AD A081319

Technical Report 401

AN ANALYSIS OF SOFTWARE DESIGN METHODOLOGIES

H. Rudy Ramsey, Michael E. Atwood,
and Gary D. Campbell
Science Applications, Incorporated

HUMAN FACTORS TECHNICAL AREA

U. S. Army
Research Institute for the Behavioral and Social Sciences

August 1979

Approved for public release; distribution unlimited.

U. S. ARMY RESEARCH INSTITUTE
FOR THE BEHAVIORAL AND SOCIAL SCIENCES

A Field Operating Agency under the Jurisdiction of the
Deputy Chief of Staff for Personnel

	WILLIAM L. HAUSER
JOSEPH ZEIDNER	**Colonel, U S Army**
Technical Director	**Commander**

Research accomplished under contract
to the Department of the Army

Science Applications, Inc.

NOTICES

DISTRIBUTION· Primary distribution of this report has been made by ARI Please address correspondence concerning distribution of reports to U S Army Research Institute for the Behavioral and Social Sciences, ATTN PERI-P, 5001 Eisenhower Avenue, Alexandria, Virginia 22333

FINAL DISPOSITION· This report may be destroyed when it is no longer needed. Please do not return it to the U. S. Army Research Institute for the Behavioral and Social Sciences.

NOTE The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
Technical Report 401

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
AN ANALYSIS OF SOFTWARE DESIGN METHODOLOGIES.

5. TYPE OF REPORT & PERIOD COVERED
Technical Rep[redacted]t.
3 Oct [redacted]77 — 3 Dec [redacted]78,

6. PERFORMING ORG. REPORT NUMBER
SAI-79-104-DEN

7. AUTHOR(s)
H. Rudy Ramsey Michael E. Atwood and
Gary D. Campbell

8. CONTRACT OR GRANT NUMBER(s)
DAHC19-78-C-0005

9. PERFORMING ORGANIZATION NAME AND ADDRESS
Science Applications, Inc.
7935 E. Prentice Avenue
Englewood, CO 80111

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
2Q762725A778

11. CONTROLLING OFFICE NAME AND ADDRESS
U.S. Army Research Institute for the Behavioral
and Social Sciences (PERI-OS)
5001 Eisenhower Avenue, Alexandria, VA 22333

12. REPORT DATE
Aug[redacted]79

13. NUMBER OF PAGES
111

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Of.
--

15. SECURITY CLASS (of this report)
nclassified

DECLASSIFICATION/DOWNGRADING SCHEDULE
--

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

(18) ARI (19) TR-401

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

--

18. SUPPLEMENTARY NOTES

Monitored technically by Jean Nichols Hooper and Edgar M. Johnson, Human Factors Technical Area, ARI.

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

Human factors engineering
Computer programs
Computer programming
Documentation
Cognitive psychology

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

Four formal software design methodologies were described and briefly analyzed: (1) Structured Design, (2) Jackson's Methodology, (3) Integrated Software Development System (Higher Order Software), and (4) Warnier's Logical Construction of Programs. Relative strengths, weaknesses, and commonalities among the methods were identified and human factors problem areas were analyzed.

over

(Continued)

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

Unclassified

i SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Item 20 (Continued)

Several major human factors deficiencies and problems were identified. Formal software design methods differ in terms of: Applicability to problems of different types, size or complexity; susceptibility to design errors; and constraints and limitations imposed on the software designer. Various methods limit the designer's ability to select an appropriate problem representation, prevent the designer from utilizing relevant knowledge and experience, or impose potentially significant information loads on the designer. Improvements in design methodologies require a better understanding of the problem-solving behavior of software designers; potential research topics in this area were identified.

Technical Report 401

AN ANALYSIS OF SOFTWARE DESIGN METHODOLOGIES

H. Rudy Ramsey, Michael E. Atwood,
and Gary D. Campbell
Science Applications, Incorporated

Submitted by:
Edgar M. Johnson, Chief
HUMAN FACTORS TECHNICAL AREA

Approved by:

Milton S. Katz, Acting Director
ORGANIZATIONS AND SYSTEMS
RESEARCH LABORATORY

U.S. ARMY RESEARCH INSTITUTE FOR THE BEHAVIORAL AND SOCIAL SCIENCES
5001 Eisenhower Avenue, Alexandria, Virginia 22333

Office, Deputy Chief of Staff for Personnel
Department of the Army

August 1979

Army Project Number 2Q762725A778	Human Factors in Software Development

Approved for public release; distribution unlimited.

ARI Research Reports and Technical Reports are intended for sponsors of R&D tasks and for other research and military agencies. Any findings ready for implementation at the time of publication are presented in the last part of the Brief. Upon completion of a major phase of the task, formal recommendations for official action normally are conveyed to appropriate military agencies by briefing or Disposition Form.

FOREWORD

The Human Factors Technical Area of the Army Research Institute (ARI) is concerned with human resource demands of increasingly complex battlefield systems used to acquire, transmit, process, disseminate, and utilize information. This increased complexity places great demands upon the operator interacting with the machine system. Research in this area is focused on human performance problems related to interactions within command and control centers as well as issues of system development. It is concerned with such areas as software development, topographic products and procedures, tactical symbology, user-oriented systems, information management, staff operations and procedures, and sensor systems integration and utilization.

One area of special interest involves the development of computer software to support automated battlefield systems. Software development is costly, unreliable, and not well understood. In this research, software design methodologies were analyzed in terms of human problem-solving behavior. The analysis indicated that the design methods imposed varying constraints and demands on the software developer and that the methods differed in susceptibility to design errors. This research is part of a larger effort to develop a conceptualization of the programming process and identify behavioral bottlenecks in software development. Efforts in this area are directed at improving accuracy and productivity in programming through the design of procedures, languages, and methods to enhance performance of software development tasks.

Research in the area of human factors in software development is conducted as an in-house effort augmented contractually by organizations selected as having unique capabilities and facilities, in this case Science Applications, Inc., under contract DAHC19-78-C-0005. The effort is responsive to requirements of Army Projects 2Q762725A778, and to general requirements expressed by members of the Integrated Software Research and Development Working Group (ISRAD).

The authors are indebted to Martha Cichelli, Margaret Hamilton, Henry Ledgard, John O'Hare, and Edward Yourdon for their helpful comments on this report.

JOSEPH ZEIDNER
Technical Director

AN ANALYSIS OF SOFTWARE DESIGN METHODOLOGIES

BRIEF

Requirement:

To describe and analyze alternative formal software design methodologies.

Procedure:

Four formal software design methodologies were described and briefly analyzed: (1) Structured Design, (2) Jackson's Methodology, (3) Integrated Software Development System (Higher Order Software), and (4) Warnier's "Logical Construction of Programs." Relative strengths, weaknesses, and commonalities among the methods were identified and human factors problem areas analyzed.

Findings:

Several major human factors deficiencies and problems were identified. Formal software design methods differ in terms of: applicability to problems of different types, size or complexity; susceptibility to design errors; and constraints and limitations imposed on the software designer. Various methods limit the designer's ability to select an appropriate problem representation, prevent the designer from utilizing relevant knowledge and experience, or impose potentially significant information loads on the designer. Improvements in design methodologies require a better understanding of the problem-solving behavior of software designers; potential research topics in this area were identified.

Utilization of Findings:

This description and analysis of software design methodologies will assist software developers in selection of an appropriate design method consistent with the problem type, size, and complexity. This analysis also provides useful information to the software designer on the potential for design errors using different techniques, and explicitly identifies areas where design methods are imprecise and may be difficult to implement. With additional information on the cognitive performance of software designers, the identified weaknesses of the design techniques reviewed may be improved.

Preceding page blank

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF FIGURES

LIST OF FIGURES (Continued)

INTRODUCTION

The production of computer software has become an area of increasing interest and concern. Software production, as currently practiced, is extremely costly and error prone. Boehm (1973) reported that the percentage of total system development cost associated with software is climbing rapidly, from less than 20% in the 1950's to 70% in the 1970's and to a projected 90% of total system costs by 1985.

Clearly, there are many factors involved in the high cost and other problems associated with today's software development process. It seems probable, however, that the greatest potential for real improvement lies in concentrating on the software design process. Boehm (1975) reports that, in several large system development efforts, analysis and design accounted for 33-46% of the total effort expended. He also estimates that each additional unit of time allocated to analysis and design saves 1.5-3 units of time in later programming, debugging, and integration stages (Boehm, 1974).

Design errors present a serious problem because they are frequent and because they are very difficult and expensive to correct. In a major study of Air Force automation requirements in the command and control area (Boehm and Haile, 1972), a review of many software projects indicated that the majority of errors were design, not coding, errors. Even more significantly, most design errors were not detected until the system test phase. Of all errors, 54% were not found until after acceptance testing. Coding errors accounted for only 9%; the other 45% were design errors.

In response to these problems, several formal techniques for software design have recently been proposed. This paper provides a review of these techniques and an analysis of the psychological issues and properties underlying the techniques.

Before proceeding, however, we will briefly consider what is meant by the term "software design". Figure 1 presents brief descriptions of software design and several other software development tasks. For the purposes of this paper, the term software design is used with a fairly precise meaning. It does not include deciding how a system should behave from the viewpoint of the user. That activity is called "systems analysis," and is not concerned with the internal structure of the software. Many different software designs might be derived, any of which would exhibit similar behavior from the user's viewpoint. Software design is also not concerned with the detailed procedural logic required to accomplish a particular computation. That is algorithm or program design, or "programming".

Logically, software design occurs after system analysis and before programming. Software design can be considered as "the process of translating functional specifications into a structural description of a system that will satisfy these specifications". There are, in general, three characteristics of this structural description. First, the description involves a "modular decomposition"; that is, the software functions required by the specifications are decomposed into a collection of units or program modules, each of which satisfies only part of these specifications. Second, the design includes specification of properties of the data flow among modules, which provides for communication among program units. Third, a software design usually includes the definition of the data structures that are required to satisfy the functional requirements, including information about data types, organization into records and files, etc.

An example may clarify the distinction among these three types of design information. A rather informal functional specification that could be given to a designer for a small system is presented in Figure 2. A possible design to meet these specifications is illustrated in Figure 3. Each of the "boxes" in Figure 3a represents a module, and the arcs connecting the modules serve to describe, at a very high level, the control flow. An appropriate decomposition of the system into

Name	Description
System Analysis or Functional Design	Determination of the desired behavior of the system, without regard for underlying hardware or software.
Software Design	Determination of software system structure, including data structures and program modules.
Programming	Detailed definition of the logical operations, procedures, or algorithms involved in a single program module.
Coding	Translation of detailed logical design of a program module into a programming language.
Debugging	Diagnosis and correction of the errors in a program module.
Integrated Testing	After known errors in individual program modules have been corrected, diagnosis and correction of errors in groups of modules together, and eventually the whole system.

Figure 1. Some of the Tasks Involved in Software Development.

PAGE-KEYED INDEXING SYSTEM

BACKGROUND.

A book publisher requires a system to produce a page-keyed index. This system will accept as input the source text of a book and produce as output a list of specified index terms and the page numbers on which each index term appears. This system is to operate in a batch mode.

DESIGN TASK.

You are to design a system to produce a page-keyed index. The source file for each book to be indexed is an ASC11 file residing on disk. Page numbers will be indicated on a line in the form /*NNNN WHERE /* are marker characters used to identify the occurrence of page numbers and NNNN is the page number.

The page number will appear after a block of text that comprises the body of the page. Normally, a page contains enough information to fill an 8 1/2 x 11-inch page. Words are delimited by the following characters: space, period, comma, semi-colon, colon, carriage-return, question mark, quote, double quote, exclamation point, and line-feed. Words at the end of a line may be hyphenated and continued on the following line but words will not be continued across page boundaries.

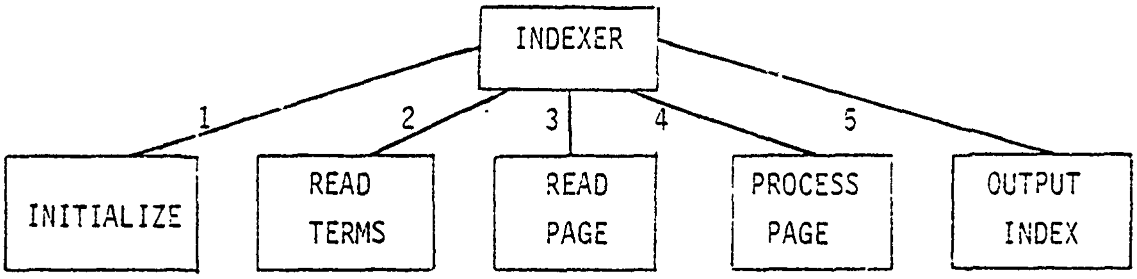
A term file, containing a list of terms to be indexed, will be read from a card reader. The term file contains one term per line, where a term is 1 to 5 words long.

The system should read the source files and term files and find all occurrences of each term to be indexed. The output should contain the index terms listed alphabetically with the page numbers following each term in numerical order.

A null source fill indicates that processing is completed. Error messages and a termination message should be written to the operator's console. Each completed index is to be stored on disk for later listing.

Figure 2. Example of a Simple Software Design Problem (adapted from Levin, 1976).

(a) Modular Structure



(b) Data Flow

	In	Out
1.	- - - -	- - - -
2.	- - - -	Index terms
3.	- - - -	Page of text
4.	Index terms, page of text	Index entries
5.	Index terms, index entries	- - - -

(c) Data Structure

```
Input Data Structures                          Output Data Structure

01 Test file                                   01 Index
   02 Page (multiple instances)                   02 Index group (multiple instances)
      03 Line of text (multiple instances)           03 Index term
      03 Page number                                 03 Index entry (multiple instances)
   02 End of file                                 02 End of file
01 Index term file
   02 Index term (multiple instances)
   02 End of file
```

Figure 3. Simplified Example of Software Design for the Problem of Figure 2.

program modules not only results in a logically simple and comprehensible design, but serves as a basis for the allocation of separate parts of the implementation effort to different programmers. Actually the MODULAR STRUCTURE of Figure 3a is at a rather high level, and a more detailed design would ordinarily be specified, even though this system is small enough to be implemented by a single programmer.

Notice that the lines connecting modules in Figure 3a are numbered. These numbers correspond to the DATA FLOW specification of Figure 3b, which illustrates the flow of data among modules. For example, when the main ("Indexer") module calls the "Process Page" module, the main module makes the index terms and a page of text available to the "Process Page" module. "Process Page" performs its function, and may return index entries to the main module. This passage of data, or data flow, is shown in line 4 of Figure 3b.

Finally, Figure 3c illustrates a DATA STRUCTURE specification for this problem. It shows, for example, that an input Text File consists of any number of text pages, each with multiple text lines and a single page number.

Not every software design involves all three types of specification shown in Figure 3, nor does every software design method address all of these. They are all included here to illustrate three basic classes of information which software designs may contain. A fourth class of design information, control flow, is illustrated only loosely in Figure 3, but will be discussed in a later section on design documentation techniques.

This review is intended to provide both descriptive and critical information about a representative set of formal software design methodologies. In particular, the survey reported here had 5 goals:

1. Enumerate the relative strengths and weaknesses of each considered technique.
2. Identify commonalities and differences.
3. Critically analyze human factors problem areas.
4. Make specific recommendations for improvements in design techniques.
5. Formulate hypotheses for the empirical analysis of software techniques.

The remainder of this report is divided into five sections. Section 2 discusses the software design process from a theoretical perspective, and provides the framework for subsequent discussion of formal design methodologies. Section 3 reviews a variety of informal design techniques which preceded the development of these formal methodologies. Section 4 discusses design documentation techniques, and clarifies further the variety of information which may appear in a finished software design.

Section 5 presents a description of the formal design methodologies which were surveyed in this study. These include:

Structured Design
Jackson's Methodology
Integrated Software Development
System (ISDS/HOS)
Warnier's "Logical Construction
or Programs"

Section 6 contains a brief analysis of formal design methodologies as problem-solving procedures and indicates potential areas for future research on software design.

THE DESIGN PROCESS

"Design" has always been an integral part of software development, but has only recently begun to receive attention. Of all the tasks involved in software development, design is perhaps the least well understood. In general, the software design process does not appear to be the type of algorithmic or mechanical process that can be easily and clearly described to others. As the Boehm and Haile (1972) study implies,design also appears to be difficult, or at least highly error-prone. Design is the one phase of software development that produces the most errors, the most serious errors, and the longest-lasting errors. This is clearly not the result of a simple process.

DESIGN TASKS IN GENERAL

Although our knowledge of software design behavior is limited, design tasks in general have been studied. Design problems have been categorized based on task requirements, and general problem-solving methods have been analyzed. In this section, some work in these areas will be reviewed, and then applied to the analysis of software design tasks.

A very general definition of design is presented in Simon's (1969, p. 59) discussions of the "sciences of the artificial" -- "design ... is concerned with how things ought to be, with devising artifacts to attain goals". Simon is concerned with contrasting "natural" and "artificial" sciences. Basically, a "natural" science is "a body of knowledge about some class of things -- objects or phenomena -- in the world; about the characteristics and properties that they have; about how they behave and interact with each other" (p. 1). An "artificial" science, in contrast, is one that is created by man and dynamically altered or molded to fit man's current conceptions of his environment.

The distinction between solving natural and artificial problems is quite clear. Natural sciences are concerned with "how things are" and the natural scientist knows (or presumably can discover) the laws. phenomena, and other techniques for dealing with such problems. Artificial sciences, however, are concerned with "how things ought to be" and the "laws" and techniques for dealing with such problems are, like the science, artificial.

The recognition that design is an artificial science lends support to Bazjanac's (1975) criticism of computer aids for architectural design. Bazjanac notes, quite correctly, that the promises of computer-aided design are largely unfulfilled. He argues that "the underlying causes of these promises are misconceptions about the design process and how design is done. The most appalling of them is the notion that one can extract formal models from the design process and that the operation and utilization of such models can be separated from other activities of design" (p. 25). This criticism is consistent with the characterization, by Bazjanac and others, of design as a "wicked" problem. One characteristic of a wicked problem is that defining the criteria which must be met by an appropriate solution to the problem is equivalent to solving the problem. It is clearly overly pessimestic to conclude that design behavior cannot, in principle, be understood. It is important to recognize, though, that such understanding may be difficult to achieve.

Bazjanac's characterization of the design process is similar to Simon's (1973) distinction between "ill-structured" and "well-structured" problems. In order to be well-structured, a problem must, among other criteria, have a clearly defined goal and a method for testing whether this goal is attained, have clearly defined components, and provide a means for the problem solver to represent and use any knowledge that is considered appropriate. Simon argues quite strongly (using architectural design as an example) that most, if not all, problems are ill-structured and that "definiteness of problem structure is largely an illusion that arises when we systematically confound the idealized problem that is presented to an idealized (and unlimitedly powerful) problem solver with the actual problem that is to be attacked by a problem solver with limited (even if large) computational capacities. If formal completeness and decidability are rare properties in the world of formal systems, effective definability is equally rare in the real world of large problems" (p. 186).

The point of view we will take in this review is to consider software design as a type of problem-solving task. It is important to determine, therefore, what type of problem software design is, from a human problem-solving perspective. Previous research on human problem solving suggests two dimensions for classifying problems--"type" and "size." Type of problem is important since different types of problems are, in general, best approached with different problem-solving techniques. Size is important, since the software designer has a finite amount of resources (human memory resources, processing resources, etc.) and it is not uncommon that the demands imposed by a given problem exceed those resources.

Greeno (1978) proposes a clarification consisting of three types of problems. In "problems of inducing structure" (e.g., analogy problems), the elements of the problem are given and the task is to discover the pattern of relations among the elements. In "problems of arrangement" (e.g., anagrams), the elements of the problem are given and the task is to generate possible arrangements and search for an arrangement that meets some criterion. In "transformation problems" (e.g., towers of Hanoi), the initial situation, desired situation, and a set of operators that transform one situation into another are given and the task is to find some sequence of operations that transforms the initial situation into the desired situation. Greeno's taxonomy, though not very detailed, suggests that certain problem-solving techniques apply to one class of problem but not necessarily to the others. For example, means-ends analysis is a generally effective heuristic for transformation problems, but is less effective (and perhaps even inappropriate) for arrangement or structure problems.

Problem-solving methods can be viewed as processes which require some input and return some output (cf. Newell, 1973). The input to a method is information about the task or problem to which it is to be applied. In effect, a method requires that certain "givens" be present in the statement of the problem. A method can, of course, require relatively many or relatively few "givens". The outputs of a method are the results it produces. A method can be guaranteed to deliver useful results or can offer only a possibility of useful results.

In general, methods that require very specific inputs produce very useful results; methods that require less specific inputs produce less useful results. These methods are called, respectively, "strong methods", and "weak methods". An additional distinction is useful; by virtue of requiring little information about a particular application, a weak method is applicable to a larger number of tasks or problems than is a strong method. For example, there are general "troubleshooting" techniques which apply equally well to medical diagnosis, tracing the fault in an automotive electrical system, and software debugging. Clearly, these are extremely weak, general methods. They are broadly applicable, but they are seldom adequate to produce a complete solution to a diagnostic problem without additional, more specific techniques.

SOFTWARE DESIGN

Although the above discussion summarized work on the design process, problem types, and methods, it is difficult to directly apply this literature to software design problems. Software design as a problem-solving task appears to involve aspects of all three classes of problems proposed by Greeno; thus, many different techniques may be required. In addition, software design problems are frequently too large for the designer to conceptualize the entire design at an appropriate level of detail. In effect, the design problem initially demands more resources than the designer has.

Consider, for example, the design of two statistical programs, one of which performs a simple t-test, while the other is a general-purpose statistical package. The t-test program is the kind of problem which is manageable in the designer's head, and little or no design behavior is explicitly observable in tasks of this sort -- the designer simply starts writing the program. It is not at all clear that any formal design methods are required, or would even be helpful, in this situation. In the case of a complex statistical package, however, the requirement for a separable design effort is clear, and it is likely that formal design methods could be beneficial. Even this problem is simple, when compared with many of the large systems being built today.

It is when the design problem is large or complex that the need for formal design methods is most strongly felt. If the problem is too big to be handled _in the designer's head_, structured procedures are needed to help avoid errors and unnecessarily complex designs.

In postulating a _general_ set of guidelines for software design, we are necessarily restricted to proposing weak methods. This is because of the wide differences which exist among software design problem types. For example, it is not at all clear that the methods useful in designing a business report generator are all applicable to the design of a programming language compiler, or vice-versa. It is, of course, theoretically possible to postulate general guidelines employing strong methods. Such approach would involve successively partitioning the larger domain of software design into smaller and smaller subdomains and identifying the strong methods appropriate for each subdomain. In effect, we would have a catalogue of procedures that are sufficient to solve any design problem.

Our current understanding of software design, however, does not allow us to form meaningful and useful partitions. By using weak methods we cannot insure that our methods will always produce useful results, but they can be applied to any software design problem, not just a subset of such problems.

Within the past few years, several prescriptive techniques for software design have been proposed. The present report concerns itself with a representative set of these techniques. Although discussions of these techniques emphasize different concepts and propose guidelines and procedures that appear to be fundamentally different, all of the techniques appear to share a common, very general approach to software design. The principal technique employed by all methodologies involves a "divide and conquer" strategy, more formally referred to as "problem reduction".

Basically, a problem-reduction approach involves generating and solving subproblems. The original problem is analyzed and decomposed into a set of smaller subproblems, whose solutions imply a solution to the original problem. Each subproblem can similarly be decomposed until subproblems are generated whose solutions are considered to be trivial.

A software design problem may be conceptually too large for the designer to manage, at least in terms of the ultimate level of detail that will be required. If the designer can conceptualize the problem on a more abstract level, however, the designer may be able to decompose it into smaller, more manageable problems. Eventually, subproblems will be produced which the designer considers to be "primitive". These primitive problems are generally solvable by algorithmic means and it is at this point that the design is completed and implementation begins.

There are obvicus advantages to using problem reduction and it appears to be an appropriate approach to software design. We will not enumerate the advantages here except to note that it is often easier to solve two (or several) smaller problems than to attempt one larger problem. Problem reduction is, however, not without potential disadvantages or difficulties.

First, successful problem reduction requires that the solutions of the subproblems imply a solution to the larger problem from which they were decomposed. Assume, for example, that the designer's initial partition of the original problem is incorrect -- it omits a necessary subgoal. Those subgoals that were identified will be expanded, and, at some point, the design will be declared "finished". The resulting design does not provide an adequate solution to the original design problem, but this may go undetected until the implementation phase or even later.

Second, successful use of problem reduction requires that the subproblems be relatively independent. Clearly, the subproblems (modules) of a design must be interrelated to some degree, but the solution of one subproblem should not affect the solutions of other subproblems. For example, if the design of one module causes changes to be made to the design of another, those modules (subproblems) are not independent. In general, modules which are independent can be implemented independently.

Applying a problem-reduction approach requires problem-reduction operators and some type of evaluation function. Problem-reduction operators are used to aid in resolving the first difficulty described, assuring the correctness and sufficiency of the design. Evaluation functions aid in resolving the second difficulty, achieving module independence. A problem-reduction operator is a method for finding some (hopefully) adequate decomposition of a problem. In general, more than one decomposition is possible and evaluation functions are used to determine whether the identified subproblems are independent.

Our current understanding of software design, however, does not allow us to form meaningful and useful partitions. By using weak methods we cannot insure that our methods will always produce useful results, but they can be applied to any software design problem, not just a subset of such problems.

Within the past few years, several prescriptive techniques for software design have been proposed. The present report concerns itself with a representative set of these techniques. Although discussions of these techniques emphasize different concepts and propose guidelines and procedures that appear to be fundamentally different, all of the techniques appear to share a common, very general approach to software design. The principal technique employed by all methodologies involves a "divide and conquer" strategy, more formally referred to as "problem reduction".

Basically, a problem-reduction approach involves generating and solving subproblems. The original problem is analyzed and decomposed into a set of smaller subproblems, whose solutions imply a solution to the original problem. Each subproblem can similarly be decomposed until subproblems are generated whose solutions are considered to be trivial.

A software design problem may be conceptually too large for the designer to manage, at least in terms of the ultimate level of detail that will be required. If the designer can conceptualize the problem on a more abstract level, however, the designer may be able to decompose it into smaller, more manageable problems. Eventually, subproblems will be produced which the designer considers to be "primitive". These primitive problems are generally solvable by algorithmic means and it is at this point that the design is completed and implementation begins.

There are obvious advantages to using problem reduction and it appears to be an appropriate approach to software design. We will not enumerate the advantages here except to note that it is often easier to solve two (or several) smaller problems than to attempt one larger problem. Problem reduction is, however, not without potential disadvantages or difficulties.

First, successful problem reduction requires that the solutions of the subproblems imply a solution to the larger problem from which they were decomposed. Assume, for example, that the designer's initial partition of the original problem is incorrect -- it omits a necessary subgoal. Those subgoals that were identified will be expanded, and, at some point, the design will be declared "finished". The resulting design does not provide an adequate solution to the original design problem, but this may go undetected until the implementation phase or even later.

Second, successful use of problem reduction requires that the subproblems be relatively independent. Clearly, the subproblems (modules) of a design must be interrelated to some degree, but the solution of one subproblem should not affect the solutions of other subproblems. For example, if the design of one module causes changes to be made to the design of another, those modules (subproblems) are not independent. In general, modules which are independent can be implemented independently.

Applying a problem-reduction approach requires problem-reduction operators and some type of evaluation function. Problem-reduction operators are used to aid in resolving the first difficulty described, assuring the correctness and sufficiency of the design. Evaluation functions aid in resolving the second difficulty, achieving module independence. A problem-reduction operator is a method for finding some (hopefully) adequate decomposition of a problem. In general, more than one decomposition is possible and evaluation functions are used to determine whether the identified subproblems are independent.

All of the software design methodologies to be considered in the following sections involve, in one manner or another, problem-reduction operators and evaluation functions. Although they differ with respect to the particular operators and evaluation functions employed, even those differences are smaller than the surface features of the methodologies suggest. There are some important differences, nonetheless, and it will be the purpose of the remainder of this report to indicate some of the similarities and differences among these methods from a human problem-solving point of view.

Given an awareness of the similarities and differences among the methodologies, the paper will consider their advantages and disadvantages. Even if we lack detailed knowledge of software design behavior, it is assumed that design practices are undesirable if they generate poor designs when correctly applied, or clearly overload known human processing or memory limitations, or lead to predictable errors based on our knowledge of human problem-solving behavior. In fact, several deficiencies of a human factors type were identified by this study, and will be discussed later.

Where possible, we have gone beyond the human factors analysis of design techniques, suggesting specific improvements in design techniques. When this was not possible because of a lack of knowledge of designer behavior, research directions are suggested which might provide the needed information.

INFORMAL SOFTWARE DESIGN TECHNIQUES

From the late 1960's to the present, a variety of prescriptive techniques for software design have been proposed. Most have been relatively informal, involving loose guidelines for modular decomposition (separation of the overall design into modules). Only a few of the techniques are more fully developed, step-by-step procedures. In keeping with the terminology of the literature, we will refer to the relatively formal, step-by-step procedures as "software design methodologies", while the others will be called "informal design techniques". In order to provide an appropriate developmental perspective for the discussion of formal approaches, this section discusses the informal techniques. The reader who is familiar with the literature on "top-down" and "bottom-up" techniques, "structured programming", "stepwise refinement", "information hiding", etc., might choose to skim or bypass this section.

BASIC APPROACHES

A review of some of these less proceduralized techniques is presented by Boehm (1975), who also considers the relative advantages and disadvantages of the techniques. The techniques considered by Boehm are "bottom-up" design or programming, two variations of "top-down", "structured programming", and a "model-driven" approach.

When using a bottom-up approach, a designer must first identify those functions or routines whose development seems most "important" to the overall design. "Importance" can be defined in terms of efficiency, cost, development effort, etc. As the term "bottom-up" implies, these functions are at the lower levels of the hierarchical structure that is being developed to represent the design. Once these routines are developed, the designer develops a "test driver" to allow testing of these modules and their interactions, a "computation monitor" to control the order in which these functions are executed, and any necessary input-output modules. Finally, input-output "controllers", initialization routines, etc., are developed and the entire design is then tested for errors.

As applied to program development, a bottom-up approach involves constructing low-level routines and then constructing "drivers" to control interactions among the low-level routines. There are two primary advantages to this approach. First, "high risk" components (e.g., processing natural language, real-time sensors, etc.) can be identified early. If it is determined that it is not feasible to implement these components as originally specified, the design specifications can be changed before a great deal of effort is expended. Second, the emphasis on the lower levels encourages the development of reuseable modules that can be applied to other designs with little or no modification.

The bottom-up approach, like the other approaches discussed in this section, can be used even for a pure design effort. The designer identifies the "important" functions, designs modules to accomplish them, and only then turns his or her attention to the design of the remaining modules.

A primary disadvantage of this approach is that very little attention is given, early in the design process, to the interactions among modules. It may well be the case that interactions among modules present more problems than the development of the individual modules. In addition, a bottom-up approach does not give a great deal of attention to overall system requirements, including user interfaces and data structures. Furthermore, in an effort to use the lower-level components that are already developed, the higher levels of the design may be "patched up". As a result, the total design may be very difficult to implement, understand, or modify.

"Top-down" methods are much more commonly used, and are often advocated, especially for the later stages of software development (programming through integrated testing). A particularly common top-down method for software development is called the "top-down stub" approach. In this approach, the designer first considers the overall system requirements and develops a top-level program to meet these requirements. This top level contains the necessary logic to control the lower-level functions, which are initially represented as "stubs". In successive design steps, these stubs are then aecomposed into control logic and necessary subfunctions, which are also represented as stubs.

As one might expect, the areas in which bottom-up methods are particularly strong -- identification of high-risk components and development of reuseable modules -- are the areas in which top-down methods are weakest. The advantages of top-down methods include early attention to the interactions among modules and a more coherently defined higher level in the design, which allows for easier comprehension, testing, and maintainability. In general, it might also be expected that discrepancies in the original problem statement (user requirements) might be detected earlier and with less effort when top-down methods are used than when using bottom-up techniques.

The "structured programming" approach to design is a direct extension of structured programming concepts (e.g., Dahl et al, 1972) to the design process. The principal concepts are the use of hierarchical modular structures, the use of a restricted set of control structures, (e.g., IF...THEN...ELSE, DO WHILE), and having a single input and output for each module. This approach is compatible with the other approaches mentioned in this section and is especially useful when demonstrations of design "correctness" are important.

"Model-driven design" attempts to relate, frequently through a matrix representation, the "requirements" that are to be satisfied and the "properties" of the computer system involved. Design generally proceeds in a top-down fashion, but the use of such a matrix allows the early identification of high-risk components that may be best developed in a bottom-up fashion. This technique has not been extensively used and appears to describe the management of design activities more than the actual processes involved in design.

Other fairly general design techniques have also been mentioned in the software design literature. "Middle-out" design requires the designer to identify and initially develop the most "important" routine or function; in this regard, this approach is similar to a "bottom-up" approach. The primary difference is that this routine need not be the lowest level of the final design. Rather than being function oriented, as in bottom-up design, the identified routine could be control-oriented, input-oriented, etc. In general, this routine is selected because of constraints on the final implementation, such as hardware constraints, user interface considerations, etc.

Like a bottom-up approach, designing middle-out tends to lead to the early identification and development of high-risk components. The principal disadvantage is that the remainder of the design may be "patched up" to work with the first routine developed, so that this high-risk component will not have to be modified. Also, like a bottom-up approach, this technique may involve the modification and use of previously developed modules. The actual advantages and disadvantages of this approach depend on where in the final design structure the initially developed module falls, since a middle-out approach could, conceivably, proceed in a strictly top-down or bottom-up fashion.

With interactive systems, design may proceed in either an "inside-out" or "outside-in" manner. An inside-out approach begins with a description of basic implementation environment capabilities and functions and attempts, through adding higher level modules, to match these basic capabilities and functions to user requirements. An outside-in approach, on the other hand, begins with a description of the user requirements and attempts to work down toward the available capabilities. While an inside-out approach leads to the development of a very efficient design, in terms of hardware and software, an outside-in approach tends to ensure that the initial statement of user requirements is practical, and if this is not the case, leads to an early reformulation of these requirements.

These approaches describe, only at a very general level, how design should be done. They do not specify, in a formal or procedural way, the actual steps involved in constructing a design. In addition, they do not provide explicit criteria along which the final design or the current state of a developing design can be evaluated. The formal methodologies, discussed later, provide much more detail in terms of procedure, decision criteria, etc. In between these two classes are several developments which involve very general procedures and criteria for modular decomposition. While not procedurally detailed, these techniques provide some high-level guidance with respect to individual design decisions.

STEPWISE REFINEMENT

In stepwise refinement (Wirth, 1971), the designer starts at the top level of the design, which is essentially a statement of the goal "solve the problem". Design then proceeds in a breadth-first, level-by-level manner. These levels can be differentiated with respect to the amount of detail involved. At the early levels, the designer does not consider specific programming languages or other aspects of the environment in which the solution will be implemented. As Ledgard (1973, pp. 45-46) points out, this stage of the design might contain statements like "compute the _n_th prime number", "find the roots of the equation", or "process the payroll". We would characterize this as the abstract plan level. Toward the lower levels of the design, the design works in terms of the implementation environment. The intermediate levels of the design, the detailed plan level, represent a transition between these very general and very specific expressions.

Ledgard (1973) extends the definition of stepwise refinement by incorporating Mill's general top-down concepts (e.g., Mills, 1971) and Dijkstra's definition of structured programming. This technique, called "meta-stepwise refinement" by Shneiderman (1976), provides a clear expression of the general concepts underlying stepwise refinement. Ledgard's approach has six primary characteristics. First, the designer must develop a clear understanding of the problem before proceeding. Second, the initial stages of the design are independent of considerations of the implementation environment; such considerations are only included at lower levels. Third, design is done in discrete levels, although Ledgard admits the possibility that it may be useful to "look ahead" to the probable functions of a lower level. That is, some design decisions may be based on the practicality or risk of the ultimate functions or modules that may be required by these decisions. Fourth, "the programmer concentrates on critical, broad issues at the initial levels, and postpones details until lower levels". Fifth, the designer must ensure that each level represents, at the appropriate level of detail, a correct solution to the problem. Finally, each level is generated by "successive refinement" of the preceding level.

Ledgard advocates that the flow of program control be organized around the data flow of the problem. He also cautions that "structured" or localized use of variables is just as important as the use of structured control flow. In meta-stepwise refinement, the analysis of data flow is used to suggest module boundaries in, essentially, an input-process-output format. The design is required to be level-structured and tree structured and each level of detail within the design must represent a complete solution to the original problem. Level-structuring and tree-structuring are the two principal evaluation functions employed and they are applied after each level of detail is refined.

"INFORMATION HIDING"

Parnas (1972) concentrates on the criteria whereby modularization is accomplished. Parnas claims that his design by "specification of information hiding modules" is both compatible with and complementary to stepwise refinement techniques.

When the "information hiding" technique is used, module boundaries are selected in such a way that each module has "knowledge of a design decision, which it hides from all others". For example, the details of a data structure might be kept in a single module, so that other modules need no information about the physical details of the data structure in order to operate on it. In effect, "information hiding" is a heuristic technique which may aid the designer in achieving high functional coherence of modules and module independence.

GOAL-DIRECTED PROGRAMMING

"Goal-directed programming", as advocated by Cichelli and Cichelli (1977) also extends the concept of stepwise refinement. The primary objective of goal-directed programming is to "group statements into functions or blocks each of which can be treated, at any arbitrary level of nesting, as a single statement" (Cichelli and Cichelli, 1977, p. 58). Like the other techniques considered in this section, this objective is concerned with decomposing a design into independent functions, or modules.

The addition made by this technique is the concept of an explicit statement of the goal to be achieved by the design. This approach involves stating the goal to be achieved, deriving an assertion that will be affirmed when the goal is true, and then deriving a logical condition, from this assertion, that will become true when the assertion becomes true. By iterating these steps, the original goal is decomposed into subgoals. At a general level, this technique is similar to the use of a means-ends analysis heuristic.

DESIGN DOCUMENTATION TECHNIQUES

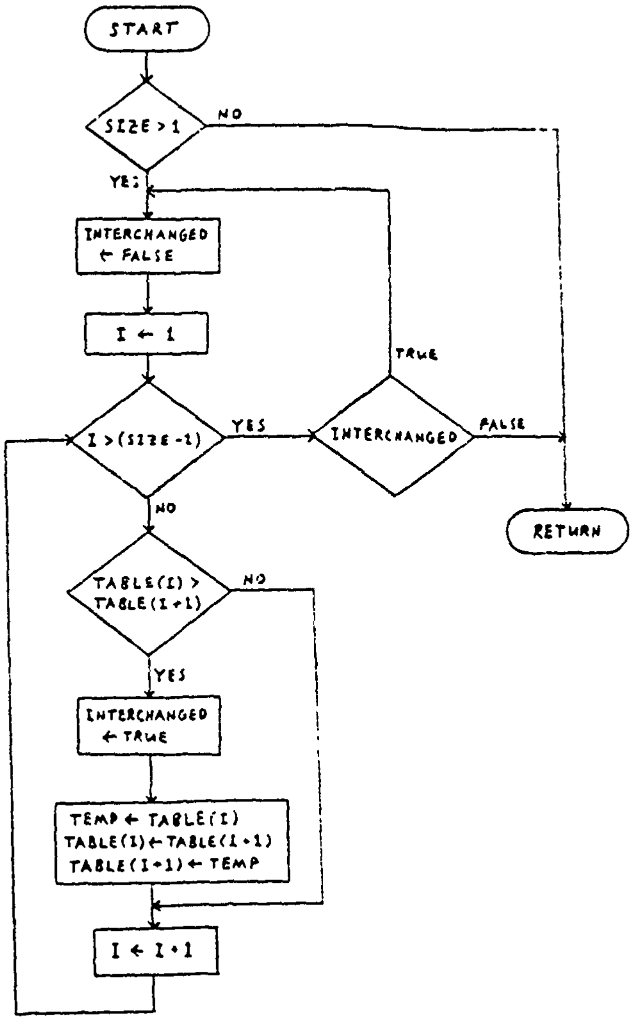
Design documentation techniques should also be considered prior to a discussion of formal design methodologies. A variety of such techniques has emerged, with widely varying information content. Like the informal design techniques of the previous section, these documentation techniques have greatly influenced the software development process. Because they may very well constrain or "lead" the designer, or at least focus the designer's attention on particular aspects of the design, documentation techniques are intimately involved with the design process itself.

Both graphical and verbal documentation techniques are in widespread use. These are not mutually exclusive, since the graphical techniques invariably contain verbal information, and the verbal techniques often make use of spatial cues, such as indentation, to convey information. The most familiar graphical technique is the flowchart (Figure 4a). Flowcharts are used less often for high-level software design than for detailed program design. This is primarily because flowcharts emphasize the flow of control, rather than program structure.

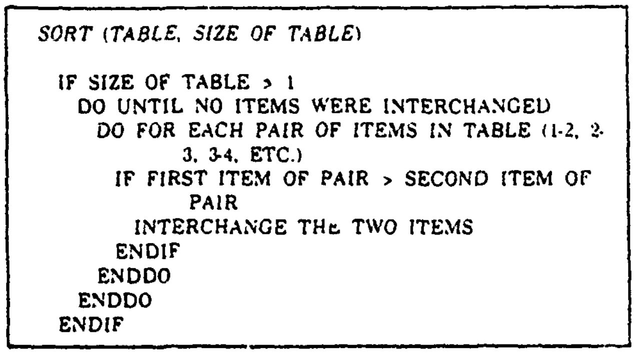
Another graphical technique which is widely used for software design is the "structure chart". A simple structure chart was presented in Figure 3a. Unlike the flowchart, the structure chart emphasizes the basic function of each software module and its relationship to other modules, but contains little information about the flow of control. In Figure 3a, for example, the "INDEXER" module may call any of the other five modules, as needed, and does so by transferring control to the called module. However, the figure does not explicitly state whether all modules are used, in what order, or how often.

Several variants on the basic structure chart have attempted to incorporate some additional information about control flow. While avoiding the detailed control flow information commonly found in flowcharts, several groups have adopted structure charts with logical "and"

(a) Flowchart



(b) Program Design Language



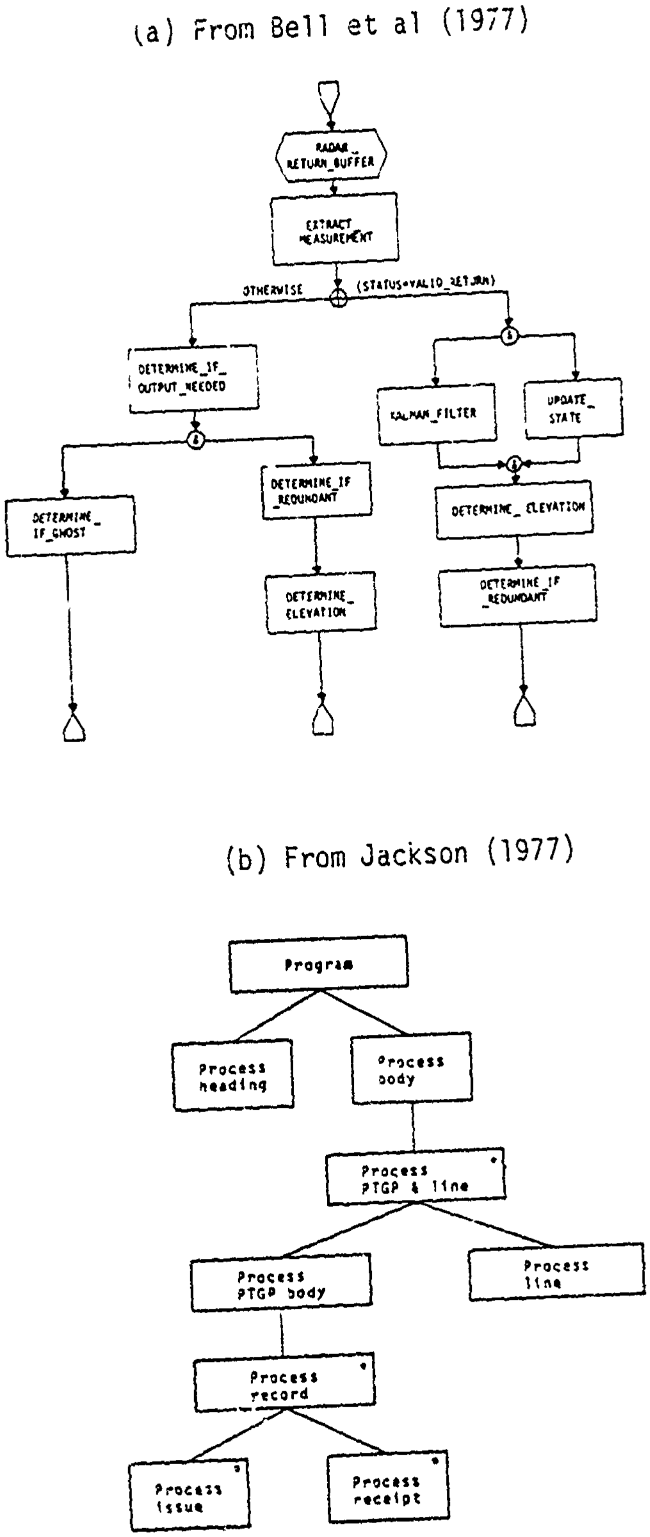
```
SORT (TABLE, SIZE OF TABLE)

  IF SIZE OF TABLE > 1
    DO UNTIL NO ITEMS WERE INTERCHANGED
      DO FOR EACH PAIR OF ITEMS IN TABLE (1-2, 2-
            3, 3-4, ETC.)
        IF FIRST ITEM OF PAIR > SECOND ITEM OF
              PAIR
          INTERCHANGE THE TWO ITEMS
        ENDIF
      ENDDO
    ENDDO
  ENDIF
```

Figure 4. Flowchart and Program Design Language Representatives of a Simple Sorting Algorithm (from Caine & Gordon, 1977).

and "or" symbols, and with explicit indications of iteration. For example, Figure 5a (from Bell et al, 1977) contains a structure chart with logical "and" ("&") and "or" ("+") symbols. Figure 5b illustrates Jackson's (1977) approach, in which asterisks ("*") are used to indicate iteration (the "Process Record" block might be read, "Process each record in turn"), and small circles are used to indicate "selection" (either "Process Issue" or "Process Receipt").

The principal verbal design documentation method is the "Program Design Language", or PDL, shown in Figure 4b. The figure shows a PDL version of the sorting algorithm of Figure 4a. PDLs are similar to programming languages in some respects, but are usually much less formal. An informal PDL might involve a specified set of control constructs (e.g., IF. ..THEN...ELSE, DO WHILE, etc.), but otherwise leave the designer free to use any wording which seems appropriate. There is evidence that the use of PDLs, rather than flowcharts, during detailed algorithm design results in superior design performance (Ramsey et al, 1978), but no controlled research on PDL use for high-level design tasks is known to us. It is clear, though, that the PDL concept can be used at any level of design, since it can be used in such a way as to emphasize either modular structure or flow of control, as desired.

A flowchart-like approach which also attempts to capture some modular-structure information is the "Chapin Chart" (Chapin, 1974) illustrated in Figure 6a. This approach uses embedded rectangles to show containment of procedural steps within a program module, and utilizes special conventions for "DO UNTIL" (slashes at left of repeated block), "IF...THEN...ELSE" (binary question, with two columns to indicate actions for the two possible conditions), and reference to a procedure defined elsewhere (name of procedure in ellipse). To make the example clearer, the corresponding Program Design Language specification is presented in Figure 6b. The Chapin Chart is an improvement over standard flowcharting in some respects, but it is rather cumbersome. It does not appear to be in wide use at present.

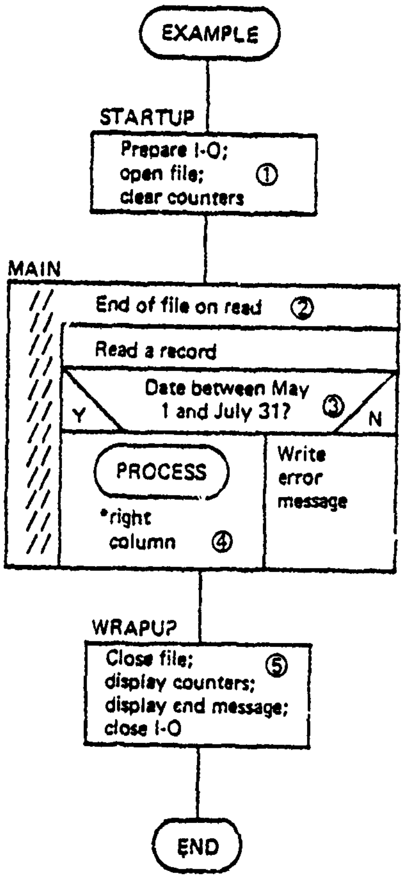


(a) From Bell et al (1977)

(b) From Jackson (1977)

Figure 5. Structure Charts with Added Control Flow Information.

(a) Chapin Chart (from Chapin, 1974)



(b) Corresponding PDL Description

```
EXAMPLE:  PROCEDURE;
STARTUP:
   PREPARE I-O;
   OPEN FILE;
   CLEAR COUNTERS;
MAIN:
   DO UNTIL END OF FILE ON READ;
      READ A RECORD;
      IF DATE BETWEEN MAY 1 AND JULY 31
         THEN CALL PROCESS;
         ELSE WRITE ERROR MESSAGE;
      END;
WRAPUP:
   CLOSE FILE;
   DISPLAY COUNTERS;
   DISPLAY END MESSAGE;
   CLOSE I-O;
END EXAMPLE;
```

Figure 6. Simple Example of a "Chapin Chart", with Corresponding PDL.

The HIPO (Hierarchy plus Input-Process-Output) chart is a graphical documentation technique with a somewhat different emphasis than any of the techniques already discussed. In its most common form (see Figure 7, from Stay, 1977), the HIPO chart identifies the input data elements, the output data elements, and the processing components of a software module, but contains little or no information about data structure. Modular structure is conveyed somewhat implicitly, in that each processing component (e.g., "Validate receipt items") can be further defined by a separate HIPO chart, if desired. Standard HIPO charts contain virtually no flow-of-control information. The primary emphasis in HIPO charts is on data flow, an aspect of the design which is not significantly addressed by the documentation methods previously discussed. Thus, the HIPO chart in the figure indicates that the subprocess, "validate receipt items", receives, as input, both "purchase orders" and "receipts". It produces, as output, "error messages" and "valid receipts". The "valid receipts", along with "price master" information, are used by the second subprocess to produce "gross item price" information, etc.

Although HIPO charts omit some relevant aspects of the design, they are very readable, and may be supplemented by information of other types (e.g., data structure definitions and procedural specifications of the subprocesses). One extension which is receiving attention is the use of a PDL specification within the "Process" block itself, rather than a simple list of process components. This provides flow-of-control information, and makes the nature of the process clearer, but is often limited by space constraints.

The HIPO chart appears to be the only major documentation technique which fully combines data flow information and software structure information in a single figure. When other documentation techniques are used, data flow is often specified in designs by use of a separate table or graph. For example, Figure 3b illustrated the use of a data flow table in conjunction with a structure chart. Directed graphs, or "bubble charts" are sometimes used for this purpose.

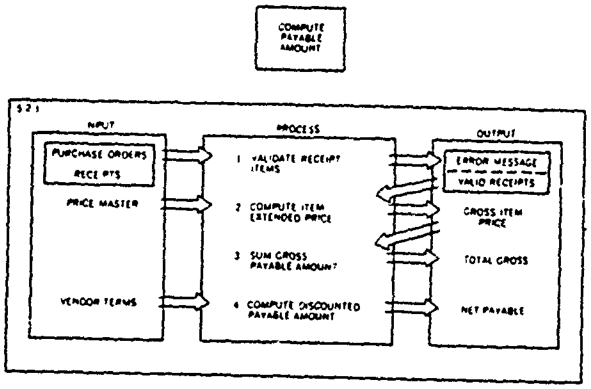
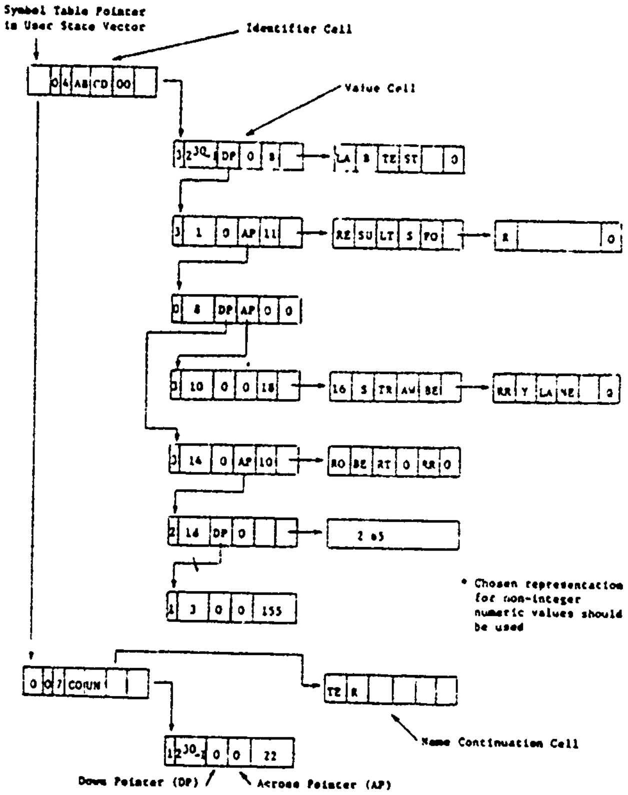


Figure 7. HIPO Chart
(from Stay, 1977).

Data structures can be described either verbally or graphically. Typically, a verbal description is used in high-level design documents in which the emphasis is on the overall data structure (e.g., Figure 3c). Later in the development cycle, when details of record layouts are known, graphical techniques may be employed as in Figure 8 (from Wasserman, 1977). Graphical descriptions are particularly helpful when complex list structures are used, since complex pointer relationships can be indicated by arrows.

In certain specialized application areas, such as programming language processing, specialized and relatively sophisticated design documentation techniques may be used. One such technique is illustrated in Figure 9. This example is taken from the design specification of a FORTRAN-based precompiler (Otey et al, 1978). The specification method involves the use of a formal language grammar which describes the allowable statements in the precompiler language. In the example (Figure 9a), the allowable forms of an "IF...THEN" or "IF...THEN...ELSE" statement are defined. In addition to this syntactic information, the grammar is "augmented" with semantic information which defines the behavior of the precompiler when such an "IF" statement is found. These actions define the meaning or effect of the statement, and are specified with a set of specialized grammar elements and through invocations of a set of "primitive" functions. The primitive functions can be defined in any way convenient for the designer. In the example, they are defined via an informal PDL (Figure 9b). More detailed information about this specification method can be found in Otey et al (1978) or Ramsey (1974).

The emphasis in this specification method is on the flow of control. Modular structure is less important in this application, since all grammar "rules" are recursive -- that is, a rule can invoke itself, either directly or through another rule. Any data structures used in this specification are defined separately.



* Chosen representation for non-integer numeric values should be used

Figure 8. Example of a graphical data structure description (from Wasserman, 1977).

(a) Augmented Grammar Specification for "IF" Statement

```
IF_STATEMENT  :=
      "IF"
      BOOLEAN_EXPRESSION    .ERROR(32)
      "THEN"          .ERROR(33)
      .DO(PUSH OPERATION_TRUE_FALSE_INDICATOR ONTO BOOLEAN_TRUTH_STACK)
      ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "FALSE")
        .SCAN_TO_LOCATION(1)
      ! .TRUE )
      STATEMENT
      .LOCATION(1)
      .DO(POP TOP ELEMENT FROM BOOLEAN_TRUTH_STACK AND SAVE AS
               OPERATION_TRUE_FALSE_INDICATOR;)
      ( "ELSE"     /* OPTIONAL ELSE CLAUSE */
        ( .TEST(OPERATION_TRUE_FALSE_INDICATOR = "TRUE")
          .SCAN_TO_LOCATION(2)
        ! .TRUE )
        STATEMENT
      ! .TRUE )
      .LOCATION(2) ;
```

(b) PDL Specification of Primitive Functions

```
                    FLTOPS PRIMITIVES

 .SCAN_TO_LOCATION (ARGUMENT):
       /*************************************************************/
       /* DISABLE ALL FLTOPS OPERATIONS EXCEPT BASIC FCF PARSING,   */
       /* UNTIL CORRESPONDING LOCATION (IDENTIFIABLE BY NUMERICAL   */
       /* ARGUMENT) OF SAME INVOCATION OF SAME RULE IS REACHED.     */
       /*************************************************************/
       IF FLTOPS FUNCTION = SYNTAX OR SCAN THEN RETURN;
          ELSE DO;
             SAVE FLTOPS FUNCTION VALUE AND RESET FUNCTION TO SCAN;
             SAVE INFORMATION INDICATING THE RULE AND LOCATION AT WHICH
                FLTOPS FUNCTIONS WILL RESUME IN LOCATION_INFORMATION;
             END;
       END .SCAN_TO_LOCATION;

 .LOCATION (ARGUMENT):
       /*************************************************************/
       /* SEE .SCAN_TO_LOCATION.                                    */
       /*************************************************************/
       IF FLTOPS FUNCTION NOT EQUAL TO SCAN THEN RETURN;
       IF CURRENT LOCATION CORRESPONDS TO THAT INDICATED IN LOCATION_INFORMATION
                SAVED BY THE LAST INVOKED .SCAN_TO_LOCATION PRIMITIVE
             THEN RESET FLTOPS FUNCTION TO SAVED VALUE;
       ELSE RETURN;
       END .LOCATION;
```

Figure 9. Use of Augmented Language Grammar and Supporting Primitives to Describe a Precompiler.

It should be clear, from the material presented in this section, that there are a wide variety of design documentation techniques in common use (many less common techniques were intentionally omitted from this discussion). These techniques may be verbal or graphical, and they differ in their relative emphasis on modular structure, flow of control, data structure, and data flow, as summarized in Figure 10. Although there is a good deal of personal preference and "institutional inertia" involved in the selection of documentation methods for particular projects, it is also clear that no one documentation technique is appropriate for all types of design. The selected technique(s) must convey the salient features of the design. Any of the basic classes of design information may be highly relevant or even irrelevant to some class of design problem.

Documentation Technique	Emphasis On:			
	Modular Structure	Flow of Control	Data Flow	Data Structure
Flowchart	Low	High	-- (a)	--
Structure Chart	High	-- (b)	-- (b)	--
Structure Chart with AND/OR Logic	High	Low	--	--
Jackson Chart	High	Low	--	--
Program Design Language	High	Medium	--	--
Chapin Chart	Medium	High	--	--
HIPO Chart	Low	--	High	--
HIPO Chart with PDL	Medium	Medium	High	--
Data Flow Table	--	--	High	--
"Bubble Chart"	--	--	High	--
Data Structure "PDL", Graph	--	--	--	High
Augmented Grammar	Medium	High	--	--

(a) Dashes (--) indicate categories of information entirely omitted by documentation technique.

(b) Often has limited control flow, data flow information added, in the form of arrows and labelled arrows, respectively.

Figure 10. Types of Design Information Conveyed by Various Documentation Techniques.

DESCRIPTIONS OF THE FORMAL DESIGN METHODOLOGIES

This section contains descriptions of the four formal software design methodologies which were surveyed in this study. For the most part, evaluative comments are deferred until the next section of the paper. The amount of descriptive information provided here is a function of the level of detail to which the approaches have been developed. For example, there is more detail here about "Structured Design" than about other approaches because the published literature describing the approach contains more detail.

Before proceeding, the reader should be aware of several terms which will be utilized in discussing the design methods. These terms are concerned primarily with evaluation functions, and represent concepts relevant to many of the design methodologies, although specialized terminology may be used for these concepts in individual methodologies:

(1) INDEPENDENCE is a qualitative, and usually subjective, assessment of the degree to which the design of one module is unaffected by the design of other modules. In general, a modular decomposition which maximizes independence is desirable.

(2) FUNCTIONAL COHERENCE is a qualitative assessment of the degree to which the components of a module are related to one another and to the accomplishment of a single simple objective. In general, a modular decomposition which maximizes functional coherence is desirable.

(3) FAN-OUT refers to the number of immediate descendant modules possessed by a module.

(4) FAN-IN refers to the number of modules (in a non-hierarchic modular structure) from which a single module is descendant.

(5) SCOPE OF CONTROL refers to the set of modules formally under the control of a particular module. Module B is in the scope of control of module A if A may directly or indirectly invoke the execution of B. Generally, the scope of control of a module includes the module itself and all of its descendants.

(6) SCOPE OF EFFECT refers to the set of all modules whose functioning can be affected by the behavior of a particular module. This set is not necessarily restricted to the module and its descendants, but may include any other module to which data are passed. For example, error flags might be used to allow a module to control other modules outside its formal scope of control.

STRUCTURED DESIGN

The first approach we will discuss is "Structured Design" as advocated by Stevens, Myers, and Constantine (1974), Myers (1975), and Yourdon and Constantine (1975). This approach involves an initial consideration of data flow followed by decomposition of the system under design into subparts. These stages are applied interactively, and rather freely, as they are required to achieve greater detail. At the higher levels, modules are described in terms of their functional effect on the data. The emphasis is on functional coherence of the modules and little attention is normally paid to the flow of program execution.

Overview

In Structured Design, the designer begins with a functional description of an overall system: its objectives, requirements, inputs, and outputs. The designer uses this information to determine the data flow of the system. The data flow definition is then used to determine preliminary module boundaries. From an analysis of data flow and identification of preliminary module boundaries, a functional description of both modules and interfaces is produced. Structured Design uses a FLOW DIAGRAM or BUBBLE CHART to represent data flow and assist in identifying module boundaries. It then uses a STRUCTURE CHART to depict the modules and their interfaces.

Depending on the properties of the system's data flow, two major classes of design problems are recognized and a unique problem-reduction operator is associated with each class. These operators are TRANSFORM ANALYSIS and TRANSACTION ANALYSIS. The principal evaluation criteria, which are used with both forms of analysis, are COUPLING (independence) and COHESION (functional coherence).

The technique called TRANSFORM ANALYSIS is used when it becomes apparent that a problem decomposition produces an AND relationship among the parts. This means that the initial task requires the performance of subtask one, AND subtask two, AND so on, for each execution of the initial task. The other possibility is TRANSACTION ANALYSIS. In this case, each execution of the initial task only requires the performance of subtask one, OR subtask two, OR one of the other subtasks. This is called an OR relationship among the subtasks.

Transform and/or Transaction analysis are used, as appropriate, along with a set of design heuristics, to iteratively decompose the design into detailed module specifications. A design is complete when a set of modules and their interfaces have been specified in sufficient detail that the designers are convinced of three things:

1. Implementation of any module is a well defined task with little impact on the implementation of any other module if all are implemented as described.
2. Correct performance of any module (including the root module and, therefore, the system itself) depends only upon the correct performance of all modules to which it fans out.
3. All modules required are defined so as to be straightforward coding tasks, requiring in the neighborhood of one or two pages of source code.

Detailed Discussion

The first step in Structured Design, whether transform analysis or transaction analysis is to be used, is the restatement of the design problem in terms of the high-level functions that will be involved, rather than the procedures required to accomplish those functions.

As an example, consider a patient-monitoring system for a hospital. This system should monitor, through an analog device, various physiological readings and notify the nurses' station if any readings are outside of a specified range. A FLOW DIAGRAM or BUBBLE CHART to represent this problem is shown in Figure 11.

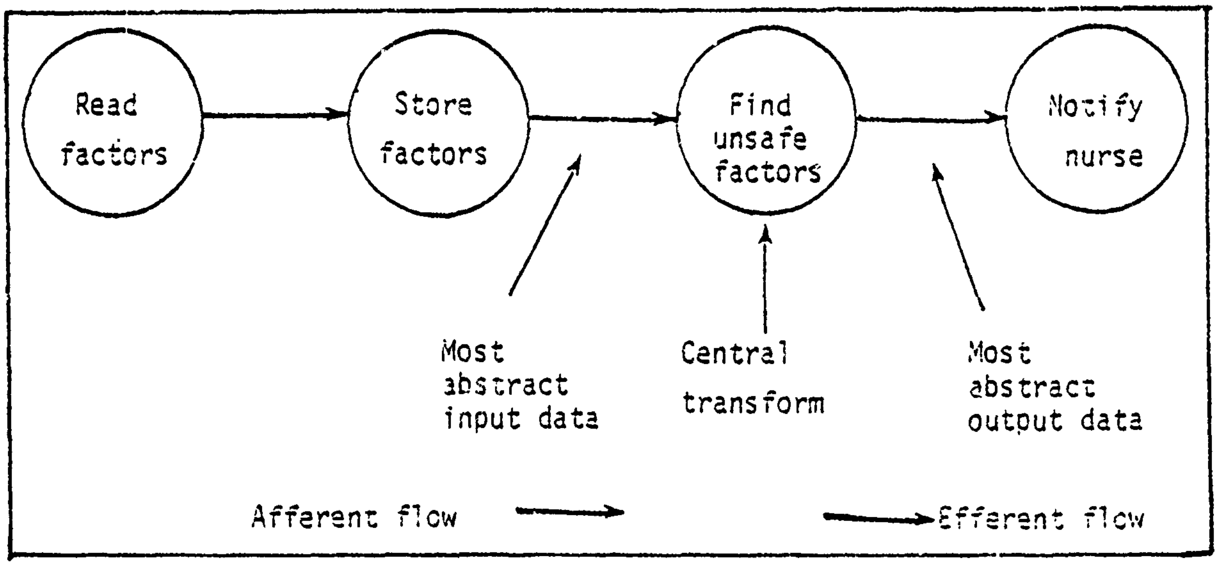


Figure 11. A Simple Flow Diagram (adapted from Myers, 1975).

Transform Analysis

When the initial levels of decomposition produce an AND type of hierarchy, such as that shown in Figure 11, TRANSFORM ANALYSIS is suggested. Unless experience with a particular problem area dictates otherwise, this is the suggested way to begin any Structured Design. The following six steps outline the general operations employed in design using the transform analysis strategy.

1. The system is described in terms of its major functional components (see Figure 11).

2. Conceptual input and output flows of data are identified from a functional standpoint. These streams may separate and/or combine. They are physical where they enter and leave the system, but become more conceptual, or abstract, farther in.

3. The major conceptual flow of data is identified from input toward the CENTRAL TRANSFORMS and backward from output to the CENTRAL TRANSFORMS. For both input and output the "point of highest abstraction" is identified. This is the point where it enters and leaves the CENTRAL

TRANSFORM REGION. (Steps 2 and 3 are also indicated in Figure 11.)

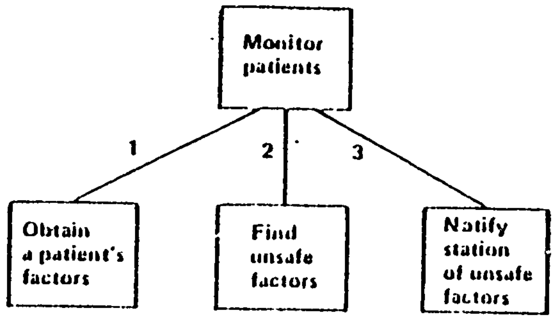
4. Using the above, generally depicted as a bubble chart showing flow, combination, separation, and transformation of data, a structure is designed. This structure depicts modules that are identified as sources, sinks, and transformers of data. The function of each module and its interfaces should be briefly described. The initial design for the problem introduced in Figure 11 is shown, as a STRUCTURE CHART, in Figure 12. The module identified as "obtain a patient's factors" is a source (or afferent) module (it requires no inputs from other modules), "notify station of unsafe factors" is a sink (or efferent) module (it produces no outputs) and "find unsafe factors" is the transform module.

   The box below the modular structure specifies the data flow within the design. This information could also be represented directly on the arcs connecting the modules (as done by Yourdon and Constantine), but we will use this type of notation (as used by Myers) primarily to increase clarity.

5. Breadth first, and level by level, each of the modules is expanded into subfunctions using all of the preceding steps. The purpose of this expansion is to identify a structure that contains the most peripheral source and sink modules and contains all of the functions necessary to support the data flow described in Steps 1 to 3.

6. A list of design heuristics, including great attention to module independence and functional coherence, is used to aid decomposition, at levels of greater detail, of the transform modules obtained in Step 5.

A complete structure chart for the system considered in this section is illustrated in Figure 13.



	In	Out
1	Nothing	TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	List of unsafe factor names and values
3	PATIENTNUM and List of unsafe factor names and values	Nothing

Figure 12. Initial Structure Chart for System of Figure 11 (from Myers, 1975).

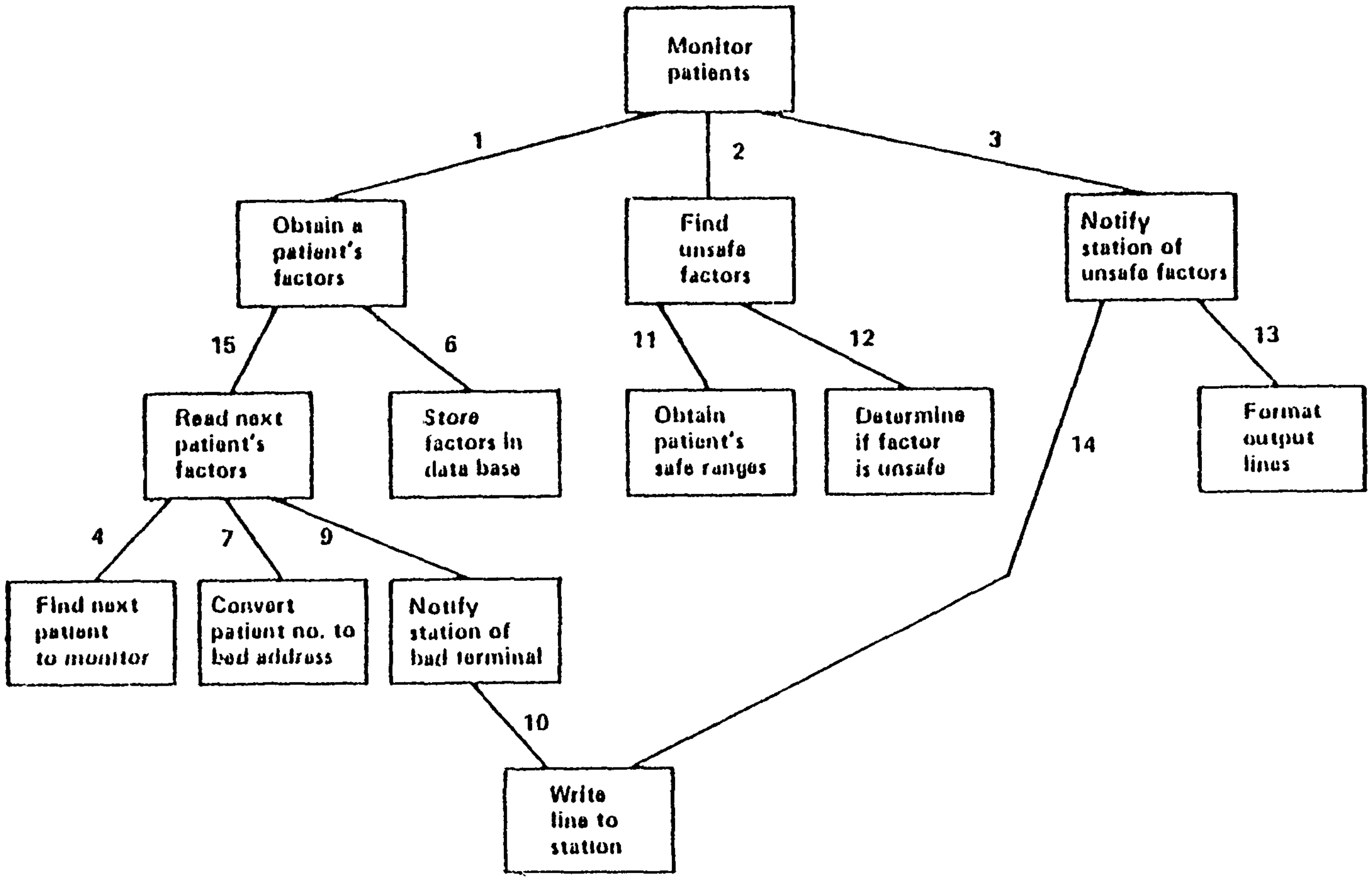

Monitor patients
1
2
3
Obtain a patient's factors
Find unsafe factors
Notify station of unsafe factors
15
6
11
12
13
14
Read next patient's factors
Store factors in data base
Obtain patient's safe ranges
Determine if factor is unsafe
Format output lines
4
7
9
Find next patient to monitor
Convert patient no. to bed address
Notify station of bad terminal
10
Write line to station

Figure 13. Complete Structure Chart and Data Flow Table for Problem of Figure 3 (from Myers, 1975) (Page 1 of 2)

	In	Out
1	-------------------	TEMP, PULSE, BP, SKINR, PATIENTNUM
2	TEMP, PULSE, BP, SKINR, PATIENTNUM	List of unsafe factor names & values
3	PATIENTNUM &list of unsafe factor names & values	-----------------
4	-------------------	PATIENTNUM
6	PATIENTNUM,TEMP, PULSE, BP, SKINR	-----------------
7	PATIENTNUM	BEDNUM
9	BEDNUM	-----------------
10,14	LINE	-----------------
11	PATIENTNUM	TEMPR, PULSER, BPR, SKINRR
12	FACTOR, RANGE	UNSAFE
13	List of unsafe factor names & values	List of lines ----
15	------------------	TEMP, PULSE, BP, SKINR, PATIENTNUM

Figure 13. (Concluded)

Transaction Analysis

When Steps 1 to 3 of Transform Analysis produce a data flow graph with a transform that splits an input data stream into several discrete output streams, then TRANSACTION ANALYSIS is suggested. Such a data flow graph is illustrated in Figure 14. In this case, "T" is the TRANSACTION CENTER and the "⊕" symbol is used to indicate that "W", "X", "Y", and "Z" are all disjunctive (an "OR" relationship). This structure should be compared with the linear ("AND" relationship) shown in Figure 11.

Another way in which TRANSACTION ANALYSIS might be suggested is when several different processing states treat the same set of input data in different ways. When the system is better conceptualized as a recognizer and dispatcher of different information sets to different subfunctions, an analysis of these transactions is the suggested design strategy. Naturally, the transaction structure will also consist of components, and the design of any subcomponent may well return to the transform analysis strategy. Using transaction analysis, the following steps are taken:

1. The sources, both data and preconditions, of each transaction are identified.
2. A structure is identified that separates the functions of transaction, identification, analysis, dispatching, state chanqe, and transformations.
3. The different transactions and the processing that each triggers are identified.
4. Ways to functionally combine processing tasks are given careful consideration, but only after Steps 1 to 3.
5. A module is specified to process each transaction or other functional task that has been identified.

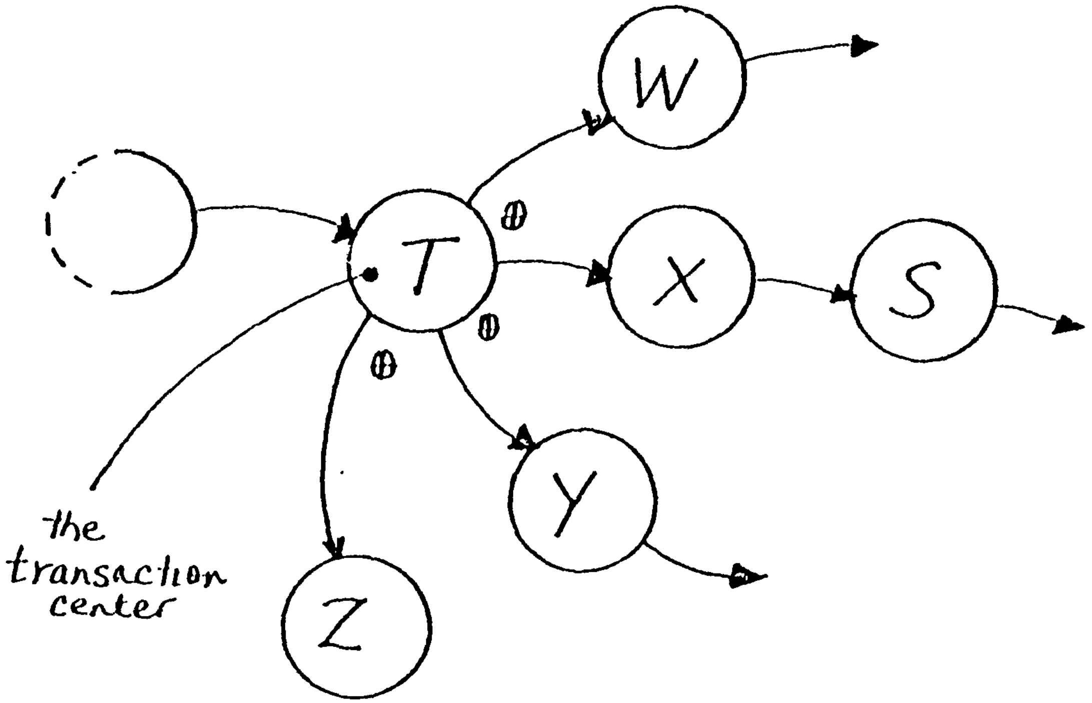


Figure 14. A Data Flow Diagram Illustrating an "OR" Relationship (from Yourdon and Constantine, 1975).

6. Levels of greater detail are reached in two steps:

   a. First each transaction module is subdivided into action modules,

   b. Then each action module is described in detailed steps.

   Similar actions and common detailed steps may be shared by these two levels.

As in Transform Analysis, the design is represented as a structure chart and the principal evaluation criteria are coupling and cohesion.

Evaluation Criteria

Structured Design is primarily a collection and description of a set of evaluation and guidance heuristics. The two primary criteria by which module boundaries are initially defined,and by which modules and their interfaces can continue to be judged, are independence and functional coherence. Each of these will be described in structured design terms.

First, however, a perspective on these measures should be introduced. The measures are subjective evaluations, intended to be made by experienced designers on their own or others' designs. When a module is being judged, the interfaces of the module, its scope of effect, and scope of control are all important. However, it is also important to consider how, and how well, these things are contained in the module's parent module. Likewise, especially in the determination of functional coherence, the components of a module and their relationships should be examined.

Coupling -- COUPLING is the term used to denote independence in the structured design literature. The objective of minimizing coupling among modules is intended to maximize the independence of modules.

Six categories, or levels, of coupling are possible. When evaluating the coupling of a module it is considered pairwise with all other modules to which it is related in any manner. The principle underlying this criterion is that coupling should be minimized and that certain types of coupling are preferable to others. Furthermore, many of the design heuristics to be described later place additional restrictions on module coupling.

The categories, or levels, of coupling are described below in the order of more desirable to less desirable.

1. DATA Coupling occurs between two modules that both reference the same data variable(s) or structure(s). In this case, the data are local to the coupled modules and inaccessible to others.

2. STAMP Coupling arises when the mechanism for making data available to a limited set of modules does exclude others from access. For example, named COMMON in FORTRAN.

These two types of coupling are distinguishable only in certain implementation environments. Certain parameter passing mechanisms are such that either description might apply. In general, the more advanced programming languages have efficient mechanisms for restricting the sharing of information to parameter passing without either form of coupling, or to data coupling only.

3. CONTROL Coupling describes the relationship that exists when the results of an operation in one component are used to direct processing in another component. This type of coupling is worst between seemingly unrelated components. Between a module and one of its immediate subcomponents, control coupling is more acceptable. If at all possible, however, it should be limited in the direction of cause and effect so that module controls component rather than the reverse.

4. EX1ERNAL Coupling occurs by way of a mechanism that allows one module to declare free access by other modules to its contents. Generally, an assembler language must rely on this type of coupling because more sophisticated mechanisms are unavailable.

Clearly, some form of control and external coupling must exist between any two related modules. However, when the passing of control is implicit, or along well established lines, and the permission and access conventions for data are disciplined and adhered to, then coupling is properly minimized. When mechanisms exist that allow unconventional coupling, then the job of evaluation becomes more difficult.

5. COMMON Coupling is made possible by mechanisms such as FORTRAN's blank common or uncontrolled use of "global" variables. Assembler and machine languages usually have no mechanism to prevent it. It occurs when data sharing and communication are obscured through the use of completely uncontrolled channels.

6. CONTENT Coupling occurs when one module refers to something in the domain of another with no explicit permission by the second module.

Both of these types of coupling arise through a lack of convention or mechanism for communication between modules. In certain environments the only way to avoid dependencies of these kinds is to make the necessary conventions and mechanisms a part of the design.

Cohesion -- A design may be decomposed into a set of modules which are highly independent but which are still low on the scale of functional coherence, or COHESION. Independence (coupling) only requires that a sharp and well defined line be drawn between each module and all others. Functional coherence addresses what a module does and why it does it. Perhaps the single most effective measure of functional coherence is the number of words it takes to describe the module in the terminology of the design. Yourdon (1976) suggests that "The function of a maximally cohesive module can be described in one sentence with a transitive verb and a single, nonplural object."

The following descriptions are used to more closely describe the cohesion of a module. These descriptions are meant to apply to the components of a module, why they were brought together, and what they accomplish as a whole. Generally, when one description applies, those lower in the list will also apply, but only the first is used for evaluation. The descriptions are seven points on the structured design scale of cohesion. The scale is not linear. A large gap exists between numbers one and two, and another large gap between five and six. High cohesion is achieved when the relationships among components of a module are near the beginning of the scale.

1. Structured Design assigns highest cohesion (called FUNCTIONAL cohesion) to a module whose components combine to form a conceptual unit. This is why the number of words to describe it is a powerful test. However, in a complex design even a tightly bound conceptual unit may be very context dependent. Modules found in programming libraries are generally those that are high in both independence and in cohesion.

2. SEQUENTIAL cohesion occurs when a module performs more or less than a single unitized function and is bound together by the fact that its components are sequential steps that go beyond such a function, or make up only part of such a function. In this case there must still be a close association between the module and a unitized function.

3. When a module contains a set of components that are primarily related to each other because they share a common set of input and/or output data, the module's strength or cohesion is COMMUNICATIONAL in nature.

4. If a module performs a set of functions which are related only by virtue of being steps in a procedure, but the module does not perform a complete function, it has PROCEDURAL cohesion.

5. When data structure does not bind the components of a module together, they may be bound by a common occurrence with respect to time. This is called TEMPORAL cohesion. "Initialization" is a term that lacks conceptual unity, and an initialization module would usually fail the tests of functional, sequential, and communicational cohesion. "Initialization" describes a temporal binding. Thus the major description of a module, the one most characteristic of its purpose, is the one to be analyzed in determining its level of cohesion.

6. A module is said to have LOGICAL cohesion when it is bound together on the basis of some logic process shared by all of its components. The module may be characterized by some descriptive phrase, but not one that has unity in terms of function, sequence, communication, or time, (for example, "recover from error"; or "initialize device" when several devices exist). If the higher forms of cohesion are missing, the components included in such a module are going to be coupled together on a basis that is very likely to change.

7. When none of the above describes the connection between components of a module it must be assumed that they are there by coincidence (COINCIDENTAL cohesion). Such a module can seldom be described in any simple terms. For reasons of optimization, or other considerations that take place during implementation, a code module might very well fall into this category. However, a design module never should.

Structured Design Heuristics

Much of the literature on Structured Design is devoted to the presentation of heuristic guidelines. These guidelines are not unique, each is compatible with the other design methodologies, and each has appeared many times in the literature of the last ten years. Nor is the list complete; it is intended rather to explain some of the most important concepts in this area of Structured Design. It reads like a list of DOs and DON'Ts.

1. Maximize the independence of each module by reducing its coupling to other modules. For modules that are coupled as components of a larger module, maximize the quality of their coupling.

2. Maximize the functional coherence of each module by insuring that its components are related on the basis of a conceptual unity. Strive for the attributes high on the list of both cohesion and independence.

3. Match the structure of the design to the structure of the problem. Input-Process-Output is often the most economical structure.

4. Keep the size of a module in the neighborhood of from 10 to 100 lines of code or description.

5. Keep the scope of effect of a module within its scope of control.

6. Try to eliminate the need for error flags, especially among several levels of callers.

7. Provide a simple solution to the immediate problem; do not attempt to generalize.

8. Reduce complexity by isolating dependencies and assumptions, as by "hiding" them within a minimum of modules.

9. Eliminate duplicate functions, but not duplicate code. Strive for a set of unique modules with high individual cohesion.

10. Strive for a "Mosque Shaped" design.

    a. Check unusually high fan-out or fan-in to determine if a level is missing.

    b. Strive for a fan-out in the neighborhood of from 3 to 9 modules.

    c. Early in the design err on the side of too much fan-out.

    d. Later in the design err on the side of too much fan-in.

11. Keep the number of parameters to a module small, but keep each item separate.

12. Within the design framework, allow for the production of:

    a. Documentation, keeping separate where the design is going, where it is, and what has been "finalized."

    b. Support tools and project conventions that are necessary to enhance the implementation environment so that the methodology can actually be carried out.

    c. Module libraries that provide for an exchange between projects and encourage the production of highly independent and coherent modules.

THE JACKSON METHODOLOGY

Overview

Using Jackson's (1975, 1977) methodology, the designer begins by specifying input and output data structures. The data structures must be hierarchical, and are constructed using only sequence, iteration, and selection. With two basic exceptions, the modular structure of the software will be developed to correspond fully to the data structure, with decision points corresponding to the iteration and selection points in the data structure. The exceptions occur when input and output data structures do not correspond ("program inversion" is used here) and when the information required for a decision is not available at the point at which the data structure requires the decision (commitment to one alternative is used, with "backtracking" as necessary).

Detailed Discussion

The Jackson methodology is advanced as a rational and teachable methodology that is practical and does not depend upon the insight or inventiveness of the designer. This necessarily places a limit on the size and complexity of designs to which it is addressed. So far, its orientation seems to be toward the commercial data processing shop and the applications programmer. It is most clearly suited to the design task facing an individual programmer who has been given a module to code. In addition, it requires that the problem statement be in terms of well defined data structures such as those available in an ongoing data processing environment, or perhaps those defined within the context of a larger project.

Given that the scope and complexity of the problem match these limitations, the Jackson approach rests on a single fundamental assumption and provides three alternate strategies for the design process. The assumption is that both procedural code and data for any given problem can be adequately described by a hierarchy of three structure components. These components are sequence, iteration, and selection.

These constructs are familiar in the context of program flow of control, and there is fairly widespread agreement that they are an adequate and appropriate set of constructs in that context. Because Jackson's approach is predicated on a one-to-one correspondence between data structure and program (design) structure, this methodology also assumes that these three constructs are adequate for describing the logical data structure. The data structure is assumed to be expressible as a serial file whose internal logical structure can be described with a hierarchy of these operators. Thus, a simple file might have a header record followed by (sequence) a group (iteration) of one or more records of either (selection) type A or type B.

A Jackson design problem is first completely specified in terms of its input and output data structures. The data structures are defined as hierarchical decompositions of serial files. This means that for each input and output stream a hierarchical structure will be used to completely specify it. The terminal elements in each hierarchy are single data items or item types. They are joined together by sequence, iteration, or selection into larger units that can be called records, lines, pages, or whatever. The intent is that each joining produces a higher-level conceptual unit of data.

When the data structures are defined, the program structure is created from them. The purpose of the program, of course, is to produce the mapping of input data to output data. It is expected that for each sequential, iterative, or selection component of the data structure there will be a corresponding program component.

Input-to-output mappings fall into one of three categories, according to Jackson. For each of these there exists a design strategy. The mapping may be straightforward, involving no data inversions or other abnormalities. When this is the case a direct correspondence between components of the data structure and components of the program structure is expected. The other two categories (not mutually exclusive) are caused by "structure clashes" and the need for programmatic backtracking.

Structure Clashes

A STRUCTURE CLASH occurs when an input-to-output mapping requires reordering, recombination, or synchronization of input item to output item. A situation of this type exists whenever the input and output data structures do not correspond all the way down their hierarchic structures to the lowest data element. As a very simple example, consider a program which must read in a matrix by row and write it out by column. The technique used to resolve such problems is called PROGRAM INVERSION. This may take several forms, involving intermediate files, coroutines, or most commonly, subroutines. In each case, though, the incompatible structures are isolated from each other by utilizing two program modules which communicate in terms of logical data elements not tied to either structure. This communication is made possible by a decomposition of the input data into logical data elements by one module, and their recombination into an output structure by the other.

Backtracking

Anotner technique often required when using Jackson's Methodology is BACKTRACKING, which refers not to the design process, but to an action which will be taken by the system under design. As a simple example, suppose that the system is to read a batch of cards, and cannot determine whether the batch is of type A or type B until the entire batch has been read. Because of the basic requirement that the system's modular structure correspond directly to the hierarchic logical structure of the input file, the "look ahead" approach is somewhat incompatible with Jackson's methods. He prefers, instead, to commit to one alternative or the other at the outset, thus preserving an exact correspondence of "select" points in the modular design and in the input data structure. When the selected alternative proves to have been incorrect, backtracking is employed to recover from the incorrect decision and implement the correct one.

Backtracking, then, is handled like ordinary selection, but in advance of some of the decision criteria. One alternative is chosen and processing is directed down that line. The alternative to be processed is selected by the designer for a combination of reasons. It may be the only case the designer initially considers. It may be the one most easily recovered from. Or it may be the most probable one. In any case, when the backtracking situation is identified, it is designed in three phases. First, the alternatives and criteria for their selection are listed, and one is chosen. Second, in the processing of the default alternative the selection criteria for the other alternatives are encountered. These are "quit conditions" for the default, and are used to select their own alternative. Third, there may be side effects of having processed the default. These may have to be undone and restored, or they may contribute to the processing of the new alternative.

Jackson's Procedure

Considered in more procedural detail, the Jackson methodology proceeds through three phases. First, the data structures are defined. The designer looks for correspondences between the structures, seeking relationships between the components of one structure and those of another. When correspondences cannot be found, structure clashes are identified and an appropriate program inversion strategy is selected.

Next, the program structure is created on the basis of the correspondences found in the data structures. Through the examples he gives, Jackson's approach seems to require a program structure with the following characteristics:

1. Each module is based on a correspondence between data structures and is, therefore, likely to be a transformation in a data flow pathway.

2. Each module is only described by a simple designator and is, therefore, likely to be a conceptual unit based on its function.

3. Module boundaries correspond to subdivisions in the data structure, increasing the probability that they are bound on the basis of functional, sequential, or communicational unity.

4. Jackson's structure notation provides no facility for describing module interfaces. This may encourage the designer to keep them simple or it might be a source of problems. There is, however, no apparent reason that a separate interface table (as in Figure 11) could not be used in conjunction with this notation.

5. The structure notation (Figure 5b) provides only the program logic of sequence, iteration, and selection. This lends itself most readily to a block-structured program during implementation.

The third stage is that of listing program steps. This corresponds to design on a level of greater detail than the second stage. The program steps required to produce the input-output data transformations are listed and assigned to program components. Program steps are classified as primary and secondary. Primary operations have to do with logic and data transformation. Secondary operations are those associated with reading, writing, and moving about in the data structure. Program steps are assigned to program components on the basis of their class with respect to the function of a module, and on the basis of where they fit with respect to sequence, iteration, and selection.

The extensive attention paid to evaluation techniques and heuristics in Structured Design is unnecessary in Jackson's approach. As long as the design problem meets Jackson's initial criteria and fits into the intended scope and complexity, it seems that the function of the heuristics is built into the design procedure and that, in most cases, the method should produce designs with the characteristics of independence and functional coherence. Whether the development of these designs is easy or difficult is another matter which will be considered in a later section.

THE "HIGHER ORDER SOFTWARE" APPROACH

Th'. Integrated Software Development System/Higher Order Software (ISDS/HOS) approach (Hamilton and Zeldin, 1976b) is included here as a representative of several methodologies developed especially for use in very large software design and development efforts. These methodologies are all broader in scope than the simple design methodologies already discussed. Although their properties vary, they tend to: (1) make significant use of automated tools, (2) address software development from the software design phase all the way through implementation including, in some cases, testing, and (3) concentrate heavily on the "validation and verification" aspects of software development.

Although these approaches are not specifically design methods, they do provide varying degrees of guidance to, and constraints on, the designer. In addition to ISDS/HOS these methods include: (1) The Software Requirement Engineering Methodology (SREM; Bell, Bixler, and Dyer, 1977) developed for the Army Ballistic Missile Defense Advanced Technology Center (this is the most heavily automated methodology of this group); (2) The Software Development System (SDS; Davis and Vick, 1977) developed at the Army Ballistic Missile Defense Advanced Technology Center; (3) The Information System Design and Optimization System (ISDOS; Teichroew and Sayani, 1971) developed by Teichroew and others at the University of Michigan; and (4) The Structured Analysis Design Technique (SADT; Ross and Schoman, 1977) developed by Softech (the least automated and formalized of the group).

These methodologies are all oriented toward the structuring of design and management of large software design efforts. They all pay particular attention to the early stages and the formal specification of requirements. These methodologies came into existence as a result of experiences gained on large projects and are attempts to remedy the major problems encountered by such projects. There is wide agreement that most failures in large projects during the detailed design, implementation, and integration stages are the result of poor requirements specification and the failure to maintain an overall project coherence.

Because they are concerned with large design projects, the "macro methodologies" specifically address the following topics:

a. Design tracking.

b. Visibility and impact of changes.

c. Control and management of the design with respect to project personnel.

d. Automating design and implementation tasks.

e. Establishing a framework and set of standards that prohibit certain problematic courses of action.

The larger the system under design, the more important are the design environment and the right management elements. Peters and Tripp (1977) concluded their review with the observation that "successful application (of methods) occurs only in supportive environments." Certainly, the documentation of these methodologies contains much on the subject of management guidelines, but there is little departure from, or addition to, the standard literature on management techniques. What sets these methodologies apart, especially ISDS/HOS, is that specific ground rules are defined based on a formal model.

Overview

"Higher Order Software" is the name of an overall approach to software development (and, incidentally, the name of the company which has developed the approach). The Integrated Software Development System (ISDS) is a system of software development tools used in conjunction with this approach. The designer who uses this approach is thus affected by the overall HOS philosophy, by the HOS model, and by the constraints and aids of ISDS. Within ISDS, there is a language (AXES) which is to be used by the designer to specify the requirements of the system under design. In specifying these requirements, the designer is assisted by ISDS, with which he or she interacts.

The HOS model contains a number of definitions, axioms, and decomposition rules which specify the allowable relationships between a module and its submodules, between a module and its input and output data, and generally, between pairs of modules which communicate with each other. As will be seen, the rules which must be followed by the HOS designer are more formal and more constraining than those of, for example, structured design. They are also rigidly enforced by the software aids.

HOS System Development Model

Before discussing ISDS/HOS as it affects the design phase, it is appropriate to introduce the reader to the overall plan and philosophy of the methodology. The HOS approach involves four phases:

1. Concept Formulation
   A complete set of system requirements is determined. Both mandatory and candidate requirements are considered. The designer draws up the functions to comprise the target system in the form of a "control map" not unlike the structure chart used in Structured Design. In this process, the designer records all questions that arise and sets up a standard way of recording the answers as they are found. Several iterations of this produces the finalized specification of the system. This final specification is defined in AXES, the formal language used in the methodology. The specification defines the target system, development standards and processes, and support systems and tools.

2. Program Validation
   This phase is based on the AXES specification. It consists of a detailed analysis of the system functions and interfaces. High-level resources are allocated to the top layer of system functions. Simulation performance testing (manual or automatic) is done to study fault tolerance, error detection, timing and accuracy, security requirements, and other aspects of target system reliability. Development and support systems that must be built are identified, scheduled, and begun.

3. Full-Scale Development

The inputs to this phase are the system, completely specified in AXES, and a set of resources out of which the target system is to be built. This phase, like the preceding two, is iterative. Each iteration involves trial allocation of resources and analysis of system hardware-software subsets. Any changes discovered in this process may cause iteration back into either of the preceding phases, or just another iteration of this phase. A resource allocation tool (RAT) is used in this phase. This tool is envisioned to eliminate the manual allocation of computer resources to functional components of the system.

During this phase the system specification control map is reconfigured into a standard architectural form. This form is hardware independent and contains as few levels as possible. It is intended to re-express the resource requirements of the system to the designers in an understandable format.

The resource allocation tool uses the architectural form to analyze the target system (control map specification) in terms of time and memory optimization. Given specific time, memory and other implementation constraints, an optimal software module configuration is generated. Automation of this process is expected to allow even the details of a particular machine to be specified and the automatic production of executable code from the machine-independent specification.

4. Production and Deployment

In this phase the target system is placed into actual use. Manuals are prepared. Feedback from initial training and use of the system may cause iteration back into the previous phase. When the target system is acceptable, it is produced and delivered to the field.

Axioms and Decomposition Techniques

In HOS the designer develops program structures in accordance with a formal set of design axioms and structural decomposition techniques, as discussed below:

1. A module is the root node of a family of functions (nodal family).

2. A module's corresponding function is the functional transformation performed by a nodal family that maps an element of the module's input space (domain) into an element of its output space (range).

3. A nodal family is constructed on the basis of rules (called AXIOMS) which are summarized below:
   a. A module has the ability to invoke and control the sequencing of only those functions which are its immediate offspring (Axioms 1 and 6).
   b. A module controls the access rights to the variables in the input and output space of each function it may invoke (Axioms 3 and 4).
   c. A module controls the rejection of invalid elements of only its own input space (Axiom 5).
   d. A module controls the responsibility for the elements of only its own output space (Axiom 2).

4. A function is decomposed into its immediate offspring according to three structures implied by the above axioms.
   a. COMPOSITION. A module may be composed of two or more functions invoked in a particular sequence. In this case, the first function has an input space, and the last an output space, identical to the input and output space of the parent module. Intermediate results are passed as output from one function and input to the next.
   b. SET PARTITION. The domain of a module (its input space) may be partitioned so that different variable values are assigned to different offspring functions.

c. CLASS PARTITION. When a module has more than one variable in its input space, class partitioning may be used such that different sets of variables are associated with different offspring functions.

The axioms of ISDS/HOS appear to be very similar to the "coupling" principles of Structured Design, although they are expressed much more formally in ISDS/HOS. In particular, ISDS/HOS requires adherence to these independence rules, whereas Structured Design stipulates only that they are highly desirable. This probably results mostly from the intention that ISDS/HOS generate designs which are susceptible to automated analysis and "validation and verification." It should be noted, though, that the axioms and decomposition techniques of this methodology can be applied in the absence of any such automated tools, if desired.

The decomposition techniques are somewhat novel. They are more formal than those of Structured Design and are much more constraining. They provide a very restricted set of decomposition "move" types which can be made by the designer, but relatively little guidance is provided for determining precisely how to select and formulate one of these "moves" appropriately. In relatively simple situations, such as the examples provided in the documentation of this approach, selection of an appropriate modular decomposition may be quite straightforward. In the more complex situations which are the primary reason for existence of the "macro methodologies," more guidance, perhaps in the form of design heuristics, may be needed. If the designer succeeds in applying this restricted set. of decomposition techniques, it would appear that high functional coherence will result almost automatically.

A final aspect of ISDS/HOS which may exert a strong influence on the design task is the language, AXES, which is intended to allow detailed description of the design. Although most of the purposes of AXES are those of design documentation, the language is also intended to provide the working medium for design development. The formulation of a single language useable as a design development medium, a design documentation medium, and a mechanism for automated processing of the resulting design specification is a formidable task. The success or failure of this effort may strongly affect the utility of this overall approach to automated aids for software design.

As actually developed (Hamilton & Zeldin, 1976a), AXES is a reasonably sophisticated language whose successful use probably requires a designer with significant background in language theory. The language is reasonably compact, but its notation may be intimidating for the less sophisticated designer. Overall, the ISDS/HOS approach makes fairly heavy demands with respect to the theoretical background of the designer.

WARNIER'S "LOGICAL CONSTRUCTION OF PROGRAMS"

Warnier (1974) calls his design methodology the "Logical Construction of Programs" (LCP). In terms of the scope and complexity of programs to be designed, it is nearly identical to Jackson's approach. Warnier is much more specific as to the design steps and stages, however, giving rules and definitions rather than relying so heavily on examples. The terminology and notation of Warnier's book are much different from Jackson's. However, the two methodologies have a great deal in common. Both are, in fact, incorporated into a single presentation by Infotech Information Limited.

Overview

LCP is a relatively mechanical design methodology which requires that the designer prepare a specific set of inputs and then apply a formal procedure which "simplifies" those inputs and transforms them into a design specification in a Program Design Language. As in Jackson's approach, the first step taken by the designer is a specification of input and output data structures, using sequence, interation, and selection. The designer then specifies the relationships between input and output structures. These relationships are described via formal logic statements. A series of prescribed operations are then performed to reduce the set of relationships to (logically) simple form, to develop from the data structures and relationships a program structure, and to evolve procedural statements (in a PDL) which fill out that structure.

Detailed Description

There are five phases in Warnier's design methodology. The first three of these correspond to design on levels of greater detail. The fourth is essentially a finalization of the design using a pseudo code. The fifth stage is verification. In Warnier's terms, this means that the final instruction sequence is checked by hand against the previous levels of detail, the program skeleton, and that is checked in turn against the analyzed data structures.

The first step in LCP is to assess the data structures (see Figures 15 and 16). As in Jackson's methodology, this is a decomposition producing, in most cases, a hierarchy. Each point of decomposition is based on repeated information or alternate possibilities. Items and sets of items (at the appropriate level) are listed sequentially. Thus the structures of sequence, iteration, and selection are, again, the building blocks. Assessment of data structures, in Warnier's approach, means not only defining the structures but considering correspondences of the input and output structures.

Warnier devotes considerable attention to the systematic and formal analysis of these correspondences. Although input and output data structures are internally simple, the relationships between them can be complex, when these two structures involve noncorresponding iteration and alternative constructs. These relationships are expressed in Boolean algebraic form, so that a statement of one such relation might be expressed (in words) as, "the output data set will contain a record of type X if the input data set contains a record of type A and either a record of type B or one of type C." A complete set of such output-to-input relational expressions is developed for any problem involving complexities of this sort. Such expressions can be represented in truth-table or decision-table form, and are susceptible to simplification via the ordinary operations of Boolean algebra.

The second step is to compose a skeleton program structure (see Figures 17 and 18). This means that, based on the structure of the data, the program logic (flow of execution) is to be outlined. Repeated items will require a program loop. Simple alternatives will require a decision and a pathway for each. In instances involving complex output-to-input relationships, the resulting program structure is determined from the Boolean or truth-table analysis discussed above. In some complex cases, special rules for deriving the program structure are presented which may

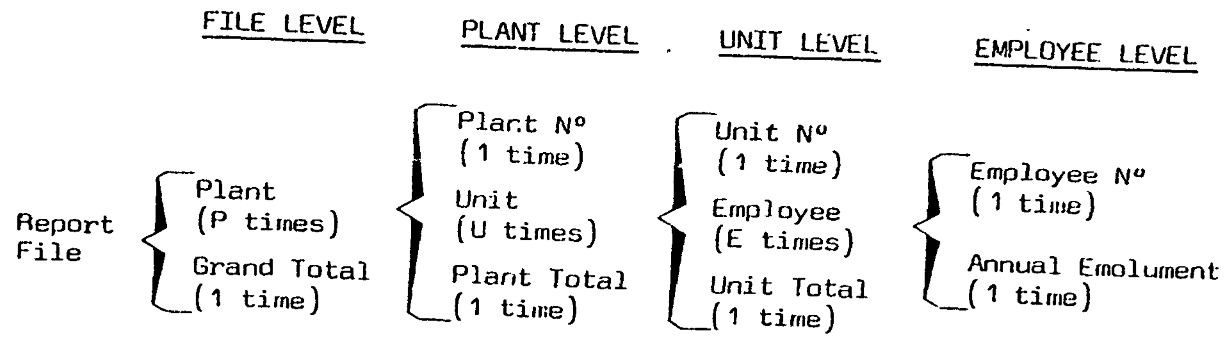


Figure 15. Hierarchical Structure of a Sample Output Data Structure (Step 1) (from Warnier, 1974).

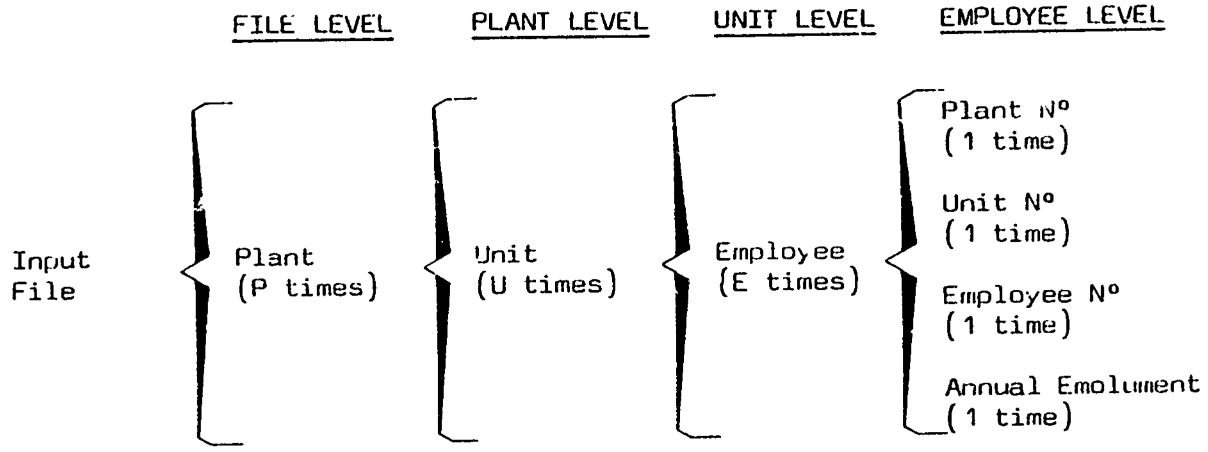


Figure 16. Hierarchical Structure of a Sample Input Data Structure (Step 1) (from Warnier, 1974).

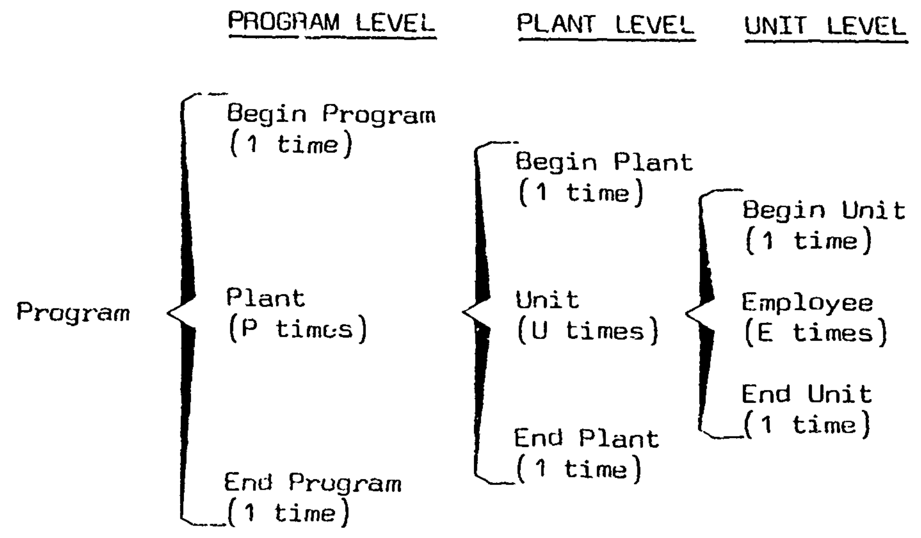


Figure 17. Hierarchical Structure of the Program (Step 2)
(from Warnier, 1974).

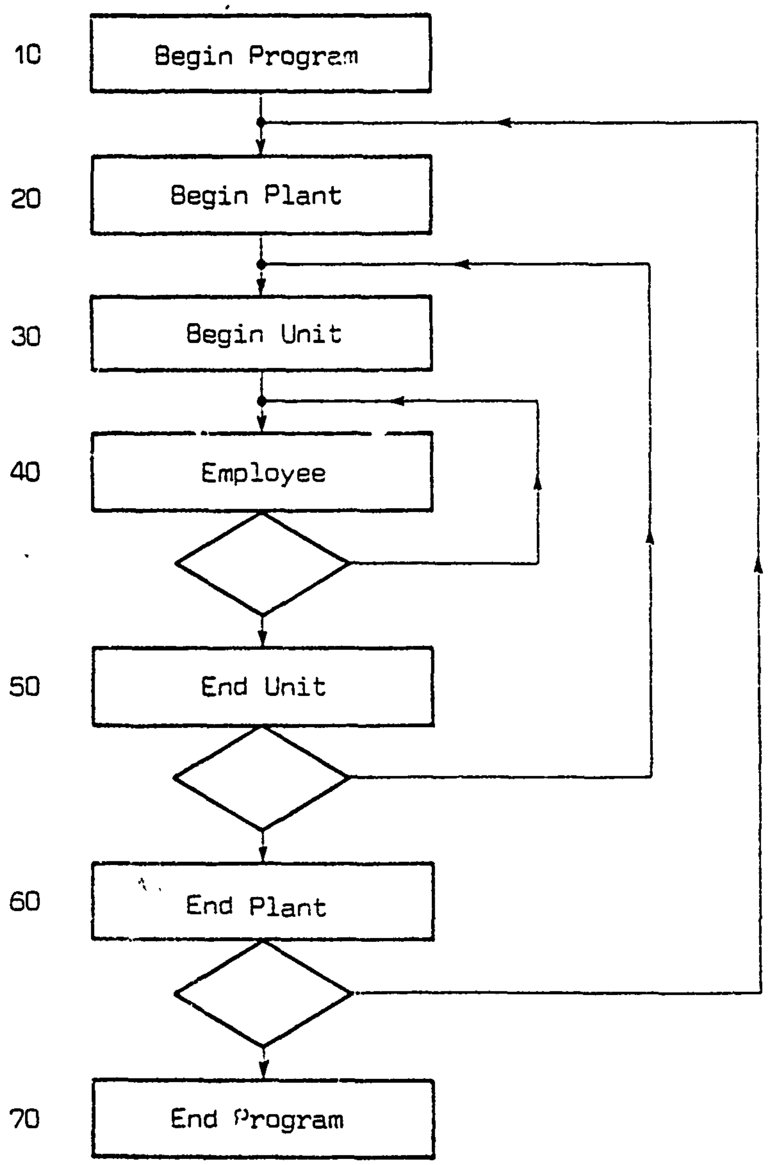


Figure 18. Skeleton Program Structure (Step 2) (from Warnier, 1974).

yield rather complex structures. In particular, if the data subsets corresponding to the actions represented in the program can be made either disjoint or wholly included, a tree structured design is produced; if not, a complex alternative solution, which considers two data subsets simultaneously at all or some decision points, is required.

When data required for decisions are not available at the time of input (as when a whole record group must be read before the appropriate action can be determined) a multiphase (or multi-pass) design is required. In this case, intermediate data structures need to be designed (since they are not apparent in the original problem statement). When these structures are specified, each phase can be designed individually using the above techniques. Warnier doesn't discuss the backtracking technique advocated by Jackson. Warnier's "processing phases" approach, however, appears to approach the same type of problem, in a very different manner, as Jackson's backtracking technique.

During the specification of the program skeleton, Warnier makes no suggestion that functional descriptions be assigned to the components of the skeleton. The skeleton is to be nothing more than an empty flowchart related directly (or indirectly through truth tables) to the data structure (see Figure 18). Different kinds of boxes, according to the functional class of instructions to be inserted later, are required in certain places. But, for the most part, this design phase completely bypasses the heuristics of functional coherence and module independence.

With the complete program structure in skeleton form, the next step is to list operations for each part (see Figure 19). The parts of the skeleton are given numbers, and the operations are listed with those numbers so that the association with the program skeleton is retained. A list of instructions is formulated to map the input data into output. The instructions are not considered or written at this stage in the order in which they will be executed. They are considered according to the broad functional classes into which they fall. Specifically, input instructions

```
10 — Read 1st Record
40 — Read another Record or EoF

40 — If iden. Unit = ident. Ref. Unit      40
50 — If ident. Plant = iden. Ref. Plant    30
60 — If $\overline{\text{EOF}}$                              20

20 — Transfer Plant N° read to Ref
     plant N°
30 — Transfer Unit N°.read to Ref
     Unit N°

10 — Clear Grand Total
20 — Clear Plant Total
30 — Clear Unit Total
40 — Add Annual Emolu. to Unit total
50 — Add Unit total to Plant total
60 — Add Plant total to Grand total

20 — Edit Plant N°
30 — Edit Unit N°
40 — Edit Employee N°
40 — Output and restore Print line
50 — Edit Unit total
50 — Output and restore Print line
60 — Edit Plant total
60 — Output and restore Print line
70 — Edit Grand Total
70 — Output and restore print line
```

Figure 19. List of Operations for Each Part of the Program Skeleton (Step 3). (from Warnier, 1974).

are considered first, branch instructions second, calculations third, output instructions fourth, and subroutines last.

The next step is to sort the instructions into the proper parts of the skeleton (see Figure 20). Within each distinct part of the program, Warnier advises that instructions are generally performed in the following order:

a. Preparation for branches
b. Calculations
c. Outputs
d. Inputs
e. Branches

Subroutines are classified according to their function to make this assignment.

When the process of assigning instructions to parts of the program skeleton is complete, a detailed flowchart for the program exists. The instructions are in a pseudo code of nigh-level, functionally oriented terms. The next step is to exhaustively follow the instructions through the flow diagram and mentally observe the transformation of input to output. This is the verification step. When the instructions were listed, in the previous step, they were listed because each was necessary to some aspect of program requirements. This final verification step is to determine if all the steps are sufficient and in the proper logical sequence.

In summary, the steps in LCP are:

a. Assess and define the data structures.
b. Compose the program skeleton based on the input data.
c. List the operations for each part of the structure. Consider them not in order of execution, but according to category: (1) Input, (2) Branch, (3) Calculations, (4) Output, (5) Subroutines.

```
10 — Clear Grand Total
     Read 1st Record
20 — Transfer Plant N° to Ref Plant N°
     Clear Plant total
     Edit Plant N°
30 — Transfer Unit N° to Ref Unit N°
     Clear Unit total
     Edit Unit N°
40 — Add Annual Emol. to Unit total
     Edit Employee N°
     Output and restore print line
     Read another record or E̅O̅F̅
     If ident. Unit = ident. Ref. Unit        40

50 — Add Unit total Plant total
     Edit Unit Total
     Output and restore print line
     If ident. Plant = ident. ref. Plant      30

60 — Add Plant total to Grand Total
     Edit Plant total
     Output and Restore print line
     If E̅O̅F̅                                   20

70 — Edit Grand Total
     Output and Restore Print line
```

Figure 20. Sorted List of Instructions (Step 4)
(from Warnier, 1974.)

d. Sort the operations into parts of the skeleton. Classify subroutines according to their function. Each component of the skeleton will generally require the following sequence: (1) Preparation for Branches, (2) Calculations, (3) Outputs, (4) Inputs, (5) Branches.

e. Verify the design by checking the instruction sequence against the skeleton, and the skeleton against the analyzed data structures.

LCP produces a hand-verified program written in pseudo code. Implementation of the design should be a very straightforward translation of this code into an actual programming language. As in Jackson's approach, data structure is a problem "given" and design begins with its specification. Data structure design is not addressed. LCP is also not guided by heuristics, but is a more disciplined step-by-step procedure. It, therefore, does not address the general problem-solving aspects of design, but provides specific approaches for certain kinds of design problems. The technique is logically robust enough to handle design problems of considerable complexity, but may very well be too cumbersome to be usable by designers in such instances. This will be discussed in a later section. Problem subsetting, module definition and interface description, and design on different levels of detail are subjects not discussed in this methodology.

Much of Warnier's concern is with implementation efficiency and a good deal of his procedural presentation is concerned with the use of truth tables and related techniques purely for the purpose of program optimization. Although the techniques advocated are quite adequate for assuring that the optimized version of the design is functionally correct, they are somewhat cumbersome and may yield designs not easily comprehensible. As a casual observation, it is not clear why Warnier goes to the (sometimes considerable) trouble of developing detailed truth tables to describe a program's decision logic and then converts those tables to a rigid (and sometimes very complex) program structure. Perhaps table-driven programs would be more appropriate.

ANALYSIS FROM A PROBLEM-SOLVING PERSPECTIVE

In a recent article, Peters and Tripp (1977) present a somewhat pessimistic view of the state of the art of software design methodologies. They criticize several design techniques for their limited applicability and the unprovable assumptions upon which they are based. Readers familiar with this literature may well concur.

From a problem-solving perspective, it is not surprising that existing software design methodologies are primitive. As was indicated in the introduction, software design problems are often complex and perhaps ill-structured. They are in the realm of artificial science, in which the "correctness" of a solution is not necessarily an objective issue. More to the point, though, the problem-solving behavior of software designers is not well understood.

It may be useful to think of design methodologies as providing problem-solving aids for the designer. In fact, much of the emphasis of the surveyed design methodologies can be related directly to the resource limitations inherent in human problem solving (cf. Norman & Bobrow, 1975). Yet, as we hope to make clear, the methodologies barely scratch the surface. Much more powerful aids may be possible, but their development depends critically on a deeper understanding of the problem-solving behavior involved.

The same comment applies to the development of "stronger" design methods (in the strong vs. weak methods sense discussed in the introduction). It may or may not be possible or desirable to develop a catalogue of strong (situation-specific) algorithms for software design. Certainly, the fact that software design is an artificial science does not necessarily preclude such methods (mathematics is an excellent counter example). Again, it is our current understanding of software design, as a human problem-solving activity, which restricts our current efforts to the development of weak, but general, methods.

In terms of their fundamental approaches to problem solving, the surveyed methods fall into two broad classes. A discussion of the properties of each of these classes will be followed by a consideration of some difficulties and errors to which the methods may be susceptible.

PROBLEM-REDUCTION APPROACHES

Several of the software design techniques considered in this review are based on some type of problem-reduction heuristic. The underlying intent is to decompose a larger problem into subproblems which are (hopefully) conceptually more manageable than the original problem. Decomposition strategies of this sort are frequently employed by human problem solvers when the initial problem is too complex to solve directly, but it can be broken into relatively independent subproblems. Such strategies are also common in artificial intelligence systems designed to solve complex problems.

Frequently, the key to success of a problem-reduction strategy is the achievement of independence in the problem decomposition. If the problem is decomposed into subproblems which are not independent, the effective complexity of the problem solver's task may not be reduced. In extreme cases, it may even increase. It is important to keep in mind that the human problem solver has fairly severe resource limitations (especially, short-term-memory limitations) within which to operate. When a problem-reduction stategy is used on a complex problem, it is very unlikely that the problem solver will be able to adequately recall and utilize the global information required to deal with complex subproblem interdependencies. Much of the benefit achievable througn such strategies involves the ability to concentrate on only one (sub)problem at a time, decomposing it on the basis of local information.

With the exception of the methodologies of Jackson and Warnier, all of the formal methodologies and informal design techniques considered here are explicitly concerned with a problem-reduction strategy. The basic problem is decomposed into subproblems, which are in turn decomposed into their subproblems, etc. The problem-reduction process is very apparent, since each problem is ordinarily expressed in terms of the function of a software module. Each module is then decomposed into a set of component functions, which become modules at the next lower level, until the design has been developed to the desired level of detail.

The cautious reader may have noticed that the concept of planning by levels of abstraction (as discussed by Ledgard, for example) does not strictly imply a correspondence between problem-reduction steps and software modular structure. However, even this approach, as actually practiced, relies primarily on modular decomposition. In practice, modules correspond to the (sub)problems to which problem-reduction operators are applied.

The problem-reduction process typically involves both problem-reduction operators (i.e., methods or guidelines for decomposing a problem into subproblems) and evaluation functions (i.e., techniques for evaluating the resulting decomposition). Structured design, for example, uses a data-flow analysis, followed by either transform or transaction analysis, to suggest useful modular decompositions. The quality of the resulting decomposition is evaluated in terms of modular independence, functional coherence, etc. Figure 21 summarizes the problem-reduction operators and evaluation functions advocated in Structured Design, ISDS/HOS, and in the several structured-design-related informal techniques which were surveyed.

Clearly, the problem-reduction heuristics of structured design are more explicit, more detailed, and more procedural than those of the other problem-reduction approaches. In the other approaches, the designer is given very general, highly subjective problem-reduction heuristics (Parnas' "information hiding"), or is given no guidance at all. This should be considered as a comment on the relative maturity

Approach	Problem-Reduction Operators	Evaluation Functions	Emphasis of Method
Stepwise Refinement	Not explicit. Design one level at a time. Highest levels are abstract, lowest is concrete machine implementation.	Not explicit. Each level is checked for completeness, correctness before proceeding.	General advocacy of problem-reduction approach.
Information Hiding	Isolate design decisions in independent modules.	Implicit in problem-reduction operator (independence, functional coherence).	One heuristic for selecting appropriate modular decomposition.
Goal-Directed Programming	Not explicit.	Independence Functional coherence	General advocacy of problem-reduction approach.
Structured Design	Restate problem as data flow graph and: (1) factor into afferent (input), efferent (output) and central transform modules OR (2) factor into transactions and their defining action modules.	Independence (coupling) Functional Coherence (cohesion) Span of control Scope of effect/control Fan-in/out Module size Simplicity (do not generalize)	Generally applicable heuristics for modular decomposition and for evaluation of resulting decomposition.
ISDS/HOS	Not explicit. Decomposition rules constrain modular decomposition, may be suggestive for designer, but provide no real guidance for selection of appropriate decomposition.	Implicit in required logical structure (independence, functional coherence). Some automated checking.	Rigid logical structure for expression and control of modular decomposition "moves". Automated aids to verify adherence.

Figure 21. Summary of Problem-Reduction Methods.

of those approaches, rather than a criticism of their fundamental principles. It is likely, though, that fairly explicit problem-reduction heuristics will be required if a problem-reduction-type design methodology is to be applied with any rigor or consistency. Without such guidance, the approach may affect the designer's attitudes about the design task, but have little direct effect on problem-solving performance.

Even ISDS/HOS offers little explicit guidance in this area. Although this methodology provides a very explicit set of "decomposition rules", those rules act primarily as formal constraints on the decomposition, rather than as aids to the selection of a useful decomposition. To express it another way, the application of the decomposition rules is straightforward, once the function of a module has been expressed in terms of subfunctions which are related by sequence, set partition, and class partition logic. But that reexpression of a module's function in terms of an appropriate partitioning into subfunctions is the crux of the problem, and ISDS/HOS offers little help here. On the other hand, there appears to be no basic imcompatibility which would prevent the ISDS/HOS designer from employing problem-reduction operators derived from the other methodologies.

The evaluation functions of these various methods, when they are explicitly present, are primarily concerned with ensuring modular independence. Again, the most detailed set of evaluation functions is found in the discussion of structured design. In some cases ("Information Hiding", ISDS/HOS), a degree of modular independence is an implicit product of the problem-reduction operator or of formal constraints on problem-reduction moves. Where explicit evaluation functions are present (e.g., structured design), the more important ones (e.g., coupling, functional coherence) are highly subjective in nature, and perhaps difficult to apply. Although ISDS/HOS may be difficult to use for other reasons, it is almost certainly the most effective of these methods for achieving low coupling and high functional coherence.

Considered as a group, the problem-reduction approaches are weak methods, broadly applicable but not extremely powerful. They involve problem-reduction operators and evaluation functions which are relatively simple and which have considerable face validity as mechanisms for ensuring that the problem is partitioned into relatively independent subproblems. As will be seen later, this appearance of independence can be misleading. Even the most extensively developed of these methodologies involves problem-reduction operators and evaluation functions which are heuristic and, in fact, highly subjective in application. This is good in some respects, and bad in others. It is desirable because it allows the methodologies to be useable on a broad range of problems by a broad range of designers, without interfering markedly with the designer's use of task-domain knowledge and personal design techniques. It is undesiraole because success depends heavily on the skill and experience of the designer to find useful problem decompositions and to recognize situations of high subproblem interdependence.

ALGORITHMIC APPROACHES

The second group of design methodologies consists of the approaches of Jackson and Warnier. These approaches differ from the first group in several ways, and it is difficult to decide which of these differences is the most basic.

Each of these methodologies starts with a formal specification of input and output data structures. A (basically mechanical) procedure is then employed to map these data structures into a corresponding modular structure. In the case of Warnier's approach, further algorithmic procedures lead to the development and sorting of individual program statements.

In a sense, these methodologies, too, involve problem-reduction operators. It is important to recognize, though, that the problem decomposition process is assumed to be a mechanical, algorithmic procedure and, unlike the problem-reduction approaches, is applied prior to, rather than concurrently with, the development of a modular structure. The problem reduction is specifically not done on the basis of the designer's problem-related knowledge or skill. Evaluation functions, in the sense discussed in the previous section, do not exist in these methodologies. Clearly, though, a basic intent of the mechanical procedures employed in these approaches is the assurance of an appropriate level of independence among system and program components.

Both of these approaches start with specifications of the input and output data structures. These specifications are expressed in the form of hierarchic structures with mandatory and/or optional elements, which may be iterative. Neither author explicitly indicates how this data structure specification should be done, or what difficulties might be encountered in the process of describing data structures in this way. In many application areas (e.g., business report generation), data structures seem naturally to take this form, and the task is probably fairly simple. In other, or more complex, situations, it may not be simple. In particular, the designer may have difficulty adopting this particular representation of the problem, even though it probably has enough logical power to handle a wide variety of design problems (see the next section for a more detailed discussion of this issue). Informal reports on the use of these methodologies (e.g., Peters & Tripp, 1977) suggest that this difficulty does occur in practice.

Once the data structure specification exists, Jackson's approach maps that data structure directly into a corresponding modular structure. If the input and output data structures correspond, this mapping is one-to-one; otherwise, a "structure clash" exists. The technique for handling this situation ("program inversion") is heuristic in nature, but appears sound and relatively straightforward in noncomplex situations. It is not apparent that program inversion would become significantly more difficult in complex design situations, but that is a possibility which should be kept in mind.

The remaining technique in Jackson's repertory is "backtracking," which is a technique for handling what might be called a "temporal clash," in which the system must either make a (reversible) decision based on incomplete information or defer the decision until more information is obtained. This problem is particularly significant to Jackson because the one-to-one correspondence of modular structure (and thus control structure) with the data structure is difficult to maintain when this situation occurs. Jackson's solution is a "commit now, then quit and recover if necessary" approach. This approach may be satisfactory in some cases, but clearly cannot be universally applicable. It is possible to devise cases which make such recovery arbitrarily difficult. In any event, "backtracking" often results in unnecessarily high module coupling, which may complicate not only the design task, but virtually all later stages of software development. It seems likely that the multiphase technique of Warnier, or some similar deferred-commitment approach, is more universally applicable.

Once these techniques have been employed to develop a modular structure, the design task is more or less completed. The detailed design of the individual modules is a programming task not formally addressed by Jackson. In terms of level of detail, then, Jackson's approach corresponds to that of the structured design methods.

Warnier's methodology is even more algorithmic and extends through the development of actual code. Again, the starting point is a hierarchic specification of the input and output data structures. The designer must then specify, via formal boolean equations, the logical relationship of the output data elements to the input data elements. From this point on, the process is entirely algorithmic. The designer uses truth tables or some similar means of simplifying the boolean equations, and uses mapping rules, sorts, etc., to generate the program.

It is evident that both Jackson and Warnier are attempting to provide objective methods for software design. In the process, though, they (especially Warnier) have developed approaches in which the designer executes very mechanical procedures. This may be a source of considerable difficulty.

Greeno distinguishes between "formal" and "informal" reasoning, where formal reasoning involves the use of syntactic information, formal languages, relatively mechanical procedures, etc., while informal reasoning involves semantic models. The reasoning processes involved may differ considerably between these two classes of problem-solving behavior. Larkin (1977) has presented data which suggest that very experienced physicists may adopt a predominately semantic (informal) approach to the solution of physics problems, whereas relatively inexperienced physics students proceed immediately to the use and solution of mathematical equations, and thus employ formal reasoning. Presumably, approaching the problem with informal reasoning would allow the problem solver to make much greater use of his knowledge of the problem domain, experience with conceptually related problems, etc.

The very formal, syntactic approach of Warnier (and, to a lesser extent, of Jackson) may very well deprive the designer of the ability to use problem-relevant knowledge to resolve difficulties which arise in the design. Of course, if no difficulties arise, this may not be an issue. At the risk of overgeneralizing, though, it would seem that in the design of very large, complex systems, difficulties always arise. If, at this point, the design is sufficiently different from the designer's internal representation of the problem, and the design process has relied heavily cn formal, syntactic reasoning or even mechanical procedures, the designer will be in trouble. Under these circumstances, one might speculate that corrections to the design will necessarily take the form of "patches," based on local knowledge and on reasoning at a fairly syntactic level.

If the speculations of the two previous paragraphs are correct, then the methods of Warnier and Jackson are probably limited to problems of moderate complexity <u>even if</u> the underlying design procedures are basically sound and the specification of data structures in the required hierarchic form is a manageable task. Of course, these latter assumptions must also be satisfied, but we have no present basis for determining whether, or in what situations, they are satisfied.

We have suggested that these "algorithmic" methodologies may be limited with respect to problem domain and problem complexity, and that they may limit the advantageous use of relevant knowledge by the experienced designer. On the other hand, such methodologies may be quite advantageous if used by <u>in</u>experienced designers for appropriate problems. The difficulty is that we have no present basis for determining the appropriate problem domain. It might also be true that inexperienced designers do not become "experienced" designers, in the sense used above, by employing such methods.

EFFECTS OF DESIGN TECHNIQUES ON PROBLEM REPRESENTATION

If satisfactory problem-solving performance is to be achieved, it is necessary that the problem solver be able to form an appropriate representation of the problem. While forming an appropriate representation can aid in problem solving, it is also known that, in some problem-solving domains, the formation of an inappropriate representation can prevent a solution from being achieved. As used here, "problem representation" refers not merely to formal notation, but encompasses more specifically the designer's perception of the logical structure of the problem, legal alternatives, etc.

In order to illustrate this issue, consider for example the problem discussed by Wertheimer (1945) of finding the area of a parallelogram (see Figure 22a). To solve this problem, it is necessary to drop perpendicular lines from the upper right and left corners and extend the base line (in this case) to the right (see Figure 22b).

In a classroom setting, Wertheimer noted that students developed one of two problem representations. The first involved recognizing that the problem requires proof of the congruence of triangles aed and bfc. The other representation is exactly as stated above: "drop perpendicular lines", etc. This second type of representation is, of course, inappropriate for parallelograms of the form shown in Figure 22c.

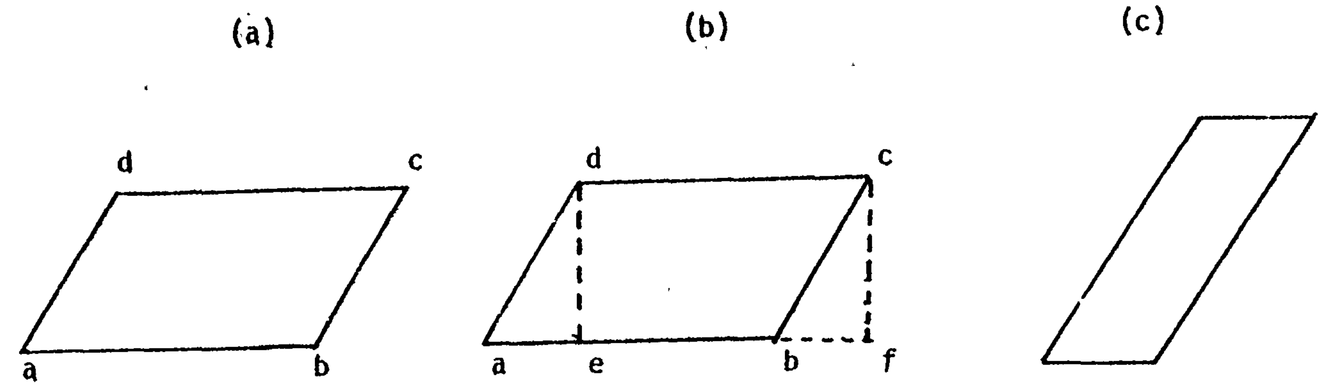


Figure 22. Calculating the Area of a Parallelogram.

The first type of representation can be characterized as "logical," "intuitive," or "global" while the second can be characterized as "mechanical" or "algorithmic". Duncker (1945) extended this aspect of problem solving by demonstrating that a particular problem representation is closely related to the manner in which a solution is attempted ("functional fixity"). Maier (1930) experimented with techniques for causing subjects to adopt more appropriate representations ("direction"). Paige and Simon (1966) demonstrate that subjects who adopt highly algorithmic problem representations will apply algorithmic methods, with great persistence, on problems that have no solution (i.e., nonsense problems that do not represent possible events). In various contexts, problem representation has been shown to affect the nature and success of solution attempts (Hayes & Simon, 1974) and the type and number of errors made during problem solving (Jeffries et al, 1977).

With the exception of ISDS/HOS, the problem-reduction techniques do not generally appear to impose significant constraints on problem representation, nor do they give significant guidance in the choice of a representation. They impose no obvious constraints which affect the designer's choice of data structure types, nor do they contain strong constraints on the modular structure or the designer's (largely experiential) criteria for modular decomposition. They do, of course, impose constraints on the modular decomposition procedure, but that is a matter of problem-solving method, rather than problem representation, and it has already been discussed.

ISDS/HOS does appear to constrain the problem representation, to a degree which may be mild or quite significant, depending on the design problem, the designer's facility with the AXES specification language, and the guidance given the designer, by ISDS, during the modular decomposition process. Clearly, the axioms and decomposition rules also impose some constraints (and give some guidance) with respect to the modular structure and decomposition alternatives, but AXES, and the interaction with ISDS, are probably even more constraining. These constraints are, of course, helpful in simplifying the interface between the designer and a system of automated aids. Furthermore, the restrictions on problem representation may not hinder the designer if they match his "native" approach to design and are appropriate to the problem. If they do not, however, the constraints may significantly hinder performance. Unfortunately, our understanding of software design as a problem-solving task and of the relevant variables of the task domain is presently insufficient to allow us to predict the circumstances under which these constraints might have deleterious effects. And once again, if the task properties and the designer's experience and predelictions are closely compatible with the constraints, these constraints may even be actively beneficial to problem solving.

The methodologies which most clearly restrict the designer's problem representation are those of Jackson and Warnier. These approaches require that the designer adopt a particular kind of hierarchic data structure, and that he represent all problem-re evant information in terms of those structures and their relationships. As noted earlier, there may be situations in which this is quite easy, but there are probably also situations in which it is not a viable representation for use by the human problem solver. Both of these authors beg the question of the difficulty of this task, but it is probably the most important determinant of the viability of Jackson's basic method, and one of several significant factors in the viability of Warnier's approach.

Of particular relevance is Durding, Becker, and Gould's (1977) study of the ability of human problem solvers to utilize a variety of data structure types. Although the subjects were able to use a variety of data structures (e.g., hierarchic, list, network structures) when appropriate, they had considerable difficulty expressing information with a data structure type which did not inherently match the logical structure of the data. While it can be objected that these subjects were naive with respect to data structure use, and that the stimulus materials were perhaps somewhat leading, the study clearly suggests a possible limitation of these data-structure-constrained methodologies. This limitation is psychological, and is not eliminated by Jackson's demonstration that hierarchic structures are logically adequate for a wide variety of situations.

PREVENTION AND DETECTION OF DESIGN ERRORS

The prevention, or subsequent detection, of software design errors is a topic of considerable importance which receives limited explicit attention in the surveyed design methodologies. Undoubtedly, many design errors are eliminated by the use of any systematic approach, while others may be implicitly prevented by particular methodologies. One has the impression, nonetheless, that the designer is mostly "left to his own devices," where error prevention and correction is concerned.

Clearly, our ability to devise heuristics or procedures for the prevention or detection of design errors depends on the nature of those errors, and on our understanding of error classes. In particular, there will probably always be errors which are so specific to one application area that they elude corrective measures compatible with general-purpose (weak) design methods. And yet, it seems possible that an analysis of design error classes, based on our understanding of human problem-solving behavior, would yield information useful for the development of general-purpose preventive measures for a significant proportion of design errors.

Consider, for example, the design shown in Figure 3a of the Introductory section of this paper. This design represents a partial solution to the problem described in Figure 2. The designer might actually decompose each of the indicated modules to several additional levels before considering the design completed. Superficially, the initial design step shown in Figure 3a seems entirely reasonable, and is probably the most common first step taken in this problem, particularly as the problem is worded. Yet it contains a design commitment which is wrong, or at least very troublesome, and which may easily go undetected and uncorrected all the way through implementation. This particular example is from a study we are now conducting. It was generated by a very capable and fairly experienced software designer whose background included significant exposure to text processing. The error in question was never detected by this designer, and was allowed to remain in the finished design.

To understand the nature of the problem, it will be necessary for the reader to reacquaint himself with the design problem described in Figure 1. Now imagine, if you will, a situation in which one page ends in the word "Civil", while the next page begins with the word "War". Suppose, further, that one of the index terms for the book is "Civil War". If the text is truly processed a page at a time, as the design suggests, no index entry will be generated in this case, because the phrase in question crosses a page boundary.

Admittedly, this problem can be corrected by provisions, at lower levels of the design, for storage of a partial phrase in a buffer, so that processing is really done almost a page at a time. This "patch" is an inferior solution, however, since it: (1) increases coupling among modules, (2) causes the design to contain a module ("Process Page") which does not perform the function that most people would infer from its name, and (3) may result in unnecessary additional data storage and data management activities. A better solution is a design which recognizes from the start that processing is done a word at a time, rather than a page at a time. In such a design, there might be a "Get Next Word" module, which calls its submodule, "Read Page", when necessary.

The real point of this example, though, concerns not the ease or difficulty with which the error can be corrected, but the high probability that it will not be detected at all. We have speculated that the designer employing a problem-reduction strategy is forced, by his own memory resource limitations, to perform problem-reduction operations primarily on the basis of local information. Although there are many variables involved here, this speculation appears, in general, to be highly defensible. If that is true, and if the designer has not already recognized this problem at the time the first design step (Figure 2a) was taken, then there is a small, and perhaps decreasing probability of detection of the error as the designer attends to lower levels of the design.

From a problem-solving viewpoint, this error has very interesting properties. This kind of error can be made very early, and high, in the design, and yet be detectable only if the designer is attending to fairly global information when he is working at a very low level in the design. In this kind of situation, it would seem that a problem-reduction approach leads the designer "down the garden path," so to speak. Errors of this sort are potentially very serious, and are probably quite common. An error similar to our example was discussed by Henderson and Snowdon (1972) and Ledgard (1973), and similar situations have been treated, though not as errors, by Jackson (1975).

There appear to be three broad approaches available to us if we wish to devise design procedures or aids capable of preventing or correcting such errors during the design phase. These three approaches will be discussed in turn. First, we might attempt to provide some sort of assistance which would aid the designer in making use of global information while he is working lower in the design. This approach might involve some sort of automated aid capable of extracting relevant global information from higher-level design steps and presenting it to the designer as he works at lower levels. It is not at all clear how this might be done and it does not promote early detection of the problem, but the possibility is mentioned here in the interest of completeness.

A second approach involves the development of better mechanisms for recognizing the error after the design step in which it occurred, but before more detailed design work is done. These mechanisms might assume any of several automated or manual forms. The automated aids of ISDS/HOS are one example, although it is not clear that they would assist with the detection of basically conceptual errors, such as that of our example. They are intended more to ensure that only allowable design moves are made (as defined by the axioms and decomposition rules), and that the various modules have compatible interfaces with one another and with the data structures.

Another kind of automated aid, which may be more relevant here, is suggested by the "critics" used in Sacerdoti's (1975) Nets of Action Hierarchies (NOAH) System. NOAH is an "artificial intelligence" system which solves problems via a hierarchic problem-reduction approach similar to that which appears to be most common in software design. However, NOAH also employs a set of automated procedures, called "critics," which clean up the solution plan, resolve inconsistencies, etc., after each level of the plan is developed. The particular critics built into NOAH are very general-purpose ones, and clearly could not detect an error such as that of the example. The concept of critics is interesting, however, and it is conceivable that a library of software-design-related critics could be constructed in connection with an automated design aiding system such as ISDS/HOS.

As suggested earlier, it is also possible that improved manual procedures for the designer, and possibly improved design review and walkthrough techniques, could be used to detect an error of this sort before more detailed design is done. This particular error is one which might very well be detected by a structured design review before other experienced designers, conducted after only the one design step has been done. This is not the way in which design reviews are ordinarily done, but waiting until several design levels have been developed may tend to cause the reviewers, too, to attend to relatively local information. In this event, they might fail to detect the error even though the review group contains personnel experienced with the class of error involved.

The third basic approach, and the one which appears most promising to us, involves the development of heuristics, or even algorithmic procedures, which are applied before a design step. These procedures are intended to detect the error-prone situation and prevent the erroneous design step from being taken in the first place. This may strike the reader as overly ambitious. However, one, and perhaps two, heuristics already exist which might have prevented this particular error -- and, perhaps, a wide variety of errors of the same general sort.

The astute reader may have been aware, all along, that the particular error, which we have used as an example, is an instance of the "structure clash" described by Jackson (1975). The structure clash is a situation in which the basic unit of input information is different from the basic unit of output information, and neither is a proper subset of the other. In the example, input is read in pages, "output" is in phrases, and only the smaller unit, the word, is common to the two. Jackson resolves this difficulty by "program inversion," in which communication is done in units which might be thought of as the smallest common divi or -- in this case, words.

The important contribution made by Jackson, though, is not program inversion, although that appears to be a sound treatment of the structure clash, once it has been identified. The important contribution is his description of a procedure for detecting the situation itself. Because Jackson's methodology contains a procedure for detecting the structure clash, it ceases to be a hidden error-prone situation and becomes, instead, a known property of the design problem -- before the erroneous design commitment is made.

Although the heuristic for recognizing structure clashes arises rather naturally from Jackson's basic approach, its use is by no means restricted to designers who are using that approach. Furthermore, once this particular class of error has been recognized as a class, other heuristics also suggest themselves. For example, an extension of the data flow analysis used in structured design could probably be used to detect this same situation.

The important prerequisite to the systematic development of heuristics or algorithms for detecting error-prone design situations is the development of a taxonomy of error classes, from a human problem-solving perspective. Previous analyses of software development errors have typically broken them down only into such categories as "conceptual," "clerical", etc. Given the perspective suggested above, though, it may be possible to categorize observed design errors not only according to their surface features, but also in terms of the processes which led to them, and their implications for later problem-solving steps.

SUMMARY

Figure 23 provides a very brief summary of some of the factors which might affect the utility of the various formal methodologies. Tables of this sort are necessarily oversimplified, and this table should be interpreted in the context of the lengthier explanations already given. For the reader who has read the previous discussion, this table may serve as an aid to simultaneous consideration of the four methodologies in terms of the principal human factors problems which were identified. The brief summary statements given below should also assist this integration.

Structured Design

Structured Design is the weakest, most broadly applicable design methodology of the four. It appears to be compatible with design problems of any size from large systems to individual programs. It is a problem-reduction approach which ordinarily proceeds top-down, but the approach could be used with other design strategies (e.g., middle-out). By a considerable margin, Structured Design is the least constraining of the four methodologies. The designer is free to adopt the most meaningful representation of the problem, and to make modular decomposition and other decisions on the basis of the designer's knowledge and experience. In fact, it depends heavily on that knowledge and experience. Structured Design

		Structured Design	ISDS/HOS	Jackson's Methodology	Warnier's Methodology
Description	General Nature	Problem-reduction	Top-down problem-reduction	Algorithmic	Algorithmic
	Basis for Modular Decomposition	Data flow	Data flow	Data structure	Data structure
	Primary Documentation Methods	Data flow "bubble chart", Structure chart	Algebraic notation, AXES design language	Structure chart with iteration, selection operators	Specialized structure diagrams, Boolean statements, Program design language
Evaluation	Size of problem to which method appears applicable	System, program	System	Simple system, program	Very simple system, program
	Degree to which method constrains designer's representation of problem	Low	Moderate	High	Very high
	Conduciveness to application of designer's knowledge and experience	High	High	Somewhat low	Very low
	Required level of theoretical sophistication of designer	Low	High	Low	Moderate
	Method is driven by basic processing requirements of system	Yes	Yes	No	No
	Assistance is provided for required data flow or data structure analysis	Yes	No, but Structured Design approach compatible	No	No
	Susceptibility to design errors	Susceptible to design errors due to subproblem interaction since designer is forced to attend to local information when dealing with low-level subproblems.		Possibly susceptible to mismatch of design and requirements, since method is not directly driven by processing requirements.	
	Final product of design effort is close to actual code in level of detail	Varies	Varies	Somewhat	Yes

Figure 23. A Somewhat Speculative Evaluation Summary of Formal Design Methodologies

provides a framework within which the designer can more readily recognize the design decisions to be made, and it helps in the evaluation of certain aspects of the resulting design, but the basic decisions are made by the designer, not by the method.

Structured design is clearly susceptible to design errors in those situations in which modules are nonindependent, but in which their lack of independence cannot be recognized on the basis of the local information considered at any single decision point in the design process. This is the "garden path" type of error discussed earlier in this section. In at least one major instance (the "structure clash"), procedures exist (or could be developed) which would aid the designer's recognition of the problem. Incorporation of such an algorithm into the methodology might be desirable.

ISDS/HOS

ISDS/HOS is also a weak, broadly applicable methodology which provides procedural and evaluative aids, but gives little guidance with respect to the content of the design decisions. The designer is constrained to a particular approach to modular decomposition which may work very well if it matches the properties of the problem and the style of the designer. Otherwise, these constraints, which are rigidly enforced by software aids, may degrade design performance. The AXES specification language (and, to a much lesser degree, the HOS model itself), appears to require a designer who is theoretically sophisticated -- in such areas as language theory, for example. For all but the most fluent users, it appears likely that the information processing load imposed on the designer by the use of the ISDS system is significant, and would probably impair performance on complex design problems. Only empirical study would determine whether such impairment offsets the benefits of ISDS/HOS, which include the development of designs with very high modular independence and coherence.

ISDS/HOS is also susceptible to "garden path" errors, and compatible with possible algorithms for detection of at least some such errors.

The method appear? to be applicable to large systems design, but it is probably inappropriate (or at least too cumbersome) for lower-level design problems (small systems or programs).

Jackson's Methodology

Jackson's methodology is a stronger, algorithmic approach, which relieves the designer of many modular decomposition decisions. The method is applicable at the program level and to simple systems, but probably becomes unworkable for very complex systems. It is also clear that the approach is not applicable to all types of design problems, but our understanding of problem types is inadequate to allow us to characterize the scope of the method.

Because of its algorithmic nature, Jackson's approach appears to have less dependence on the designer's knowledge and experience than do the problem-reduction methods. To some extent, this is illusory, since important design decisions must be made in devising the data structure which is the input for the method. Furthermore, the methodology dictates most modular decomposition decisions, but the responsibility for satisfying the processing requirements still rests primarily with the designer, and only loose guidance is given for this task.

The data structure constraints imposed by this method represent fairly heavy constraints on the designer's representation of the design problem. Only empirical study can determine the circumstances under which this results in unacceptable performance degradation.

Warnier's "Logical Construction of Programs"

LCP is an extraordinarily algorithmic approach to software design. LCP begins with a definition of data structures and relationships, and proceeds, by an almost purely algorithmic process, to develop a low-level design. The approach is probably restricted to programs and very simple systems, and is probably highly restricted in terms of problem type, but the precise nature of the latter restriction is unclear. For such problems as report generation, the approach may be highly satisfactory.

Like Jackson's method, Warnier's approach moves some of the design decision making into the data structure specification, and provides no assistance with this task. An even more difficult task is the detailed specification of data structure relationships, and no assistance is given here, either. It appears likely that the strong constraints on problem representation, and the highly "syntactic" mode of problem solving will interfere with a designer's ability to utilize knowledge and experience to advantage. Furthermore, the resulting design may diverge significantly from the designer's conceptualization of the problem. A likely result of such divergence is a tendency to "patch" complex designs when requirements change or design difficulties are encountered. Once again, though, only empirical study can really determine the significance of these problems.

CONCLUSIONS AND RECOMMENDATIONS

We have attempted to present a systematic and critical analysis of current and emerging software design methodologies from a human factors perspective. We have not provided an exhaustive coverage of all software design techniques and methods, but rather have focused on those techniques and methods that are in widespread use and that appear to have potentially significant effects on the software design process.

This review effort was originally intended to satisfy several goals:

1. Enumerate the relative strengths and weaknesses of each considered technique

2. Identify commonalities and differences
3. Critically analyze human factors problem areas
4. Make specific recommendations for improvements in design techniques
5. Formulate hypotheses for the empirical analysis of software design techniques

We believe that goals 1 and 2 have been accomplished. We have attempted to satisfy goal 3, to the degree allowed by our currently limited understanding of software design as a human problem-solving task. In several cases, we feel that we have identified the most important human factors problem areas, but were unable to provide any clear resolution of the problems without further research.

With respect to goal 4, we had hoped to be able to make fairly specific recommendations concerning the use of the various design methodologies. In several cases, it is clear that particular design techniques must be restricted with respect to design problem type, complexity, designer experience, etc., but we lack sufficient information to identify the actual boundaries of the design problem domain to which they are applicable. As a result, concrete recommendations of this sort are not yet justifiable. A number of suggestions, of a more minor nature, were discussed earlier in this report, and were summarized in the previous subsection.

An even more ambitious undertaking would be the synthesis of a more powerful system of design techniques, based on the techniques used in existing methodologies and the new ideas which have emerged, and will emerge, from a consideration of the problem-solving aspects of software design. As a long-term goal, this undertaking appears both attractive and feasible. To have any serious hope of success, though, such an effort must be preceded by a program of research intended to improve our understanding of the software design problem domain, the problem-solving processes used by software designers, and the kinds of errors made through the application of these processes.

It should be clear, then, that there are some fairly fundamental gaps in our knowledge of the behavioral aspects of software design. Goal 5 of the project was the identification of research areas and hypotheses which might help to fill these gaps. We have identified 16 topics which might usefully be addressed by a long-term empirical research program on the behavioral aspects of software design and design methodologies. These topics are discussed in detail in Appendix A. Of these topics, the following appear to be appropriate for immediate pursuit:

"Propositional" analysis of software design information

Taxonomy of the software design problem domain

Taxomony of software design errors

Analysis of the effects of ISDS/HOS problem representation constraints

Analysis of the effects of Jackson-Warnier problem representation constraints

Analysis of the effects of design documentation medium on design performance

The reader is referred to Appendix A for a more detailed treatment of these topics. The six listed above appear to be addressable now, while some of the other goals (e.g., synthesis of an improved system of design techniques) depend on information not now available about designer behavior. Thus, the six recommended research efforts are those which appear tractible, are worthwhile, and do not have other research efforts as their logical predecessors.

If anything is clear from this survey, it is that there is considerable room for improvement, both in design methods and in our understanding of design behavior.

REFERENCES

Atwood, M. E., & Ramsey, H. R. Cognitive structures in the comprehension and memory of computer programs: An investigation of computer program debugging (ARI Technical Report No. TR-78-A21). Alexandria, Virginia: U. S. Army Research Institute for the Behavioral and Social Sciences, August 1978.

Atwood, M. E., Turner, A. A., Ramsey, H. R., & Hooper, J. N. An exploratory study of the cognitive structures underlying the comprehension of software design problems (Technical Report 392). Alexandria, Virginia: U. S. Army Research Institute for the Behavioral and Social Sciences, July 1979.

Bazjanac, V. The promises and the disappointments of computer-aided design. In N. Negroponte (Ed.), Reflections on computer aids to design and architecture. New York: Petrocelli/Charter, 1975, 17-26.

Bell, T. E., Bixler, D. C., & Dyer, M. E. An extendable approach to computer-aided software requirements engineering. In P. Freeman, & A. I. Wasserman (Eds.), Tutorial on software design techniques (2nd Ed.). Long Beach, California: IEEE Computer Society, 1977, 96-107.

Boehm, B. W. Software and its impact: A quantitative assessment. Datamation, May 1973, 49-59.

Boehm, B. W. Some steps toward formal and automated aids to software requirements analysis and design (Technical Report No. TRW-SS-74-02). Redondo Beach, California: TRW, May 1974.

Boehm, B. W. The high cost of software. In E. Horowitz (Ed.), Practical strategies for developing large software systems, Reading, Massachusetts: Addison-Wesley, 1975, 3-14.

Boehm, B. W., & Haile, A. C. Information processing/data automation implications of Air Force command and control requirements in the 1980s (CCIP-85): Executive summary (Revised ed., Report No. SAMSO TR-72-122). Los Angeles, California: USAF Space and Missile Systems Organization, February 1972.

Caine, S. H., & Gordon, E. K. PDL: A tool for software design. In P. Freeman, & A. I. Wasserman (Eds.), Tutorial on software design techniques (2nd Ed.). Long Beach, California: IEEE Computer Society, 1977, 168-173.

Chapin, N. New format for flowcharts. Software: Practice and Experience, 1974, 4, 341-357.

Cichelli, R. J., & Cichelli, M. J. Goal directed programming. SIGPLAN Notices, 1977, 12(7), 51-59.

Dahl, O. J. Dijkstra, E. W., & Hoare, C. A. R. Structured programming. New York: Academic Press, 1972.

Davis, C. G., & Vick, C. R. The software development system. IEEE Transactions on Software Engineering, 1977, SE-3, 69-84.

Duncker, K. On problem solving. Psychological Monographs, 1945, 58, No. 5 (Whole No. 270).

Durding, B. M., Becker, C. A., & Gould, J. D. Data organization. Human Factors, 1977, 19, 1-14.

Greeno, J. G. Natures of problem solving abilities. In W. K. Estes (Ed.), Handbook of Learning and Cognitive Processes, Vol. 5. Hillsdale, New Jersey: Erlbaum, 1978, 239-270.

Greeno, J. G. Talk presented at Office of Naval Research Cognitive Processes Contractors' Conference, Boulder, Colorado, May 1978.

Hamilton, M., & Zeldin, S. AXES syntax description (Technical Report TR-4). Cambridge, Massachusetts: Higher Order Software, Inc., December 1976. (a)

Hamilton, M., & Zeldin, S. Integrated software development system/higher order software conceptual description (Technical Report ECOM-76-0329-F). Fort Monmouth, New Jersey: U. S. Army Electronics Command, 1976. (b)

Hayes, J. R., & Simon, H. A. Understanding written problem instructions. In L. W. Gregg (Ed.), Knowledge and Cognition. Potomac, Maryland: Erlbaum, 1974.

Henderson, P., & Snowdon, R. An experiment in structured programming. Bit, 1972, 12, 38-53.

Jackson, M. The Jackson design methodology. In P. Freeman & A. I. Wasserman (Eds.), Tutorial on software design techniques (2nd Ed.). Long Beach, California: IEEE Computer Society, 1977, 219-234.

Jackson, M. A. Principles of program design. New York: Academic Press, 1975.

Jeffries, R., Polson, P. G., Razran, L. & Atwood, M. E. A process model for missionaries-cannibals and other river-crossing problems. Cognitive Psychology, 1977, 9, 412-440.

Kintsch, W. The representation of meaning in memory. Hillsdale, New Jersey, Erlbaum, 1974.

Larkin, J. H. Problem solving in physics. Berkeley, California: University of California, Department of Physics, July 1977.

Ledgard, H. F. The case for structural programming. Bit, 1973, 13, 45-47.

Levin, S. L. Problem selection in software design (Technical Report No. 93). Irvine, California: Department of Information and Computer Science, University of California, November 1976.

Maier, N. R. F. Reasoning in humans. I. On direction. Journal of Comparative Psychology, 1930, 10, 115-143.

Mills, H. D. Top-down programming in large systems. In R. Rustin (Ed.), Debugging techniques in large systems. Englewood Cliffs, New Jersey: Prentice Hall, 1971.

Myers, G. J. Reliable software through composite design. New York: Petrocelli/Charter, 1975.

Newell, A. Artificial intelligence and the concept of mind. In R. C. Shank & K. M. Colby (Eds.), Computer models of thought and language. San Francisco: Freeman and Company, 1973, 1-60.

Norman, D. A. & Bobrow, D. G. On data-limited and resource-limited processes. Cognitive Psychology, 1975, 7, 44-64.

Otey, D. A., Ramsey, H. R., & Willoughby, J. K. Flight Test Oriented Precompiler System (FLTOPS): Design specification (Technical Report SAI-76-061-DEN). Englewood, Colorado: Science Applications, Inc., August 1978.

Paige, J. M. & Simon, H. A. Cognitive processes in the solving of algebra word problems. In B. Kleinmuntz (Ed.), Problem solving: Research, method and theory. New York: Wiley, 1966.

Parnas, D. L. On the criteria to be used in decomposing systems into modules. Communications of the ACM, 1972, 15, 1053-1058.

Peters, L. J., & Tripp. L. L. Comparing software design methodologies. Datamation, November, 1977, 89-94.

Ramsey, H. R. PLANS: Human factors in the design of a computer programming language. In Proceedings of the Human Factors Society 18th annual meeting. Santa Monica, California: Human Factors Society, 1974, 39-41.

Ramsey, H. R., Atwood, M. E., & Van Doren, J. R. A comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs (ARI Technical Report No. TR-78-A20). Alexandria, Virginia: U. S. Army Research Institute for the Behavioral and Social Sciences, 1978.

Ross, D. T., & Shoman, K. E., Jr. Structured analysis for requirements definition. IEEE Transactions on Software Engineering, 1977, SE-3, 6-15.

Sacerdoti, E. D. A structure for plans and behavior (Technical Note 109). Menlo Park, California: Stanford Research Institute, August 1975.

Shneiderman, B. A review of design techniques for programs and data. Software - Practice and Experience, 1976, 6, 555-567.

Simon, H. A. The sciences of the artificial. Cambridge, Massachusetts: The MIT Press, 1969.

Simon, H. A. The structure of ill-structured problems. Artificial Intelligence, 1973, 4, 181-201.

Simon, H. A. & Hayes, J. R. The understanding process: Problem isomorphs. Cognitive Psychology, 1976, 8, 165-190.

Stay, J. F. HIPO and integrated program design. In P. Freeman & A. I. Wasserman (Eds.), Tutorial on software design techniques (2nd Ed.). Long Beach, California: IEEE Computer Society, 1977, 174-178.

Stevens, W. P., Myers, G. J., & Constantine, L. L. Structured design. IBM Systems Journal, 1974, 13, 115-139.

Teichroew, D., & Sayani, H. Automation of system building. Datamation, August 1971, 25-30.

Warnier, J. D. Logical construction of programs. Leiden, Netherlands: Stenpert Kroese, 1974.

Wasserman, A. I. Case studies in software design. In P. Freeman & A. I. Wasserman (Eds.), Tutorial on software design techniques (2nd Ed.). Long Beach, California: IEEE Computer Society, 1977, 261-280.

Wertheimer, M. Productive thinking. New York: Harper-Row, 1945. (Revised 1959).

Wirth, N. Program development by stepwise refinement. Communications of the ACM, 1971, 14, 221-227.

Yourdon, E. Talk presented to Rocky Mountain Chapter, Association for Computing Machinery, Denver, Colorado, April 1976, and subsequent personal communication.

Yourdon, E. & Constantine, L. L. Structured design. New York: Yourdon, Inc., 1975.

APPENDIX A. POSSIBLE RESEARCH AREAS

EMPIRICAL COMPARISON(S) OF ALTERNATIVE DESIGN METHODOLOGIES

The conduct of a controlled experimental comparison of two or more of the surveyed design methodologies is an obvious candidate research activity. We believe that it is not a viable candidate, however, for several reasons. There are numerous difficulties associated with the cost, selection of subjects, experimental control, and definition of appropriate performance measures. Those difficulties are not insurmountable, though, and the experiment might be feasible. The real difficulty is that other approaches appear much more likely to produce useful information. Considered in pairs, the surveyed methodologies are either extremely similar, in which case a comparison hardly seems warranted, or they differ in many relevant respects. In the latter case (e.g., a comparison of Structured Design with Jackson's approach), it would probably be difficult to attribute an observed performance difference to any particular property of the methodologies, and it would certainly be difficult to generalize the result to other design problems, levels of designer experience, etc. While such a comparative study may eventually be relevant, it appears more cost-effective, at present, to undertake a basic program of exploratory studies. Such studies may help establish an understanding of the task domain, individual properties of design techniques, effects of designer experience, etc., which is needed to conduct this more explicit comparison in a useful way.

"PROPOSITIONAL" ANALYSIS OF SOFTWARE DESIGN INFORMATION

The manner in which the designer initially perceives the design task has obvious and significant effects on the design process. Previous research has shown that the manner in which text passages (Kintsch, 1974) and computer programs (Atwood & Ramsey, 1978) are perceived, or understood, can be represented in terms of a propositional hierarchy. Using the theoretical and empirical techniques developed in this research, the propositional structures constructed by a software designer to represent a design task would be investigated. Such research might result in methods for identifying likely sources of errors, metrics of design difficulty, measures of the expected difficulty of implementing a design, or notational schemes and guidelines to aid the designer in forming an accurate understanding of a given design task. An exploratory experiment along this line has been conducted as a part of the same research program which produced the present report, and is described by Atwood et al (1979).

TAXOMONY OF THE SOFTWARE DESIGN PROBLEM DOMAIN

On a intuitive level, there are different types, or classes, of software design tasks. For example, designing a compiler appears to be different than designing a business report generator. It may well be the case that each type of software design task is best approached by specific techniques, aids, etc. It seems even more likely that particular techniques may be precluded by certain problem properties. An understanding of the properties of the problem domain which are most relevant to designer problem-solving behavior would be quite helpful. We are attempting to do this, by analysis, in connection with another research effort, but empirical methods might be expected to yield more valid and useful results. A possible first step in the empirical development of such a taxomony of software design tasks might involve developing and analyzing questionnaires to be completed by experienced designers, and/or Delphi techniques. While the results of such studies must always be viewed with a certain healthy skepticism, they can provide valuable insight and direction to analytical and experimental inquiries.

TAXONOMY OF SOFTWARE DESIGN ERRORS

As suggested earlier in this report, the development of a useful taxonomy of software design errors, from the viewpoint of the problem-solving behavior involved, appears feasible and promising. This activity could proceed at two different levels. First, error analyses could be performed on designs developed in connection with other exploratory studies suggested herein. This approach is quite inexpensive and should yield useful insights into design errors associated with small design problems in a limited domain. The second approach is on a much larger scale, and would involve a similar analysis of archival design error data which have been collected by DoD and perhaps other agencies. The success of this effort will clearly be affected by the form and content of the archival data base, and further preliminary analysis should precede a firm commitment to undertake such a study.

ERROR-PREVENTIVE HEURISTICS

With greater insight into designer problem-solving behavior, and with a taxonomy of design error types, it seems reasonable to expect that useful error-preventive heuristics can be devised for at least some of the important error types.

SOFTWARE DESIGN "CRITICS"

"Critics" are techniques that a designer, or a design-aiding system, applies to ensure that modular decompositions are correct, in the sense that submodules are independent and appropriate for the design task. Based on a taxonomy of design errors, it may be possible to define an appropriate set of critics to correspond to these errors. This would allow the designer to construct modular decompositions only on the basis of local information and rely on the critics to ensure that global considerations are satisfied. Eliminating the need to explicitly consider global information reduces the demands for cognitive resources imposed on the designer and should allow the designer to perform more effectively and more efficiently.

QUANTITATIVE MODELING OF SOFTWARE DESIGN PROBLEM-SOLVING BEHAVIOR

Developing quantitative models of the performance of software designers on complex design tasks would be an extremely difficult task. However, research on planning coula provide the necessary background for the eventual development of such models. In addition to providing an explanation of design behavior, such models could also be used as research tools to investigate the effects of design aids, techniques, etc.

BEHAVIORAL ANALYSIS OF OMITTED SOFTWARE DESIGN METHODOLOGIES

This survey was intentionally restricted to a subset of the software-design-related methodologies currently in use. The intent of the survey was to include all current major methodologies incorporating manual methods, and to include one example (ISDS/HOS) of the computer-aided "macro-methodologies" as an aid to the development of a better perspective on the problem. In the process of this survey, however, we have become convinced that: (1) an analysis in terms of human problem solving can provide useful insights concerning individual methodologies, and (2) the information derived from single methodologies can significantly improve our overall perspective. It might be worthwhile, therefore, to extend the present survey to include the remaining "macro-methodologies" (e.g., SREM, SDS, SADT, ISDOS). A preliminary analysis should be done first, to quickly determine the degree to which these methodologies differ from those already surveyed. Only those which seem likely to contribute significantly to our understanding should be surveyed in detail.

ANALYSIS OF THE EFFECTS OF ISDS/HOS PROBLEM REPRESENTATION CONSTRAINTS

From the viewpoint of ISDS/HOS use, it would be desirable to have a greater understanding of the effects of the problem representation constraints resulting from the syntactically constrained decomposition rules and the AXES specification language. Empirical observation of the use of the method for actual design would help clarify this issue. Such a study should utilize moderately detailed protocol analyses, and should ideally involve several types of software design problems.

ANALYSIS OF THE EFFECTS OF JACKSON-WARNIER PROBLEM REPRESENTATION CONSTRAINTS

A similar problem exists with respect to the algorithmic methodologies. In particular, the constraints on data structure types may interfere with designer performance. An empirical study, using multiple problem types and protocol analysis, would do much to clarify the problem types to which these approaches are applicable and the degree to which the constraints interfere.

RELATION BETWEEN DESIGN SPECIFICATION AND DESIGN

As suggested earlier, we assume that a designer, when reading a design specification, constructs a propositional structure that represents the designer's perception of the software design task. This structure determines, in large part, the overall success of the design effort. The appropriateness of this structure, in turn, is largely determined by the manner in which the design requirements are specified. It may be the case, for example, that what are generally classed as "design errors" are actually due to errors or ambiguities in the requirements specification. Although there are several approaches to this problem, the most productive approaches would involve an analysis of the propositional structures underlying the requirements specification and a comparison of these structures with the internal representation constructed by the designer. A serious analytical study of this issue might also be productive.

ANALYSIS OF THE EFFECTS OF DESIGN DOCUMENTATION MEDIUM ON DESIGN PERFORMANCE

In a previous study (Ramsey et al, 1978) we found that the documentation medium (flowchart or program design language) used by a programmer for the design and specification of a computer program had a significant effect on the nature and quality of the resulting design. This may very well also be the case at the level of system design (modular decomposition, etc.). The documentation media used here include structure charts, HIPO charts, etc., as well as flowcharts and PDLs. This issue seems tractible, and could be addressed by methods similar to those employed in the previous study. Any such undertaking should, however, be preceded by a serious behavioral analysis of the documentation media and their role in the design process.

ANALYSIS OF THE RELATIONSHIP OF DESIGN TO PERFORMANCE IN SUBSEQUENT SOFTWARE DEVELOPMENT ACTIVITIES

Although design obviously affects subsequent stages in the software development cycle, these effects are not well understood. There are two questions that seem most relevant here. First, is it possible to classify design errors with respect to where they will be detected? For e.ample, do some types of design errors become apparent during programming while others remain undiscovered until coding or even testing? Second, how is an error determined to be a design error as opposed to, for example, an error in requirements specification, programming, etc.? The identification of the source of an error could aid in making an appropriate correction, as opposed to merely a "patch" at some later time in the software development cycle. These issues are, to a degree, related to that of developing a taxonomy of design errors, but are also subject to independent analytical study.

DEVELOPMENT OF IMPROVED DESIGN REVIEW TECHNIQUES

Various types of design review techniques are in common use. The general intent of these techniques is that communicating the design to others helps to ensure that the designer's perception of the problem and his efforts to solve that problem are correct and complete. Some types of review techniques, however, should be more effective than other. The principal questions appear to be (1) how should the review be organized, (2) how should the design be presented, and (3) when should the review take place? We feel that the last question is particularly important, but all are candidates for research.

DEVELOPMENT OF ADDITIONAL AUTOMATED SOFTWARE DESIGN AIDS

Throughout this paper, we have mentioned various potential software design aids. These aids are consistent with our current understanding of the software design process. A more detailed analysis of the problem-solving processes involved in software design can be expected to lead to the identification of the cognitive processes involved and the definition of aids that correspond to these processes. Such aids could, potentially, be concerned with forming appropriate internal representations of design problems, determining how to decompose a module into submodules, etc.

SYNTHESIS OF AN IMPROVED SYSTEM OF DESIGN TECHNIQUES

This is a very attractive long-range goal. The current survey has provided several useful insights, and it seems likely that the other research activities suggested here would provide sufficient information to justify this attempt. It is important, though, to recognize that this is not a short-term effort. While performance improvements may very well result directly from the application of the findings of the more basic research activities described here, the development of a new, comprehensive design methodology should await the establishment of a better fundamental understanding of software design behavior.