A COMPARATIVE STUDY OF SOFTWARE DESIGN METHODOLOGIES

by

Michael Tzu-cheng Yeh

SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE
DEGREES OF

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1982

Signature of Author_____
Department of Electrical Engineering and
Computer Science, May 7, 1982

Certified by_____
Stuart E. Madnick
Academic Supervisor, Department of Management

Certified by_____
Paul A. Szulewski
Company Supervisor, C. S. Draper Laboratory

Accepted by_____
A. C. Smith, Chairman
Department Committee on Graduate Students

A COMPARATIVE STUDY OF SOFTWARE DESIGN METHODOLOGIES

by

Michael Tzu-cheng Yeh

Submitted to the Department of Electrical Engineering and Computer Science on May 7, 1982 in partial fulfillment of the requirements for the Degrees of Bachelor of Science and Master of Science in Computer Science

ABSTRACT

A comparative study was carried out using four software design methodologies to design an experimental stream editor. The resulting designs were evaluated in terms of module strength and module coupling. Documentation of the designs were done with a design aid system called Design Aids for Real-Time Systems (DARTS).

The goal of this study is to examine one methodology in particular - the Systematic Design Methodology (SDM). SDM was one of the methodologies used in the above experiment, and through the evaluation of the editor designs several weaknesses in SDM were discovered. Also, as a result of the experiment, several extensions for SDM were found.

Thesis Supervisor: Professor Stuart E. Madnick

Title: Professor of Management

Thesis Supervisor: Mr. Paul A. Szulewski

Title: Member of Technical Staff,

C. S. Draper Laboratory

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my supervisors, professor Stuart Madnick (MIT), Mr. Paul Szulewski (CSDL), and Dr. Barton DeWolf. Professor Madnick, to whom I owe the original idea for this thesis, has been the most delightful person to work with. Paul has been most helpful both technically and spiritually throughout the project. I especially owe Paul a warm thanks for correcting the numerous errors in my thesis. To Bart I owe an extra special thanks for taking me on as a VI-A student, and arranging for me to work on a topic of my own choice.

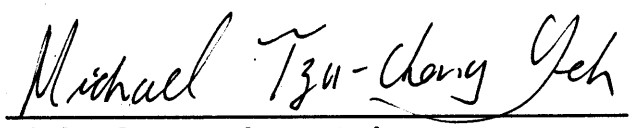
A sincere note of thanks also goes to Jim Lattin for the use of his SDM package. Also, I wish to thank Mei, Rich and Tarak (PhD candidates at the Sloan School) for the many fruitful discussions.

During the course of the research I had the priviledge to obtain the services of the CSDL library staff, without whose help this thesis would not be possible. Therefore I wish to thank the librarians for their efforts. I also had the priviledge to have my thesis proofread by Natalie Lozoski who corrected my numerous grammatical errors.

I wish to thank Susan Wang as well. Towards the end of the project when things became hectic, Susan provided me with encouragements and spiritual support. Her help is deeply appreciated.

Finally I would like to thank the Charles Stark Draper Laboratory for providing the financial support. Funding for this project has been most generously provided by the Internal Research and Development program.

I hereby assign the copyright of this thesis to the Charles Stark Draper Laboratory, Inc., Cambridge, Massachusetts.



Michael Tzu-cheng Yeh

Permission is hereby granted by the Charles Stark Draper Laboratory, Inc. to the Massachusetts Institute of Technology to reproduce any or all of this thesis.

CONTENTS

LIST OF FIGURES

Chapter I

INTRODUCTION

As computer systems become more complex, software development and maintainance costs begin to dominate. The problem is that software has become large and difficult to manage, and it is evident that the ad-hoc way of programming only makes matters worse. What is needed is a disciplined methodology for software development.

Past research has revealed various phases in the software development life-cycle (see figure 1).

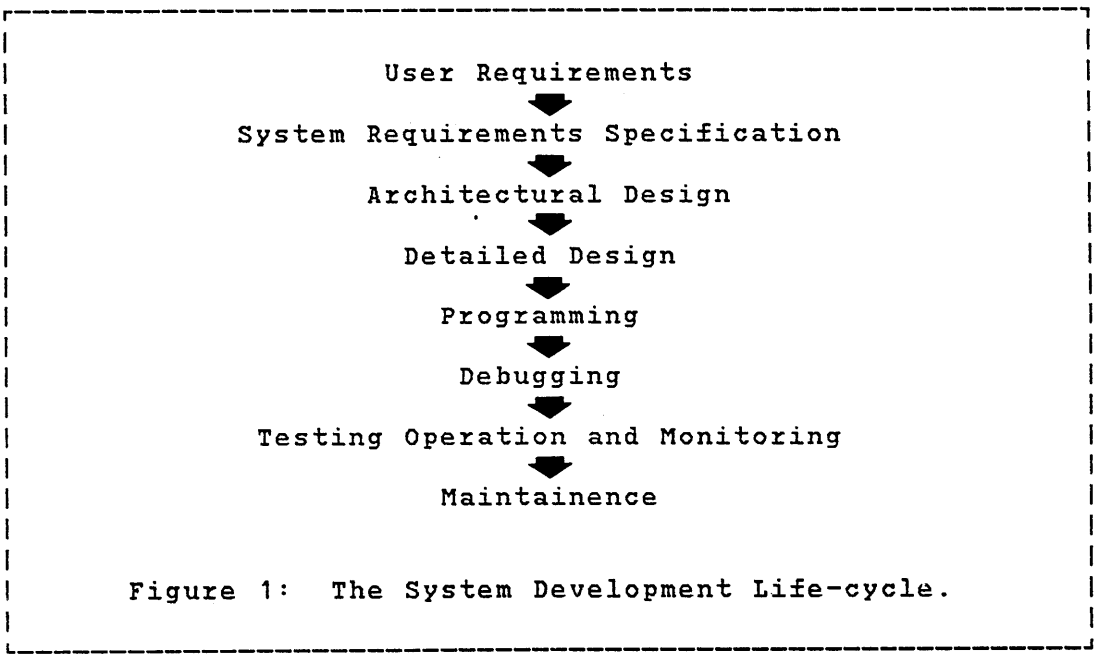


Figure 1: The System Development Life-cycle.

Furthermore, it has been found that problems arising in debugging and testing usually have their roots in an earlier phase. In the case of ad-hoc programming, the cause is the absence of any architectural design effort.

In recent years, an attempt has been made to attack the above problem. The result is a body of technology called "Software Design Methodologies". These methodologies differ in approach and in specific areas of application. Some provide techniques with which a design can be mechanically derived. Others merely give guidelines and require the designer to work out the design. Some can be applied best on programs of small size, others are meant for large system designs. However, these methodologies are united in their final objective - to fill the void between requirements specification and actual coding.

Essentially, design methodologies attempt to provide designers with a structural framework for software planning. The framework is designed to provide a way to manage the complexity of the design task (usually in the form of a divide-and-conquer tactic). It is also designed to embed desirable qualities in the software under development. But most importantly, it is designed to allow the designer to think about the program carefully before he starts the actual coding.

One methodology in particular is the integral part of this thesis: The Systematic Design Methodology (SDM). SDM

is a methodology developed by the MIT Sloan School of Management. It has been used in the design of a Database Management System (DBMS), an Operating System, a Budgeting System, and a test program preparation facility. So far the reports on the methodology have been encouraging. However, to get a better idea of SDM's potential, it would be beneficial to compare it against other methodologies.

Thus the goals of this thesis are to compare SDM against other methodologies, develop qualitative as well as quantitative evaluations of methodologies, and to suggest extensions or improvements for SDM.

The project proceeds from an examination of the general design problem to specific methodologies. Five stages are identified:

1. Design Theories.
2. Design Techniques,
3. Design Methodologies,
4. Comparing design methodologies through actual design,
5. Extending SDM.

The rest of this thesis is organized in a similar manner. Chapter II presents a few of the popular views on the nature of design problems. Chapter III presents a list of design methodologies. Chapter IV describes an experiment in comparing several design methodologies. Finally, the concluding chapter presents suggestions for modifying and extending SDM.

Chapter II

DESIGN THEORIES

Before a study of design methodologies can be undertaken, it is necessary to understand design from a broad perspective. Peter Freeman speaks of this need [FREE77]:

> "Without an understanding of broad classes of phenomena, one is condemned to understand each new instance by itself."

Indeed, the essence of science is to discover unifying characteristics in the environment around us. Having this knowledge, we can predict the outcome of specific actions. Furthermore, we can use this knowledge to choose those actions which yield desired results.

The purpose of this chapter, therefore, is to present popular views on the nature of design, and how it relates to software development. The first section describes three different viewpoints on design. These viewpoints are collected from three different disciplines - Architecture, Civil Engineering, and Artificial Intelligence. The second section puts design in perspective with software development.

2.1 DESIGN THEORIES

Although software design is a recent development, the subject of design in general has been around for a long time. Consider the Egyptian Pyramids and Roman Cathedrals, these are all complex structures which required careful planning and design. Therefore it is logical to expect a significant understanding of design from the fields of Architecture and Civil Engineering. In fact, the first two views on the nature of design are taken from these fields. Christopher Alexander is an architect who received his education from MIT and Harvard. His views on design [ALEX64] have become the backbone of SDM. Marvin Manheim is a professor of Civil Engineering at MIT. Professor Manheim's work in Urban Planning [MANH64,67] also represents an unique view on design.

A third area from which interesting results have emerged is Artificial Intelligence. Professor Herbert Simon of Carnegie Mellon looks at design from a psychological perspective. His views can be found in [SIM069].

2.1.1 Discovering the Structure

The first approach to design presented here is advocated by Christopher Alexander [ALEX64]. It is of special interest because it is the basis for the SDM approach.

In Alexander's view, the major difficulties in design are caused by the complexity of the design problems. The number

of competing factors a designer must consider has become so large that it is not possible to keep track of all of them. Therefore, it is necessary to have a systematic way of reducing the design problem into manageable pieces. These separate pieces can then be attacked one by one and a more satisfactory solution to the original problem can be produced.

Before proceeding further, it is necessary to understand what causes complexity. It is not merely the size of the problem, for surely we can solve a thousand arithmetic problems easily; when each problem has nothing to do with the others. Alexander points out that the real culprit is the interaction among requirements. In other words, the solution to one requirement is often dependent on the solutions to many of the other requirements.

Thus, Alexander proposes a way to model the interactions between requirements, and extract the optimal decomposition from that model. The basic idea is to model the situation with a graph. The requirements are the nodes, and the interaction between requirements are the links. The designer can then apply an algorithm to the graph and find the decomposition which minimizes the interaction among the resulting components. This decomposition allows the designer to deal with each component independently, thus reducing the complexity faced by the designer. This process is, in effect, the search for the problem structure, where structure is de-

fined to be the underlying components and their interactions.

2.1.2 Problem Solving

Manhiem [MANH64,67] expressed the view that complexity management is the major difficulty in design. The variety of options open to a designer are so numerous that it is difficult just to list them exhaustively. To find the optimal solution is somewhat an unrealistic goal.

Manheim proposes a model called the Problem Solving Process (PSP). The basic activities of the PSP are search and select. Search generates a set of alternatives and select makes a decision on which alternative to follow.

The search and select procedures are used repetitively, until a satisfactory solution is found. More specifically, the search procedure generates a set of options which it feeds into the select procedure. Select then employs some evaluation techniques to assign a priority ordering to the options. The option with the highest priority is pursued further, and the search-select process is repeated until a solution is found. The designer must decide when the options provide a sufficient decision space and the solution is satisfactory.

Manheim also suggests that a computer be used in PSP, especially the graphic capabilities because the human mind can work much better with a picture than with sentences. Fur-

thermore, the entire PSP can be automated to increase its efficiency as well as its effectiveness. Moreover, Manheim asserts that the complexity of design problems precludes a provably optimal solution. Nevertheless, one can still develop an optimal design process.

2.1.3 Reducing the Difference

In yet another view, Herbert Simon [SIM069] defines design as a process of reducing the difference between the present state and the desired state. At every stage of the design, some sensors (e.g., human or machine) describe the state of the world, then the state is compared to the desired state. A set of differences is generated, and solutions are devised to resolve the differences.

Simon models the above process as a General Problem Solver (GPS). At any moment in the design, the GPS asks the question, "What shall I do next?". To answer this question, the GPS stores information in its memory about states of the world and about actions. It also stores associations between changes in states and the actions that bring about these changes. Now the GPS's question can be answered by searching for a series of actions which produce the desired changes in the state of the world.

The difficulty in the above scheme lies in the search procedure. Simon suggests a breadth-wise search. The GPS starts with a set of alternative actions, and assigns a val-

ue to each. The value corresponds to the relative likelihood that the desired state can be reached through that path. The GPS investigates several of the most promising actions in parallel. An alternative is eliminated as more information is gathered, and the path begins to look less promising than others. In effect, the GPS is building up a decision tree in its memory. It gathers more and more information about the design as decisions are explored. This process continues until a path to the desired state is found.

Note that Simon does not incorporate any notion of optimization in his model. In fact he believes that design can be best described by the word "satisficing". In other words, most designs are merely a satisfactory solution to the problem. Optimization is not possible because of the enormous complexity of most design problems.

2.2 SOFTWARE DESIGN AND THE SYSTEM LIFE-CYCLE

Early in the growth of software engineering research, it was found that typical software systems go through 9 different phases:

1. User requirements specification,
2. System requirements specification,
3. Architectural design
4. Logical design (or detailed design),
5. Programming,

6. Debugging,
7. Testing,
8. Operation and monitoring,
9. Maintainence.

Further research revealed that the greatest costs were incurred when components are put together and debugged. This phenomenon is caused by the lack of overall system planning or architectural design. Thus, software design can be effectively defined as the bridge between requirements specification and programming. It is the activity of looking ahead and planning out the organization of the program.

Software design can be further divided into architectural design and logical design. Architectural design deals with the entire system. It usually involves decomposition, setting up communication between components, and making decisions that have global effects (e.g., creating a module to handle file I/O, or an executive module to do dispatching). In effect, the architectural design is analogous to the bone structure in the human body. Logical design, on the other hand, deals with local issues. It involves the details of each component (e.g., how data is taken out of the file and put into a buffer, or what to do when the input is the number 0). This subphase can be thought of as the flesh which envelops the framework of bones. Together the flesh and the bones perform the functions of the system.

Chapter III

THE DESIGN METHODOLOGIES

The last chapter identified the basic issues in software design. This chapter looks at the technology which addresses these issues. The technology is called "Software Design Methodology".

Formally, a software design methodology has the following properties (see also [HUFF79]):

1. A structured approach. This property requires the methodology to have a definite view on how the design should proceed. It may be top-down, bottom-up, or a mixture of the two. The important thing is that the methodology provides the designer with a direction. It can be thought of as a compass, a top level strategy, or an overall plan.
2. A procedure. The procedure specifies the detailed steps in carrying out the structured approach. It adds a sense of mechanization to the design process. More specifically, the procedure provides clear instructions and specifies well defined tasks. In essence, the procedure is like an assembly line, and the designers are the workers who actually put the parts together.

3. Tools. Tools are mechanical helpers for the designer. They may vary from simple graphic representations of design to design languages, or even programs which produce programs.

The rest of the chapter describes a collection of methodologies. The first 4 sections describe methodologies that are used in the experiment (chapter 4). The final section contains descriptions of 4 other methodologies, briefly overviewed and included for completeness.

3.1 HIERARCHICAL DEVELOPMENT METHODOLOGY (HDM)

HDM ([LEVI80], [ROBI79], [SILV79]) is primarily an aid in structuring and recording decisions made during the design process. Based on a decision model, HDM employs several different software engineering concepts for the structuring and the grouping of decisions. HDM also emphasizes the need for flexibility in a design methodology. Citing instances where an error in early decisions produced catastrophic results, HDM proposes that some methodologies force designers to make premature decisions. Therefore HDM strongly emphasizes flexibility in its proceduralized version.

The basic idea in HDM is the decomposition of a program into levels of abstract machines. The machines are linearly ordered, and each machine can only communicate with the machines directly above and below itself. Within each abstract machine further decomposition takes place. Each machine also has its own abstract data structure and a set of abstract operations. These operations are implemented on the machine of the next lower level. The data structure of each machine cannot be accessed from the outside, except through the defined operations of that machine.

3.1.1 The Decision Model

HDM views software design as a process of decision making. At any stage of its development, the software results from a sequence of decisions. Each decision depends only on the decisions occurring before it. Although decisions occur linearly in time, as in figure 2, in reality each decision depends only on a subset of the decisions occurring before it. This can be thought of as partial sequences, possibly intersecting at some point (see figure 3).

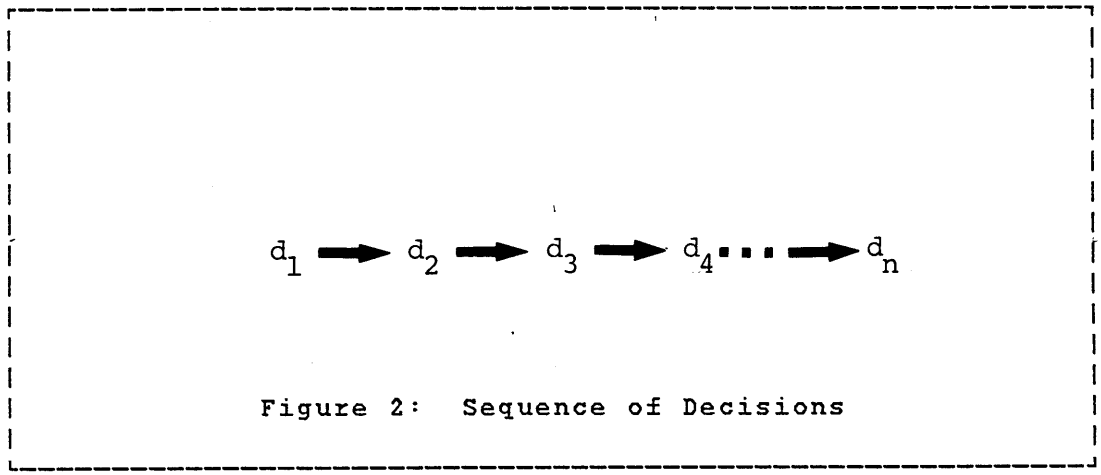


Figure 2: Sequence of Decisions

Two issues of importance in the decision model are the time when decisions are made, and the dependency between decisions. Organizing the decision-making process to address these issues, HDM uses the following concepts as guidelines:

1. Abstraction - Abstraction is defined as the process of isolating a subset of the properties characterizing a system, such that the system can be understood

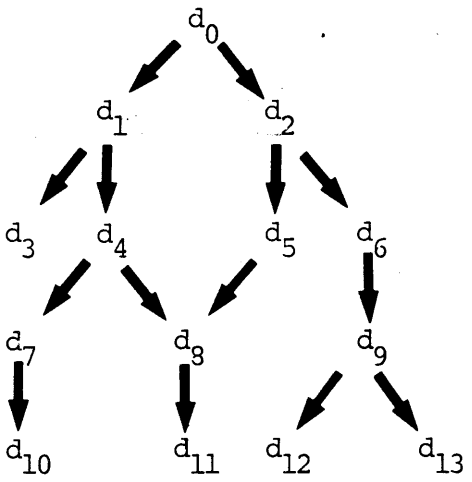


Figure 3: Dependencies Among Decisions Form a Tree-like Structure

more easily and the system can be used as if it possessed only that subset of properties. The concept of levels of abstraction provides a way to describe the interdependence between decisions (i.e., each level only depends on the level before it), and to minimize that dependence (i.e., abstract only the necessary features of decisions on the level below).

2. Hierarchies of Abstract Machines - The next step in applying the idea of abstraction is the concept of hierarchy. Any programming problem can be thought of as a set of instructions to an abstract machine. The

abstract machine is in turn realized by another abstract machine, and so on. The design eventually reaches a level where the abstract machine can be realized by a physical machine (e.g., a programming language processor). In HDM, an abstract machine has the following properties.

a) A set of internal data structures which define the state of the machine.

b) A set of operations by which the data structures can be manipulated and changed.

c) A data structure internal to each machine which can be accessed only through the defined operations of that machine.

The hierarchy of abstract machines provides a structure for decision making. It also enables the grouping of decisions into individual machines.

3. Modularity - As defined by HDM, modules are parts of a system which can be easily replaced. In order to be easily replaceable, HDM requires a module to have a well-defined external interface. This allows another module satisfying the requirements of the interface to replace the original module without any knowledge of internal details. Modularity is used to localize the effect of decisions and to minimize dependencies between decisions of different modules.

4. Formal Specification - Formal specification in HDM is meant to provide a complete documentation of all decisions made during design. Each module is precisely described by a specification language called SPECIAL (SPECIfication and Assertion Language). A further goal of formal specification is machine-checked consistency and well-formedness.

5. Formal Verification - Formal verification refers to techniques whereby programs can be mathematically proven correct. HDM employs the inductive assertion technique developed by Floyd ([LEVI80]). The purpose of verification is to provide a means for determining the consistency among decisions.

6. Data Representation - Many situations arise in programming where mathematical concepts are modelled by data structures. Verification of the model is often difficult, therefore a technique called "data representation function" is used. This function defines a mapping between mathematical concepts and data structures. Verification can be performed by examining the mappings.

3.1.2 The Procedure

HDM warns against strict step-by-step procedures. An erroneous decision made in an early step often did not get discovered until it became too costly to change that deci-

sion. Instead, HDM provides guidelines for decision-making and a seven stage development scheme. The development scheme is meant to provide a series of milestones by which progress can be measured. Although the seven stages appear ordered in time, HDM emphasizes that it is not necessary to carry out the design in that order. In fact HDM encourages the designer to follow the natural course of the decision making process.

The seven stage scheme is as follows:

1. Conceptualization - Conceptualization is the process of identifying the design problem, and stating the problem as requirements. This is also known as requirements specification.
2. External Interface Definition - Defining the abstract machines that interact with the outside world. This consists of the top-most machine in the hierarchy and the bottom-most machine in the hierarchy. Also done during this step is the decomposition of these machines into modules.
3. System Structure Definition - Defining the intermediate abstract machines and decomposing them into modules. Intermediate machines can be defined in three different directions: Top-down, bottom-up, or middle-out.
4. Module Specification - This step is carried out using SPECIAL (SPECIfication and Assertion Language). Pre-

cise and explicit descriptions of decisions made in stages 2 and 3 are recorded in SPECIAL.

5. Data Representation - Define the data structures of every non-primitive machine (i.e., every machine except the bottom most machine) in terms of the data structures of the machine on the next lower level.

6. Abstract Implementation - Implement operations of each non-primitive machine as an abstract program running on the machine of the next lower level. The abstract programs are written in ILPL (Intermediate Level Programming Language).

7. Concrete Implementation - Translate abstract programs, written in stage 6, into executable code. This can be done by translating ILPL programs into a modern programming language, or directly compile ILPL into machine code.

3.2 JACKSON METHODOLOGY

The Jackson Methodology [JACK75] is based on the philosophy that the program structure should match the problem structure as closely as possible. To discover the problem structure, Jackson theorizes that the data structure reflects the problem structure very accurately. Once the input and output data structures are found, a program can be created to transform input data into output data. Furthermore, data structures can be expressed with the same kinds of components as those used for the program structure (i.e., the sequence component, the iteration component, and the selection component). This observation enables the designer to build the data structure out of components which can be directly translated into program components.

However, anomalies between input and output data give rise to situations where it is impossible to directly translate data structure into program structure. Jackson identifies two typical situations: structure clash and backtracking.

Structure clash occurs when an input data structure does not match the output data structure. This mismatch can be caused by a different ordering among components, by a many-to-one mapping, or by a one-to-many mapping. For example, if the input data are rows of a matrix, and output data are columns of the matrix, then a structure clash occurs.

Backtracking problems arise when execution of a task is dependent upon a condition, but the condition cannot be checked unless the task is executed. For example, when performing a table lookup, the condition for retrieving the value of an entry is that the entry should actually exist. However, one cannot determine whether the entry exists or not unless a lookup is performed. In this case, one would have to perform the task first, then determine the value of the condition. If the condition is false then backtrack to a point before the task was performed, and take another path.

3.2.1 The Procedure

Jackson's methodology can be divided into three stages:

1. Define the input and output data structures.
2. Create the program structure from the data structures. In other words, transform data structure components into program modules.
3. List the program tasks as executable operations, and allocate each task to a program component.

3.2.2 Data Structure Definition

Jackson's methodology provides a graphical representation of the three structuring components. They are given in figure 4. The components are represented in a hierarchical fashion. This implies that the program structure is also hi-

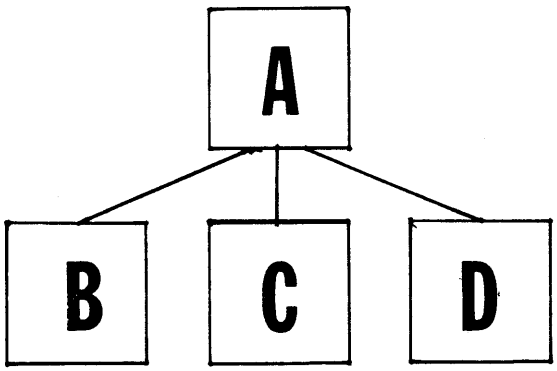
erarchical, because program structure is expressed with the same components.

3.2.3 Program Structure Definition

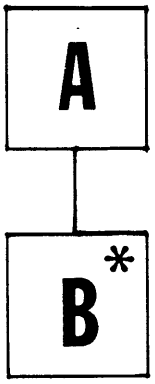
Jackson also provides a general heuristic for solving each of the problems caused by data anomalies. Structure clash is solved by program inversion. Recall that structure clash is caused by inconsistencies between input and output data structures. The program inversion technique breaks input data into elementary components, then recombines them to form the output data. For example, in the matrix problem considered earlier, the rows could be broken into individual elements, then recombined later to form the columns.

Backtracking problems are solved by a three step process:

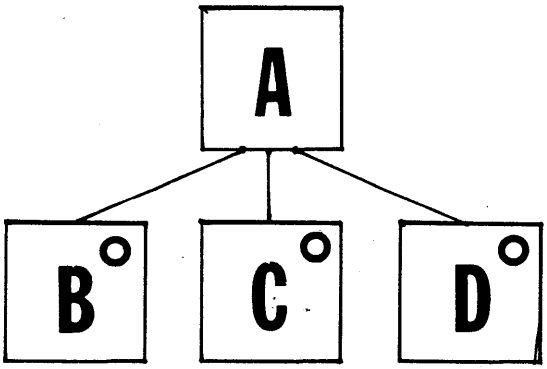
1. Structure the problem as a sequence, ignoring the impossibility of evaluating the condition, and execute one branch of the conditional. Recall that a backtracking problem arises when the condition cannot be evaluated without first performing a task whose execution depended on the condition. This step is equivalent to executing the then-clause.
2. Determine whether an incorrect choice has been made, then either proceed or do a conditional transfer to the else-clause.
3. Consider the side effects caused by execution of the then-clause, and make appropriate corrections before the original else-clause is executed.



The Sequence Component



The Iteration Component



The Selection Component

Figure 4: The Three Basic Structuring Components

3.2.4 Unique Features

Jackson makes several claims about the methodology. First, the methodology provides a simple criterion by which to judge whether a program structure is correct. More specifically, if the program structure matches the correct data structure, then the program structure leads to a "good" program (but the critical step of constructing the data structures is not at all trivial). Second, the methodology has a unifying principle, and every aspect of the methodology can be validated from that principle. The principle is: Data structure reflects problem structure, and a good solution should reflect the problem structure. Third, the methodology is easy to learn, easy to use, and does not depend on an individual designer's ability. That is, the process of discovering data structures and transforming them into a program structure is very mechanical, therefore requires little ingenuity. Finally, the methodology produces a hierarchically structured program. According to Jackson this is synonymous to good design.

3.3 STRUCTURED DESIGN (SD)

The SD [YOUR79] approach is inspired by a study of the morphology of systems. It is found that low-cost systems are usually shaped like a mosque (see figure 5).

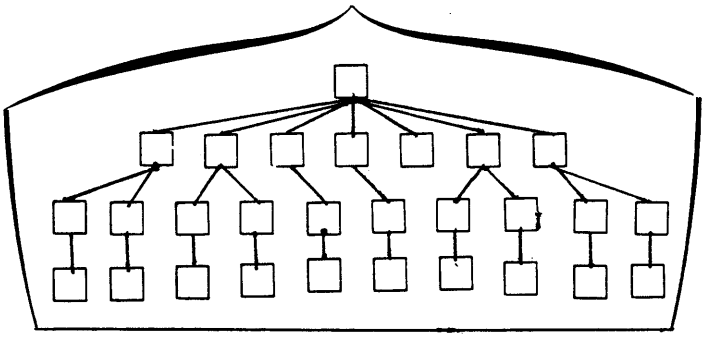


Figure 5: The Mosque Shape of Low-cost Systems (adapted from [YOUR79])

Furthermore these low-cost systems are centered around various aspects of the system functions. Most important of the different types of centers are the transform center and the transaction center. The transform center consists of those modules which transform the input data stream into the output data stream. The transaction center refers to places in the system where the data stream is split into many sub-streams.

To identify these centers, SD uses the Data Flow Graph (DFG). The DFG is a pictorial way to describe the transformation of input data into output data.

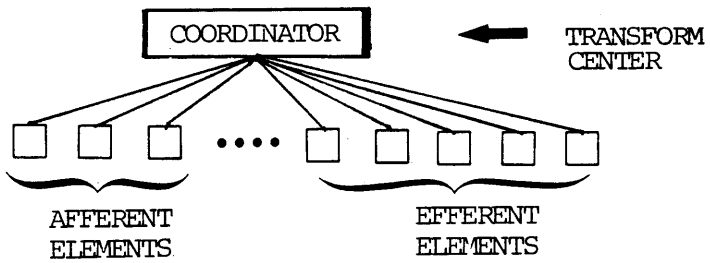
Having identified these centers, SD provides two techniques for constructing a system architecture: Transform Analysis and Transaction Analysis. These techniques will be discussed in the following sections.

3.3.1 The Procedure

1. Translate the design problem into a DFG. (See [YOUR79] chapter 10 for examples of DFG)
2. If the DFG represents a sequence of transforms each of which must be performed, then use Transform Analysis. If the DFG represents a selection among many alternative routes then use Transaction Analysis.

Transform Analysis:

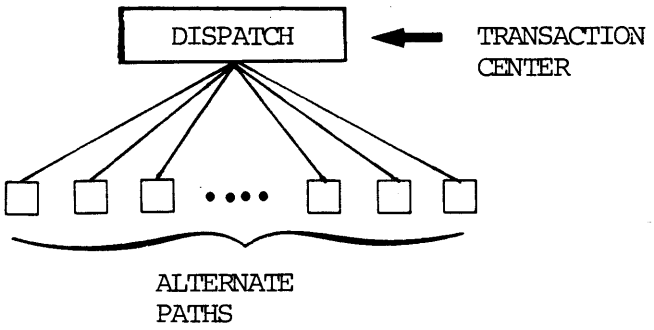
1. Identify the input stream and the output stream. These elements of the DFG are also called afferent data elements and efferent data elements respectively.
2. Identify the transform center. The transform center consists of one or more transform elements.
3. Translate the DFG into a system structure as follows:



4. Go back to step 1 and apply the procedure to each module in the system.

Transaction Analysis:

1. Identify the transaction center.
2. Translate the DFG into a system structure as follows:



3. Go back to step 1 and apply the procedure to each module.

3.3.2 Data Flow Graph (DFG)

The DFG can be thought of as a decomposition tool. It allows the designer to describe the processing of input data in terms of well defined transformations. For example, figure 6, shows the processing of inputs from a medical monitor device.

However, it is not always easy to translate a design problem into a DFG. SD does not provide a systematic way to

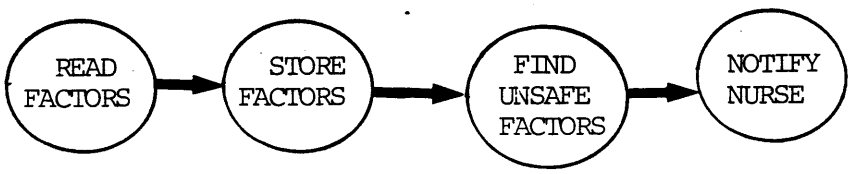


Figure 6: An Example of DFG (Adapted from [YOUR79])

solve this problem. Instead, a set of guidelines are given. They are summarized below:

1. Work from different directions: Input to output, output to input, middle out. When one approach fails, switch to another.
2. Do not show control logic. In other words, just show what needs to be done to the data, not how it is done.
3. Ignore initialization and termination.
4. Label data elements.
5. Use * (AND) and + (OR) symbols to indicate the type of data stream splitting.
6. Do not show unnecessary details, but if in doubt, show more detail rather than too little.

3.3.3 Select a Technique

The difference between Transform Analysis and Transaction Analysis can be viewed in another way. Transform Analysis is applied whenever the DFG represents an AND relationship between the bubbles in the DFG. Transaction Analysis, on the other hand, is applied when the DFG represents an OR relationship between the bubbles in the DFG.

3.3.4 Transform Analysis

The identification of afferent, efferent and transform elements is not a clearly defined task. When and where the input data becomes output data depend a great deal on the taste of the designer. SD offers the following definitions toward the resolution of the above problem:

1. Afferent data elements are those high-level elements of data that are furthest removed from physical input, yet still constitute inputs to the system.
2. Efferent data elements are those high-level elements of data that are furthest removed from physical output, yet still constitute outputs to the system.
3. The transform elements are everything in between the afferent elements and the efferent elements.

The translation of the DFG into a system structure is a mechanical process. Each bubble in the DFG becomes a module and a coordinate module is created to manage them.

3.3.5 Transaction Analysis

The transaction center is much easier to identify than the transform center. The unmistakable characteristic of a transaction center is the two or more "OR" branches extending from a bubble.

Having identified the transaction center, it is a simple step to create the system structure. A dispatch module is created to perform the selection among the alternative paths.

3.4 SYSTEMATIC DESIGN METHODOLOGY (SDM)

SDM's philosophy is adopted from Alexander's work on design theory [ALEX64]. Essentially, this approach views design as a process of discovering the inherent structure of the problem. Structure being the set of underlying sub-problems and how the sub-problems interact with each other. This idea of structure can be applied to each sub-problems as well. In other words, the sub-problems themselves can have sub-problems.

Looking at it from a software perspective, the SDM philosophy involves discovering a decomposition of the design problem into modules, or sub-problems. These modules are themselves decomposed into still smaller modules, and this process could be carried to any degree of detail.

To carry out the top-down decomposition, SDM uses a graph model (see [HUFF79]). The idea is to model individual requirements as nodes, and interactions between nodes as links. However, because interactions can be weak or strong, every link is given a weight between 0 and 1.

Having translated the design problem into a graph decomposition problem, the next step is to formulate the decomposition criteria. The SDM approach is to maximize the following conditions:

1. Strong interdependencies between members of a group.
2. Weak interdependencies between members of different groups.

In summary, SDM's procedure can be divided into 4 steps:

1. Specification of the functional requirements,
2. Determine the degree of interdependency between all pairs of requirements,
3. Represent the requirements as nodes and the interdependencies as links between nodes,
4. Apply a decomposition algorithm to the graph.

Each of these steps is further expanded upon in the following subsections.

3.4.1 Requirements Specification

Requirements specification is capability oriented, not process oriented. Put another way, specifications should be non-procedural, it should not state how something is to be done, merely what is to be done.

Specifications must have the following three characteristics

1. Unifunctionality - Each statement describes a single function to be incorporated in the target system.
2. Implementation Independence - Each statement should be non-procedural. In other words, each statement should specify what needs to be done, not how it is to be done.
3. Common Conceptual Level - All requirements should be on the same level of generality.

A set of seven requirement statement templates was developed to meet the above criteria, and also to ease the trans-

lation of requirements stated in other forms/languages. A list of the templates is given below (adapted from [HUFF79]).

The 7 Templates:

1. Existence

   There (can/will) be <mod> <obj>

2. Property

   <mod> <obj> (can/will) be <mod> <property>

3. Treatment

   <mod> <obj> (can/will) be <mod> <treatment>

4. Timing

   <mod> <obj> (can/will) <timing relationship> <mod> <obj>

5. Volume

   <mod> <obj> (can/will) be <order statement> <index> <count>

6. Relationship (Subsetting)

   <mod> <obj> (can/will) contain <mod> <obj>

7. Relationship (Independence)

   <mod> <obj> (can/will) be independent of <mod> <obj>

3.4.2 Interdependency Assessment of Requirements

The next step is the determination of interdependencies between all pairs of requirement statements. These interdependencies are expressed as weights. A weight is a number

between 0 and 1. The smaller the number the weaker the interdependence, while a larger number has the opposite effect. Therefore the value of 0 means absolute independence, while a value of 1 means absolute dependence.

Through experience, SDM users have discovered that most interdependency assessments fall into three categories: Strong(0.8), average(0.5) and weak(0.2). This three way breakdown is much easier to use while still providing a meaningful measure of interdependency.

There still remains the question of how interdependencies are determined. SDM does not provide a systematic method to meet this need. However, SDM offers the following points as guidelines:

1. The designer must have in mind at least some idea of how to implement related requirements. From this implementation scheme, the designer can then give an assessment of interdependency. If it is possible to develop several schemes for implementation, then the designer can evaluate the different schemes and assign weights according to how important a pair of requirements is within each of the schemes. In other words, if a pair of requirements seem to be related in all the schemes, then a weight of 0.8 should be assigned to it.
2. In other cases, trust intuition.

3.4.3 Graph Modelling of Requirements

Each requirement stated using the template method can be represented by a node, and the interdependencies can be represented by links between nodes. The weights can be recorded in a square matrix, and the square matrix can then be used in the decomposition process.

3.4.4 Graph Decomposition Techniques

Several different clustering algorithms have been developed in connection to SDM. Among them are the Interchange Algorithm and the Hierarchical Clustering Techniques. Detailed discussion of the algorithms are found in [HUFF79], [WONG80] and [LATT80].

3.5 OTHER DESIGN METHODOLOGIES

In this section we describe 4 other methodologies. These methodologies were not chosen for the design experiment for various reasons. However, they still provide interesting insights to software design.

3.5.1 Event-based Design Methodology (EDM)

EDM [RIDD79] is a methodology based on the top-down design approach, and it is particularly good for the architectural design phase. The methodology consists of iteratively applying four basic steps. These steps form a stage of the design. The input to a stage is a partial design, corresponding to the current state of the design effort.

The four steps are as follows:

1. Identify events occurring in the part of the system which is being considered in the present stage. An event could be a "happening" in the system which requires the system to respond in some specified fashion. Or an event could be an action taken by the system in response to some stimuli. In other words, events definition is specification of the input and required system behavior in response to these inputs.
2. Establish constraints on the occurrences of events. This step is basically the specification of correct system behavior in response to stimuli. The difference between this step and step 1 lies in the scope

of consideration. In step 1, desired system behaviors are stated in term of what needs to be done. It does not consider how it is done nor does it consider interactions with other parts of the system. Therefore by establishing constraints such as the sequence in which events (i.e., system responses) should take place, the designer can gradually evolve requirements into functional modules.

3. Define components which would perform tasks specified by the events. This step defines a module for each of the events that correspond to a desired system behavior. The defined module can be completely new or it can be an existing module created during a previous stage.
4. Define the necessary interactions between modules. The interactions would be subject to the constraints of step 2. The developers of this methodology plan to provide verification capabilities based on this step. That is, if during this step within every stage, the designer can prove he has satisfied all the constraints, then it could be the basis for a proof of correctness.

This methodology presents a variation to the strictly top-down approach described earlier. The basic decomposition procedure has been extracted from the top-down approach, and given more freedom in its use. The four step

procedure described above is actually a formalization of the basic decomposition process in the top-down approach. Moreover, this formalized procedure can be treated as a basic building block with which a complete design procedure can be constructed.

A direct consequence of the above is the increased flexibility over a strictly top-down approach. In a top-down approach, the design problem is decomposed in an orderly way, but in this methodology the desomposition step can be applied to any subproblem at any time. This is true because the only input to the four step procedure is a partial design. This partial design need not be the result of a series of decomposition steps, because it could be independently defined. This feature is especially useful in combining the top-down approach with other approaches.

3.5.2 Higher Order Software (HOS)

HOS [HAMI76] was born out of many years of experience in designing and implementing NASA projects. The methodology is oriented toward large real-time systems. Because of the complexity usually associated with such systems, HOS is supported by a system of tools called the Integrated Software Development System (ISDS). The designer interacts with ISDS to produce a specification of the target system. Then the specification is fed into the Design Analyzer and the Structuring Executive Analyzer. The output of the Analyzers is a

complete system specification. Given this specification the designer can perform architectural design.

The HOS procedure can be split into three phases:

1. Specify requirements according to a set of axioms.
2. Feed specifications into the Designer Analyzer and Structuring Executive Analyzer.
3. Produce architectural layers.

HOS provides a metalanguage called AXES with which to specify the requirements. The metalanguage is based on six axioms to which the designer must adhere. They are as follows:

1. A given module controls the invocation of the set of valid functions on its immediate, and only its immediate, lower level.
2. A given module is responsible for elements of only its own output space.
3. A given module controls the access rights to each set of variables whose values define the elements of the output space for each immediate, and only each immediate, lower level function.
4. A given module controls the access rights to each set of variables whose values define the elements of the input space for each immediate, and only each immediate, lower level function.
5. A given module can reject invalid elements of its own, and only its own, input set.

6. A given module controls the ordering of each tree for the immediate, and only the immediate, lower levels.

After the requirements are specified in AXES, they are fed into the analyzers. The analyzers check for violation of the axioms in the specifications. The Design Analyzer checks for static consistency, and the Structuring Executive Analyzer checks for dynamic consistency.

In addition to consistency checks, study can be done in the following areas: Fault tolerance, error detection, timing and accuracy, security requirements, system reliability.

Having the analyzed specifications, the designer can begin the architectural design. In this phase the designer develops a system architecture and allocates the available resources. The architecture is constructed in layers, similar to the concept of a hierarchy of abstractions. Then the resources are allocated to the system components using the Resource Allocation Tool (RAT). The RAT uses the architectural form to analyze the target system in terms of time and memory optimization. An optimal module configuration is produced by the RAT.

3.5.3 Logical Construction of Programs (LCP)

The LCP philosophy [WARN74] is that program structure should be derived from the input and output data structures (similar to Jackson's methodology). The data structures are constructed from three basic elements: Sequence, Iteration,

and Selection. Furthermore, the data structures are expressed in a hierarchical format called the data structure diagram.

Having defined the data structures, the designer constructs a flow chart. The flow chart would express the logical sequence of actions to be performed. The chart is then translated into an instruction list, which finally gets translated into code.

LCP does not provide a step by step procedure, however, the following steps are implied:

1. Define input and output data structures.
2. Construct a flow chart based on the data structures.
3. List operations to be performed by each part in the flow chart.
4. Translate the list of operations into code.

The LCP data structure diagram is based on the building blocks shown in figure 7.

The data structure diagrams can be directly translated into a flow chart. The sequence element becomes a sequence of nodes. The iteration element becomes a loop and the selection element becomes a decision box.

Finally, the designer generates a list of operations for each node in the flow chart. The list of operations is intended to be a buffer from the confusion which may occur if the designer had translated the flow chart directly into code. By listing all the operations for each box of the

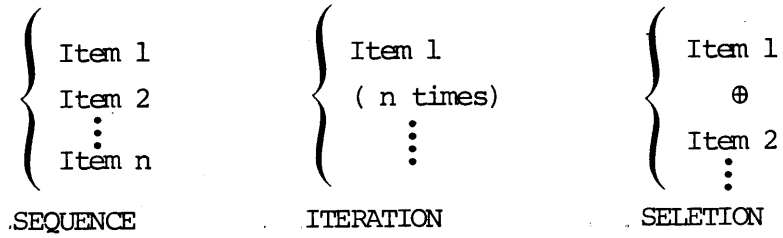


Figure 7: Data Structure Definition for LCP

flow chart, the designer can take the design one step lower in detail without being tied down by the details of a programming language.

3.5.4 WELLMADE

The primary goal of WELLMADE is to add a mathematical dimension to the conventional top-down approach. The designers of WELLMADE [BOYD78] claim that the major difficulty in software development is the lack of mathematical discipline. Therefore, WELLMADE has focused on proving the correctness of a program.

The general approach to proof of correctness is based on Dijkstra's idea of predicate transformer [BOYD78]. Basically, a program can be regarded as a transformation of input

into output. From this point of view, one can then work backwards from the output states and derive all the possible inputs under the transformation. To do this an explicit statement of all output states and a mathematical statement of the transforms is needed. If the resulting input state space, derived from the above process, contains the specified legal input states, then the program performs correctly for all legal inputs.

In terms of system architecture, WELLMADE advocates a layered design. The concept is similar to the idea of stepwise refinement. The design begins from a highly abstract machine with abstract data and abstract program declaration. Then the modules in the abstract machine are decomposed in greater detail. Often the decomposition results in another abstract machine. The decomposition continues in this fashion until a level of detail is reached where coding is possible.

There are only two phases in the WELLMADE methodology. They are applied repetitively until the design is finished.

1. Requirement Specification,
2. Program Design.

WELLMADE does not have its own specification language. However, the following features are needed in the specification in order to prove correctness:

1. Be able to represent assertions, predicates, program states, invariant relationships and requirements.

2. Include first-order predicate calculus.

3. Include notations for describing data types.

4. Be able to record performance information.

WELLMADE does not provide a method to decompose the design. There is, instead, a program design language for representing detailed procedural logic and data structures.

Chapter IV

AN EXPERIMENT IN SYSTEM DESIGN

In this chapter, four different designs of a text editor produced by four different design methodologies are presented. The purpose of this exercise is twofold. First, it is a learning experience intended to increase one's insight into design. Secondly, the experiment provides a way to compare different methodologies in a concrete manner.

The target system for the experiment is a stream editor. In this system, text is considered as a sequence (or stream) of characters. In other words, carriage return and linefeed are just characters which produce a special effect on the screen.

The rest of the chapter is organized as follows: Section one is a list of user requirements for the editor. Section two specifies the data structures. Section three explains documentation of the designs. Section four contains the actual designs. Finally, section five is a comparative evaluation of the methodologies.

4.1 REQUIREMENTS FOR A STREAM EDITOR

The following requirements were derived for an experimental system. That is, the target system does not have real-time interaction with a terminal. Instead, the terminal is simulated by a matrix of characters. Also, the target system is intended to be device independent, therefore we did not include any specific operating system considerations. Lastly, PL/I was chosen as a model language for the design to provide the necessary data structures.

1. The system should support the view that text is a stream of characters.
2. The design shall be independent of any particular system.
3. The programming language is PL/I compatible.
4. The screen is represented by an 80x30 matrix of characters.
5. The editor shall accept the full set of Ascii characters.
6. A portion of the text is displayed on the screen automatically.
7. A cursor shall be provided to indicate the current point of reference.
8. Text is automatically inserted after the cursor, there is no need to issue an insert command.
9. The editor shall have an internal buffer for storing the text.

10. Multiple editor commands can be issued in sequence.
11. Text input and command input are separated by a control character (e.g., ∂).
12. Multiple editor commands are separated by control characters. That is, every individual command is preceeded by a control character.
13. Editor commands are preceeded by a control character.
14. The following commands should be implemented:
    a) Read File - Format is "∂r <file-name>". This command will store the content of a file in the internal buffer of the editor.
    b) Write File - Format is "∂w <file-name>". This command will write the content of the editor's internal buffer onto a file.
    c) Character Forward - Format is "∂cf". This command moves cursor to the next character.
    d) Character Back - Format is "∂cb". This command moves the cursor back by one character.
    e) Character Delete - Format is "∂cd". This command removes the current character from the buffer (i.e., the character pointed to by the cursor).
    f) Word Forward: Format is "∂wf". This command moves the cursor to the first character of the next word (words are delimited by a space).
    g) Word Back: Format is "∂wb". This command moves cursor to the last character of the previous word.

h) Word Delete: Format is "∂wd". This command removes the current word from the buffer beginning at the cursor.

i) Next Line: Format is "∂lf". This command moves the cursor to the line immediately after the current line (lines are delimited by a carriage return). Cursor is left in the same relative position, unless the previous line is too short. In the latter case the cursor is put at the end of the previous line.

j) Previous Line; Format is "∂lb". This command moves the cursor to the line immmediately before the current line. Cursor positioning follows the same rules as ∂lf.

k) Delete Line: Format is "∂ld". This command removes the current line from the buffer beginning at the cursor.

l) Top of File: Format is "∂t". This command moves the cursor to the first character in the current buffer.

m) Bottom of File: Format is "∂b". This command moves the cursor to the last character of the current buffer.

n) String Search: Format is "∂s <String>". This command will place the cursor at the first character of the next occurrence of <String>.

o) String Replace: Format is "∂r <String> <New-string>". This command will do a String Search for <String>, then replace <String> by <New-string>. The cursor is left at the first character of the replaced string.

p) End: Format is "∂e". This command terminates the editor program.

4.2 DATA STRUCTURES

During this phase of the design process, the data structures needed by the system are identified. There are two types of data objects: position indicators and storage elements. The need for storage elements arise in two parts of the systems. First, there must be an internal buffer to store the text. Secondly, the screen must hold a portion of the text at all times. Therefore the matrix which simulates the screen is also a storage element.

Each of the storage elements also needs a position indicator. Therefore a cursor is used to indicate to the user his position on the screen. A cursor is also needed to indicate to the system its position within the buffer.

The specific requirements are stated in the following subsections.

4.2.1 Internal Buffer

Two issues arise in the design of a buffer. First, the system must read/write the buffer contents from/to datasets (or files). Therefore file access facilities available in PL/I must be considered. Secondly, the kinds of buffer operations used by the system has to be considered. Furthermore, the buffer should be designed in such a way so that these operations can be performed efficiently.

Two kinds of file access facilities are available in PL/I: Stream I/O and Sequential I/O. Stream I/O views text as a sequence of characters. Sequential I/O is record oriented, which in this case, means that read/write is done line by line.

The kinds of buffer operations the system will be doing are insertion and text editing. Insertion does not require any special treatment because the rate of input from the terminal would be so slow compared to the computer's processing rate. Editing of text, however, requires fast response time. Three types of structures are considered for the buffer: character linked structure, word linked structure, and line linked structure (see figure 8).

The final selection is the line linked structure for the buffer. This structure gives a close representation of text (i.e., a close resemblance to reality). Stream input from the terminal is selected to support the stream oriented approach. Sequential I/O is selected for file access functions because the buffer is record oriented.

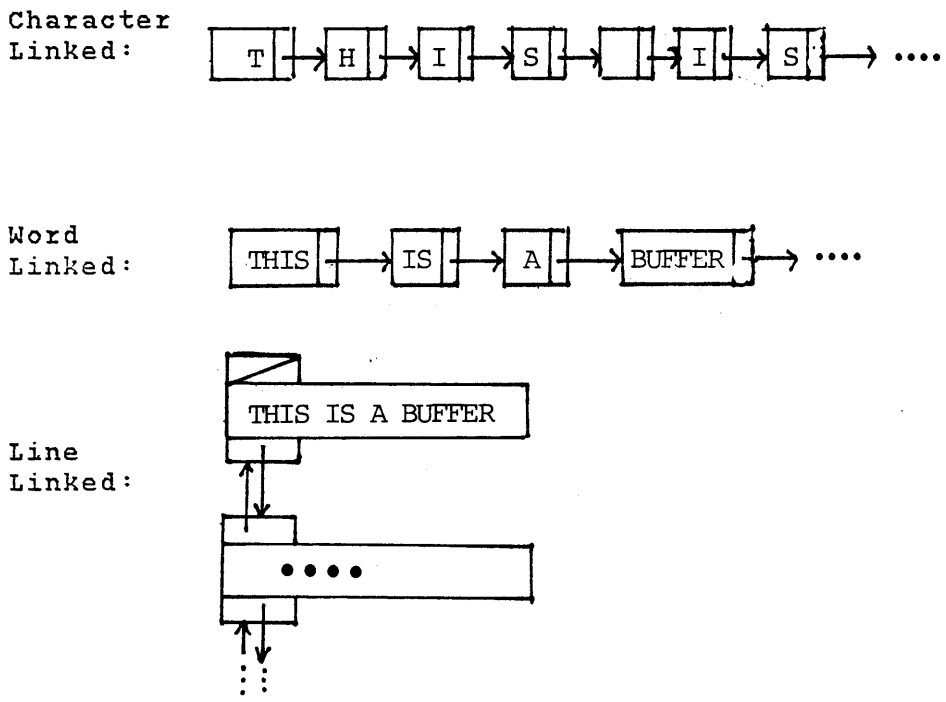


Figure 8: Three Different Buffer Structures

The basic component of the line linked buffer is the line structure. The line structure has two pointers, one points to the previous line structure and the other points to the next line structure. The storage capacity of the line structure is 80 characters. This number is chosen to match the width of the screen. If the user inputs a line of text longer than 80 characters, then the status bit will be set and the remainder of the line will be continued on the next line structure (this is referred to as overflow). In other words, the status bit indicates whether or not the stored string of characters ends with a carriage-return character.

4.2.2 Buffer Position Indicator

In order to position the cursor at any point in the buffer, two things are necessary: a pointer to the line structure, and an offset indicating the position of the character within the text string. Also, for efficiency reasons, two more pointers are added. The first points to the head of the buffer and the other points to the tail. Finally, a three element array is provided to record incremental movements of the position indicator (This will expedite the repositioning of the cursor on the screen). The "inc" array is used only by Jackson's methodology and Structural Design. The position indicator will be called "BUFFER".

```
dcl BUFFER
1 line-ptr ptr
1 offset  fixed binary
1 head ptr
1 tail ptr
1 inc  array(3)
```

4.2.3 Simulated Screen

The screen is simulated by an 80x30 matrix. In PL/I this is represented by a string array with 80 elements. Each element is a string of 30 characters.

4.2.4 Screen Position Indicator

Associated with the screen is a position indicator which points to the user's current position on the screen. It is simply 2 integers. The first integer represents the horizontal position (the x-coordinate), and the second integer represents the vertical position (the y-coordinate).

```
dcl SCREEN
1 scr string(30) array(80)
1 x-coord fixed binary
1 y-coord fixed binary
```

4.2.5 The Package Structure

In order to simplify the passing of the above data structures, the package data structure is creatd. It has 2 elements: a pointer to a BUFFER structure, and a pointer to a SCREEN structure.

```
dcl PACKAGE
1 Buff-ptr ptr
1 Scr-ptr  ptr
```

4.3 DESIGN REPRESENTATION

The system designs are described in three different ways: DARTS call graph, DARTS trees, and functional descriptions. DARTS call graphs are used to describe system architecture. DARTS trees are used to describe the logic with the modules. Functional description specifies the name of the module, the parameter and the module's functions.

Each of the above design representation techniques will be discussed in the ensuing subsections. In the remainder of this introductory section, a general description of DARTS is presented.

Design Aids for Real-Time Systems (DARTS) is a tool developed at the C. S. Draper Laboratory. It is used to aid in the definition of computer systems. It is intended to increase productivity, and improve quality and efficiency. In order to accomplish this, DARTS provides diagrams and tables to document the design, consistency checks, quality metrics, and simulations.

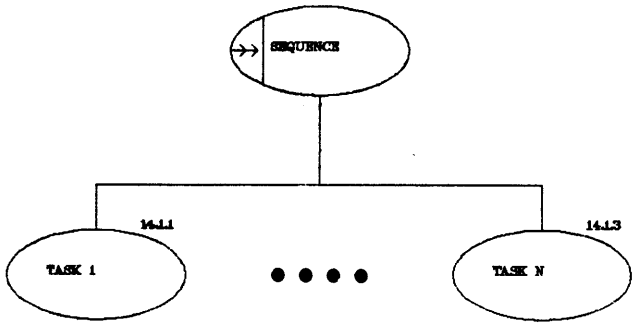
DARTS represents systems as trees. These design trees describe a set of communicating entities called processes, each of which consists of nested sequential control logic. This scheme encourages top-down development, and structured control flow. The hierarchical tree structure also allows the design to be viewed from different levels of detail.

4.3.1 DARTS Call Graph

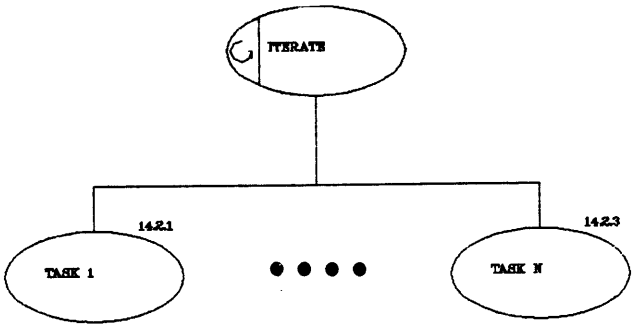
A call graph is a graph which describes the communication between modules. Thus, it also describes the architecture of the system. The graph consists of interconnected rectangular boxes. Each box represents a module, and inside the box is the name of the module. Two boxes are connected by a line if one module calls the other.

4.3.2 DARTS Trees

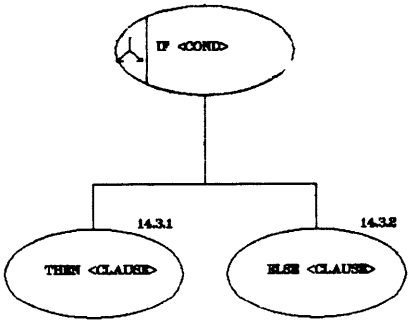
To describe the logic of a module, DARTS provides an hierarchical tree structured technique. Basically, the designer describes his design using three components: the iterator, the selector and the sequencer. The functions of each type of component is self evident. The DARTS tree is represented graphically by elliptical shapes in figure 9.



Sequencer (Do tasks 1 through n sequentially)



Iterator (Do tasks 1 through n repetitively until the stopping condition is satisfied)



Selector (The if-then-else selector)

Figure 9: DARTS Tree Components

4.3.3 Functional Description

In addition to the graphical representations, each module is described in terms of its parameters and function. A freehand format is used, the following is an example:

Module name: Print

Parameter: <filename>

Function: Place a copy of file <filename> in the printer queue.

4.4 EDITOR DESIGNS

As was mentioned earlier, four different methodologies are used for this design experiment. They are the Jackson Methodology, Structured Design, Systematic Design, Systematic Design Methodology and the Hierarchical Development Methodology.

A problem that is immediately apparent is how to prevent earlier designs from biasing results of methodologies used later. Certainly design is a refinement process, therefore, having thought through a design once would surely improve the quality and efficiency of the second design. To eliminate this bias would be impossible, but some control can be maintained by strict adherence to the procedures defined by each methodology. Furthermore, the order they are used is HDM, Jackson, SD and SDM, where HDM is the methodology which is the least mechanical.

The following subsections present briefly the design process for each methodology. Then a discussion of the evaluation technique and the results of the evaluations are presented.

4.4.1 Hierarchical Development Methodology

HDM consists of 7 steps. First the requirements are specified. Then the top and bottom abstract machines are specified. This is followed by the specification of the intermediate machines. Then the data structures for each machine is determined and the operations of each machine is implemented as a program on the next lower machine. Lastly, the abstract machines are translated into a programming language.

First, we identify the abstract machines. HDM does not provide any guidelines for doing this, therefore a trial and error method is used to derive the hierarchy of abstract machines shown in figure 10.

The data representation for each machine is self evident. Each of the abstract machines is further modularized. The EDITOR machine consists of 17 modules. A DISPATCH, and one module for each editor command (see figure 11).

The BUFFER machine consists of modules which operate on the buffer. They are divided into five types: Constructors, Selectors, Mutators, Testers, and Cursor Movers. (see figure 12).

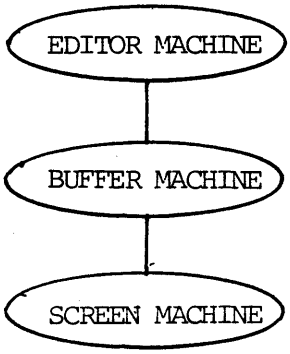


Figure 10: HDM's Abstract Machines for the Editor

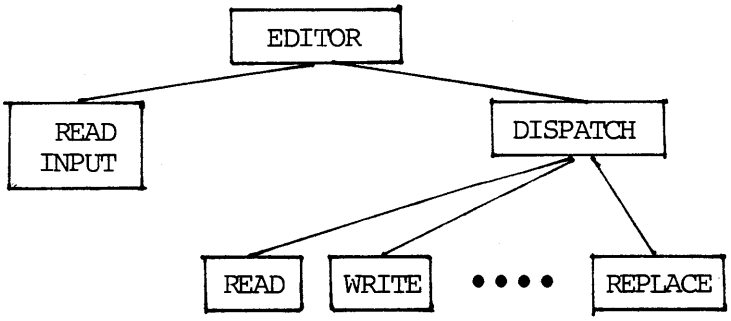


Figure 11: EDITOR Machine

The SCREEN machine provides operators to manipulate the screen (see figure 13). The entire architecture is shown in figure 14.

The DARTS tree representation for a selected set of modules is given. Many of the BUFFER machine modules are straightforward, therefore only functional descriptions are given for them.

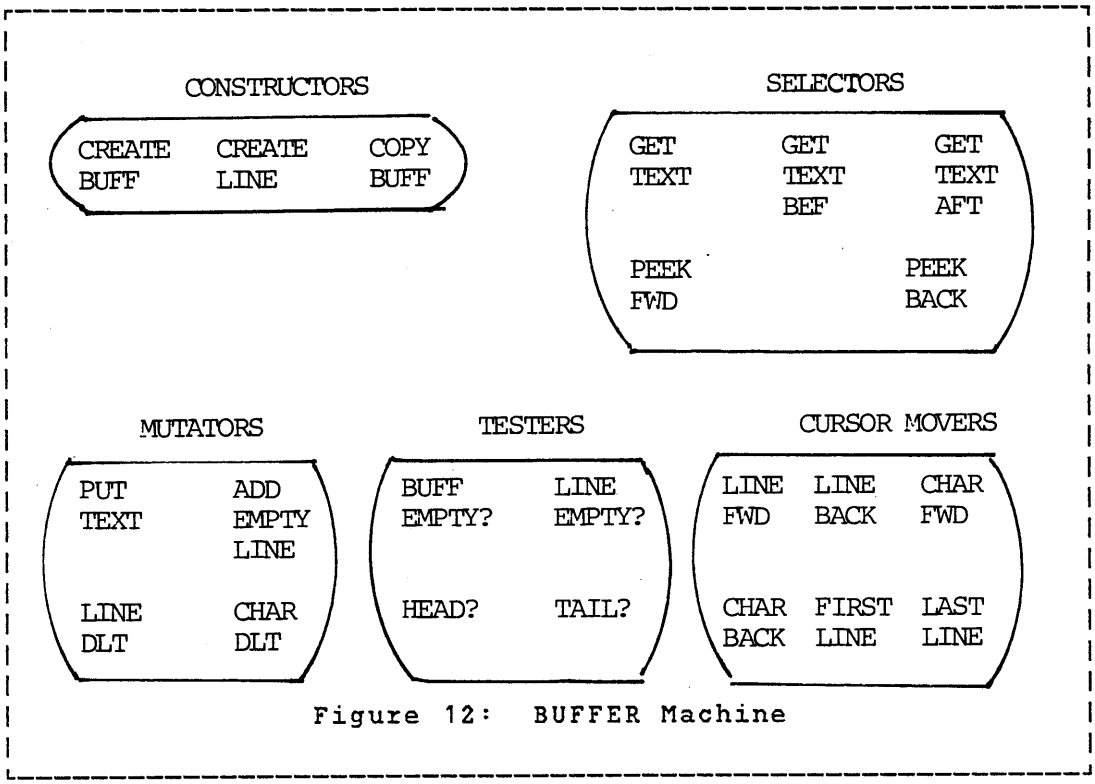


Figure 12: BUFFER Machine

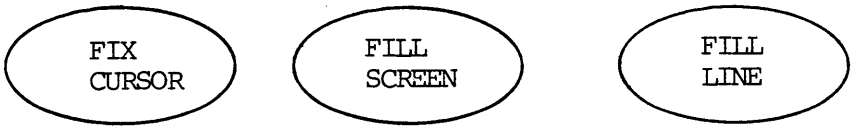
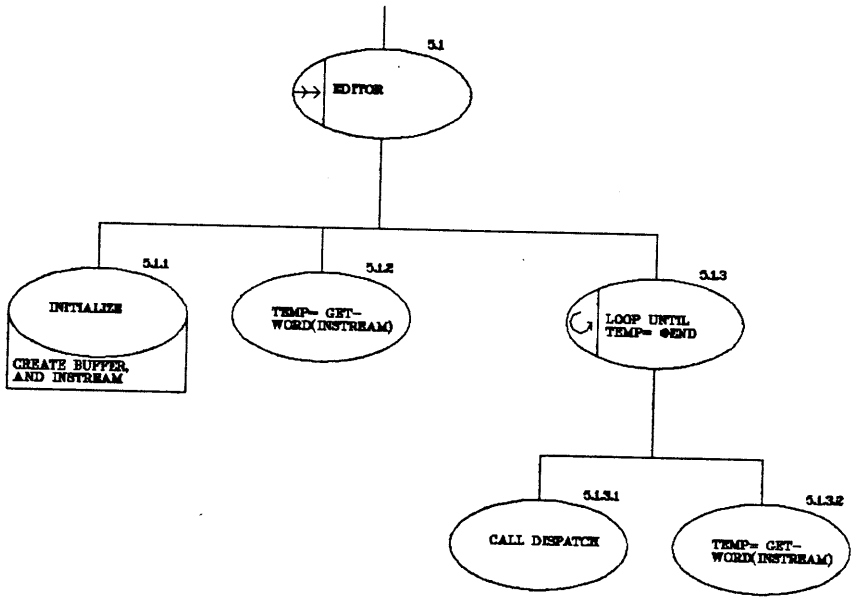


Figure 13: SCREEN Machine

Module Name: EDITOR

Parameter: none

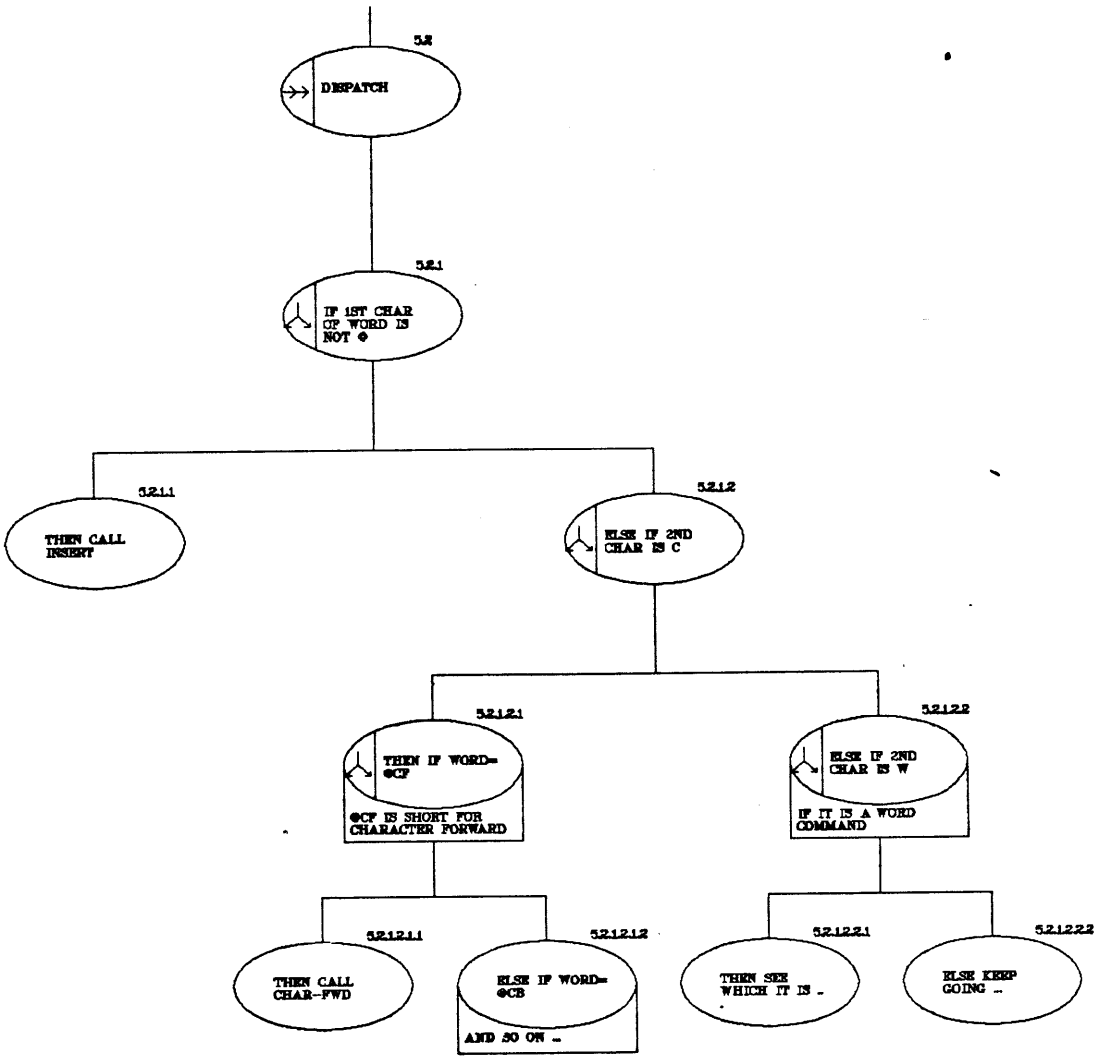
Function: Get a word from the input stream, and dispatch.



Module Name: DISPATCH

Parameter: PACKAGE, WORD

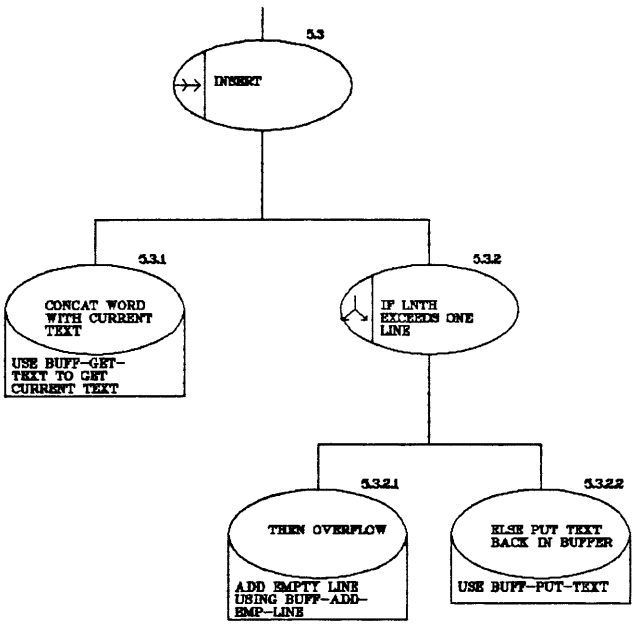
Function: Dispatch WORD to the appropriate subroutine.



Module Name: INSERT

Parameter: PACKAGE, WORD

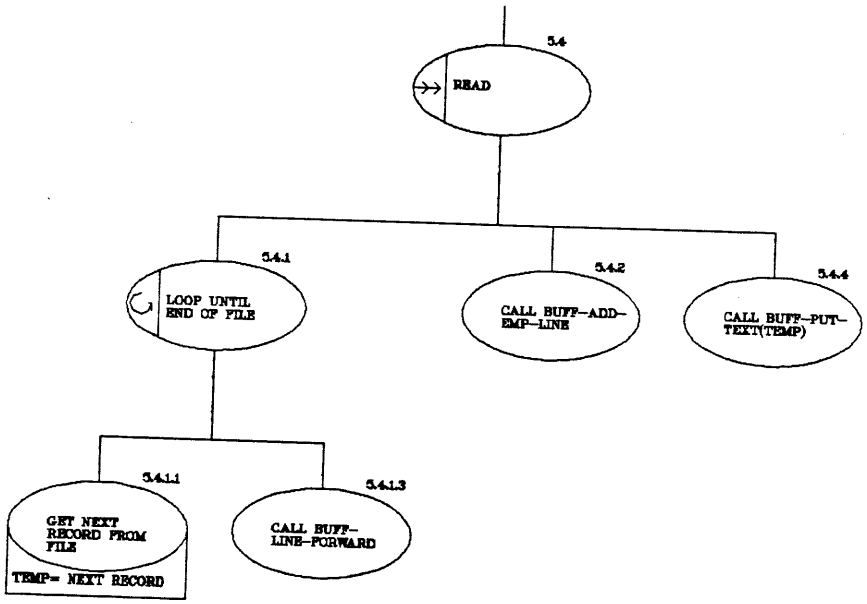
Function: Insert WORD before the cursor.



Module Name: READ

Parameter: PACKAGE, FILENAME

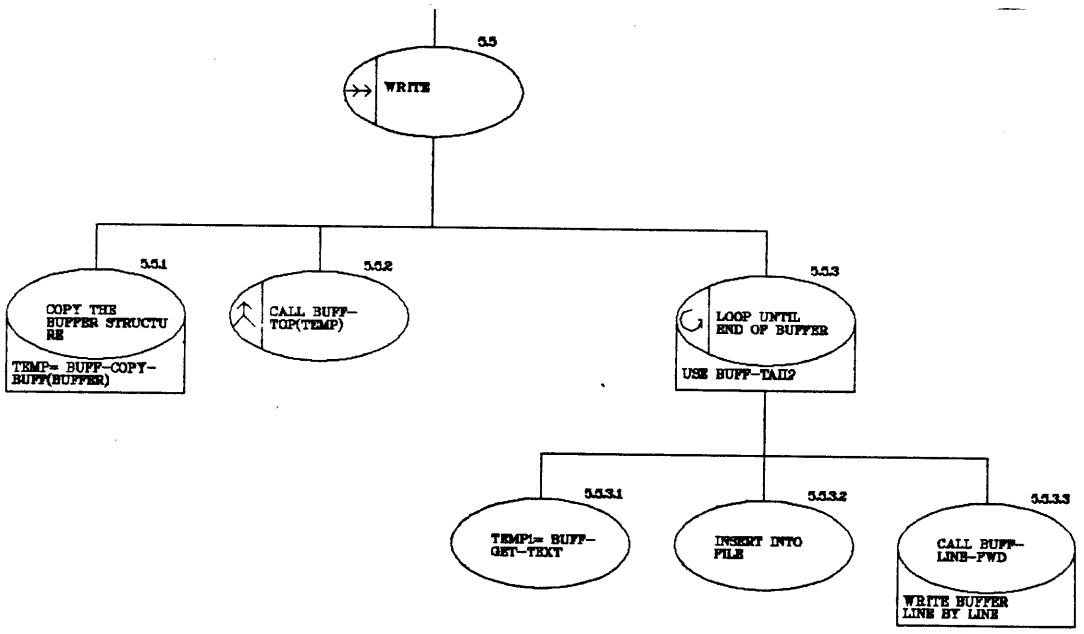
Function: Insert contents of file FILENAME before the cursor.



Module Name: WRITE

Parameter: PACKAGE, FILENAME

Function: Copy the entire buffer into a file under FILENAME.



Module Name: CHAR-FWD

Parameter: PACKAGE

Function: Move cursor forward one character (Just call BUFF-CHAR-FWD).

Module Name: CHAR-BACK

Parameter: PACKAGE

Function: Move cursor back one character (Just call BUFF-CHAR-BACK).

Module Name: CHAR-DLT

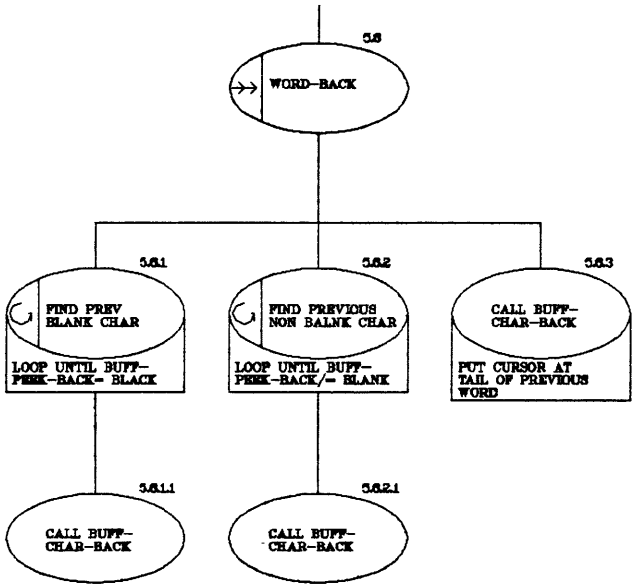
Parameter: PACKAGE

Function: Delete character pointed to by the cursor (Just call BUFF-CHAR-DLT).

Module Name: WORD-BACK

Parameter: PACKAGE

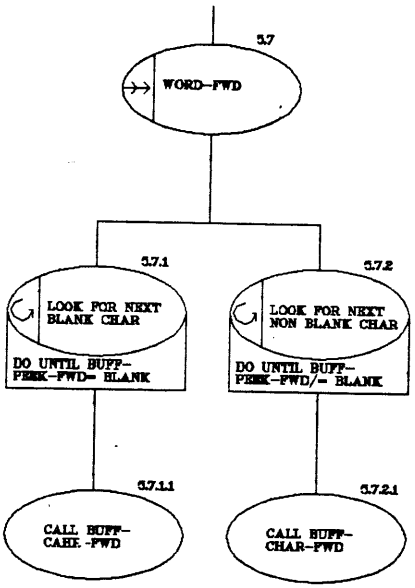
Function: Move cursor to the tail of the previous word.



Module Name: WORD-FWD

Parameter: PACKAGE

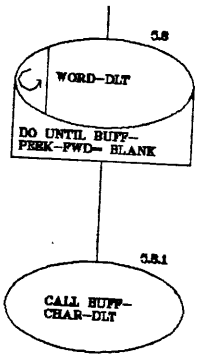
Function: Move PACKAGE's cursor to the head of the next word. Uses buffer machine operators.



Module Name: WORD-DLT

Parameter: PACKAGE

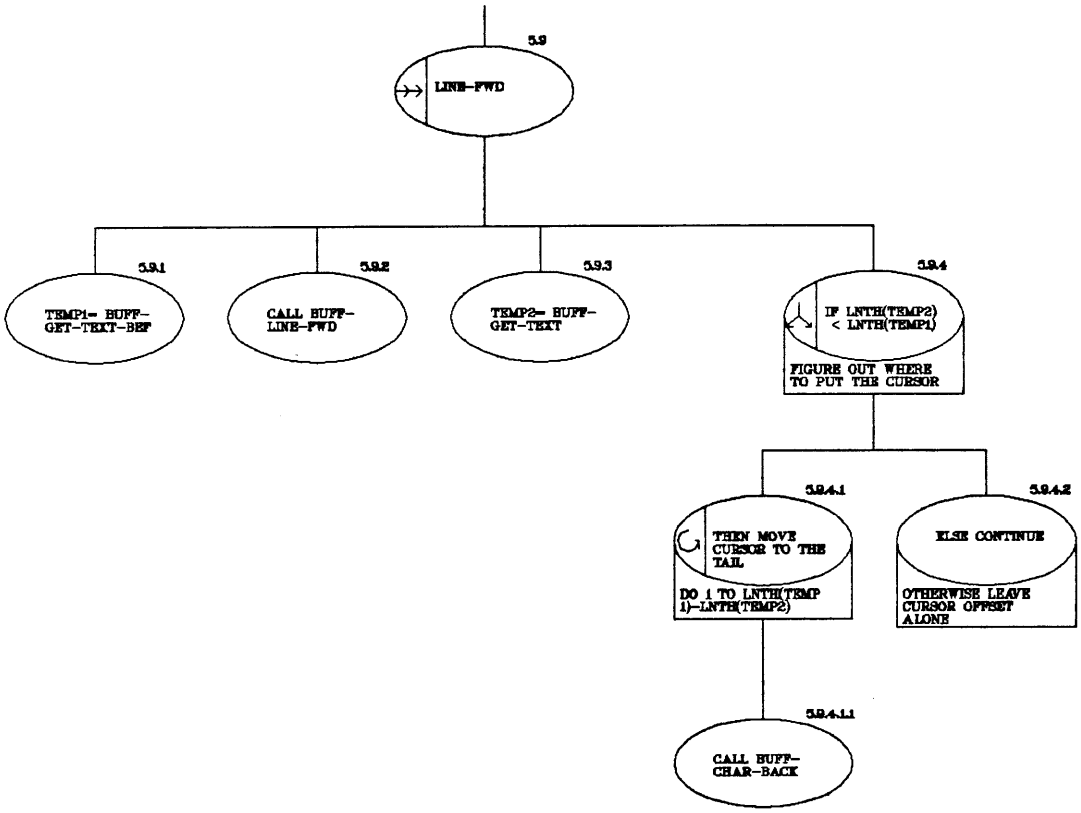
Function: Delete everything between the cursor and the next space.



Module Name: LINE-FWD

Parameter: PACKAGE

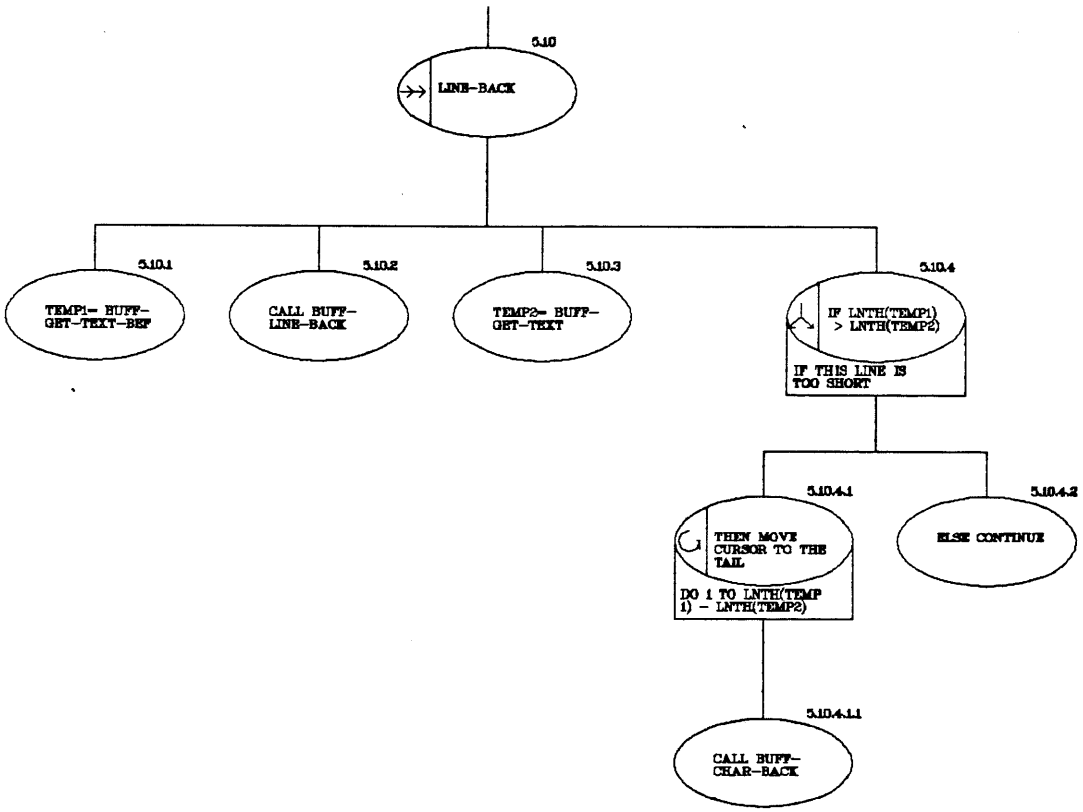
Function: Move cursor to the next line. If next line is too short, the cursor is placed at the tail of the line.



Module Name: LINE-BACK

Parameter: PACKAGE

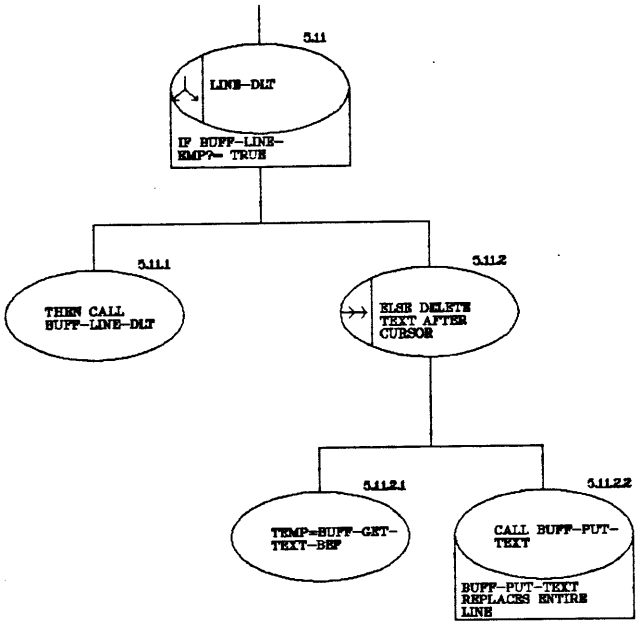
Function: Move cursor to the previous line. If previous line is too short, cursor is placed at the tail of the line.



Module Name: LINE-DLT

Parameter: PACKAGE

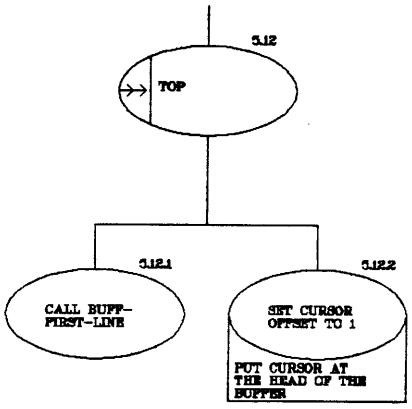
Function: Delete everything from cursor to the end of the line. If line is empty, remove it from the buffer.



Module Name: TOP

Parameter: PACKAGE

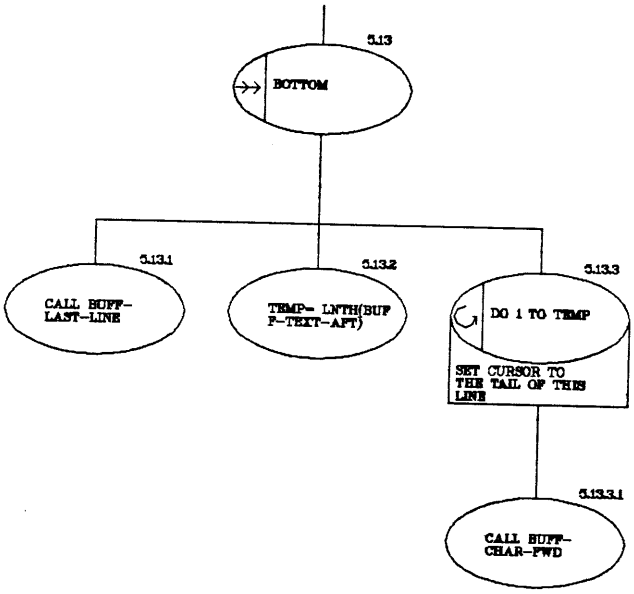
Function: Place cursor at the first character of the buffer.



Module Name: BOTTOM

Parameter: PACKAGE

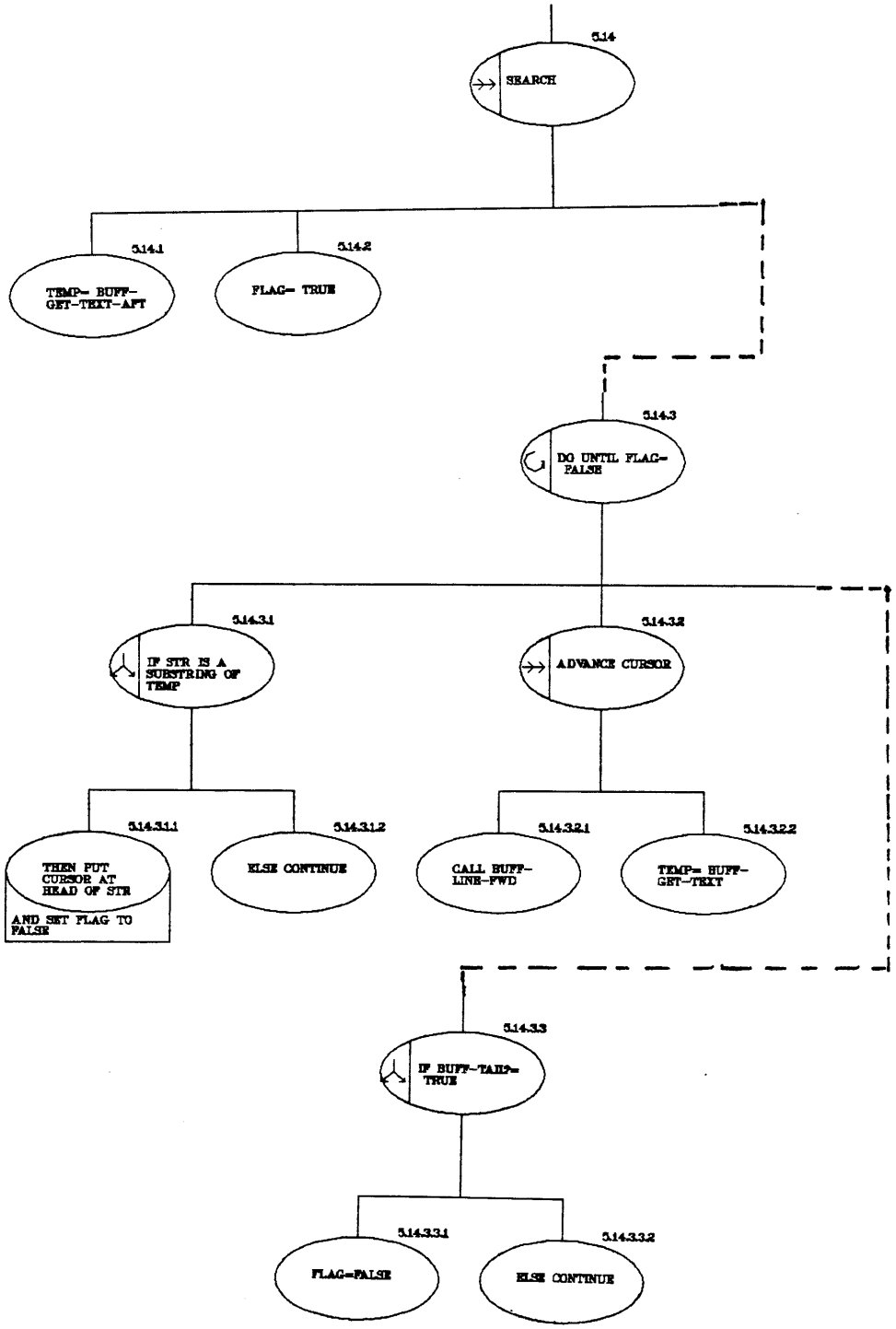
Function: Place cursor at the last character of the buffer.



Module Name: SEARCH

Parameter: PACKAGE, STR

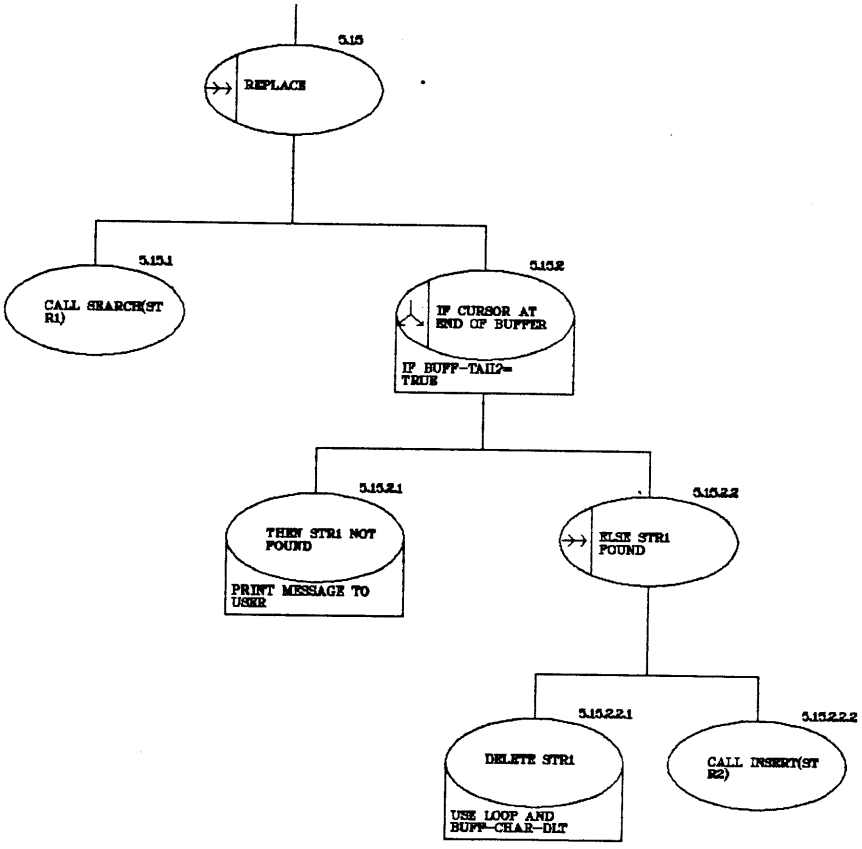
Function: Find first occurrence of STR after cursor. Place cursor at the head of STR.



Module Name: REPLACE

Parameter: PACKAGE, STR1, STR2

Function: Replace the first occurrence of STR1 after cursor by STR2.



Module Name: CREATE-BUFF

Parameter: none

Function: Return a pointer to a buffer structure.

Module Name: COPY-BUFF

Parameter: PACKAGE

Function: Return a pointer to a copy of the buffer structure.

Module Name: CREATE-LINE

Parameter: none

Function: Return a pointer to a line structure.

Module Name: BUFF-GET-TEXT

Parameter: PACKAGE

Function: Return the contents of the current line.

Module Name: BUFF-GET-TEXT-BEF

Parameter: PACKAGE

Function: Return text string before the cursor.

Module Name: BUFF-GET-TEXT-AFT

Parameter: PACKAGE

Function: Return text string after the cursor.

Module Name: BUFF-PEEK-FWD

Parameter: PACKAGE

Function: Return the next character without moving the cursor.

Module Name: BUFF-PEEK-BACK

Parameter: PACKAGE

Function: Return the previous character without moving the cursor.

Module Name: BUFF-PUT-TEXT

Parameter: PACKAGE, STR

Function: Insert STR after the cursor.

Module Name: ADD-EMP-LINE

Paramerter: PACKAGE

Function: Add an empty line after the current line.

Module Name: BUFF-LINE-DLT

Parameter: PACKAGE

Function: Delete the current line of text beginning at the cursor, then merge with the next line.

Module Name: BUFF-CHAR-DLT

Parameter: PACKAGE

Function: Remove character at the cursor. Use substring function and concatenation. Call SCREEN-FILL-LINE.

Module Name: BUFF-LINE-EMPTY?

Parameter: PACKAGE

Function: Return true if contents of current line is the null string.

Module Name: BUFF-BUFF-EMPTY?

Parameter: PACKAGE

Function: Return true if there is nothing in the buffer.

Module Name: BUFF-HEAD?

Parameter: PACKAGE

Function: Return true if cursor points to the first character in the bufer.

Module Name: BUFF-TAIL?

Parameter: PACKAGE

Function: Return true if cursor points to the last character of the buffer.

Module Name: BUFF-LINE-FWD

Parameter: PACKAGE

Function: Move cursor to the next line. Cursor offset remains the same unless the next line is too short. If the latter is true then put cursor at the end of the next line.

Module Name: BUFF-LINE-BACK

Parameter: PACKAGE

Function: Move cursor to the previous line. Cursor position follows the same rules as BUFF-LINE-FWD.

Module Name: BUFF-CHAR-FWD

Parameter: PACKAGE

Function: Increment the offset of current line by 1. If result crosses line boundary, then set offset to length of previous line, and current line pointer to previous line. Finally, call SCREEN-MOVE-CURSOR.

Module Name: BUFF-CHAR-BACK

Parameter: PACKAGE

Function: Decrement the offset. If result crosses line boundary, then put the cursor at the end of the previous line. Call SCREEN-MOVE-CURSOR.

Module Name: BUFF-FIRST-LINE

Parameter: PACKAGE

Function: Set current lihe pointer to the pointer to the first line. Call SCREEN-FILL-SCREEN.

Module Name: BUFF-LAST-LINE

Parameter: PACKAGE

Function: Set current line pointer to the pointer to last line. Call SCREEN-FILL-SCREEN.

Module Name: SCREEN-CREATE-SCREEN

Parameter: none

Module Name: SCREEN-MOVE-CURSOR

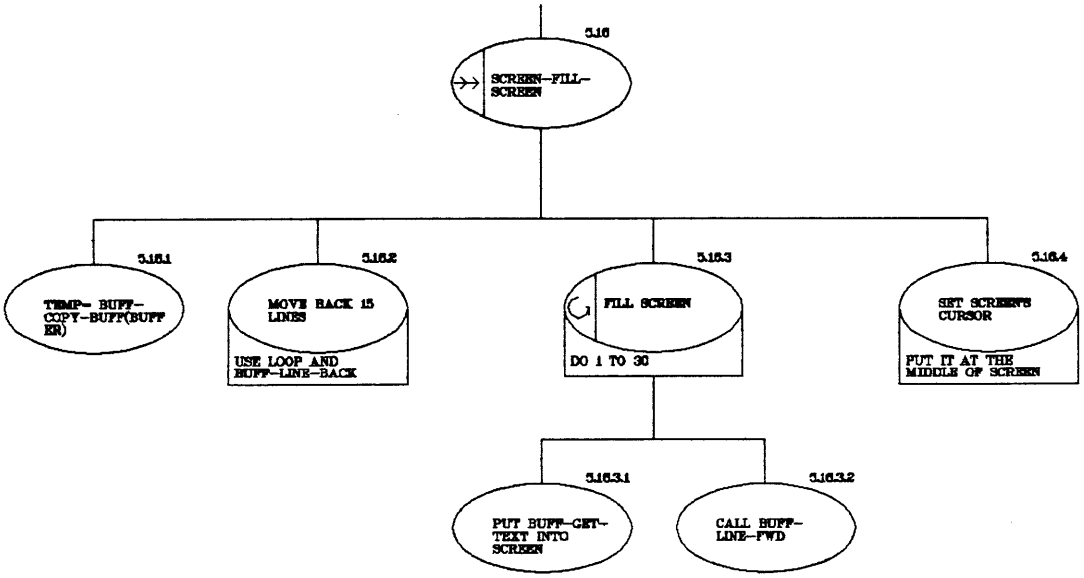
Parameter: PACKAGE, X, Y

Function: Add X to horizontal index, and add Y to vertical index.

Module Name: SCREEN-FILL-SCREEN

Parameter: PACKAGE

Function: Refill the screen. Place current line at the middle.



4.4.2 Jackson Methodology

As was stated in chapter 3, Jackson's methodology consists of 3 steps:

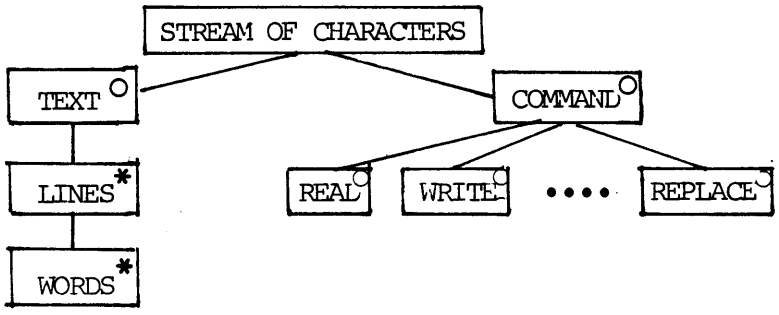
1. Define the input and output data structures.
2. Transform the data structures into program structures.
3. List the program tasks as executable operations, and allocate each task to a program component.

The input and output data structures for the editor are as shown in figure 15.

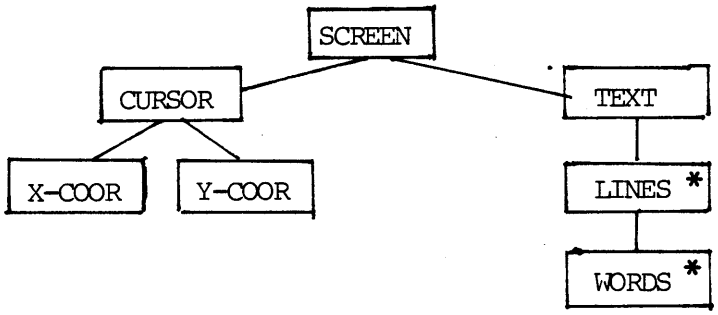
There is a structure clash between the input and the output data structures. However, there is a substructure which both the input and output share - the text substructure. This leads us to the buffer structure directly. We can use the program inversion technique to contruct the architecture: Input Buffer Output

Program Inversion separates the problem into two parts. The first part deals only with the transformation from input into the buffer. The second part deals with transforming the buffer to output.

The input conversion can be handled by the system structure shown in figure 16. Notice that it is exactly analogous to the "Command" substructure of the input data structure.



Jackson System Input Data Structure



Jackson System Output Data Structure

Figure 15: Jackson's Input and Output Data Structures

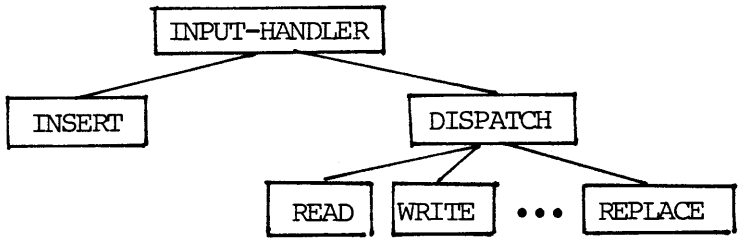


Figure 16: Jackson's Input Handler

The output handler is contructed from the output data structure as shown in figure 17.

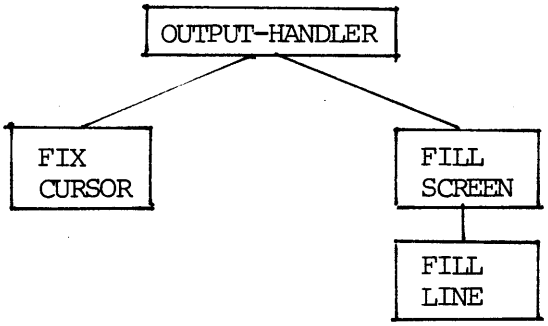


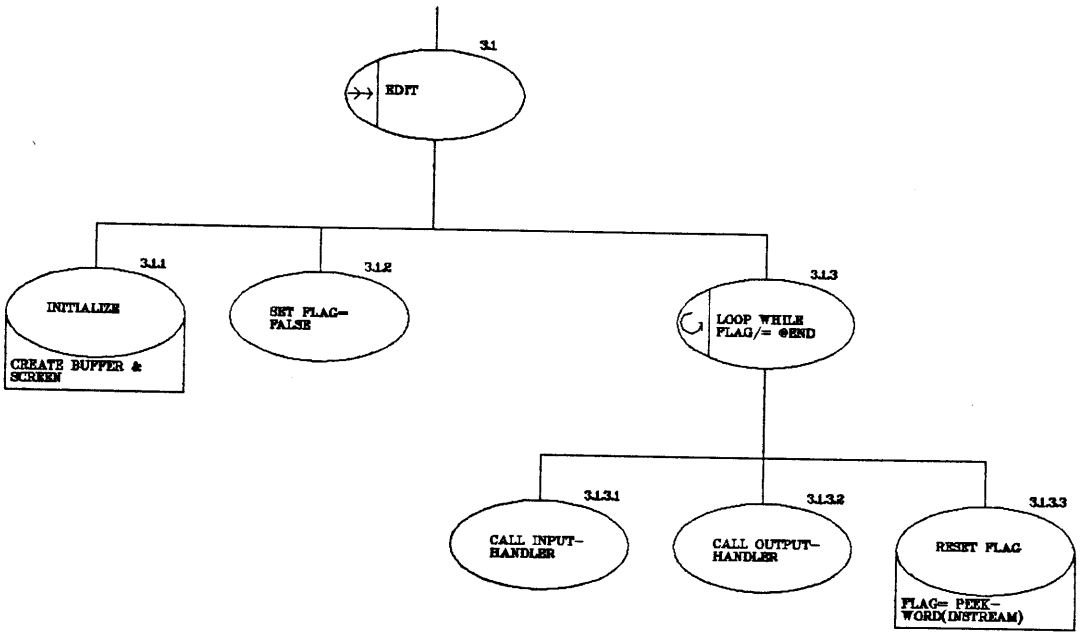
Figure 17: Jackson's Output Handler

The entire system architecture is shown in figure 18. Detailed design is done with DARTS. The logic of each module is displayed using DARTS Trees. They are listed in the following pages.

Module Name: EDITOR

Parameter: none

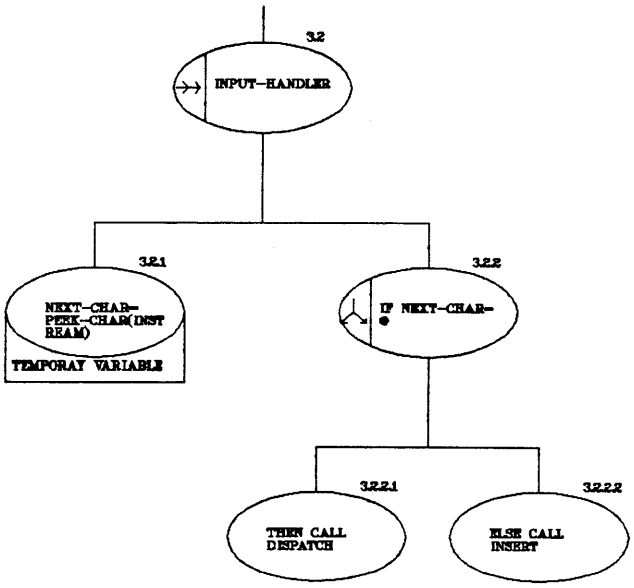
Function: Coordinates activities between input and output.



Module Name: INPUT-HANDLER

Parameter: PACKAGE, INSTREAM

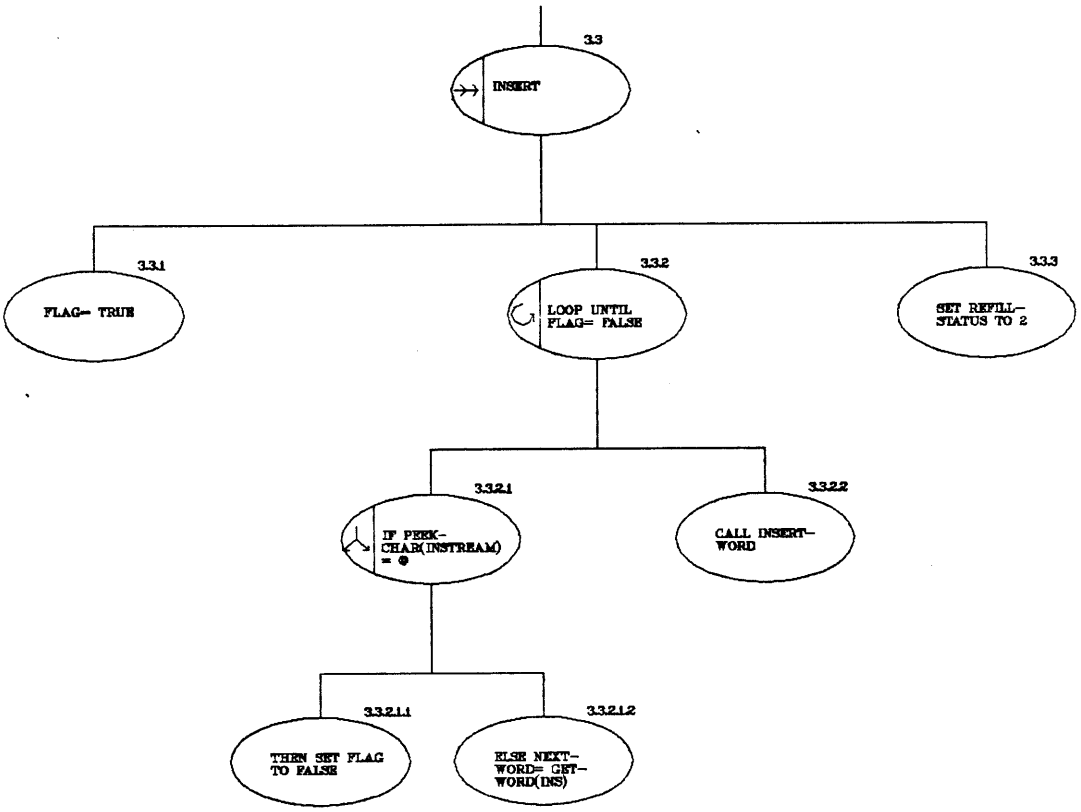
Function: Dispatch on the type of the input string. If input word is a command then call DISPATCH, otherwise call INSERT.



Module Name: INSERT

Parameter: PACKAGE, INSTREAM

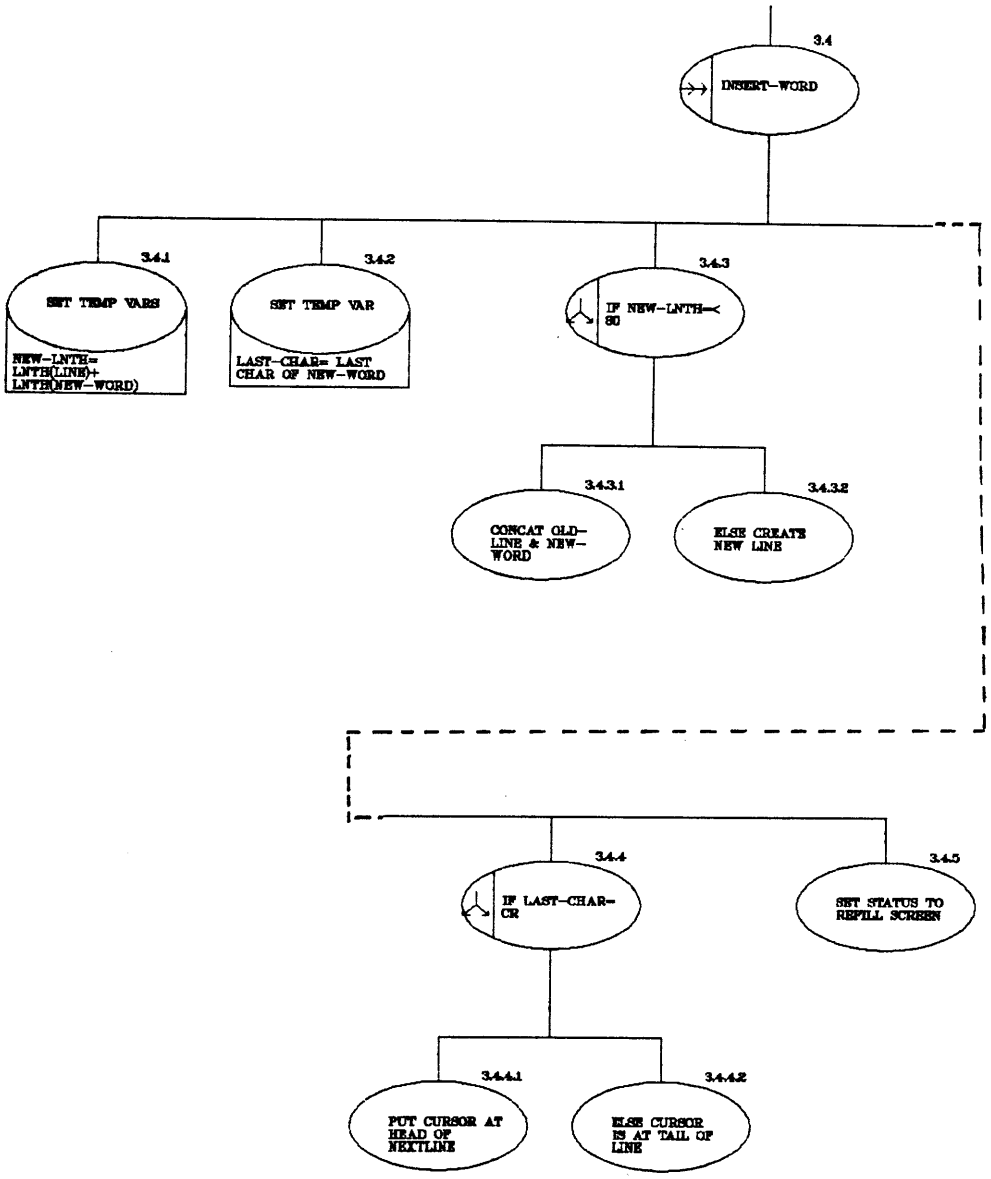
Function: Driver-loop for insertion of words one at a time.



Module Name: INSERT-WORD

Parameter: PACKAGE, WORD

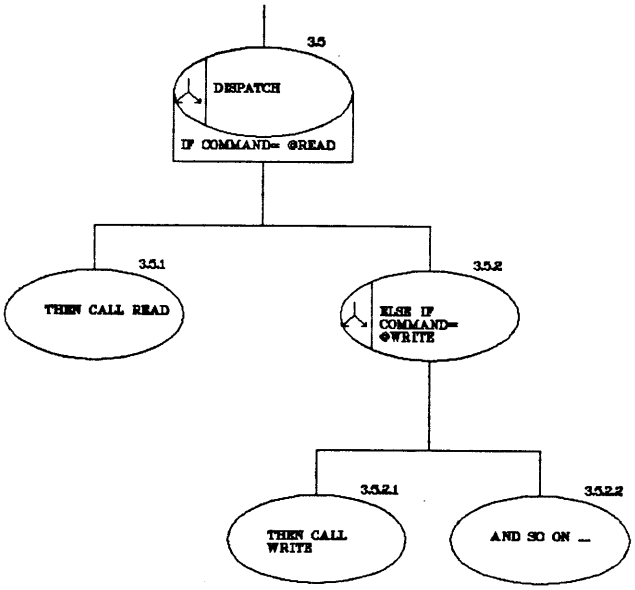
Function: Add word to the buffer. Put PACKAGE's cursor at the end of WORD.



Module Name: DISPATCH

Parameter: COMMAND, PACKAGE, INSTREAM

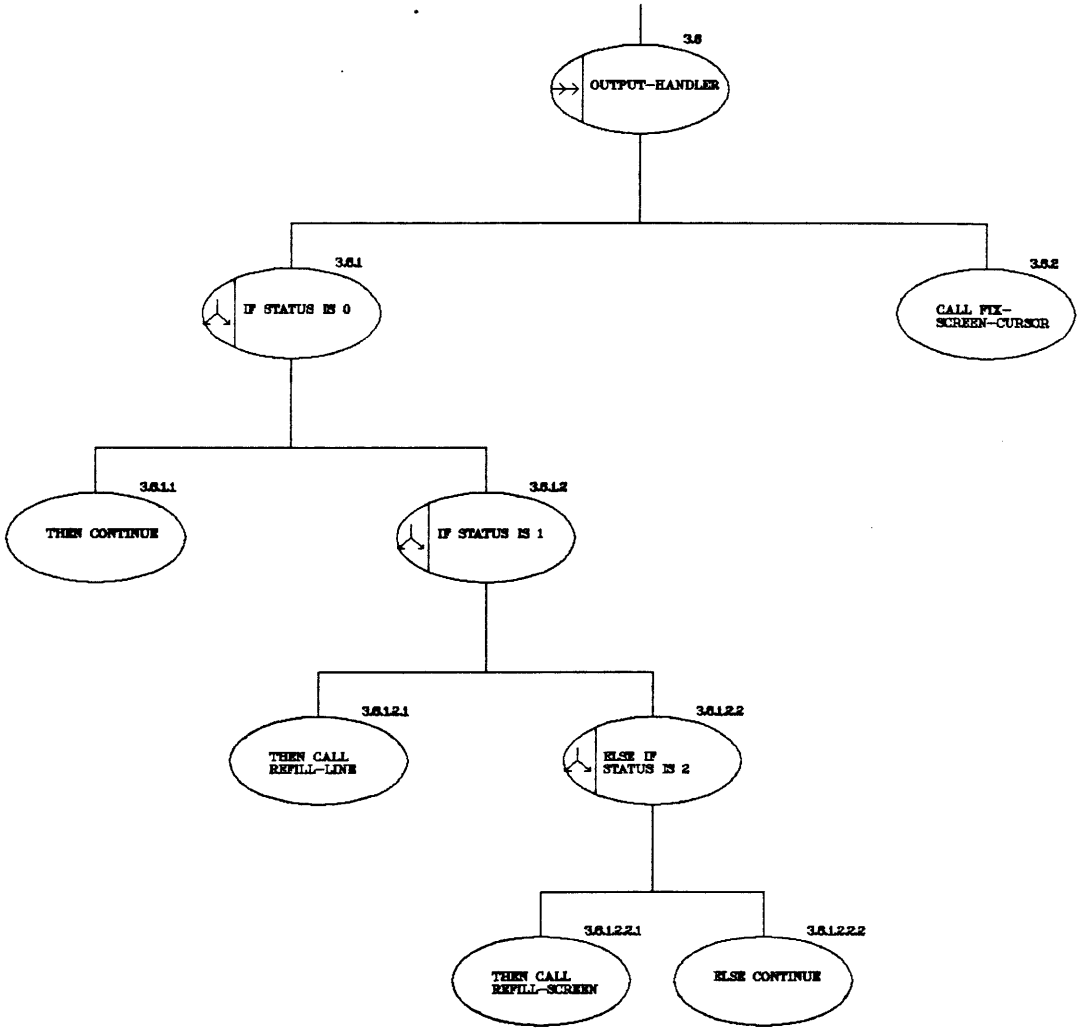
Function: Dispatch the command to the appropriate command processor.



Module Name: OUTPUT-HANDLER

Parameter: PACKAGE

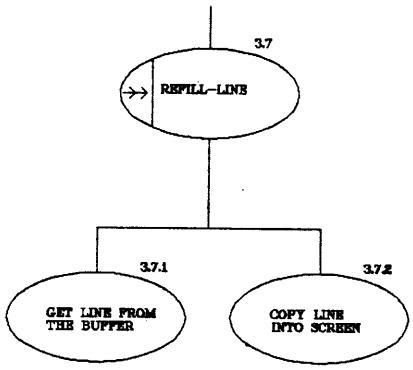
Function: Determine whether to refill one line of the screen or refill the entire screen. Also coordinates the adjustment of the screen cursor.



Module Name: REFILL-LINE

Parameter: PACKAGE

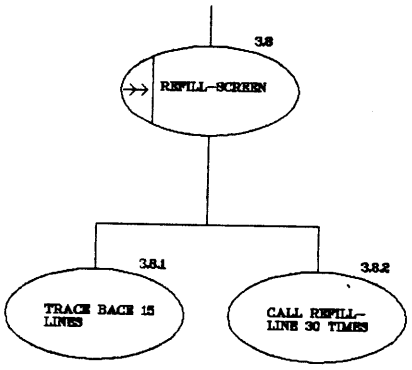
Function: Copy the current line of text from the buffer into the screen.



Module Name: REFILL-SCREEN

Parameter: PACKAGE

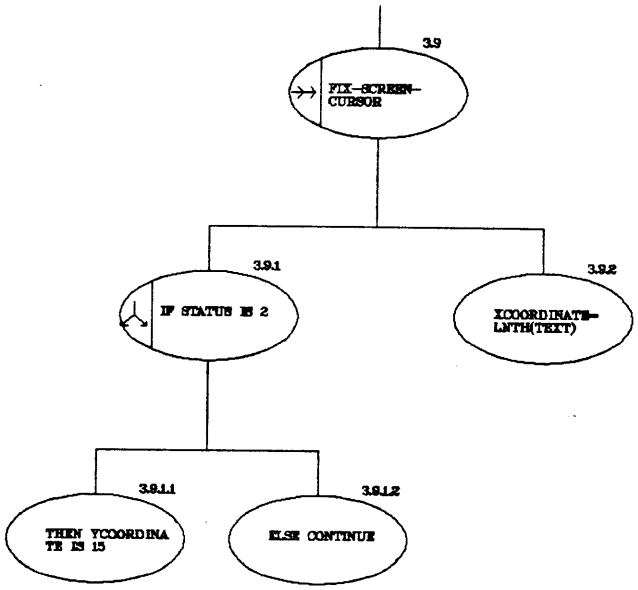
Function: Refill the screen array, placing the current line at the middle of the screen.



Module Name: FIX-SCREEN-CURSOR

Parameter: PACKAGE

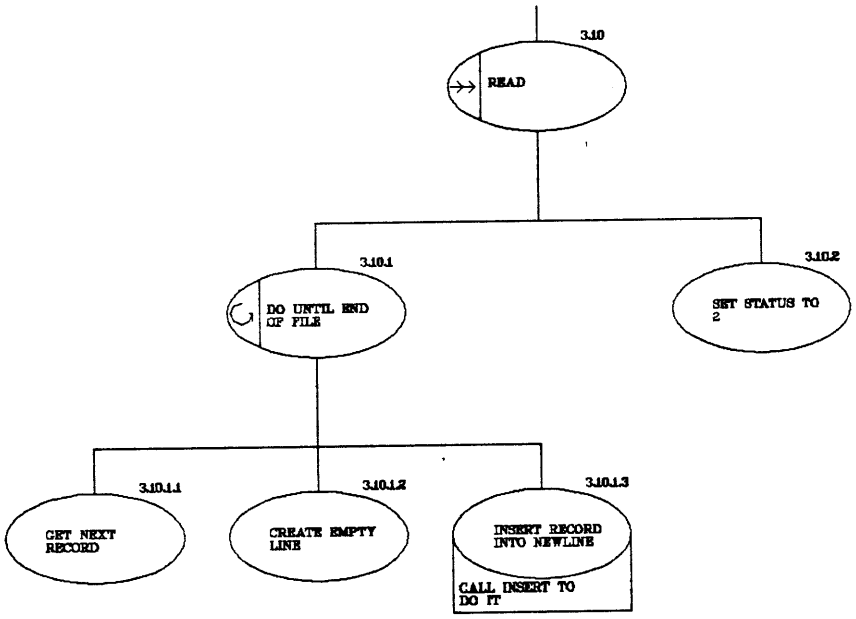
Function: If refill status is 0 or 1 the adjust the xcoordinate. if it is 2 then set ycoordinate to 15 (middle of the screen), and adjust xcoordinate as before.



Module Name: READ

Parameter: PACKAGE, FILENAME

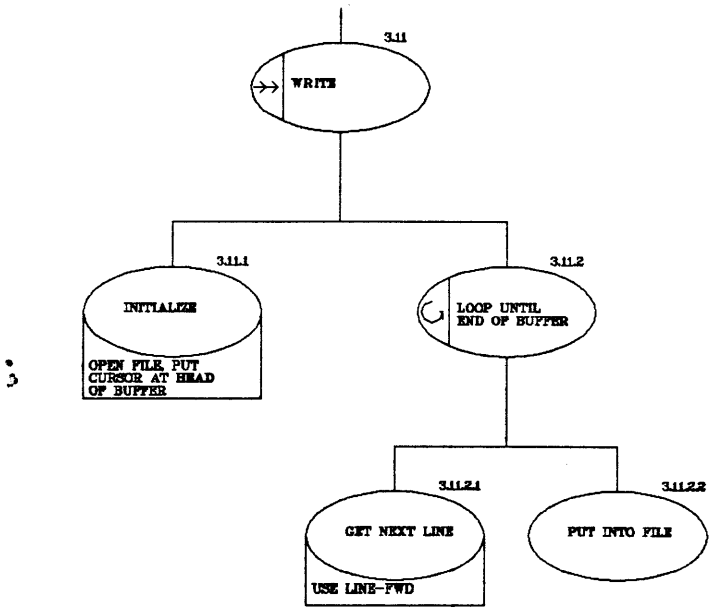
Function: Copy the contents of the file into the buffer.



Module Name: WRITE

Parameter: PACKAGE, FILENAME

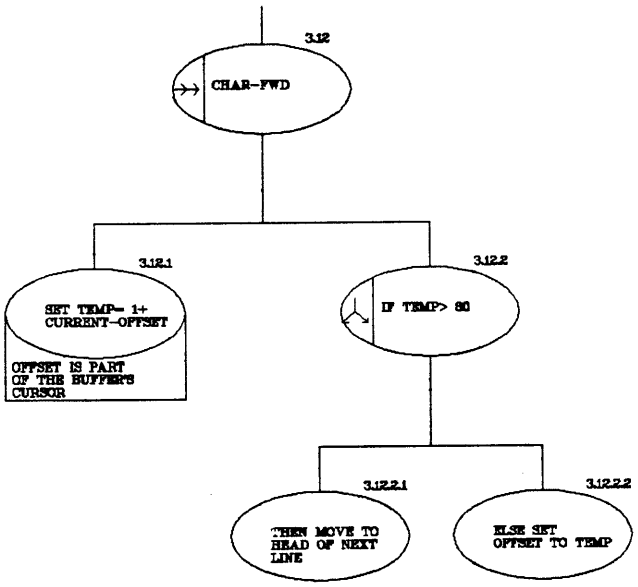
Function: Copy the entire buffer into a file under FILENAME.



Module Name: CHAR-FWD

Parameter: PACKAGE

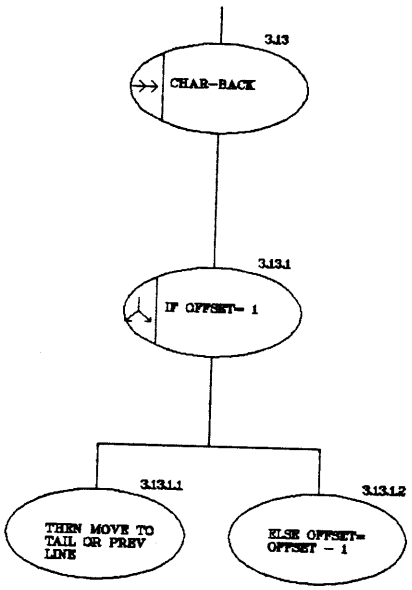
Function: Move cursor position forward one character.



Module Name: CHAR-BACK

Parameter: PACKAGE

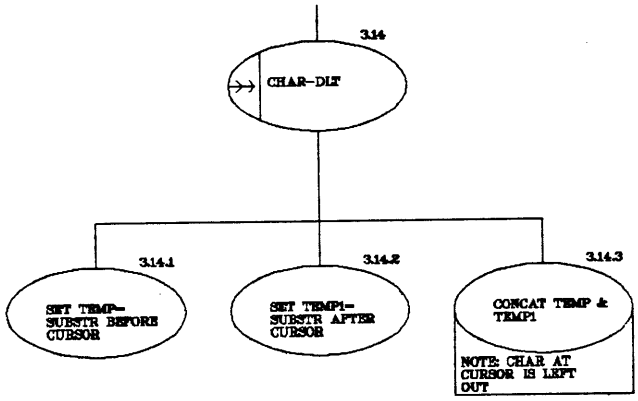
Function: Move the cursor position back one character.



Module Name: CHAR-DLT

Parameter: PACKAGE

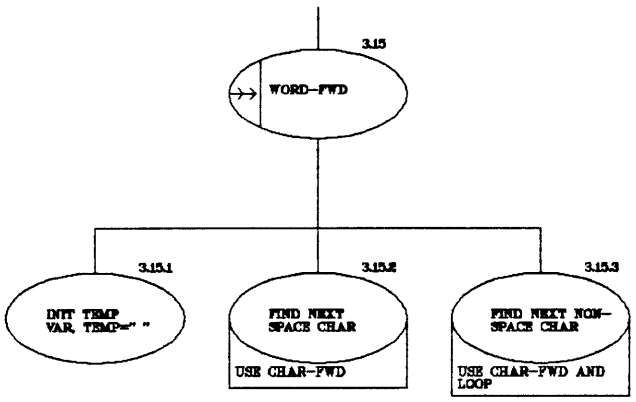
Function: Remove the character pointed to by the cursor.



Module Name: WORD-FWD

Parameter: PACKAGE

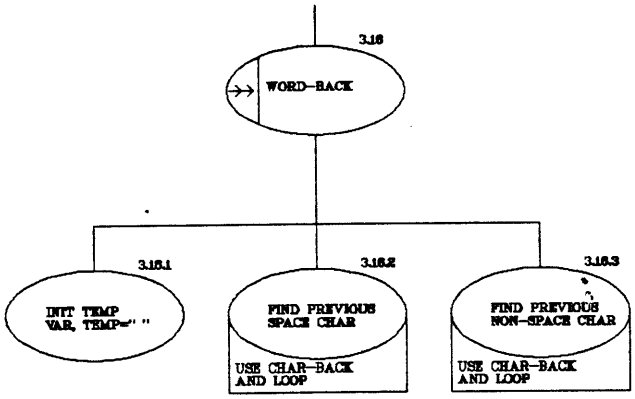
Function: Move cursor position to the head of the next word.



Module Name: WORD-BACK

Parameter: PACKAGE

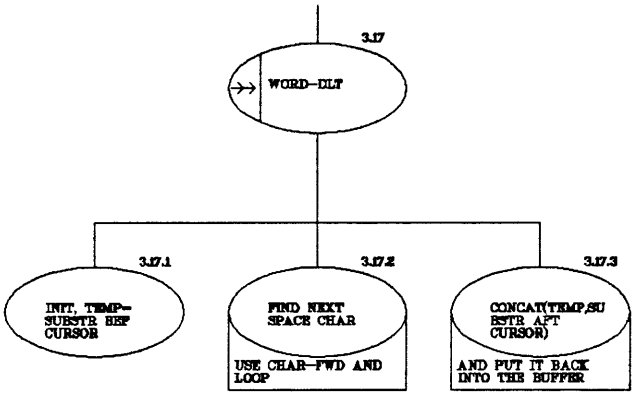
Function: Move the cursor position to the tail of the previous word.



Module Name: WORD-DLT

Parameter: PACKAGE

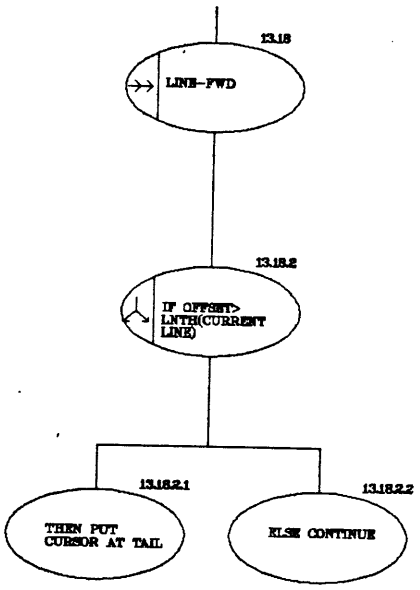
Function: Remove the string of non-space characters beginning at the current position until the next space character.



Module Name: LINE-FWD

Parameter: PACKAGE

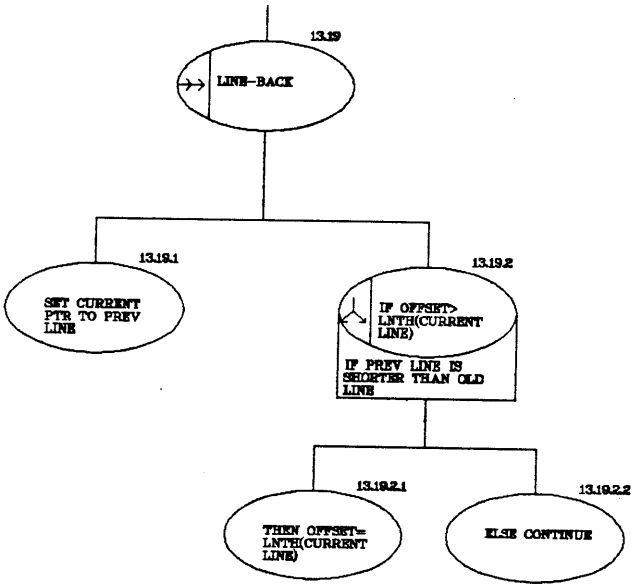
Function: Move cursor to the next line. The offset does not change unless the length of next line is too short. In the latter case, the cursor is placed at the tail of the next line.



Module Name: LINE-BACK

Parameter: PACKAGE

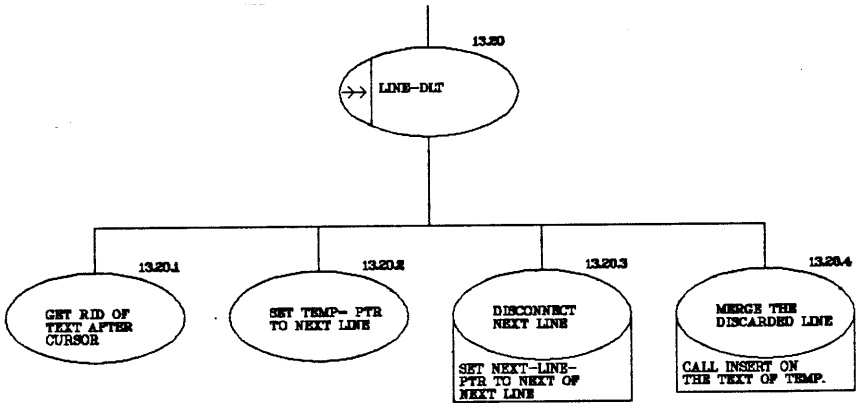
Function: Move cursor to the previous line. Offset unchanged unless text on previous line is too short.



Module Name: LINE-DLT

Parameter: PACKAGE

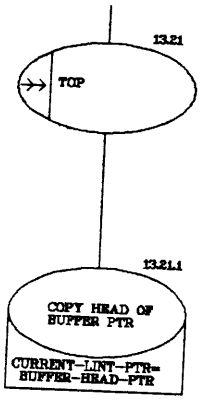
Function: Deletes text from the cursor to the end of the line. Cursor is left where it was. Adjoin next line to the remainder of current line.



Module Name: TOP

Parameter: PACKAGE

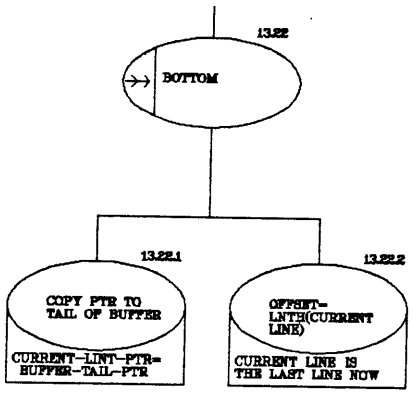
Function: Place cursor at the head of the buffer.



Module Name: BOTTOM

Parameter: PACKAGE

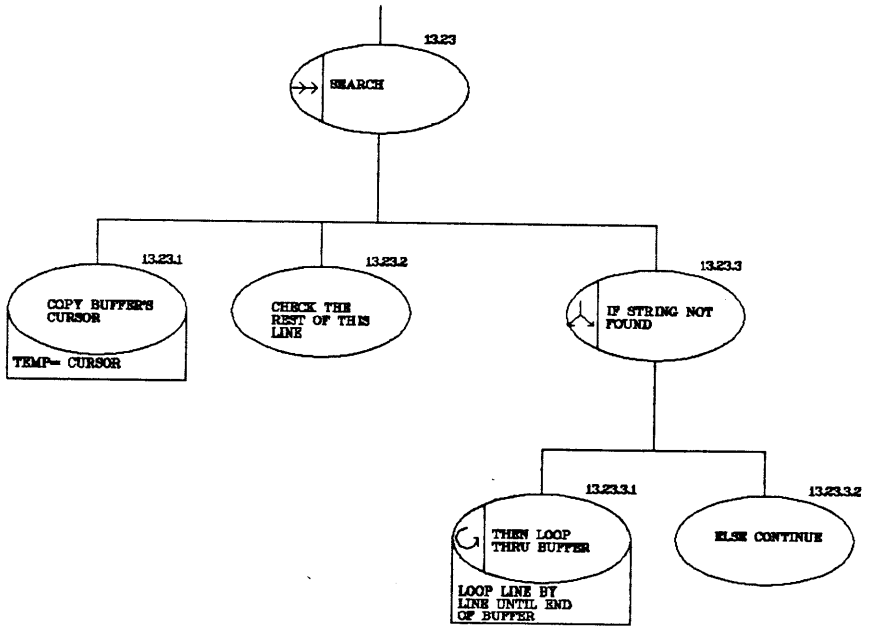
Function: Place cursor at the end of the buffer.



Module Name: SEARCH

Parameter: PACKAGE, STRING

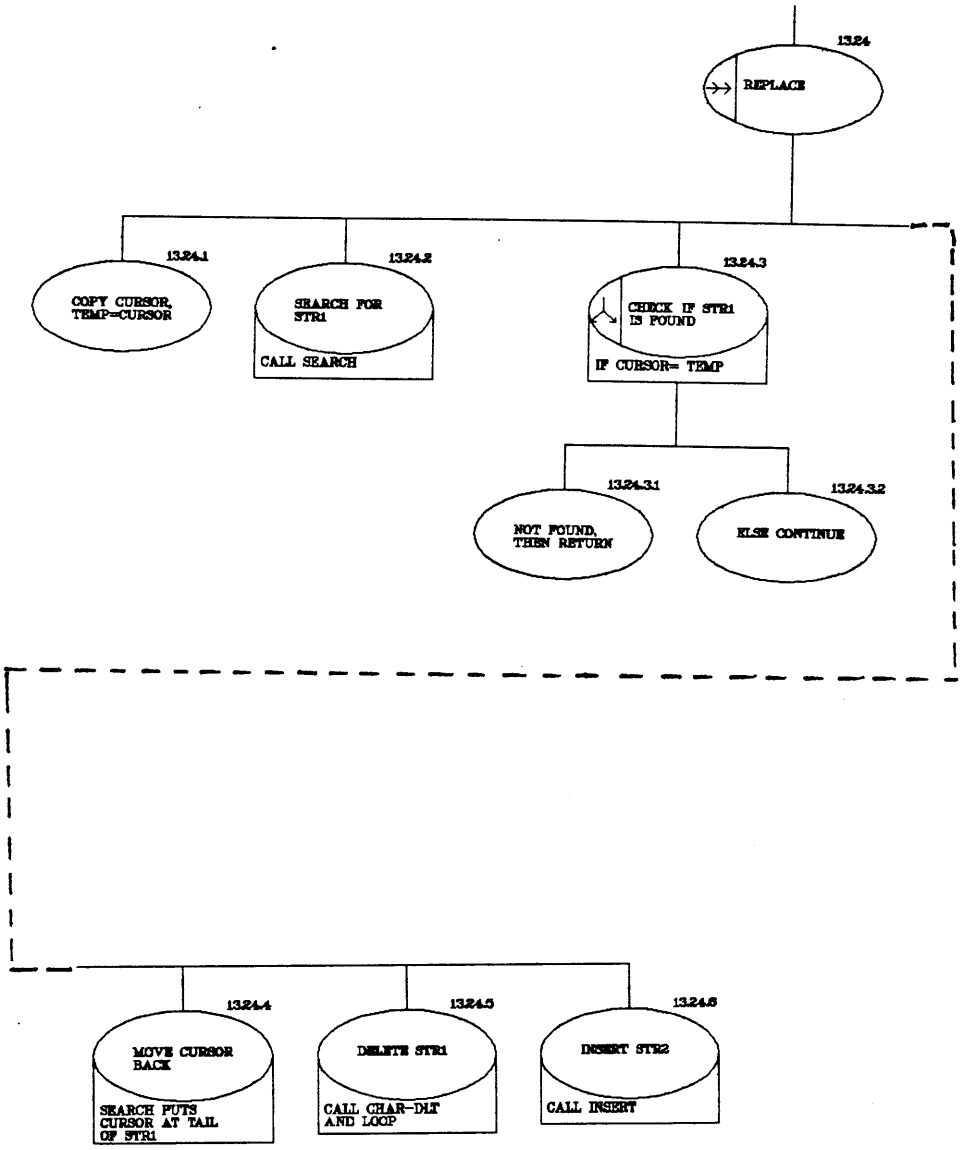
Function: Place cursor at the end of the first occurrence of string oafter the cursor. If not found then leave cursor alone.



Module Name: REPLACE

Parameter: PACKAGE, STR1, STR2

Function: Find STR1 using SEARCH, and replace STR1 by STR2. If not found then nothing happens, if found, then cursor is placed at the head of STR2.



4.4.3 Structured Design

Structured Design consists of two simple steps. First translate the design problem into a data flow graph. Second, use either Transform Analysis or Transaction Analysis to create the architecture.

The data flow graph is as shown in figure 19.

This DFG is a selection among different alternatives. Therefore Transaction Analysis is used to construct the architecture.

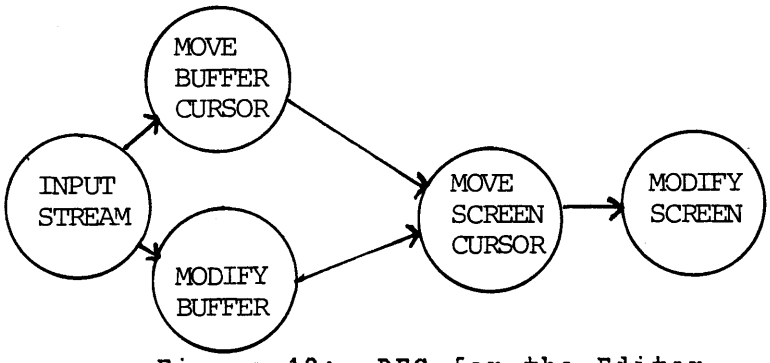


Figure 19: DFG for the Editor

The transaction center is the bubble "Input Stream", because at that point the input data stream branches to three different paths. The three input bubbles can be replaced by the dispatch architecture in figure 20.

Now the DFG reduces to a sequence of bubbles (see figure 21), and we can use Transform Analysis to derive the architecture for the remainder of the system. The entire architecture thus derived is shown in figure 22.

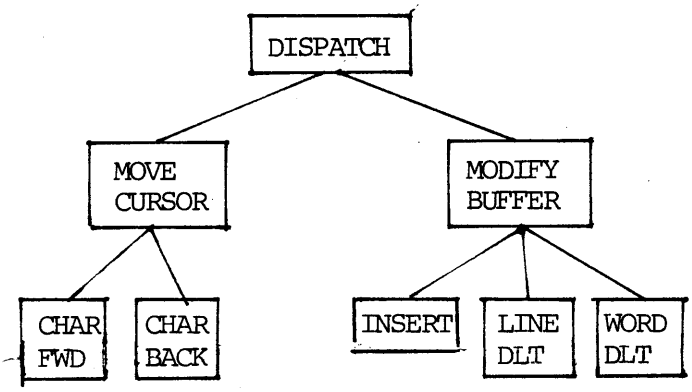


Figure 20: The Transaction Architecture

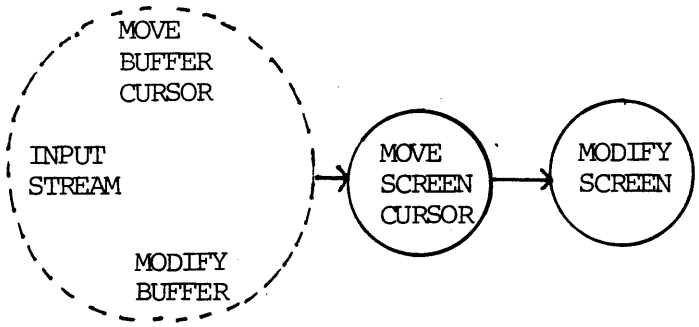


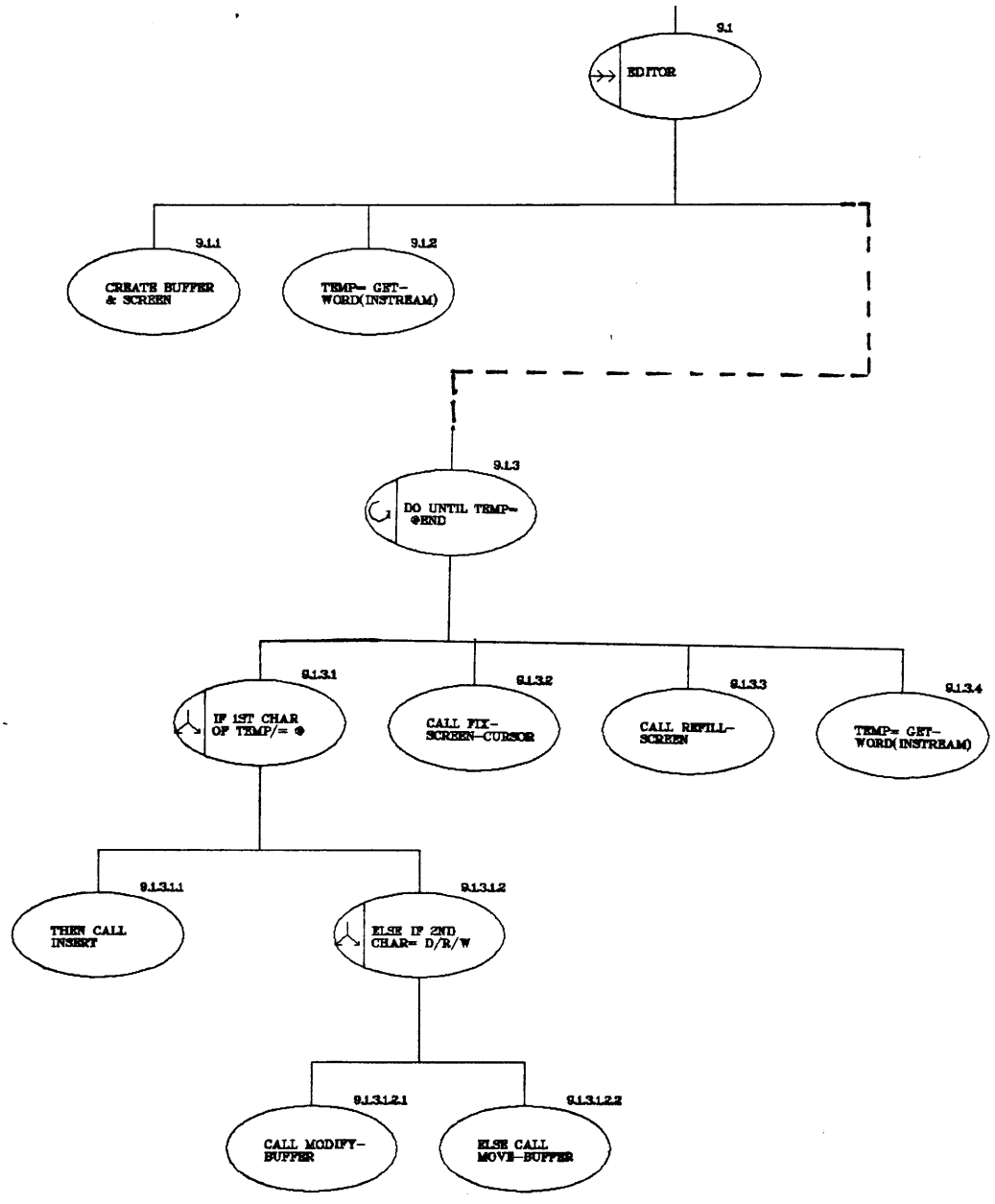
Figure 21: The Reduced DFG

The detailed design of each module is listed in the following subsection. Many of the modules are similar to corresponding modules in Jackson's editor design, therefore only a selected group have been illustrated with DARTS Trees. However, all of the modules are described functionally.

Module Name: EDITOR

Parameter: none

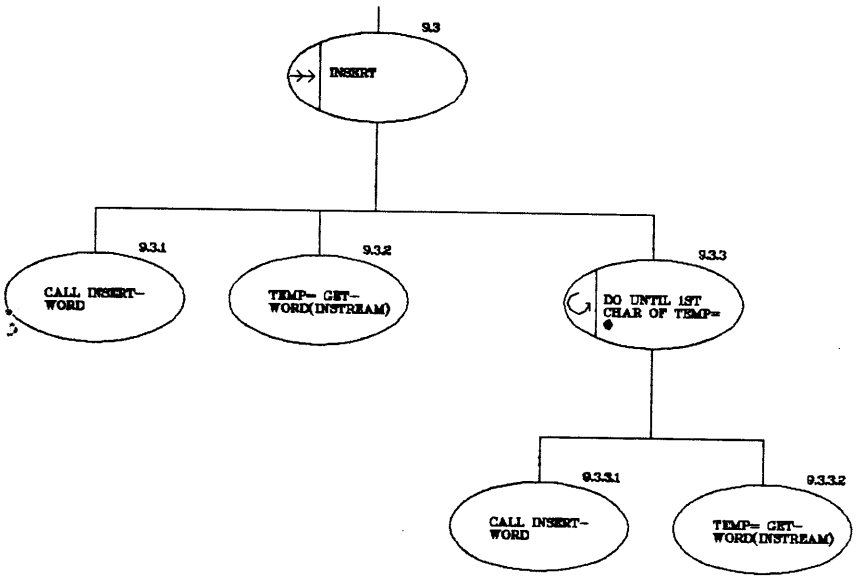
Function: Input-output driver loop.



Module Name: INSERT

Parameter: PACKAGE, WORD, INSTREAM

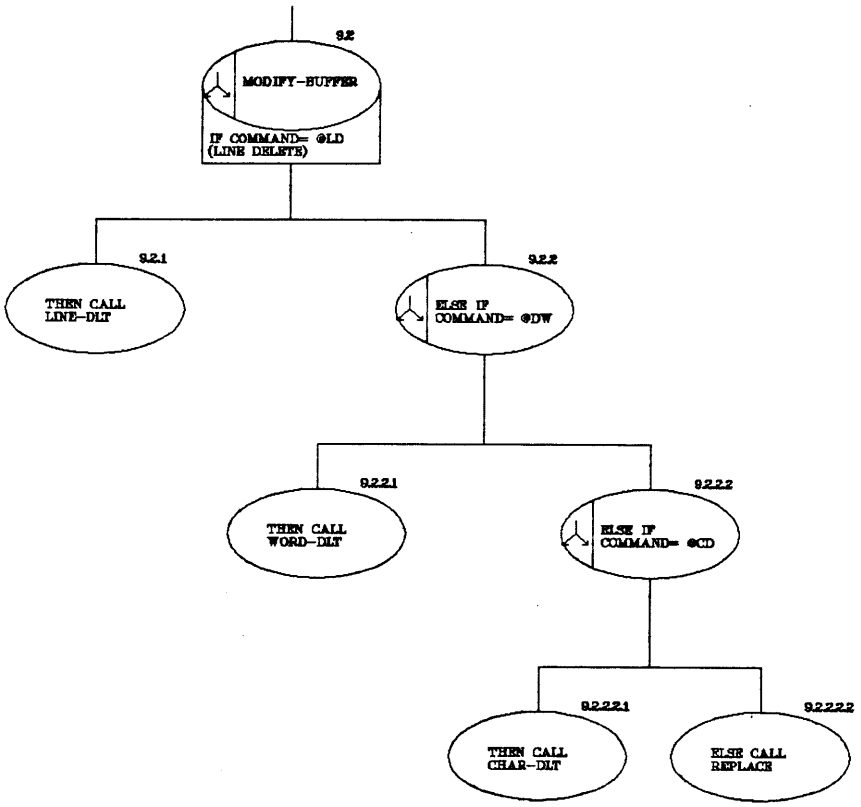
Function: Read text from instream and insert them into the buffer, until a command is encountered.



Module Module Name: MODIFY-BUFFER

Parameter: COMMAND, PACKAGE

Function: Dispatch Command to the appropriate subroutine.



Name: INSERT-WORD

Parameter: WORD, PACKAGE

Function: Add WORD into existing buffer. WORD is placed in front of the cursor.

Module Name: READ

Parameter: PACKAGE, FILENAME

Function: Copy the contents of the file into the buffer.

Module Name: WRITE

Parameter: PACKAGE, FILENAME

Function: Copy the entire buffer into a file under FILENAME.

Module Name: CHAR-DLT

Parameter: PACKAGE

Function: Remove the character pointed to by the cursor.

Module Name: WORD-DLT

Parameter: PACKAGE

Function: Remove the string of non-space characters beginning at the current position until the next space character.

Module Name: LINE-DLT

Parameter: PACKAGE

Function: Deletes text from the cursor to the end of the line. Cursor is left where it was. Adjoin next line to the remainder of current line.

Module Name: REPLACE

Parameter: PACKAGE, STR1, STR2

Function: Find STR1 using SEARCH, and replace STR1 by STR2. If not found then nothing happens, if found, then cursor is placed at the head of STR2.

Module Name: MOVE-PACKAGE-CURSOR

Parameter: COMMAND, PACKAGE

Function: Dispatch COMMAND to the appropriate subroutine.

Module Name: CHAR-FWD

Parameter: PACKAGE

Function: Move cursor position forward one character.

Module Name: CHAR-BACK

Parameter: PACKAGE

Function: Move the cursor position back one character.

Module Name: WORD-FWD

Parameter: PACKAGE

Function: Move cursor position to the head of the next word.

Module Name: WORD-BACK

Parameter: PACKAGE

Function: Move the cursor position to the tail of the previous word.

Module Name: LINE-FWD

Parameter: PACKAGE

Function: Move cursor to the next line. The offset does not change unless the length of next line is too short. In the latter case, the cursor is placed at the tail of the next line.

Module Name: LINE-BACK

Parameter: PACKAGE

Function: Move cursor to the previous line. Offset unchanged unless text on previous line is too short.

Module Name: TOP

Parameter: PACKAGE

Function: Place cursor at the head of the buffer.

Module Name: BOTTOM

Parameter: PACKAGE

Function: Place cursor at the end of the buffer.

Module Name: SEARCH

Parameter: PACKAGE, STRING

Function: Place cursor at the end of the first occurrence of string after the cursor. If not found then leave cursor alone.

Module Name: FIX-SCREEN-CURSOR

Parameter: PACKAGE

Function: Fix the x and y coordinates of the screen's cursor. Use the information in the status codes.

Module Name: REFILL-SCREEN

Parameter: PACKAGE

Function: Fill the screen with 30 new lines of text. The line pointed to by buffer's cursor is placed in the middle (i.e., the 15th line).

4.4.4 Systematic Design Methodology

SDM can be simplified to 4 basic steps. First specify the requirements using templates. Then assess the interdependencies between requirements. Next, apply the clustering algorithm to the interdependencies. Finally, create an architecture from the clusters.

The requirements have already been specified in section 4.1. However, in order to use the clustering program, the requirements are replaced by integers from 1 to 29. The interdependency assessments are given in appendix A.

The resulting clusters are as follows (also see Appendix B):

Cluster 1: 1,7,16,17,18,19,20,21,22,23,24,25,26,27,28.

Cluster 2: 8,9,14,15.

Cluster 3: 10,11,12,13.

Cluster 4: 2,3,4,5,6.

The sub-clusters of cluster 1 are:

Cluster 1: 16,19 (Char-fwd, Word-fwd).

Cluster 2: 17,20 (Char-back, Word-back).

Cluster 3: 18,21 (Char-dlt, Word-dlt).

Cluster 4: 22,23,24,25,26,27,28 (Line-fwd, Line-back, Line-dlt, Top, Bottom, Search, Replace).

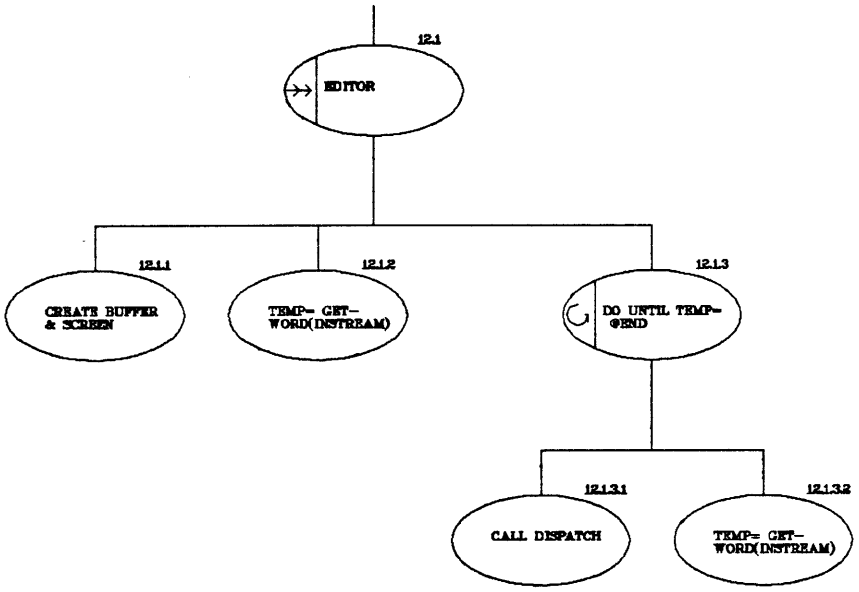
After the clusters are determined, the designer must organize the clusters into a system. Here SDM does not provide any guidelines for the designer. A choice is arbitrarily made to follow the pipelined architecture of HDM (i.e., data goes in one end and comes out the other, unlike Jackson and SD which have a tree traversal type behavior). The complete architecture is shown in figure 23.

Detailed design is listed in the following pages. However, only a selected few have been used to illustrate the general flavor of the design.

Module Name: EDITOR

Parameter: none

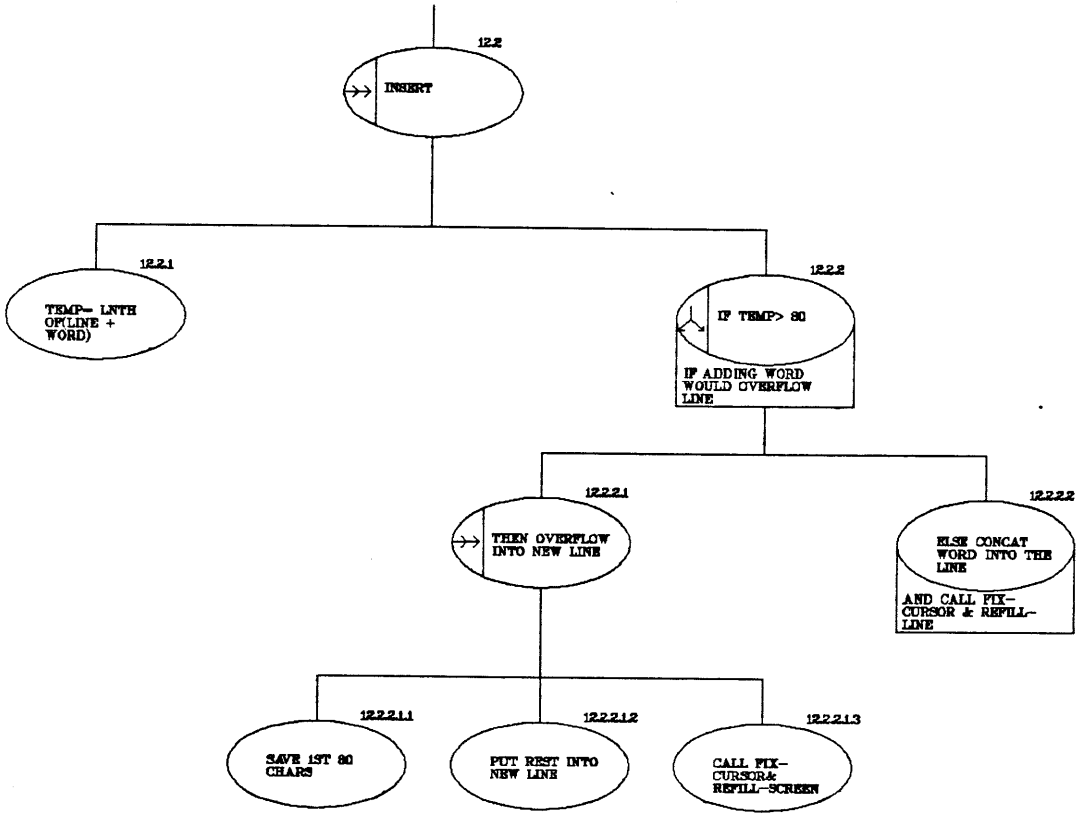
Function: Driver loop for dispatch. Read a word from in-stream and send it to dispatch.



Module Name: INSERT

Parameter: WORD, PACKAGE

Function: Add WORD into existing buffer. WORD is placed in front of the cursor.



Module Name: DISPATCH

Parameter: PACKAGE, WORD

Function: Send WORD to the appropriate command processor.

Module Name: READ

Parameter: PACKAGE, FILENAME

Function: Copy the contents of the file into the buffer.

Module Name: WRITE

Parameter: PACKAGE, FILENAME

Function: Copy the entire buffer into a file under FILENAME.

Module Name: CHAR-FWD

Parameter: PACKAGE

Function: Move cursor position forward one character.

Module Name: WORD-FWD

Parameter: PACKAGE

Function: Move cursor position to the head of the next word.

Module Name: CHAR-BACK

Parameter: PACKAGE

Function: Move the cursor position back one character.

Module Name: WORD-BACK

Parameter: PACKAGE

Function: Move cursor to the tail of the previous word.

Module Name: CHAR-DLT

Parameter: PACKAGE

Function: Remove the character pointed to by the cursor.

Module Name: WORD-DLT

Parameter: PACKAGE

Function: Remove the string of non-space characters beginning at the current position until the next space character.

Module Name: LINE-FWD

Parameter: PACKAGE

Function: Move cursor to the next line. The offset does not change unless the length of next line is too short. In the latter case, the cursor is placed at the tail of the next line.

Module Name: LINE-BACK

Parameter: PACKAGE

Function: Move cursor to the previous line. Offset unchanged unless text on previous line is too short.

Module Name: LINE-DLT

Parameter: PACKAGE

Function: Deletes text from the cursor to the end of the line. Cursor is left where it was. Adjoin next line to the remainder of current line.

Module Name: SEARCH

Parameter: PACKAGE, STRING

Function: Place cursor at the end of the first occurrence of string oafter the cursor. If not found then leave cursor alone.

Module Name: REPLACE

Parameter: PACKAGE, STR1, STR2

Function: Find STR1 using SEARCH, and replace STR1 by STR2. If not found then nothing happens, if found, then cursor is placed at the head of STR2.

Module Name: TOP

Parameter: PACKAGE

Function: Place cursor at the head of the buffer.

Module Name: BOTTOM

Parameter: PACKAGE

Function: Place cursor at the end of the buffer.

Module Name: REFILL-SCREEN

Parameter: PACKAGE

Function: Fill the screen array with 30 new lines. The line pointed to by the buffer's cursor is placed at the 15th line.

Module Name: REFILL-LINE

Parameter: PACKAGE

Function: Get the line pointed to by buffer's cursor, and copy it into the line pointed to by the screen's cursor.

Module Name: MOVE-CURSOR

Parameter: X, Y

Function: Replace the x and y coordinate of screen's cursor by X and Y.

4.5 DESIGN EVALUATION

The best way to judge the soundness of a methodology is to examine its product. Thus it is the object of this section to reflect on the design methodologies by evaluating the designs. The approach taken here is directly aimed at the Systematic Design Methodology. SDM employs a graph clustering algorithm to obtain modules which are strongly dependent internally, and weakly dependent between modules. Therefore the strategy is to measure the designs with respect to the above criteria.

More specifically, the evaluation measures the strength of each module's internal dependency, and the looseness of dependency between modules. These measures are called module strength and module coupling respectively [Myer75].

4.5.1 Module Strength

The basic intent of module strength is to provide a measure of the cohesiveness of the module. Furthermore, if the module is considered to be a functional transformation over some data, then the task can be reduced to an examination of the functionality and data structures of a module. For example, a module which performs a single well defined function over a single data structure clearly possesses more strength than one that performs several functions over a single data structure.

Thus the following ranking for classifying modules is attained (see [MYER75] chapter 3). They are listed in order of increasing strength:

1. Multiple functions related in time. Multiple and unrelated data structures.
2. Multiple functions logically related over multiple data structures (i.e., functions transforming from one data structure into another).
3. Single function over multiple data structures.
4. Multiple functions over a single data structure.
5. Single function over a single data structure.

4.5.2 Module Coupling

Module coupling is determined by how much modules know about one another. In modern programming languages there are generally three ways that modules can communicate: global variables, data items (e.g., variables) passed as parameters, and data structures (e.g., arrays, trees) passed as parameters. In the editors designed for this thesis there are no global variables. The only forms of communication were direct passing of arguments and passing of control information in the buffer structures.

Again a ranking is achieved by considering typical module communication techniques (see [MYER75] chapter 4): (In order of increasing amount of coupling).

1. Only data items are used, and they are all passed as arguments.
2. Data structures (includes data items) are used, and are all passed as arguments.
3. Control information is used (such as flags, function code).
4. One module references an internally defined variable of another module (e.g., free variables in dynamically scoped Lisp).

The designs are evaluated in terms of module coupling, and two kinds of analysis are made: average coupling and couplings per module. Average coupling is the average ranking for the couplings in the design (i.e., sum of coupling ranks / number of couplings). Couplings per module provides a measure of the complexity of the design (i.e., number of couplings / number of modules).

The results of the evaluation are listed in the following tables.

HIERARCHICAL DEVELOPMENT METHODOLOGY

Module Name	Module Strength	Module Coupling
Editor	5	2- Dispatch
Dispatch	2	2x16- Each command processor
Insert	5	2- Buff-get-text-bef 2- Buff-get-text-aft 2- Buff-put-text 2- Buff-add-emp-line 2- Buff-line-fwd
Read	3	2- Buff-add-emp-line 2- Buff-line-fwd 2- Buff-put-text
Write	3	2- Buff-copy-buff 2- Buff-top 2- Buff-tail? 2- Buff-get-text 2- Buff-line-fwd
Char-fwd	5	2- Buff-char-fwd
Char-back	5	2- Buff-char-back
Char-delete	5	2- Buff-char-delete
Word-fwd	5	2- Buff-peek-fwd 2- Buff-char-fwd
Word-back	5	2- Buff-peek-back 2- Buff-char-back
Word-delete	5	2- Buff-peek-fwd 2- Buff-char-delete
Line-fwd	5	2- Buff-get-text-bef 2- Buff-line-fwd 2- Buff-get-text 2- Buff-char-back
Line-back	5	2- Buff-get-text-bef 2- Buff-line-back 2- Buff-get-text 2- Buff-char-back
Line-delete	5	2- Buff-line-kill 2- Buff-line-empty? 2- Buff-get-text-bef

		2- Buff-put-text
Top	5	2- Buff-first-line 2- Buff-get-text-bef 2- Buff-char-back
Bottom	5	2- Buff-last-line 2- Buff-get-text-aft 2- Buff-char-fwd
Search	5	2- Buff-get-text-aft 2- Buff-line-fwd 2- Buff-get-text 2- Buff-tail?
Replace	4	2- Search 2- Buff-char-delete 2- Insert
Buff-create-buff	5	-
Buff-copy-buff	5	-
Buff-Create-line	5	-
Buff-get-text	5	-
Buff-get-text-bef	5	-
Buff-get-text-aft	5	-
Buff-put-text	5	-
Buff-peek-fwd	5	-
Buff-peek-back	5	-
Buff-line-fwd	5	2- Screen-move-cursor
Buff-line-back	5	2- Screen-move-cursor
Buff-line-delete	5	2- Screen-fill-line

Buff-line-empty?	5	-
Buff-buff-empty?	5	-
Buff-head?	5	-
Buff-tail?	5	-
Buff-char-fwd	5	2- Screen-move-cursor
Buff-char-back	5	2- Screen-move-cursor
Buff-char-delete	5	2- Screen-fill-line
Buff-first-line	5	2- Screen-fill-screen
Buff-last-line	5	2- Screen-fill-screen
Buff-add-emp-line	5	2- Screen-fill-screen
Screen-Create-scr	5	-
Screen-move-cursor	5	-
Screen-fill-line	3	2- Buff-get-text
Screen-fill-screen	3	2- Buff-get-text

Number of Modules = 44
Average Strength = 4.71
Average Coupling = 2
Coupling per Mod. = 1.7

Figure 24: Evaluation of HDM

JACKSON METHODOLOGY

Module Name	Module Strength	Module Coupling
Edit	1	2- Input-Handler 2- Output-Handler
Input-Handler	5	2- Dispatch 2- Insert
Insert	2	2- Insert-Word
Insert-word	4	3 Output-Handler
Dispatch	2	2x15- Each command processor
Output-Handler	4	3x15- Each command processor
Refill-line	3	-
Refill-screen	3	-
Fix-screen-cursor	3	-
Read	3	2- Insert-word
Write	3	-
Char-forward	5	-
Char-back	5	-
Char-delete	4	-
Word-forward	5	-
Word-back	5	-
Word-delete	4	-
Line-forward	5	-
Line-back	5	-
Line-delete	4	-

Top	5	-
Bottom	5	-
Search	5	-
Replace	4	2- Search

Number of Modules = 24
Average Strength = 3.92
Average Coupling = 2.39 (Sum of couplings/ number of couplings)
Coupling per Mod. = 1.71 (Number of couplings/ number of modules)

Figure 25: Evaluation of Jackson's Methodology

Module Name	Module Strength	Module Coupling
Editor	1	2- Insert 2- Mod-buff 2- Move-buff-cursor 2- Fix-screen-cursor 2- Refill-screen
Insert	5	2- Insert-word
Insert-word	4	3- Fix-screen-cursor 3- Refill-screen
Modify-buff	2	2x6- Each buffer modifying command
Char-delete	4	-
Word-delete	4	-
Line-delete	4	-
Read	3	2- Insert-word
Write	3	-
Replace	4	2- search
Move-buff cursor	2	2x9- Each Cursor moving command
Char-fwd	5	-
Char-back	5	-
Word-fwd	5	-
Word-back	5	-
Line-fwd	5	-
Line-back	5	-
Top	5	-
Bottom	5	-
Search	5	-
Fix-screen-cursor	3	3x15- Each command processor

Refill-screen	3	3x15-	Each command processor

Number of Modules = 22
Average Strength = 3.95
Average Coupling = 2.58
Coupling per Mod. = 2.5

Figure 26: Evaluation of Structured Design

SYSTEMATIC DESIGN METHODOLOGY

Module Name	Module Strength	Module Coupling
Editor	5	2- Dispatch
Dispatch	2	2x16- Each command processor
Insert	2	2- Refill-screen
Read	3	2- Refill-screen
Write	3	2- Refill-screen
Char-fwd	5	2- Move-cursor
Word-fwd	5	2- Move-cursor
Char-back	5	2- Move-cursor
Word-back	5	2- Move-cursor
Char-delete	4	2- Refill-line
Word-delete	4	2- Refill-line
Line-fwd	5	2- Move-cursor
Line-back	5	2- move-cursor
Line-delete	4	2- Refill-screen
Search	5	-
Replace	4	2- Search
Top	5	-
Bottom	5	-
Refill-screen	3	-
Refill-line	3	-
Move-cursor	5	-

```
Number of Modules = 21
Average Strength  = 4.1
Average Coupline  = 2
Coupling per Mod. = 1.43
```

Figure 27: Evaluation of SDM

	HDM	Jackson	SD	SDM
No. of Modules	44	24	22	21
Avg. Strength	4.71	3.92	3.95	4.1
Avg. Coupling	2	2.39	2.58	2
Coupl / Mod	1.7	1.71	2.5	1.43

Figure 28: Summary of the Evaluations

4.5.3 Summary

After analyzing the designs, two underlying architectures are discovered. The first one is the pipelined architecture of HDM and SDM. In this organization, the data goes into the system at the top and the output comes out at the bottom. There are no coordinator modules to coordinate the activities in the system. Each module has the responsibility to pass control to the next module.

The second architecture is a tree organization (e.g., Jackson and SD), where the data goes down one branch of the system, gets returned back up, and then goes down another branch. In this type of system, activities are controlled by coordinator modules. For example, in Jackson's design, the input goes to the INPUT-HANDLER, then gets returned to the EDITOR module, and finally the EDITOR module passes the data to the OUTPUT-HANDLER.

The pipelined structure proved to be stronger in module strength (4.71 for HDM and 4.1 for SDM). This can be explained by the fact that the system is organized in layers. Each layer deals exclusively with one data structure. For example, in the HDM design, the EDITOR machine only knows about the buffer abstractly. It does not know about the line linked implementation. The BUFFER machine knows about the line linked structure and it contains modules that manipulate the structure. The EDITOR machine would then perform its tasks by calling on the BUFFER machine modules.

On the other hand, modules in the tree architecture must manipulate both the BUFFER and the line linked structure. This caused many of them to have a module strength of 3.

The tree organization is also weaker in module coupling (2.39 for Jackson and 2.58 for SD). The major cause of the higher average coupling measure is the control information passed through the INC array. Recall that the INC array was used to store incremental movements of the cursor. This information was passed from the command processors to the output modules.

The coupling problem is inherent in the tree architecture. The only way modules on different branches of the tree can communicate is by passing control information through the coordinator. Otherwise a global variable must be used, but that often leads to high debugging cost.

Overall, the HDM design is the best for this particular case. The strength and coupling rankings for HDM are the best among the four. It is also easiest to add extra editor commands to HDM's design. The BUFFER machine modules are really a meta-language and new commands can be easily composed from it.

Finally, it is worth noting that SDM's clusters could have been organized into either type of architecture. It was purely arbitrary that SDM had the pipelined architecture.

Chapter V

CONCLUSION

This chapter presents the insights gathered through the project. The first section discusses the basic problem in software design. The second section discusses the various approaches to develop a methodology. The third section points out the weaknesses of SDM in comparison to other methodologies. Finally, this chapter concludes with suggestions for further research.

5.1 THE COMPLEXITY PROBLEM

By far the most dominating difficulty in software design is complexity. Often the designer knows all the requirements but cannot think about all of them simultaneously, or it might be that the designer has ideas for satisfying individual requirements, but can not put all the different solutions into one system. The problem is not the lack of ideas, but the lack of unity and cohesion.

Where exactly does complexity come from? Through the study in design theory, it is evident that complexity arises from the interaction between requirements [Alexander], and from the numerous options available as solution [Manheim]. The key word here is interaction. Complexity arises because

earlier decisions often add to the requirements of later decisions since decisions interact with each other.

One might think of complexity as a bunch of molecules tied to each other in some complicated way. Each molecule is trying to fly off in its own direction, and in doing so, it alters the flight of all molecules tied to it. The designer's job is to try and get these molecules into a state of equilibrium, or to get the whole network of molecules to move in one direction.

Thus, the task of architectural design is really a task in complexity management. Having the architecture, the designer can then devise solutions for each requirement. He now knows that his solution must somehow fit into the architecture. Furthermore, he knows that if individual solutions conform to the conventions of the architecture, then they will function together. In other words, the problem is shifted from "dealing with all the other requirements", to "dealing with the architecture". Therefore, architectural design is in many ways analogous to providing an organizational structure. Given such a structure, the designer can then solve the problem of "How can the solution to a requirement integrate into the structure?", rather than the much more difficult problem of "How can the solution of this requirement co-exist with the solutions of other requirements?".

5.2 STRATEGIES FOR DEVELOPING A METHODOLOGY

There are two basic approaches in designing software methodologies. The first is to survey existing systems, and select those systems that are successful. These systems are then scrutinized to identify similar features. Having identified the constant factors among successful systems, techniques can be developed to derive system architectures from those constant factors. For example, Jackson's methodology derives an architecture from the data structure, and Structured Design derives an architecture from data flow graphs.

The second approach is to adopt a design theory, upon which conjectures are made about what constitues good design. Examples of this approach are Hierarchical Development Methodology's decision model and abstract machines; Systematic Design Methodology's graph model and module decomposition criteria. In HDM, the design process is modelled by a sequence of interdependent decisions. A hierarchical abstract machine structure is then provided for grouping decisions. SDM, on the other hand, models design as a process of grouping requirements into clusters. These clusters are formed in a way that maximizes intra-cluster dependency and minimizes inter-cluster dependency.

It is important to point out that of all the methodologies surveyed, only SDM provides a sense of optimality. Most methodologies suggest one architecture, but do not claim it to be the best architecture. Whereas SDM provides the best modularization.

Also, methodologies are aiming toward a black box technology. A methodology is envisioned as a box which takes in some kind of data and returns an architecture. This trend leads to the development of design tools which will eventually generate programs automatically.

One important strategy that has emerged is that architecture should be constructed to match the problem structure. This concept has an obvious justification in systems which have real life interpretations. But even in abstract applications it also makes sense to construct the system to match the mental picture. Having this close correspondence between system architecture and mental picture, the system can then be modified and debugged with ease. That perhaps is the reason several methodologies (such as Structured Design) go through a problem analysis phase before constructing the architecture.

However, the task of creating a system architecture that matches problem structure is not trivial. There are constraints imposed by the programming language and the computer hardware. For example, the typical programming language has three kinds of control flow constructs, sequence, iterate, and branch. Therefore the system architecture must use these control constructs and nothing else.

5.3 MODIFICATIONS FOR SDM

One weakness of SDM arises from the way interdependencies are assessed. SDM suggests that the designer should have an idea of how the requirements can be implemented, and assign weights accordingly. This method is too subjective. Different designers will most assuredly come up with different wights. Therefore the sense of reproducibility and absoluteness is lost.

A way to deal with this problem is to identify data structures used in the system. Weights can now be assigned according to whether the requirements deal with the same data structure.

Another thing to do is to separate requirements into levels of generality. Requirements such as modifiability and fault tolerance are highly universal in their effects. Whereas a requirement like "implement a delete command" is a much more detailed requirement.

To perform this separation, nodes can be linked together according to their level of generality. If two nodes are very close in generality, then they are assigned a high weight, otherwise they are weighed lightly. Then the clustering algorithm can be applied to the graph to obtain clusters which represent different levels of generality.

After grouping requirements this way, the usual SDM procedures can be performed on each group. This results in clusters for each level of generality, which can then be

treated as a single node. The result is a three dimensional clustering situation as shown in figure 29

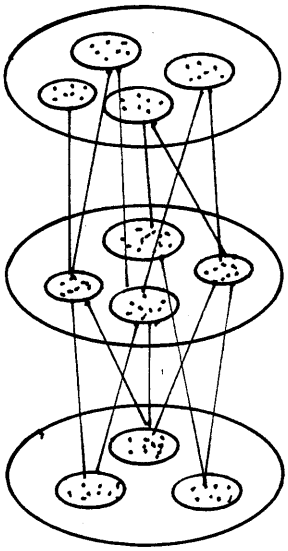


Figure 29: 3-D Clustering

Now the problem is how to describe each cluster-node. The idea of abstraction can be applied here. Each cluster would probably have some general characteristic. For example the editor commands character-forward and line-back have the common characteristic of moving the cursor. Therefore these requirements can be described by a new node called cursor movement.

Another weakness of SDM is that it does not produce an architecture. SDM tells the designer which requirements are

highly dependent on each other, and therefore must be considered together. Furthermore, SDM claims that the clusters should correspond to modules in the system. However, this still leaves the designer with the difficult task of organizing the modules into a structure.

Other methodologies have largely concentrated on exactly the above problem. Jackson, Structured Design and HDM, all these methodologies provide a system architecture. However, they are weak in the area of problem structure definition. Therefore, the natural thing to do is to combine SDM with other methodologies.

One such combination can be made between SDM and Structured Design. There are two ways to do the combination. First, SDM can be used to help derive Structured Design's data flow graph. Weights can be assigned according to whether or not two requirements are doing the same kind of data transformation. The resulting clusters would provide the individual bubbles in the data flow graph. Having the data flow graph, the designer can proceed according to the procedure of Structured Design.

The second way to combine SDM and Structured Design is to use the data flow graph for SDM interdependency assessment. Weights can be assessed by considering how strongly two nodes relate to the same bubble in the data flow graph. This method of weight assessment is implementation independent. However, there is a drawback in that the set of clus-

ters would probably have a one to one correspondence with the bubbles of the DFG.

Another combination is HDM and SDM. A hierarchy of abstract machines can be identified, then SDM can be used to link each requirement to a machine. Otherwise, SDM can be used to determine the machines themselves. Weights would be assigned according to whether two requirements should be in the same machine. Then the resulting clusters would represent the machines.

5.4 SUGGESTIONS FOR FURTHER RESEARCH

The major weakness of SDM is its one-sidedness. SDM attacks the problem analysis phase but does not provide much help in constructing the architecture. As was mentioned before, in the experiment, the SDM clusters could have been organized into either the Jackson type architecture or the HDM type architecture. Thus the area of architecture construction definitely needs more attention.

A second weakness of SDM lies in the interdependency assessment phase. A method to assign weights independent of designers' personal biases is needed. Data structures provide a promising path in this direction.

Finally, it is worthwhile to consider coupling SDM with another methodology. SDM is strong in the problem analysis phase while other methedologies are strong in the architecture construction phase. If used together, each could com-

plement the other. The next step in this direction is to define formal interfaces.

Appendix A

SDM INTERDEPENDENCY ASSESSMENTS

The weights shown on the next page are arranged in a format recognizable to the clustering program (see [LATT81]). This is the short form option (the other option is called regular form).

All of the nodes linked to each node are listed on the same row but in the second column. Nodes are represented by three digit numbers. Underneath the list of "neighbor" nodes are the weights for the corresponding links. For example, suppose node 001 is linked to nodes 002 and 003 by the wights of 2 and 8 respectively, then the following is the short form representation:

```
001 002003
      2  8
```

```
028
001 007
      8
002 004005006007
      8  5  5  8
003 004005007009
      8  2  8  8
004 002003006007008
      8  8  8  5  8
005 002003
      5  2
006 002004007009014
      5  8  5  2  2
007 001002003004006008014016017018019020021022023024025026027028
      8  8  8  5  5  2  5  8  8  8  8  8  8  8  8  8  8  8  8  8  8
008 004007009014
      8  2  8  8
009 003006008014015
      8  2  8  8  8
010 001011012013
      2  8  8  8
011 001010012013
      2  8  8  8
012 001010011013
      2  8  8  8
013 001010011012
      2  8  8  8
014 006007008009
      2  5  8  8
015 009
      8
016 007019027028
      8  8  8  8
017 007020
      8  8
018 007021028
      8  8  8
019 007016027028
      8  8  8  8
020 007017
      8  8
021 007018
      8  8
022 007
      8
023 007027028
      8  8  8
024 007
      8
025 007
      8
026 007
      8
027 007016019023028
      8  8  8  8  8
028 007016018019023027
      8  8  8  8  8  8

R;
```

Appendix B

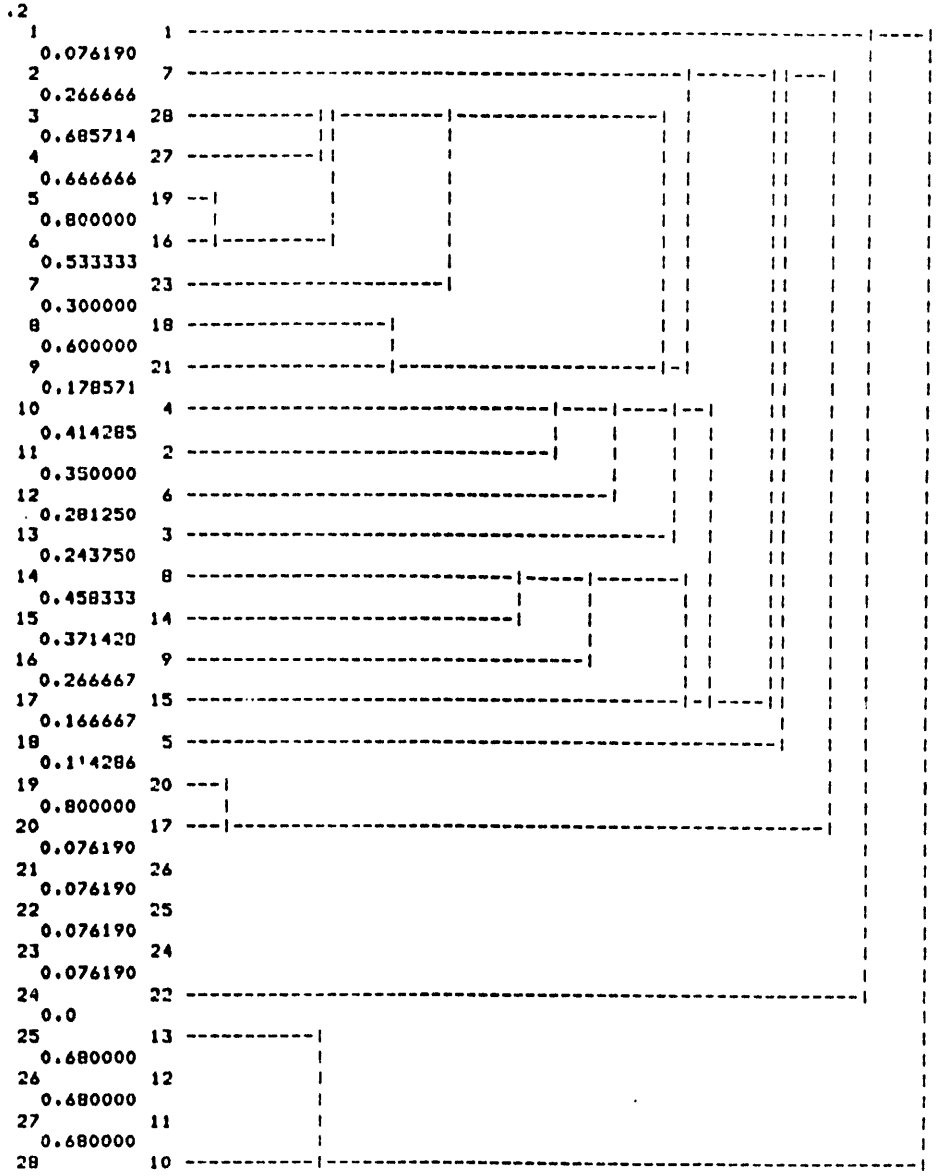
SDM CLUSTERS

The SDM clustering algorithm has been implemented in Fortran by Jim Lattin (see [WONG80] and [LATT81]). It currently resides under the CMS account LATTINA in MIT. Access to the program can be obtained from Jim Lattin.

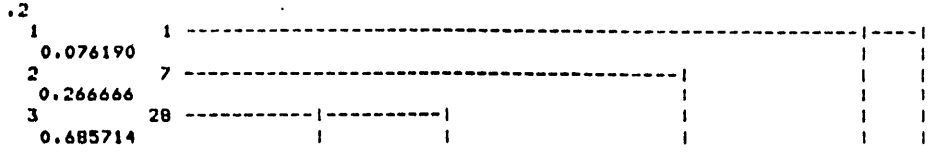
To use the program one must first define the weights as shown in appendix A, then run the exec program FIDEF4. The session during which the editor requirements were decomposed is shown on the next page.

```
 fidef4 mtu4 data
T=0.16/0.36 14:30:23
EXECUTION BEGINS...
NODLIM =   900
ARCLIM =  4500
INPUT FORMAT:
(1)  REGULAR
(2)  SHORT
.2
TOTAL NODES =    28
TOTAL ARCS =    53
AVERAGE NUMBER OF ARCS INCIDENT TO EACH NODE =    1.89
   1
   2
   3
   4
   5
   6
   7
   8
   9
  10
  11
  12
  13
  14
  15
  16
  17
  18
  19
  20
  21
  22
  23
  24
  25
  26
  27
  28
DENSITIES CALCULATED IN          3 HUNDREDTHS CPU SECS
TREE FORMED IN            1 HUNDREDTHS CPU SECONDS
PRINT TREE TO:
(1)  FILE
(2)  TTY
.2
  1          1 ----------------------------------------------------|----|
    0.074480
```

```
TREE FORMED IN            1 HUNDREDTHS CPU SECONDS
PRINT TREE TO:
(1)  FILE
(2)  TTY
.2
```

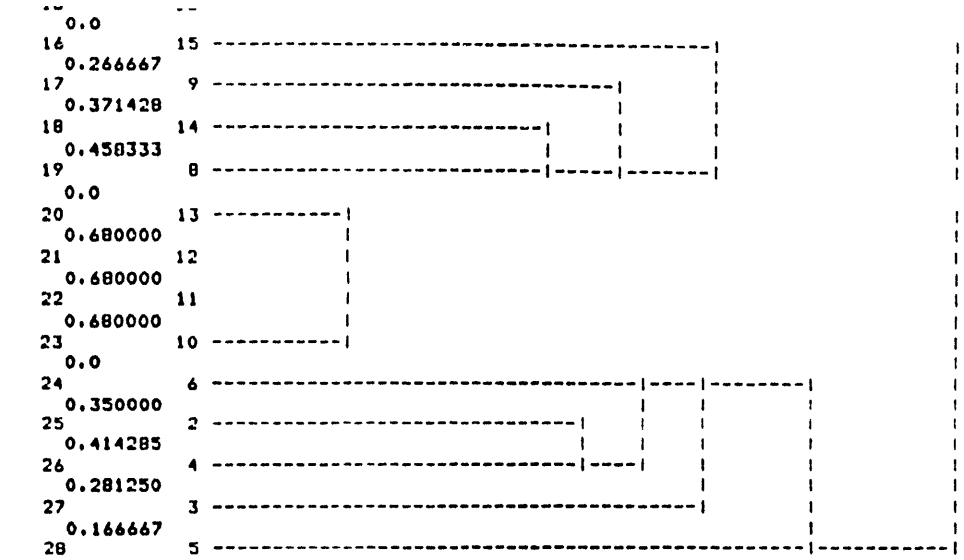



```
TREE FORMED IN            0 HUNDREDTHS CPU SECONDS
PRINT TREE TO:
(1)  FILE
(2)  TTY
.2
```



```
27          11
  0.680000
28          10 ------------|-----------------------------------------------|
```

```
TREE FORMED IN              0 HUNDREDTHS CPU SECONDS
PRINT TREE TO:
(1)  FILE
(2)  TTY
.2
 1           1 ---------------------------------------------------------|----|
  0.076190                                                              |    |
 2           7 ----------------------------------------|                |    |
  0.266666                                             |                |    |
 3          28 -----------|----------|                 |                |    |
  0.685714                |          |                 |                |    |
 4          27 -----------|          |                 |                |    |
  0.533333                           |                 |                |    |
 5          23 ----------------------|-----------------|                |    |
  0.076190                                                              |    |
 6          26                                                         |    |
  0.076190                                                              |    |
 7          25                                                         |    |
  0.076190                                                              |    |
 8          24                                                         |    |
  0.076190                                                              |    |
 9          22 ----------------------------------------------------------|    |
  0.0                                                                        |
10          21 -----------------|                                         |
  0.600000                      |                                         |
11          18 -----------------|                                         |
  0.0                                                                       |
12          20 ---|                                                       |
  0.800000        |                                                       |
13          17 ---|                                                       |
  0.0                                                                       |
14          19 --|                                                        |
  0.800000       |                                                        |
15          16 --|                                                        |
  0.0                                                                       |
16          15 ----------------------------------------|                  |
  0.266667                                             |                  |
17           9 ----------------------------------|     |                  |
  0.371428                                       |     |                  |
18          14 ------------------------------|    |     |                  |
  0.458333                                  |    |     |                  |
19           8 ------------------------------|----|-----|                  |
  0.0                                                                       |
20          13 -----------|                                                |
  0.680000                |                                                |
21          12            |                                                |
  0.680000                |                                                |
22          11            |                                                |
  0.680000                |                                                |
23          10 -----------|                                                |
  0.0                                                                       |
24           6 ------------------------------------|----|--------|          |
  0.350000                                         |    |        |          |
25           2 --------------------------------|    |    |        |          |
  0.414285                                     |    |    |        |          |
26           4 --------------------------------|----|    |        |          |
  0.281250                                           |    |        |          |
27           3 ----------------------------------------|    |        |          |
  0.166667                                                |        |          |
28           5 ----------------------------------------------------|----------|
TIME FOR ENTIRE PROCESS:          164 HUNDREDTHS CPU SECS
REACHED BLDPAR
BUILDING PARTITION TOOK           2 HUNDREDTHS CPU SECS
CLUSTER NUMBER   1
       1
       7
      16
```




```
TIME FOR ENTIRE PROCESS!          164 HUNDREDTHS CPU SECS
REACHED BLDPAR
BUILDING PARTITION TOOK          2 HUNDREDTHS CPU SECS
CLUSTER NUMBER   1
       1
       7
      16
      17
      18
      19
      20
      21
      22
      23
      24
      25
      26
      27
      28
CLUSTER NUMBER   2
       8
       9
      14
      15
CLUSTER NUMBER   3
      10
      11
      12
      13
CLUSTER NUMBER   4
       2
       3
       4
       5
       6
EVALUATION MEASURE EQUALS 0.105
MINPER =       4

(1)  KEEP PARTITION
(2)  ENTER DIFFERENT MINPER
.2
ENTER NEW MINPER (IN FMT I3)
.002
REACHED BLDPAR
BUILDING PARTITION TOOK          1 HUNDREDTHS CPU SECS
CLUSTER NUMBER   1
       1
       7
      22
      23
```

```
      16
      17
      18
      19
      20
      21
      22
      23
      24
      25
      26
      27
      28
CLUSTER NUMBER   2
       8
       9
      14
      15
CLUSTER NUMBER   3
      10
      11
      12
      13
CLUSTER NUMBER   4
       2
       3
       4
       5
       6
EVALUATION MEASURE EQUALS 0.105
MINPER =       4

(1)  KEEP PARTITION
(2)  ENTER DIFFERENT MINPER
.2
ENTER NEW MINPER (IN FMT I3)
.002
REACHED BLDPAR
BUILDING PARTITION TOOK          1 HUNDREDTHS CPU SECS
CLUSTER NUMBER   1
       1
       7
      22
      23
      24
      25
      26
      27
      28
CLUSTER NUMBER   2
      18
      21
CLUSTER NUMBER   3
      17
      20
CLUSTER NUMBER   4
      16
      19
CLUSTER NUMBER   5
       8
       9
      14
      15
CLUSTER NUMBER   6
      10
      11
      12
      13
CLUSTER NUMBER   7
       2
       3
       4
       5
       6
EVALUATION MEASURE EQUALS 0.124
MINPER =       2
```

```
       6
EVALUATION MEASURE EQUALS 0.105
MINPER =       4

(1)  KEEP PARTITION
(2)  ENTER DIFFERENT MINPER
.2
ENTER NEW MINPER (IN FMT I3)
.002
REACHED BLDPAR
BUILDING PARTITION TOOK          1 HUNDREDTHS CPU SECS
CLUSTER NUMBER   1
       1
       7
      22
      23
      24
      25
      26
      27
      28
CLUSTER NUMBER   2
      18
      21
CLUSTER NUMBER   3
      17
      20
CLUSTER NUMBER   4
      16
      19
CLUSTER NUMBER   5
       8
       9
      14
      15
CLUSTER NUMBER   6
      10
      11
      12
      13
CLUSTER NUMBER   7
       2
       3
       4
       5
       6
EVALUATION MEASURE EQUALS 0.124
MINPER =       2

(1)  KEEP PARTITION
(2)  ENTER DIFFERENT MINPER
.1
CHANGE CURRENT PARTITION?
(1)  YES
(2)  NO
.2
T=0.78/2.17 14136155
R!
```

BIBLIOGRAPHY

[ATWO78] Atwood, M.E.; Turner, A.A.; Ransey H.R.; Hooper, J.N., "An Exploratory Study of the Cognitive Structures Underlying the Comprehension of Software Design Problems"; Science Publications Inc., Army Research Institute for the Behavioral and Social Sciences, Alexandria, VA.

[ALEX64] Alexander, C.; "Notes On the Synthesis of Form"; Harvard University Press, 1964.

[BASI81] Basili, V.; Reiter, R.W., "A Controlled Experiment Quantitatively Comparing Software Development Approaches"; IEEE Transations on Software Engineering, Vol. SE-7, No. 3, May 1981.

[BATE77] Bate, R.R., "Software Design Procedure"; AIAA/NASA/IEEE/ACM Computer in Aerospace Conference, Collection of Technical Papers, Published by AIAA, New York, NY. 1977.

[BATE78] Bate, R.R.; LIgler G.T., "Software Development Methodology: Issues, Techniques, and Tools"; Proceedings of 11th Internation Conference on Systems Science, West Period Co., North Hollywood, CA. 1978.

[BERG78] Bergland, G.D.; Tranter, W.H.; "Software Design Techniques"; Proceedings of the National Electronic Conference, Vol. 32, Chicago, IL. , Oct, 1978.

[BERG81] Bergland G.D.; "A Guided Tour of Program Design Methodologies"; IEEE Computer, Oct. 1981.

[BOYD78] Boyd, D.L.; Pizzarello, A.; "Introduction to the WELLMADE Design Methodology"; IEEE Transaction on Software Engineering, Vol. SE-4, No. 4, July 1978, pp 276-282.

[BROO77] Brooks, F.P.; "The Mythical Man-month"; IEEE Tutorial on Software Design Techniques, 1977, pp 18-25.

[BRUG79] Bruggere, T.H.; "Software Engineering: Management, Personnel and Methodology"; Proceedings of 4th International Conference on Software Engineering, 1979, pp361-368.

[CAIN75] Caine, S.H.; Gordon, E.K.; "PDL - A Tool for Software Design"; Proceedings of AFIPS National Computing Conference, May 1975, pp271-276.

[CAMP80] Camp, J.W.; "Computer-Aided Design Applied to Software Engineering"; IEEE Proceedings of National Aerospace Electronic Conference, NAECON 1980, Vol. 1, pp33-37.

[CAMP78] Campos, I.M.; Estrin, G.; "SARA Aided Design of Software for Concurrent Systems"; Proceedings of AFIPS National Computing Conference, 1978, pp325-358.

[CHES80] Chester, D.; Yeh, R.T.; " An Integrated Methodology and Tools for Software Development"; University of Texas at Austin, Software and Data Base Engineering Group, TR 411983.

[CHOW78] Chow, T.S.; "Analysis of Software Design Modelled by Multiple Finite State Machines"; Proceedings of COMPSAC, 1978.

[CHU75] Chu, Y.; "Methodology for Software Engineering"; IEEE Transactions on Software Engineering, Vol. SE-1, No. 3, Sept. 1975, pp262-270.

[CORN79] Corner, C.; Halstead M.H.; "A Simple Experiment in Top-down Design"; IEEE Transactions on Software Engineering Vol. SE-5, No. 2, March 1979.

[DAHL74] Dahl, W.J.; Nunamaker, J.F.; "Interactive Information Systems Design Software"; Proceedings of 5th Annual Pittsburgh Conferecce on Modelling and Simulation, 1974, pp24-26.

[DELF80] Delfino, A.B.; Begun, R.A.; "Design of a Software Development Methodology Emphasizing Productivity"; IEEE Applications of MINI and MIcrocomputers, March 1980.

[DEWO78] Dewolf, J.B.; Whitworth, M.; "Methodology Research Measures"; C.C. Draper Lab. Technical Report 1167, August 1978.

[ELEC81] "Today's Software Tools Point to Tomorrow's Tool Systems"; Electronic Design, July 23, 1981.

[ENOS81] Enos, J.C.; Van Tilburg, R.L.; " Software Design"; Tutorial Series 5, IEEE Computer, February 1981.

[FREE77] Freeman, P.; "The Nature of Design"; IEEE Tutorial on Software Design Techniques, 1977, pp29-36.

[GOMA79] Gomaa, H.; "Comparison of Software Engineering Methods for System Design"; Proceedings of National Electron Conference, Chicago, IL., October 1979.

[GRAH73] Graham, B.M.; Clancy G.J.; DeVaney, D.B.; "A Software Design and Evaluation System"; Communications of the ACM, Vol. 16, No. 2, February 1973.

[HAMI76] Hamilton, M.; Zeldin, S.; "Higher Order Software - A Methodology for Defining Software"; IEEE Transactions on Software Engineering, Vol. SE-2, No. 1, March 1976.

[HAMM78] Hammond, L.S.; Murphy P.L.; "System for Analysis and Verification of Software Design"; Proceedings of COMPSAC '78.

[HUFF79] Huff, S.L.; "A Systematic Methodology for Designing the Architecture of Complex Software Systems"; PhD thesis, Sloan School of Management, M.I.T., 1979.

[JACK75] Jackson, M.A.; "Principles of Program Design"; Academic Press, New York, N.Y., 1975.

[JENS81] Jensen, R.W.; "Structured Programming", IEEE Computer, March 1981.

[LATT81] Lattin, J.M.; "Implementation and Evaluation of a Graph Partitioning Technique Based on a High-density Clustering Model"; Technical Report #15, Center for Information Systems Research, Sloan School of Management, M.I.T., 1981.

[LEVI80] Levitt, K.N.; Newmann, P.G.; Robinson L.; "The SRI Hierarchical Development Methodology (HDM) and its Application to the Development of Secure Software"; SRI International, Menlo Park, CA., 1980.

[MANH64] Manheim, M.L.; "Hierarchical Structure: A Model of Design and Planning Processes"; MIT Report No. 7, Civil Engineering Dept.

[MANH67] Manheim, M.L.; "Problem-Solving Processes in Planning and Design"; Professional Paper P67-3; Dept. of Civil Engineering, MIT.

[MAYF77] Mayfield, J.P.; "Software Development Methodology Selection Criteria"; AIAA/NASA/IEEE/ACM Computer in Aerospace Conference, Collection of technical papers, 1977.

[McCl75] McClure C.L.; "Top-Down, Bottom-Up, and Structured Programming"; IEEE Transactions on Software Engineering, Bol. SE-1, No. 4, December 1975.

[McG076] McGowan C.L.; Kelly J.R.; "A Review of Some Design Methodologies"; Softech TR053, Sept 1976.; Invited Presentation for Info Tech State of the Art Conference on Structured Design, October 1976.

[MYER75] Myer, G.J.; "Reliable Software Through Composite Design"; Petrocelli/Charter, New York, 1975.

[PETE77] Peter, L.; Tripp, L.; "Comparing Software Design Methodologies"; Datamation, Vol. 23, No. 11, November 1977.

[RAMA73] Ramamoorthy, C.V.; Meeker, R.E.; Turner, J.; " Design and Construction of an Automated Software Evaluation System"; IEEE Symposium on Software Reliability, May 1973, pp28-37.

[RAMS79] Ramsey, H.R.; Atwood, M.E.; Campbell, G.D.; "An Analysis of Software Design Methodologies"; Science Application Inc., Army Reseasrch Institute for the Behavioral and Social Sciences, Alexandria VA., 1979.

[RICH79] Rich, C.; Shrobe, H.E.; Waters, R.C.; "Computer Aided Evolutionary Design for Software Engineering"; MIT Artificial Intelligence Lab., Memorandum, January 1979.

[RICH77] Richards, P.K.; "Developing Design Aids for an Integrated Software Development System"; AIAA/NASA/IEEE/ACM Computing in Aerospace Conference, Collection of technical papers; published by AIAA, New York, N.Y., 1977.

[RIDD78] Riddle, W.E.; Wilden, J.C.; Sayler, J.H.; "Behavior Modelling During Software Design"; Proceedings of 3rd International Conference on Software Engineering, 1978.

[RIDD79] Riddle, W.E.; Schneider H.J.; "An Event Based Design Methodology Supported by DREAM"; Formal Models and Practical Tolls for Information Systems Design, Oxford, England, April 1979.

[RZEV79] Rzevski, G.; Adey, R.A.; "On the Design of Engineering Software"; Engineering Software, Southampton, England, Sept 1979.

[SIM069] Simon, H.A.; "The Sciences of the Artificial"; MIT Press, 1969.

[SUTT81] Sutton, S.A.; Basili, V.R.; "The Flex Software Design System: Designers Need Languages Too"; IEEE Computer, November 1981, pp95-102.

[WARN74] Warnier, J.D.; "Logical Construction of Programs"; Van Nostrand Reinhold Co., New York, N.Y., 1974.

[WILL79] Willis, R.R.; Jensen, E.P.; "Computer Aided Design of Software Systems"; Proceedings of 4th International Conference on Software Engineering, 1979, pp116-125.

[WONG80] Wong, M.A.; "A Graph Decomposition Technique Based on A High-density Clustering Model on Graphs"; Technical Report #14, Sloan School of Management, M.I.T., 1980.

[YOUR79] Yourdon, E.; "Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design"; Prentice-Hall, 1979.