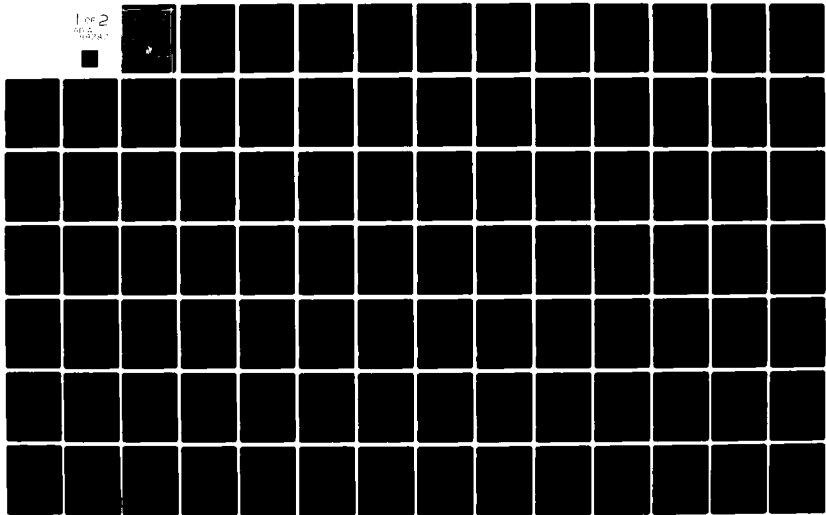
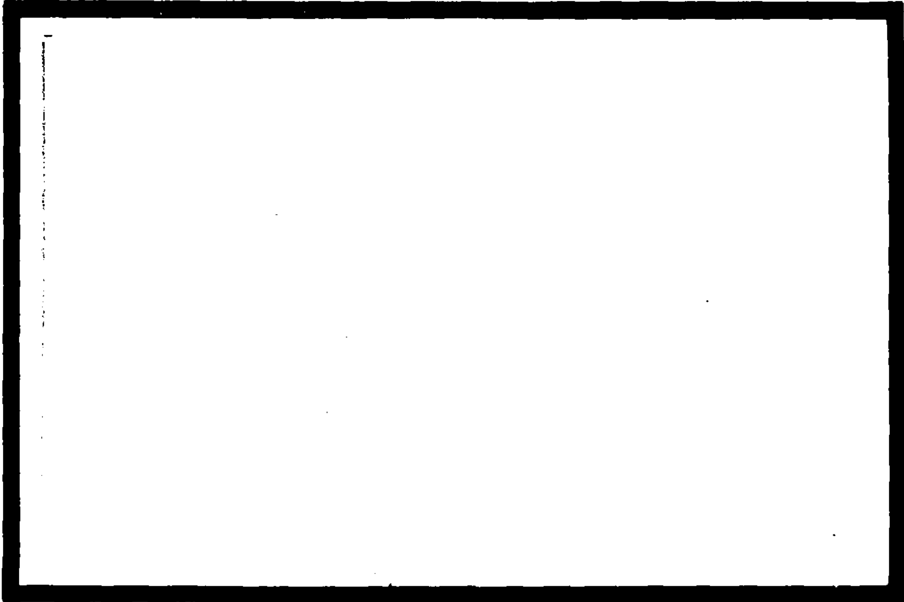
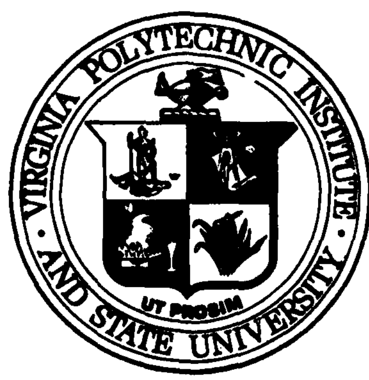
AD-A118 287 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG --ETC F/G 9/2
HUMAN-COMPUTER SYSTEM DEVELOPMENT METHODOLOGY FOR THE DIALOGUE --ETC(U)
MAY 82 T YUNTEN, H R HARTSON N00014-81-K-0143
UNCLASSIFIED CSIE-82-7 NL



AD A118287

DTIC FILE COPY





DTIC
ELECTE
AUG 13 1982
S D
D

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

DISTRIBUTION STATEMENT A
Approved for public release;

82 08 13 014

CSIE-82-7 May 1982

HUMAN-COMPUTER SYSTEM DEVELOPMENT METHODOLOGY
FOR THE
DIALOGUE MANAGEMENT SYSTEM

Tamer Yunten

H. Rex Hartson

Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

TECHNICAL REPORT

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

Approved for Public Release; Distribution Unlimited

Reproduction in whole or in part is permitted
for any purpose of the United States Government

BLANK PAGE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER CSIE-82-7	2. GOVT ACCESSION NO. A118 287	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) HUMAN-COMPUTER SYSTEM DEVELOPMENT METHODOLOGY FOR THE DIALOGUE MANAGEMENT SYSTEM		5. TYPE OF REPORT & PERIOD COVERED Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Tamer Yunten 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 442 800 North Quincy Street Arlington, VA 22217		12. REPORT DATE May 1982 13. NUMBER OF PAGES 97
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 system, lifecycle, human factors engineer, human-computer dialogue, dialogue independence, dialogue author, software engineer, programmer, user, Dialogue Management System, methodology

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

In this report a system development methodology for human-computer systems is constructed. The methodology views humans as functional elements of a system in addition to computer elements. The disciplined approach of the software engineer (SWE) and the user oriented approach of the human factors engineer (HFE) are combined into a methodology which features a parallel and cooperative work environment. Parts of the software which implement human-computer interaction are separated from the rest of the soft-

DD FORM 1473, 1 JAN 73 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-LF-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. ABSTRACT (continued)

ware. The Dialogue Management System (DMS) is a human-computer system which provides automated tools to support the use of the new methodology by the SWE and HFE to produce other human-factored human-computer systems. The methodology is applied to the development of the DMS too.

ACKNOWLEDGMENT

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 authors wish to thank Debby Johnson for her help in preparation of this report.

1. INTRODUCTION

The computer systems of today are difficult to use because of poorly human-engineered user interfaces. In the construction of large computer systems, the software developer is faced with a great deal of complexity, even when human engineering for the end user is not considered. Hence, software engineering, being a young field, has so far devoted most of its effort to reducing this developmental complexity without much concern for human-factors that affect the performance of the end user. Consequently, the current software development methodologies make the job of the software developer easier by providing tools and guidelines for system analysis, design, and maintenance. However, they do not embody guidelines, methods, and tools for producing human-engineered software interfaces. As a result, with the methodologies of today, the effectiveness of the human-computer interface varies with the human-factors knowledge of the software developer. It also varies with the emphasis that the developer puts on this interface (which is often close to none).

On the other hand, human-factors engineering is a system development discipline by itself. The human-factors specialists require the software engineers to work in cooperation with them during computer-related system development. However, just as a software developer's knowledge of human-factors might be varying, the human-factors specialist's knowledge of computer programming might also vary. Therefore, for these two specialists to work effectively, a methodology is required which embodies the needs and principles of the two disciplines, while providing the appropriate development and communication tools.

This report addresses the development of a human-engineered methodology to be used by the "human-factors engineer" and the "software engineer" during their cooperative work of developing and maintaining human-engineered systems. The ultimate objective is to provide an automated environment for the specialists mentioned, by supporting this developmental methodology with automated tools which also serve the human-factors needs of these specialists.

The methodology has been greatly influenced by the way a system is viewed. In general, a system is defined as a collection of elements that interact with each other to achieve certain goals. Usually, in the software engineer's world, the word "system" addresses a collection of computer elements to be used by humans (e.g., a software system). Humans are not included in the definition of a system, and what is modelled and analyzed is a computer system. In our view, both humans and computer are functional parts of the same system, namely a human-computer system. What we shall model and analyze will be the operation of a human-computer system.

In section 2, the human-computer system concept is presented and the need for a disciplined approach in system development is discussed. It is shown that, because of the need to cooperate with the human-factors engineer in system development, the use of a disciplined approach is vital. At the end of the section, the Dialogue Management System, which will contain the automated environment to support the human-factors engineer and the software engineer, is introduced. In sections 3 and 4, the views of the human-factors engineer and the software engineer in system development are presented, respectively. In section 5, these two

views are integrated and the fundamentals of a human-engineered methodology for human-computer system development are constructed. The use of a human-engineered methodology is also motivated for the development of the Dialogue Management System itself. In section 6, the methodology for development of the Dialogue Management System is discussed and the current status of the system is presented.

2. HUMAN-COMPUTER SYSTEM (HCS) DEVELOPMENT

Computer users and their managers have become increasingly aware that poorly designed human-computer interfaces, while perhaps initially cheaper to design, result in significant costs during the operational lives of the systems. To show the negative contribution of a non-human-engineered interface on the operational-life cost of a software system, let us start with defining a human-computer system (HCS) as shown in

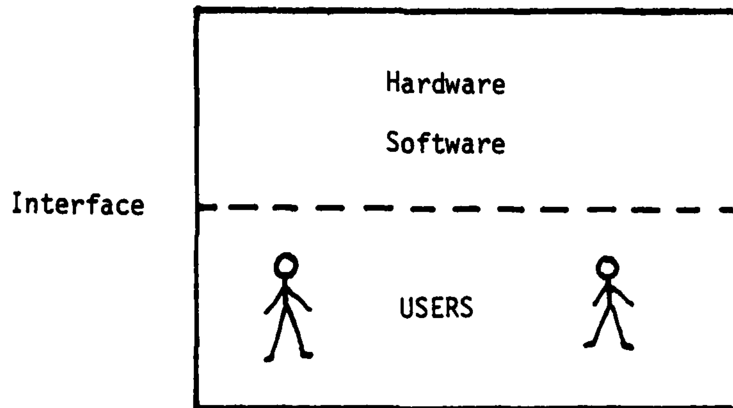


Figure 2.1. A human-computer system.

Figure 2.1.

A human-computer system is composed of hardware and software on one side of an interface and the humans (users) on the other side. Examples of HCSs are airline reservation systems, air traffic control systems, automated office systems, interactive text editors, and project teams developing other HCSs. An HCS is an example of what industrial engi-

neers call a man-machine system, where the machine in an HCS is the computer. Hence, most industrial engineering criteria for man-machine system development apply to HCS development (e.g., work design, time and motion studies, human-factors, cost, etc.). One aspect of an HCS development is the human engineering of the human-computer interface. As shown in the figure, the users and the computer interact with each other through the interface provided by the developers of the system. Obviously, if the interface is human-engineered, the productivity of the user (and likewise that of the HCS) increases; otherwise productivity decreases.

A common objective in the creation of human-computer systems is to get maximum return for minimum cost. "Return" is the useful work obtained from the system during its life. "Cost" has the following four components:

1) Operation Cost of HCS
2) Software Maintenance Cost
3) Software Development Cost
4) Hardware Cost

The productivity of the HCS is determined by the efficiencies of both the user and the computer. Therefore, in the development of the system, the user's role in the HCS should be modelled and analyzed. Unfortunately, until recently, the emphasis has been only on the software system during its development. Although the phrase "human-engineered" might appear in the list of system requirements, it mostly remains as a "wish", rather than becoming an implemented reality. (Note that, when humans start using the software engineer's system, the even-

tual system is an HCS, but the parts usually do not fit each other.) In current software methodologies, the major emphasis is on the last three cost elements above. The major concern is minimizing the total cost of software development, software maintenance, and hardware by balancing the hardware cost (e.g., resource utilization, speed of execution) against development and maintenance cost (e.g., elimination of cryptic tricks which save microseconds and bytes in favor of development and maintenance simplicity). In other words, the current methodologies address cost within the software development and maintenance environment. The productivity of the system during use is mostly ignored. For instance, while "computer response time" is a major issue in the development of interactive systems, "user response time" is almost never discussed. On the other hand, the efficiency with which the user can interact with the computer greatly influences the productivity of the HCS of which the person is a part. Our goal is to develop a system development methodology which considers the user performance as well as the computer performance.

One of the reasons for inefficient human-computer interfaces in current interactive systems is that human-computer dialogue is built inextricably into the computational software, and software vendors become committed to designs that are too expensive to reprogram [HARTH82]. Hence, our starting point in the design of the software part of an HCS is the separation of the human-computer communication parts of the software from the computational parts of the software. In fact, a new role of dialogue author, to design and construct the human-computer dialogue, is kept separate from the role of programmer which is to design and construct the computational software [JOHND82].

In the following we shall discuss alternative approaches to HCS development.

2.1. AD HOC APPROACH IN HCS DEVELOPMENT

Many computer systems of today are still developed in an ad hoc way, without the use of an engineering-like disciplined approach. The analysis and design phases frequently are left out. The developers attempt to produce the target system directly through use of the implementation facilities (e.g., O.S. commands, editors, compilers, debuggers) provided

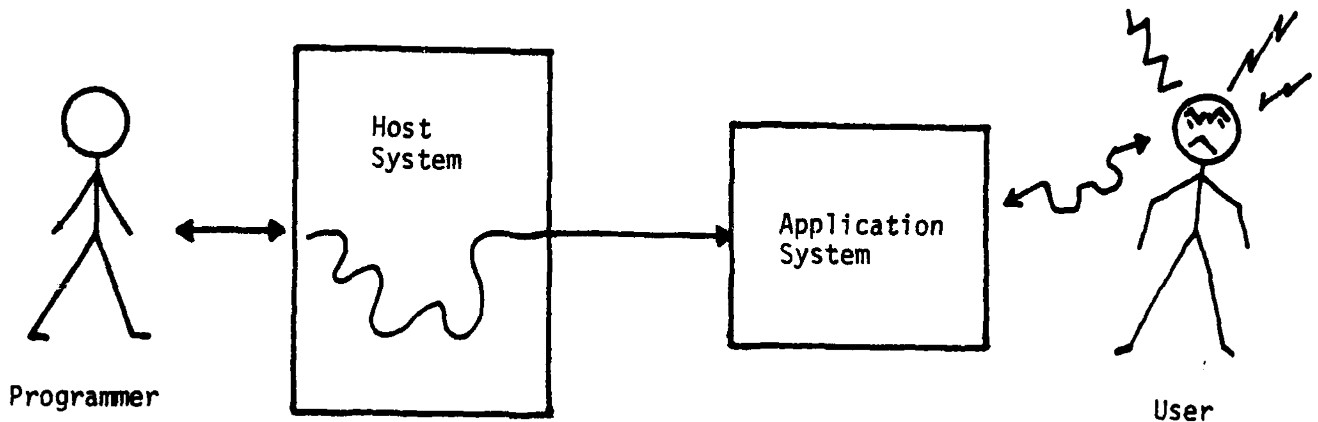


Figure 2.2. Ad hoc system production.

by the host computer. The situation is shown in Figure 2.2. Construction of the system begins before the system developers know exactly what are the system specifications. Trying to construct a system purely

through use of the host system services, i.e. simultaneously searching for both the system model and the implementation solution, is a very difficult task. This is similar to performing the construction and blue-print preparation of a house simultaneously [WASSA80]. This approach results in loss of time and effort. Although the developer gets the false feeling of being productive, much of what is produced often gets discarded, lowering the overall productivity. This approach might be successful in developing small and simple systems, but simply cannot deal with the complexity of large systems. Consideration of human-factors for the user would only increase the complexity. Thus, the developer is mainly interested in getting the system working, no matter how it interacts with the user. The messages displayed are often meaningful mostly to the developer and often those messages assume a thorough knowledge of the structure of the system. Because of the ad hoc development process, the system is seldom complete. Some of the functions which ought to be in the system are not considered by the developer and thus are not included.

If there is more than one person involved in the developmental process, things get worse. The participants have communication problems. Since most of the communication is done in an ad hoc way, the developers have difficulty understanding each other. They might have different models in their minds and different levels of knowledge of the system.

As Wasserman [WASSA80] states, software creation based on ad hoc techniques has resulted in severe time delays and cost overruns and sometimes even the complete failure of a project. The systems produced with this approach were not maintainable because the code was incompre-

hensible (as a result of poor programming practices) and usually no system representation existed other than the code. In addition to being difficult to maintain, these systems contained unpredictable errors and did not meet user requirements.

2.2. DISCIPLINED APPROACH IN HCS DEVELOPMENT

It is a well accepted fact in the software engineering community that ad hoc approaches used in the development of large software systems are very likely to end up with failure. As Weinberg says [WEINV80], "Experiences with projects - which have been late, over budget, inflexible to change, and unacceptable to the users - support the point better than anything that could be said." Some experiments made by Basili [BASVR79] have shown that a disciplined approach has resulted in higher development efficiency compared to an ad hoc approach, in multi-person teams. Hosier [HOSIJ78] states the common view of many software engineering authorities as : "The need for a disciplined approach to the software design process seems beyond question."

A disciplined approach uses a methodology and has several advantages. First, it helps the developer in managing the complexity of the developmental process. Second, when a methodology is followed by multi-person teams, the communication problems are reduced; the methodology provides a framework for the context of the discussion and the necessary tools (e.g., graphical tools such as functional flow diagrams) for unambiguous communication. The situation is shown in Figure 2.3. The dif-

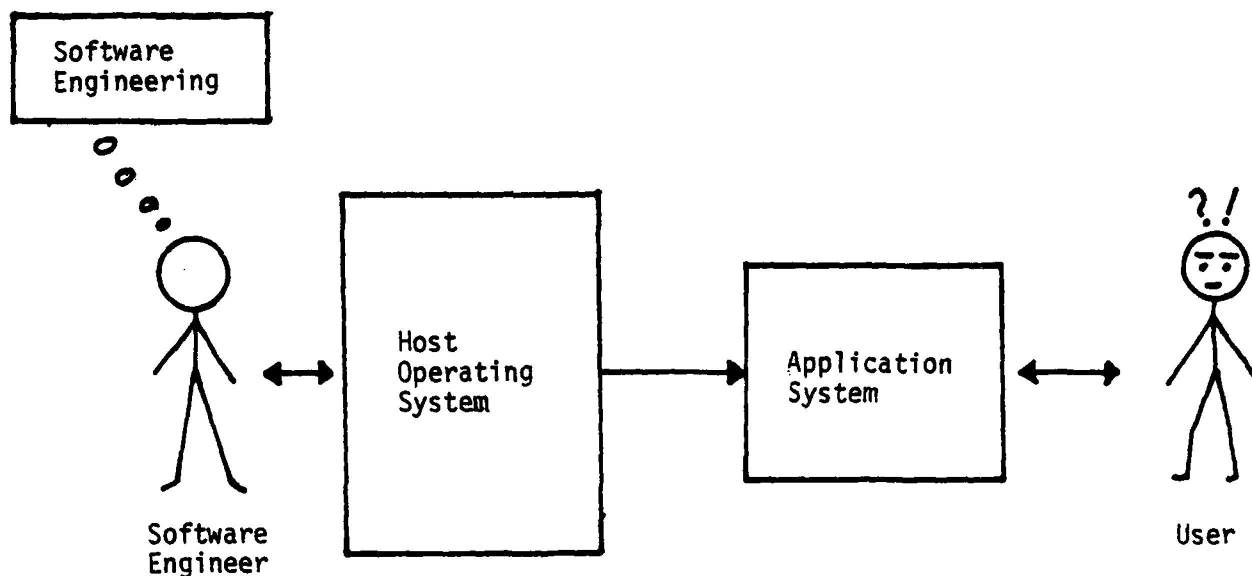


Figure 2.3. Disciplined approach in HCS development.

ference between Figure 2.2 and Figure 2.3 is that the software developer is following a disciplined approach instead of an ad hoc one. The guidelines for the approach are from the discipline of software engineering. But still, the HCS developed by a software engineer in a disciplined manner is not necessarily a human-engineered HCS. This is mainly because the current software engineering methodologies do not embody the guidelines and principles for human-engineered HCS development. As was mentioned in the introductory part of this report, the extent to which the product HCS is human-engineered depends on the human-factors knowledge of the software engineer (generally inadequate). Therefore the software engineer needs the help of the human-factors engineer. In the next sections this parallel work environment will be discussed.

2.2.1. HCS Development Methodology

While it is very difficult for programmers alone to be productive and to communicate among themselves with an ad hoc approach, it is even more difficult when a specialist from a different discipline is

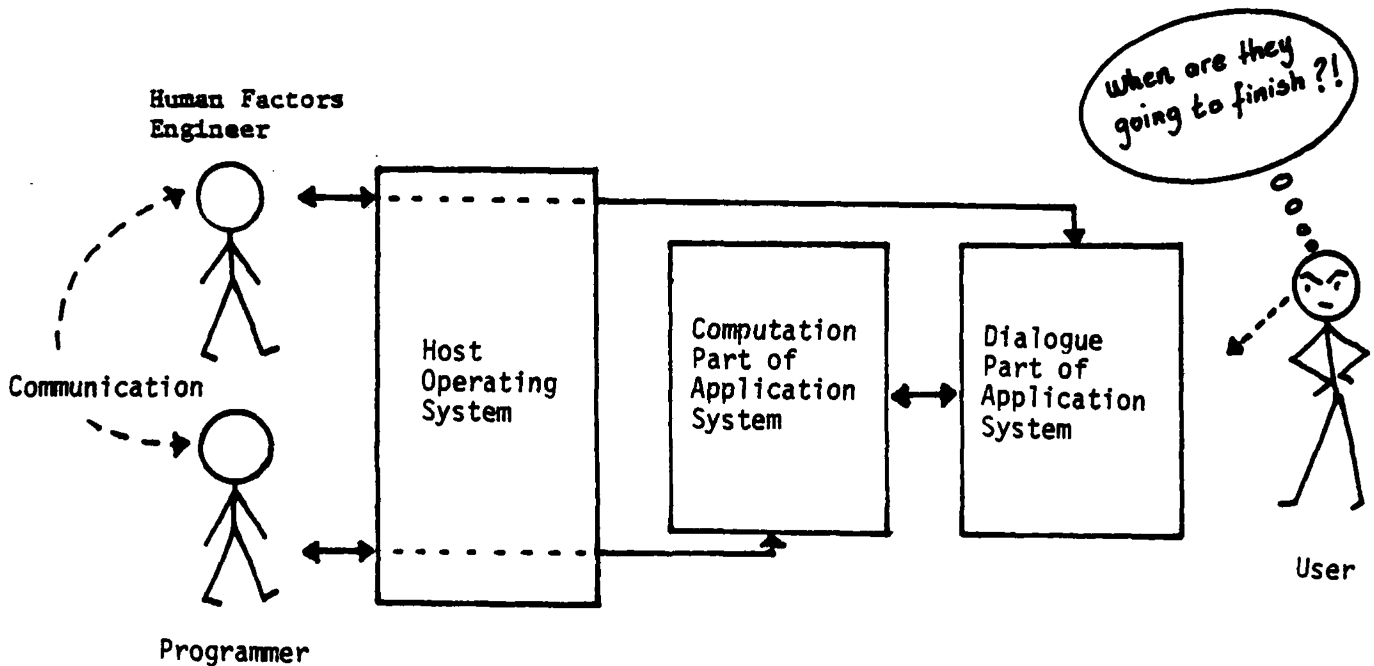


Figure 2.4. Human-factors engineer and the programmer in ad hoc system development.

involved, as illustrated in Figure 2.4.

In Figure 2.5 the human-factors engineer and the software engineer are shown using an HCS development methodology, which will be discussed in detail in later sections. There are many roles associated with the development of large software systems. To reflect the separation of software into computational and dialogue components, the roles are

divided into two classes. All human-engineering activities, including dialogue authoring, are performed by the class denoted as HFE (human-factors engineer). Similarly the class called SWE (software engineer) will represent roles which typically include systems analysts and pro-

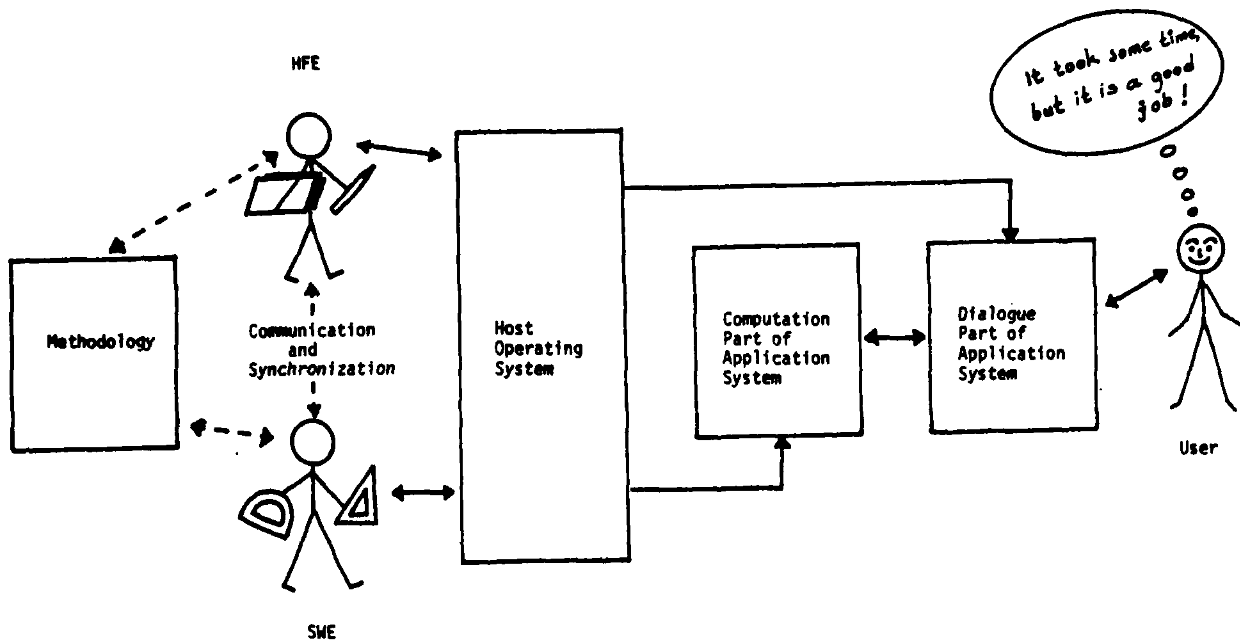


Figure 2.5. HFE and SWE using a methodology in HCS development.

grammers. The dialogue component of the target application software, which provides the user-computer communication, is implemented by the HFE and the rest of the software (computational component) is implemented by the SWE. In this figure, although a methodology is used, the SWE and HFE work using the host system interface. There is no automated

aid in the development of the parts of the application system. The disadvantage of this configuration is that the HFE is required to know the host system facilities to implement the user interface of the target application system. There are two alternatives to eliminate this disadvantage:

1. The specifications of a human-engineered system can be produced by SWE and HFE manually (e.g., using graphical tools), and the construction of both the dialogue and computational components can be done by the SWE alone (either manually or by automated support).
2. A high level dialogue development facility enables the HFE to construct the dialogue component of the target software without having to use the debuggers, editors, and programming languages of the SWE's world.

The first alternative above requires the communication between the SWE and the HFE to occur down to the level of implementation details of the user-computer interface. This detailed communication requirement would slow down the system development process. Besides, if the first alternative is chosen, the HFE would be idle much of the development time. Therefore, providing the HFE with the tool for constructing the dialogue component of the target software results in a more productive environment. This tool is called the Author's Interactive Dialogue design Environment (AIDE) and is discussed in [JOHND82]. The Dialogue Management System provides this automated tool and is presented in the next section.

2.2.2. Dialogue Management System

The Dialogue Management System (DMS) itself is an HCS. Humans in the DMS include the HFE and the SWE. The system development part of DMS software (as opposed to the execution environment part) represents the

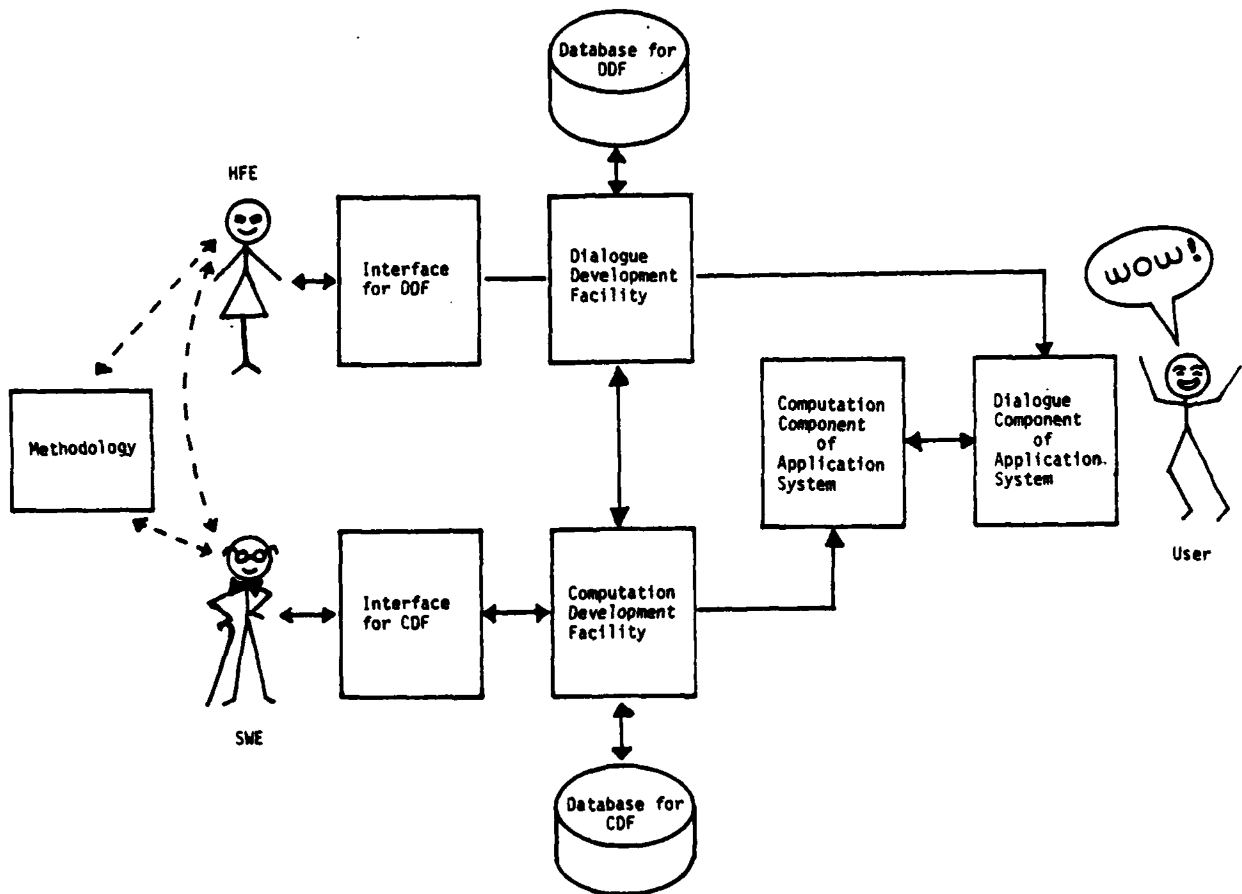


Figure 2.6. Dialogue Management System.

automated tools to be used by the SWE and the HFE. The general architecture of this part is shown in Figure 2.6. The HCS operates to

develop, test, experiment with, and maintain other human-engineered HCSs. The HFE and the SWE are working in parallel. They are using the HCS methodology to develop the application system. The verbal communication and activity synchronization of the two specialists is indicated by the dashed line between them. Each specialist has a separate developmental facility. The HFE and SWE are using the automated tools in the "Dialogue Development Facility" and the "Computation Development Facility", respectively. As shown in the figure, the output of DMS has the same logical structure at the two facilities of DMS (e.g., it consists of a dialogue component and a computational component). Each development facility provides its respective user with the means to access and modify the source modules in their respective databases. The source module databases include the modules of DMS, as well as the modules of the product systems.

3. VIEW OF THE HUMAN-FACTORS ENGINEER

> "The starting point for our consideration of human-factors discipline is obviously system development. Conceivably we could confine our interests merely to the use of equipment after the equipment had been designed and produced. In that event, however, we would merely be describing how people use the equipment, which is interesting but not crucial."

As the above quote from Meister (MEISD71) indicates, the HFE should be involved in the system development process, right from the beginning. The HFE is involved in:

a) function allocation between user and the computer

b) function allocation between users in a multiple-user team

c) human-engineered logic flow

d) matching equipment characteristics to user capabilities

e) arrangement of controls and displays.

3.1. FUNCTION ALLOCATION BETWEEN USER AND THE COMPUTER

An HCS is developed to achieve certain goals in an environment where these goals are the overall system requirements. An HCS takes its inputs from the environment and processes these inputs to achieve the system goals. For example, in Figure 3.1, the input-process-output form of an HCS, the goal of which is to develop software systems (much as this is the goal of DMS), is illustrated. To achieve the system goals the HCS must perform certain functions and also must satisfy the performance requirements and the constraints imposed by the environment.

One criterion for function allocation is the cost of system development and maintenance. Allocation of the system functions between humans

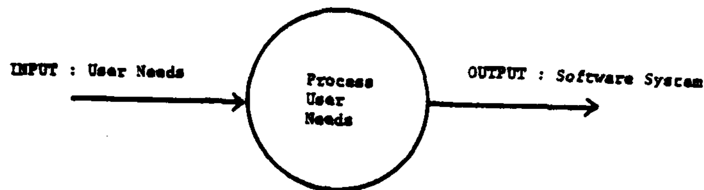


Figure 3.1. An HCS which produces software systems.

and the computer affects the performance of an HCS as well as its cost. While a high degree of automation might increase the reliability (assuming reliable software) and speed of HCS operation, it also increases the cost of development and maintenance. Conversely, a lower degree of automation results in lower development and maintenance cost while increasing the dependency of HCS performance on human capabilities. Hence, for potential function allocation configurations, the HFE must ensure that the user can perform the functions allocated to that person and satisfy the system performance requirements. General examples of performance requirements are : "directions to the aircraft have to be given at most in twenty seconds"; and "to prevent customer accumulation, average service-time for an airline reservation system must be three minutes". If average operator capability (e.g., speed, accuracy) is less than the system performance requirement, an automated solution for

the function implementation has to be considered. On the other hand, if a system performance requirement can be satisfied by a manually oriented implementation, automating the function would only increase development and maintenance costs, while possibly providing some comfort for the operator. Hence, there are tradeoffs involved in human engineering the HCS operation and cost of development and maintenance.

Note that to be able to compare the average user-capability with the system-operator performance requirement, data on average user-capability is needed. Collection of these data is a major activity of human-factors research.

Another criterion for function allocation is that humans are better in inductive reasoning while machines (computers) are better in deductive reasoning. Meister [MEISD71] claims that the engineer has the tendency to automate anything whether it is effective or not. If the computer is assigned to do a function which can be performed more efficiently by the human (or vice versa), the overall performance of the HCS will be lowered.

If an attempt is made to automate a function which requires inductive reasoning (e.g., construction of a functional flow diagram), the computer part will need the human's assistance for inductive reasoning through a dialogue. If this function can be more comfortably performed without any automated aid, no matter how good the dialogue is, the design will not be a human-engineered design. For instance, in an automated functional flow diagram (FFD) construction, if the device is a CRT terminal and the FFD under construction does not fit the screen, it would not be very comfortable to scroll in all directions to study and

develop the diagram. This function might be more effectively performed manually where work can be done on a large sheet of paper and where all parts of the diagram can be seen at one glance. An example of the assignment to a human of a function which could be better performed by the computer is one which would expect the user to manually find the logarithmic value of a number from a logarithm table.

The HFE begins an analysis by constructing a functional flow diagram from the system requirements [MEISD71]. The first version of the FFD does not differentiate between who performs the functions or how the functions are performed. Next, the performance requirements for the functions (if any) are inferred from the constraints imposed by the environment. Finally, the most cost effective function allocation configuration that satisfies the system performance requirements is selected.

In determining the functions to implement a system goal, the HFE usually works from the system goal backwards to the beginning. As an example of a FFD construction, consider the analysis of the HCS in figure 3.1, the goal of which is software system production. To produce an application software system, one has to implement the system (Figure 3.2). Obviously, to implement the system, one has to design the system. The interaction between design and implementation functions is shown in Figure 3.3.

Before designing the system, one has to analyze the system requirements (Figure 3.4). Prior to analysis of the system requirements, one has first to specify the system requirements. The completed FFD, in its most abstract form, is shown in Figure 3.5. Similarly, each function

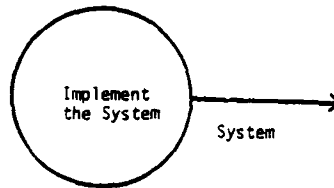


Figure 3.2. Partial FFD for producing an application system.

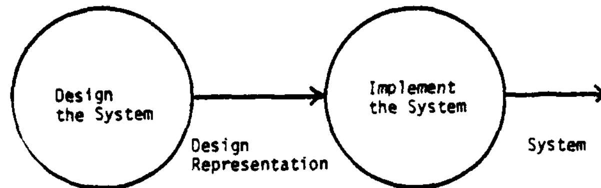


Figure 3.3. Partial FFD for producing an application system.

shown in Figure 3.5 can be broken into its component functions through levels of abstraction until the terminal functions, the processes of which are unambiguous, are reached.

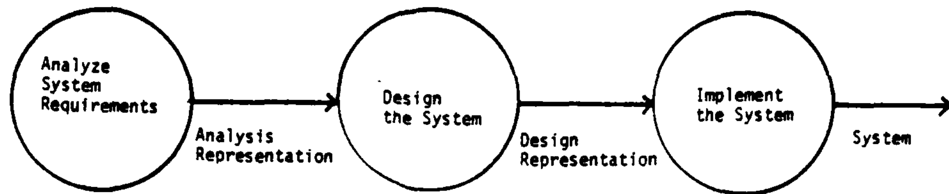


Figure 3.4. Partial FFD for producing an application system.

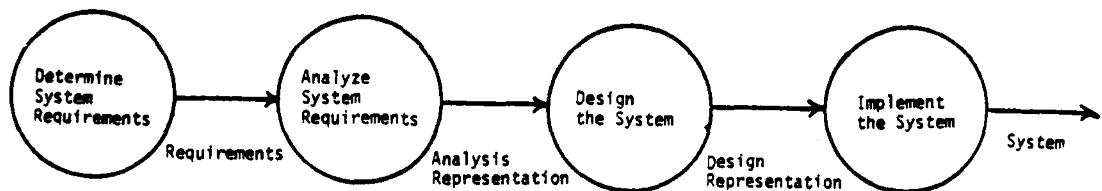


Figure 3.5. A high level FFD for producing an application system.

3.2. FUNCTION ALLOCATION AMONG USERS IN A MULTIPLE-USER TEAM

The same discussion applies if the user part of the HCS consists of more than one user. The functions to be performed by the users should be allocated according to their capabilities. An example of this activity is demonstrated in the DMS development. The dialogue author and the application programmer are users of DMS having different professional

backgrounds and knowledge. It is clear that we can neither expect a dialogue author (in general) to do systems programming nor the programmer to do dialogue authoring. Also, the function allocation between these two system developers will directly influence the design of the automated facilities (in Figure 2.6, the dialogue development facility and the computation development facility) and hence the communication interface between these facilities. For instance, if the HFE is involved only in the human-factors analysis and not in implementation, then the dialogue development facility becomes a programmer tool and not a tool for the HFE.

3.3. HUMAN-ENGINEERED LOGIC FLOW

The logical structure of a system can greatly affect the quality of the user interface, because the sequencing of the human-computer dialogue is dependent upon that structure. As an example, part of a function sequencing for an airline reservation might be:

Get customer name

Get number of people to fly

Get flight request

Normally one would expect "get flight request" to be the first function of the sequence. Obviously, if there are no seats available there is no need of going through the rest of the dialogue. Such a sequence would both result in irritated customers and loss of work time. The example above is trivial. But most sequences are not as trivial as the example above, and might not be caught by the programmer for a complex task.

3.4. MATCHING EQUIPMENT CHARACTERISTICS TO USER CAPABILITIES

If the computer demands more than the user can supply, the user will respond erroneously, out of synchronization (delayed response), or perhaps not at all [MEISD71]. An example of this is also demonstrated by DMS. The dialogue author will be provided with a human-engineered dialogue development facility (AIDE) because the dialogue author is not expected to be a computer scientist. The author would not be very happy if the following request was received: "Type the escape sequence for the text line." In general, during the development of a human-engineered HCS, it is the HFE's duty to help the SWE to specify a system which matches user capabilities.

3.5. ARRANGEMENT OF CONTROLS AND DISPLAYS

This activity is the last step of the analysis. After all the analyses above are finished, the HFE has a human-engineered work design. The logical structure of the system has already established what is to be communicated between the user and the computer. By arrangement of controls (e.g., buttons instead of typed commands) and displays (e.g., choice of colors in a CRT display), the HFE specifies the form of the communication. The performance requirements greatly affect the form of interactive user-computer communication. For example, for an HCS function which requires fast interaction of the operator and the computer, a function button would be a better choice than a typed input. For human engineering the control panel, the HFE might choose arrangement princi-

ples like: (a) arrangement by activation sequence of the software functions, (b) arrangement on the basis of the commonly used software functions, and (c) grouping similar function-activators together [MEISD71]. To be able to use these principles, the HFE has to know the functional layout of the system operation. The system FFD provides this information.

4. VIEW OF THE SOFTWARE ENGINEER

Software engineering was developed as a result of need for an engineering-like discipline in software development [WASSA77]. Software engineering has attempted to replace the ad hoc system development process with a more systematic approach [BATED77]. As a result, the life-cycle concept has been developed. Software is born with a set of requirements and goes through certain phases during its life, until it becomes obsolete or unmaintainable [HOSIJ78].

Freeman [FREEP77c] describes the phases of the software life-cyle as follows:

1. Needs Analysis - The needs of the user are established and analyzed.
2. Specification - In this phase, the functional description of the target system and the constraints on the system (e.g., hardware resource utilization) are specified. These specifications are prepared in view of their producibility with respect to financial and time resources of the development process.
3. Design - This phase receives the output of the specification phase as input. The system specifications are converted into a physical system representation. This representation can be done at a very general level or can be detailed down to a level approximating the program code it represents.
4. Implementation - Conversion of system specifications into code, testing, debugging, and performance measurement is done in this phase.
5. Maintenance - This phase involves all changes due to bugs discovered or for adapting the software to changes in system requirements.

4.1. BENEFITS OF EARLY EFFORT IN THE LIFE-CYCLE

In spite of all that is said about the negative consequences of ad hoc approaches, and of the explicit recognition of the phases of the software life-cycle, the approach still does not get the attention it deserves. The cost of discovering errors during early phases of the life-cycle is much lower than the cost of recovery in later phases [BOEHB76]. Recovery cost of errors discovered during the construction phase may be at a magnitude of hundreds of times the cost of effort spent at the requirements analysis phase [HOSIJ78]. The more effort that is put on the phases that precede implementation (e.g., requirements analysis, specification, and design), the less is the effort required in testing, debugging and maintenance [WASSA80]. Consequently, more effort at early stages results in less cost during the development and maintenance phases and less total cost over the life-cycle.

4.2. REQUIREMENTS ANALYSIS

As was the case with the HFE, the purpose of the requirements analysis is to define the problem. This single statement describes the importance of requirements analysis. The complete specification of the functions to be performed by a system can only be done after the problem is completely understood. The importance of preparing a complete specification of the system is described as follows [WASSA77]: (a) All parti-

cipants in the project development process can evaluate the specifications and the specifications can be brought to a final form which is accepted by everyone (users, analysts, etc.). (b) The complete specifications lead to the complete design of the system. (c) Testing of the requirements can be planned and performed systematically only if the requirements are defined.

The consistency and the completeness of the requirements definition play the largest role in determining the success of a software development process. Because user requirements have been ignored, some large software projects have resulted in complete failure, while some have had up to 95% of the finished product redone [WINSR75]. Ross [ROSSD77] states the same fact : "[that] a problem unstated is a problem unsolved seems to have escaped many builders of large computer application systems." He lists the consequences of this ignorance as "skyrocketing costs, waste and duplication, disgruntled users, endless series of patches and repairs euphemistically called maintenance." The following conditions listed by Royce [WINSR75], are reasons why inadequate requirements analysis will lead to the serious consequences described above:

1. Top-down design is impossible. Since the overall picture of the problem is not formed, bottom-up development is unavoidable. As a result, there is an undirected system-production and problem-understanding process. While certain functions are left unrecognized, certain functions are wastefully duplicated.
2. User is left out. If there is no specification that the user can follow, the user cannot know if the system is what is actually wanted.

3. Management is not in control. Without a proper requirements specification, it is not possible to determine in which state of the life-cycle the project is. The developers might claim that they are about to finish implementation, but if the implemented code does not reflect the actual system requirements, the project is virtually at the requirements definition phase.

Royce [WINSR75] makes the following claim about the above problems caused by the poor requirements analysis: "Although these problems are lumped under various headings to simplify discussion, they are usually all variations of one theme: poor management." The basic reason for poor requirements analysis is that for most developers the measure of productivity is the physical output they produce (e.g., physical design, or code). Therefore the requirements analysis phase of the life-cycle is left behind after a quick survey. But especially in the development of large and complex systems, this approach results in large recovery costs to repair the damage.

4.2.1. Sources of Requirements Errors

The following are major sources of requirements errors [MARKR76, CANNR77]:

1. Use of Natural Language - Due to the variety of expressions and the flexibility of the natural language, requirements usually cannot be specified in an explicit and unambiguous way using natural language.

If the problems and requirements are defined ambiguously, it is unlikely that a program which meets the intended needs can be developed. Marker [MARKR76] emphasizes this point strongly : "If there are two interpretations of the same statement, Murphy law guarantees that the wrong one will be followed." An example of a requirement stated in a vague and ambiguous way is "the system must have a comfortable interface" [FREEP77a]. Such requirements must be explicitly defined in terms of functions to be performed and data to be communicated. As Hosier [HOSIJ78] describes, currently the requirements to be satisfied by a software system are usually discovered during design upon informal and ambiguous discussions between the system users and the system developers, and specified using ambiguous natural language.

2. Premature Design Influence - Most of the time the requirements of a system are specified with respect to a design model that is the designer's own idea of the system. Some designers consider the first model that they conceive as the ultimate system model. This approach results in the elimination of many other system models which might very well be better than the chosen one.

4.2.2. The Need for Maintainability in DMS

There are two difficulties in the formation of DMS specifications:

1. DMS was defined above as being a system to be used in experimenting with and developing human-computer systems with human-engineered interfaces. Because of its research nature, it is unlikely that a

complete system specification can be formed at once. No matter how much analysis is done at the beginning, some of the specifications will appear during the course of research.

2. It is desired that DMS be a human-engineered system. How can a system embody the principles of human engineering, while it will be used to search for these principles?

Hence, maintainability, of prime importance in the development of any system, becomes of paramount importance in the development of DMS. We shall present DMS development methodology in section 6.

4.2.3. How to do Requirements Specifications

The objective of requirements analysis is to define the problem. One way of specifying the system requirements explicitly is to go directly into physical system architecture and define the system requirements in terms of module structures, physical devices, physical communication characteristics, etc. This approach combines the definition of what to do with the specification of how to do it, which are conceptually distinct activities. While the former is a logical activity, the latter is mapping the product of the former activity into a physical form as it will appear on the host system. This approach fails to take advantage of a divide and conquer policy, and the number of things which are to be considered simultaneously is increased. Also a system specification formed this way is of a more detailed, technical nature. As a result, the cooperation of the user (or of any person from

a different field) in the evaluation and verification of the specification is made difficult. The user is not interested in the physical architecture of the system. The user is interested in the functions that the system should perform, how it is going to be used, and the performance requirements that the system should satisfy. Specifications formed with this approach define only the computer part of an HCS, making human-factors analysis difficult. The place of the user is implied and the HFE must do the human-factors analysis concurrently with physical architecture development (e.g., how are functions allocated with this physical architecture; does this architecture represents a machine that matches user capabilities; and what is the logic sequence through this architecture?). There is a limit to the imagination of a human, while there is virtually no limit to the size and complexity of a system. Attempting to map the system requirements directly on the physical system specification is criticized by Ross [ROSSD77] as follows:

> "One fundamental weakness in current approaches to requirements definition is an inability to see clearly what the problem is. It is common practice to think of system architecture in terms of devices, languages, transmission links, and record formats. At the appropriate time in system development, this is quite proper. But as an initial basis for system thinking, it is premature and it BLOCKS from view precisely the key idea that is essential to successful requirements definition: The algorithmic nature of all systems."

Well known requirements analysis and definition techniques are: PSL/PSA [TEICD76], SADT [ROSSD77], and SSA [WEINV80]. Among these, SADT and SSA are suitable for HCS analysis, since both use a functional decomposition approach. In general, this approach relies on a division of the user requirements for an HCS into two classes : (a) functional requirements and (b) performance requirements. Functional requirements

are the functions to be performed by the HCS. Performance requirements are the constraints on these functions. The required functions and the algorithmic relationships between the functions can be specified as a functional flow diagram of abstract levels. The performance requirements can be noted on the function symbols of the FFD's. The data flow between the functions can be specified. The data structures and/or databases can be represented as logical entities. Data flow between the functions and the data stores can be shown. This logical representation of the system can be refined to a desired level of detail [WEINV80]. Understanding and evaluating this logical specification does not require technical knowledge and the users (as well as persons from other fields) can effectively cooperate in definition development and evaluation.

The output of the requirements analysis and specifications phase is the input for the design phase, which is discussed in the next section.

4.3. DESIGN

> "In no other industry would tooling-up precede completion of fundamental design. It is a special reflection on the history of the software development that this kind of postponing deserves special mention." [HOSIJ78]

4.3.1. Introduction

Even though the requirements analysis and definition phase of the software life-cycle is of critical importance, if this phase is skipped, one expects that there is at least going to be a design activity. Unfortunately, most programs are not even designed [MYERJ78].

4.3.2. What is Design?

Design is a process of moving from a "requirements definition" to the "specification of a mechanism or structure" which achieves that goal [FREEP77a]. Wasserman [WASSA77] defines the place of design in the software life-cycle as follows : "While the requirements analysis and definition activity is used to determine what is to be done in a software system, the design phase is used to determine how to do it." The design phase receives the requirements definition as input and converts this into a representation of the target system as it will appear on the host system. Hence, the nature of the activities during the design phase are determined by the nature of the requirements definition phase output. For example, at one extreme, the definition activity might produce only a list of requirements. In this case, both the logical specification of the requirements and the conversion of this specification into a physical representation of the target system have to be done in the design phase. On the other extreme, the specification phase might output a complete logical representation (e.g., logical design) of the target system. In this case, the design phase converts this logical speci-

fication into a physical representation. The latter case [ROSSD77] is

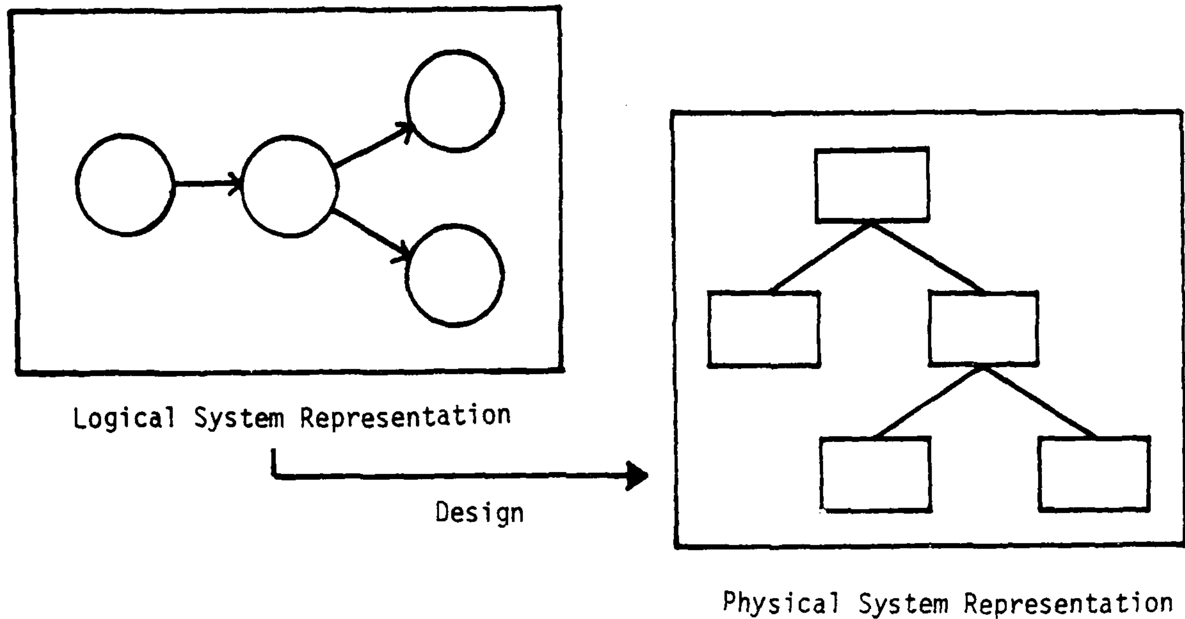


Figure 4.1. Logical and physical system representations.

shown in Figure 4.1. Although the specification and design phases can be integrated in small problems, it is better to treat these activities separately in large projects. This way, the activity of understanding the problem (logical specification) is not confused with the physical design activity.

In general, design can be divided into two distinct phases [WASSA80]: architectural design and detailed design. During architectural design, the issues of concern are the intermodule structure, the

communication interfaces among modules, and communication paths between modules and databases. In detailed design, the architectural design representation is refined by specifying the internal logic of the modules, detailed structure of the databases, etc. The output of the design activity is a representation of the system to be implemented. If this representation is to provide a comprehensible and unambiguous statement for the implementer, the architectural design must be refined with detailed design. A tool for the detailed design is structured English (e.g., PDL [CAINS75]). A design representation prepared in this form (a software blueprint) is also very helpful during the maintenance phase of the software life-cycle.

4.3.3. Why Design?

Freeman [FREEP77b] gives the following basic reasons for design activity :

1. To make correct structural decisions. As Freeman [FREEP77a] states, "In complex situations, the nature of the environment and the necessary structure is rarely obvious, and this structure has to be discovered through a careful thought process and design."

   If the necessary effort on the underlying structure of the problem is not spent, and if a wrong structural decision is made, all of the work done from that point on might be thrown away, once the real structure is discovered. The system structure must be compatible with the system requirements.

An example for this situation is the initial achitecture which was defined for DMS. One of the system requirements is the provision for concurrently running programs. The architecture was defined as

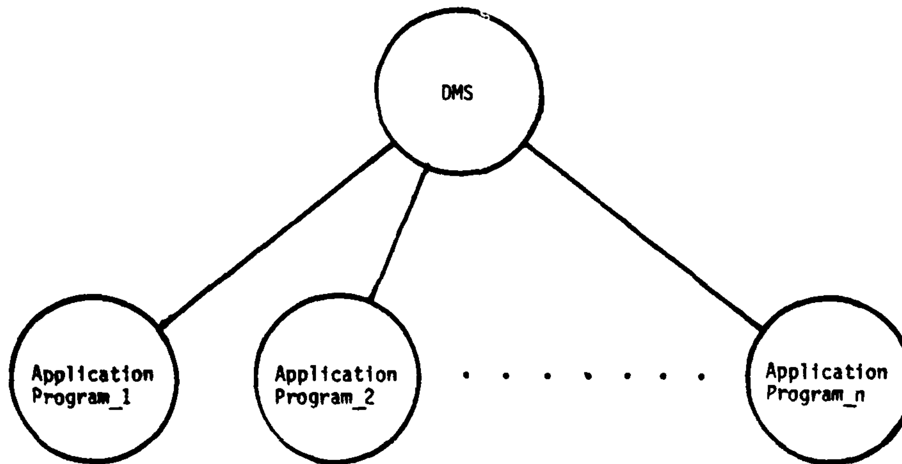


Figure 4.2. Initial architecture of DMS.

shown in Figure 4.2. Here, each node represents a concurrently running entity (called a process in many operating systems). The node for DMS contains all the supervisory and working modules for dialogue management and user interaction. DMS, as shown in the figure, can create application programs which run concurrently and can execute the user dialogue of these programs. At first sight, it looks as though the concurrency requirement is satisfied. But this architecture was the result of wrong interpretation of concurrency requirements. As shown in the figure, no two dialogue modules can run con-

currently because all dialogue modules are enclosed in the same process. Concurrency was needed to run human-factors experiments with regard to the response capabilities of users on freely-moving objects on the screen. Since all display privilege is given to dialogue modules, the objects had to wait until the user responded, instead of moving freely. Consequently, this architecture was replaced, allowing concurrent execution of dialogue modules as well as of computation modules.

2. To reduce development cost by managing complexity. In the development of complex systems there are many details with which the developers must deal. If this complexity is not controlled by the abstraction process of design, system development becomes very time consuming, and hence costly.

3. To reduce maintenance cost. The degree of maintainability of a system greatly affects the life-cycle cost. The systems cannot be made maintainable after they are finished. This is a built-in characteristic, which must be very carefully considered during design.

4. To have a system representation. The system representation is valuable both during development and during maintenance. Neither the developer nor the maintainer can be productive using a system representation of five hundred pages of code. The system representation helps the developers to keep track of what they have done and what they should do. The finished representation helps the maintainer in understanding the system structure, in finding the parts to be maintained, and in finding the relationships of those parts with the other parts. In addition, all during the system life-cycle, design

representation is a valuable communication tool [WASSA77]. Freeman [FREEP76] supports this point as : "The ability to communicate is clearly recognized as a key quality for software engineers".

4.4. THE DISTINCTION BETWEEN DESIGN AND CODING

Coding, once the major activity of software system development, now occupies about 20% of a system life [WASSA80]. It is useful to note the distinction between design and coding. The distinction is most clear when the outputs of the two activities are examined. The output of coding is information in machine executable form. The output of the design is a representation of this machine executable information. This representation can be at a very high level (e.g., a block diagram of concurrently running processes), can be at a modular architecture level (e.g., structure charts [YOURE79], HIPO [STAYJ76]), or at a level where the code to be produced is unambiguously defined (e.g., the logic of the modules defined by PDL).

The activities during system design are [FREEP77b,FREEP77c]:

1. Representing explicitly what is to be done by the software being designed.
2. Comparing alternatives and making tradeoffs between variables such as human-factors, maintainability, cost, etc.
3. Determining and describing the parts of the system.
4. Determining and specifying the data and control connections between the parts.

During coding, system representation is converted into machine executable form. Some coding activities are:

1. Creating the parts and the interfaces specified in design representation.
2. Declaring variables and data structures.
3. Devising and coding algorithms.

While there is not a clear boundary between requirements specification and design, there is an explicitly clear boundary between design and coding. The consequences of coding before finalizing design have been discussed. Since section 4 addresses the view of the software engineer, what follows are views of some software engineering authorities on design and coding :

Wasserman [WASSA77] :

> "One of the most difficult aspects of the programming discipline for the experienced programmer is to refrain from writing code at the early stages of software development. Most programmers have acquired bad habits when they first learned to program, often because they were not required to develop the necessary discipline in writing these programs. They were usually able to write them directly from the problem statement; they retain this tendency to write the code immediately even as they progress to more complex problems.
>
> While a great deal of innovation and creativity is possible at the design stage, the coding stage is more tightly constrained. The design must be completed to a level of detail that makes it possible to evaluate the quality of the design and to identify any remaining design problems.
>
> It should be noted that NO CODING SHOULD BE DONE TO THIS POINT. Since it is often necessary to redo the specification and design, any code written previously might very well be useless."

Brown [BROWR77] :

> "There is a story about IBM under OS where the choice was to take (in 1962) ten of their best people and do a serious design effort, or take 100 people who were waiting to program and turn them loose on programming. They concluded that they

could not afford to have the 100 people sitting around waiting for the designers to do the correct design. Unfortunately OS in its early versions reflected that."

4.5. MAINTENANCE

Maintenance occurs due to integration of requirements which were left unrecognized during the construction of the system, or due to changes in user requirements, i.e. the need for new functions, changes in performance requirements due to changes in the environment, or the improvement of the human-factors of the system. Experiences with systems developed show that the maintenance cost is the largest part of the software life-cycle cost. Hence, reduction of complexity is of prime importance in the development of large systems.

Myers [MYERJ78] lists the means of reducing the complexity during design and maintenance as follows:

1. Partitioning the system into parts having identifiable and understandable boundaries.
2. Representing the system as a hierarchy.
3. Maximizing the independence among the parts of the system.

4.5.1. Partitioning the System (Modularization)

Partitioning the system into its component parts is motivated by two major reasons: (a) the reduction of complexity and (b) simultaneous production of modules by a number of workers.

Constantine [STEVW74] describes the need for partitioning as follows:

"We design and build things out of discrete units for a number of reasons. The most important has to do with HUMAN PSYCHOLOGY. It is well known that our capacity to process information, especially simultaneous manipulation of details, is quite limited. Modularity is the name of the game in designing large complex systems."

Dijkstra [DIJKE76] states the same fact as:

"We have small heads and cannot think about many things simultaneously and, besides that, get tired and unreliable when we have to think about a very great number of things in succession. The way to avoid these situations with which we can hardly cope has been captured more or less by many phrases : The exploitation of one's powers of abstraction, divide and rule, the judicious postponement of commitments, the separation of concerns etc."

As a result, one objective of partitioning is to reduce the number of factors that a human mind has to keep track of simultaneously. Another objective as Myers [MYERJ78] describes is as follows:

"Partitioning a program creates a number of well defined, documented boundaries within the program. If we are interested in analyzing a particular datum, the interfaces show us where the datum is and is not used, thus narrowing our focus of attention."

Partitioning (or modularization) by itself is not necessarily effective. As Liskov [LISKB72] states, arbitrary partitioning can actually increase the complexity and intermodule connection complexity which might be generated if

1. a module performs too many related (and yet different) functions which cause its logic to be obscured,
2. common functions are distributed throughout many modules rather than being isolated in one place, or
3. modules interact on common (global) data in unexpected ways.

Hosier [HOSIJ78] states some criteria for modularization :

1. Module size.

2. The information hiding capability of a module.

3. Module strength.

4. Module coupling.

Module Size - As mentioned by Dijkstra [DIJKE76] and Constantine [STEVW74], the smaller the modules, the more managable they are. But if the size of a module is reduced beyond a point, the module's function might not be clearly defined, which distorts understanding of the module. Also, as functional meaning diminishes, interconnections with the other modules increase, making the control structure complex.

Information Hiding - This point is especially emphasized by Parnas [PARND72]. According to Parnas, a module is associated with a design decision (e.g., implementation details of an algorithm) which the module hides from higher level modules. Modules should be defined around the decisions that are likely to change. Such modules will have interfaces that are unlikely to change and modifications to the system will be confined to one module.

Cohesion and Coupling - The reason behind applying the coupling and cohesion criterion is to develop modules which are functionally independent and have minimum intermodule connections. This criterion is extremely valuable in producing maintainable programs. (Cohesion and Coupling are discussed in detail in the Appendix.)

4.5.2. Inter-module Organization

It was mentioned above that the average human mind has a limit on the number of facts with which it can simultaneously deal. Hierarchical organization is a powerful tool both for construction and graphical representation of program structure. As Myers [MYERJ78] states: "Hierarchical views aid our understanding by directing our span of attention and allowing us to cope with the system at various levels of detail."

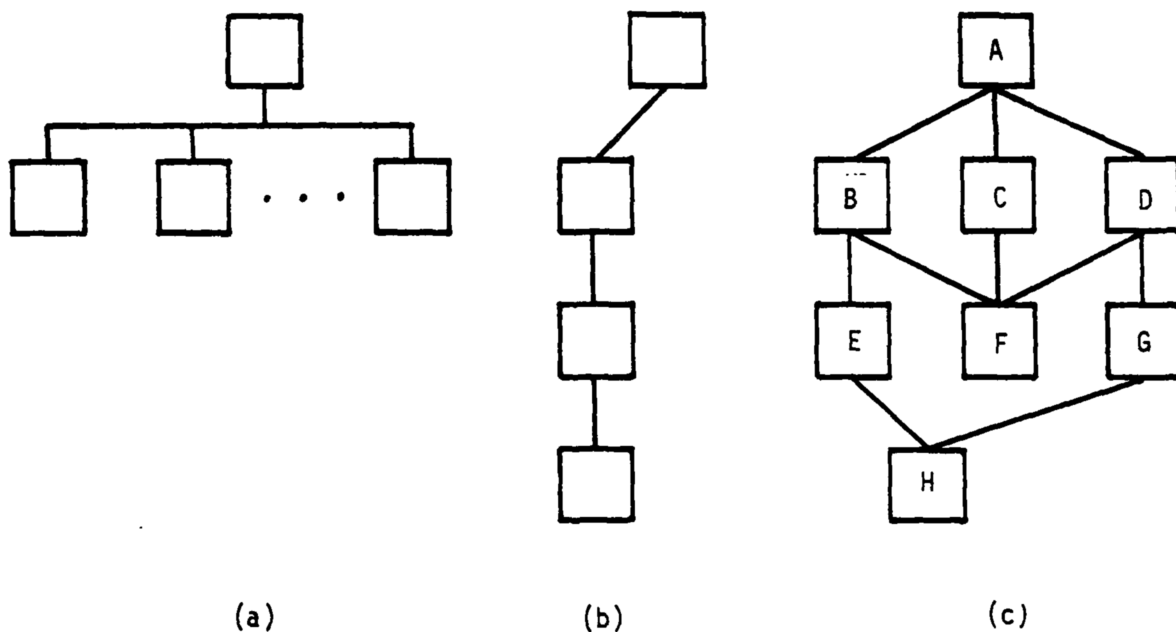


Figure 4.3. Bad [(a) and (b)] and good (c) hierarchical organizations.

In Figure 4.3, three types of hierarchical organizations are shown. Organizations (a) and (b) are undesirable. In (a), the top module is dealing with a large number of functions and concepts, and it is implied that it has a complex logic. In (b), most probably all the work is done by the bottom-most module and the superordinate modules probably do not have enough responsibility to become a module. In (c), neither of the above extreme cases is observed and this is the desired organization.

4.5.3. Functionally Independent Partitions (Modules)

Partitioning a program into a hierarchy does not considerably reduce the complexity of the program unless the modules are functionally independent. For example, consider the hierarchical modular organization (c) of Figure 4.3. Assume that part of the function of module B is included in module C. Hence, modules B and C are connected to each other. Module A must have a control mechanism, to perform the function that is split between B and C. On the other hand, if module B is independent of module C, there is no need for a control mechanism in module A. To perform the function, module A only issues a call to module B.

As a result of the above discussion, for the complexity to be reduced, the goal should be to partition a program into a hierarchy such that each module is as independent from all other modules as possible. This last sentence outlines the Structured Design methodology. In the following, we will present a simple example to show the essence of the Structured Design methodology, and to make clear the distinction between structured programming and Structured Design.

4.6. A DESIGN EXAMPLE

The ultimate objective of Structured Design is to obtain a modular structure in which the modules are of maximal strength (single function), and minimally coupled (only data communication). Remember that this objective might not be fully achieved in real life due to other trade-offs. But if a program is designed in such a way, both the understandability and maintainability are achieved to a maximum extent. Especially, the addition of new functional requirements becomes simple, as we shall see in the following example.

The problem is to design a program which will take a trigonometric function (e.g., Sine, Cosine) and an angle value (e.g., 2.3 radians) and compute the value of the function (e.g., Sin(2.3)). The reason for choosing this problem is that we want to show the advantages of single function, minimally connected modules.

The following are the hypothetical events that happen along the life of the program:

Step 1. Let us assume that the analyst has a very poor knowledge of trigonometry. The analyst knows only about the Sine and Cosine functions . So the initial, incomplete requirements specification is that the program must contain the Sine and Cosine functions. But, the analyst has some doubts about whether this requirements specification is complete (just as we feel about DMS). So the analyst designs the modular structure as in the structure chart shown in Figure 4.4. (Note that we shall show only that part of the program which will be affected by the changes.) Modules "Choose function", "Compute Sine" and "Compute

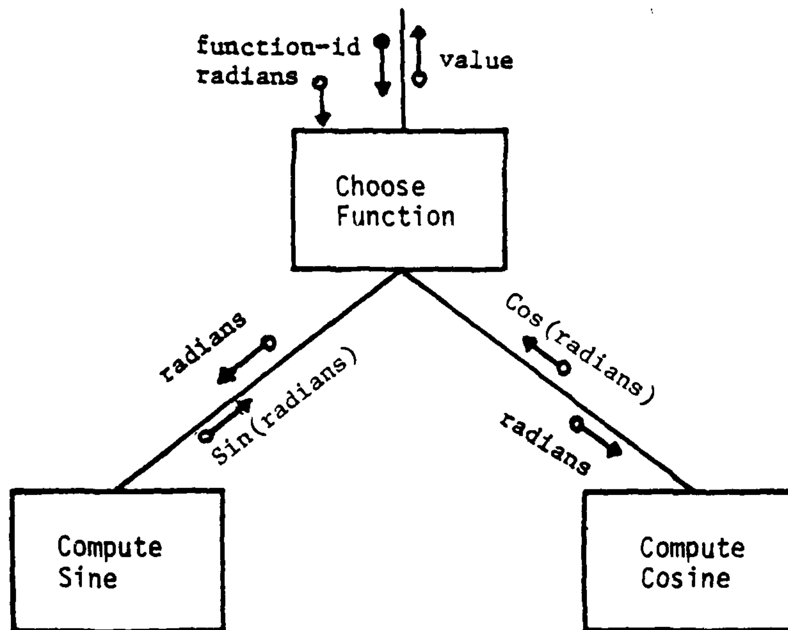


Figure 4.4. Structure chart for Sine and Cosine.

Cosine" are all separately compiled modules. "Choose Function" calls one of its subordinates (according to the function-id it gets) and passes the "radians". The called module computes and returns the value of the function.

Step 2. Soon, the analyst realizes that the Tangent function is missing. The analyst discovers that Tan(x) = Sin(x) / Cos(x) and adds the Tangent function as shown in the structure chart in Figure 4.5. "Compute Tangent" first calls "Compute Sine", then calls "Compute Cosine", computes Sine/Cosine and returns the result to "Choose Function".

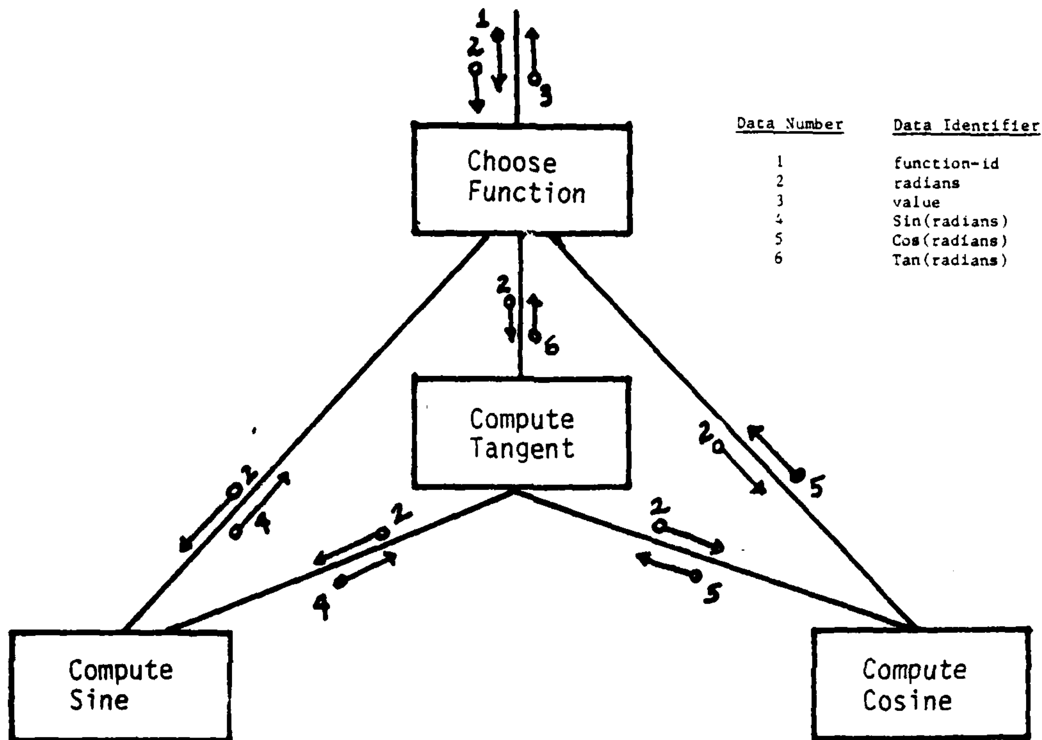


Figure 4.5. Structure chart for Sine, Cosine, and Tangent.

Step 3. Next, the analyst realizes that the Cotangent function is missing. The analyst adds the Cotangent module as shown in the structure chart in Figure 4.6. "Choose Function" calls "Compute Cotangent", "Compute Cotangent" calls "Compute Sine" and "Compute Cosine", computes the value and returns it to its caller.

Step 4. The users complain about response time. The programmer goes into the Sine module, and replaces the algorithm with a better one. As a result, Sine, Tangent, and Cotangent all give faster response.

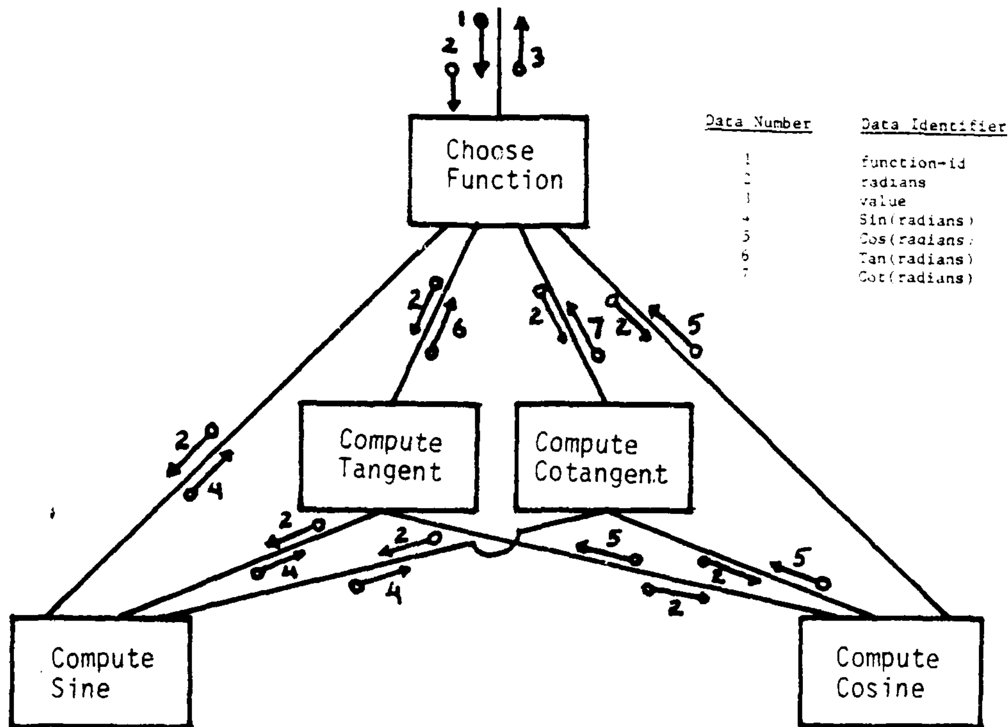


Figure 4.6. Structure chart for Sine, Cosine, Tangent, and Cotangent.

Step 5. Users complain about the response time of Cosine although they are happy with the rest. Again, the programmer goes into the Cosine and improves the algorithm. Cosine works faster while Tangent and Cotangent are even faster than before.

Step 6. Somebody wants the result for Tan(45 deg). Instead of "1", the analyst gets "2.5". There is a bug in the system. The maintenance programmer looks in the Tangent module but cannot find anything. The programmer then tests "Compute Sine" and gets correct answers. The pro-

grammer tests "Compute Cosine" and determines that the bug is in the Cosine module. Without ripple effects, the bug can be fixed.

To see the difference between Structured Design methodology and structured programming consider a few steps of the above example using

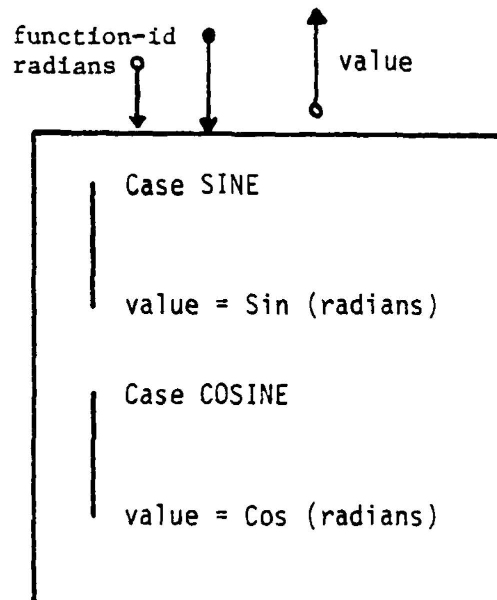


Figure 4.7. Structured code for Sine and Cosine.

structured programming, but not Structured Design.

Step 1. The analyst thinks that both Sine and Cosine are small enough that they can be combined in a single module (remember that Structured Design calls for one function per module). But the analyst has the discipline of structured programming and constructs a block-structured module, shown in Figure 4.7. The module gets the function-id as a function switch. Note that the arrow with the closed circular tail indicates that it is a function switch. (Structured Design methodology

requires that these switches must be removed by breaking the module into its component functions. Otherwise, as is the case here, the module is "control coupled" with its caller.)

Step 2. The analyst realizes that Tangent function is missing. This new functional requirement is not as easy to accomodate as it was before. The analyst must now study the internal logic of the module, whereas the structure chart was sufficient in the previous example. The analyst fixes the problem, as shown in Figure 4.8, by incorporating the

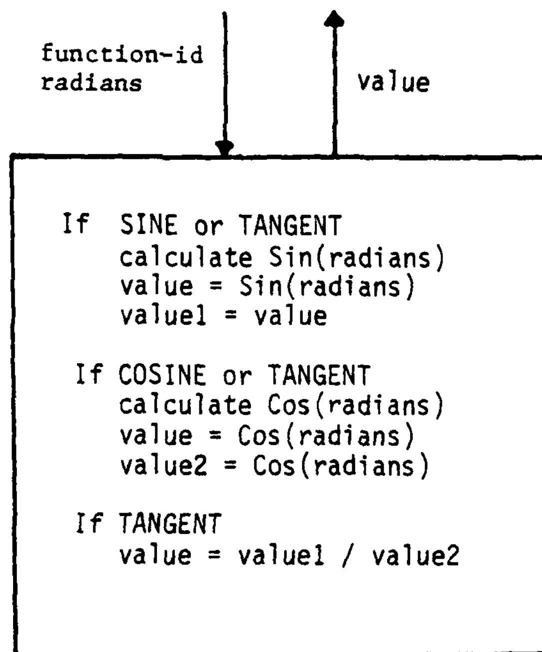


Figure 4.8. Structured code for Sine, Cosine and Tangent.

following sequence:

If the function is Sine, value=Sin(radians). If the function is Tangent, save the value of Sin(radians) in value1.

If the function is Cosine, value=Cos(radians). If the function is tangent, save the value of Cos(radians) in value2.

If the function is Tangent, value=value1/value2.

Compare the transition from Figure 4.7 to Figure 4.8 with the transition from Figure 4.4 to 4.5. Compare all the extra effort spent and the complexity introduced into this module. Obviously, things will become worse in the rest of the steps mentioned above. The reader can imagine how much more complexity will be built into this module to add the Cotangent function, and how difficult it will become to understand and to modify this module for new requirements.

The application area for the example above is a very simple one. The advantages of Structured Design, for a program composed of a few hundred modules instead of the four or five modules discussed above, is clear.

5. HCS METHODOLOGY

The logical representation of DMS was shown earlier in Figure 2.6. The human-factors engineer (roles: human-factors analysis, experimentation, dialogue authoring) and the software engineer (roles: system analysis, design, programming) work together. The system analysis, design, implementation, and maintenance are made by cooperation of these two experts and the various related roles, through automated support of the system. Note that if there is no automated support, then there is no DMS and the two experts are constructing a human-engineered HCS using HCS methodology but with conventional tools (e.g., the host operating system interface). This latter case was shown in Figure 2.5.

Our objective is to analyze the manual case shown in Figure 2.5 and to produce the automated system shown in Figure 2.6. The first step in doing this is to develop a methodology which can be used by the two experts towards the common goal of a human-engineered HCS. The key requirements for the methodology are the following:

1. It must allow the two experts to communicate with each other.
2. It must have tools to serve both experts in their own special fields.

In short, the methodology must integrate human-factors engineering and software engineering in a human-factored way.

5.1. CONSTRUCTING THE METHODOLOGY

Our objective in this section is to bring the HFE's view (in section 3) and the SWE's view (in section 4) together and to briefly evaluate the current software development methodologies in view of the requirements mentioned above. We shall borrow the useful techniques from these methodologies to construct the fundamentals of an HCS developmental methodology. The weaknesses and deficiencies of the methodology will be augmented or replaced during the course of research.

The four most widely used methodologies [HOSIJ78] are given below:

1. The Michael Jackson Methodology [JACKM75]
2. The Warnier-Orr Methodolgy [HOSIJ78]
3. SADT (Structured Analysis and Design Technique) [ROSSD77]
4. SSA (Structured System Analysis) and Structured Design [STEVW74, YOURE79, MYERJ75, MYERJ78, WEINV8Ø, STEVW81].

All the mentioned methodologies use top-down development strategy. The first two methodologies above (Michael Jackson and Warnier-Orr) use data as the basis of design and both derive the program structure from data structures. Since the HFE is basically concerned with the functional flow of a system rather than the data structures used to derive the program structure, the first two methodologies would not help the HFE's analysis.

The other two methodologies (SADT, SSA and Structured Design) use the process as the basis of design and build the design around the functions that the final system is to provide. Although SADT is a powerful methodology in requirements specifications, as Wasserman [WASSA8Ø]

states, "The modules built in SADT often do not lend themselves to immediate transformations into design or implementations. It takes considerable training and experience to become a skilled SADT user."

Thus, SSA and Structured Design is the methodology that most suits the needs of the HFE, while concepts of SADT (e.g., author-reader cycle through levels of functional decomposition) can be used in requirements definition. SSA uses data flow diagrams (DFD) as part of the logical architecture representation. The data flow diagrams [WEINV80] are the functional flow diagrams which the HFE needs in modelling user-computer interaction. Other major tools are the logical data structure diagrams (DSD) and a data dictionary (DD). A logical DSD hides the implementation details of a data structure (e.g., a linked list is represented as an array of list elements), and the DD contains information that explicitly defines the abstract objects which appear on system representation (e.g., definition of an abstract data type, PDL representation of a function's logic, etc.). Data flow diagrams are then used by the Structured Design methodology, which provides a very high degree of maintainability. (The Structured Design methodology is discussed in a larger detail, in the Appendix.)

As was mentioned in the introductory part of this report, our view of a system is an HCS, as opposed to the traditional view which considers only part of an HCS (e.g., the computer part) as a system. The methodology that we shall present considers both humans and software as functional parts of the same system. We also divide the software part of an HCS into two components (e.g., dialogue and computation), and so a target HCS has three functional elements: human, dialogue, and computation.

There are basically two classes of roles in HCS analysis (i.e., HFE and SWE). A conventional DFD (which is also called a bubble chart) has a circular node for representing the functions. We are augmenting the DFD notation by introducing analysis symbols, to explicitly identify the target system parts (human, dialogue, and computation) which fall in the activity domain of each class mentioned above. (With the user function descriptors, the DFD subsumes the function of "Operational Sequence Diagrams", a tool used in man-machine system analysis.) The fundamental symbols of the structured HCS analysis diagrams, or user-dialogue-computation (UDC) diagrams, are shown in Figure 5.1. In the following, the semantics of the symbols are briefly discussed:

a. General function - Used in requirements definition phase, in constructing the levels of a system functional flow diagram, prior to function allocation activity. As functions are allocated, each of these function symbols is replaced by one of the other function symbols.

b. User function - Represents a function allocated to the user (e.g., manual task).

c. Dialogue function - Represents a software function which provides the communication between the user and the computer. The contents will be written by the dialogue author and will communicate with the user in a human-factored way.

d. Computational function - Represents a software function which will perform only a computation. The function must communicate with the user only indirectly through a dialogue function. The dialogue function is the virtual user for the computational function. The dia-

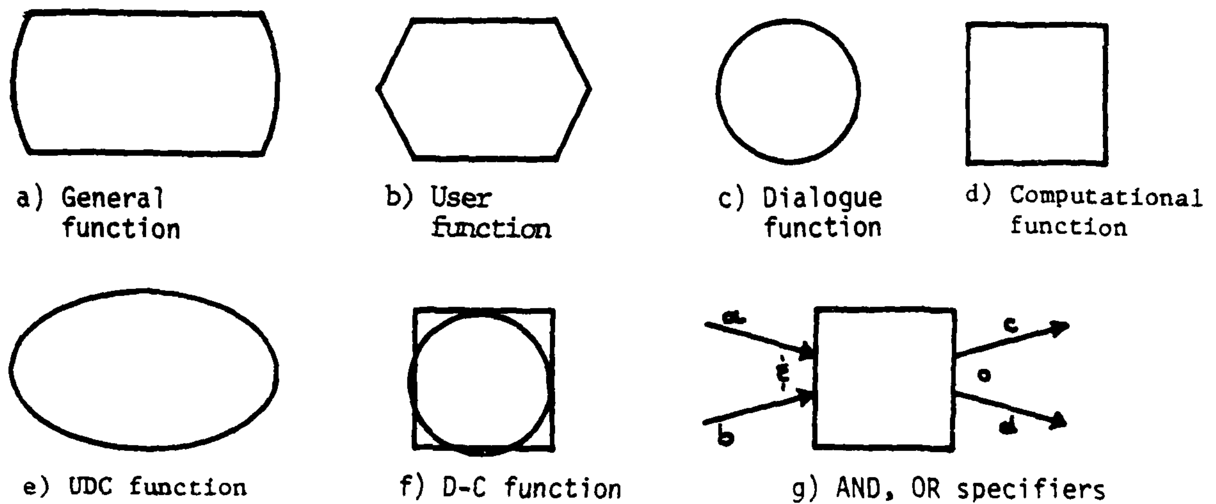


Figure 5.1. Symbols of an HCS flow diagram.

logue function serves as the communicator between the user and the computational function. The computational function is written by the programmer.

e. User-Dialogue-Computation (UDC) function - A high level HCS function composed of user, dialogue, and computational functions. The function is to be decomposed into its components for further analysis.

f. Dialogue-Computation (D-C) function - A high level software function composed of both dialogue and computational functions. The function is to be decomposed into its components for further analysis.

g. Logical "and", logical "or" specifiers : "&" and "o", respectively - In Figure 5.1.g, the function "D" gets both data "a" and "b" and sends out either data "c" or "d".

The HCS development methodology is shown as a functional flow diagram in Figure 5.2. (Note that the phases are represented by "gen-

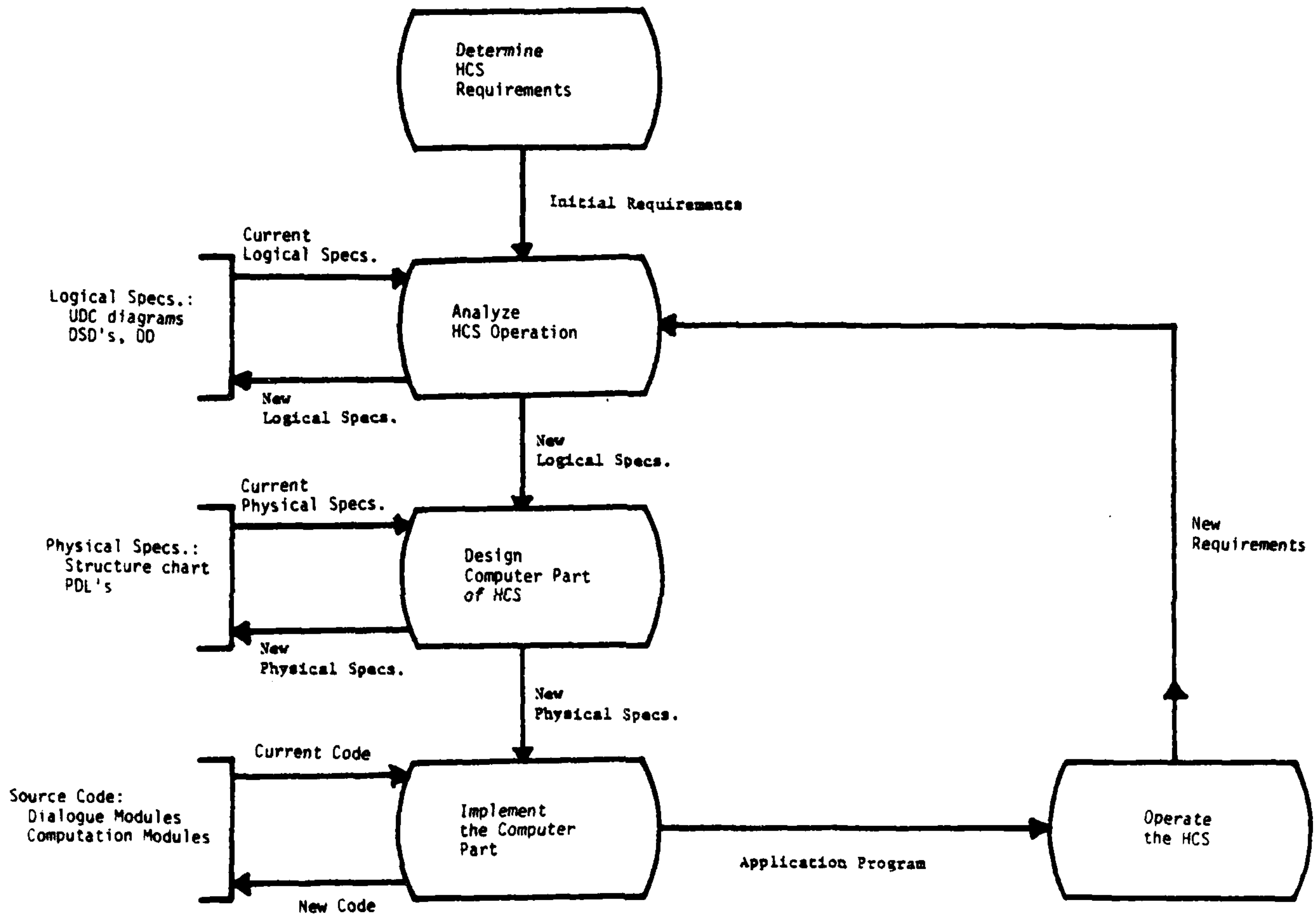


Figure 5.2. HCS Development Methodology.

eral function" symbols.) In the following, we shall discuss the phases of the methodology.

5.1.1. Requirements Definition Phase

HCS requirements are defined by the cooperative work of the HFE, SWE, and the user (customer), through abstract levels of functional flow diagrams. The functional requirements are derived from system goals and the performance requirements are inferred from environmental conditions. The requirements definition does not differentiate between human and computer functions in this phase (unless allocation of some functions are already decided by the customer). This initial requirements definition is passed to the analysis phase.

5.1.2. Analysis Phase

INPUT:

1. Initial Requirements.
2. New Requirements. These are the requirements that arise during the use of the system, either for improving the current system or modifying the system as a result of environmental changes in which an HCS operates.
3. Logical Specifications. Initial logical specifications are formed during the first cycle through the analysis phase. The logical specifications are then fed to the analysis phase during maintenance cycles.

PROCESS:

The HFE and the SWE work on the FFD formed in the requirements definition phase. The objective of this phase is to decide on a func-

tion allocation configuration and to develop the logical design of the software. The analysts obtain alternative function allocation configurations which satisfy the performance requirements. These alternative configurations are refined to lower level functions and are evaluated with respect to human-factors and computer characteristics. The most cost-effective configuration is selected for design. The UDC diagram explicitly shows the user functions as well as the computer functions. The diagram is refined to specify a complete logical separation of the dialogue and computational functions as well as the data communication between these functions. The result is a UDC diagram, picturing user functions, dialogue functions, computational functions and the communication between these functions.

Including the user functions in the HCS representation has several advantages:

1. The model can easily be understood by the user, since the user can explicitly see where the user fits into the system. Hence, communication problems with the customer during specification verification are reduced and the model can be used as a teaching aid for the human part of the HCS.
2. The HFE can explicitly see where the user is along the logical paths of the system, can examine the effects of the logical flow on the user (e.g., too much load on user memory), can determine what sort of feedback should be provided at dialogue functions (e.g., reminding the user about what has been done so far or warning the user about the future consequences of actions), etc.

3. The user nodes are useful for the experimenter, in keeping statistical data such as average delay, error percentage, etc. (If the data are overcrowded, the nodes can be used as pointers to these data sets.) With this, the bottlenecks can be determined and located within the program logic.
4. Type of communication between user and dialogue functions (e.g., voice I/O, CRT display, touch panel, etc.) can be explicitly represented on the data paths, with appropriate symbols.
5. A UDC diagram shows the current function allocation status of an HCS. Environmental changes or human-factors may require the modification of this configuration, later in the HCS life. Hence, a UDC is a valuable HCS representation in maintenance cycles.
6. User nodes facilitate explicit modelling of an HCS with multiple users working in parallel.
7. A UDC diagram can be used for representing the overall operational picture of an organization and computerized activities can be explicitly seen. Upon changes in the organization (e.g., addition of new departments), parts of the computer system affected by the changes can be readily observed.

An example of a UDC function is shown in Figure 5.3. The elliptical node indicates that user, dialogue, and computational functions are involved in performing the function. The UDC node computes the value of a trigonometric function (e.g., Sine, Cosine, Tangent, or Cotangent) and uses the computed value.

In Figure 5.4, the user and computer components of Figure 5.3 are illustrated. A menu of functions is displayed to the user by a dialogue

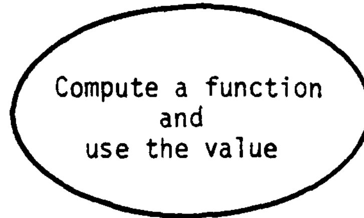


Figure 5.3. A UDC Function.

function. The user enters the function-id, and the angle-value in radians. The D-C function (shown by an overlapped circle-square pair) computes the function and returns its value to the user. The D-C node indicates that this function contains both dialogue and computation but has yet to be broken into its dialogue and computation functions. During discussions with the user (customer) for system verification, the expansion of D-C functions should be avoided, which would otherwise expose the user to unnecessary detail.

The expanded form of Figure 5.4 is shown in Figure 5.5. The dialogue function "get user input" gets user inputs and passes them to "compute function". The computed function is passed to "display value" where it is displayed to the user. The HFE and SWE study this logical system representation and make additional requests if what they see does not satisfy them. For example, look at the data flow between "compute

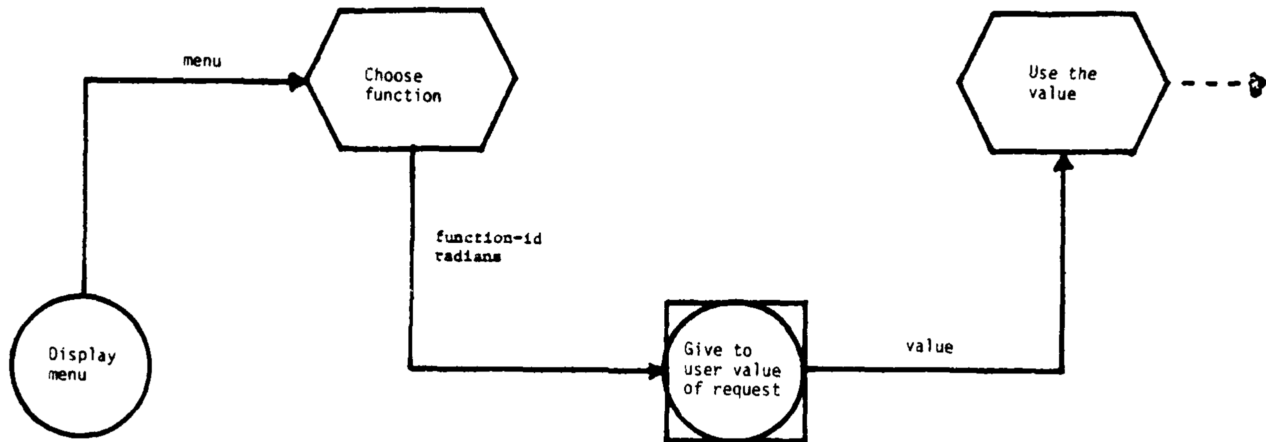


Figure 5.4. User and Computer Parts of Figure 5.3.

function" and "display value". The only item shown is "value" of the function. The HFE might prefer to display the function value next to its function identifier (e.g., Sin(0) = 0, instead of just a zero). But the HFE cannot do it unless "compute function" gives the function-id to "display value".

Note that "compute function" is a computational type node and it is shown by dashed lines. The dashed lines indicate that it has to be further expanded to its components. The semantics of the dashed lines are identical to the nonterminals of the Backus-Naur Form. It is particularly useful to leave a dialogue or a computational function in this form during the course of cooperative discussion. Neither of the parties would be interested in the expanded form of a function which does not concern them. It is better to make such expansions separately. But

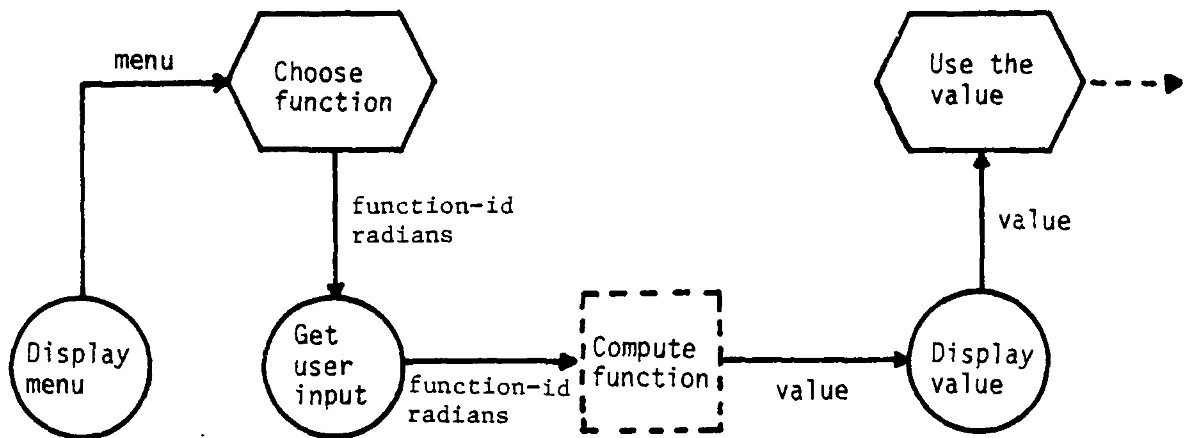


Figure 5.5. Refined Form of Figure 5.4.

once the diagrams are completed, it might be useful to bring them back into a compact form, as long as they do not become too crowded.

The completed diagram, with expanded form of "compute function" is shown in Figure 5.6. Note that only one of the functions (Sine, Cosine, Tangent, Cotangent) will be executed and the "o" symbol indicates a logical OR relationship between data paths.

OUTPUT:

Outputs of the analysis phase are the UDC diagrams, logical data structure diagrams, and a data dictionary. There are two output paths. One goes to logical specification storage (for reference in maintenance cycles) and the other goes to the design phase.

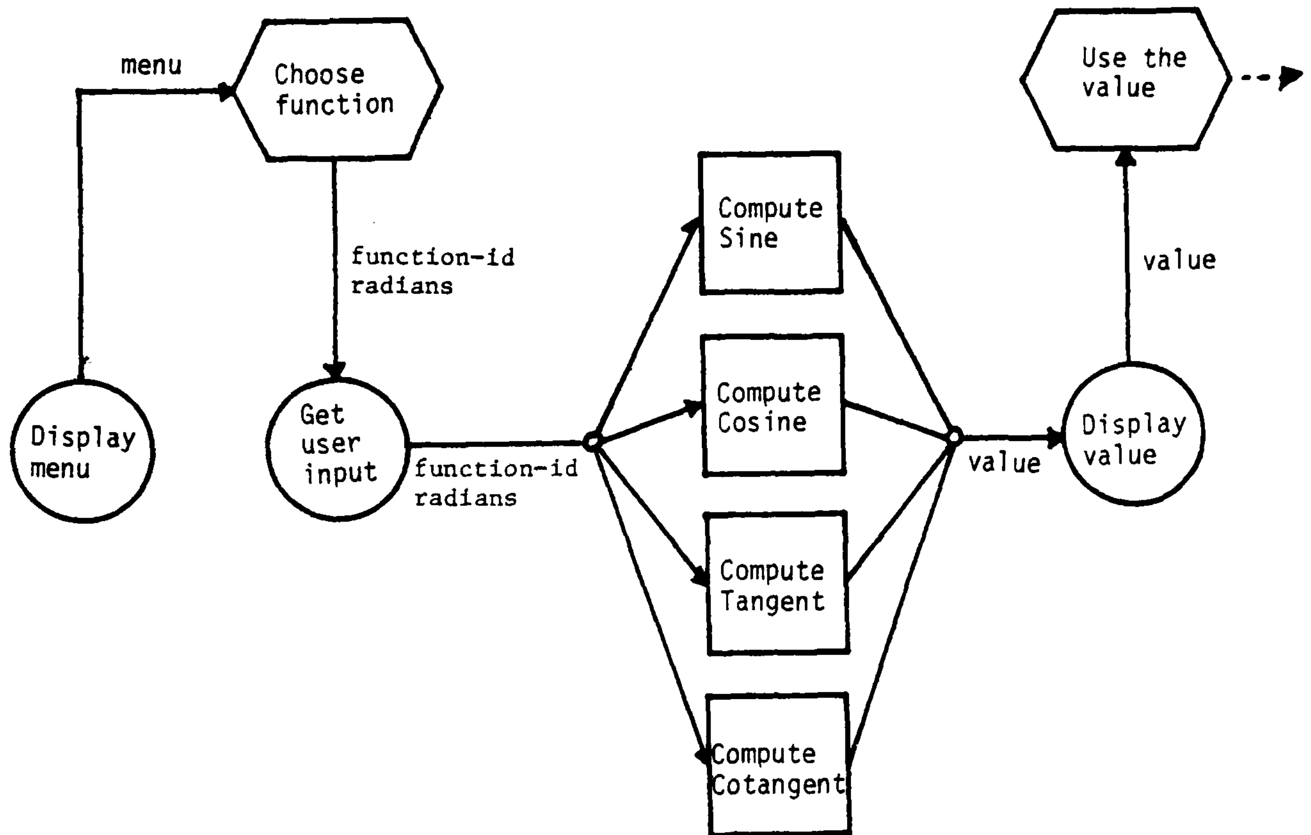


Figure 5.6. Expanded Form of "Compute Function".

5.1.3. Design Phase

INPUT:

The logical specifications produced in the analysis phase are input to the design phase.

PROCESS:

In the design phase the logical description of the HCS is converted into the physical specification of the computer part. The user func-

tions are cleared and the UDC is converted into a "structure chart" by the HFE and the SWE. The HFE and the SWE both work on the same structure chart, which shows what dialogue and program modules call what other dialogue and program modules. The design is refined by coupling and cohesion considerations and this process may require several immediate iterations between the analysis and design phases. The internal design of the dialogue modules and the computational modules then can be done by the HFE and the SWE, respectively. The use of a pseudo code (e.g., PDL) is helpful at this stage both for communication and for hiding the details of the implementation language until the design is complete.

In the above section, after several layers of refinement, we obtained a UDC which explicitly shows the user, dialogue, and computational functions. In Figure 5.7, the structure chart corresponding to Figure 5.6 is shown. The structure chart shows the hierarchy of modules and the communication interfaces. While a UDC represents the functional layout of the HCS, the structure chart represents the architecture of the computer part of the HCS. Earlier, in section 4, it was mentioned that mapping the system requirements onto the physical architecture results in communication problems. Obviously, the structure chart is not as readable as the UDC because it does not show the function sequencing. Although the convention is to arrange the modules in their execution order from left to right, this is not always possible. To see the logic sequencing, the analysts have to go through the internal logic of the modules.

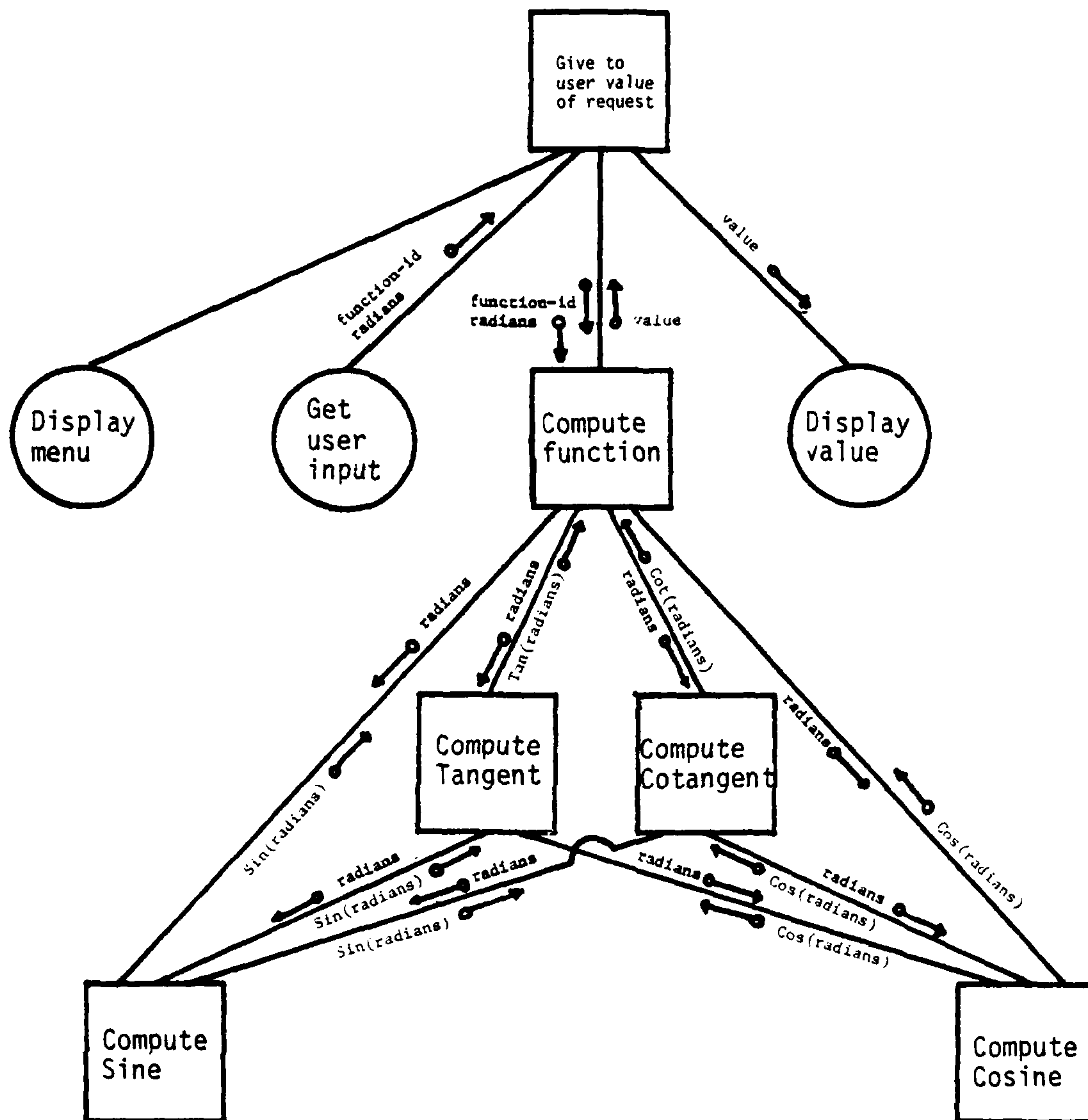


Figure 5.7. Structure Chart for the UDC in Figure 5.6.

Note that the modules in the structure chart are single function modules. The major advantage of this approach is the ease of maintenance and sharability of the modules. Especially in an experimental environment this maintainability is of vital importance.

OUTPUT:

Output of the design phase is physical system representation in the form of structure charts, data dictionary entries, and PDL's.

5.1.4. Implementation Phase

INPUT:

The physical representation of the system produced in the design phase is the input for the implementation phase.

PROCESS:

Once the structural design and the detailed design of the modules are finished, the top-down implementation and testing of the modules are carried on. The structure chart showing the modular structure and the PDL's specifying the procedures in the modules are used to obtain the final source code for the system under construction. The dialogue modules are written by the dialogue author and the computational modules by the programmer.

OUTPUT:

1. Dialogue and Computation Source Modules - Source code of the dialogue and computation modules is stored as documentation.
2. Finished System - Output of the implementation phase is a ready-to-use system.

5.1.5. HCS Operation Phase

The operation of an HCS shows whether the system developmental efforts have been successful. Deficiencies result in maintenance. Although the testing and debugging of the software is done before HCS operation (e.g., in implementation phase), the deficiencies in human-factors can most effectively be discovered and/or tested when the software is used by its intended users (i.e., during HCS operation). In an experimental environment, the HFE keeps data during HCS operation. The UDC diagrams are the tools for recording data on user behavior. In Figure 5.2, all cycles following the first one are called maintenance. Maintenance can be classified as local changes and global changes.

Local changes are those changes which can be confined to the procedural body of the modules. For example, changing the contents of a dialogue module is of this type; the dialogue author can alter the contents of a dialogue module (e.g., changing the format of a display). These activities of the dialogue author do not concern the programmer at all, as long as the changes attempted by the dialogue author are internal to the dialogue modules. Similarly, the programmer can change the contents of the computational modules, for instance to optimize an algorithm. Hence, separation of the user-computer dialogue from the computational part of the software enables the independent manipulation of these parts. When such changes interior to a module are made, it is important to update the PDL representation of the module logic, to keep a consistent documentation of the system.

Global changes are all changes other than these local changes; for example changes in the modular structure (e.g., changing the logical sequence or introducing a new functional requirement) or changes in the intermodule communications. Such changes may involve only dialogue structures, only computational structures, or both. In case of changes in structures of just one type, the respective specialist (dialogue author or the programmer) makes the necessary changes and updates the documentation. In any case, the changes should be checked against the overall system architecture to make sure that such changes are only dialogue or only computational structural-changes without affecting each other. If they affect each other, the two specialists have to work in cooperation. Examples of such cooperative work include the addition of new requirements which need both dialogue and computational parts, or changes in dialogue modules which require more information from the computational part.

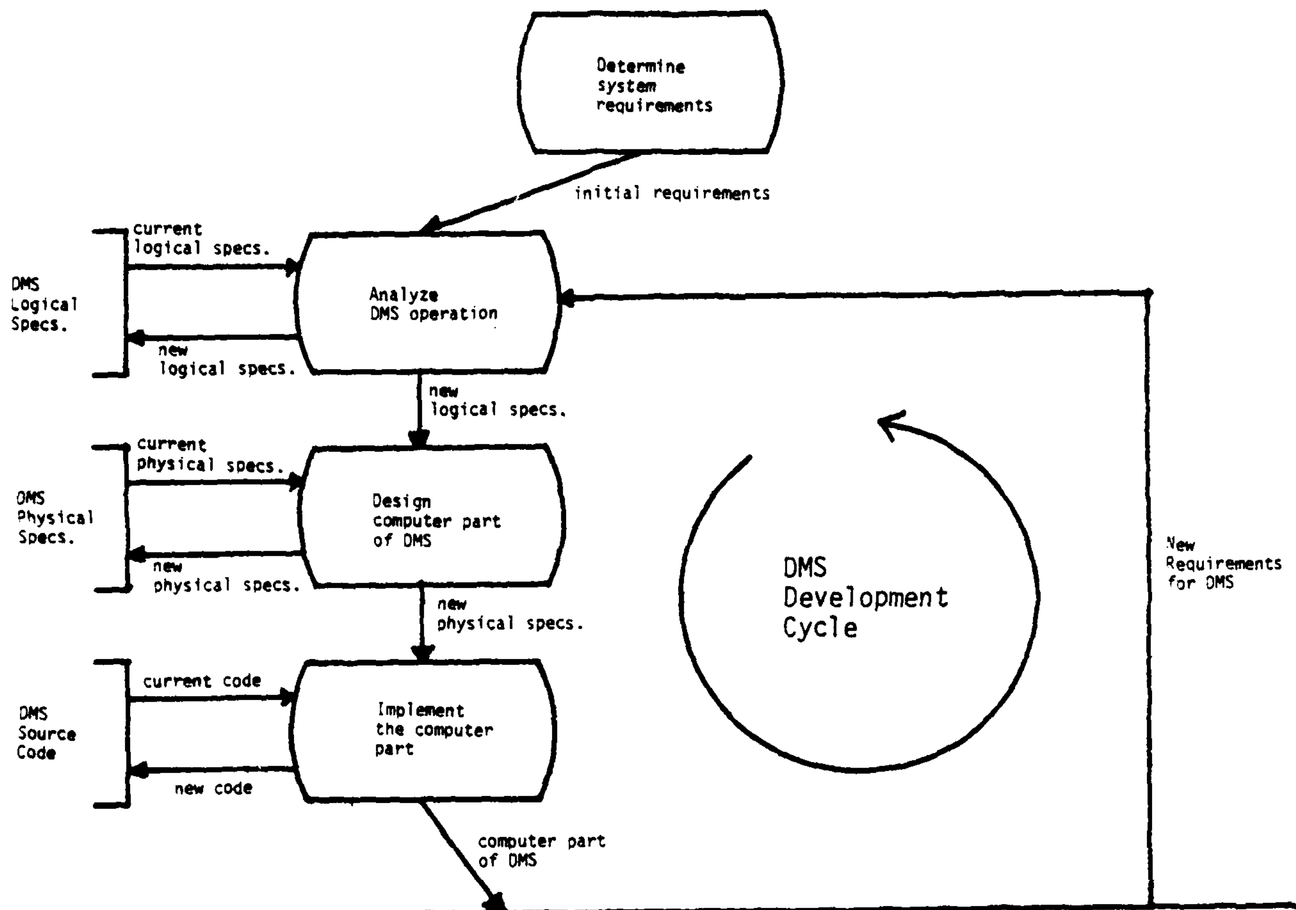
As an example of cooperative maintenance, look at Figure 5.6. The user is required to input the angle-value in radians. This means that if the user has the angle-value in "degrees" (which is often the case), the conversion of "degrees" to "radians" has to be done manually. To remove this deficiency, a computational function which makes the conversion has to be added. Also, the dialogue function "get user input" must be changed to make the distinction between the angle-value type.

6. METHODOLOGY FOR THE DEVELOPMENT OF DMS

The methodology used by the DMS project team will be discussed next. An important part of DMS will be a computer-based environment in which the HFE and SWE work together using automated tools to support HCS developmental methodology. Thus, DMS is itself to be a human-engineered HCS and needs the same methodology applied to its development. The diagram of Figure 5.2, illustrating the phases of the methodology is the input FFD for the DMS analysis.

Figure 6.1 shows the DMS developmental methodology. The upper portion shows DMS development, and the lower portion shows the operational phase of DMS, where the DMS-supported methodological tools are used in the development of another HCS. System development starts with constructing an initial system by going through the analysis, design, and implementation phases of DMS. Then, the tools provided by DMS are tested in application system development, using the same HCS-generation methodology. If the tools provided by DMS are not powerful enough or not well enough human-engineered (discovered during system operation), these deficiencies are recycled back to the DMS analysis phase for improvement of DMS.

A Generic Environment for Interactive Experiments (GENIE) has been constructed for human-factors experimentation by the one CS group of the project. The needs of the human-factors group as they interact with the CS group are being analyzed to be fed to the DMS design phase of Figure 6.1.



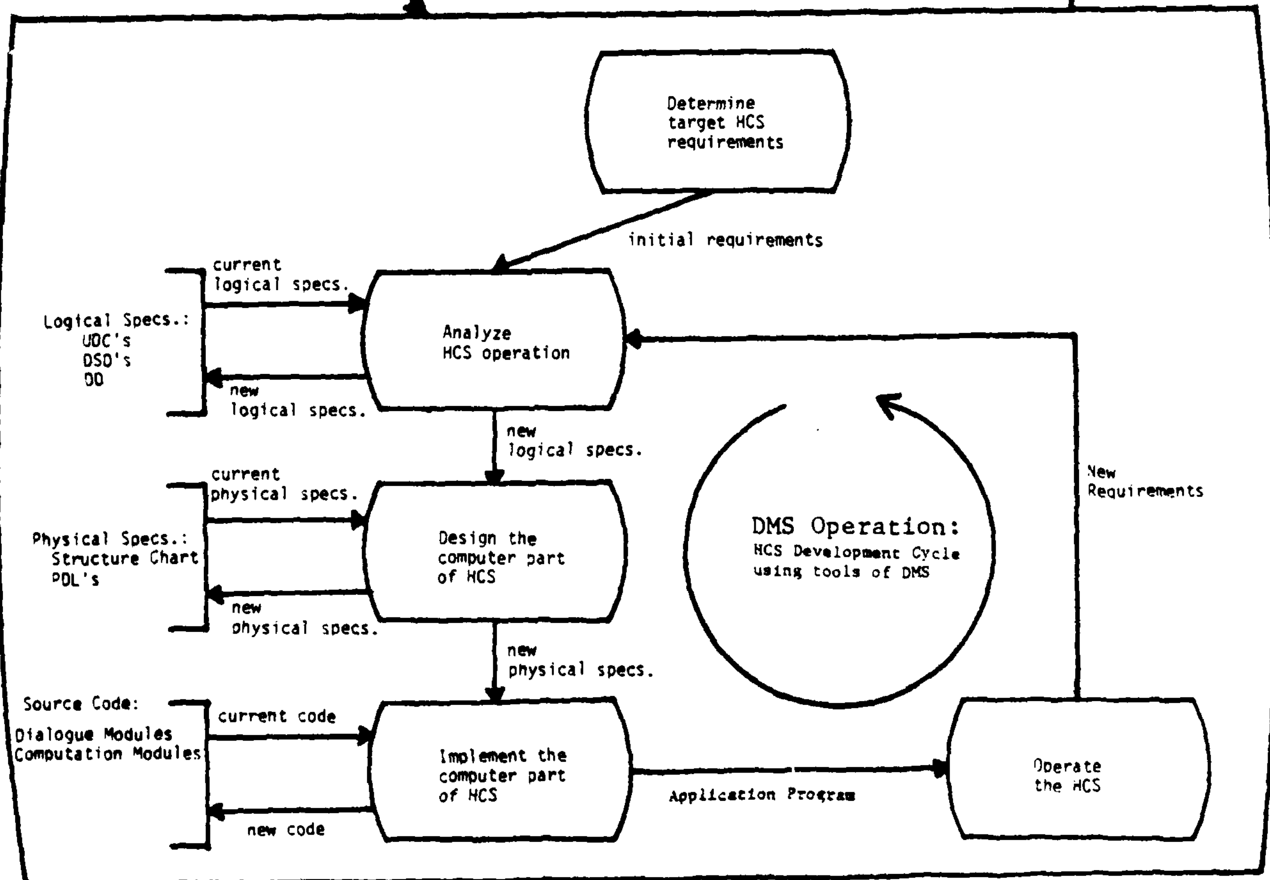


Figure 6.1 DMS development cycle

The DMS representation shown in Figure 2.6 should be clear now. The facilities provided for the HFE and SWE represent the automated part of the methodology in Figure 5.2 and the methodology box of Figure 2.6 represents the manual part of the methodology shown in Figure 5.2. The methodology will be improved during the course of our research, according to the interactions of the HFE and the SWE.

The current function allocation status for DMS is shown in Figure

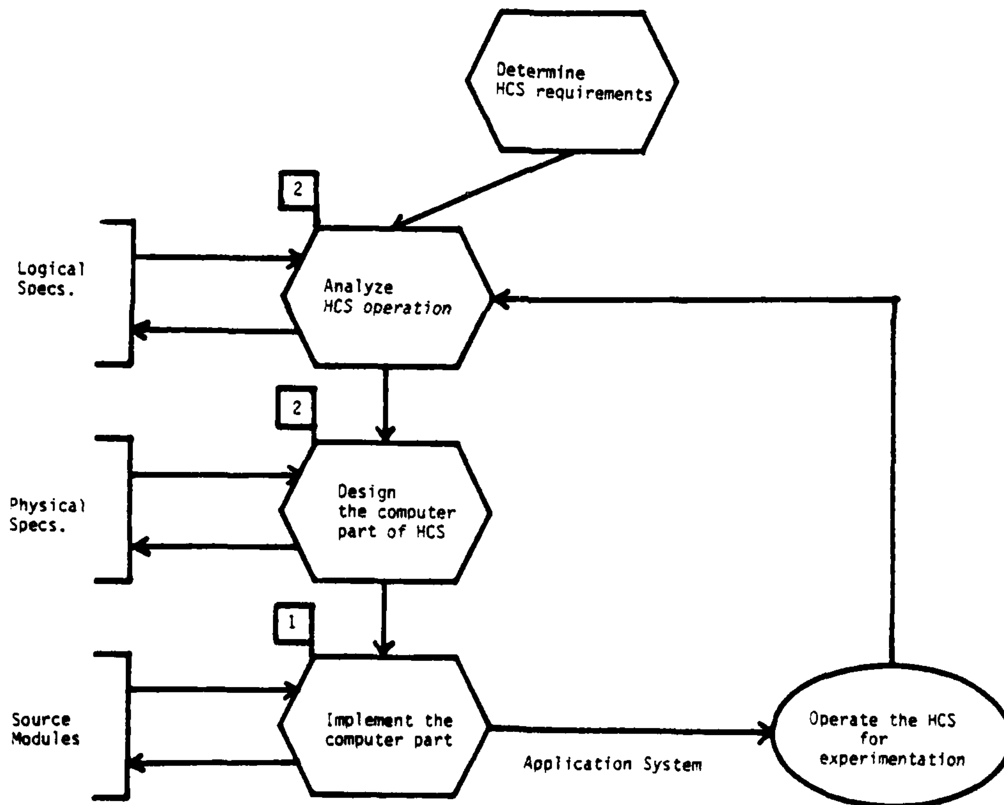


Figure 6.2. Current Function Allocation Status of DMS

6.2 (lower portion of Figure 6.1). [Note that the authors of this report, while writing this discussion, are in the analysis phase of the

DMS life-cycle (upper portion of Figure 6.1).] In Figure 6.2 the analysis, design, and implementation functions of DMS operation are shown with hexagonal boxes, which indicate manual activity. In other words, the system users do not yet have interactive tools which guide them in application system development, so application system development is done manually. (Note that the interactive interface of the host operating system is not part of DMS and therefore its use is considered as a manual activity). The currently available development tool is a high level interprocess communication construct, which hides the host system services for VAX/VMS interprocess communication from the developer. In addition, a set of I/O routines are programmed, to help the developer.

Because the HFE and SWE developmental facilities are not currently implemented, the only interactive function in Figure 6.2 is the one in the right bottom corner, which shows the use of the application system (e.g., GENIE) generated. This interactive service consists of a dialogue function which gets from the user the name of the program wanted to be executed and activates the requested program (which had been manually constructed by the DMS users). The high level process creation and communication constructs are translated to host system services and executed. Note that there are vertical flags attached to hexagonal nodes of analysis, design, and implementation functions. These flags indicate that further functional refinement and function allocation will be made. The numbers in the flags show the priority profile of the developmental work plan. The refinement of the "implementation" function has first priority. This means that further analysis (e.g., constructing lower level UDCs) of the "implementation" function will be done before that of "analysis" and "design" functions.

If the functional decomposition and function allocation is performed on the "analysis" phase of DMS (for instance), automated support might include a graphics editor for constructing functional flow diagrams, and an information system for documentation storage and retrieval (e.g.,the data dictionary). These are both interactive functions and are not of key importance at this time in DMS development, since they can be performed manually. The construction of implementation services, which are to be used in building computational and dialogue modules and performing experiments, is more important.

Note that the UDC in Figure 6.2 shows the current physical status of the system and a management plan for further development. Although the DMS implementation function is definitely going to be designed as an interactive service (e.g., dialogue development facility and computation development facility of Figure 2.6), it is not shown with a UDC function symbol (ellipse). This status FFD is held separately from analysis and design work FFDs so not to confuse what has been done with what remains to be done. The status FFD indicates only the functional description of the services which have been made operational, and the development policy.

7. CONCLUSION

In this report, we presented a methodology for developing human-engineered automated environments (human-computer systems). The key aspects of the methodology are:

1. Humans and their capabilities are considered as system elements and constraints in HCS system view. This is in contrast to the conventional software system view, in which only the computer part of an HCS is modelled and analyzed.
2. The HFE and the SWE work together in cooperation.
3. The dialogue and computational parts of the software are separated.

The Dialogue Management System provides an automated environment to support the use of the methodology by the SWE and the HFE. The methodology and its life-cycle are applied to DMS itself. Further DMS development activities will involve cycling back through the methodology, both for improving the methodology itself and for improving the automated environment in which the methodology is embodied.

8. APPENDIX

8.1. STRUCTURED DESIGN

Structured Design methodology, developed by Constantine, Myers, and Stevens, is a design technique which greatly reduces the development and, especially, the maintenance cost. The use of single-function seperately compiled modules facilitates the addition of new requirements. This makes structured design a very valuable design technique in environments where system requirements cannot be completely specified and/or where system requirements are subject to change over time. Both situations apply to the development of DMS.

Maintainability is provided by minimizing the complexity of the system structure and by reducing the connections between parts (modules) of the system. Hence, when a change in the current system requirements or addition of a new requirement is to be made, the system maintainer works on a structure representation, designed and represented in such a way that the effect of changes is minimal. This characteristic is especially valuable for the independent construction of dialogue modules and computation modules.

Advantages of the methodology are listed by Stevens [STEVW81] as follows:

1. People are less likely to make errors, because less code has to be dealt with at a time. Each module is small in size and performs a single function. The function of the module can be described with a

simple phrase, which makes it easy to understand and remember what the module does.

2. Less code has to be considered to implement changes. Modules contain only the code required by the module function. Hence a module does not contain code for other functions which are not related to the function to be changed.
3. Side effects of changes are reduced. Ideally, each single-function module communicates only utility data. Modules do not interfere with the logic of each other, and the logic of a module can be independently changed, provided that the specified communication interface is not destroyed.
4. Modules can be programmed relatively independently.
5. Design representation is comprehensive. A structure chart used in representing the design is a valuable tool both for the developer and maintainer. The undesirable module connections (coupling) show up on the structure chart and design can be improved by eliminating them. Hence, the final design is easily understandable and maintainable.
6. Reusability of modules is high. Since the modules are single-function modules with one-in, one-out communication interfaces, the modules are reusable in any new functional requirement which needs them.

For achieving high module independence (the essence of structured design), the criteria are maximization of relationships within modules and minimization of relationships among modules [MYERJ78]. In the following we shall discuss module strength (relationships within a module) and module coupling (relationships among modules), as adapted from Myers [MYERJ78] and Stevens [STEVW81].

8.2. MODULE STRENGTH (COHESION)

The main purpose of this module strength discussion is to show how undesirable multiple-function modules are produced. There are seven categories of module strength [MYERJ78, STEVW81]; the scale from highest to lowest strength is:

1. Functional strength
2. Sequential strength
3. Communicational strength
4. Procedural strength
5. Classical strength
6. Logical strength
7. Coincidental strength

A functional strength module (item 1 above) performs a single function and is the most desirable one. The module strength types given above will be discussed from the worst (coincidental) to the best (functional).

8.2.1. Coincidental Strength

Myers [MYERJ78] defines a coincidental strength module as follows :

"1. Its function cannot be defined (i.e. the only way of describing the module is by describing its logic).

2. It performs multiple, completely unrelated functions."

The first type of a coincidental module can be formed by modularizing a sequence of instructions which appear in the program at many places. This may be done to save space and coding time. But the problems arise when one of the calls is changed so that it requires the coincidental module to be changed. If the coincidental module is changed, error is injected into the program because the other callers need it in its earlier form. Stevens [STEVW81] describes how these modules were used at the beginning of the software era:

> "Although these kinds of modules are probably seldom designed anymore, they used to result from tricks to save memory. On the 1401 for example, memory was often so constrained that anything which saved memory allowed more functions to be included. One trick was to look for repeated sequences of instructions. If the instructions were longer than the linkage necessary to branch to them and back, memory could be saved each time they were reused."

An example for the second type of a coincidental module is as following:

input customer-name

get the pilot's experience data

read the populations of the states

None of the functions above are related to each other and there is no reason for them to be together in a module. Since the function of the module cannot be defined, one has to study the interior of the module, to see what the module is doing. When another module wants to use one of the functions given above, a control mechanism has to be introduced in the module, as well as a function switch at the interface.

These switches and control mechanisms increase the complexity of the program.

8.2.2. Logical Strength

A logical strength module consists of logically related functions. The Sine/Cosine example which was given in Figure 4.7 is a logical strength module. One of the functions is explicitly selected by the calling modules, by passing a multiway switch. We have already showed what would happen when some of the functions are needed by functions which come as a result of new requirements.

Difficulties with a logical strength module are the following:

1. Understandability. The module has a single interface for multiple functions and hence it is hard to understand the interface definition as well as the module logic. In addition, understanding the module which makes the call might be difficult. A calling module might not need all the formal parameters specified at the interface, in which case it must make the call with dummy arguments.
2. Consistency. The arguments may have different interpretations depending on which function is called.
3. Modifiability. If the module given in Figure 4.7 was programmed with overlapping code, a change for the Sine function in the overlapping part would require the consideration of the logic of the Cosine function. If the overlapping nature of code is not recognized by the

maintainer, an error might be introduced in the module. The maintainer normally would test the Sine function, to make sure that the change has been made correctly. But the error introduced in the Cosine function would propagate to Tangent and Cotangent functions, too.

Another problem is due to the changes at the interface. If one of the functions is changed such that the interface definition has to be changed, not only the modules which call for the changed function have to be changed, but all the other calling modules (which do not even need the changed function) also have to be changed.

8.2.3. Classical Strength

A classical strength module performs multiple, sequential functions where there is a weak relationship among all of the functions. Examples of classical strength modules are initialization and termination modules. A module which opens all the files and initializes all the tables used by a program does so far away from the functions which are going to use them. If a file is opened, or a table is initialized at the place where it is needed, the scope of consideration for them is reduced.

8.2.4. Communicational Strength

A communicational strength module consists of functions which reference the same data stream. An example of such a module is given in Figure 8.1. The functions "verify user name" and "display user status" are

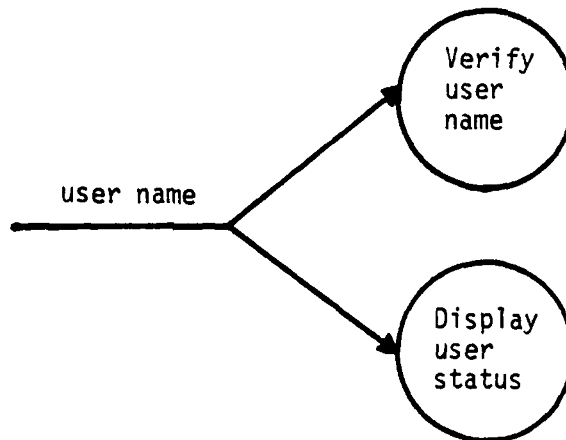


Figure 8.1. A Communicational Strength Module.

both in the same module, because both use the same input data.

The reusability of each function alone requires a control switch to be added to the interface and a control mechanism to the inside of the module. As a consequence, all modules calling this module must be changed (e.g., the control switch must be declared and added to the argument list of all call statements) and all calling modules should be

recompiled. In such cases, the module ends up being a logical strength module, a situation where the overall complexity is increased and further maintenance is made less managable.

8.2.5. Sequential Strength

A sequential strength module consists of multiple functions, which execute sequentially in time, and the output of each function is the input for the next function in the sequence. An example of such a module is shown in Figure 8.2. The functions "get user name" and "verify user name" are in the same module. Output of the former is the input to the latter. Reusability of each function separately requires a control

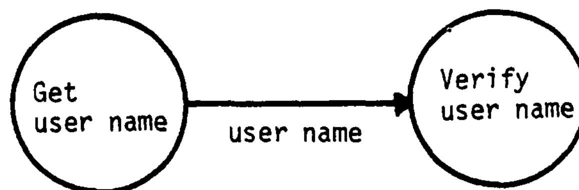


Figure 8.2. A Sequential Strength Module.

switch addition to the interface and a control mechanism in the module. All calling modules then have to be changed and recompiled, and the module ends up being a logical strength module.

8.2.6. Functional Strength

A functional strength module is the ultimate objective. Each module performs a single specific function. The question here is: what is the definition of a single specific function? The definition of a function, given by Stevens [STEVW81] is as follows: "A good way to determine if a module is of functional strength is to write a phrase fully describing what the module does. Analysis of the phrase can indicate whether or not the module is functionally bound. The module is not functionally bound (it performs more than one objective) if you cannot avoid one of the following:

- A compound sentence such as "display flights AND get user choice."
- More than one phrase such as " edit customer-name, address."
- More than one verb such as "get and store the selected flight."
- Words such as "initialization", "termination", "house-keeping", "clean-up". (These imply classical strength.)
- Not using a specific object (e.g., "edit all data"). The lack of a single specific object usually indicates a logical strength module."

8.3. COUPLING

Coupling measures the strength of relationships between the modules. The lower the coupling, the less likely that other modules will have to be considered in order to understand, debug, or change a given module.

In the order of increasing desirability the scale of coupling is as follows [MYERJ78, STEVW81]:

1. Content coupling
2. Common coupling
3. External coupling
4. Control coupling
5. Stamp coupling
6. Data coupling

8.3.1. Content Coupling

Two modules are content coupled if one directly references the insides of the other or if a module branches to (rather than calls) the other module. In the latter case, the normal linkage conventions are bypassed, and the modules are "chained" to each other with no return (GO TO at modular level).

Myers [MYERJ78] describes why content coupled modules are seldom created, as follows:

> "First, virtually everyone recognizes it is a poor programming practice. Secondly, content coupling is extremely difficult to create in a high level language."

Although due to this reasoning we can assume that this kind of coupling is unlikely to occur, "common coupling", discussed in the next section, is very commonly used.

8.3.2. Common Coupling

Modules referencing global data structures are common coupled (e.g., Fortran COMMON). The eight serious problems of common coupling as described by Myers [MYERJ78] are :

1. Readability. Global data inhibits program readability. Top-down readability of the program is destroyed. In an attempt to understand what is happening in the program, one might need to study all the modules which refer to global data. Worst case is that if the code is part of a system containing asynchronously executing processes, every module in each of the other processes could be referencing the global data.
2. Side Effects. Data communication is not apparent in the interface. Assume that we have the following call:

```
Call get_user_name (user_name)
```

Although what this module does is understandable by looking at the call, if the module get_user_name is referencing a global variable, this module has a side effect, which makes it difficult to understand the program logic.

3. Extra Module Interdependencies. Consider the case that a set of modules contains the following globally declared structure:

```
declare 1 glblarea external
          2 name character(18);
          2 numbers fixed decimal (5,5)
```

Assume that modules X and Y make reference only to "numbers" and modules M and N reference only to "name". A modification to modules X and Y that requires an expansion of the size of "numbers" will, at the minimum, require recompilation of M and N, provided that common coupling with these modules is recognized, otherwise a bug is introduced into the program.

4. Usability. Assume that a module, instead of getting all its inputs as dynamic parameters, gets some of its inputs from a global area. If it has multiple calling modules, these modules must place the input in this fixed global area. However, each module has to save and restore the data in the global area, since each module cannot be sure that some other module is not using this area. The result is additional complexity. Especially in a parallel processing environment, the global area becomes a critical region, which requires mechanisms such as semaphores, flags, etc.

5. Naming Dependency. Common-coupled modules are bound together through variable names. In contrast to a module communicating through parameters, if a module is communicating through a common area, related modules must refer to these data with the same name.
6. Creating Dummy Structures. In order to use the module X given in item 3 above, which only refers to "numbers", it is necessary to construct the whole global data structure. Probably this structure will not have any meaning in the new module, but it has to be declared so that X can be used.
7. Exposure to Unnecessary Data. Some of the global data is unnecessarily accessible. In addition to being a possible source of confusion, access to data not used by a module is an open invitation for errors.
8. Uncontrolled Data Access. Difficulties in controlling and managing data access.

Myers [MYERJ78] expresses his judgment on the use of global data as follows:

> "In spite of the serious consequences of global data, people still defend its use by using one of two arguments: "my program cannot be designed without using global data", or "eliminating global data leads to performance problems because of excessive argument transmission". The first argument is false because any program with global data can be transformed into an equivalent one. The second argument usually comes from the misinformed: the cost can be measured in microseconds. If the program is having efficiency problems, it seems rather unlikely that shaving microseconds from CALL statements will eliminate these troubles; more fruitful concerns are looking for better algorithms, improving the use of resources etc."

8.3.3. External Coupling

External coupling is the same as common coupling, except that the data communicated is homogeneous [MYERJ78]. Some examples of homogeneous data are:

1. A single scalar variable (e.g., a character string).
2. A data structure composed of a repeating sequence of a single unit of the same meaning and format (e.g., an array where each element has the same meaning).

In common coupling, the heterogenous data are of concern (e.g., data structures where entities may have different formats and meanings). External coupling is an improvement on common coupling, because the heterogeneous data are replaced by homogeneous data. The following problems of common coupling disappear:

- extra module interdependencies
- creating dummy structures
- exposure to unnecessary data

The other problems (readability, side effects, use in multiple contexts, naming dependencies, uncontrolled data access) remain.

8.3.4. Control Coupling

Two modules are control coupled if one of the modules explicitly controls the logic of the other. Examples of control data are:

- control switch arguments
- function codes transmitted to logical strength modules

- module name arguments.

Control coupling implies that one module knows something about the logic of the other module. The module given in Figure 4.7 is control coupled with its callers. The module should be broken into its component functions, and the caller should call the function it needs, instead of passing its code.

8.3.5. Stamp Coupling

Stamp coupling occurs when a nonglobal data structure (e.g., a record) is passed through modules, where all the fields are not used by the modules.

An example of stamp coupled modules is given in Figure 8.3. Module B in the figure might use only a few fields of the record_x and passes it to module C, or module B might not be using any of the fields of record_x. The problems with stamp coupling are:

1. A change in record_x can result in changes and recompilation of all modules that reference record_x, even if some have nothing to do with the changed fields.
2. The sharing of stamp coupled modules is difficult. For example, a program which wants to use module B above, and which has the same record fields but a different record organization, cannot call B. The record in the program has to be reorganized as declared in B.
3. A module is exposed to more data than needed. If module B in Figure 8.3 uses only four of the fifteen fields of the record_x, the com-

AD-A118 287 VIRGINIA POLYTECHNIC INST AND STATE UNIV BLACKSBURG --ETC F/G 9/2
HUMAN-COMPUTER SYSTEM DEVELOPMENT METHODOLOGY FOR THE DIALOGUE --ETC(U)
MAY 82 T YUNTEN, H R HARTSON N00014-81-K-0143
UNCLASSIFIED CSIE-82-7 NL

2 of 2

END
DATE FILMED
9-82
DTIC

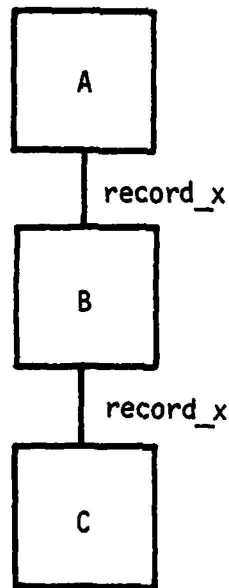


Figure 8.3. Stamp Coupling.

plexity of the module is increased if the programmer makes use of these extra data.

Stamp coupling is better than common coupling because

1. It eliminates the problems of readability, side effects, use in multiple contexts, naming dependencies.
2. It retains the problems of extra module interdependencies, creating dummy structures, exposure to unnecessary data.
3. It reduces the problem of uncontrolled data access.

The following are means of eliminating stamp coupling:

1. Instead of passing a whole record, pass only the fields that the module needs.

2. In cases when the receiving module passes the received record to another module without using any part of it (e.g., module B in Figure 8.3) pass a pointer to the record. For example in Figure 8.3, if B makes no use of the record, it just declares the pointer variable and passes it to C. In C, the record is declared and used. B does not even see the contents of the record.

8.3.6. Data Coupling

A data coupled module is the ultimate objective. A data coupled module is a module which does not have any of the above forms of coupling. It gets only the data it needs in the form of parameters, with no global structures, no control coupling (function codes and switches being passed back and forth), and no excess data.

REFERENCES

BASVR79 Basili, V. R., & Reiter, R. W. "An Investigation of Human Factors in Software Development", IEEE (December 1979).

BATED77 Bates, D. (ed). "Software Engineering Techniques", Infotech State of the Art Report (1977).

BOEHB76 Boehm, B. W. "Software Engineering", IEEE Trans on Computers (1976).

BROWR77 Brown, R. R. Transcript, Infotech State of the Art Conference (1977).

CAINS75 Caine, S., & Gordon, E. "PDL - A Tool for Software Design" Proc. AFIPS, vol 44, pp. 271-276, (1975).

CANNR77 Canning, R. G. EDP Analyzer (July 1977).

DIJKE76 Dijkstra, E. "Programming Methodologies, Their Objectives and Their Nature", Infotech International, pp. 203-206, (1976).

FREEP76 Freeman, P. "Essential Elements of Software Engineering Education", Proc. 2nd Int. Conf. on Software Engineering (1976).

FREEP77a Freeman, P. "Nature of Design", Tutorial on Software Design Techniques, Inst. of Electrical and Electronics Engng., pp. 29-36, (1977).

FREEP77b Freeman, P. "The Context of Design", Tutorial on Software Design Techniques, Inst. of Electrical and Electronics Engng., (1977).

FREEP77c Freeman, P. "Software Reliability and Design: A Survey", Tutorial on Software Design Techniques, Inst. of Electrical end Electronics Engng., pp. 75-85, (1977).

HARTH82 Hartson H. R., Ehrich R.W., & Roach J. W. "The Management of Dialogue for User/Software Interfaces", Department of Computer Science, V.P.I. & S.U., Technical Report (1982).

HOSIJ78 Hosier, J. (ed.). "Structured Analysis and Design", Vol. 1: Analysis and Bibliography, Infotech State of the Art Report, pp. 195-208, (1978).

JACKM75 Jackson, M. A. Principles of Program Design, Academic Press (1975).

JOHND82 Johnson, D. H., & Hartson H. R. "The Role and Tools of a Dialogue Author in Creating Human-Computer Interfaces", Department of Computer Science, V.P.I. & S.U., Technical Report CSIE-82-8 (May 1982).

LISKB72 Liskov, B. H. "A Design Methodology for Reliable Software Systems", Proceedings of the 1972 Fall Joint Conference, Montvale N. J., AFIPS Press, pp. 191-199, (1972).

MARKR76 Marker, L. R. "Software Requirements Engineering", in Infotech State of the Art Report Conference on Structured Design, Amsterdam, (October 1976).

MEISD71 Meister, D. Human Factors : Theory and Practice, John Wiley & Sons, Inc., (1971).

MYERJ75 Myers, G. J. Reliable Software through Composite Design, Mason/Charter Publishers, (1975).

MYERJ78 Myers, G. J. Composite/Structured Design, Litton Educational Publishing, Inc., (1978).

PARND72 Parnas, D. L. "On the Criteria to be used in Decomposing Systems into Modules", CACM, (December 1972).

ROSSD77 Ross, D. T. & Schoman, K. E., "Structured Analysis for Requirements Definition", in Tutorial on Software Design Techniques, Inst. of Electrical and Electronics Engng., (January 1977).

STAYJ76 Stay, J. F. "HIPO and Integrated Program Design", IBM Systems Journal, Vol 15, no. 2, pp. 13-18, (1976).

STEVW74 Stevens, W. P., Myers, G. J., & Constantine L. L. "Structured Design", IBM Systems Journal, v. 13, (1974).

STEVW81 Stevens, W. P. Using Structured Design, John Wiley and Sons, Inc., (1981).

TEICD76 Teicherow, D. "PSL/PSA --A Computer Aided Technique for Structured Documentation and Analysis of Information Processing Systems, Proc. 2nd Int'l Conf. on Software Engineering (1976).

WASSA77 Wasserman, A. I. "On the Meaning of Discipline in Software Design and Development", Tutorial on Software Design Techniques, Inst. of Electrical and Electronics Engineering, pp. 37-44, (1977).

WASSA80 Wasserman, A. I. "Information System Design Methodology", Journal of the American Society for Information Science (1980).

WEINV80 Weinberg, V. Structured Analysis, Prentice-Hall, Inc., (1980).

WINSR75 Winston, R. W. "Software Requirements Analysis", in Practical Strategies for Developing Large Software Systems, Horowitz, E. (ed.), Addison Wesley, (1975).

YOURE79 Yourdon, E. & Constantine, L. L. Structured Design, Prentice-Hall, Inc., (1979).

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