system. Additionally, it consists of both a unified methodology and a set of automated aids which guides and facilitates the design of human-factored computer systems. Many of these tools, in an integrated environment called AIDE (Author's Interactive Dialogue design Environment), are used by a dialogue author to create dialogues that form human-computer interfaces.

In a very general sense, DMS can be viewed as a hierarchical, tree-structured system as shown in figure 1. At the top level is DMS itself, which consists of three main subsystems: the dialogue author's interface (AIDE), the analogous application programmer's interface, and the multipro-

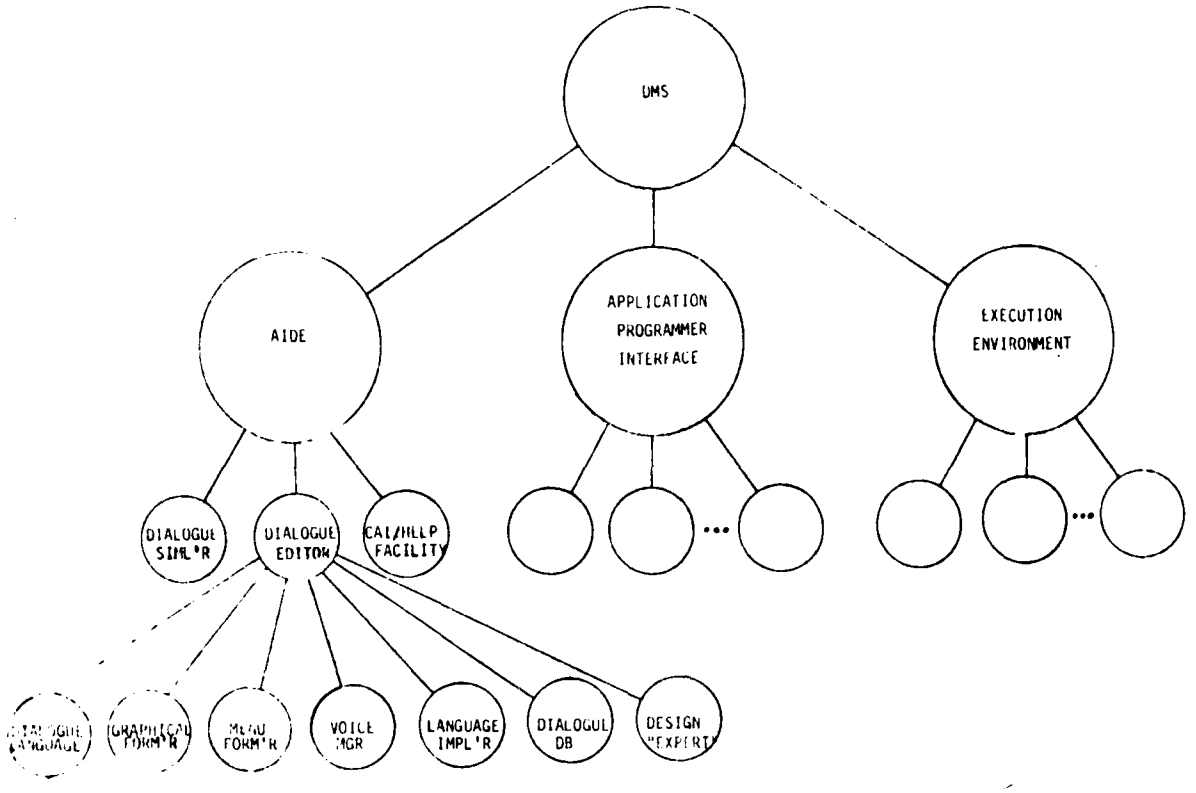


Figure 1. Hierarchical Development of DMS

cess execution environment of these two components as well as all application systems they are used to create. At the third level from the top, AIDE consists of the various tools for the dialogue author, specifically a dialogue simulator and a dialogue editor, including its subtools such as a dialogue language, a graphical formatter, a language implementer, a dialogue database system, and a dialogue design "expert." Each of these tools will be discussed in detail later. Similarly, the application programmer's environment is composed of its various tools for use by the application programmer. The execution environment is made up of those system components which control execution of AIDE, the programmer's environment, and the application

systems that are developed. This multiprocess environment necessitates the use of interprocess communication procedures which will be addressed in more detail in a later section. Underlying this level are those elements which comprise the specific tools.

The DMS development team has taken a top-down approach in the design of DMS. That is, the high-level elements (AIDE, the programming environment, and the execution environment) have been identified, and then each of these broken into its various constituents. This process continues down the tree. The purview of this report is the dialogue author's environment (AIDE), including the design and implementation of the specific tools of AIDE.

Thus, this report will focus on the role of the dialogue author and its place in human-computer interface development, particularly as it relates to DMS. The automated tools of AIDE and the methodology which are also a part of DMS and which allow the dialogue author to work effectively and efficiently will also be discussed. The ideas presented here are, at this time, primarily theoretical in nature. The purpose of this paper is therefore neither to report on the implementation of AIDE and DMS, nor to give an evaluation or commentary of real-world, hands-on experiences with such a system. Rather, this report serves as a high-level proposal for the overall design and architectural organization of this type of system, particularly including the various components that we presently envision as an integral part of it. The next step in our research is, of course, a shift to the design, implementation, and evaluation of the environment and the tools described here. The authors are presently working on the development of a language implementer, a component of AIDE which will be used in the design, representation, and recognition of command languages. This tool will be discussed in detail in a subsequent report.

2. CURRENT LITERATURE

About a decade ago, interest in human-oriented computer interface design first began to appear. James Martin, in Design of Man-Computer Dialogues, [MARTJ73] discusses numerous facets of human-computer interaction and their relationship to design of effective human-computer dialogues. Intended to be a comprehensive guide to development of human-computer dialogues, the book addresses interactions with traditional CRTs as well as graphics terminals, voice units, and alternative I/O devices. It also considers user psychology as related to a computer system, quite a novel idea nearly ten years ago! Many of the ideas and recommendations put forth in this book are still highly applicable today, a testament to Martin's farsightedness. However, one issue which Martin's book does not address in any detail is that of empirical evaluation of guidelines and principles for dialogue design. The importance of quantitative evaluation of the human-computer interface is greatly emphasized in Ben Shneiderman's book on Software Psychology: Human Factors in Computer and Information Systems [SHNEB80]. As Shneiderman suggests, technological advancement is encouraged by understanding and incorporating fundamental principles, which, in turn, should be based on experimental results.

A few journal articles dealing with effective interface design also began to appear in the mid-seventies. Papers by D.R. Cheriton [CHERD76],

T.C.S. Kennedy [KENNT74], and J. Foley [FOLEJ74] enumerated several ways in which both textual and graphical interface components could be made more human oriented. Today, entire issues of journals are being dedicated to research developments in this field. Specifically, such areas as the psychology of human-computer interaction [ACMCS81], dialogue design [ERGON80], and human factors in computing [IBMSJ81] have each received the attention of a special issue of a major periodical. Also, as cited at the beginning of this report, such widely circulated, non-technical publications as Newsweek are recognizing and addressing the influx of computers into everyday life. Additionally, several recent conferences have been devoted to human factors in computer systems ([SIGS081], [HFICS82]), providing forums for presenting "state-of-the-art" research and for the exchange of ideas.

3. DIALOGUE : A NEW CONTEXT

3.1. TRADITIONAL SYSTEMS AND THE ASSOCIATED PROBLEMS

As is common knowledge to anyone who has ever done any software development, traditional software systems design is aimed mainly at development of functional computational code. Since getting the system debugged and operational depends mainly on the correctness of the computational part of the software, the "correctness" (or effectiveness) of the dialogue part gets little consideration. But this dialogue comprises the all-important inter-

face with which human users must interact. If that interface is not created so that most humans can perform the task they set out to do, it matters little whether the computational part of the code works properly or not. Also in traditional systems, both computational code and dialogue generally are written by an application programmer and therefore are interspersed, so that the dialogue is actually a part of the computational software. This often makes it difficult to easily change either. The dialogue to be altered must first be found in a maze of functional code and then the desired changes made. However, when the dialogue is embedded in the computational code, changes to dialogue propagate all the problems associated with changes to the computational program. Since an application programmer typically writes the dialogues that form the human-computer interface, these dialogues are likely to be poorly human-engineered. An application programmer rarely has either the skills and knowledge or the time and patience to create effective, human-factored dialogues.

3.2. FUNCTIONS OF HUMAN-COMPUTER DIALOGUE

Dialogue, as it relates to a human-computer interface in the traditional sense, refers to the interaction between a human and the computer system being used. It is the means by which the human gives commands or queries to the computer and by which the computer responds to or queries the user. This discourse is thus the actual exchange of words, phrases, parameterized commands, and other symbols and actions, i.e., a conversation, between a human and a computer.

This human-computer dialogue is composed of two separate parts: the computer's part, which is actually determined in the software, and the human's part, which is any input a user may give to the system. Thus, it is confusing and indeed erroneous to think of a human-computer dialogue as being only what is actually displayed to the user by the system, i.e., what is coded into the software itself. Instead, both the human side and the computer side of the dialogue must be considered in the design of a specific discourse.

Human-computer dialogue has two distinct functions: 1) to request information (either from the user or from the computer) and 2) to transmit information (either to the user or to the computer). Most programs, in order to execute, must have input (e.g., "name", "SSN", "edit", numerical data) from the user. This input is typically obtained through prompts or queries which ask the user for specific information. These requests are coded at appropriate points into the computational software itself. When the system needs to transmit information to the user (e.g., "searching database", "unmatched left parenthesis"), this is also done through some predetermined dialogue built into the software. The content and format, as well as the logical sequencing, of this dialogue is extremely important in determining how well the user can understand and manipulate the system. Finally, a combination of transmitting and requesting information is manifest in a menu, which displays possible options and then asks the user to choose one. In each of these instances, the human is interacting directly with output generated from the software itself. Thus the interface between the human and the system is a dialogue, in its traditional sense. (See figure 2.)

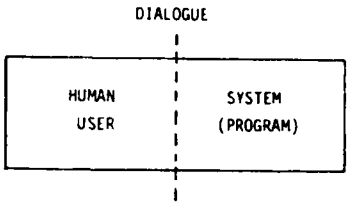


Figure 2. Traditional Human-Computer Dialogue

3.3. DIALOGUE INDEPENDENCE

One important criterion for the design of effective human-computer interfaces is fast, easy modification of the dialogue of a specific system. Because part of our research involves empirical studies on human-computer interaction and dialogue design, this requirement is particularly important. The dialogue may need revision both for the experiments themselves, as well as after the experiments, to incorporate the results of the experiment back into the dialogues. Obviously, our experimentation will be pointless if the principles learned from it are not utilized in our own system!

For several years, database researchers and designers have known that data independence is a key issue in database development. Data independence is a concept that directs database design in such a way that changes to data are independent from, and therefore do not usually necessitate changes to, the application program which manipulates that data. Dialogue independence is an analogous concept, in which changes to dialogue do not mandate modifications to computational code.

In traditional software systems, dialogue and computational code are inextricably interwoven together so that the need for a change to the dialogue may require searching through countless pages of source code to find the lines that need to be changed. Indeed, following a program's computational logic is very difficult when READs and WRITEs to request and transmit information between the user and the system are constantly interrupting the natural flow of the computational code. Dialogue independence will help to alleviate some of these difficulties by allowing easy alteration of dialogues without the associated problems of changing computational code. Dialogue independence is achieved by both logical and physical separation of dialogue from computational software. The part of the software that does computation will be logically distinct from the dialogue part as early as the system analysis phase. This division will continue and be physically manifest at implementation time when the dialogue author creates the dialogue component and the programmer creates the computational component of the system. The two components will communicate by procedure calls and parameter passing between themselves and will be bound together either before or at execution time.

3.4. INTERNAL AND EXTERNAL DIALOGUES

Under DMS, the computational program contains no mechanism for direct communication with the user; i.e., it contains no dialogue in the traditional sense, since the dialogue which interfaces with the user is separate from the computational component of the program. Under DMS, at the lowest level, any specific application system consists of distinct dialogue modules

and computational modules. The dialogue modules contain the actual text which is displayed to the user, and where necessary, contain the mechanisms for receiving user input and transmitting it to the computational program. The computational modules, in the usual software engineering sense, process data and information, but have no direct interaction with the user. When the computational component needs information or data from the user to continue processing, it makes a call to the appropriate dialogue module, which then interacts with the user to obtain the necessary information and return it to the computational module. Similarly, when the computational component needs to transmit information to the user, it again calls the appropriate dialogue module which displays the information to the user. The collection of computational modules thus interfaces with the user only through the dialogue modules. From the traditional point of view, this collection of computational modules is a dialogue-free program. Further thought, though,

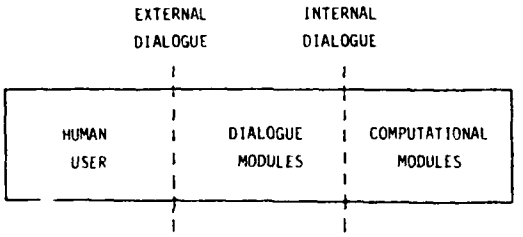


Figure 3. Internal and External Dialogues

reveals that it is indeed not dialogue-free (see figure 3), but is carrying on an "internal dialogue" with the dialogue modules so that the program can obtain data in order to execute.

The human's portion of the dialogue is the interaction between the user and the dialogue modules. This "external dialogue" is the interaction between the system's outputs via the dialogue modules and the user's inputs. External dialogue is thus the interaction which is typically thought of as the traditional human-computer interface. An external dialogue may be highly varying in form and content. The part expressed to the user by the computer is limited only by the imagination of the dialogue author and principles of good human-computer communication. Similarly, the possible inputs that the human can give to the computer are virtually infinite. Certainly only a very small subset of these possible inputs will be valid, but the potential for the others still exists and must be dealt with by appropriate error messages.

A multi-process execution environment in which dialogue modules and computational modules are coordinated at run-time has been developed in a closely related research effort. This environment will not be directly addressed in this paper, but is discussed in detail in "The DMS Multiprocess Execution Environment" [EHRR82b] and in "DMS - A System for Defining and Managing Human-Computer Dialogues" [EHRR82a].

4. THE DIALOGUE AUTHOR : A NEW ROLE

4.1. TRADITIONAL ROLES IN SYSTEM DESIGN

For many years, the two main roles involved in software development were those of the application programmer and the end user of the system. These two types, however, frequently had severe communication problems. The programmer, impatient to begin coding, often had difficulty understanding the user's requirements and needs for the system. Similarly, the baffled user had trouble understanding the strange "computerese" in which the programmer tried to explain what was happening. The unsatisfactory, insufficient communication between these two often resulted in systems that were not what the user wanted or needed, but what the programmer decided to provide. Gradually, the need was recognized for someone who could understand both the technical (programmer) and non-technical (user) sides of the system. This role is that of a systems analyst.

In addition, the user might possibly be knowledgeable in only the one or two facets of the system that were used most. So the role of application expert evolved, one who knew all aspects of system requirements and uses thoroughly. The systems analyst serves as a liaison between application expert (and/or users) and application programmer, aiding and guiding both in defining and communicating system requirements. But neither the systems analyst nor the application expert was concerned primarily and explicitly with the human-computer interface. In the last few years, human factors specialists have become an increasingly important part of computer system

design and development teams for just this reason. Obviously, they attempt to emphasize the incorporation of human factors guidelines into computer systems, and especially the dialogues, so that the user has an effective interface with which to interact.

But a human factors expert frequently knows little about implementation of dialogues. Thus, a new role, that of a dialogue author, has been created to design and implement dialogues.

4.2. FUNCTIONS OF THE DIALOGUE AUTHOR

The independence of dialogue modules from computational modules indicates the need for separate roles to create these two components. Thus, the role of a dialogue author has been developed. The dialogue author is a skilled specialist whose ultimate goal is to create and implement dialogue modules using automated tools (to be discussed later) provided by DMS. Both the tools and the methods used will be aimed at incorporating human factors principles into the dialogues. Because application programmers will not be writing dialogues under DMS, the dialogue author "will have sole responsibility for creating an interface for a user. The dialogue author is not specifically a programmer, but rather is a specially trained, human factors-oriented professional concerned with the high-level sequencing of computer-user interaction, as well as the form, style, and content of specific dialogues" [HARTH82]. Thus, the dialogue author, even though creating the dialogue component of a software system, does not program the dialogue modules in the way that an application programmer programs the computational modules. Instead, the dialogue author has a special environment designed

specifically to aid in the implementation of dialogue modules. This environment will be discussed in detail later.

In addition to simply creating human-factored dialogues, the dialogue author is interested in evaluation and revision of dialogue modules. Particularly since our research encompasses empirical testing of dialogues, the dialogue author is interested in metering the dialogues and in incorporating newly discovered principles and guidelines into the dialogues. Again, the separation of dialogue and computational components of the software through dialogue independence allows these modifications to be made quickly.

Communication with a machine is very different from communication with a human. Thus, an application programmer, whose skill has been developed to deal with computers, is not a likely candidate to produce human-oriented dialogue. The knowledge and skills required to create effective human-computer interactions are quite different from those needed to write computational software. The programmer may even find it annoying in the midst of a logically complicated section of code to have to "change modes" and write dialogue to interact with the user, for example, to solicit an input value. It is even more distracting from the computational programming task to have to check the value received from the user for all possible errors and to respond, in case of an error, with consistently worded error messages and prompts for correct inputs. Thus, these interactions are frequently written as quickly as possible, with little thought given to their form or content. On the other hand, a dialogue author, who has been trained to utilize human factors principles in dialogue design, and whose sole task is creation of human-engineered dialogues, is able to produce consistent, understandable dialogues and thus create a better human-computer interface.

4.3. A UNIFIED METHODOLOGY

4.3.1. Parallel Methodology for Author and Programmer

The addition of the dialogue author to the overall system design team increases communication complexity in an already sizable group (including application programmer, systems analyst, application expert, user, and human factors expert). One of the most important areas of our research is that of development of a design methodology that facilitates inter-role communication between the dialogue author and the application programmer.

The need for a structured methodology for the development of software systems is widely recognized. Part of our research efforts have been directed toward the creation of a system development methodology that integrates methods of human factors analysis with software engineering. The addition of a dialogue author to the system design cycle necessitates modification of the methodology so that this role is clearly delineated from the traditional role of application programmer. This approach facilitates inter-role communication between the two roles. A theoretical presentation of the complete methodology is discussed in a separate technical report entitled "Human-Computer System Development Methodology for the Dialogue Management System" [YUNTT82]. An example of its application to the design and implementation of one of the tools of AIDE (the menu formatter) is given in "Application of the DMS Methodology to the Development of the Menu Formatter, an AIDE Tool" [JOHD82a].

The main phases of the methodology are requirements specification, analysis, design, implementation, and maintenance. The requirements specification phase involves defining and understanding the problem, so that the initial system requirements can be determined. Analysis attempts to define a solution to the problem, so that system functions, objects, and sequences can be determined without regard to implementation issues. The result is a logical model of the system, with careful delineation between dialogue functions and computational functions. During the design phase, the modular structures of the functionally independent dialogue and computational components are designed. Physical system documentation, produced during the design stage, allows the dialogue author and the application programmer to proceed with actually writing dialogue and computational modules in the subsequent implementation phase. Finally, once the application system is completed and put into use, deficiencies and the associated modifications are determined. Human factors experts, systems analysts, application experts, and, of course, the dialogue author and the application programmer are all a part of this development cycle. The systems analyst and human factors person are mostly responsible for the creation, at the analysis and design phases, of the system documentation which allows the dialogue author and the application programmer to write their respective modules, as well as to communicate effectively.

The dialogue author interfaces with the application programmer primarily during system implementation. This communication raises numerous issues that must be resolved. For example, the dialogue author must know when a dialogue is needed, and what the function of that particular dialogue will be. The parameters to be passed between dialogue modules and computational

modules must be specified and understood by both dialogue author and application programmer. Such problems as input data type checking and verification, which can be a fairly complex programming task, must be considered.

In order to resolve some of these problems, formal declarative specifications are necessary. These specifications detail, in a format easily understandable, usable, and modifiable by both dialogue author and programmer, exactly what is needed by both dialogue modules and computational modules, i.e., the precise separation of the two components at their interface. Thus, the specifications for the separation of and communication between dialogue and computational components constitute the interface between the dialogue author and the application programmer. These specifications for communication between the dialogue modules and the computational modules are created in the analysis and design phases of the system development process. During implementation, these specifications are manifest as the internal dialogue between dialogue modules and computational modules.

4.3.2. Application of Methodology to DMS Development

The group of researchers working on DMS design and implementation is, in fact, utilizing the basic format of this methodology at a very high level in the development of DMS itself. This report informally represents the requirements specification and analysis phases of system design, by addressing the issue of what functions or tools are needed in DMS, and incorporating these needs into a written form of initial system requirements. The next phase, that of system design, will result in a plan for creating the various DMS system components in a prioritized and integrated manner. This

will be followed by actual implementation of these parts according to the predetermined design. Finally, the operational DMS will be used by both the dialogue author and the application programmer to create other human-factored application systems.

5. DIALOGUE MANAGEMENT

5.1. OVERVIEW OF DMS

Figure 4 is a high-level structural diagram of the organization of DMS. Because DMS is used to develop human-engineered application systems, it must provide an environment which facilitates the design and implementation of both dialogue and computational software in parallel. The components of the environment that effect this parallel development are shown.

The application programmer, through the application programmer interface, uses the computational development facility to create the computational component of the application system. Only computational code is developed through this portion of DMS. No external dialogue, or interaction with the user, is produced. However, the internal dialogue, or the interaction between the dialogue and computational components, is a part of this software.

At the same time, the dialogue author, through the dialogue author's interface, uses the Author's Interactive Dialogue design Environment (AIDE)

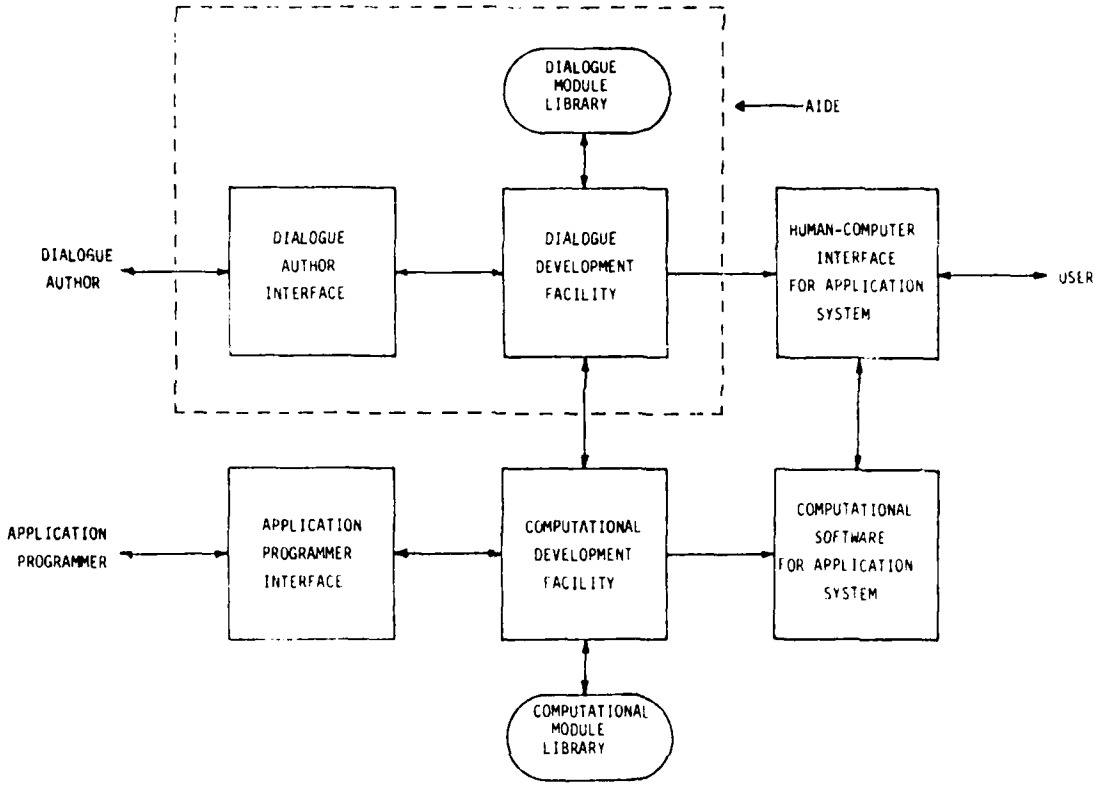


Figure 4. High-Level Organization of DMS

to create the dialogue portion of the application system. This is actually the external dialogue, or the interface through which the user communicates with the application system. The components of DMS which the dialogue author uses to create dialogue are shown in the dotted box in figure 4. AIDE is the subject of many of the remaining sections of this paper.

The integrated dialogue and computational components form the completed application system. The dialogue component is actually the human-computer

interface with which the user interacts. DMS itself is comprised of human-engineered dialogue and computational software that form this tool and the interactive interface for its users (the dialogue author and the application programmer). That is, DMS provides a human-engineered application system whose interface is used to create other human-engineered application systems and their interfaces.

5.2. AUTHOR'S INTERACTIVE DIALOGUE DESIGN ENVIRONMENT (AIDE)

5.2.1. Purpose and Architectural Structure

The Author's Interactive Dialogue design Environment (AIDE) is shown in figure 5. This is the portion of the complete DMS which a dialogue author uses to design, implement, simulate, test, and modify dialogues. AIDE provides tools that assist the dialogue author in the creation of dialogues. The dialogue author interface is the communication link between the dialogue author and these tools, while the coordinator is the communication link among these tools and with the underlying host computer system. Thus, the dialogue author interface is itself an external dialogue consisting of the dialogue component of the tools. Similarly, the computational component of the tools forms an internal dialogue with the coordinator. Each of these components of AIDE is further discussed below.

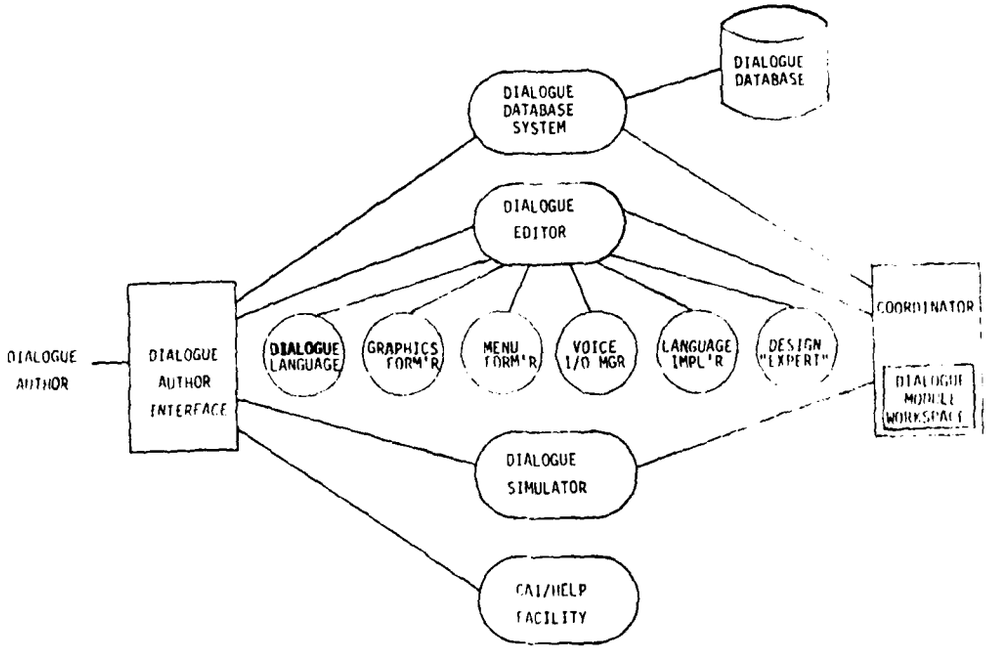


Figure 5. Author's Interactive Dialogue Design Environment (AIDE)

5.2.2. The Tools: An Overview

Numerous tools for the dialogue author are necessary to provide a human-factored interface for the author to use in the design and implementation of human-factored dialogues. At the highest level, the dialogue editor and the dialogue simulator are the two main tools which comprise the interface through which the dialogue author works. The dialogue editor consists of several other tools which are used by the dialogue author in creating dialogue modules, including a dialogue language, a graphical formatter, a menu formatter, a voice I/O manager, a language implementer, a dialogue

design "expert," and a dialogue database system. Because the functions of these tools determine the complexion of AIDE, each of them will be discussed in some detail later in this paper.

It is important to note that these tools basically fall into two major categories: those that are used as dialogue design aids (e.g., the dialogue design "expert" and the dialogue simulator) and those that are used as dialogue implementation aids (e.g., subtools of the dialogue editor, including the dialogue language, the graphical formatter, the language implementer, etc.). The dialogue design aids help the dialogue author determine the content of the dialogue modules, while the dialogue implementation aids help write dialogue modules with that content.

As each of these tools is used to create the various components of dialogue modules, an interpretable representation of the contents and format of each dialogue module is created and stored in the dialogue module database. Then, when needed (e.g., at execution time or for modifications to the dialogue modules), this representation is retrieved and interpreted. At execution time, the dialogue author sees only the interpreted output of the dialogue modules, not the actual source code of these modules.

An overall framework is needed to provide integration and easy use of all the various tools. The software which must be developed in order to create such an interface is a sizable task. AIDE will therefore be implemented in stages, with new tools being designed for easy integration as their development is completed.

5.2.3. The Dialogue Author's Interface

The dialogue author communicates with these tools through a high-level interface that itself uses a human-factored dialogue to facilitate use of the tools by the author. This interface provides consistent interactions between the various tools, so that, for example, the dialogue author does not have to communicate in one way with the dialogue simulator and in a totally different manner with the graphical formatter. Additionally, the transitions in going from one tool to another should be smooth and immediate, despite the highly differing functions of the various tools and despite the fact that several tools may be used in the implementation of a single dialogue module (e.g., the dialogue language, the graphical formatter, and the language implementer). A well-integrated, consistent language for communicating with the tools and consistent error messages facilitates the dialogue author's use of the many facilities in AIDE.

5.2.4. The Coordinator

Just as a high-level interface is needed for the dialogue author to communicate with the tools, a similar framework is needed in order for the tools to communicate with the underlying host computer system and with each other. This structure, which synchronizes and controls both the tools and and the created dialogue modules, is called the coordinator. Such features as consistent interactions and natural transitions between tools, which are manifest in the dialogue author interface, are implemented and controlled through the coordinator. It is composed of two parts: 1) the synchronization and control mechanisms that allow the various tools to interact with

each other, as well as facilitate their communication with the system, and 2) the workspace in which the dialogue author creates, modifies, and tests dialogue modules. Thus, the coordinator is the structure which coalesces and supports the functions and the interactions of the entire computational portion of AIDE.

6. THE AUTOMATED TOOLS OF AIDE

6.1. DIALOGUE EDITOR

As early as the analysis phase of the system development process under DMS, data flow diagrams (DFDs) are created to provide the logical architecture of the application system, from the highest-level functions to the separation into dialogue and computational modules. Automated production of these DFDs is a powerful tool for the dialogue author in the development of the application system. (This will be discussed in more detail in a later section on the dialogue simulator.) The dialogue editor provides the facility for creating on-line DFDs at all levels of refinement. The dialogue editor also provides the automated tools for the dialogue author to use in implementing dialogue modules. Then, by displaying the DFDs, the dialogue editor aids the dialogue author in determining which dialogue modules of the application system are complete and which ones need more development by a

specific representation of the corresponding nodes in the DFDs. For example, completed modules might be displayed in blue and incomplete ones in red, or incomplete ones might be blinking. Thus, the dialogue editor is the manager of dialogue development, beginning with automated production of DFDs and continuing through until all dialogue modules are completed.

As the dialogue author begins to write dialogue modules for a specific application system, that person employs several AIDE tools in the creation of a single dialogue module. For example, a dialogue module containing both textual and graphical components might require the use of the dialogue language or the text formatter (for the text), the graphical formatter (for the graphical content), and the language implementer (for parsing inputs from the user). Finally, once the dialogue module is completed, it must be stored in the dialogue module database. If each of these tools requires the dialogue author to interact with it in a different way than with the other tools, the resulting interface will be confusing and frustrating for the dialogue author. The dialogue author should, when creating or modifying a dialogue module, be unaware that many tools are being used to implement various features in a dialogue module. Thus, the dialogue editor provides a high-level, uniform interface between the dialogue author and each of the tools associated with creating and modifying dialogue modules. Those implementation tools which the dialogue editor unifies are the dialogue language, the graphical formatter, the menu formatter, the voice I/O manager, the language implementer, the dialogue design "expert", and the dialogue database system.

The dialogue editor thus provides the means for managing development of dialogues for an application system, from the earliest stages of functional

specification in the DFDs to implementation of the actual dialogue modules themselves. It helps the dialogue author keep track of the state of development of each dialogue module, it draws attention to those modules that may be incomplete and need more development, and it helps with planning and scheduling of all dialogue development activities.

6.2. DIALOGUE LANGUAGE

6.2.1. Programming vs. Construction of Dialogue Modules

One of the goals of our research is to relieve the dialogue author of the necessity to "program" dialogue modules, since this is counter to the basic principles of DMS. As an initial temporary situation, dialogue modules are written in a subset of a programming language. As the tools of AIDE are implemented, creating dialogue modules will become less and less like programming for the author. In an advanced version of AIDE, a programming-like dialogue language will no longer be needed at all by an author. Instead, the dialogue author will use the various tools of AIDE to construct dialogue modules without having to write code.

Dialogues, however, will always be manifest in the system as some form of software, either explicitly (written directly in a dialogue language by a dialogue author) or implicitly (generated by tools in a later version of AIDE), to interact with computational code. There are several reasons for beginning with a programming-like language for the writing of dialogue modules. It would be very difficult to create a high-level AIDE that could

produce its own dialogue modules (in some underlying language) without having ever considered what features are needed and what problems are inherent in such a language. Also, since AIDE will evolve in progressive steps, not all tools will be available immediately. Therefore, an interim means is needed to write the parts of dialogue modules for which tools do not exist. A dialogue language serves this purpose. Because this makes the author's job more like programming, one goal is to keep this interim dialogue language as simple as possible, since the dialogue author is not intended to be a sophisticated programmer. In order to determine both the advantages and disadvantages of existing high-level languages under consideration for use as a dialogue language, dialogue modules have been written in a subset of Pascal, as well as in a subset of Fortran.

6.2.2. Interim Dialogue Languages

Use of Pascal as a dialogue language reveals that it contains most of the basic constructs which are needed to write dialogues. Another reason for writing dialogues in Pascal is that, under another task of this research project, a Pascal interpreter is being built. When completed, this interpreter can be used so that the dialogue modules can be interpreted (rather than compiled) and are therefore immediately executable and testable. In addition, Pascal is readily available on most systems. The use of a CASE statement (such as is available in Pascal) is useful for recognition of a menu selection input by a user. Pascal has some disadvantages for use as a dialogue language. Its I/O is notoriously cumbersome, particularly the need for padding varying length strings with blanks up to the declared length of

the string. Also, DMS interprocess communication is accomplished through the use of several high-level procedures. Each of these procedures, if it is used in a dialogue module, must be declared at the beginning of that module if it is written in Pascal. This could be quite tedious for a dialogue author to handle. Pascal user-defined types are a very powerful construct for a good application programmer, but it is unlikely that a dialogue author will be knowledgeable enough about programming to take advantage of them.

Use of Fortran as a dialogue language has several advantages over Pascal. If Fortran unformatted I/O is used, the problem of padding blanks is avoided. Additionally, it is not necessary to declare all called procedures at the beginning of the dialogue module. While Fortran does not have a CASE statement, the same function can be accomplished by use of multiple IF-THEN statements. Neither Pascal nor Fortran is a clear-cut choice for a dialogue language. As just discussed, both have unique advantages and disadvantages. A choice between them can be made only after more dialogue modules are written using both languages. As previously emphasized, these concerns are temporary, and will not be an issue when the dialogue language is no longer needed.

Some consideration has been given to the development of a customized dialogue language. Language design and implementation, however, is quite a large task. During the summer of 1981, a very small version of such a "node language" was attempted, with mixed results. The scope of the task of language development was evident when, after three months of full-time effort by one good programmer, the only language constructs that had been implemented were those such as IF-THEN-ELSE and READ/WRITE (I/O) statements. Several absolutely necessary features were not included, such as WHILE...DO

(iteration on condition), nesting of statements, and simple arithmetic operations (e.g., incrementing and decrementing of variables). Still other problems were encountered with parsing the language. In addition, the problems with complicated, parametric coding could not be totally avoided, and it would have been necessary to provide extensions to a customized language similar to those necessary for either Pascal or Fortran. The final conclusion was that, rather than directing efforts toward creating yet another language (not the intent of our research), and particularly in light of the temporary nature of the dialogue language, a subset of an existing high-level language should be chosen and the appropriate enhancements implemented to make the difficult constructs as transparent as possible to a dialogue author.

6.2.3. DMS Services in the Dialogue Language

Some of those aspects of writing dialogues in either Fortran or Pascal that are potentially difficult for a dialogue author include, for example, terminal escape sequencing for screen control, since it is highly parameterized and cryptic. File handling, too, can be complicated, since several lines of rather terse code are necessary in order to declare, open, reset, read, write, and close files. Similarly, interprocess communication under DMS, while possibly straightforward to a programmer, involves some fairly sophisticated communication services that would not readily be understood or usable by a dialogue author. Data checking may be difficult for a dialogue author who has little knowledge of data typing requirements and conventions. Use of various I/O devices, especially voice and touch panels, requires spe-

cial terminal handling. Metering of user performance, particularly important to the human factors experimenters working with us, is also necessary.

These kinds of implementation requirements for writing dialogue modules must be made as simple as possible for the dialogue author as long as dialogue modules must be created using the dialogue language. Therefore, numerous high-level constructs are being implemented to alleviate these sorts of technical, language-dependent problems as much as possible. These constructs, or DMS services, are callable procedures that an author can use as necessary. An attempt will be made to identify as many of these potentially confusing situations as possible, and a DMS service created for each. For example, escape sequencing involves cursor control, carriage return, line feed, page feed, screen clear, screen splitting, etc. Screen clear on the VT100 terminal, for example, in Pascal is:

```
WRITE (CHR(%x1B),'[2J');
```

A service handles this in such a way that the author simply needs to know that the screen should be cleared at a particular point in a dialogue and that a NEW_FRAME procedure accomplishes this, for example:

```
call NEW_FRAME ('tt', 'top')
```

An alternative approach for a screen-handling service might be a call to a subprocedure SCREEN, which would then present menus to guide the author in dealing with screen clear, cursor control, etc. In fact, both methods might be implemented; the menus would likely be used more by a novice dialogue author, and the first method would be used as the author gained experience in writing dialogues.

Similarly, the DMS interprocess communication mechanism, necessary to execute dialogue modules and computational modules in different processes under DMS, could be simplified by providing a basic construct for communication which would cause the system first to determine whether the executing environment for the system under development is single- or multi-process and then to complete the dialogue module appropriately.

6.2.4. Data Validation

A major issue in writing dialogue modules is that of user input validation. This validation is actually a computational function, since it involves searching a database, checking a data type, etc. Despite this, at this point in our research, our philosophy is that the dialogue author, and not the application programmer, should be responsible for checking whether user input is valid. Our reasons for this are that all user input is a part of the external dialogue and, as such, should be dealt with by the dialogue author, not the application programmer. Requiring the application programmer to do validity checking would require passing the user's input to the computational component, doing the validation, and then continuing on within the computational program if the input is valid. However, if the user's input does not prove to be valid, the application programmer must request a dialogue module to transmit this information to the user, request another user input, and begin the whole validation procedure all over again. Obviously, if several inputs for the same query are erroneous, the calls between computation and dialogue become numerous. Thus, we propose to automate user input validity checking as much as possible, so the dialogue author can

easily do it within the dialogue modules. This should avoid the increase in communication that would occur between the dialogue author and the application programmer, as well as between the dialogue and computational components of the software, if the application programmer were responsible for input validity checking. It also provides a "clean" interface for the application programmer in that it insures that whatever is passed to the computational component has been validated and is therefore ready to use. This allows the application programmer to treat the dialogue as a function that returns a valid input, so that person does not have to interrupt development of the computational component to deal with validation.

Automation of data type checking can be implemented so that when the system receives user input, the dialogue author makes a call to a VALIDATE function, passing the just-received input, a predefined error type for which this particular input is to be checked, any parameters associated with this error type, and a return code to indicate whether the validation was successful. For example, a simple validation sequence in a dialogue module (using DMS services of OUTPUT and INPUT) might be:

```
call OUTPUT (..., 'Please enter your name', ...);
call INPUT (..., name, ...)
call VALIDATE (name, error-type, parameters..., return-code);
```

Upon receiving the user's input, a call to the VALIDATE procedure would determine whether the input is legal. In order to implement such a function, a classification of possible error types will be needed. Possible types of checks for an input include checking for existence in a database, existence in a data structure (e.g., table look-up), existence in a menu listing, and correct data type, length, and range (i.e., conformity to a specific data

declaration). A complete taxonomy of error types should facilitate implementation of an automated input data validation service.

6.2.5. Summary

The above examples are just a few of the types of services that might be implemented in DMS using high-level constructs. As the number of these services increases, the dialogue author will need to use the low-level constructs of the dialogue language less and less to write dialogue modules. These services therefore relieve the author of the most tedious aspects of using the dialogue language. Ultimately, direct use of the dialogue language by the dialogue author will evolve into the use of a very high-level set of tools in AIDE that will generate the "code" for dialogue modules, and the underlying language will become transparent to the author. Thus, the capabilities of the underlying language, and not the syntax of the language itself, are a much more important consideration.

6.3. GRAPHICAL FORMATTER

With the tremendous increase in interactive graphics, human-computer dialogue now involves much more than simply words displayed on a CRT. Sophisticated graphical tools are being used to greatly improve the usability and understandability of the human-computer interface [FEINS82]. Such images as graphs, pictures, and geometric shapes are fast becoming a regular part of screen displays in an attempt to make them more human-factored and

more comprehensible. Often a graph is more quickly understood than the corresponding columns of figures. A graphical formatter, as one of the tools that is available to a dialogue author under DMS, provides all the standard graphical editor features: color, simple shapes (lines, curves, etc.), geometric shapes (squares, circles, rectangles, etc.) as well as convenient ways of manipulating and modifying these features.

Displaying a circle on the CRT screen can be used as a simple example of how a graphical formatter might facilitate the creation of the graphical component of a dialogue module by the dialogue author. In a conventional graphics environment, drawing a circle might be done by "programming" it, e.g., a call to a CIRCLE subroutine:

CALL CIRCLE (xcenter, ycenter, radius, line-thickness, color).

This may seem simple enough to a programmer, but determining the parameters may be confusing to a dialogue author who is a naive computer user. Additionally, any change to the size, color, or location of the circle involves finding and altering the appropriate line of code. That is, the author must actually know a graphical programming language and is forced to write and debug in it. In contrast, the dialogue author, through the graphical formatter, might request a "shapes" menu, which would display a number of geometric forms: square, circle, triangle, rectangle, etc. Once the author chooses the desired shape (in this case, a circle), appropriate and easily understandable questions relating to the dimensions and display requirements of that shape might be presented. In the case of a circle, these would be queries about radius, line thickness, and color. Initial location might be determined by touching the desired screen position. Modification of loca-

tion and radius might be accomplished with cursor positioning function keys. Changes to an existing circle would be effected in a similar, "interactive query" manner. For example, the dialogue author might choose from a menu to move an existing graphical item, such as the circle, to another location specified by touching the display panel.

6.4. MENU FORMATTER

Structures, such as menus, that can be easily standardized with reasonable human factors principles can be created through the use of an automated formatter. A menu formatter is being developed so that a dialogue author can interactively create, modify, and store menus for use in dialogue modules. The format for a menu can be specified so that the dialogue author can be led through its implementation by the use of appropriate prompts. The menu formatter is being designed so that the author is queried for all inputs that are necessary to create a menu in a pre-specified standard format. This includes queries for the title and purpose of the menu, the menu options and their selection codes, and the query that the user should answer by the choice of a selection code. The completed menu is displayed to the dialogue author, who is given a chance to confirm its content, or to make modifications as desired. When the menu is completed to the author's satisfaction, an interpretable representation of the menu is stored until needed by a dialogue module at execution time, or until the author retrieves it to make further modifications. Such a menu formatter enforces creation of a fairly invariant format for all menus that are designed using it. In an

advanced version of the menu formatter, the dialogue author can change the prespecified menu format. However, one of the major principles of human-factored display design is consistent formatting throughout an entire application system. Use of the menu formatter encourages this consistency, as well as simply making the dialogue author's job much easier with the use of an automated tool for menu implementation.

6.5. VOICE I/O MANAGER

Recent technological advances in voice recognition and voice synthesis have made the use of speech a viable option in human-computer dialogues. Just as the incorporation of graphics is becoming increasingly popular, the use of voice as both an input medium and an output medium is giving a new dimension to human-computer dialogues. Associated with this new aspect of the human-computer interface are several issues that directly affect the dialogue author as that person incorporates voice I/O into the dialogues. Foremost among these are special terminal handling requirements necessary to coordinate the simultaneous use of more than one type of I/O device in a single interface. Otherwise, input from and output to each device may be interleaved with that of other I/O devices and become meaningless. For example, if voice input to a system is done at the same time as keyboard input, the receipt of this input from each device must be recognized and handled separately. A tool to synchronize multiple I/O devices relieves the dialogue author from dealing with somewhat complicated terminal handling and synchronization problems.

Additionally, two issues directly affect use of the synthesis and recognition devices themselves. The first one involves creating or synthesizing the words that comprise the vocabulary of the voice synthesizer that is used for output from the computer. On most current systems, vocabulary words must be created using the available phonemes. This process can be fairly time-consuming and tedious, and distracts the dialogue author from creating dialogues. Possible use of a "voice dictionary," containing frequently used words as well as words pertinent to a specific application system, makes such words readily available for the author to use in dialogues. An automated tool to aid in synthesizing other words would also be very useful to the author.

The second issue involves training the voice recognizer to recognize each user's voice. This device, used for human input to the computer, must be trained for every person that uses it. Retraining or refinement may even be necessary for an individual from time to time. This raises a concern for determining the validity and correctness of user inputs. As long as an input is not recognizd at all, there is no problem. Inclusion of redundant coding, e.g., bells, is frequently used to give audible feedback to indicate that an input error has occurred. However, when an input is recognized incorrectly (e.g. an input of "five" is recognized as "nine") and an action is taken, unexpected problems can arise. If, for example, in an emergency situation the user needed to convey to the computer the minutes remaining until system shut-down, the recognition of "five" as "nine" would cause the computer to execute emergency precautions as if there were nine minutes remaining, not just five. The dialogue author may need a special kind of validity checking to insure, as much as possible, that voice inputs that are

incorrectly recognized do not have undesired or disasterous results on system performance.

Automation of as many aspects of voice input/output as possible encourages the dialogue author to easily incorporate this state-of-the-art component into dialogues.

6.6. LANGUAGE IMPLEMENTER

6.6.1. Interactive Command Languages

All dialogue or interaction which occurs between a human and a computer necessarily involves some kind of language which the human uses to communicate with the system. Such a language can be textual, graphical, or, with the current trend in computer technology, even vocal in nature. At this stage in the development of AIDE, we are most interested in textual language, which is often a keyword command language. The design of a command language can have a direct influence on the ease of use of the system by the human. If the command language is difficult and unnatural for a human to understand, then a user attempting to communicate with a system through such a language has very low efficiency, and will probably become frustrated and even give up. Conversely, a natural, easy-to-use command language encourages a user and improves performance. There are various forms of command language components, including keyword, menu, voice, function keys, graphics, and parameterization. A dialogue author, when writing a dialogue module, may find it necessary to include more than one of these components.

Implementation and recognition issues for each of these command language modes can be very different. A language implementer, as one of the tools that is a part of AIDE, will assist the dialogue author in the design, representation, and recognition of command languages. The language implementer consists of two main components: 1) a tool to aid in creating a command language and its representation for a specific application system, and 2) a tool to generate a parser to recognize this command language, once it is incorporated into the application system. That is, the language implementer is used to create a command language and its recognizer.

Additionally, a language implementer alleviates the difficult problem of determining whether the dialogue author or the application programmer should do input parsing. Input from the user to the system is really part of the external dialogue, and as such, should be handled by the dialogue author. But an author probably does not have the skills necessary to create parsers, while the application programmer should not be dealing with something so directly related to external dialogue. The language implementer is an automated tool in AIDE that the dialogue author can easily use to do input recognition when needed.

6.6.2. Language Representation

In order to design a command language, some way of representing or specifying it symbolically must be available. Several issues are prevalent in the design and representation of command languages. Foremost among these is the mode of presentation for a particular language and therefore for a dialogue that may utilize that language. Specifically, complete keyword, com-

mand completion, function keys, prefix decoding, and spelling correction are all ways in which command language keywords can be input. Although the language itself and its mode of presentation are technically separate issues, the mode of presentation must be considered when the language is used in a dialogue. Once the method of input for a particular command (or command language) has been determined, representation of that method ought to be part of the command language specification. For example, with command completion, the user-input portion of the command and the system-completed portion should be somehow differentiated in the formal representation. Likewise, upper and lower case letters must be handled, depending on their context. Many formats also exist for numeric values.

Specifying delimiters between keywords or other components of a command is another difficult problem. What is a delimiter in one input string may be a valid input character in a different situation (e.g., use of a "/" or a single space or a "," can have various different meanings depending on context). Problems are immediately encountered in attempting to create a notation for a formal, easy-to-understand, yet complete representation of a command language syntax. These include representation of required versus optional elements of the command keyword, notation for a choice of alternatives, notation for conditionally optional components, including the conditions themselves, constraints of alternatives caused by a priori choice of, for example, a delimiter, and limitations on valid characters in a string. Other issues include choice of the command keyword itself and any device dependencies or device representations that might be needed.

Several creative ways to approach this representation problem interactively are possible; that is, using color and graphics on a CRT screen.

Such ideas include color matching (e.g., for components that must be consistent, or red for all required components), blinking (e.g., for optional components), and highlighting (e.g., for those components not yet input). Additionally, adaptive specifications that change dynamically as the command components are input are possible. All of these issues are discussed in detail in "Interactive Language Representation and Recognition under DMS and AIDE" [JOHD82b].

What is needed for language specification is, of course, a metalanguage, which is a language to describe the syntax of other languages. Many such languages already exist and are familiar to computer science people: regular expressions, "counter" grammars, Vienna Definition Language, and Backus-Naur Form (BNF), for example. Recent new developments include multi-party grammars [SHNEB82], which make an attempt to distinguish between the human and computer portions of a language, while the human's cognitive processes are being included as a part of the language definition by other researchers [REISP82]. BNF is possibly the most widely used of these, but it has some specific drawbacks which make it somewhat undesirable for our purposes. For example, it is not human-factored and is not easily understood by a non-computer science person.

In order to determine what is needed to describe command languages as generally as possible for the language implementer, our initial step is to collect a large number of specified grammars that are as varying as possible. The list thus far consists of the grammars for the SAM editor, GENIE experimenter and GENIE subject commands, a "node language" used for an airline reservation system, a Mini-Database System being developed as a computer science Master's project for use in the graduate Computer Science data-

base courses, and Shneiderman's multi-party grammar. An attempt is being made to classify and taxonomize the various characteristics of these grammars. Generally, specific deviations from a regular grammar which appear to be necessary or useful in command languages will be identified, and these deviations incorporated into the metalanguage being designed. As with all other tools being created as part of AIDE, it is extremely important that the language implementer be human-factored and easily usable by the dialogue author.

6.6.3. Summary

Basically, then, a language implementer is a tool to aid a dialogue author, who has little formal knowledge of language design, in developing, implementing, and parsing command languages for application systems. Without such a tool, a separate language and parser would have to be written for every application system designed under DMS. The undesirability of this is obvious.

Research on the design and implementation of this tool has just begun, and many major issues are still unresolved. A taxonomy of languages and their associated language representations (hopefully in a human-readable, human-factored form) must be determined, and a metalanguage developed for describing these languages. Then, automation of a language implementer both to create and to recognize these various types of languages can be addressed. When completed, the language implementer will be used by the dialogue author in conjunction with the other tools in AIDE, particularly the dialogue language, the menu formatter, and the graphical formatter, to construct dialogues.

6.7. DIALOGUE DATABASE

The dialogue database serves as a storage facility for dialogue modules as they are developed and tested, and from which they can be retrieved as needed by the execution environment of the application system of which they are a part. The dialogue author may want to manipulate single dialogue modules, as well as groups of modules, for: 1) implementation, both at the time the module is originally created and when the dialogue module needs modification, and 2) testing and simulation, after the dialogue module is implemented. Thus, while the dialogue author interacts with the dialogue database through the dialogue editor, the database is shared by the dialogue simulator as well.

There are numerous ways that a dialogue author may want to retrieve dialogue modules once they are stored in the database. These include the following:

-- By specific dialogue content (both textual and graphical)

-- By the name of the dialogue module itself

-- By the format and style of the module

-- By I/O devices used

-- By the names of parameters that are passed between modules (either dialogue or computational)

-- By calling relationships to other dialogue modules (modules called, or modules called by)

-- By calling relationships to computational modules

-- By whether metering of user performance is used

-- By a functional description of what the module does

-- By parts of the formal specification of internal dialogue of the dialogue module

-- By immediate access from the execution environment (i.e., the author may want to stop the execution of the application system and retrieve the currently executing dialogue module)

-- By immediate access from the dialogue simulator (i.e., the author may want to halt execution of the dialogue simulator and retrieve the currently executing dialogue module)

-- By all other potentially affected instances of a change (i.e., a change to one dialogue module may necessitate a similar change to other dialogue modules, for example, to maintain consistency)

This multitude of ways to search a database implies the need for a hybrid data model that consists of both a traditional relational database as well as a dialogue module library file which contains the actual source code for each dialogue module. Each dialogue module is represented by one or more records in the relational database. As shown in figure 6, this record contains the fixed attributes for that module, as well as a pointer to the location in the dialogue module library that contains the source code and contents of the dialogue module. Most of the searching mechanisms listed above can be implemented by representation of the dialogue module with fixed relational attributes in a more or less conventional relational database, using the standard relational database retrieval techniques based on attribute-value matching. Retrieval by dialogue module content and by functional description requires text searching of the contents of the individual dialogue modules themselves. This text searching can make use of inverted files as indexes and will require some sophisticated searching methods such as closest match and partial match techniques in order to retrieve appropriate modules [SALTG80].

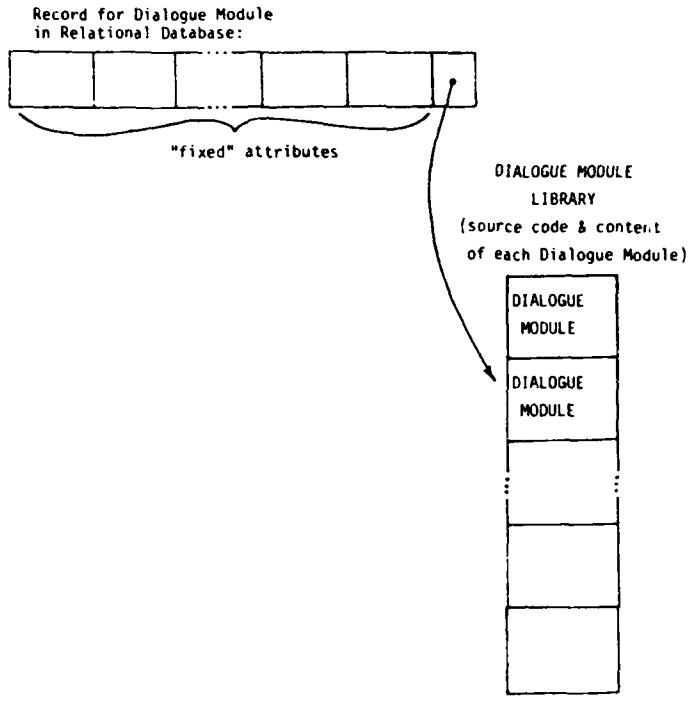


Figure 6. Hybrid Data Structure for Dialogue Database

Because of the lack of specificity that may be associated with a search request, the relational part of the request may indicate a large number of modules to be searched in the text matching part of the request. Since textual searching can be time-consuming and therefore costly, such searching can be limited in various ways. For example, when searching for specific text that is output to the application user, it would be necessary to search only statements that directly do output (e.g., output statements in the dialogue language). This kind of optimization can be built into an intelligent text searching algorithm.

6.8. DIALOGUE DESIGN "EXPERT"

There are literally hundreds of guidelines and principles that have been developed for the design of effective interactive dialogues. Some of these have been empirically studied and refined; many are merely ad hoc, intuitive ideas. A compendium of about five hundred of these principles has been compiled in "User Considerations in Computer-Based Information Systems" by Williges and Williges [WILLB81]. That document is not intended specifically to be a guide for interactive system design, but rather a collection of principles to be used in designing behavioral research in this field. Similarly, S. Smith, in "Man-machine interface (MMI) requirements definition and design user considerations: a progress report" [SMITS81], has provided an extensive checklist to be used for designing dialogues based on type of system, user needs, hardware, etc. The sheer bulk of such collections of guidelines points out the difficulty of creating reasonable, consistent dialogues.

One of the aspects of current computer systems that can be most confusing to a user is the inconsistency of interactions both within and between systems. For example, a menu interaction has virtually infinite possibilities for implementation. It can begin in the leftmost column or be centered on the screen. The menu items can be upper and/or lower case letters. The user's choice can be selected by keying in a number or letter, or through touch panel or light pen selection. The query for which the menu is intended can be displayed at the top or at the bottom of the screen.

A simple example that easily demonstrates the point here is the choice of 'yes' or 'no' as the user's response to a query from the system. Although this is one of the simplest of all types of query/response interac-

tions, the options for an appropriate input are surprisingly numerous: Y, y, Yes, YES, yes, N, n, No, NO, no, etc. Dialogues often will accept only the entire word 'yes' or 'no' as a correct response (if it is a very flexible system, it may accept both upper and lower case letters!). Imagine a user of an interactive system who has been contentedly responding to each 'yes/no' query from the system with a 'y' or an 'n'. But suddenly, in a random dialogue, an input of 'y' elicits a response message such as:

```
Invalid input string.
?????
```

The poor user is totally dismayed now, simply because whoever wrote that particular dialogue failed to make 'y' a valid response, even though it had been one for all other dialogues.

Thus, failure of the dialogue author to conform to consistent formats, contents, syntax, responses, etc. can be confusing and difficult for a user. AIDE helps to reduce such inconsistencies by providing an automated tool which guides the dialogue author in creating standardized dialogues throughout an entire system. Such a tool, probably a production rule-based system, contains current knowledge about dialogue design principles. When the dialogue author needs to know whether the specific dialogue being created would be more effective, for example, as a menu than as a keyword, information in the "expert" can be accessed to guide that choice. Thus, this differs from the other tools in that it is actually a dialogue design decision aid, rather than a dialogue implementation aid.

6.9. OTHER DIALOGUE EDITOR SUBTOOLS

As the high-level tools of AIDE are implemented, the dialogue author will need to use the dialogue language less and less. Ultimately, the dialogue language will not even be used for creating the textual part of a dialogue module. This is accomplished by the use of a text formatter. Such a tool allows the author to easily and rapidly change both the textual content and format of a dialogue module. Formatters for the creation of structured dialogue components can also be automated and incorporated into AIDE. These include a formatter for creating fill-in-the-blanks-type forms, and a formatter to aid in creating on-line CAI/help facilities for application systems developed using DMS. Other tools are being created to assist the dialogue author in the use of alternative I/O devices in dialogues, such as touch panels, joy sticks, etc.

6.10. DIALOGUE SIMULATOR

6.10.1. Simulation of Dialogues: Data Flow Diagrams to Dialogue Modules

The ability to observe the logical flow, as well as the form and content, of the dialogue of an application system is needed at the earliest development phases of analysis and design. This need continues all the way through the development process until the application system is implemented and operational, as well as later, during modification cycles. Recent similar developments in this research area have produced an Interactive Dialogue

Synthesizer (IDS), consisting of a display builder and a dialogue controller. The dialogue specification language in which system designs of the IDS are documented serves as input to the simulation mechanism, which presents the appearance of the system for various interactive devices [HANAP80].

The methodology for development of an application system under DMS includes the use of hierarchical data flow diagrams (DFDs) to represent the flow of data between modules in the system [YUNTT82]. At the highest level, DFDs are basically a representation of the system functions, without specific attention to the separation between dialogue and computational components. These DFDs are then refined into successively more detailed, specific DFDs until they contain explicit representations of dialogue modules and computational modules and the data flow between them. Thus, DFDs, produced during the analysis phase of system development, are a representation of both dialogue and computational components, and the data relationships between them, early in system development. Additionally, they provide an early representation of both dialogue sequencing and dialogue content.

A part of the dialogue simulator is a tool to allow automated presentation of DFDs at all levels. Such a tool allows the dialogue author to see the logical sequencing of an application system as soon as sequences of DFDs are created. The dialogue simulator provides an execution trace through the nodes of the DFDs, allowing for branching, iteration, etc., as necessary.

For the higher-level DFDs, this trace represents only the logical flow of the system dialogue, with at most general descriptive information about dialogue content or format. But this early ability to visualize the logical sequencing of the system is extremely important if the system is to be

human-factored. The earlier that system sequences can be simulated, the easier it is to identify and make changes to them as needed to improve the resultant user interface. Providing, as a part of the dialogue simulator, a DFD trace facility gives information about the system sequencing long before dialogue and computational modules are separately defined, much less implemented.

As system analysis and design continues, the high-level DFDs are refined into successively more detailed diagrams. Through this refinement, DFDs become UDCs (User-Dialogue-Computation Diagrams), which are DFDs that specifically indicate which nodes are dialogue, which are computation, and which are both. Those nodes that are both are eventually further decomposed into UDC nodes that are either pure dialogue or pure computation. Once the system is refined to the point where separate dialogue and computation functions are represented in the UDCs, actual implementation of both types of modules can begin.

Once the dialogue modules are written, the best way to test them is by executing them. Normally, this requires that all the computational modules be completed and that all the dialogue modules and the computational modules be integrated and debugged. Thus, the entire system, or at least some clearly delineated, functionally complete portion of it must be implemented.

Clearly, some method of executing dialogue modules without the need for completed computational modules would be of tremendous use to a dialogue author. Therefore, another aspect of the dialogue simulator is an environment which supplies a facsimile of the internal dialogue and global execution environment needed by a dialogue module, so that it can be executed and tested in isolation both from the computational components and from other

dialogue modules. Such a system allows testing and debugging of the dialogue modules in their earliest stages of implementation. Then, once they are syntactically and semantically correct, preliminary testing and evaluation of such aspects as sequencing (both high-level and intermodule sequencing), display formats, and specific contents can be done. The dialogue author can, in effect, simulate the system as if it were completed. That person can analyze the overall presentation, specifically looking for inconsistencies, illogical or confusing sequences of events, even misspelling or other typographical errors that are difficult to spot when embedded in the dialogue language code. Appropriate modifications and revisions can then be made, both at the local, single module level (e.g. a slight change in the wording of a display) or at the global level (e.g. a change to the sequencing of two or more modules). Obviously, the local changes involve only the dialogue author, whereas global changes may involve the application programmer as well, if such changes affect the computational code, too.

The dialogue modules cannot all be implemented at once, but are constructed over a period of time. Thus, it is necessary to maintain the ability to "simulate" the system's logical flow, even though during the implementation phase some nodes of the UDCs are in their original UDC descriptive form and some are in executable source code. These two types of nodes of the UDC must be compatible in the dialogue simulator, so that the logical sequence trace is still available even as implementation of individual dialogue modules occurs. During this phase, the dialogue simulator provides a sequence of display events in varying levels of detail and depth, depending on whether a node of the UDC has been implemented into a dialogue module or not.

6.10.2. Implementation Issues

Implementation of the automated presentation of DFDs might be through the use of two CRTs. (See figure 7.) One presents a portion (a window) of a DFD, its nodes labelled with system functions and its edges labelled with

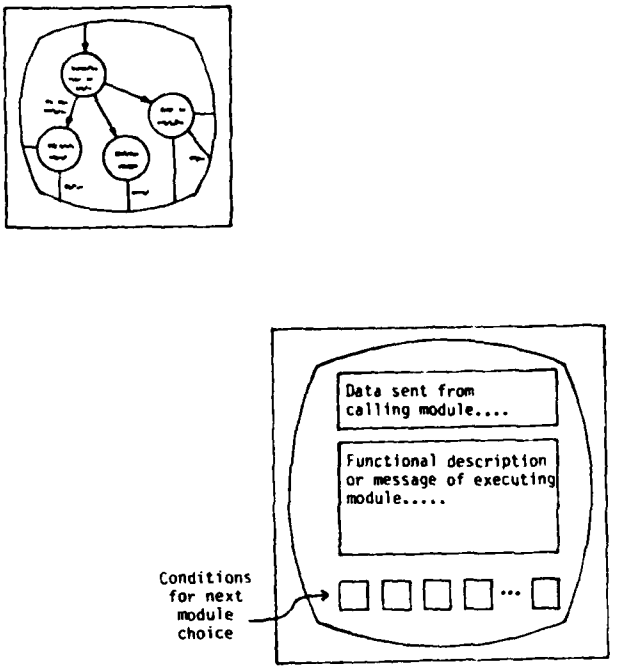


Figure 7. Dialogue Simulator DFD Trace Facility

data that will be passed. The currently executing node of the trace might be blinking in the center of the screen, with other connected nodes surrounding it on the screen. Simultaneously, another CRT screen shows the sequence trace as the DFD nodes are "executed." The screen displays, for the currently executing node, the data sent from the calling node, a functional description of the executing node, and all possible choices or

branches that are possible successor nodes to the current one. Upon selection of one of these choices (through a menu or touch panel, for example), the simulation proceeds next to the chosen node.

A key issue in the implementation of such a dialogue simulator is how to generate the values that are necessary for execution of dialogue modules when the computational code which would normally supply these data is not operational. The dialogue simulator execution environment must provide the dialogue modules with all the data needed to support the simulation of a module or a group of modules in isolation from the rest of the application system. To do so, it is necessary to completely and formally define the total execution environment, including all calls to each module, all parameters passed to each module, and all global variables and data structures known to each module. A solution typical to simulation situations is to use stubs, precoded tables, random number generators, etc. to provide the necessary values. Another alternative might be to use an extension of the concept of formal data type declarations to provide sample values. Sample data values could be included in the declaration of a variable, for example:

```
var name: char
ex (Smith, Jones);
```

6.10.3. Summary

The advantages of a dialogue simulator to allow sequence traces of DFDs/UDCs at an early stage of design and execution of dialogue modules without requiring computational code during the implementation phase are obvious. Of all the tools that will be implemented, at least initially, this one has the most potential for immediate and very visible impact on the

human factors people with whom we are working. In addition to providing early visualization of the logical sequence of the system, it allows them to pretest the dialogues for their interactive experiments and make the appropriate modifications, rather than having to wait until the system is operational to discover that some aspect of it is not satisfactory for their needs.

6.11. CAI/HELP FACILITY

Finally, as the tools are implemented and AIDE becomes a viable, workable system, a dialogue author must learn how to use the dialogue author interface to interact with AIDE. Some sort of CAI tutorial will provide initial training to dialogue authors learning to use AIDE to create dialogues. Additionally, an on-line help facility can be accessed by the dialogue author for assistance when a reminder is needed about how to use a specific tool to perform a particular task. A well designed tutorial and help facility will be a great influence on how well a dialogue author likes and uses AIDE.

7. DESIGNING A DIALOGUE UNDER DMS

Creating an application system, including both dialogue and computational components, under DMS is different from the development of a traditional software system. This is due mainly to the separation of dialogue and computational modules through the concept of dialogue independence and because of the introduction of the dialogue author to the system development team. Thus, it is important that the utilization of the methodology, briefly presented earlier in this report, be viewed in the context of DMS. Only when this development procedure is understood by both the dialogue author and the application programmer, in their respective roles, can DMS be effectively used to create complete application systems.

To aid in this understanding, the following is a very simplified version of the overall application system development process, presented from the dialogue author's viewpoint. Of particular concern is the communication that must occur between the dialogue author and the application programmer in order to establish compatible interfaces between the dialogue and computational modules so that the end result is a fully integrated, operational application system.

7.1. APPLICATION SYSTEM DEVELOPMENT PROCEDURE

Phase 1 - Requirements Specification:

The application expert and/or systems analyst creates a comprehensive list of system functions and defines the purpose of each of these functions in some detail.

Phase 2 - Analysis:

The dialogue author and application programmer (possibly in conjunction with applications expert and system analyst) define objects, operations on those objects, and sequences of those operations to accomplish the functions specified in Phase 1. Operational Sequence Diagrams (OSDs) and/or Data Flow Diagrams (DFDs) are produced at this point. Decomposition of the DFDs into UDCs (User-Dialogue-Computation Diagrams) shows the separation of dialogue and computational functions. These diagrams show the basic flow of the logical structure of the system. One particular difficulty at this stage is determining all possible paths through the logical sequences of a system. The DFD trace facility of the dialogue simulator, discussed earlier, is especially helpful at this stage to aid both the dialogue author and the application programmer in conceptualizing and actually visualizing the logical flow of the system very early in the development cycle.

Phase 3 - Design:

The dialogue author and application programmer, using the logical structure specified in the DFDs/UDCs/OSDs from Phase 2, determine separation of dialogue modules and computational modules, respectively. Individual module specifications and intermodule communication are also determined. Structure charts are produced to show the physical system architecture, including how all modules are related by calls to each other. For example, the dialogue author may experiment with some dialogue scenarios (transcripts), dividing them into dialogue modules. The programmer does the same for computational modules, determining interface specifications that are compatible with

the dialogue modules. The dialogue simulator can be used to see lower-level logical sequences represented by the UDCs.

Phase 4 - Implementation:

The dialogue author uses tools of the Author's Interactive Dialogue design Environment (AIDE) to create the content of executable dialogue modules by creating text (using the dialogue language or the text formatter), incorporating graphics (using the graphical formatter), incorporating voice (using the voice I/O manager), and generating parsers (using the language implementer). The dialogue editor helps the dialogue author manage dialogue modules at various stages of development. The dialogue simulator can again be used to display logical sequencing, as well as specific display content of implemented dialogue modules. The application programmer creates computational components at the same time, using the programmer's interactive program design facility in DMS.

Phase 5 - Usage and Testing/Modification:

The dialogue author uses the dialogue simulator to help visualize the dialogue, to aid in debugging dialogue modules, and to determine what early modifications might be needed in the system. This can be done both for a single module as well as for a "real-life" sequence of dialogue modules. Finally, as both dialogue modules and computational modules become completed, they are integrated together into the completed application system, ready for testing and use by the end users.

7.2. IMPLEMENTING A SAMPLE DIALOGUE

In order to understand what it is like to be a dialogue author, creating dialogue modules using AIDE under DMS, it is necessary to go through all phases of the development procedure for a specific dialogue. Given below is a small portion of dialogue between the computer and the user of a hypothetical interactive airline reservation system. Following this transcript, the activities necessary to actually implement a part of this dialogue are discussed for each phase in the system development procedure. In the dialogue transcript, "H:" represents the human's portion of the dialogue and "C:" represents the computer's part. The content of this transcript is not meant to emphasize human factors in an airline reservation system. Rather, it is given specifically to be used as an example by which the system development process can be explained.

7.2.1. Sample Transcript for an Airline Reservation System

C: Please type in your agent id.

H: Debbie

C: Sorry, that id is not valid. Please try again.

H: Debby

C: Welcome to the airline reservation system.
Which of the following would you like to do?

F: Obtain flight information for an itinerary
R: Make a reservation
P: Make a payment
T: Issue a ticket
M: Modify an existing reservation
C: Cancel an existing reservation
I: Obtain customer information for an existing reservation
A: Alter interaction mode from menu to function keys
or from function keys to menu

Enter your choice:

H: r

C: Departure airport:

H: Roanoke

C: Destination airport:

.

.

.

7.2.2. Dialogue Development Activities

Phase 1 - Requirements Specification:

A list of high-level system functions and their purposes is produced.

FUNCTION	DESCRIPTION
Flight information	To obtain flight information about scheduled flights, including from/to airport, departure/arrival times, carrier and flight number, and seat availability
Reservation	To reserve seat(s) on a specific flight on a specific date for any number of passengers
Payment	To apply a payment to a reservation with cash, check, or credit card
Ticketing	To issue ticket(s) to a customer for a reserved flight
Modification	To modify any portion of an existing reservation, including date, time, number of seats, class, departure/arrival airport, carrier and flight number
Cancellation	To cancel any existing reservation
Customer information	To obtain customer information about any portion of an existing reservation; e.g. flight confirmation

Phase 2 - Analysis:

DFDs are produced which show data flow between functions of the system. In the nodes of the diagrams given in figures 8 and 9, a circle represents a dialogue function, a rectangle represents a computational function, a hexagon represents a user function, and a circle within a

rectangle represents both combined. Such combined nodes must eventually be decomposed into pure dialogue and pure computational modules. Along the edges are the data that are being passed from one node to the next. Information along the edges enclosed in < > indicates a predicate or choice selection which conditionally determines the next node in a sequence.

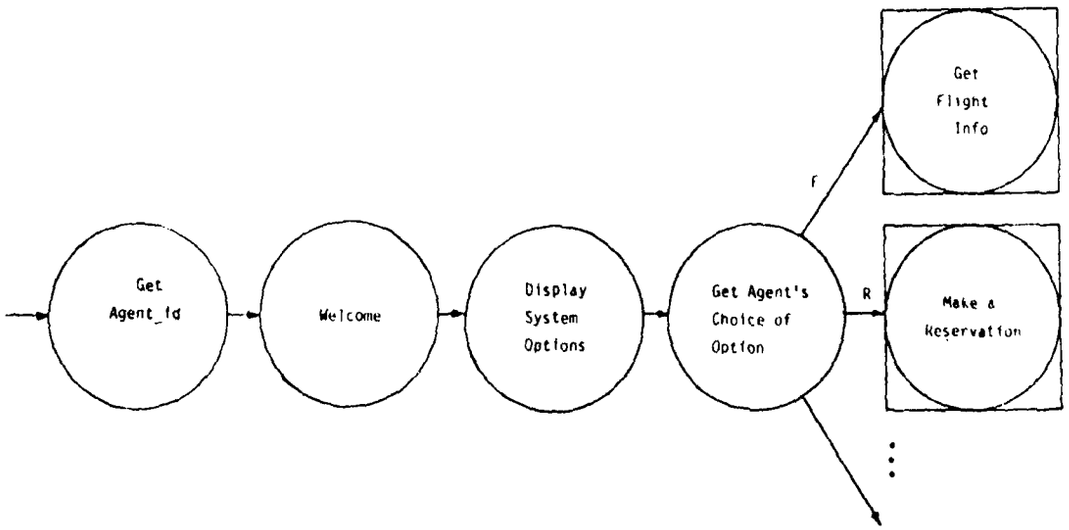


Figure 8. High-Level DFD for an Airline Reservation System

Expanding the "Make a reservation" node following the "R" choice gives the UDC shown in figure 9, with dialogue and computation separated. DFDs can be "executed" using the dialogue simulator DFD trace facility, so that the dialogue author and the application programmer can see the logical flow of the system at this point in its development.

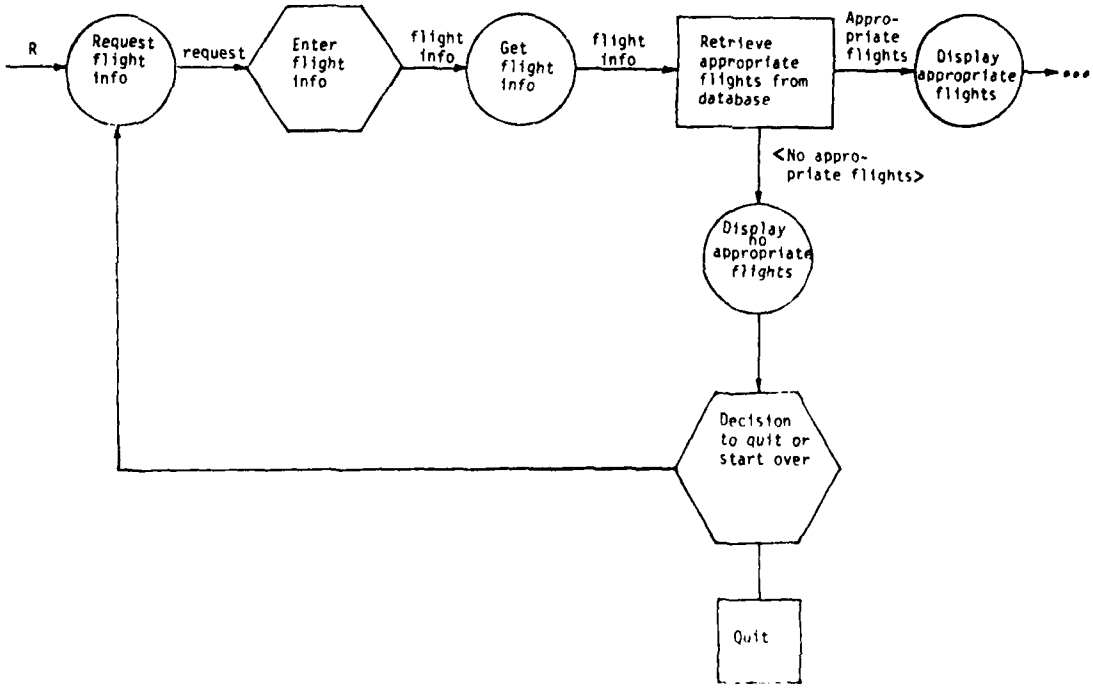


Figure 9. Partial UDC Diagram for "Reservation" Function

Phase 3 - Design:

Intermodule communication specifications are determined, primarily from the DFDs and UDCs. A structure chart relates the dialogue and computational modules by calls between them. (See figure 10.)

Phase 4 - Implementation:

The dialogue author constructs the dialogue modules using the automated tools in AIDE so that they are executable. At the same time, the application programmer codes the computational modules. Fragments of the source code for the dialogue modules and the computational mod-

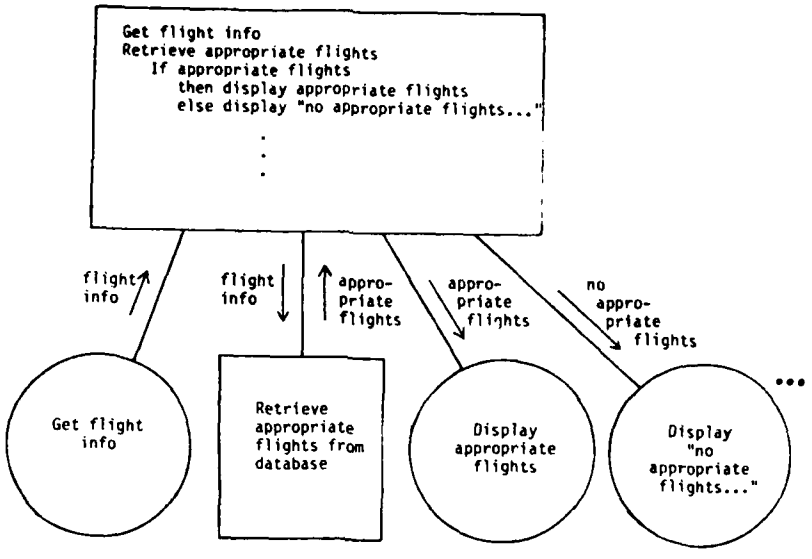


Figure 10. Structure Chart for "Reservation" Function

ules are given below using DMS procedures of REQUEST, ACCEPT, SEND, RECEIVE, END_REQUEST, INPUT, and ALPHA_TEXT. These procedures coordinate communication between the processes containing dialogue modules and those processes containing computational modules in a multiprocess execution environment. A complete explanation of these procedures and their functions is given in [EHRR82a and EHRR82b]. As noted earlier in the discussion of the DMS services, this interprocess communication will eventually be automated so that it is transparent to the dialogue author.

Note the calling convention used throughout: the computational program REQUEST parameter, in each case, is the required information to be obtained (e.g., 'agent_id' or 'options') or the function to be

performed (e.g., 'welcome'). The dialogue module to be called as a result of each REQUEST becomes the parameter name suffixed with _DM (e.g., agent_id_dm, etc.). This helps to maintain a correspondence between the REQUESTed information or function and the dialogue module which is executed.

COMPUTATIONAL PROGRAM FRAGMENT:

```
                .
                .
                .
[1]       [establish need for agent id]
[2]       REQUEST ('dmexec','agent_id',0);
[10]      RECEIVE (agent_id, 12):
          END_REQUEST;

[11]      REQUEST ('dmexec','welcome',0);
          END_REQUEST;

[13]      REQUEST ('dmexec','options',0);
[18]      RECEIVE (option,1);
          END_REQUEST:

[19]      case option of
             f: REQUEST ('dmexec','flight_info',0);
[20]         r: REQUEST ('dmexec','reservation',0);
             p:  ....
               .
               .
               .
```

'DMEXEC' (executor for dialogue modules):

```
                .
                .
                .
[3]         ACCEPT (request_name);
[4]         case request_name of

               agent_id:
                  begin
                     AGENT_ID_DM;
[9]                  SEND (agent_id, 12):
                  end;

               welcome:
                  WELCOME_DM;

               options:
                  begin
                     OPTIONS_DM;
[17]                 SEND (option,1)
                  end;

               flight_info:
                  begin
                     FLIGHT_INFO_DM;
                           .
                           .
                           .
               reservation:
                  .
                  .
                  .
```

DIALOGUE MODULES (written in dialogue language):

```
[5]     proc AGENT_ID_DM (input,output);
          [declarations]
                .
                .
                .
          repeat
[6]         ALPHA_TEXT ('tt', 4,1, 'Please type in your agent id.',
                  29, 'abs');
            i=10
[7]         INPUT ('tt', agent_id, i, 'echo&edit');
            VALIDATE (agent_id, error_type, parameters...,
                  return_code);
            case return_code of
[8]           0:  return;
              1:  ALPHA_TEXT ('tt', 5,1, 'Sorry, that id is not
                        valid. Please try again.', 46, 'abs')
             end;
          until forever
        end;


[12]    proc WELCOME_DM (input,output);
          [declarations]
                .
                .
                .
          NEW_FRAME ('tt', 'top')
          ALPHA_TEXT ('tt', 4,1, 'Welcome to the airline
               reservation system.' 42, 'abs')
        end;
```

```
[14]     proc OPTIONS_DM (input,output);
           [declarations]
                .
                .
                .
           ALPHA_TEXT ('tt', 5,1, 'Which of the following would
                you like to do?', 44, 'abs');
           ALPHA_TEXT ('tt', 6,1, '-------------------------
                ----------------------', 50, 'abs');
           ALPHA_TEXT ('tt', 7,1, '   F: Obtain flight information
                for an itinerary', 48, 'abs');
                   .
                   .
                   .
           ALPHA_TEXT ('tt', 17,1, 'Enter your choice:', 18,
                'abs');

           repeat
             i=1
[15]         INPUT ('tt', option, i, 'echo&edit');
[16]         VALIDATE (option, error_type, parameters...,
                        return_code);
             case return_code of
               0: return;
               1: ALPHA_TEXT ('tt', 20,1, 'Not a valid option.
                        Please try again.', 30, 'abs')
             end;
           until forever
         end;
```

In order to understand what all these various pieces of source code mean, and how they fit together, some explanation is necessary. Line numbers, enclosed in [ ], refer to the corresponding numbers at the left side of the page in the above code. System execution in this particular scenario establishes the need, in the computational program (line [1]), for a valid agent id. The DMS REQUEST procedure is invoked (line [2]) to call the dialogue module which will query the user for that information.

The ACCEPT procedure (line [3]) in the dialogue module executor (DMEXEC) obtains the name of the request for service (in this ins-

tance, agent_id) sent from the computational program in the REQUEST. The computational program and the dialogue executor are running concurrently at this point, in a multiprocess execution environment. DMEXEC waits until it gets a REQUEST and then calls the appropriate dialogue module to do the necessary dialogue.

The case statement (line [4]) determines that AGENT_ID_DM (line [5]) is the dialogue module which should be called at this point. Note that, with the exception of the VALIDATE function (discussed earlier in the dialogue language section), the dialogue module AGENT_ID_DM consists primarily of writing to (line [6]) and reading from (line [7]) the screen, i.e., dialogue with the user. When a valid agent id is input by the user (indicated by case 0 (line [8]) in the dialogue module), that valid agent_id is SENT (line [9]) back to the computational program, where it is RECEIVEd (line [10]) and computation then continues.

The WELCOME_DM REQUEST is executed (line [11]), and again received in DMEXEC through the ACCEPT procedure (line [3]). Note that a dialogue module can be a single line of output, in this case, WELCOME_DM (line [12]), because the system greeting ('Welcome to the airline reservation system') will be displayed only when the user first logs on, and not every time the options menu (immediately following the welcome in this particular instance) is displayed.

Now, the computational program REQUESTs the OPTIONS_DM dialogue module (line [13]), which is ACCEPTed in DMEXEC (line [3]) and the system options are displayed by the dialogue module OPTIONS_DM (line [14]). The user's choice of one of the options (in this transcript,

"r" for the reservation option) is received (line [15]) and validated (line [16]). At this point, the SEND procedure in DMEXEC (line [17]) passes the chosen option to the computational program where it is acknowledged by the RECEIVE procedure (line [18]). A case statement in the computational program (line [19]) determines the next system action (i.e., call and execute, through a REQUEST (line [20]), the dialogue modules that allow a reservation to be made).

Examining all parts of the source code for this very simple transcript demonstrates how dialogue and computational components of an application system will be written initially under DMS. Once the interprocess communication procedures are fully automated, this configuration will be straightforward to implement.

Phase 5 - Usage and Testing/Modification:

The implemented dialogue modules and computational modules are integrated into the completed application system, ready for the end users. As needed modifications are determined, the appropriate dialogue modules (and computational modules, as necessary) can be easily changed using the same automated tools provided by AIDE by cycling back through the development process.

8. SUMMARY

The need for effective human-computer interfaces is well-recognized. One goal of our research efforts in this area is to produce a human factored automated Dialogue Management System (DMS) for producing other human-factored application systems. DMS consists of not only the automated environment in which a dialogue author and application programmer work in parallel to produce application systems, but it also consists of a standard methodology to be used in the development of dialogues for those application systems. Our research has produced the concept of dialogue independence, which results in the separation of dialogue and computational components within a software system. This separation allows easy modification of dialogues so that a human-computer interface can be quickly modified to meet the needs of its users. The role of a dialogue author, whose main purpose is to create dialogues for human-computer interfaces, is also a new idea. By automating as much of the dialogue development and implementation process as possible, the dialogue author is freed from much of the tedium of "coding" dialogues and can concentrate on incorporating human factors into the dialogues to create an effective human-computer interface. Such an Author's Interactive Dialogue design Environment (AIDE) is being developed for use by the dialogue author. This environment consists of numerous automated tools for assisting in both dialogue design and implementation. This places the emphasis of human-computer interface development on what a dialogue will contain, not on how it will be implemented.

Many aspects of human factoring an application system user interface seem intuitively obvious. But these aspects are all too frequently ignored

in the rush to produce an operational system as soon as possible. Anything that takes time away from programming is considered by many people to be non-productive. But application systems which are developed without proper attention to the human-computer interface are frequently ineffective for the end user. With ever-increasing numbers of interactive computer users having wide ranges in experience, problems, and expectations, the need for an effective human-computer interface becomes a primary facet of system design. Ad hoc design methods and traditional manual development tools will rarely produce satisfactory systems. Providing a standard methodology and automated tools for the creation of useful, effective human-computer interfaces is "a challenge to scientific competence, engineering ingenuity, and artistic elegance" [SHNEB80].

REFERENCES

ACMCS81 ACM Computing Surveys, Special Issue on "The Psychology of Human-Computer Interaction." Vol. 13, No. 1 (March 1981).

CHERD76 Cheriton, D. R. "Man-Machine Interface Design for Timesharing Systems." Proceedings of ACM Annual Conference (1976).

EHRR82a Ehrich, R. W. "DMS - A System for Defining and Managing Human-Computer Dialogues." to appear in Proc. of Conference on Analysis, Design, and Evaluation of Man-Machine Systems, Baden-Baden, Germany (1982).

EHRR82b Ehrich, R. W. "The DMS Multiprocess Execution Environment." Department of Computer Science Technical Report, VPI&SU (1982).

ERGON80 Ergonomics, Special Issue on "Man-Computer Communication: Ergonomics and the design of computer dialogues." Vol. 23, No. 9 (September 1980).

FEINS82 Feiner, S., Nagy, S., and Van Dam, A. "An Experimental System for Creating and Presenting Graphical Documents." ACM Transactions on Graphics. Vol. 1, No. 1 (January 1982).

FOLEJ74 Foley, J. D. and Wallace, V. L. "The Art of Natural Graphic Man-Machine Conversation." Proceedings IEEE. Vol. 62, No. 4 (1974).

HANAP80 Hanau, P. R. and Lenorovitz, D. R. "Prototyping and Simulation Tools for User/Computer Dialogue Design." Computer Graphics SIGGRAPH 1980 Conference Proc. Vol. 14, No. 3 (July 1980).

HARTH82 Hartson, H.R. and Ehrich, R. W. "The Management of Dialogue for User/Software Interfaces." Department of Computer Science Technical Report, VPI&SU (1982).

HFICS82 Proceedings of the Conference on Human Factors in Computer Systems, Gaithersburg, Md. (March 1982).

IBMSJ81 IBM Systems Journal, Special Issue on "Human Factors." Vol. 20, Nos. 2 & 3 (1981).

JOHD82a Johnson, D. H. and Hartson, H. R. "Application of the DMS Methodology to the Development of the Menu Formatter, an AIDE Tool." Department of Computer Science Technical Report, VPI&SU (in progress).

JOHD82b Johnson, D. H. and Hartson, H. R. "Interactive Language Representation and Recognition under DMS and AIDE." Department of Computer Science Technical Report, VPI&SU (in progress).

KENNT74 Kennedy, T.C.S. "The Design of Interactive Procedures for Man-Machine Communications." Int. J. of Man-Machine Studies. Vol. 6 (1974).

MARTJ73 Martin, J. Design of Man-Computer Dialogues. Prentice-Hall, Inc., Englewood Cliffs, N.J. (1973).

REISP82 Reisner, P. "Further Developments toward Using Formal Grammar as a Design Tool." Proc. Conf. on Human Factors in Computer Systems, Gaithersburg, Md. (March, 1982).

SALTG80 Salton, G. "Automatic Information Retrieval." IEEE Computer. (September 1980).

SHNEB80 Shneiderman, B. Software Psychology, Human Factors in Computer and Information Systems. Winthrop Publishers, Inc., Cambridge, Mass. (1980).

SHNEB82 Shneiderman, B. "Multi-Party Grammars." IEEE Transactions on Systems, Man, and Cybernetics (1982).

SIGSO81 Proceedings (Part II) of the Conference on Easier and More Productive Use of Computer Systems, Ann Arbor, Mich. (May 1981).

SMITS81 Smith, S. "Man-machine interface (MMI) requirements definition and design user considerations: a progress report." MITRE Corp., Bedford, Mass. (February 1981).

WILLB81 Williges, B.H. and Williges, R. W. "User Considerations in Computer-Based Information Systems." Department of Industrial Engineering & Operational Research Technical Report, VPI&SU (1981).

YUNTT82 Yunten, T. and Hartson, H. R. "Human-Computer System Development Methodology for the Dialogue Management System." Department of Computer Science Technical Report CSIE-82-7, VPI&SU (1982).

OFFICE OF NAVAL RESEARCH

Code 442

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

Capt. Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Department of the Navy

Engineering Psychology Programs
Code 442
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Communication & Computer Technology
Programs
Code 240
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Tactical Development & Evaluation
Support Programs
Code 230
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Manpower, Personnel and Training
Programs
Code 270
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Information Systems Program
Code 433
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Physiology & Neuro Biology Programs
Code 441B
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Commanding Officer
ONR Eastern/Central Regional Office
ATTN: Dr. J. Lester
495 Summer Street
Boston, MA 02210

Commanding Officer
ONR Western Regional Office
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco, CA 96503

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375

Dr. Michael Melich
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. Louis Chmura
Code 7592
Naval Research Laboratory
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

Department of the Navy

Dr. Jerry C. Lamb
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N-71
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation Group
Naval Training Equipment Center
Code TAEG
Orlando, FL 32813

Dr. Albert Colella
Combat Control Systems
Naval Underwater Systems Center
Newport, RI 02840

K. L. Britton
Code 7503
Naval Research Laboratory
Washington, D.C. 20375

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Mr. Warren Lewis
Human Engineering Branch
Code 8231
Naval Ocean Systems Center
San Diego, CA 92152

Dr. A. L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Department of the Navy

HQS, U.S. Marine Corps
ATTN: CCA40 (MAJOR Pennell)
Washington, D.C. 20380

Commanding Officer
MCTSSA
Marine Corps Base
Camp Pendleton, CA 92055

Chief, C^3 Division
Development Center
MCDEC
Quantico, VA 22134

Naval Material Command
NAVMAT 0722 - Rm. 508
800 North Quincy Street
Arlington, VA 22217

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, D.C. 20361

Commander
Naval Air Systems Command
Crew Station Design,
NAVAIR 5313
Washington, D.C. 20361

Mr. Phillip Andrews
Naval Sea Systems Command
NAVSEA 0341
Washington, D.C. 20362

Commander
Naval Electronics Systems Command
Code 81323
Washington, D.C. 20360

Dr. Arthur Bachrach
Behavioral Sciences Department
Naval Medical Research Institute
Bethesda, MD 20014

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab.
Naval Submarine Base
Groton, CT 06340

Department of the Navy

Head
Aerospace Psychology Department
Code L5
Naval Aerospace Medical Research Lab.
Pensacola, FL 32508

Dr. James McGrath
CINCLANT FLT HQS
Code 04E1
Norfolk, VA 23511

Navy Personnel Research and
Development Center
Planning & Appraisal Division
San Diego, CA 92152

Dr. Robert Blanchard
Navy Personnel Research and
Development Center
Command and Support Systems
San Diego, CA 92152

LCDR Stephen D. Harris
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Dr. Julie Hopson
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

Mr. Jeffrey Grossman
Human Factors Branch
Code 3152
Naval Weapons Center
China Lake, CA 93555

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93042

Mr. J. Williams
Department of Environmental
Sciences
U.S. Naval Academy
Annapolis, MD 21412

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Department of the Navy

Human Factors Section
Systems Engineering Test
Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

Dr. Robert Carroll
Office of the Chief of Naval
Operations (OP-115)
Washington, D.C. 20350

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D.C. 20310

Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Director, Organizations and
Systems Research Laboratory
U.S. Amry Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S Army Human Engineering Labs.
Aberdeen Proving Ground, MD 21005

ARI Field Unit-USAREUR
ATTN: Library
C/O ODCSPER
HQ USAREUR & 7th Army
APO New York 09403

Department of the Air Force

U.S. Air Force Office of Scientific
Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Department of the Air Force

Dr. Earl Alluisi
Chief Scientist
AFHRL/CCN
Brooks AFB, TX 78235

Foreign Addressees

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LJ
ENGLAND

Dr. Kenneth Gardner
Applied Psychology Unit
Admiralty Marine Technology
Establishment
Teddington, Middlesex TW11 OLN
ENGLAND

Director, Human Factors Wing
Defence & Civil Institute of
Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

Dr. A. D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, CB2 2EF
ENGLAND

Prof. Brian Shackel
Department of Human Science
Loughborough University
Loughborough, Leics, LE11 3TU

Other Government Agencies

Defense Technical Information Center
Cameron Station, Bldg. 5
Alexandria, VA 22314

Dr. Craig Fields
Director, System Sciences Office
Defense Advanced Research Projects
Agency
1400 Wilson Blvd.
Arlington, VA 22209

Other Government Agencies

Dr. M. Montemerlo
Human Factors & Simulation
Technology, RTE-6
NASA HQS
Washington, D.C. 20546

Other Organizations

Dr. Jessee Orlansky
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

Dr. Robert T. Hennessy
NAS - National Research Council
Committee on Human Factors
2101 Constitution Ave., N.W.
Washington, D.C. 20418

Dr. Elizabeth Kruesi
General Electric Company
Information Systems Programs
1755 Jefferson Davis Highway
Arlington, VA 22202

Mr. Edward M. Connelly
Performance Measurement
Associates, Inc.
410 Pine Street, S.E.
Suite 300
Vienna, VA 22180

Dr. Richard W. Pew
Information Sciences Division
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02238

Dr. Stanley N. Roscoe
New Mexico State University
Box 5095
Las Cruces, NM 88003


FILMED
9 8